Sequence Language User's Guide (SLUG)

Introduction

This page contains the user's guide for the LASP Awesome Sequence Engine Language (LASEL).

Sequence Tools Details

There are three tools available:

- Builder - create a binary file to load to the target
- Decoder - decompile a binary file into its low-level instructions
- Simulator - run the sequence outside the flight environment to see how it works

All Tools

The following options are available to all sequence tools.

- Set the directory where sequence binaries are located
    - `-d sequenceDir, --seq-dir sequenceDir`
- Set the config file
    - `-c configFile, -config configFile`
- Display the usage statement
    - `-h, --help`

Builder

These instructions will create binary sequence files that can be executed via the flight, pc-linux, or sequence simulator targets.

Execution Steps

1. Create a sequence file in a text editor. See section below for details.
2. Execute the builder on the file or set of files:

    a. `$ run_seq_build [options] path/to/file(s)`

Decoder

The purpose of this tool is to translate a binary file into readable text for debugging and troubleshooting. High level constructs such as if statements are translated into lower level instructions by the builder, so the output of the decompiler is mostly useful to developers.

The output is written to stdout.

Execution Steps

1. Create a binary file via steps above.
2. Execute the decoder on the file or set of files:
    a. `$ seq_decode [options] path/to/file(s)`

Simulator

The simulator will execute all instructions in a binary file in an interactive manner.

Execution Steps

1. Create a binary file via the steps above.
2. Start the simulator in interactive mode:
    a. `$ run_seq_sim sequenceName_1`

Sequence Creation

Sequences are text files that conform to the below format. The extension of these files does not matter (txt, seq, etc). Multiple sequences can be in one input file. Separate binary files will be produced for each one.

> *It is important that there are spaces between all keywords and operators.*

A Note About Case

Sequence files are case-insensitive with respect to the following items:

- sequence keywords
- command names
- command parameter names
- telemetry item names
- state conversions for command parameters and telemetry

Variable names, labels, command parameter values (other than states), file paths, etc are all case sensitive.

Sequence Name

Sequence names are in the form sequence_name_XX where XX is a 16-bit sequence identified that gets compiled into the sequence binary. This format should be used in the filename (ie. sequence_name_XX.txt) and in call/spawn/start statements (ie. call sequence_name_XX). Sequences names that are just raw numbers (ie. XX) are also allowed.

Sequence Execution

The Adamant sequence will synchronously execute all sequence instructions (up to a compile-time defined limit) until a blocking condition is encountered. Blocking conditions include: waiting on a command execution to complete, waiting on a sequence start/spawn/call to complete, waiting on an absolute or relative time, or waiting on a telemetry condition to become true. If the compile-time defined limit is reached, the sequence will exit with a runtime error. This limit exists to prevent a sequence from overrunning its allocated execution time, which would have adverse affects on other tasks running in the FSW. In order to avoid hitting this limit, try not to write sequences with long series of non-blocking operations, such as computing the 100th digit of pi.

Basic Sequence

All sequences must contain the following lines:

```
seq sequenceName_1
;Comment
endseq
```

The sequenceName_1 is used for the binary file name that is produced. The maximum sequenceName_1, including the .blk or .seq extension, is limited to a 20 character length. The ';' character is used for comments, which are ignored by the builder.

Sending Commands

Command statements start with the keyword cmd, followed by the command name and any parameters.

- The parameters do not need to be in any particular order.
- If a parameter is omitted, the default bit pattern is used instead

The current format for commands is extElement_cmd_cmdName.

The following example sends two dummy commands, one with parameters and one without.

```
seq sequenceName_1
cmd fsw_cmd_dummy1
cmd fsw_cmd_dummy2 param1 ON param2 1.5
endseq
```

If a parameter has a state conversion, the converted value or the raw value can be used as the parameter.

Variables

Sequences can have local variables, which are only in scope for that sequence and no other engine or sub-sequences can access them. There is a limit of 16 local variables for each sequence.

