USE OF THE GROUND SUPPORT EQUIPMENT OPERATING SYSTEM (GSEOS) SOFTWARE ON THE MESSENGER MISSION: A CASE STUDY

Thomas F. Hauck

GSE Software, Inc.
Marina del Rey, CA 90292
hauck@gseos.com

Jeremiah V. Finnigan

The Johns Hopkins University Applied Physics Laboratory
Laurel, MD 20723
jerry.finnigan@jhuapl.edu

ABSTRACT

The need for low-cost, flexible, and maintainable spacecraft ground systems is a common denominator across most space programs. General requirements for such software should emphasize reduction of risk and cost across mission phases. The software package Ground Support Equipment Operating System (GSEOS) was designed to support the testing and integration of instruments and small spacecraft and runs on the Microsoft Windows® platform. The structure, capabilities, and future direction of GSEOS are based on reusability and cross-mission phase deployment. GSEOS can be used for integration and test (I&T) as well as flight operations with only small changes to the system configuration. GSEOS uses the open-source programming language Python as its scripting language. Python's flat learning curve facilitates a quick turn-around time for the development of command scripts, telemetry decoders, and test scripts. These attributes are demonstrated with the case study of the MESSENGER mission.

The MESSENGER (MErcury Surface, Space ENvironment, GEochemistry, and Ranging) mission [1] is a NASA Discovery program executed by The Johns Hopkins University Applied Physics Laboratory (JHU/APL). The science payload [2] consists of seven instruments and redundant Data Processing Unit (DPU) computers. The GSEOS-based MESSENGER Instrument Ground Support Equipment (IGSE) design supports instrument and DPU flight software development, unit-level test, spacecraft integration, and mission operations. A modular software design approach facilitates code reuse across the instrument suite and a broad range of test environments. The MESSENGER IGSE software is partitioned into common and custom modules; common modules contain methods and data reused by all instruments, whereas instrument-specific customizations are partitioned into separate modules simplifying code maintenance. An object-oriented command definition scheme allows rapid instrument customization by external organizations with minimal code development.

The MESSENGER Mission Operations Center (MOC) utilizes a proprietary scripting language called Satellite Test and Operations Language (STOL). The MESSENGER IGSE design incorporates STOL parsing and execution capability, allowing early testing of flight procedures and test script reuse. The similarity of Python and STOL syntax yields a surprisingly simple implementation.

The MESSENGER mission experience demonstrates that the GSEOS platform is well suited for developing low-cost IGSE. Partitioning of the IGSE software into common and custom modules, coupled with object-oriented design, results in effective code reuse.

1. INTRODUCTION

A common requirement of many spacecraft programs is the need to test and operate the hardware prior to launch and monitor its operation after launch. In recent years, spacecraft and missions have grown more complex while schedules have grown ever tighter. While such an environment puts pressure on all aspects of a mission, it is particularly difficult on the elements that support ground test and operations, since these elements must be in place to support early hardware development and yet must last throughout the mission.

Typically, this capability has often been met by developing and supporting several independent systems. The first, often called bench checkout equipment (BCE), is designed primarily to help engineers test the flight hardware in a stand-alone configuration. Due to the need to support early testing, the BCE's capability is often limited. Despite these limitations, and the fact that it is not intended for long-term use, the BCE development often requires significant resources to insure that the hardware delivery is not imperiled.

A second system, developed in parallel with the BCE, is intended primarily to support mission operations. Because such systems can't be fully checked until late in the hardware development cycle, significant resources are expended to run simulated hardware interface tests. However, such tests are not a complete substitute for testing with real hardware and invariably problems are discovered in the hardware and/or software. A third system may also be used solely to support system-level testing during spacecraft integration. It shares problems

with both of the systems described above; it must be available in time for system integration, but the opportunity for testing may be limited. The hardware and software elements that comprise these systems are often completely different, not only from each other but from those used on previous missions. This incompatibility arises because the engineers responsible for the systems have their own preferences and experiential backgrounds. In addition, the rapid changes in the hardware and software markets make such changes tempting, if not necessary, to avoid obsolescence. Besides system problems user interface problems arise due to the fact that the end user has to be trained and able to operate various disparate systems.

