Test Scripting – An Alternative Approach to Real-Time Instrument Control

Yönet A. Eracar, Ph.D.
Teradyne Inc.
N. Reading, MA, USA

Michael F. McGoldrick
Teradyne Inc.
N. Reading, MA, USA

Abstract— **In many test applications involving high speed digital data buses, the time required to receive data from a Unit Under Test (UUT), process that data, and provide a response to the UUT may be small enough to preclude use of the computer controlling the overall test system, even if the nature of that data processing is not very complex. In these situations, Test Program Set (TPS) developers can use local computing power that is embedded in a particular test instrument. However, developing test programs to run on this embedded computing power can be difficult and involve purchasing additional software tools beyond what is provided with the test system. In addition, the embedded computer program does not run independently, but rather in tandem with and under the control of, the test program running on the test system computer. So, in addition to developing the algorithms to receive, process, and transmit data, the TPS developer must also provide the framework by which the overall test program can control these operations.**

The solution proposed in this paper helps simplify the TPS development effort by; 1) providing a simple scripting language that TPS developers can use to create test scripts that execute on the test instrumentation, 2) providing them with tools to code and compile those test scripts, 3) providing the necessary framework for them to allow the overall test program to control the operation of the test scripts, and relieving the TPS developers of that burden, and 4) providing an alternative to them having to purchase expensive, real-time programming tools.

This paper identifies test applications where test scripting may be used effectively in reducing overall TPS development time, lists basic features of a test scripting language, describes the capabilities needed of the accompanying tool set, and how the test scripts would be developed and integrated into a TPS. The paper also includes a practical example of a test script used to dynamically modify data as part of a Fibre Channel based test application.

Keywords—tester latency; test instrumentation; embedded programming; scripting

I. Introduction

When testing UUT digital interfaces in functional test applications or as part of end-environment emulation, there is often a need for the test system to react to incoming events with either canned or algorithmic response data. We call the time it takes for the test system to respond to these events "tester latency". If the performance demands of UUTs increase, it is reasonable to assume that the required tester latency would correspondingly decrease. If the test system's controlling computer is responsible for processing the UUT events and providing a response, the processing time required may exceed the required timing for the UUT. As a result, test system designers often resort to providing processing capabilities on the instrumentation, avoiding the extra delays in routing data to and from the test system's controlling computer.

Writing programs that execute directly on the test instrumentation can be challenging. Such instruments often provide an operating system, possibly a real-time one, and one that is often not the same as the one running the test system's controlling computer. As a result, a different set of tools can be required to develop these so-called "embedded" programs, and the skill set of the TPS developers needs to be expanded to make use of these tools. In addition, the resulting program still needs to coordinate its activities with the test system's controlling computer, which is responsible for overall TPS execution and instrument coordination. This coordination needs to be coded into both the embedded program and the TPS, making both more difficult to develop and debug. Even without this coordination, real-time performance requirements may make developing the embedded (now a real-time) program more challenging and costly.

For some UUTs, developing such a real-time program may be unavoidable. However, in many cases, having to invest in real-time development tools and training, and the resulting extra engineering effort involved in developing those programs, may be overkill. Just as in desktop applications where scripting languages can provide simple alternatives to full-scale application development, a test scripting environment may offer an alternative approach to developing programs that execute directly on the test instrumentation. Rather than executing a test program directly on the test instrument, the test instrument would execute an interpreter program that is responsible for running the TPS developer's test script.

The rest of this paper describes the components of test scripting approach in detail. Section 2 provides an overall overview of the test scripting approach. Section 3 lists basic features of a test scripting language. Section 4 describes the capabilities needed of the accompanying tool set, and how the test scripts would be developed and integrated into a TPS. Section 5 explains a practical example of a test script used to dynamically modify data as part of a Fibre Channel based test application.

II. Overview of the Test Scripting Approach

Test scripts do not eliminate the need to develop an embedded program. Quite the opposite: test scripts execute in the context of and under the control of an embedded program. The embedded program controls the basic operation of the instrument, and provides hooks that let a TPS developer customize the behavior of that program at certain points, or test events, in its execution. The test scripts are organized as a collection of callback routines that are invoked at instances of those test events. What those test events are depend on the nature of the test application.

