Scripting Language Extensions

Jeffery McDougle, Thomas Blocker, and David Prestwich
Schweitzer Engineering Laboratories, Inc.

Presented at the
13th Annual Western Power Delivery Automation Conference
Spokane, Washington
March 29–31, 2011

Scripting Language Extensions

Jeffery McDougle, Thomas Blocker, and David Prestwich, *Schweitzer Engineering Laboratories, Inc.*

Abstract—**Proprietary software is often used to collect data, including fault records and event reports, sequential events records, meter information, settings, and diagnostics, from microprocessor-based devices. This paper discusses techniques to connect, collect, and parse ASCII-based responses from multiple-vendor devices.**

I. Introduction

Electric utility engineers often need to collect more data than an individual software vendor can provide. The primary goal for creating and working with these data is to place configurations, events, and device application information in a central location. Organizing these elements helps to eliminate mistakes and increase productivity throughout an organization. Centralizing this information provides users with the highest level of flexibility. Types of information that can be collected include connection information, maintenance schedules, audit data, and security requirements and levels. Automating the data collection allows for periodic maintenance checks that help customers meet the ever-growing North American Electric Reliability Corporation (NERC) demands, while eliminating the need for new equipment and costly maintenance tasks. Understanding and implementing data collection using customized scripts gives users the highest level of flexibility and better compliance to NERC PRC-002 and PRC-005 as well as CIP-002 through CIP-009.

This paper evaluates the importance of flexible scripting that allows users to retrieve information by adding new commands or reports to the device, without being dependent on or delayed by revisions to proprietary device-specific collection software. Project development and maintenance are far more efficient when report-gathering software includes capabilities to tailor the collection to specific needs using scripting. This allows users to easily build and test their own access scripts, user-based command scripts, and external parsers in order to perform the following tasks:

- Connect to a device.
- Navigate the device to a desired state via ASCII dialog.
- Retrieve data from the device via ASCII capture or a standard file transfer format.
- Save the results of the capture or file transfer for subsequent viewing and analysis.
- Execute an external script or program, automatically passing the name of the file to that script or executable file.
- Send emails and data logs, or create and schedule follow-up jobs via these external scripts or programs.

Collection software should have the capability to poll data at a preconfigured interval. This provides the ability to start gathering important information from a device without having to wait for manufacturers to update software. Plus, scripts can be optimized to collect only the information that is required. This improves the overhead of manually processing data for the pertinent information that will be collected.

II. SCADA Systems and Engineering Access

Supervisory control and data acquisition (SCADA) systems have become widely accepted and have grown into feature-rich applications, allowing for supervision and control. From the first-generation monolithic systems to modern networked installations, SCADA continues to evolve into various industry standards as well as a wide variety of supported protocols.

Increased requirements lead engineers to evaluate what classes of information are appropriate for SCADA communications links. Engineers must answer the following questions: Do we have the capacity to support full event reports sent across the system? Should operators be sent even more information on what is critical or normal operation? How do we allow engineering units access to these data while continuing to secure communication?

Security issues have also come to the forefront of modern industry discussions. Connections and data volumes between SCADA systems and wide-area network/local-area network (WAN/LAN) infrastructures have increased. Internet cybersecurity threats are forcing companies to make choices, such as removing outside access. Eliminating the ability to gain outside access limits the amount of oscillographic, Sequential Events Recorder (SER), and maintenance data that can be gathered. These limits can increase costs and threaten the progress made over the past decade in developing and maintaining settings configured for installations. In general, SCADA misconceptions include the following:

- No one attempts to exploit the vulnerability because SCADA is not widely known.
- SCADA is already physically secure.
- SCADA lacks authentication.
- SCADA is immune to attacks because it is disconnected from the Internet.
- It costs more to install and maintain a larger network infrastructure.

Similar to the evolution of SCADA, engineering access to devices and remote locations has changed over the years. What once required hours of driving to and from site installations can now be done from the comfort of an engineering workstation. Modems have been replaced by virtual private networks (VPNs) and high-speed communication. Printouts of settings or events are sent using email communication. Improved reliability in substation-ready computers allows for gigabytes of data to be stored, collected, and evaluated across an organization.

As was the case with SCADA, increased features introduce more questions that need to be answered for modern substation designs (an example design is shown in Fig. 1). To what location does this user have access? Who connected to this location last? What was changed? Did the user collect the correct information? How do we store all of these data over long periods of time?

