The Next Generation of Ground Operations Command and Control; Scripting in C# and Visual Basic

George Ritter[1]
Computer Science Corporation (CSC), Huntsville, Alabama, 35806

And
Ramon Pedoto[2]
COLSA Corporation, Huntsville, Alabama 35806

Scripting languages have become a common method for implementing command and control solutions in space ground operations. The Systems Test and Operations Language (STOL), the Huntsville Operations Support Center (HOSC) Scripting Language Processor (SLP), and the Spacecraft Control Language (SCL) offer script-commands that wrap tedious operations tasks into single calls. Since script-commands are interpreted, they also offer a certain amount of hands-on control that is highly valued in space ground operations. Although compiled programs seem to be unsuited for interactive user control and are more complex to develop, Marshall Space flight Center (MSFC) has developed a product called the Enhanced and Redesign Scripting (ERS) that makes use of the graphical and logical richness of a programming language while offering the hands-on and ease of control of a scripting language. ERS is currently used by the International Space Station (ISS) Payload Operations Integration Center (POIC) Cadre team members. ERS integrates spacecraft command mnemonics, telemetry measurements, and command and telemetry control procedures into a standard programming language, while making use of Microsoft's Visual Studio for developing Visual Basic (VB) or C# ground operations procedures. ERS also allows for script-style user control during procedure execution using a robust graphical user input and output feature. The availability of VB and C# programmers, and the richness of the languages and their development environment, has allowed ERS to lower our "script" development time and maintenance costs at the Marshall POIC.

Nomenclature

ASP	=	Active Server Page
CLI	=	Command Line Interface
CSC	=	Computer Science Corporation
C2	=	Command and Control
C3ISR	=	Command, Control, Communications, Intelligence, Surveillance and Reconnaissance
C#	=	'C' "sharp", a programming language
EHS	=	Enhanced HOSC System
EPC	=	Enhanced Personal Computer (MSFC HOSC Telemetry and command tool-set)
ERS	=	Enhanced and Redesigned Scripting Language
HOSC	=	Huntsville Operations Support Center
HTML	=	Hypertext Markup Language
ISS	=	International Space Station
MSFC	=	Marshall Spaceflight Center
PHP	=	Hypertext Preprocessor
POIC	=	Payload Operations Integration Center
SCL	=	Systems Control Language
SLP	=	Scripting Language Processor
STOL	=	Systems Test and Operations Language
TSTOL	=	The System Test and Operations Language
VB	=	Visual Basic
VS	=	Visual Studio

[1] Software Development Team Lead, Software Engineering, CSC, 310 Bridge St., Huntsville, Al. 35806
[2] Computer Scientist, Software Engineering, COLSA Corporation, 6728 Odyssey Drive, Huntsville, Al. 35806

I. Introduction

A Script serves as a pre-planned set of instructions to execute or perform a task such as the dialog in a play or a small job in a computer. While traditional programming languages evolved primarily for the purpose of solving complex, computational intensive problems, scripting languages or scripts have become mainstream tools for solving more of the "house-keeping" computer problems like managing file systems. Where the focus on compiled programming languages is on performance, script commands are geared towards ease of use. In the typical script command, task performance is less of an issue permitting the use of an interpreted environment.

Control of space systems from ground operations sites requires a certain amount of repetitive or scripted actions to control remote systems. The ease of use of script commands makes creation and modification of ground initiated flight procedures simple and less prone to error. A number of flight operation scripting languages have evolved from today's commonly used computer scripting languages. These scripting languages integrate with the native command and telemetry systems through unique script commands.

The Marshall Space Flight Center's (MSFC) Huntsville Operations Support Center (HOSC) has developed a tool called the Enhanced and Redesigned Scripting Language (ERS). ERS combines the ease of use and low risk potential of the typical interpreted scripting language with the power and richness of a full programming language. ERS has streamlined development of scripts and enhanced remote control of on-board systems at the Huntsville Payload Operations Integration Center (POIC).

II. The Value of Scripting in Flight Operations

Since the beginning of scripting languages like IBM Job Control Language (JCL) to the current object oriented languages like Ruby and Perl, system administrators and software developers have sought to "avoid the compiler" in an effort to simplify common and repetitive tasks that are not performance oriented. Space ground systems have adapted and extended scripting concepts to the command and telemetry processing domains. What makes the interpreted languages so useful in the Space Operations world? Is there a more advanced solution that combines the graphical richness of a complied language with the ease and flexibility of a script development environment?

A. Mini-History of Scripting Languages

In the early 1960's IBM introduced Job Control Language (JCL) on their OS/360 where files could be copied from one location to another using only 9 lines of instructions! JCL was followed by Data General's Command Line Interface (CLI) and later followed by the Unix Bourne Shell. These Scripting Languages store a series of commands in a file. Data General called them "macros". Unix called them "shell scripts." In all cases, the scripts ran as interpreted statements where no time was spent compiling. Local variables and flow control were slowly added. As complexity grew, these languages continued to be interpreted, most likely because compute power was also increasing.

A small list of the more popular scripting languages of today, often calling themselves dynamic programming languages, includes Perl, Python, Ruby, Hypertext Preprocessor (PHP), Active Server Page (ASP), and JavaScript. Pearl is known for its text processing capabilities; Python for readability and object oriented constructs; Ruby for object oriented; and PHP, ASP, and JavaScript for their ability to be used on Web applications inside Hypertext Markup Language (HTML) files.

Today's scripting languages have progressed beyond simple file manipulation to common (and even complex) programming tasks. They satisfy the needs of providing fast-to-develop and easy-to-maintain programming solutions in many of today's computer problem domains.

B. Some Space Ground Operations Scripting Languages

Companies such as SRA International and some NASA Centers have found scripting languages very powerful for space operations applications.