A. Illustrative Example

Consider as an example a test instrument in a Fibre Channel test application. Assume that in this application, the UUT has a Fibre Channel port and makes use of the FC-AE-RDMA upper level protocol to communicate externally. Providing support for the FC-AE-RDMA upper level protocol in a test instrument would most likely involve embedded processing and close interaction with the test instrument hardware to perform the appropriate physical layer signaling at the appropriate times. The effort to develop such an embedded program is significant, but, once complete, the work should be able to be leveraged across any UUT that makes use of this protocol.

However, the behavior of a UUT may require that when certain I/O operations occur, specific I/O operations particular to that UUT must ensue. These UUT specific operations must be programmed by the TPS developer. In doing so, the nature of the I/O operations does not change – they are still FC-AE-RDMA I/O operations; it is the content of those operations that varies from UUT to UUT.

Test scripts allow a TPS developer to program these UUT-specific operations, while leveraging the existing work in the embedded program in supporting the operation protocols. Test scripts are not intended to control the detailed operation of a test instrument. Instead, they are more geared to controlling the data passed to and/or making decisions based on data received from the UUT.

To make use of test scripts, the FC-AE-RDMA upper level protocol embedded program must provide the hooks to invoke those test scripts when test events occur, in addition to supporting the I/O operations themselves. The test events may include:

- Start of test
- UUT I/O operation initiated
- UUT I/O operation completed
- Test instrument I/O operation initiated
- Test instrument I/O operation completed

This does add a small amount of complexity to the embedded program, but has the benefit of making that program usable across several TPSs.

B. Test Script Execution

The test script code is stored in one or more text files. Using a tool provided by the test instrument vendor, the TPS developer compiles the contents of these files into a machine-independent binary form optimized for execution on the test instrument. The resulting binary code does not form a single executable program, but rather a collection of callback routines that are invoked on test events during the execution of a TPS.

Prior to running a test, the TPS developer uses instrument driver API calls to copy the test script binary down to the test instrument and associate specific callback routine names with test events, in addition to any of the other instrument configuration operations needed in the TPS.

During test execution, as the test instrument embedded program executes and a test event occurs, the embedded program determines if a test script routine has been associated with the test event. If so, the address of that routine in the test script binary is passed to a component called the Script Interpreter, which is responsible for executing test scripts. The Script Interpreter component is a bytecode interpreter that steps through the test script binary code executing the commands contained in that code. Based on the test application, the test instrument embedded program may then wait for the script routine to complete or simply continue execution immediately, resulting in the script routine and embedded program executing in parallel.

C. Test Script Language Selection

Despite the wide variety of scripting languages available today, we believe that a custom scripting language is best suited for this application. A custom scripting language can more easily incorporate support for test instrumentation control and TPS integration, and is not burdened with programming constructs that are not applicable to test instruments. Examples of the latter include console and perhaps file I/O since file systems are not always available on embedded computing platforms.

The next section describes a scripting language that we developed to support this type of instrument control.

III. Test Scripting Language

Test scripting language consists of the common elements that can be found in all modern programming languages: lexical elements, declarations, assignments and expressions, code constructs, procedures, and functions. In addition, test scripting language introduces special elements, called resources, to control the real-time execution of the test script and script routines that are the access points for external programs to communicate with the test script.

A. Lexical Elements

A test script source file consists of a set of tokens consisting of characters from the ASCII character set. Tokens are separated by the same white space characters that serve that purpose in a C program [1] file. There are several kinds of tokens:

- Identifiers are sequences of case-sensitive letters, digits, and underscores. A user identifier must not begin with a digit, and cannot match a reserved word. Identifiers are case-sensitive.

- Reserved words are identifiers that have a special meaning in test script code, and cannot be used as user identifiers. There are two types of reserved words: keywords and built-ins (see Table 1). Keywords are identifiers that form part of the language. Built-ins are function, procedure, and record type names that are part of the basic test script language.

Table 1: Test Scripting Language Keywords

AND	ARRAY	BOOL	BREAK
BREAKIF	BY	CHAR	COUNTER
ELSE	ELSEIF	END	ENDFOR
ENDIF	ENDWHILE	EQ	EXTERN
FALSE	FOR	FROM	FUNCTION
GE	GOTO	GT	IF
INCLUDE	INT16	INT32	INT64
LABEL	LE	LET	LT
MSGBUF	NE	NOT	OR
PROCEDURE	QUEUE	REAL	RECORD
REF	REGION	REPEAT	RESOURCES
RETURN	ROUTINE	RUN	THIS
THRU	TIMER	TRUE	UINT16
UINT32	UINT64	UINT8	UNTIL
VAR	WHILE		

- Constants take the form of *integer*, *floating point*, *character*, or *string*. Integer constants can be decimal or hex. A decimal constant is a string of digits. Hex constants are denoted by a leading 0x or 0X prefix, and consist of a string of hex digits (the letters A-F, a-f, and digits 0-9). Floating point constants follow the same rules as C floating point constants, with the exception that there is no trailing floating point suffix. Character and string constants also generally follow the C convention except there is no wide character support, and escaped characters using a numeric code must use hex values.