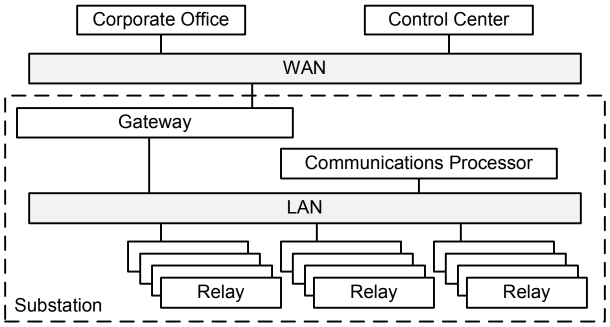


Fig. 1. Example of a modern substation design

In general, engineering access concerns include the following:

- Identity-based access controls and audits for who or what can connect to the device and from what location.
- Length of time and cost to collect required data from remote locations.
- Amount of training required to implement solutions.
- Human interaction, which can be prone to error and costly mistakes.
- Scalable and manageable solutions.

III. SCRIPTING

Scripting is a programming language that allows control of one or more software applications or processes. Scripting languages came about largely because of the development of the Internet as a communications tool. JavaScript, Active Server Pages (ASP), JavaServer Pages (JSP), Hypertext Processor (PHP), Perl, Tool Command Language (Tcl) and Python are examples of scripting languages [1]. Additionally, some large application programs include an idiomatic scripting language tailored to the needs of the application user. An application-specific scripting language can be viewed as a domain-specific programming language specialized to a single application.

While traditional scripting languages may have new terminology and feature sets, the concept of taking data, defining a desired result, and acting on those data is not new to the industry. Engineers have used similar if-then logic within intelligent electronic devices (IEDs), programmable logic controllers (PLCs), and human-machine interfaces (HMIs).

A. Scripting Language Extensions

With scripts, the core application can be extended to perform tasks other than its primary functionality, as shown in Fig. 2.

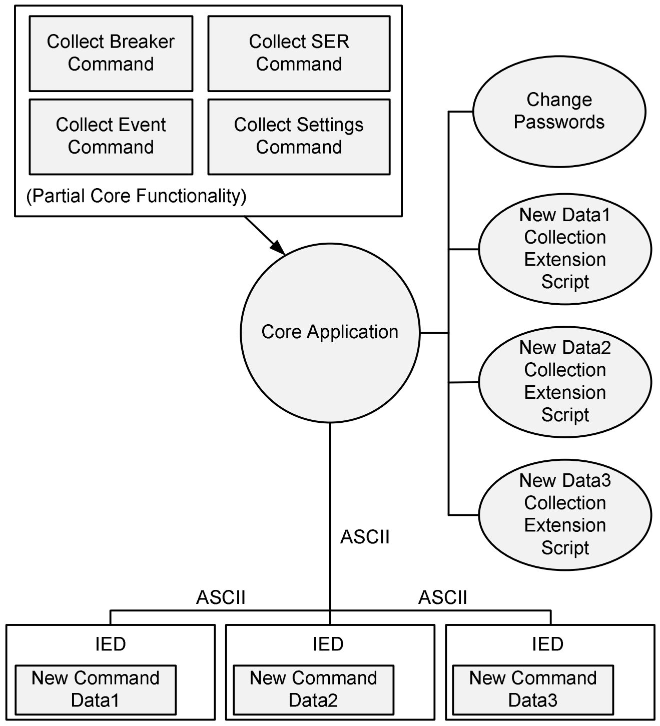


Fig. 2. Core application can be extended to take action or collect newly added command data from IEDs through extension scripts

In a typical application, a design is implemented to meet the requirements at the time of development. Often, after a program is delivered, the user wants added functionality, or different users require custom functionality based on their specific needs. In order to accommodate these situations without requiring a complete rewrite or causing a develop/compile/test/ship scenario, a framework needs to be implemented to allow for future additions of modules without breaking the existing code base.