Systems Control Language (SCL) developed by SRA International "*is a full-featured scripting language with which you can easily build, test and operate diverse control systems across mission critical command and control (C2) and Command, Control, Communications, Intelligence, Surveillance and Reconnaissance (C3ISR) domains. SCL greatly reduces workload and automates routine tasks through procedural, time-sequenced and event based*

responses to real-time data." We would have to have a closer relationship with SRA to show more details of SCL, but they do go on to list features of SCL that include "Full-featured scripting language."[1] SCL also provides interfaces to let you adapt your proprietary C2 and telemetry acquisition systems to the SCL engine.SCL was chosen by Kennedy Space Center as the script engine for the Constellation Launch Control System.

The Systems Test and Operations Language (TSTOL) is an interpreted language developed at Goddard Space Flight Center and "*is derived from generations of the Systems Test and Operations Language (STOL) used in existing NASA satellite control centers. TSTOL is a procedural command language consisting of a core set of generic commands, supplemented by mission-specific extensions*."[2] TSTOL includes typical programming capabilities such as various data types, arithmetic, logical, and relational operators, global and local variables, and looping constructs. TSTOL also has built in "procedures" or commands specific to the Goddard Mission Systems. TSTOL allows for the creation of custom procedures or commands so that the language can be adapted to new programs and interfaces.

The Scripting Language Processor (SLP) at the MSFC POIC and also currently used by the Chandra X-ray Observatory Control Center is based on the TSTOL design and provides the operations teams with the ability to develop scripts and to control script execution. SLP scripts consist of text files made up of statements, called directives, which the SLP interpreter can recognize and execute. In SLP many of the same things can be done as in TSTOL with respect to arithmetic, logical, and looping functions. The SLP directives also include POIC (and Chandra) spacecraft command and telemetry specific actions for remote control of the ISS Payload (and Chandra spacecraft).Examples of the SLP directives are included in Table 1.

Table 1. Sample SLP Directives

ask *prompt [variable]*	Prompts the operator for input and waits until a **resume** directive is entered. The value entered is stored in *variable*. If *variable* is not specified, the process is halted until the user enters a **resume** directive. *Prompt* is a quoted string that will be displayed to the operator.
Sample next *MSID_identifier function*	Samples the oldest unread received value(s) of the specified MSID, and stores the returned telemetry value(s) into the local or global script variable specified.
update command *command_mnemonic* **from file** *binary_image_filename*	Updates a variable length DLC command. The file must be in the predefined binary image format, and may not contain more data than can fit in a single command. The file must reside on the PIMS server. See Appendix C: Data Load Commands.
uplink command *mnemonic* [**verify car/fsv/crr**]	Initiates the transmission or uplink to the spacecraft for the command referenced by a unique mnemonic as defined in the Operational Command Database.

C. Why Scripts for Ground Operations

Compared to compiled languages, scripts are easier to write due to the simple and limited language syntax. When the language is simple, training time is decreased and the need for language experts is lessened. Scripts are also faster to develop because each statement does lots of work, i.e., "uplink command," and because the script developer gets immediate feedback from the interpreter. Scripting languages also provide easy to use constructs for user-controlled program flow. In many operations scenarios, it is necessary to have flight operations personnel monitor the script or operations procedure progress and respond to queries. With SCL, TSTOL, and SLP, the script user has many control options.

Scripting languages are not historically known for being rich in graphic capabilities. At the MSFC POIC, the operations Cadre personnel use a combination of scripts (SLP and now ERS), a data display tool, and custom programs or comps (compiled languages) to provide them with the best combination of all the features they need to automate their flight operations tasks.

We wondered if it was possible to combine the features of a scripting language like TSTOL and SLP with the graphical richness of a simple programming language like Visual Basic. Some of the scripting language syntax for standard program operations had become more complex and more work to maintain just to provide features that basic programming languages already offer. Visual Basic and C# programmers are becoming more readily available. The idea was to let programming languages do what they do best, and add in classes and methods (script commands or directives) that do lots of work, i.e. "uplink command." We also had to develop a way to provide run-time "script" control that made execution feel and act like a script. Thus was born the Enhanced and Redesigned Scripting language or ERS.

III. ERS

The Enhanced and Redesigned Scripting (ERS) is one of a suite of applications of the Enhanced Personal Computer (EPC) tool at the MSFC POIC. EPC is a Windows based tool-set that allows local and remote POIC users to display and analyze telemetry, and uplink commands to the ISS payload systems for control purposes. ERS is the latest "scripting" product developed at Marshall to support procedural style remote operations control.

A. Script Creation and Script Help

ERS uses the Microsoft Visual Studio programming environment for development and also offers custom controls that enable an ERS developer to extend the Visual Basic and Visual C# languages to interface with the POIC telemetry and command system. Although the "programs" are compiled, ERS offers execution control features that make an ERS program feel and operate like a script. Visual Basic and Visual C# provide a rich set of graphical development tools that make for an extremely user friendly end-product. In ERS, the richness of a full featured programming language is combined with the flexibility and user control of a script. ERS consists of ERS Operations, Microsoft Visual Studio, and a number of custom wizards developed at Marshall to provide point and clink integration with the POIC ground system's core libraries.