- Operators and Separators: Unlike in C, operations are performed using reserved words instead of special characters. For example, to add two items together you would use the built-in ADD(x,y), rather than x+y. As such, there is only a single operator symbol, the = sign, which is used in making assignments. The following are the separators supported in test script language: (,), [,], <,], ;, :, and white space.

- Comments: Test script test language supports the same block and line comment mechanisms as C.

B. *Numeric Types*

Test scripting language supports the following numeric types:

- CHAR: A signed 8-bit integer or ASCII character
- INT<N>: A signed N-bit integer, where N can be 16, 32, or 64.
- UINT<N>: An unsigned N-bit integer, where N can be 8, 16, 32, or 64.
- REAL: A double precision floating point number.
- BOOL: Boolean, with values of TRUE or FALSE, a byte in size.

It is also possible to create single-dimensioned arrays of these numeric types. Array elements are referenced in the same style as C with 0-based indices enclosed in brackets: foo[3]. Strings are not a native type, but rather an array of CHAR types.

C. *Resources*

Test scripting language makes use of special items called resources that can be used to control script execution. There are several types of resources:

- TIMER: Timers are resources that generate events (and cause code to execute in response to those events) when a certain period of time has elapsed. In addition to having a duration, a timer can operate in manual or auto mode. In manual mode, once the timer has elapsed, it generates an event and stops. The timer can be reused, but the user must explicitly restart it. In auto mode, the timer restarts automatically when it elapses.

- COUNTER: Counters are resources that generate events when a certain number of counts have occurred. Counters operate much like timers except they track actions rather than time. A counter's elapsed flag is set when the counter reaches its maximum value. The range is a 32-bit integer literal.

- QUEUE: Data queues are resources that provide a way to pass data from one script routine to another in a FIFO fashion. The user can define event handlers that execute if the queue is full when writing to it, and empty when reading from it.

- REGION: Regions are shared memory resources of a particular size. Shared memory resources provide another way of sharing data among script routines. Regions are regarded as arrays of bytes and their contents can be accessed or modified using array syntax, i.e. the name of the region followed by subscripts.

- MSGBUF: Message buffers are special shared memory regions that contain data not just controlled by script code, but also by the host PC and the real-time application. Message buffers are allocated by code running on the real-time processor. When allocated, the user associates a name with them. The name uniquely identifies a message buffer, and is the means by which one is referenced in script routines. The operations that can be performed on message buffers are the same that can be performed on regions.

Resources are global in nature. They are referenced within routines, functions, or procedures, but are defined outside of any executable code in special RESOURCES constructs. Individual resources are named so that they may be referred to in executable code.

D. *Declarations*

Identifiers must be declared before being used in executable code. Identifiers for variables, arrays, and resources are declared in declaration statements.

The scope of a declaration extends over the block in which it is located (including contained blocks). In situations where an embedded statement block includes a declaration with the same name as one declared in an outer statement block, code in the inner statement block refers to the inner statement block declaration. In effect, the inner statement block declaration hides the outer statement block declaration. Outside of the inner statement block, the outer statement block declaration is in effect.

1) *Variable Declarations*

Variables declarations start with the VAR keyword, followed by the name of the variable, a type specifier, and optionally an assignment of its initial value. All numeric variables are initialized to 0 and Boolean variables to FALSE by default.

```
// Declares an unsigned 16-bit integer named “foo”
VAR foo : UINT16;
```

2) *Resource Reference Declarations*

A resource reference declaration is used to make a reference to a globally defined resource. A resource can be used by its name or by a variable assigned to a resource with the given name.

```
// Declares a timer referenced named “timer” that
// refers to the timer resource “timer1”
VAR timer : TIMER = timer1;
```

3) *Array Declarations*

Test scripting language supports single dimensioned arrays of variables. Array declarations start with the ARRAY keyword, followed by the name of the variable, its type, and then its length. All elements of a numeric array are automatically initialized to 0, Boolean arrays to FALSE, and resource references are left uninitialized.

```
// Declares an array of 10 32-bit integers.
ARRAY fooArray : INT32 [10];
```