When declaring a variable, the type and size must be provided. Sizes less than 32 are currently treated as 32 bits until a future release.

```
seq sequenceName_1
declare myVar1 i32
declare myVar2 u32
declare myVar3 f32
declare myEnum d32 0/FALSE 1/TRUE
endseq
```

The following types are valid for variable declarations:

- unsigned (u)
- signed (i)
- floating-point (f)
- discrete (d) (as of 0.17.0)

To declare a discrete type, the enumerations must be provided in the declaration statement as above. All discrete types are treated as unsigned integers. The enumeration strings are not case sensitive.

To reference a variable in the sequence it must first be declared and then preceded by the $ character.

```
cmd fsw_cmd_dummy2 param1 ON param2 $myVar3
```

Variables can be used as command parameters, in logic statements and arithmetic.

To set a variable, use the set keyword.

```
set $myVar1 = 1
set $myVar2 = -1
set $myVar3 = 1.5
```

For signed numbers and floating point, it results in the most efficient sequence if the type is obvious when using constants. For instance, using 1.0 instead of 1 for floating point and +1 or -1 for signed. Otherwise instructions to cast values to the proper type will be inserted.

Hexadecimal representations for constants should be preceded by 0x.

Expressions

For all expressions, as of this writing, only one or two operands and a single operator is permitted. Use multiple lines to perform complex calculations.

The expression is to the right of the equal sign.

```
set $myVar = 1 + 1
set $myVar = $myVar + 1
set $myVar = $myVar + $myVar
```

Telemetry items can be used in expressions. If the item has an external element, it should be prepended to the item name with a single _.

```
set $myVar = fsw_cmdacptcnt
set $myVar = fsw_cmdacptcnt + 1
```

An example of a complex boolean expression broken over multiple lines: (var1 && var2) || var3

```
set $myVar = $var1 && $var2
set $myVar = $myVar || $var3
```

Print Statements

The print statement embeds a string or variable print in an event message. All messages are sent with the same message ID and the type is set in the instruction. A print statement can either contain a literal string, or a reference to a local variable. The two cannot be mixed in the same statement at this time. Literal strings must be quoted if they contain spaces.

The event message types are: info, debug, critical, and error.

```
print info "This is my informational statement"
print error "Bad thing is happening!"
print debug "The value of 2 + 2 is 4"
print critical "Burn it all down"
print info $myLocalVar
```

Wait Statements

Sequences can be paused by inserting wait statements in between commands and other actions. There are three types of waits: absolute, relative, and value.

```
wait for 1 ;relative wait
wait until 120000 ;absolute wait (VTC)
wait until 2017-200T00:00:00 ;absolute wait (UTC)
```

The sequence can also wait until an expression is true, or for a timeout.

```
waitvalue available_cmdacptcnt == 1 ? for 5 ;relative wait
waitvalue fsw_cmdacptcnt == 1 ? until 120000 ;absolute wait (VTC)
waitvalue fsw_cmdacptcnt == 1 ? until 2017-200T00:00:00 ;absolute wait (UTC)
```

As of this writing the time resolution for sequences is 1 second. The only allowed time format is VTC (Vehicle Time Code).

UTC time must be formatted as YYYY-DOYTHH:MM:SS or it will not be recognized as valid.

A variable can be supplied for the wait time, instead of a constant.

```
wait for $myTime ;relative wait
wait until $myTime ;absolute wait (VTC)
waitvalue fsw_cmdacptcnt == 1 ? for $myTime ;relative wait
waitvalue fsw_cmdacptcnt == 1 ? until $myTime ;absolute wait (VTC)
```

If a wait times out, then the internal variable WAIT_TIMEOUT is set to 1. This variable can be used in an if statement or expression just as any other local variable.

Wait statements will always use the current value of an item in the database. An alternative statement, waitnewvalue, can be used to ensure that the telemetry value is updated after the wait has commenced.

```
waitnewvalue fsw_cmdacptcnt == 1 ? for $myTime ;relative wait
waitnewvalue fsw_cmdacptcnt == 1 ? until $myTime ;absolute wait (VTC)
```

This keyword can be used in place of *waitvalue* in order to guarantee that the timestamp of the telemetry item is newer than the time at the beginning of the wait. This ensures that the telemetry being compared is "new" and not stale. Using *waitvalue* can result in comparing a telemetry value that was produced long before the wait began. This can be desirable behavior for telemetry items like the system mode, which is only changed sporadically when a command is sent to change the mode, but it may not be desirable for a periodically updated telemetry item like a temperature, which should never be used if too old. The choice to use *waitvalue* vs *waitnewvalue* should be determined for each usage depending on the type of telemetry being compared to.