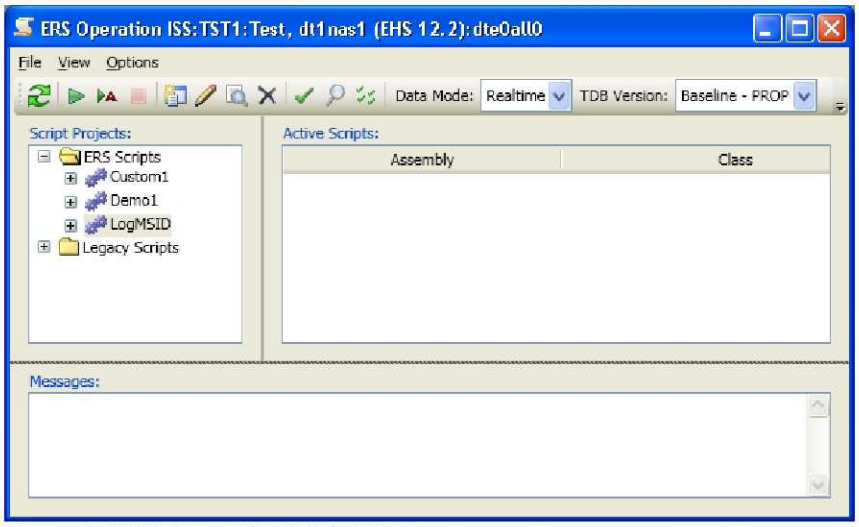


Figure 1. ERS Operation Main Window

The ERS Operations application, Fig. 1, is the main entry-point into the ERS system. Here the user can create, edit, delete, and run ERS scripting projects. Each ERS "script" is actually a .NET project that must be first compiled before it is run. Part of the compiling phase includes steps that validate the correctness of an ERS script. From ERS Operations, scripts can be validated and then run in various data modes including real-time and playback.. ERS Operation also displays any currently running ERS scripts.

When creating a new ERS project, the ERS developer selects "add new script" from the File menu option in ERS Operation, which will launch Visual Studio's New Project Wizard, Fig. 2. In the New Project Wizard, the user then selects a desired programming language, the ERS Script Project template and a project name.

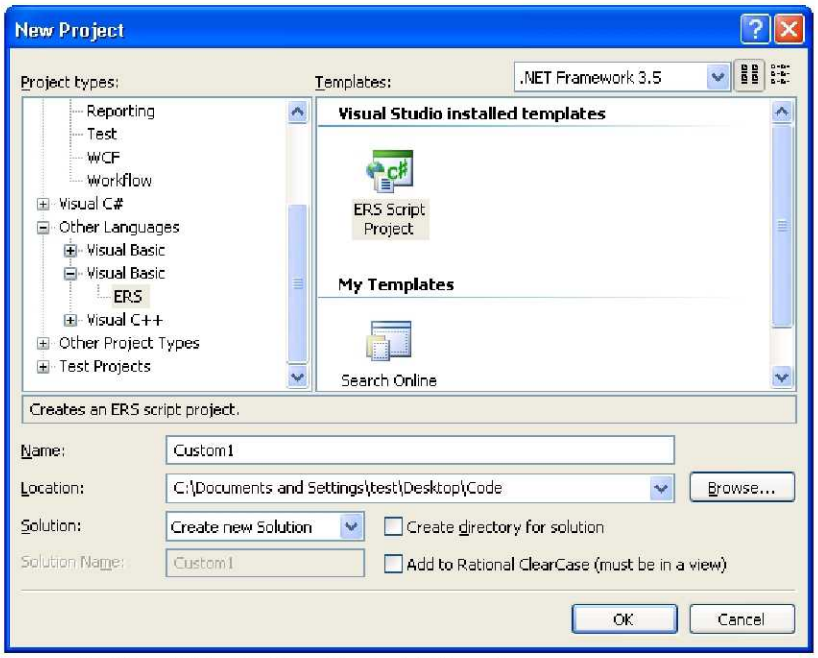


Figure 2. Visual Studio New Project Wizard

Clicking "okay" brings up the ERS New Project Wizard, Fig. 3, and the user must select an active EPC session to run against. The EPC session selection connects the ERS project to a specific set of telemetry and command meta-data within the POIC ground system. Clicking "okay" causes this Wizard to create a new .NET project that contains a code file plus an ERS-specific Validation File (Fig. 4: "DemoScript.Validation.ers"). The code file is where the user-defined scripting logic is inserted. The validation file is used to map EPC telemetry and commanding objects into the project. These objects then become accessible in the user-defined code.

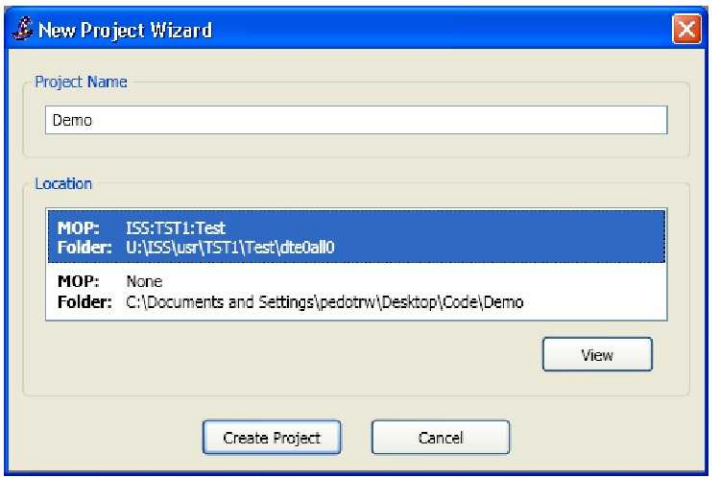


Figure 3. ERS New Project Wizard

In programming terms, an ERS script is just a custom .NET class that extends a base class called "ScriptEngine". This base class is defined in the ERS library and contains many useful EPC-related methods. These methods will appear in Visual Studio's Intellisense, Fig. 4, whenever the user types code. Included in the Intellisense are descriptions for each method. A list and description of some of the ERS methods available for use within ERS script classes are found in Table 2. Notice the similarities between these methods and the directives in Table 1 for the SLP.