4) *Record Declarations*

Records are similar to structs in C. They represent a collection of related typed items. The definition of a record starts with a RECORD keyword, followed by the name of the record, and a terminating semicolon. Records are terminated with END and a terminating semicolon. Between the record start and end statements are a list of declarations for the members of the record.

```
RECORD myRecord;
     VAR field1 : INT32;
     ARRAY field2 : UINT8[32];
     VAR field3 : TIMER;
END;
```

Records are defined at global scope. Once defined, you may declare variables of that type. All fields in the record (or array of records) are set to appropriate default values (0 for numeric values, FALSE for Booleans, and unassigned for resources). Items in a record are accessed by specifying the name of the record and the name of a field within the record, separating the two with a dot.

```
VAR x : RECORD <myRecord>;
LET x.field1 = 2;
```

5) *Reference Declarations*

References are a way to perform a dynamic reinterpretation of data, for example to recast a region or message buffer as an array of records (since regions and message buffers are treated as UINT8 arrays), or a sequence of 8 bytes as a UINT64 value. References look just like regular variable and array declarations, but also include the REF keyword.

References may be used in place of declared variables and arrays in assignment statements, expressions, procedure calls, and function calls. However, before they can be used, they must be mapped to actual data. This is done using a built-in procedure, MAP_REF. Using an unmapped reference results in a run-time error. References cannot be used in definitions of RECORDs nor as formal parameters in a function or procedure definition.

E. *Assignments*

Assignment statements in test scripting language take the form:

```
LET <lvalue> = <expression>;
```

The <lvalue> indicates the item that is to be modified, and the result of the assignable expression is the new value for the item. If the type of the expression result does not match the type of the item being modified and the expression produces a numeric value, then the expression result is converted to the <lvalue> type. If a conversion is not possible, a compilation error is reported.

F. *Expressions*

There are several kinds of expressions:

- Anything that can be an <lvalue>
- Literal constants
- Another expression enclosed in parentheses
- Function calls (built-in and user-specified)
- Conditionals

Conditionals are formed from a single relational expression or multiple relational expressions that are combined using AND, OR, and NOT operators.

Relational expressions are either expressions that have a Boolean value (like a Boolean variable/array element/record entry or a function that returns a Boolean value) or use one of the relational operators to relate a pair of expressions. The relational operators are:

- EQ (is equal to)
- NE (is not equal to)
- GT (is greater than)
- GE (is greater than or equal to)
- LT (is less than)
- LE (is less than or equal to)

G. Code Constructs

In test scripting language, the executable portion of a program consists of a series of code constructs. Code constructs consist of one or more statements. Statements are terminated with semicolons. The assignment statement is an example of a simple, single-statement code construct. The set of supported code constructs include:

- Assignment statements
- If constructs (IF / ELSEIF / ELSE / ENDIF)
- While constructs (WHILE / ENDWHILE)
- Repeat constructs (REPEAT / UNTIL)
- For constructs (FOR / ENDFOR)
- GOTO statements
- Procedure calls (built-in or user defined)
- Label statements
- Return statements
- Break statements

H. Procedure and Functions

Procedures and functions provide a way for the user to segment and reuse executable code. Procedures and functions are syntactically similar. The difference between them lies in the fact that functions return values and procedures do not. Thus calls to procedures are stand-alone statements, while calls to functions are elements of an expression.

A procedure definition has the following structure:

```
PROCEDURE <name> (<parameter declaration>, …);
     <statement-block>
END;
```

A function definition has a similar structure:

```
FUNCTION <name> (<parameter declaration>, …) :
<return-type>;
     <statement-block>
END;
```

Procedures and functions can have 0 or more parameters. Each parameter must be given a name, by which it is referred in the body of the procedure, and a type. The parameters can be of any non-array type, including resource types and records, but not references. All parameters are passed by value.

The RETURN keyword signals the end of execution for a procedure, while the RETURN keyword followed by an expression does the same for a function.

I. Script Routines

Script routines are special procedures that are invoked by the test instrument's embedded program in response to scriptable events. There are two such kinds of events:

- Resource events
- Test events

Resource events include:

- Timer expired
- Counter elapsed
- Data queue full
- Data queue empty

Test events occur under the control of the particular embedded test program. Since test events directly invoke script routines and those routines have associated data, there needs to be some sort of contract between the embedded test program and the test script to allow the embedded program to pass the data in the script routine call. This contract is in the form of a RECORD.

