Flexible Data Processing with Plug-In Enabled Tools

George C. Leussis[1]
Northrop Grumman, Cambridge, MA, 02138

In today's space operations environment, processing requirements evolve rapidly. Changing data transport formats frequently requires significant alterations to ground processing software simply to get data into the system. Further, end users may desire output not envisioned by the systems designers. I will show how these problems can be solved by applying the concept of plug-ins. A plug-in is small software component that acts on the data in independent stages and can communicate with the system as a whole by passing signals, or API-defined data structures called "messages". A plug-in can also communicate with other plug-ins using either signals, messages, or their own shared structures. Plug-ins utilize the system's API to either provide input to, or output from the system. They can access the entire data stream or be restricted to a subset. They can work in concert with the database or completely ignore the system's definitions to perform processing not previously thought necessary.

I. Introduction

Prior to launch, the Chandra Ground Operations Team decided it needed a way to analyze certain aspects of the telemetry data stream. As a result, the Telemetry Controller's Toolkit (Toolkit) was born. Over the years it has grown into a much larger suite of tools as the needs of the project have evolved. This paper will summarize the plug-in features of Toolkit and highlight some of the ways that the use of plug-ins has allowed Toolkit to evolve over time. In particular, we will focus on the use of plug-ins as a means to maximize software flexibility when modifications were needed at the level of input, output or processing.

The original goal of the Toolkit software was to provide a means of determining the data quality, or completeness, of a particular set of data. As the ground system and the ground team's role evolved, new functions were needed. These included the ability to packetize the frames for transmission to the telemetry server, the ability to examine data within the frames, and other utilities that have helped in troubleshooting various issues we have encountered.

[1] System Engineer, Chandra Ground Systems Engineering, 60 Garden St MS33 Cambridge MA, 02138

Toolkit was initially written to process TDM data, as Chandra's output format is TDM with a small appended header to make it CCSDS compliant at the transport layer. DSN then wraps each Chandra frame in an SFDU. Toolkit could still pretend that each SFDU was a TDM frame due to its fixed nature. Later, the SFDU format changed slightly giving us a reason to process the SFDU as it is truly defined. As Toolkit evolved it became especially important to allow for changes in external interfaces, so it could process differing types of data and in many different formats. These many changes in requirements and functionality resulted in frequent modifications to the tool. We were constantly adding sections and recompiling the tool. These frequent updates could cause errors as one change sometimes affected other parts of the code. Further, every change to Toolkit needed to be presented to the Flight Director's Board since it was a change to the main tool, even for minor changes such as adding the option to tweak the output precision of a displayed data point. The use of plug-ins has allowed for a more modular and efficient approach to software modifications.

II. Architecture Overview

The Toolkit application is designed as a frame pipeline. The main routine receives an input frame, then transfers the frame to downstream processing routines. Results of processing routines are handled internally to the routine. Optional frame output, which could be considered a variation on frame processing, can pass the entire frame out in one of several different frame formats. Initially only hex/octal dump or binary file output were supported, but later network interfaces with additional output formats were added, while retaining all of the previously defined output formats.

Internally, Toolkit passes a frame structure, which consists of a set of descriptors and a pointer to the binary data. The frame is mostly described by the information specified in the configuration file. These frames are passed through each processing stage of the pipeline. Typically a frame consists of an SFDU, however several other formats are supported in most of the usual routines.

Globally, internal frame structures are mainly driven by buffer size, configuration file parameters, and the database. As more and more changes were desired and the Toolkit codebase grew, we decided to add the ability to dynamically load these functions into the code on an as-needed basis. This was the origin of Toolkit’s plug-in interface.

By deciding to use plug-ins with Toolkit, we add only necessary components of the code at any given time. In particular, the plug-ins that have been developed fall into one of three groups: input, processing, or output. The plug-ins are defined by an external shared object with a couple of well-defined functions. The shared objects are loaded via the standard c dlfcn.h library. The requirements on the set of functions in the linked library are as follows:

- The function named <function> (the assigned name) shall have the following arguments (void *input, void **arguments, void **output). The function is responsible for proper casting of the arguments.
- If it has an initialization function it must be named <function>_init and will take as arguments (void *args, void **out). The function should allocate memory for and set *out (usually a structure containing any information <function> needs operate properly)
- If it has a cleanup function it must be named <function>_cleanup and will take as arguments (void *args). Usually this should free any memory allocated by the other two functions.
- The arguments pointer is stored internally and can only be allocated by an initialization function.

The initialization function is called as soon as a plug-in is loaded. The cleanup function is called when the function is unloades. It is not necessary for any code using the plug-in to have to manage the calling of these functions. It is all handled in the dl_library functions.

