GNU ccScript Scripting Guide IV

David Sugar
GNU Telephony

2008-08-20

(The text was slightly edited in 2017.)

Contents

1 Introduction

This document covers the "4th" major release of the GNU ccScript engine. GNU ccScript is a highly specialized embedded scripting engine and virtual execution environment for creating applications that may be automated through scripting. This system is a C++ class library which allows one to derive application-specific dialects of the core language as needed through subclassing. GNU ccScript is not meant to run as a "shell" or offer interactive user sessions.

What makes ccScript different from similar packages is its deterministic runtime, and its use of step execution for integrating with realtime state transition and callback event-driven systems. That is, rather than invoke a function which then parses an interpreter, one executes a single `step` statement. This is useful when a callback service thread is monitoring multiple devices; each device can have a ccScript interpreter instance, and a single thread can dispatch events upon demand.

GNU ccScript is also meant to script systems without downtime. This has one curious effect: when ccScript is started, a core script is converted into a reference-counted object. When an interpreter instance is connected to process script requests, it attaches a reference. If an active script is modified and reloaded, a new "image" is created, and new requests are then given this new image. When the last reference against an old script image is terminated, the image is also purged. This allows one to compile and rebuild scripts on the fly and load them into memory even while active interpreter instances are running.

A number of specialized optimizations also exist in GNU ccScript to greatly reduce runtime heap management, especially for running massively concurrent instances of the interpreter. Another core feature of the GNU ccScript system is support for extensive compile-time script analysis. This reduces the need for runtime error checking, and the risk of using incorrect scripts.

2 Script file layout

A single text file is used to represent an application "script". This text file is compiled into a reference-counted image that is then executed at runtime. The form and layout of a script application has been defined in release IV as follows:

First, there is an initialization block, which appears at the start of the script file without a label. The initialization block is called anytime a runtime script is "attached", whichever part of the script is run first. Only certain statements can be used in the initialization block. They are generally related to defining and setting global applications and constants.

Some special commands must appear at the very start of the script file. The `strict` command is implemented this way. Derived application servers may have other special commands which have this requirement, or other special commands that can only be placed in an initialization block.

The next part, immediately following the initialization block, may contain `define` statements introducing user-defined functions, and `template` statements introducing event handlers. The user-defined functions are written in the scripting language, and operate as if they were new builtin script commands. Variables can be scoped to a function; arguments can thus be passed as either values or references. Templates are used to apply event handlers to script "sections" so they do not have to be individually recoded.

The script sections follow any defines and templates. Each section begins with a `@` and a label. There are two special labels, `@main`, which is the default entry point for starting a script application unless the attach method chose a different label, and `@exit`, which is used when the application service script terminates.

Each script section can have under it one or more named event handlers, each of which has script statements. Multiple event handlers can be listed next to each other directly, thereby sharing the same script statements. All of them now have free-form definitions. The exact meaning or use of predefined event handlers will depend on the derived application service. The `^error` event however is always used when the script engine finds a runtime error, and the `^exit` handler, if found for the currently running script, is called before the `@exit` section on termination events.

Each script may also have a special `^init` event handler. This is called each time a script is started, whether from initial attach of `@main`, or as a result of a `goto` or `gosub`. The `^init` section is called before the main part of the script. Any events that occur will be blocked until `^init` completes, and so this can be used to complete initialization of all variables before the script – which itself can otherwise be interrupted by an event – actually starts, or (for example in the case of Bayonne) to play prompts that are not interrupted, in a script that also has key events.

Starting with ccScript 4.0.1, "templates" can be created which can then be applied to existing script sections. When this is done, the `^event` handlers of the template will be added to the script `@section` they are applied to. If the script `@section` has defined its own local handlers, these will override the template handlers.

Templating allows one to define the event handlers that are shared between multiple sections of a script once and for all, rather than repeatedly recode each handler in each section. Only one template can be applied to a `@section`, and `apply` must be the first command of the section.

Templates can be applied to definition blocks as well as to script sections. The only requirement is that the template must appear in the script file before it is applied. An `apply` command can also be used within the template itself. Thus one can compose a template from the event handlers of multiple templates which can then be collectively applied to a script section or definition block.