The general form of a script routine is:

```
ROUTINE <<record-name>> <name>;
     <statement-block>
END;
```

The fields of the associated record are accessed via the keyword THIS.

The script routine RECORD type contains application specific information is not a part of the language core but rather of an application framework. An application framework consists of the test events that can invoke script routines, the definitions of the RECORDs for those script routines, and a list of built-in functions and procedures that let a script routine initiate operations on the test instrument. Application frameworks are in essence application-specific extensions to the test scripting language.

J. Built-in Library of Procedures and Functions

The built-in library contains a variety of procedures and functions to perform common processing, including basic arithmetic operations (see Table 2 for a complete list). Built-ins also are highly flexible in the number and type of parameters they support. For example, the ADD function can take any number of numeric parameters, and can even take array parameters, something that is not supported in user defined functions and procedures.

Table 2: Built-In Library Members

Built-in Function Group	**Group Members**
Arithmetic	ADD, SUB, MUL, DIV, IDIV, MOD, IMOD, NEG, ABS, MIN, MAX
Integer Math	Bit-wise: BWAND, BWOR, BWXOR NOT SHIFTL, SHIFTR, ROTATEL, ROTATER PARITY, CRC16, CRC32
Floating Point Math	Trigonometry: SIN, COS, TAN, SINH, COSH, TANH, ASIN, ACOS, ATAN Logarithms and Exponents: LOG10, LOGN, EXP, POW, SQRT Rounding: INT, FRAC, CEILING, FLOOR, ROUND Random Numbers: RAND, SRAND
Resource Functions	Timer: TIMER_START, TIMER_ISDONE, TIMER_STOP, TIMER_RESTART, TIMER_WAIT Counter: COUNTER_TICK, COUNTER_ISDONE, COUNTER_VALUE, COUNTER_RESET, COUNTER_WAIT Queue:

	QUEUE_ENQUEUE, QUEUE_DEQUEUE, QUEUE_CLEAR, QUEUE_ROOM, QUEUE_USED
Array based	COPY, FILL, XFORM, MASK, TOGGLE, UNION
Miscellaneous	SIZEOF, COUNT, BYTE, WORD, DWORD, FLIP_BYTES, MAP_REF, ABORT, HOST_EVENT, DEBUG_PRINT
String based	STRLEN, STRCHR, STRRCHR, STRCMP, STRICMP, STRSRCH, STRUPPER, STRLOWER, STRCAT, STRCPY, TOSTRING

K. *Test Script Source Files*

A program written in test scripting language may span one or more test script source files with the extension ".rtsl". If sharing code snippets among multiple test script source files is desired, special test script files, called header files with the extension ".rtsh", can be used and referenced in test script source files with the INCLUDE keyword.

IV. Test Script Editor

The interpretation of test scripting language is done not on the test script source code, but rather on bytecode emitted as part of a compilation and bytecode generation process. The bytecode generation process compiles one or more test script source files into bytecode and ensures that cross-file references are resolved.

Figure 1 shows the proposed architecture for an integrated test script development tool called the Test Script Editor. The Test Script Editor consists of:

- a graphical user interface that provides tools for test script source file managements and editing
- a compiler that provides parsing and validation of test script source files
- a bytecode generator that serves as a linker and a generator
- a debugger that runs the generated bytecode in a simulated environment

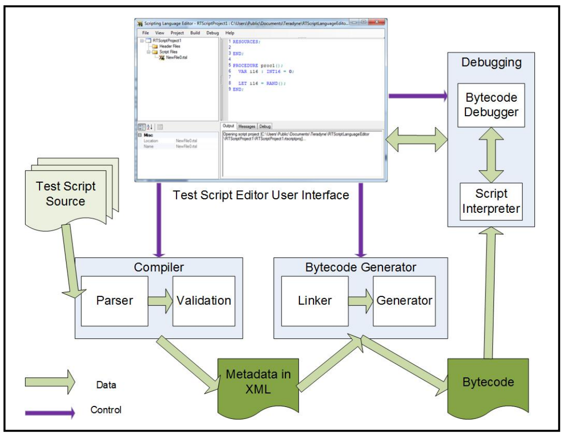

Figure 1: Proposed architecture for an integrated test script development environment

A. *Test Script Editor User Interface*

The Test Script Editor provides a powerful text editor to create and edit script source files. The text editor also provides common editing and search capabilities, context sensitive menus, coloring and highlighting text based on lexical elements of the test script source, capabilities that are expected from modern development environments.