As shown in the Fig. 2 example, the primary functionality of the core application is the ability to collect breaker command data, SER data, event report data, and settings from an IED. When new firmware releases for these IEDs, it often includes new functionality. New functionality often brings new commands to read data. The obvious solution for the supporting software is to release updates to the software alongside the IED firmware releases, so software and firmware are matched and supported. This is fine for most IEDs with propriety software, but maintaining and upgrading multiple-vendor software can be a daunting task. An alternative solution is to create infrastructure that includes communications applications that can be extended in such a way that any and all data can be collected from a particular IED. Once the data are collected, the same core application can then execute a customized script extension that completes the work.

B. *Building More Powerful Programs*

Being able to extend a program adds flexibility and makes the program "programmable." The scripting interface allows users to easily modify the behavior of the program without modifying the original contents. The benefits of this are numerous. In fact, think of the large software packages that people use every day. Nearly all of them include special macro languages, configuration languages, or even scripting engines to allow users to make customizations.

C. *Extending the Extensions*

With scripting, a system can be easily extended to perform additional functions. For example, a script that collects data could invoke a script to parse the data and another script to send an email notification. See Fig. 3.

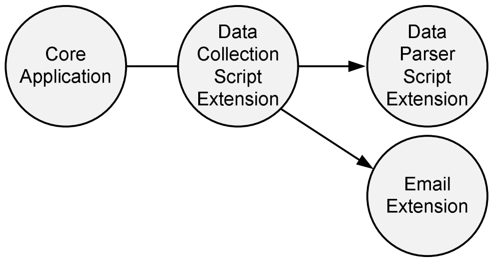

Core Application
Data Collection Script Extension
Data Parser Script Extension
Email Extension

Fig. 3. Extend the data collection system to perform additional tasks

IV. Automating the Data Collection System

This section outlines the solutions to problems that can be solved one time, up front, and in a generic way. This eliminates many difficult problems that would otherwise be left to the person creating the scripts to communicate with the devices. These solutions can be encapsulated within a standalone application. This application can be implemented as a standalone service (e.g., Windows® Service or Linux® Daemon) and provide the following solutions:

- Automatic script invocation. The application can be capable of invoking individual scripts and providing support services.
- Periodic scheduling. The application can provide scheduling services that invoke the scripts based upon a predefined schedule (e.g., once per month). The script author need not be concerned with how or when the script is executed. Scheduling becomes a separate concern.
- Triggered scheduling. The application can provide scheduling services that invoke the script or scripts based on the occurrence of some external triggering event (e.g., system fault). The script author need not be concerned with this triggering but only deals with data written to and read from the connected device.
- Record keeping. The application can assume the responsibilities of maintaining the recorded history of the script executions. Thus the application records when the script is executed and any status information relating to that execution. The script author need not be concerned with where or how this information is recorded.
- Confirmation of successful execution. The application can assume the responsibility of retrying the script if system errors cause a script failure. The script author need not be concerned with retries. The application tries until it succeeds. The script author simply trusts that this occurs.
- Information passing. The application can pass information that the script requires via command line arguments or an equivalent method. Thus it becomes easier for the script author to discover what device is connected and other information relating to the work to be performed. All external information needed by the script is provided by the application, or the script can be given the information needed to find that external information on its own.
- Communication. The application can establish communication with devices using an external connection directory and become a communications tunnel between the user-authored script and the connected devices. The application can manage the communication when one or more pass-through devices are involved (e.g., communications processors). The application can communicate using virtually any protocol, such as Transmission Control Protocol/Internet Protocol (TCP/IP) and serial. However, the script author need not be concerned with the communications protocols, how the connections are made, or how many devices are involved. The script author is only concerned with the content of those communications and the responses to data sent by the connected devices. The accidental complexities of these communications are removed from the script author's focus, as are the complexities of simultaneous device communications.
- Device authentication. The application can provide authentication services for the scripts, thereby eliminating the need for usernames and passwords to appear in the scripts. The script author need not know the usernames or passwords.
- Code signing. The script can be digitally signed and that signature can be permanently recorded. Before the application executes the script, it can test the script against the saved signature to ensure the script has not been modified or replaced. Thus the application only executes approved scripts, further enhancing system security.