Such fragmentation and duplication of effort results in higher costs, more schedule and technical risk, and reduced capability. Clearly, the way to address these problems is to utilize test systems that can be used throughout a mission lifecycle and can be easily adapted for use on several different missions (see Figure 1).

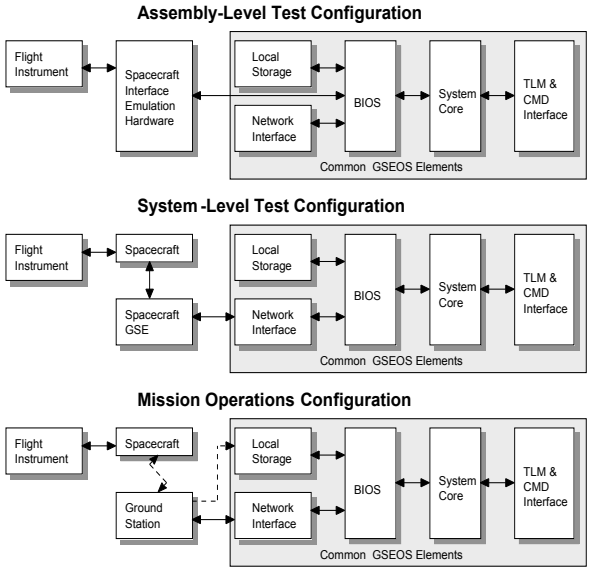


Figure 1. Mission lifetime configurations

2. GSEOS ARCHITECTURE

The Ground Support Equipment Operating System (GSEOS) tries to address the above mentioned challenges and reduce overall system costs. The next paragraphs give a general overview of the GSEOS system architecture and design philosophy. GSEOS is strictly a software product, but it is designed to interface easily to the hardware layer that is necessary to simulate physical spacecraft or system interfaces, such as the electrical and logical interface between an experiment and the spacecraft. If standard interfaces (MIL-STD-1553, RS-232, RS-422) are used, commercially available hardware can be purchased. The primary tasks of the GSEOS software are to send commands to and receive data from a subsystem, instrument, or spacecraft. Since telemetry and command definitions change quite frequently, especially in the beginning of the development phase, GSEOS is built to accommodate those changes quickly. For a flexible design data abstractions on the physical as well as the logical level are required. In GSEOS the physical data abstraction is done in the BIOS (Basic Input/Output System) module. This module is comparable to a device driver in an operating system and is strictly hardware dependent.

The basic assumption about the logical data format is that it is a repetitive data stream, which is true for most spinning spacecrafts that produce telemetry data. The data of interest are usually embedded in several protocol layers. The job of the BIOS is to accept data from the hardware and generate GSEOS data blocks. At this level it doesn't really matter how the BIOS splits up the serial data stream into data blocks. It is usually a good idea to use a data unit that is inherent in the interface hardware (e.g., the MIL-STD-1553 BIOS uses 32-word data blocks).

Static Data Representation

Data blocks are one of the underlying architectural concepts in GSEOS. Data are grouped in blocks that usually represent logically related data. This representation integrates especially well with the repetitive, fixed-frequency nature of most telemetry streams, but it also supports non-repetitive data streams such as those found with higher-level, "bursty" data products. GSEOS uses a **block definition file** that describes the layout of every block on bit level. The block description is similar to a "C" union construct and allows us to assign a name to every data item in a data block.

```
SubPacket
{
 ( Header[2] , , , 16; )
  Length   , , ,  9;
  Sequence , , ,  4;
  CF       , , ,  1;
  SensorID , , ,  2;
  SubSector , , ,  4;
* for all other data
  DataID   , , ,  2;
  DetectorID , , ,  2;
 ( Data8[980] , , ,  8; )
 ( ImgHdr   , , , 32; )
 }
```

This naming of data items introduces a layer of indirection in the access of data items that results in the

logical data independence. The data are referenced using these defined data item names with a BlockName.ItemName syntax throughout GSEOS. If, for example, the layout of a particular data block changes, only the block definition needs adjustment; the rest of the system data displays and scripts remain untouched.