In addition to the text editor, the user interface includes an error list window to list the errors generated during compilation or linking, and an output window that displays logged information on all actions taken in the Test Script Editor. The error list window is integrated with the text editor such that a user can display the source of an error in the text editor by double clicking an error listed in the error list window.

During debugging, the text editor displays a text version of the bytecode interleaved with the original source code. The user can follow test script execution by setting break points in the text editor, executing the test script until a break point is hit, then resuming execution by single-stepping through the generated bytecode.

The Test Script Editor manages multiple input files (source ".rtsl" and header ".rtsh" files), intermediate files (metadata in XML format and other debugging support files), and the output bytecode file. In addition, the Test Script Editor tracks user preferences; e.g. intermediate and output directories, file names, and endianness of the generated bytecode. In the Test Script Editor, all these files and user preferences are treated as a bundle and called a *test script project*. The Test Script Project Manager is a UI component in the Test Script Editor and maintains all project information in a project specific data file in XML format.

B. *Compiler*

The Compiler consists of a Parser and a Validation component. The Parser is a LR(1) parser, where the Parser reads the test script file in a **L**eft to right direction within each line, and top to bottom across the lines of the full input file [2]. The Parser produces a reversed **R**ightmost derivation, and is allowed to peek ahead 1 input symbol before deciding how to parse earlier symbols [3]. After the Parser successfully parses the script file input, it creates an internal representation of the script information in its local memory and passes that information to the Validation component to check for semantic errors; i.e. data type mismatches, procedure/function input parameter count and type mismatches, use of undefined variables, arrays, or procedure/functions. Essentially, the Compiler uses a two-pass scheme for syntactic and semantic validation of the script input. If the Parser finds a syntax error, it reports the error in the Error list window and exits immediately. Therefore, only one syntax error can be found at a time.

When both the Parser and the Validation components complete without any syntax or semantic errors, the Compiler generates an XML file (**debug.metadata**) containing the script metadata, which is an XML (human/machine readable) representation of the script file. Separation of the compiler from the linker and the bytecode generator is crucial to handling future modifications to the bytecode format and bytecode generator with minimum tool change effort.

C. Bytecode Generator

The Bytecode Generator consists of a Linker and a Generator. The Linker is responsible for reading in the script metadata in XML format generated by the Compiler and resolving external references in projects with multiple script files.

The Generator converts the metadata with resolved links into bytecode format. The bytecode starts with a lookup table that lists all the script routines, their names, and their start offsets from the beginning of the bytecode. The Script Interpreter uses this lookup table to find, verify, and execute the script routines. The bytecode uses opcodes that represent flow of control instructions (i.e. CALL, INCR, MOV, JMP, JMPF, POP, PUSH, RETURN), types (i.e. CHAR, INT16, INT32, INT64, UINT8, UINT16, UINT32, UINT64, REAL), resources (TIMER, COUNTER, QUEUE, REGION, MSGBUF), and built-in functions. The Generator performs the necessary calculations to handle the stack for local memory usage and shared memory for the resources, and inserts the necessary memory offsets for any variable, array element, record element, or resource element references in the generated bytecode.

In addition, the Generator creates two files to support debugging the bytecode using the Test Script Editor:

- **<project name>.bin.debug**: This file is a human readable version of the generated bytecode with the original script source lines in comments. This file is used for setting breakpoints and for tracing bytecode execution.
- **<project name>.bin.debuginfo**: This file provides additional information, e.g. memory stack changes, offset information for procedures and functions, to assist in debugging the test script.

D. Debugging

Debugging involves the execution of the generated bytecode by the Script Interpreter on the test system's controlling computer and the Test Script Editor displaying the execution on the Text Editor. This instance of the Script Interpreter is internally the same as the one running on the real-time processor during TPS execution. The difference is that the user is debugging in a simulation environment rather than he actual code running on the instrument.

A bytecode debugger component manages the interaction between the Script Interpreter and the various debugging windows on the Test Script Editor. A debugging session starts with the user selecting a script routine and initializing its input parameters. Procedures or functions cannot be used as starting routines, and only one script routine may be debugged at a time. The user may display the **<project name>.bin.debug** file in the Text Editor and set breakpoints for step by step debugging before starting the execution. When the execution stops at a breakpoint set by the user, the Test Script Editor provides two debugging windows for the user: 1) a Local Variables window to see the stack memory, local variables, their types, and current values and 2) a Call Stack that lists all the procedures/functions and current opcode offset with each scope. When execution stops at a breakpoint or due to a run-time error, a detailed log message is displayed in the output window.