- Auxiliary services. The application can provide the following auxiliary services:
  - Error reporting. The application can encapsulate the infrastructure needed to report errors or other interesting events occurring during the script execution. The script author need not be concerned with how or where errors or other events are recorded.
  - User notification. The application can encapsulate the infrastructure needed to send email or perform other messaging services, providing immediate notification of device data or status to those who have a vested interest in the data retrieved from the device. For instance, if a script determines that a breaker needs service, the script can instruct the application to email a notification to the correct individuals.
  - File transfer. The application can provide file transfer services, such as YMODEM or File Transfer Protocol (FTP), for the script, greatly simplifying the process for the script author. The script author need not know the details of performing these operations. The script simply instructs the application to carry out the operation, and the application notifies the script when the operation is finished.
  - Other. Virtually any operation that is not specific to a single script can be encapsulated in the application, providing the script author a simple application programming interface (API) to perform these operations.

V. Example Scripts and Practical Uses

The industry has made advances in the automatic collection of data from various media. Collection can be completed through scheduled tasks or on demand. With the data collected, engineers can create and generate customized scripts to reduce time and effort during the development of troubleshooting tools.

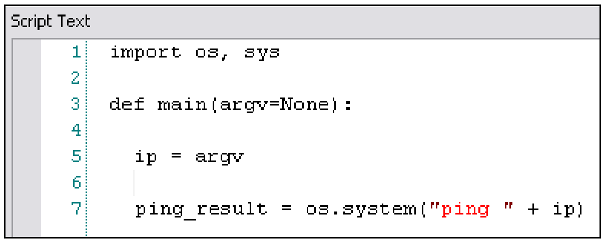
NERC PRC standards have introduced new functional requirements, dealing with everything from cyberassets and physical security to the full implementation of recovery plans. Using scripting languages within a system allows users to better automate many of the requirements that directly impact day-to-day activities. In particular, automated scripting could be applied to better meet CIP and PRC-005-2 requirements.

A. CIP-003: Cyber Security – Security Management Controls

CIP-003 requires the establishment of change control and configuration management for all critical cyberassets. This change control includes any change to hardware or software components of the critical cyberasset.

Once the device has been identified as a critical cyberasset, scripting techniques can be used to continually monitor settings and values without the introduction of human interaction, which can lead to errors or poor interpretations. Examples of applications include the following:

- Validation of settings. Devices have the ability to display settings in a wide variety of ways. Settings can collect and store in a delaminated format. Quickly parse and compare these settings with a set of baseline values using standard scripting environments.
- Status verification. Scripts can be created for validation of status. These scripts can be as simple as verifying a ping response (Fig. 4) or connecting to a device and verifying the device identifier (Fig. 5).



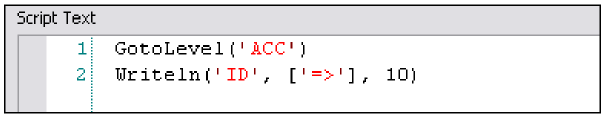
```
import os, sys

def main(argv=None):

  ip = argv

  ping_result = os.system("ping " + ip)
```

Fig. 4. Example of a script that pings an IP address and receives a result



```
GotoLevel('ACC')
Writeln('ID', ['=>'], 10)
```

Fig. 5. Example of a script that connects to a device and issues the ID command

B. PRC-005-2: Protection System Maintenance

The main purpose of PRC-005-2 is to ensure the maintenance of all transmission and generation protection systems affecting the reliability of the bulk electric system. The bulk electric system includes protection systems, underfrequency and undervoltage load-shedding systems, and special protection systems, such as remedial action schemes.

As is commonly known but important to review, the definition of protection systems includes the following components:

- Protective relays, including internal diagnostics, analog-to-digital (A/D) converters, relay trip outputs, settings, and firmware verification.
- Associated communications systems necessary for correct operation of protective devices.
- Voltage and current sensing inputs to protective relays.
- DC control circuitry.
- Station dc supply and batteries.

As stated by PRC-005-02, each transmission owner and generator owner shall establish a protection system maintenance program (PSMP) that shall include the following:

- Take inventory of each critical protection system component.
- Identify whether each protection system component is addressed through time-based, conditioned-based, performance-based, or a combination of these maintenance methods, and identify the associated maintenance interval.
- Have a documented maintenance program with procedures, test intervals, and records showing that maintenance was performed.