As the instrument hardware and software evolve, the test system can easily be extended to accommodate the new functionality to be tested. From a users perspective, the data blocks and data items are the entities used to access the data. The user can interactively create display screens and place data items in various numerical and graphical formats on these screens. He can also record the data blocks to a file and play them back at a later time, export blocks over the network, write decoder scripts, etc. Every aspect of GSEOS deals with data blocks and data items; they are the backbone of the system. Although the block description allows for a static description of the data layout, it is not sufficient to model dynamic data.

Dynamic Data Representation

Due to bandwidth limitations, science data are usually highly packed or compressed, and housekeeping data are often sub-commutated. With only a static description of a data block it is not possible to display properly this kind of data packing. To model this dynamic layout we have to introduce additional complexity. In GSEOS this is done with a decoder script.

GSEOS historically used various proprietary, customized macro languages for decoding, monitoring, and system configuration tasks. This approach ran into scalability problems as we encountered more complex instrument and spacecraft requirements. All custom macro languages were replaced by a common Python interpreter that is integrated in GSEOS. Python scripts can access all GSEOS system features such as data blocks and data items as defined in the block definition file. This allows for a very coherent configuration and programming approach.

A decoder script written in Python can access all data items as defined in the block definition. The general idea behind the decoder script is to wait for a certain block (or multiple blocks) to arrive, perform the necessary algorithm on the data, and generate a new data block (which must already be defined in the block definition file). This new data block can then be used in the same manner as all other data blocks in the system (e.g., in displays, etc.). Most importantly, it can also be used in another decoders as a source block to generate further blocks, thus supporting a hierarchical approach to processing data blocks using decoder scripts.

A simple example shows how raw telemetry blocks are decoded into Status data when the application ID (ApID) indicates Status data. The resulting status block is then decoded further to generate a de-sub-commutated housekeeping array.

```
# Generate 20 byte status block from TLM block
def fDecTLM(TLM):
  if (TLM.ApId == STATUS_APID):
    BlkStatus.Data[0:20] = TLM.Data[0:20]
    BlkStatus.send() # Send status block

# Create decoder
oDec = Decoder.Decoder('TLM', fDecTLM, [Status])

#Register TLM Decoder on RawTLM arrival
RawTLM.Decoders.append(oDec)
```

This approach of decoding the instrument data leads to high visibility and easy debugging of the instrument hardware and software. It is easy to detect where the decoding fails by examining the integrity of the data as they are transformed through various levels of decoding. All the data products created on this decoding path are themselves only general data blocks and can be displayed by the system. Again, this allows GSEOS to adapt to the current development state of the instrument system and leads to a layered structure by defining decoder scripts together with block definitions and screen displays for every protocol layer. Usually this happens naturally as the complexity of the instrument evolves.

Layering the data products in this fashion allows multiplexing of data blocks from different data streams into common destination blocks. For example, an instrument may have a MIL-STD 1553 interface to a spacecraft. During stand-alone bench checkout, GSEOS may create 1553Raw Blocks from the data it receives via a 1553 card in the PC. The system then uses decodes these 1553Raw blocks into TLM blocks. During mission operations, however, the ground segment may deliver data to the PC in Standard Formatted Data Unit (SFDU) format via a TCP/IP link. GSEOS might create SFDU blocks, which would then be further decoded by a SFDU-specific decoder into TLM blocks. In both cases, the result is identical telemetry blocks, even though the input formats are completely different. That means all the displays and further decoding which is derived from the TLM block can be reused without any change. **This feature is one of the reasons for GSEOS' scalability**. Operators may become accustomed to certain screen layouts and command interfaces and use them throughout the system's lifecycle. Another advantage is that it is possible to switch instantly back to bench test configuration for calibration purposes if the need arises.

Commanding

The system must also accept and process command inputs. The same block structure utilized for telemetry applies for commanding. GSEOS uses a dedicated command block to pass commands within the system.

GSEOS offers several different command user interfaces. Once defined, the instrument commands can be assigned to buttons placed on display screens, invoked via a hierarchical command menu (see Figure 2), or entered through the command line editor. These methods all generate the same command block that contains the command to be issued. This architecture allows for easy extension of the command user interface, remote commanding (by sending a command string over the network), or even recording a command session and playing it back at a later time. Use of the pull-down menus is particularly helpful, since it eliminates the source of most errors during command generation and allows casual users a simple, intuitive means of interacting with the hardware.