Plug-ins are loaded in the order specified in the configuration file. Only one input and output plug-in are allowed but any number of processing plug-ins may be loaded and implemented serially. The plug-in functions are arranged in a priority queue. It is set up such that if no priority is given all plug-ins have the same priority (99). Plug-ins with the same priority are executed in the order that they are specified in the configuration file. Processing plug-ins that are tied to a particular data point are linked to the structure defining the data element.

A plug-in is defined in the configuration file as follows:

```
START_DYNAMIC_FUNCTION
  DYNAMIC_FUNCTION_LIBRARY  ./cmrjcntb_plug-in.so
  DYNAMIC_FUNCTION          cmrjcntb
  DYNAMIC_FUNCTION_ARGS     32         ## The initial count
  DYNAMIC_FUNCTION_PRIORITY 1
END_DYNAMIC_FUNCTION
```

The example here is for a processing plug-in however input and output plug-in definitions are similar.

A library file (shared object) may contain many plug-ins as long as they come in sets defined by the DYNAMIC_FUNCTION name. A library file is only opened once. If a library file contains several sets of plug-ins subsequent loads only import the new functions specified in DYNAMIC_FUNCTION. A processing plug-in can be specified for more than one data point. For example a function may provide a certain calibration for both A and B side equipment. If a function is already loaded it is assumed it is to be run for each event but is not reloaded. If a function that has already been loaded needs to be reloaded with different arguments a new function structure is built linking to the same symbols in the library. This allows the same function to be run several times with different input. For example, several data points have the same type of calibration applied with different coefficients.

A plug-in is loaded into a structure like so:

```
typedef struct  _DLfunction
{
  int priority;
  unsigned char flags;

  DLement *library;

  void *args;
  void (*init_function) (void *args, void **out);
  void (*run_function)  (void *in, void **args, void **out);
  void (*clp_function)  (void *args);

  struct  _DLfunction *next;
} DLfunction;
```

The priority determines where in the queue the plug-in sorts. Plug-ins of the same priority are chained together as a list in the order they were specified. The core software determines if there is a plug-in associated with the current processing stage; if yes, it runs the appropriate function.

Toolkit recognizes certain output values as valid. If the output is NULL the code assumes that an error has occurred. The plug-in should also set errno appropriately and allow the main application to interpret and deal with it appropriately. If the core code receives no indication of why the error occurred then the main code attempts to exit cleanly.

Plug-ins can communicate with the main code and with other plug-ins in several ways including signals, SysV shared memory, and IPC sockets. Signal handlers in the main code are configured to receive certain standard signals from the functions and take certain action. This is one way that external applications written to interact with the plug-ins can send and receive messages. It's also one way that plug-ins can provide certain alternate output options. In addition some plug-ins use signal catchers as global flags intended to inform one plug-in of events in another. SysV style shared memory and semaphores can be used to pass information between plug-ins, however it is more common to use the more generic IPC sockets. These two IPC mechanisms are used to pass larger structures between plug-ins. There is no feature built in to Toolkit to support generic data feedback. These other mechanisms, allow the plug-ins to act in concert and gives one plug-in the ability to control the operation of another.

III. Plug-in Types

Input plug-ins are configured to read new data types and pass out a frame structure. In general the frame structure contains an array of binary data and a length. You can also pass a self-defined data structure assuming that the processing routines on the other end are provided knowledge of the structure as well. If an input plug-in is specified, it is called with the current configuration parameters and then the calling routine expects that it returns a pointer to data. That data is then passed to other routines in the architecture. For internally-defined routines the data is usually either TDM or SFDU as defined in the configuration file settings. In other words, it is globally expected that the data are binary with some regularly defined structure. Frequently, the input plug-ins convert whatever external data type they receive into an SFDU for the internal routines to process.

Processing plug-ins can be tied to a given point in telemetry or to the stream as a whole. These plug-ins usually update certain information contained either in the data stream or the ancillary structures, thus allowing downstream functions to process them. There are two types of processing plug-ins whose difference is chiefly the triggering mechanism. A general processing plug-in is called in priority order as the first step in the frame processing segment of the pipeline. They are always passed the entire frame for analysis. A processing plug-in tied to a particular data point is only called when processing that point. It is typically passed the whole stream in addition to the data point being modified, however it can be restricted to receive only a copy of the bits associated with the data point.

Output plug-ins are crafted to allow the system to interface with other systems or with users. Built-in output plug-ins include SFDU and TDM and two different internally defined packets. Output functions actually handle the output themselves, so that the report writer generates its own output file and writes the contents. Frequently, it is desired that the data output can be easily imported into spreadsheets or plotting tools. Some of the newer Toolkit plug-ins incorporate the ability to write out gnuplot scripts associated with the data set, or can call gnuplot directly to just display a plot of the desired data. Output plug-ins also allow the system to reformat the data and write it out to a file or to the network.