Starting with ccScript 4.0.3, conditional compilation is supported using one or more `requires` blocks. The conditional section is ended with an `endreq` statement. The `requires` statement tests for the presence of keywords in the interpreter and/or definition blocks. If the required commands are not found, then compilation is skipped until the next `requires` statement can be tested or an `endreq` statement is used. A `requires` statement can make any part of a script file conditional, including labels and event handlers.

Testing for the absence of a command can be done using `!token`. Hence, a conditional block can be started with "`requires token`", and alternate code can be substituted using "`requires !token`". The entire block can then be ended with `endreq`. If multiple command tokens are listed, then the presence (or absence) of all the listed tokens must be true for the `requires` block to conditionally compile.

3 Statements and syntax

Each script statement is a single input line in a script file. A script statement is composed of a script command statement, any script command arguments that might be used, and any keyword value pairs. White spaces are used to separate each part of the command statement, and white spaces are also used to separate each command argument. The newline terminates a script statement.

GNU ccScript IV is case-sensitive. All built-in keywords are lower case. Symbol names and defines which are in mixed case have to be used in the same manner and case they are defined. Hence `sET` WOULD NOT BE the same command as `set`.

Commands are composed of – and keywords can be assigned with – literal strings, or numbers (integer or decimal), or even references to symbols.

String literals may be enclosed in single quotes, double quotes, or a pair of `{}` braces. Literals can also be a single integer or decimal number.

Symbols are normally referenced by starting with the `%` character. Special variable formatting rules may be specified with the `$` character followed by the format option, a colon (`:`), and the symbol name, such as `$len:string` to retrieve the length of the variable `%string`. Symbols, when used to define or assign new symbols, usually are specified without any prefix, but may contain a "`:`" suffix with a size or type field.

Symbols may be scoped. If a symbol comes into existence as part of a statement in a `define` block, or as part of the arguments of the command which invoked the `define` block, that symbol can only be referenced from within that block. All other symbols are exposed and scoped globally. Locally scoped symbols in a `define` block can hide/have the same name as a global one.

Some statements may be conditional statements. These may use either a single test which starts with `-` or `!`, the latter being verified if the test fails rather than succeeds, or two values joined by an expression, "`%myvar = 2`" for instance. Multiple conditions can be joined together with either `and` or `or` into a more complex expression.

`if` is a special command where two statements may exist on one line. This happens when `if` is used with a conditional expression followed by a `then` clause. Any single script statement may then appear, and will be executed if the expression is true. If there is no `then` clause following the `if` statement, then a multi-line `if` block is assumed, which may include `elif` and `else` sections, and requires an explicit `endif` line.

4 Loops and conditionals

Scripts can be broken down into blocks of conditional code. To support this, we have both if-then-else-endif constructs, and case blocks. In addition, blocks of code can be enclosed in loops, and the loops themselves can be controlled by conditionals.

All conditional statements use one of two forms; either two arguments separated by a conditional test operator, or a test condition and a single argument. Multiple conditions can be chained together with the `and` and `or` keywords.

Conditional operators include `=` and `<>`, which provide numeric comparison of two arguments, along with `>`, `<`, `<=`, and `>=`, which also perform comparison of integer values. A simple conditional expression of this form might be something like "`if %val < 3 then break`", which tests to see if `%val` is less than 3, and if so, breaks a loop.

Conditional operators also include string comparisons. These differ in that they do not operate on the integer value of a string, but on its effective sorting order. The most basic string operators include `==` and `!=`, which test if two arguments are equal or not. All comparisons are case-sensitive.

A special operator, `?`, can be used to determine if one substring is contained within another comma-separated string. This can be used to see if the first argument is contained in the second. For example, the test "`th ? fr,th,is`" would be true, since "th" is in the list. As in Perl, the `~` operator may also be used. This will test if a regular expression can be matched with the contents of an argument.

A special string "match" function is defined with the `$` and `!$` operators.

This may depend on the derived application service, but by default may be used to perform a case-insensitive search.