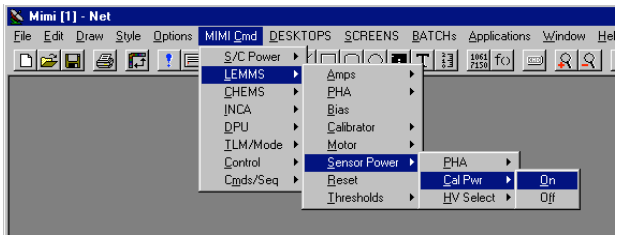


Figure 2. GSEOS Command Menu

GSEOS User Interface

One of the main design goals of GSEOS is its ability to enable users to rapidly display the instrument data in various formats. The displays can be easily created using the built-in screen editor. The editor allows users to "draw" telemetry data items, static text, and graphic objects on a screen and modify their attributes. Some of the options are binary, hex, decimal, integer, floating, histogram, 2-D scatter plot, “stripchart”, etc. Colors and fonts may likewise be modified to improve appearance and legibility. All this can be done while the system is running and receiving data. This feature is especially useful to create screens "on the fly"; e.g., while debugging the instrument data one can create specific views of the data to get a better insight into the instrument behavior.

Users will generally create different screens for different aspects of the instrument (see the layering approach above). Multiple screens can be arranged on a page to provide useful views of the data on the desktop (see Figure 3). These pages can also be grouped together and saved as desktop files; tabs can be used to quickly access the different pages on the desktop. Such groupings have proven particularly useful when testing instruments with multiple sensors; it also simplifies the task of rapidly switching test configurations.

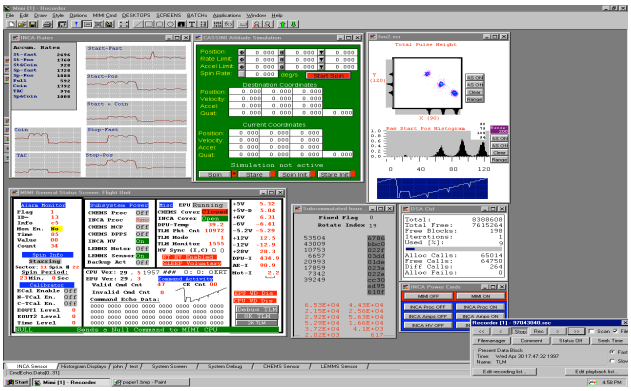


Figure 3. GSEOS User Interface

Extensibility

An open system structure allows support of the various instrument requirements. Usually all the common decoding issues as well as more demanding tasks such as image compression can easily be coded as a Python script. However, if the performance should prove insufficient, it is fairly easy to move code from Python to C. Python was designed with extensibility in mind, so critical parts can be coded in C and loaded from an extension dynamically linked library (DLL). There are a number of readily available off-the-shelf Python modules; for example, use of an existing database module would allow one to export data directly into a database. Writing a BIOS module for custom hardware is done via a Python extension module.

3. MESSENGER INSTRUMENT SUITE ARCHITECTURE

As an example, consider the application of the GSEOS platform to the development of the MESSENGER mission Instrument Ground Support Equipment (IGSE). The MESSENGER science payload consists of a suite of seven instruments (see Figure 4). Each instrument communicates with redundant Data Processing Unit (DPU) computers through a pair of serial RS-422 interfaces. The DPU computers reformat instrument telemetry packets and forward them to the Integrated Electronics Module (IEM) computers for downlink. Instrument commands that are uplinked to the IEM are forwarded to the DPU via the 1553 bus. The DPU forwards the command to the appropriate instrument based on the Application ID (ApID) of the command header. The MESSENGER instrument design incorporates a common hardware element called an Event Processing Unit (EPU). All EPUs support a common set of commands and telemetry definitions to facilitate code

reuse amongst all instruments. In addition, each instrument may define instrument-specific commands and telemetry packets.