IV. Examples

SLE Data (Input Plug-in)

When Chandra was required to switch to an SLE interface with DSN, an input plug-in was designed to handle SLE PDUs. An external application handles the protocol and passes the PDUs to the plug-in for processing. SLE PDUs are transformed into an SFDU for internal processing. The PDU fields, specifically earth receive time, antenna id, data link continuity, and the DSN private annotation, are used to construct the SFDU header structure. This functionality is maintained as part of the Toolkit package mostly for analysis purposes since the external application that passes PDUs to Toolkit also handles forwarding data to the downstream clients.

Incorporating STK predictive data into reports (Processing Plug-ins)

STK predictive data is useful for analyzing many problems, and incorporating this data into Toolkit has proven handy when troubleshooting. A plug-in was crafted to retrieve data from the local instance of STK using STK's Connect language. The plug-in connects to STK and retrieves the predictive data for the timespan desired in the output. The processing plug-in associated with the particular data point being analyzed selected the appropriate STK data to return with the actual received data. The output was then received and plotted as part of the report. Currently this functionality exists only as report output but future versions of the software will allow users to view these plots in real-time. The use of a plug-in to incorporate STK predictive data into reports is presented as a detailed example in the appendix.

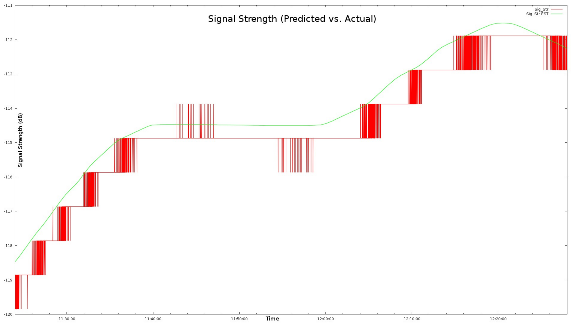

Figure 1: Predicted vs. Actual Signal Strength

Use of the system as a crude simulator (Processing Plug-ins)

A crude simulator was needed to take older Chandra data and modify it to reflect the current spacecraft configuration and respond to some very basic commands. For the details and the reasons please see my SpaceOps 2010 paper[1]. A general processing plug-in was written to allow Toolkit to perform the basic configuration changes needed. A stand-alone command processor was written to receive and process commands. Since a goal of the exercise was to simulate an uplink communications problem, only a few commands needed to be processed in detail, all others were simply "rejected on board”. A plug-in was written and tied to each of the data points that needed to be individually updated by the command software. The plug-ins and the command processor were tied together via IPC sockets and shared memory segments. The command processor received a command and decided what needed to be updated. It then flagged the plug-in handling the telemetry point that needed to be updated. This plug-in would then read pertinent information off the socket about how to update the point and modify the value in the telemetry stream. The data stream was fed to a standard output module that writes the data out in the internal system format. The result, from the point of view of the operators in the simulation, was normal-feeling communications with a limping version of Chandra that was implemented using the existing Toolkit tools and a minimal amount of additional code that was built exclusively for this purpose.

Data Encoder (Output plug-in)

While performing recent testing with DSN, it became apparent that the spacecraft data test files they had were not only out of date and of low quality but also contained data that reflected the spacecraft in safe mode. Since receiving these test data at the control center was problematic for several reasons, it was decided that Chandra would provide DSN with a new set of test files. These new files would be accurate, complete and well-defined depictions of the spacecraft telemetry, and would reflect normal operation rather than safe mode telemetry (which if accidentally released to the wider Chandra community during a test could cause worry). In other words, we would know exactly what was represented in the test file and could tell if something was different. To aid in generating the test data for DSN, an output plug-in was constructed to extract the spacecraft frame from the SFDU, then apply both Reed-Solomon and convolutional encoding. The files delivered to DSN were indistinguishable from data sourced from the spacecraft.

V. Conclusion

Moving forward, several improvements to the use of plug-ins can be identified. One such improvement would be the redesign of the core in C++, as this would allow C++ features such as constructors, destructors and overloaded functions to produce more efficient code. This would also allow the use of Boost plug-in libraries to provide more consistent implementation and use compared with individually-generated libraries.

The modular manner in which plug-ins can be implemented allows for minimal disruption to core components, thus maximizing flexibility. Further, a plug-in architecture allows users operating within the framework of an established software package, like Toolkit, to more easily manipulate necessary components with minimal investment. In this scenario, users would also be able to exchange code with relative ease. Plug-ins are a very valuable but little used tool that allow for many changes in the core operation of a data analysis pipeline. The use of plug-ins, while embraced by the software industry, remains largely under-utilized within the space community.