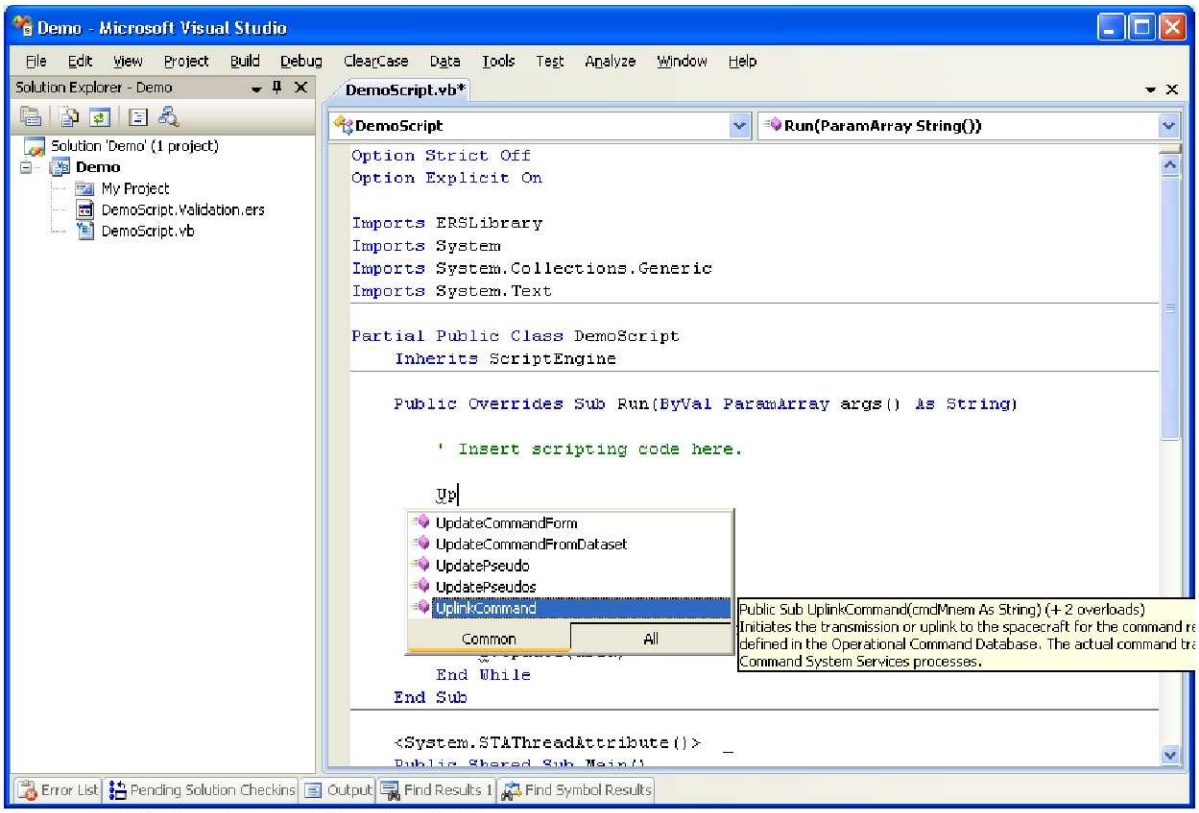


Figure 4. ERS Development Showing Intelescense

Table 2. A Sample of ERS Methods

Ask(variable)	Prompts the operator for input and waits until a resume directive is entered. The value entered is stored in variable. If variable is not specified, the process is halted until the user enters a resume directive.
Ask<(Of <(T>)>)(variable)	Prompts the operator for input and waits until a resume directive is entered. The value entered is stored in variable. If variable is not specified, the process is halted until the user enters a resume directive.
AskPulldown(String, array<String>[]()[])	Create a dialog box that will prompt the operator to select a single answer to a prompt by clicking a selection from a list of text items in a pulldown list.
AskPushButton(String, array<String>[]()[])	Creates a dialog box that will prompt the operator to select a single answer to a prompt by pressing a pushbutton with the specified text.
LoggingOptions	Contains various options for controlling logging behavior. These options must be set prior to calling any logging methods. Upon calling the first logging method, these options become read-only.
SampleLatest()()()	Updates all MSIDs with their latest packet values.
SampleMSID<(Of <(T>)>)(String, Processing, Boolean, Boolean)	Initializes a MSID object from which values and statuses will be sampled.
SampleNext()()()	Updates all MSIDs with their next packet values.
StartDisplay(String)	Starts the identified display within an instance of the Display Operations application.
UpdateCommandForm(String)	Shows the update form for the specified command.
UplinkCommand(String, Verify)	Initiates the transmission or uplink to the spacecraft for the command referenced by a unique mnemonic as defined in the Operational Command Database.

Wait(Int32)	Suspends execution until the time has elapsed and then resumes execution with the next statement in the script.
Write(Object, array<Object>[]()[])	Constructs a string of text and displays it on the operator's screen. Expressions may consist of user defined variables, intrinsic functions, and quoted strings. All expressions will be formatted in ASCII. The message will NOT be sent to Message Handler.
WriteFormat(String, array<Object>[]()[])	Logs a string using the composite formatting feature of the .NET Framework.

More detailed help is also available for each method by pressing the "F1" key while in the Visual Studio IDE. This brings up the full set of ERS documentation, Fig. 5, which contains all the methods and objects that can be used in an ERS project. ERS Help is also searchable by keywords.