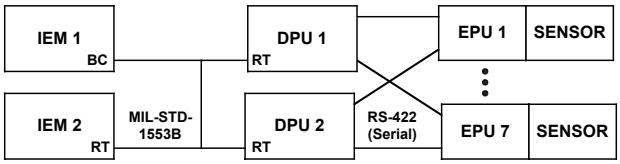


Figure 4. MESSENGER Instrument Suite Architecture

4. MESSENGER IGSE REQUIREMENTS

The MESSENGER IGSE was required to support a wide range of test configurations. In the instrument bench-top test configuration (Figure 5a) the IGSE is referred to as a DPU Emulator and interfaces to the flight instrument through a signal conditioning electronics box that provides a serial RS-422 command and telemetry interface to the instrument. The DPU Emulator is required to support a proprietary Instrument Transfer Frame (ITF) communication protocol to the instrument.

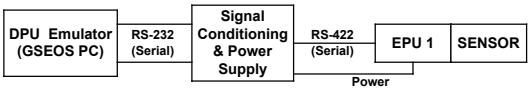


Figure 5a. MESSENGER Instrument Benchtop Test Configuration

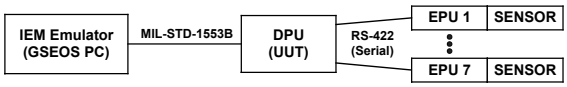


Figure 5b. MESSENGER DPU Benchtop Test Configuration

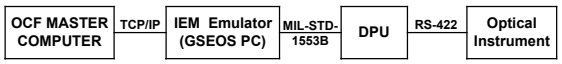


Figure 5c. MESSENGER Optical Calibration Facility (OCF) Test Configuration

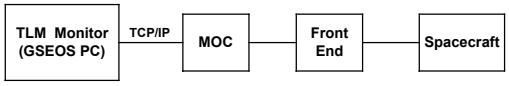


Figure 5d. MESSENGER MOC Telemetry Monitor Configuration

In the DPU bench-top test configuration (Figure 5b) the IGSE is referred to as an "IEM Emulator". It interfaces to the flight DPU hardware through a commercial off-the-shelf (COTS) MIL-STD-1553 interface board, and is required to support a proprietary Inter-processor Transfer Frame (IPTF) communication protocol with the DPU. The Optical Calibration Facility (OCF) test configuration (Figure 5c) uses a variation of the IEM Emulator that can act as a slave under the control of the OCF master computer through a TCP/IP socket. The OCF master computer may send commands to the optical instrument and receive telemetry from the optical instrument through the IEM Emulator acting as a slave unit. Finally, in the Spacecraft Integration and Test (I&T) configuration (Figure 5d) the DPU Emulator or IEM Emulator is configured as a telemetry monitor, receiving telemetry packets from the Mission Operations Center (MOC) during I&T or mission operations.

5. MESSENGER IGSE SOFTWARE ARCHITECTURE

The MESSENGER IGSE software design presented several challenges, including rapid development of the 8 IGSE test sets, each being reconfigurable to support a multitude of test configurations, and ensuring compatibility and maintainability of custom code developed by geographically separated teams. To meet these challenges, the MESSENGER IGSE software architecture incorporated modular code designed for reuse, reconfiguration, and ease of software maintenance. This approach is illustrated by the design of the IGSE command processing mechanism (Figure 6). The high-level command processing and routing features of the IGSE were implemented in a GSEOS Python module named 'core.py'. All flavors of the IGSE start by loading this module.