Appendix A

Incorporating STK predictive data into reports.

Processing Plug-in Example

(Note: some of the code has been edited for brevity)

From the configuration file

The original data point:

```
START_DATA
  ID CARRIER_POWER_RCV
  TYPE FEEE
  << Data Point Info >>
END
```

The new data point to be retrieved from STK: It is tied to the actual received data and only generates output when the original data point would generate output. $TCTKDS is a system variable that points to the install location of the software.

```
START_DATA
  ID CARRIER_POWER_RCV_EXP
  TYPE TIEIN
  TIEIN_MSID CARRIER_POWER_RCV
  START_DYNAMIC_FUNCTION
    DYNAMIC_FUNCTION_LIBRARY $TCTKDS/lib/stk_rcv_sig_str.so
    DYNAMIC_FUNCTION         stk_rcv_sig_str
  END_DYNAMIC_FUNCTION
END
```

The argument structure: In this particular case, it contains only the STK connect information. I probably could have just used the connect structure as the argument, however, it received its own structure in case additional parameters were needed.

```
typedef struct __STK_Rcv_Sig_Str_Data
{
  GSTK_Connect STK; // All the connection info from STK
} stk_rcv_sig_str_data;
```

The initialization function: It allocates the memory for the arguments and connects to the STK server. It is important to note that in the initialization function it is important to write the desired arguments out. If you want to retain those from the configuration file, they must be rewritten on the output.

```
void stk_rcv_sig_str_init (void *read_args, void **write_args)
{
  stk_rcv_sig_str_data *srssd;

  // Allocate space for srssd
  srssd = malloc (sizeof (stk_rcv_sig_str_data));

  // Get the configuration information for the STK server and connect to it
  GSTK_Load (srssd->STK);
  srssd->STK.read_config ();
  srssd->STK.stkconnect ();

  // Put the structure as the args for the main function
  *write_args = (void *)srssd;
} // END stk_rcv_sig_str_init ()
```

The main function: This creates the data for the new data point. It retrieves the needed data from the STK instance and calculates the signal strength. It then places that into the well-known data point structure that downstream functions can use to manage data point output.

```
void stk_rcv_sig_str (void *input, void **args, void **output)
{
  G_Frame *frame;        // The input frame
   dat_pos_info *dpi;       // The place to store the output value to pass to the downstream functions
  stk_rcv_sig_str_data *srssd;
  Frame_Time ft;         // The frame Time structure (CCSDS Time format)
  int site;              // DSS ID
  char *sr, *att;        // STK response for slant range and antenna angle
  float sig_str;         // The calculated signal strength

  // Give the arguments their actual types
  frame = (G_Frame *)input;
  dpi = (dat_pos_info *)(*args);
  srssd = dpi->function->args;

  // Use built in functions to get the time stamp and site ID from the frame
  ft = Get_Frame_Time (frame);
  site = Get_Frame_Site (frame);

  // Retrieve the data from STK
   sr = srssd->STK.command (< Get slant range from site to antenna at the same time as the frame data >);
   att = srssd->STK.command (< Get spacecraft antenna angle to site at the same time as the frame data >);

  // Calculate expected signal strength
   dpi->value = < Calculate expected signal strength based on slant range and antenna angle >;

  // Convert the signal strength into the data point structure
  dpi->value = f2c (sig_str);
} // END stk_rcv_sig_str ()
```

The cleanup function disconnects from STK and frees the memory the init function allocated

```
void stk_rcv_sig_str_cleanup (void **args)
{
  stk_rcv_sig_str_data *srssd;
  srssd = (stk_rcv_sig_str_data *)(*args);

  srssd->STK.stkdisconnect ();
  free (srssd);
  *args = NULL;
} // END stk_rcv_sig_str_cleanup ()
```

Appendix B
Acronym List

API	Application Programming Interface
CCSDS	The Consultative Committee for Space Data Systems
DSN	Deep Space Network
IPC	Inter-Process Communication
PDU	Protocol Data Units
SFDU	Standard Formatted Data Unit
SLE	Space Link Extension
STK	Satellite ToolKit
SysV	Unix System V
TDM	Time-Division Multiplexing

Acknowledgments

I would like to thank Jeff Holmes of Northrop Grumman and the Chandra team at SAO, specifically Adam Frye, for their support during the development of the tools and the writing of this paper. This work was supported under NASA contract NAS8-03060.

References

[1] Leussis, G., Shropshire, D., Wright, G., and Holmes, J., *Surprise Simulation: Managing an Anomaly Simulation Without Participant Knowledge*, SpaceOps 2010 Proceedings, Huntsville, AL, 2010