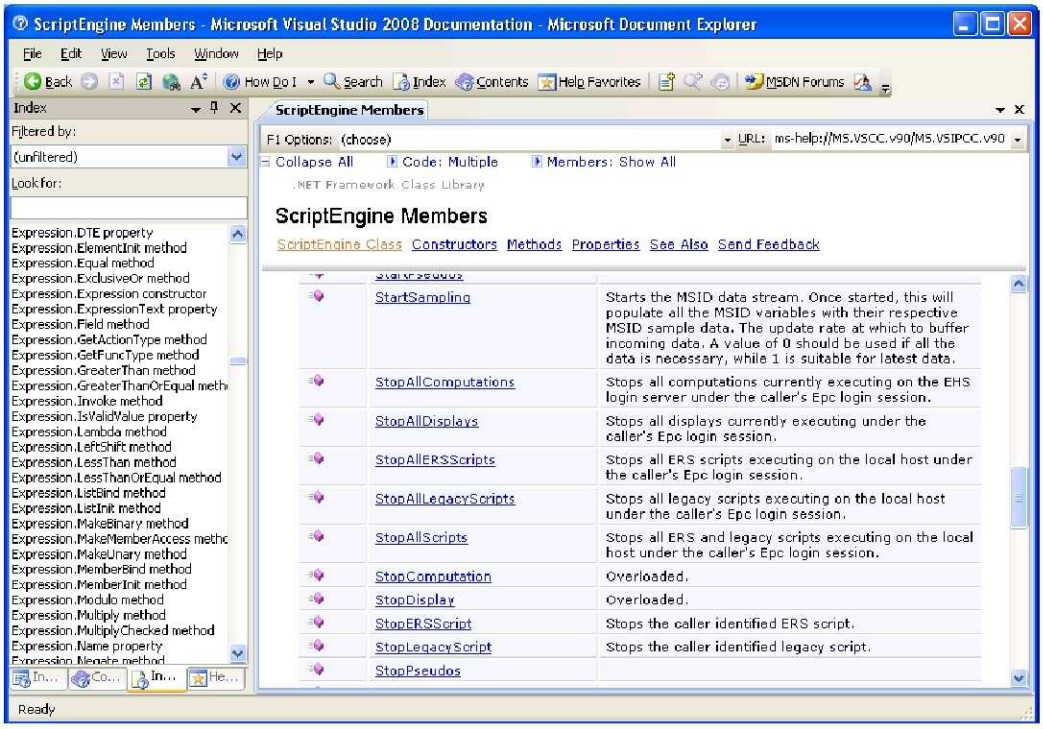


Figure 5. ERS Detailed Help

B. Measurements and Commands as Script Variables

To properly use many of the methods or procedures previously presented, individual telemetry mnemonics or Measurement Stimulus Identifiers (MSIDs) and Command Mnemonics must be assigned to variables in the program. The validation file (Fig. 4: "DemoScript.Validation.ers") that was generated at project creation time is used to map these objects into an ERS project. Double-clicking on the validation file will display the validation dialog shown in Fig. 6. Here the user can define the telemetry measurements and commands to include in the code project.

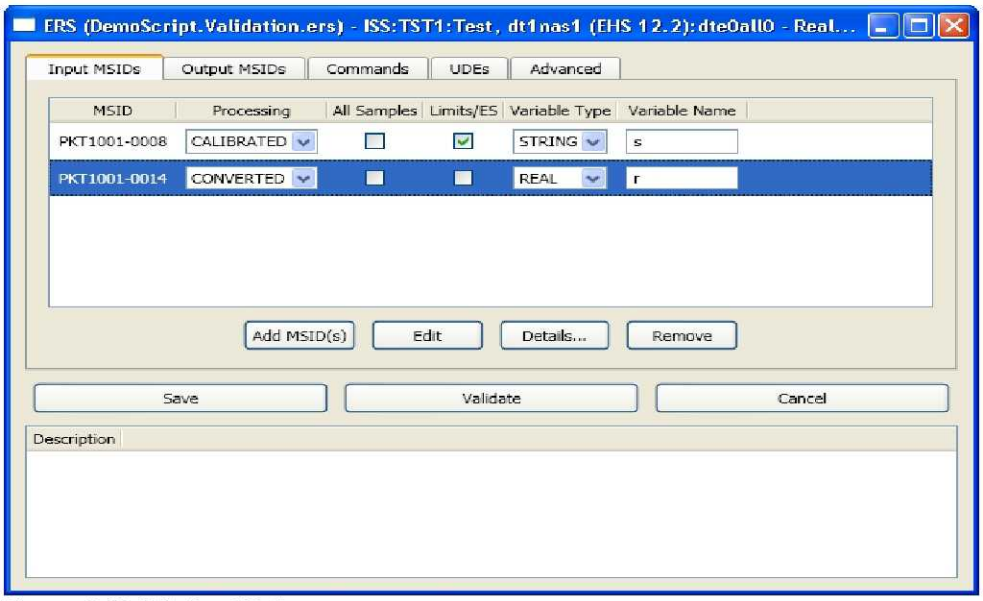


Figure 6. Validation Dialog

The validation dialog (Fig. 6) maps an EPC object (i.e., a measurement, command, etc.) to a variable name. The variable can then be addressed by the user in the code file. The variable itself exposes properties and methods that make sense for that particular EPC object type. For example, a measurement variable will have properties that return a sampling status or value. In the code segment below, the "r" measurement variable is being inspected for new data.

```
If STRING.IsDataNew Then
    Write("New data is available!")
End If
```

A search dialog is used to select the telemetry measurement for variable assignment (Fig. 7). Measurements can be searched on several criteria, including whether or not t he measurements have pre-defined limits or expected states.