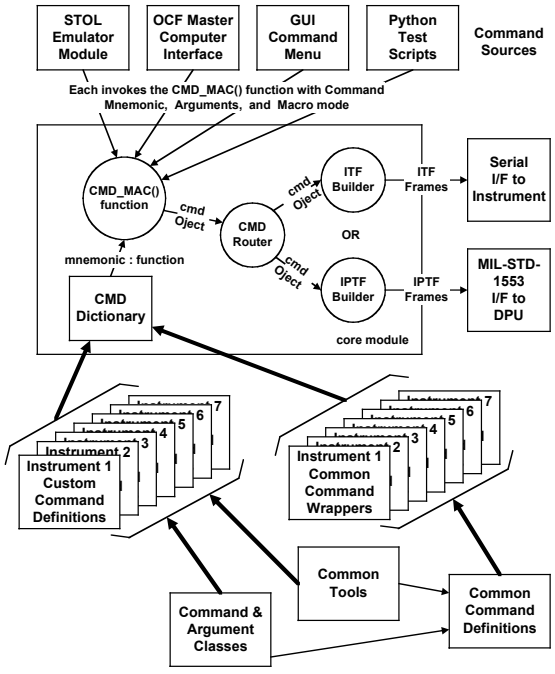


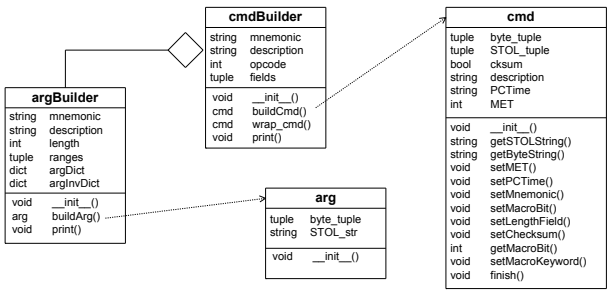
Figure 6. MESSENGER IGSE SW Command Processing Architecture

Depending on the initialization parameters specified in the configuration files gseos.ini and startup.cpb, the core module imports command dictionaries defined in various common and custom folders and appends them to the core command dictionary. The central element of the core module is the CMD_MAC() function that takes the command mnemonic and the command arguments (specified either as integers or as mnemonics) as inputs,

builds the corresponding instrument command, and sends it to the command router. The command router forwards the command to the appropriate formatting module (either ITF of IPTF) depending on the IGSE configuration. The command dictionary (a Python hash table) in the core module uses the instrument command mnemonic (represented as a string) as the key, which is mapped by the hash table to a reference to the function that builds the corresponding command. When the CMD_MAC() function is invoked, it finds the command-building function by using the supplied cmd_mnemonic to access the command dictionary and passes the command arguments to this function. If cmd_mnemonic is not in the dictionary key list, an exception is raised containing a description of the error

The command-building functions that are referenced in the core command dictionary must meet the following requirements: they must verify the correct number and type of arguments received, verify that all arguments are within the valid range, and raise an exception to the calling method with a description of the error if any of the command arguments are invalid. Otherwise the function must build and return a 'cmd' object to the calling method. The cmd object includes a data member that defines the byte-tuple representation of the command. (Definition: byte-tuple – a tuple of integer values, each value in the range of 0x00 through 0xFF.) Thus the core module's command dictionary, along with the requirements imposed on the command-building functions referenced in this dictionary, effectively decouple the high-level processing and routing of commands in the core module from the lower-level implementation details of the command-building functions.

The cmd class is one of four classes that are used for command definition and processing within the MESSENGER IGSE (see Figure 7). The cmd class encapsulates all of the data related to an instrument command including the byte-tuple representation of the command, the STOL string invocation that represents the command, the mission elapsed time (MET) when the command was created, the local time when the command was created, and a text description of the command.



Figue 7. Command Definition and Processing Classes

A cmdBuilder class that encapsulates all of the data and methods required for defining and processing an instrument command simplifies construction of the command functions. A related class is the argBuilder class, which encapsulates the definition and processing of a single argument field of a command. The buildArg() method of this class accepts either string or integer-valued inputs and determines if the input is either a valid mnemonic string or an integer within the allowed range. If an error is detected, the argBuilder.buildArg() method raises an exception with a description of the error. Otherwise it returns an arg object.

The arg class encapsulates the description of a single argument field of a command. It includes data members that describe the byte-tuple representation of the argument field plus the STOL string representation of the argument (either the mnemonic corresponding to that value or the string representation of the value if no mnemonic exists).

The cmdBuilder class constructor requires a list of parameters that defines the command mnemonic, the command description, the opcode, and a list of zero or more argBuilder objects that represent the argument fields of the command. The cmdBuilder.buildCmd() method is the command-building function that is referenced in the core module command dictionary. It accepts command arguments as input and builds and returns the corresponding cmd object (if no input errors are detected). It verifies that the correct number of arguments are received and then passes each argument to the buildArg() method of each corresponding argBuilder object stored in the cmdBuilder object's 'fields' tuple. The buildCmd() method builds the byte-tuple representation of the command by concatenating the byte-tuple representation of each argument field returned by the argBuilder.buildArg() method invocations and adding the appropriate command header prefix and checksum suffix. The resulting cmd object is returned to the invoking routine (the CMD_MAC() method in this case). In this way, the details of the command and argument processing are pushed to the lowest level of the call-chain, encapsulating the details of this processing in lower-level classes. Higher-level modules such as 'core' and the STOL Emulator are decoupled from the details of the command and argument processing, thus facilitating future code reuse.