V. RTScript: A Practical Example

For a concrete example of test scripting, we will once again make use of the FC-AE-RDMA [5] application referred to earlier. First, we describe the test script support that could be available to a TPS developer for this test application, and then we show how to use that support to solve a simple test problem.

A. FC-AE-RDMA Test Script Support[1]

The FC-AE-RDMA test instrument initiates I/O operations to send messages to a UUT, and responds to I/O operations to receive messages from the UUT. In this application, these messages are identified by number. Messages with the same number contain the same type of data, and multiple instances of the same message may be sent or received.

As mentioned earlier, test script support starts with knowing the particular test events that are supported in the embedded program running on the test instrument. We listed those events earlier:

- Start of test
- UUT I/O operation initiated
- UUT I/O operation completed
- Test instrument I/O operation initiated
- Test instrument I/O operation completed

For these test events, the TPS developer may register a script routine. The record type associated with the I/O operation test event script routines are similar, and for the purposes of this test are defined as:

```
RECORD $RDMA_MESSAGE;
     VAR messageNumber : INT32;
     VAR length : INT32;
     VAR msgBuf : MSGBUF;
END;
```

For this application, the message data to be sent to and received from the UUT is stored in MSGBUF resources, a reference to which is included in the record definitions for the I/O operation test event script routines to allow the routines to modify that data prior to transmission or after reception. The MSGBUF resources are defined in the TPS and referenced in the script routines. At creation time, the TPS developer assigns each MSGBUF an integer key value, by which they are can accessed in the script routines.

The record type associated with the start of test event is assumed to be empty.

In addition to being able to respond to test events, it is useful to be able to initiate some test instrument operations from the test script. In this case, we will assume the FC-AE-

[1] The support described here is based on an actual FC-AE-RDMA test application developed for use on the Teradyne High Speed Subsystem.

RDMA application framework provides the following built-in procedure to send a message to the UUT:

```
    PROCEDURE SEND_RDMA_MSG(msgNum : INT32);
```

B. *Example Test Problem*

To illustrate the use of test scripts, assume we need to test a UUT that communicates with a second unit in a point-to-point Fibre Channel topology [4]. The UUT sends instances of messages 1-16 to the target unit. That unit is expected to send an instance of message 0 back to the UUT every second. The payload of that message 0 instance contains 16 pairs of integers, one pair per message 1-16. The first integer is the number of instances of that message received in the last interval, and the second integer is the total amount of data received. Table 3 shows the layout of the message 0 payload.

Table 3: Example Message 0 Payload Layout

Byte Offset	Size	Contents
0	4	# of Message 1 instances received
4	4	# of bytes of Message 1 data received
8	4	# of Message 2 instances received
12	4	# of bytes of Message 2 data received
…	…	…
120	4	# of Message 16 instances received
124	4	# of bytes of Message 16 data received

C. *Example Test Problem Solution*

We propose the following solution to the posed test problem:

- Create various resources to assist in solving the problem:
  - A timer resource with a 1 second expiration.
  - A counter resource to track the number of event occurrences.
  - A queue resource to data associated with those event occurrences.
  - A region to act as a global variable to track the number of event occurrences that have been processed
  - Reference the MSGBUF for message 0 defined in the TPS
- In the TPS code, associate test script routines with the UUT operation completed test event for all 16 messages. In this case, we will use the same test script routine for all of the received UUT messages. In that routine, increment the counter and insert data into the queue to indicate which message was received and the amount of data associated with it.
- Define a test script routine that is associated with the expired event for the timer. Use that script routine to send out message 0. Populate the message 0 payload by extracting the entries accumulated in the queue and updating the message 0 payload accordingly.
- Define a test script routine that is associated with the start of test event. Use that script routine to start the timer.