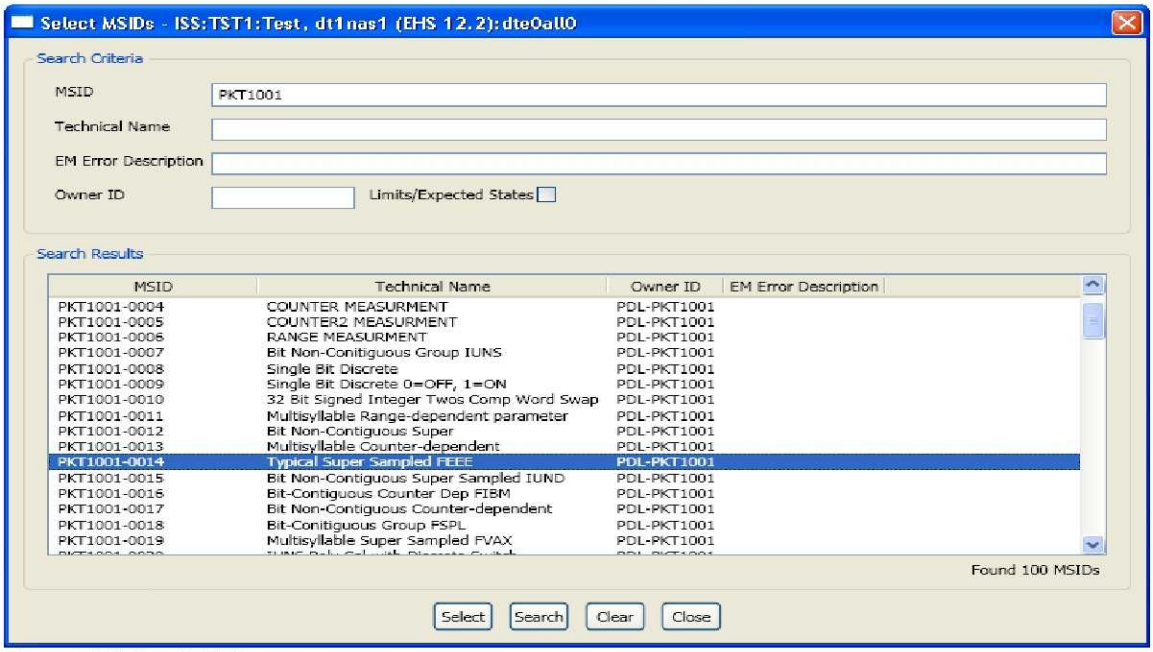


Figure 7. Select MSIDs

From the measurement search dialog, the details of each displayed measurement can be examined, Fig. 8.

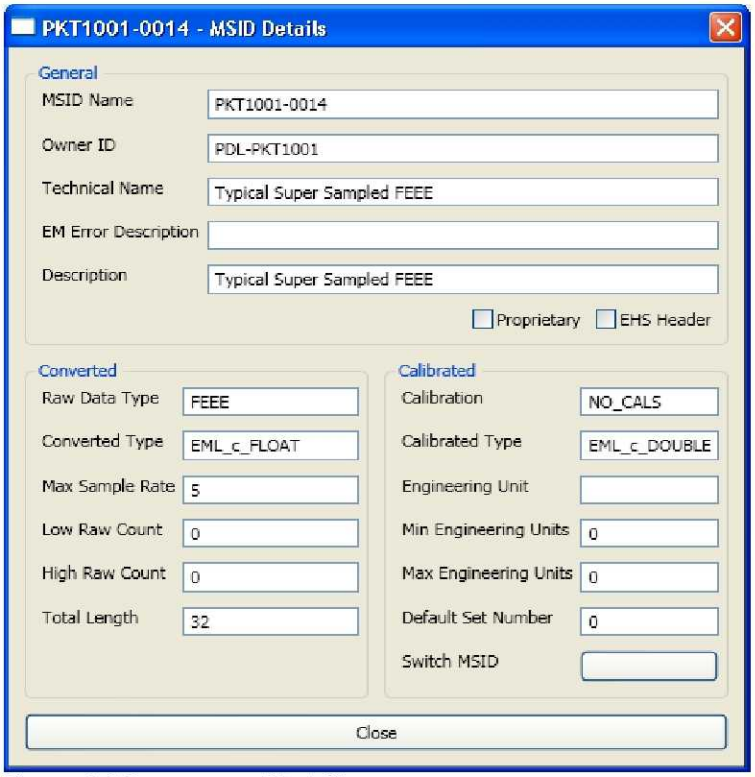


Figure 8. Measurement Detail

Commands can also be added to an ERS project via the command tab back on the Validation Dialog, Fig. 6. Like MSIDs, the commands available are searchable by name (Fig. 9). From the Select Commands dialog, the details of each command can be examined in a window similar to Fig. 8.

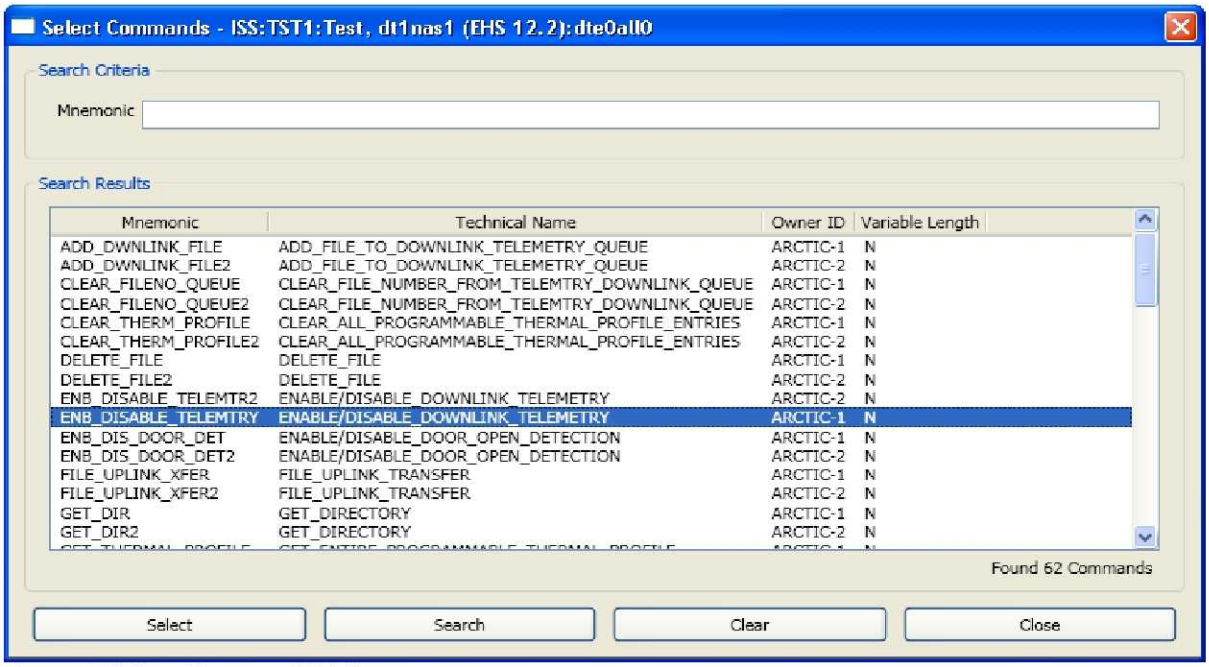


Figure 9. Select Command Dialog

C. Script Control

ERS scripts can be controlled at creation time in debug mode inside the Visual Studio debugger where the script developer is provided with a rich set of the latest source level debug tools provided by Microsoft.