The design of the instruments, based on a common EPU hardware and software design, facilitated the reuse of code in the IGSE. A common set of commands was required to be supported by all of the instruments. This common set of commands is implemented once as a set of generic commands that can be customized for each instrument using a wrapper function that specifies the command mnemonic and destination code to supply in

the command. Thus all common commands are defined in one module and reused for each instrument DPU flavor. A change to a common command requires updating the command definition in only one place.

Instrument custom commands are implemented in separate modules contained in an instrument-specific folder. This partitioning was performed to simplify code maintenance. Common commands and the required wrapper functions for each instrument were developed and maintained in the common folder by the JHU/APL IGSE development team in Laurel, MD. External development teams in Greenbelt, MD, and Boulder, CO, created and maintained the definition of instrument-specific custom commands for two of the instruments. Partitioning the code into separate common and custom folders allowed each team to update/modify their command definitions without the need to merge the contents of individual files. On startup, the core module imports two command dictionaries for each instrument – one from the common folder that contains the common commands, and one from the instrument specific folder that contains the instrument-specific commands. The IEM Emulator was required to be able to support instrument commanding for all of the instruments simultaneously, so it imports both of these command dictionaries for all instruments. The DPU Emulator flavors load the custom and common command dictionaries only for a particular instrument.

6. STOL EMULATOR

The MESSENGER spacecraft uses a version of the Satellite Test and Operations Language (STOL), a programming language to control and verify system operation. It is desirable to be able to develop and execute these STOL scripts directly within the GSEOS test environment. This capability will help migrating command scripts from I&T to flight operations and will allow for the option of developing and testing those scripts in a simulated environment before integrating them into the ground system.

The STOL Language

The STOL language is a simple procedural, loosely typed, scripting language. It supports integer, real, character, and time formats with no type checking at compile time and uses a coercion mechanism to map the types appropriately. Basic arithmetic and mathematical functions as well as conversion functions for telemetry points are available. STOL allows the use of local and global variables, which need to be declared before use. Basic flow control statements such as:

IF…THEN…ELSE…ENFIF
DO WHILE…ENDDO

are supported. The GOTO statement is supported as well. Procedure calls are available and invoke a new script. An interesting concept is **Synchronous Telemetry Access.** The following statements add telemetry points to a frame list and execute a procedure on the arrival of a telemetry point:

Frame Add Point1 Procedure1
Frame Add Point2 Procedure2
Frame Add Point3 Procedure3
Frame Process

This arrangement facilitates the synchronous handling of asynchronous telemetry events and therefore a procedural approach for writing closed-loop test scripts.

The Python Language

Python is an interpreted, object-oriented, high-level programming language with dynamic semantics. It facilitates high-level built-in data structures, combined with dynamic typing and dynamic binding. Python's simple, easy-to-learn syntax emphasizes readability and therefore reduces the cost of program maintenance. Python supports modules and packages, which encourages program modularity and code reuse.

Implementing STOL with Python

The STOL emulator started out as a simple text preprocessor. We wrote our STOL scripts and embedded Python code in STOL comments. This Python code more or less performed the same functions as the STOL code. At this point it was the job of the engineer to write the appropriate Python code to simulate what the STOL code was intended to do. The preprocessor ignored all STOL statements and extracted the embedded Python commands and executed them. There are a couple of drawbacks with this approach. First, the code gets duplicated as STOL with Python code in comments, which makes readability suffer. Second, the appropriate Python code needs to be written, which requires knowledge of STOL and Python, as well as the GSEOS-specific extensions. Third, the code needs to be kept in sync manually. It quickly became obvious that a more integrated solution was needed. The goal was to interpret the STOL code and to execute it within GSEOS to solve all the aforementioned problems. Since we already started with generating Python code for the STOL procedures (in a manual way) we decided to use Python as the back-end language, that is, the STOL emulator takes STOL code, translates it into Python code, and executes it within GSEOS. We found that most STOL statements and constructs resemble those of Python. This made the translation process straightforward, namely a simple lookup table that converts STOL keywords into Python keywords where appropriate. This approach covered

about 80% of the STOL syntax. However, there are some STOL constructs that require more attention, including implementation of the unstructured GOTO statement, synchronous telemetry access, and procedure calls. The following sections address these challenges and how we implemented them.