In addition to the conditional operators, variables may be used in special conditional tests. These tests are named `-xxx`, where "`-xxx <argument>`" will check if the argument meets the specified condition, and "`!-xxx <argument>`" will check if it doesn't. The following conditional tests are defined in ccScript (additional ones may exist in an implemented application service):

Conditional	**Description**
`-defined`	tests if a given argument is a defined variable
`-empty`	tests if the argument or variable is empty or not
`-const`	tests if a given argument is a constant variable
`-modify`	tests if a given argument is a modifiable variable
`-integer`	tests if a given argument is an integer number
`-digits`	tests if a given argument is only composed of digits
`-number`	tests if a given argument is an integer or decimal number

The "`if <condition>`" statement can take two forms. It can be used by itself, or in "`if <condition> then ...`" constructs where the `then` clause is executed if the condition is true. The "`if <condition> then ...`" block continues until an `endif` command is reached, and may support `elif` and `else` options as well. This form is similar to the Bash shell if-then-fi conditional.

The `case` statement is followed immediately by a conditional expression, and can be used multiple times to break a group of lines up until the `endcase` is reached or a loop exits. The `otherwise` keyword is the same as the default case in C. The `break` command can force a case block to immediately exit through its `endcase` statement.

The `do` statement can be used to enclose a loop. This loop can be ended either with the `loop` statement, or with `until`. The latter supports a conditional clause. A "`do ... loop`" block will loop indefinitely. However, all loops, including `do`, may also be exited with a `break` statement, or repeated again with the `continue` statement.

The "`while <condition>`" statement can be used together with `loop` to form a conditional looping block of code so long as the condition remains

true. A "`for <var> <value1> <value2> ...`" loop can be used to assign a variable from a list of values. "`foreach <var> <value>`" is used to assign a variable from a comma-delimited list, or from the comma-delimited contents of a symbol. All loops other than `case` blocks may be nested, as well as the if-xxx-endif clauses.

5 Symbol Formatting

Symbol formatting allows the value of a symbol to be transformed, or accessed in a manner different from the default content. This can be useful for extracting fields from a comma-delimited keyword list, to get the length of a symbol, or even to create special rules such as phrasebook expressions.

The rule is in the form `$rule[/option]:symbol`. If no rule is specified, then `$symbol` by itself is the same as `%symbol`. The actual ruleset can be extended in application services, but the following are predefined:

Rule	Description
`$bool`	gets symbol as true/false boolean
`$dec`	decrements symbol and returns next value
`$find/name`	gets named member from comma-delimited content
`$head`	gets first item in a comma-delimited list
`$inc`	increments symbol and returns next value
`$index/val`	gets partial string by offset index
`$int`	gets symbol as integer value
`$key`	gets keyword of a key/value pair
`$len`	gets length of symbol
`$lower`	converts symbol to lower case
`$map/sym`	maps a symbol to use as find or offset
`$num`	gets numeric symbol to decimal precision of runtime
`$pop`	pops last item from a comma-delimited list
`$pull`	pulls first item from a comma-delimited list
`$offset/val`	gets tuples from specified offset
`$size`	gets storage size of symbol, 0 if const
`$tail`	gets last item in a comma-delimited list
`$unquote`	removes quoting from symbol
`$upper`	converts symbol to upper case
`$val`	gets value of a key/value pair or unquotes a list item

6 Command Reference

These are the initial built-in commands of the core ccScript engine. Application servers may add further commands of their own.

add *symbol[:type or size] value ...*
Sets an existing symbol or creates a new global symbol. If the symbol exists, values will be appended to it.

apply *definition*
Applies the event handlers of a script definition to the current script section. If used, this must be the first statement in a section.

break
Exits a `case`, `do`, `while`, `for` or `foreach` block.

case *condition*
Executes block of code if the condition is true. Otherwise tries next `case` section, an `otherwise` block, or reach `endcase`.

clear *%symbol ...*
Clears one or more symbols.

const *symbol=initial ...*
Initializes one or more read-only constants. If the `const` is in a `define` block, the symbol is created in local scope.

continue
Repeats a `do`, `while`, `for` or `foreach` block.

do
Begins a do-loop block. This can be ended with either a `loop` or an `until` statement.

elif *condition*
Enters section of an `if` block if the condition is true, and no other `elif` section has been entered. Hence, if an `if` statement is true and executes lines, control skips to `endif` when `elif` is reached .

else
Used for the `else` section of an `if` block.

endcase
Ends a `case` block.

endif
Ends an `if` block.

error *text...*
Generates a runtime error. The text is copied into the internal `%error` symbol, and the script's `^error` handler, if any, is called.