ERS provides script control for a verified ERS script at run-time through ERS provided input/output methods. These include methods to write out messages and to prompt a user for input. An ERS script that performs input and output is assigned its own logging window where both the input prompts and output messages are displayed. Figure 10 is an example of an ERS script prompting the user for a number, which it then uses to output a measurement sample that number of times.

Note that both the input and output of an ERS script are tagged with a time stamp. This is because, by default, all input/output generated by a script is also logged to a standard file associated with that script. The log files can be accessed from the logging window itself or via the ERS Operations application for post flight analysis.

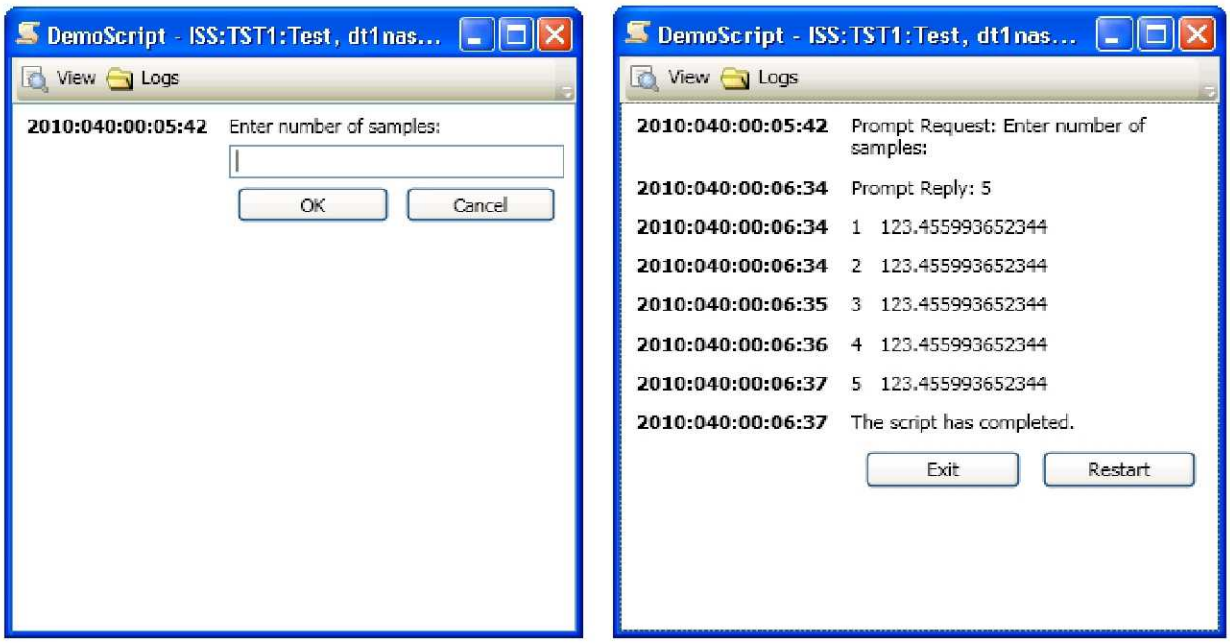


Figure 10. ERS Input and Output Windows

IV. Conclusion

Scripts for flight operations control have proven to be very useful. They offer a relatively simple syntax that does not require specialized training and an environment that is integrated well with the host ground system. Scripts also offer flow-control functions that make them ideal for remote control including execution confirmation. Other than the very newest Web-oriented scripting languages, most are graphically challenged and maintaining the typical program constructs that are not specific to ground systems functions becomes time consuming and even unnecessary.