GOTO Statements

Python is a structured programming language and does not provide a GOTO statement. The challenge is to implement the GOTO statement with only structured flow control mechanisms. We chose to model the STOL GOTO statement with Python exceptions. During the compile phase we generate Python code that raises the STOLGoto exception and passes the target as exception arguments. The exception is caught at the end of the procedure and transfers control to the beginning. Now the problem is to move to the proper position within the generated code. There is no direct way of accomplishing this, however. We decided to prefix every possible GOTO target (the LABEL statement) with some conditional code to check for the target arguments to match. This way the code falls through until it finds the proper GOTO target and resumes execution there.

Synchronous Telemetry Access

GSEOS implements a class Sequencer that allows the event driven nature of block arrivals to be addressed with a sequential programming model. This makes writing closed-loop test scripts simple and easy to read and maintain. STOL's FRAME processing was implemented using GSEOS sequencers.

Procedure Calls

STOL procedures are separate STOL scripts that can be invoked with the START directive or with a FRAME PROCESS statement. Parameters can be passed either by position or by keyword. For our implementation a STOL procedure invocation looks just like starting a new STOL script. Procedure calls are implemented recursively by instantiating a new instance of the STOL emulator class, since we hold all global STOL variables in the Python global namespace and all local STOL variables local to the specific STOL emulator instance.

Graphical Front-end

The STOL emulator interacts with a graphical user interface to display the source code during execution as well as giving the operator the opportunity to start, stop, pause, and skip wait statements. This is a GSEOS extension module written in C# using Microsoft's .NET framework.

Limitations

Strict error checking proved difficult. If valid Python statements that are not valid STOL code are entered, the script will run fine in GSEOS and fail on the spacecraft. In general errors are caught as Python exceptions and have to be translated into a meaningful STOL error. Due to STOL's simplicity, sophisticated error checking is not of utmost importance. Implementing STOL is easy where the functionalities of STOL and Python overlap. Some features of STOL like the GOTO statement take some effort to implement in Python. The synchronous telemetry access of STOL can be mapped to GSEOS' Sequencers. Including a graphical user front-end, the entire effort could be completed within two person weeks.

7. SUMMARY

Due to the embedded Python interpreter GSEOS has substantial flexibility and potential for customization. This was demonstrated by the intricate implementation of the MESSENGER commanding scheme as well as the implementation of the STOL programming language within GSEOS. Code modules allow a natural way to structure complex systems and facilitate team development and customization.

8. REFERENCES

[1] Solomon, S. C., R. L. McNutt, Jr., R. E. Gold, M. H. Acuña, D. N. Baker, W. V. Boynton, C. R. Chapman, A. F. Cheng, G. Gloeckler, J. W. Head, III, S. M. Krimigis, W. E. McClintock, S. L. Murchie, S. J. Peale, R. J. Phillips, M. S. Robinson, J. A. Slavin, D. E. Smith, R. G. Strom, J. I. Trombka, and M. T. Zuber, The MESSENGER mission to Mercury: Scientific objectives and implementation, *Planet. Space Sci., 49,* 1445-1465, 2001.

[2] Gold, R. E., S. C. Solomon, R. L. McNutt, Jr., A. G. Santo, J. B. Abshire, M. H. Acuña, R. S. Afzal, B. J. Anderson, G. B. Andrews, P. D. Bedini, J. Cain, A. F. Cheng, L. G. Evans, W. C. Feldman, R. B. Follas, G. Gloeckler, J. O. Goldsten, S. E. Hawkins, III, N. R. Izenberg, S. E. Jaskulek, E. A. Ketchum, M. R. Lankton, D. A. Lohr, B. H. Mauk, W. E. McClintock, S. L. Murchie, C. E. Schlemm, II, D. E. Smith, R. D. Starr, and T. H. Zurbuchen, The MESSENGER mission to Mercury: Scientific payload, *Planet. Space Sci., 49,* 1467-1479, 2001.