exit
Exits the script. Calls a `^exit` handler if there is one, or the `@exit` section of the script.

expand *tuples symbol[:type or size] ...*
Expands a variable or literal holding a list of tuples into a list of symbols. If there are nested tuples, then the nested set is assigned as a new tuple list to a symbol.

expr *symbol[:type or size] = value [op value] ...*
This is used to assign a symbol from a simple math expression, such as "`expr %myvar = 3.5 * %somevar`". The decimal precision can be overriden with the `decimals=` keyword; `decimals=0` ensures integer results. One can also perform expressions in assignment, for example, to increment an existing variable with a `+=` or decrement with `-=`. Hence, one can use "`expr %myvar += 3.5`" for example.

for *symbol[:type or size] value ...*
Begins a `for` block of code, assigning each value to the symbol in turn and calling the statements in the block until `loop` is reached.

foreach *symbol[:type or size] value[offset]*
Begins a `for` block of code, assigning each member of a comma-delimited value to the symbol in turn and calling the statements in the block until `loop` is reached. Optionally, a number of entries may be skipped at the start of the loop. If this option is used and the number is a variable, it is automatically reset to 0 when the `foreach` loop is entered.

goto *@label or ^event*
Transfers control to a new label or an event handler in the current script.

gosub *@label*
Calls a script label as a subroutine.

if *condition* **then** *statement*
: If the specified conditional expression is true, the statement after the `then` is executed.

if *condition*
: Used to start an `if` block when the condition is true. If false, the `elif` may be tried, and finally either the `else` clause will execute, or the `endif` will be reached.

index *value or expression*
: Sets the current index of a `for` or `foreach` loop to an absolute position, and then restarts the loop. If the index position is past the limit of the loop, or evaluates to 0, then the command acts like `break`. The `continue` command can be simulated with "`index %index + 1`", the `previous` command can be simulated with "`index %index - 1`", and the same item can be repeated over again with "`index %index`".

loop
: Repeats a `do`, `while`, `for` or `foreach` block.

nop
: Does nothing.

otherwise
: Used as a default for a `case` block when no case conditions are entered.

pack *symbol[:type or size] value/key=value ...*
: Sets an existing symbol or create a new global symbol, and packs it with a comma-delimited list of values. If the symbol exists, values will be appended to it.

push *symbol[:type or size] [key] value*
: Appends an optionally key-paired data value to a symbol list. If the list does not exist, it is created. Member values are normally single-quoted.

pause
: Used to guarantee a "scheduler" pause in the stepping engine.

previous
: Restarts a `for` or `foreach` loop using the prior element in the list. If already at the first index element, then the command will "break" the loop, exiting at the `loop` statement.

repeat

Restarts a `for` or `foreach` loop using the same index over again. This can be thought of as similar to `continue`.

restart

Restart the current labelled script `@section`. This is convenient since if you are in a defined function, you can still identify the parent script. Restart does NOT re-execute the `^init` handler, as the state of the script `@section` is already presumed to be initialized. For this reason, restart can also be used together with `if` to conditionally end an active `^init` segment early.

return

Returns from a script section subroutine (see `gosub`) or a defined script. Scripts also automatically return when the end of the current section, define, or event handler is reached.

set *symbol[:type or size] [assignment] value ...*

Sets an existing symbol or creates a new global symbol. If the symbol exists, its content will be replaced with the list of values. Alternately, an assignment operator can be used before the list of values. The two assignment operators supported are `:=` and `+=`. If `+=` is used, then `set` becomes the same as `add`.

strict *var ...*

Must be first statement and specifies strict compile mode. In strict compile mode, all symbols must be defined before use, and this is verified at compile time. Additional vars can be specified to be defined. At minimum, one must use "`strict error`" to enable the use of the internal `%error` symbol. Some application servers may define additional "internal" symbols which should be stated in strict mode, otherwise their use will generate a compile-time error.

until *condition*

Repeats a `do` block until the condition becomes true.

var *symbol[:type or size][=initial] ...*

Initializes one or more symbols. If `var` is used in a `define` block, the symbol is created in local scope.

while *condition*

While condition is true, enters a loop block. If the condition is false, then falls through the `loop` command, like `break`.

7 Copyright