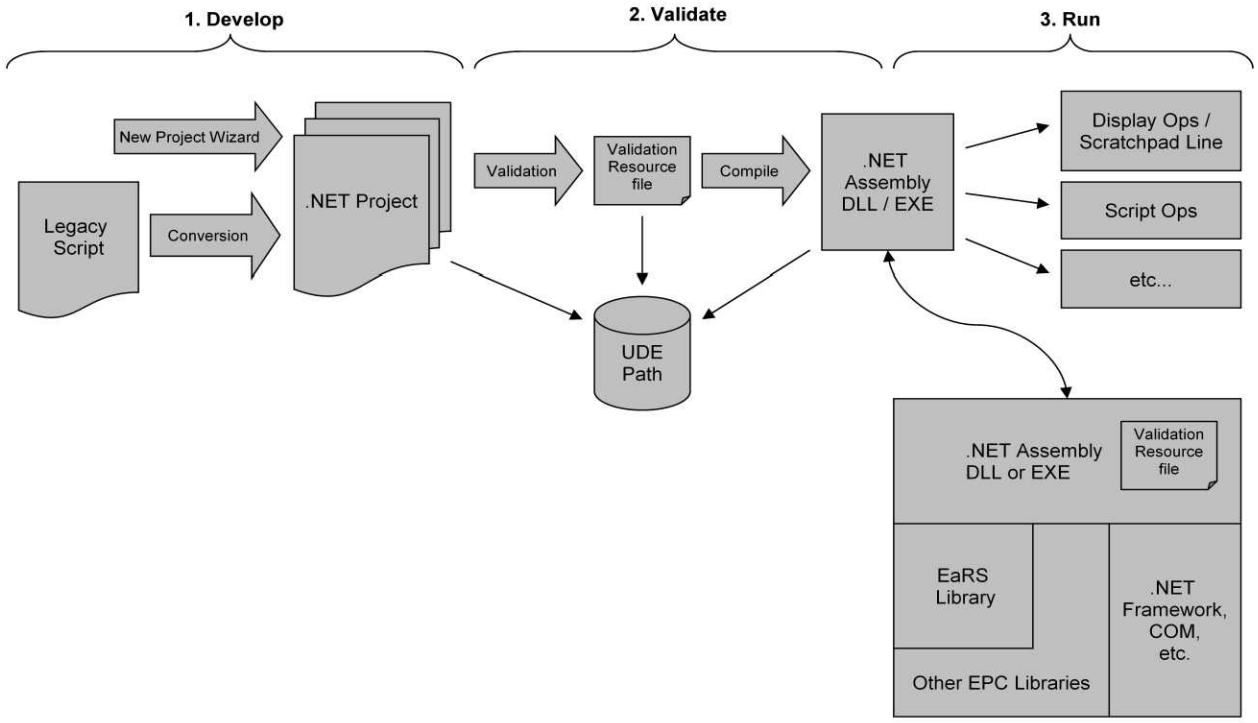
ERS makes use of Microsoft's Visual Basic (and C#) language which is also simple to understand and easy to learn. ERS adds "Wizards" to allow program access to the ground systems telemetry and command objects. Using a standard programming language removes the need to maintain basic programming syntax in a script interpreter and frees us to concentrate on new ways to interface to our base system. ERS also provides a powerful input/output flow-control feature that allows for the user to monitor progress and for confirmation execution of critical tasks if necessary, while also recording all actions in textual log files.

ERS programs are not scripts. But ERS programs offer the same features as many scripting languages, such as the HOSC's SLP, while also including the ability to include rich graphical components. ERS is an Enhanced and Redesigned Scripting environment that is lowering our scripting development and maintenance costs while increasing our ground systems automation capability.

Appendix

ERS Development Architecture

The development environment for ERS consists of a computer with both EPC (version 6.5 or higher) and Visual Studio 2008 (or later) installed. As part of the EPC installation, Visual Studio is extended to include an ERS project wizard and a new editor for an ERS-specific file type. Once installed, creating ERS "scripts" consists of three distinct stages discussed below.



1. Development Stage

A new ERS project is created within Visual Studio via a custom ERS project wizard. This creates a .NET project consisting of code files, written in either C# or VB.NET, plus an ERS-specific file used to map EPC's telemetry and commanding objects into the project. The code includes methods defined in the ERS Library to do Enhanced HOSC System (EHS)-related directives, as well as any other user code that is allowed by the .NET languages. All the Visual Studio IDE features such as IntelliSense, help, and debugging are now part of the scripting development experience.

In programming terms, an ERS "script" is just a .NET class that extends a base ERS class called "ScriptEngine". This base ERS class contains EHS-related methods. In addition to using the EHS-related methods, the user can work with EPC telemetry measurements and commands. These are represented as .NET objects with various methods and properties that the user can code against. Telemetry and commanding objects are introduced into an ERS project by way of a custom editor provided by ERS. The custom editor is associated with a "validation" file that is part of the ERS project. From this editor, the user can select various telemetry measurements and commands. Once selected, they become mapped to a variable and, thereafter, are accessible by code.

2. Validation Stage

Building a scripting .NET project is similar to building a regular .NET project, except the script project has an additional custom build step that handles validation. Building a project and validating a project go hand-in-hand. Validation involves converting the validation file of an ERS project into proper set of .NET variables, by generating "behind the scenes" code. This generated code is then compiled together with the user code. The final output of this build is a .NET assembly .dll. This assembly contains the ERS validation information, "script" code, and any other code and/or resources the user included into the project.

A script project can be validated in one of two ways. The first way, as mentioned above, is by building the ERS project inside of the Visual Studio IDE. Alternatively, an EPC application called Bulk Validation, can be used to build one or more ERS projects.

3. Run Stage

A .NET assembly containing ERS scripting code can be run via the EPC Scratchpad Line, a scratchpad line in Display Operations (i.e. button) or via Script Operations. Additionally, because it in a .dll form, ERS scripting code can be used as a method by other libraries or executables. Alternatively, an ERS project can be converted into an EXE project type and run as a standalone application.

Regardless of how it is started, before any ERS -related methods are executed the ERS code must initialize itself with an EPC session. This initialization process verifies that the ERS code was validated correctly against the same database version in the EPC session. Referenced telemetry measurements and commands are further checked to make sure they exist in the specified EPC session.

Script Syntax Compared

Sample **Unix Shell Script** that accepts one argument:

```
if [ $# = 0 ]; then
echo Error! Parameter required.
exit 1
fi
if [ "$1" = "Dog" ]; then
echo Bow Wow!
elif [ "$1" = "Cat" ]; then
echo Meow!
else
echo Error!: Invalid Parameter
fi
```

Scripts that update a command's bandwidth field and then uplinks the command:

SLP Script	ERS "Script"
begin_script script_name *declarations* *system_section* *global_section* *local_section* *end_declarations* *update command SET_TEMP_BANDWIDTH fields* *BANDWIDTH = 10* *endupdate* *uplink command SET_TEMP_BANDWIDTH* *end_script*	*With SetTempBandwidth.Fields* *.Bandwidth = 10* *End With* *SetTempBandwidth.Update()* *SetTempBandwidth.Uplink()* Or *SetTempBandwidth.Update(10)* *SetTempBandwidth.Uplink()*

References

[1] SRA International, http://www.sra.com/scl/, 2010.

[2] TSTOL Manual, The Systems Test and Operations Language, Goddard Spaceflight Center, 1993.

[3]Scripting Language and Scratchpad Line Detailed Specification HOSC-EHS-2057. March 2000