The resource definition section of the script code looks like:

```
RESOURCES;
     TIMER timerHeartbeat
             DURATION 1.0
             RESTART AUTO
             ON_DONE TxMsg0;

     COUNTER counterEvents
             RANGE 0xFFFFFFFF
             RESTART MANUAL;

     // Sizes are in bytes
     QUEUE queueMsg 1024;

     REGION regionProcessedEvents 4;

     // The value 10 is the user-specified key
     // assigned to the buffer when it was
     // created in the TPS and associated with
     // message 0.
     MSGBUF msgBuf0 10;
END;
```

The script routine used to handle the incoming UUT messages is fairly simple: the routine extracts the message number and its length from the routine's input record, inserts that as a unit into the queue resource, and increments a count of such entries.

```
RECORD msgEventQData;
     VAR msg : INT32;
     VAR len : INT32;
END;

ROUTINE <$RDMA_MESSAGE> UutMsgRx;
     VAR evData  : RECORD <msgEventQData>;
     VAR success : BOOL;

     LET evData.msg = THIS.messageNumber;
     LET evData.len = THIS.length;

     LET success = QUEUE_ENQUEUE(queueMsg, evData);
     COUNTER_TICK(counterEvents);
END;
```

The script routine TxMsg0, used to populate the outgoing message 0 payload, is a bit more complicated (refer to the code snippet below).

The MAP_REF instructions after the variable declarations are used to dynamically cast the message 0 MSGBUF into an array of msgEntry records and bytes, and the region used to hold the count of processed events into an integer. The message payload is then cleared. Next, a count of the number of events to process in this instance is determined from taking the difference between the current received message count (from the counter resource) and the number of messages processed up to now. The resulting value is used to control a FOR loop.

In the FOR loop, an entry from the queue resource is read out, the corresponding message number is determined from that entry's "msg" field, and that message's entry in the payload is updated, by incrementing the message count and adding the current instance size to the total amount of data processed for that message so far.

Once the payload is updated, the script routine instructs the embedded program to send out the message 0 by calling the application framework function SEND_RDMA_MESSAGE.

```
RECORD msgEntry;
     VAR msgCnt   : INT32;
     VAR dataSize : INT32;
END;

ROUTINE <$TIMER_EVENT> TxMsg0;
  VAR success      : BOOL;
  VAR msgIdx       : INT32;
  VAR evCount      : INT32;
  VAR i            : INT32;
  VAR evData       : RECORD <msgEventQData>;
  VAR thisCnt       : INT32;

  REF VAR lastCnt  : INT32;
  REF ARRAY m0Data : RECORD <msgEntry>[16];
  REF ARRAY m0AsBytes : BYTE[128];

  MAP_REF(m0Data, msgBuf0, 0);
  MAP_REF(m0AsBytes, msgBuf0, 0);
  MAP_REF(lastCnt, regionProcessedEvents, 0);

  FILL(m0AsBytes, 0);

  LET thisCnt = COUNTER_VALUE(counterEvents);
  LET evCount = SUB(thisCnt, lastCnt);
  LET lastCnt = thisCnt;

  FOR (i FROM 1 THRU evCount);
    LET success = QUEUE_DEQUEUE(queueMsg, evData);
    LET msgIdx = SUB(evData.msg, 1);
    LET m0Data[msgIdx].msgCnt =
      ADD(m0Data[msgIdx].msgCnt, 1);
    LET m0Data[msgIdx].dataSize =
      ADD(m0Data[msgIdx].dataSize, evData.len);
  ENDFOR;

  SEND_RDMA_MSG(0);
END;
```

VI. CONCLUSIONS AND FUTURE WORK

We hope that we have been successful in demonstrating some of the advantages of a test script as a means of improving TPS developer efficiency in test applications with real-time requirements. The test scripting language we have described in this paper offers a simpler alternative to TPS developer authored embedded program development.

While a good start, we do not believe that our work in this area is complete. We have identified areas where we can extend the language, including improved support for multi-threading, additional application frameworks, and additional mathematical analysis functions. Further improvements in the toolset are also planned, including improved debugging support.

ACKNOWLEDGMENT

Thanks to Stephan Krach for his work with the Script Interpreter.

REFERENCES

[1] B. W. Kernighan, D. M. Ritchie, “The C Programming Language,” Prentice Hall, April 1988.

[2] A. Aho, M. Lam, R. Sethi, J. Ullman, “Compilers: Principles, Techniques, and Tools”, 2nd ed., Prentice Hall, 2006.

[3] LR parser, http://en.wikipedia.org/wiki/LR_parser, July 2013.

[4] R. W. Kembel, “Fibre Channel A Comprehensive Introduction” Northwest Learning Associates, 2003.

[5] ANSI INCITS TR 40: INCITS Technical Report for Information Technology – Fibre Channel – Avionics Environment – SCSI-3 Remote Direct Memory Access (FC-AE-RDMA), January 2005.