Telemetry Items

Telemetry items in Adamant live in an onboard database and are fetched when a sequence requires them. If an item is not found in the database or is stale (see *waitnewvalue* feature above), a timeout will be issued if a new telemetry item does not appear in the database within 10 seconds.

Starting and Stopping Sequences

There are three options for starting a new sequence within a sequence: start, call, and spawn.

The "start" command begins a new sequence in the same engine as the previous and overwrites the previous sequence. This is mostly useful to chain sequences together in the same engine: ie one sequence ends and starts the next one in the line.

The "call" command begins a new sequence as a sub-sequence to the caller. When the sub-sequence is complete, the caller will continue execution. The number of sub-sequences allowed is 2.

The "spawn" command begins a new sequence in another engine. The engine number can be provided, or the any keyword can specify the next available.

If the sequence is not at the base path (/ram/seq), the relative path must be included in the command so the sequence can be found.

```
call subsequence
start newsequence
spawn on 1 newsequence
spawn on any newsequence
```

Parameters can be passed to sequences when they are started. To define input arguments for a sequence, the argument keyword is used.

```
seq subsequence
argument myArg1 i32
argument myArg2 i32
endseq
```

The syntax for arguments is the same as declare. Argument declarations must precede local variable declarations within the file, and the total number of arguments + local variables cannot exceed 16.

The order in which the arguments are defined determines the order they must be specified in the start command.

```
call subsequence -1 -1
```

Discrete Parameters

While a discrete variable can be passed to a sequence call, a discrete literal cannot. This means that if an argument in the callee sequence is declare as discrete, the caller sequence must still pass in the integer value, not the enumeration string for the value.

Sequences nominally end when there are no more instructions to process, or the return command is encountered. They can also be killed explicitly.

```
kill engine 1 2 ;terminates engines 1 and 2
kill engine 5 10 ;terminates engines 5 through 14
return
return 1
return $returnVar
```

Killing by engine number allows a range of engines to be provided. The first parameter is the starting engine number, and the second is the number.

The return value can be accessed by the calling sequence via the RETURN_VAL variable. This variable is an internal sequence variable and can be used in an expression just as any other local variable.

Flow Control

Flow control constructs change the order in which sequence statements are executed. All of the constructs described can be mixed and matched and nested inside each other.

Gotos

A goto statement is the simplest flow control available. Execution will always jump to the location provided in the command.

```
seq mySeq

LABEL1:

LABEL2:
goto LABEL1
endseq
```

The argument of a goto command must be a label. Labels are single words that end in the ':' character.

If Statements

The if statement branches if the expression in it evaluates to zero. An optional else statement can be used as well.

```
if expr
;Do something
else
;Do something else
endif
```

The expr can be any of the expressions described above.

While Loops

The while loop repeats statements until the expression evaluates to 0.

```
while expr
;do something
endwhile
```

An alternative form is the do-when statement, where the contents of the loop are always executed at least once.

```
do
;fun things!
when expr
```

For Loops

The for loop executes a specific number of times, rather than waiting for a condition. The loop variable will increment or decrement each time through the loop and can be used as a normal variable. The default increment is +1, but that can be changed as shown below:

```
declare i u32
for $i = 0 : 10 ;Increment i by 1 while i <= 10 (loop 11 times)
endfor

for $j = 0 : 10 : 2 ;Increment j by 2 while j <= 10 (loop 6 times)
endfor
```

Switch Statements

The switch statement is a compact way to simplify a change of if and else statements. Its form is similar to that used in C/C++.

```
switch $myVar
case 1
;do something
break

case 2
;do something else
break

default
;do something default
break

endswitch
```

Other Info

Keywords

A useful construct is the keyword, which is a builder only directive to substitute one string for another. The scope of a keyword can be restricted to the sequence or apply to the entire file. If defined in the config file, the keyword will apply to all sequences that reference that file.

```
keyword globalKey Hello

seq seqName
keyword localKey World
cmd fsw_cmdhelloword param1 @globalKey param2 @localKey
endseq
```

Keywords are dereferenced with the '@' character.