- Combining a device ASCII interface with a common scripting language allows a user to create any number of batch scripts to be run on demand or during a set interval for verification of these PSMP procedures. Examples of items to verify include the following:
- IED internal diagnostics
- IED A/D converter
- Current transformer (CT) and potential transformer (PT) sensing devices or circuit inputs
- IED trip outputs
- IED peer-to-peer communication
- Firmware revisions
- IED settings

C. Collecting and Prioritizing SER Records

Traditionally, SER records provide data relevant to reclose intervals and carrier issues. Today, however, SER records can contain security alarm information, manipulate settings, and control data when using more modern communications processors. With these valuable data, scripting tools allow users to not only collect the data but to parse, categorize, and make quick decisions and/or recommendations.

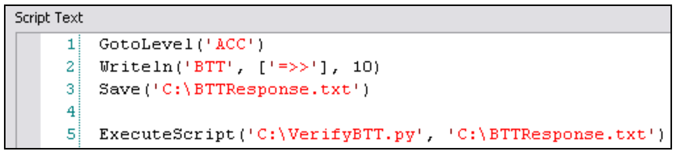
D. Battery Status Command Verification With Email Notification

Processor-based devices send battery status commands (BTT) that generate an output, as shown in Fig. 6.

```
=+>BTT<ENTER>
Battery Charger Board FAILED
=+>
```

Fig. 6. Example of BTT command output

When the BTT command is issued, the output response can be captured and saved to a file, as shown in Fig. 7.



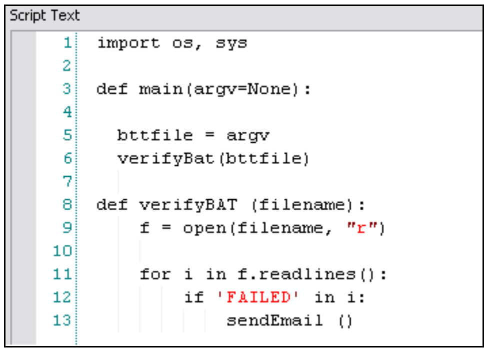
Script Text

```
GotoLevel('ACC')
Writeln('BTT', ['=>>'], 10)
Save('C:\BTTResponse.txt')

ExecuteScript('C:\VerifyBTT.py', 'C:\BTTResponse.txt')
```

Fig. 7. Example of a script that issues a BTT command, captures the response, and calls another script to verify the data captured

The output can be parsed, run through logic by comparing with desired limits, and emailed in reports to state, for example, that a battery level is becoming too low. Fig. 8 shows an example of a script where the BTT response was captured in a file and then opened and examined to look for a failed condition. If the failed condition is detected, an email notification is sent.



Script Text

```
import os, sys

def main(argv=None):

  bttfile = argv
  verifyBat(bttfile)

def verifyBAT (filename):
    f = open(filename, "r")

    for i in f.readlines():
        if 'FAILED' in i:
            sendEmail ()
```

Fig. 8. Example of a script that analyzes the BTT command response and sends an email if a battery has failed

VI. Conclusion

Scripting extends core applications to allow the collection of new data or the control of devices. Script extensions remove the need for software application updates, which also removes the burden of installation from the user. Over time, the core application evolves to consume scripting extensions as they become more stable and complex, so the operator does not have to maintain the scripts. This improves speed and efficiency throughout the application. Another advantage of scripting extensions is the ability to tailor the scripts in such a way that only high-priority data items are collected, analyzed, and acted upon, which helps satisfy NERC PRC and CIP requirements.

VII. Reference

[1] L. Greiner, "PHP, JavaScript, Ruby, Perl, Python, and Tcl Today: The State of the Scripting Universe," *CIO Newsletter*, August 2008. Available: http://www.cio.com.

VIII. Biographies

Jeffery McDougle received a 2-year electronics degree from ITT in 1995 and a BS in Computer Engineering from Washington State University in 2002. He joined Schweitzer Engineering Laboratories, Inc. in 1995, where he has been involved in the development of various software and firmware products.

Thomas Blocker received his BS in Computer Information Systems from Eastern Washington University in 2000. He joined Schweitzer Engineering Laboratories, Inc. in 2005, where he has been involved in the design and development of various Windows applications. He is also a 15-year U.S. Navy veteran with more than 10 years of experience operating ship power plants.

David Prestwich received his BS in Mathematics from the University of Idaho in 1999. He has been a product engineer and development manager for the PC software group at Schweitzer Engineering Laboratories, Inc. (SEL) for the past 5 years and has led several projects touching a wide variety of SEL products. Prior to joining SEL, he worked as an automation engineer implementing control systems around the world.