Keywords are substituted before another processing occurs, so can be used for any sequence directive:

```
seq seqName
declare myVar i32
keyword setKey set

@setKey $myVar = 1
endseq
```

Long Lines

Normally all sequence commands must be contained on one line with no breaks. If a break is needed due to length, the '\' character must be the final character before the new line.

```
set $myVar = \
1
```

Config File

All of the sequence tools read the same configuration file. It is a text file with a similar format to sequences. The default location for the file is config /build_config.txt. A different file can be provided with the -c flag to any of the described tools.

User Commands and Telemetry

Custom commands and telemetry items can be provided in the configuration file. The flight definitions are read from the XTCE files. Custom items are useful for writing test sequences or defining an item that isn't yet in the official XTCE release.

For commands, the bit pattern must be provided. Any headers, including the CCSDS header must be included in the pattern. If there are parameters, the locations for the values should be set to zero, unless the parameter has a default value.

```
userCmd cmdName pattern 12345600000000
userCmd cmdName parameter param1 8 32 I
userCmd cmdName parameter param2 8 40 D
userCmd cmdName parameterState param2 OFF 0
userCmd cmdName parameterState param2 ON 1
```

The parameters are provided with their name, size in bits, offset in bits, and type. See the CT database schema for more details on these fields. Optional state conversions can be defined for parameters with type D.

Telemetry items are defined similarly:

```
userTlm tlmName details 0x800 8 96 U
userTlm tlmName state OFF 0
userTlm tlmName state ON 1
```

For telemetry, the details needed are msgId, size in bits, offset in bits, and type. See the CT database schema for more details.

Paths

This section lists the default paths that can be changed in the config file:

```
sequencePath big|little ../bin ;Path for binary sequence outputs
```

The sequencePath is used by the builder when it creates sequences, as well as the simulator when it searches for them. The type specifies the endiannes of the files at the provided path. Adamant expects sequences to be compiled in big endian format regardless of the CPU architecture it is running on.

Signature Files (build 0.19.0)

Signature files contain the calling signature of sequences. This signatures are used by the sequence builder to check call/spawn/start statements against the called sequence to make sure the number of input parameters match and the parameter types are correct. Casting will be applied to constants if needed. Otherwise, if a parameter does not match, the sequence will not be built.

If a signature is not available, then the builder will issue a warning but will process the statement the best that it can. As many files as needed can be listed in the config file.

```
sigFile path/to/filename.txt
```

The signature of a sequence consists of its name, argument names, and argument types. Future builds will include range checking for arguments as well. An example signature file is below:

```
seq test_seq1
  argument arg1 u32
  argument arg2 d32 0/OFF 1/ON

seq test_seq2
  argument arg1 u32

seq test_seq3
seq test_seq4
```

Command Bit Pattern Decoding and Aliases

When decoding a command bit pattern back to a string, the simulator and decoder follow the following steps:

1. Search database for exact match to existing bit pattern (sequence count and grouping flags not included)
2. If no exact match is found, Extract Message ID and opcode from bit pattern
    a. If Message ID is 0x1A34 and opcode 0 (COM pass through) it gets the real opcode from the pass through field instead of the default location (CCSDS secondary header)
3. Search database for bit pattern that matches Message ID and opcode, and does not contain the word Parent in its LongDescription
4. If still no match, return Unknown Command
5. If more than one match, print first result and mark with a *

This behavior can be superseded by providing a bit pattern to string mapping in the config file.

```
aliasCmd ThisIsTheCmdString ABCD12345678
```

In this example, if the bit pattern ABCD12345678 is encountered, it will use the command string provided instead of querying the database.