DR Control Reference v04Mar2015

Table of Contents

DR Control Protocol and Language

DR Control Protocol

The DR control protocol has several key features:

- It is a **request-response** message protocol
- An **error reporting** mechanism is provided
- The message syntax is **text based**
- Messages may carry **data** payloads

DR receivers are controlled using a request-response pattern. A controller sends commands to the device as **request** messages, and receives in return **response** messages which contains information about success or failure of the command, and possibly a data payload. Flow control is simple - issue a request and wait for the response. Receipt of the response message signals that the device is ready for another request, and so on.

The object, or **target** of a command is a **property** of the device, or an **action** to be executed.

Two request-response modes are available, which differ in the nature of the response message:

- **Normal mode** - the response message contains an indication of success or failure, and a data payload (if required).
- **Verbose mode** - the response message contains an indication of success or failure, the name and address of the command target, and a data payload containing the current value of the command target (if required).

The verbose mode supports certain 3rd party control programming styles where the response message needs to be self-describing. Since both the name of the command target and its value are included, the response message can be processed independently of the request message.

DR commands are divided into 3 types:

- **Action command** - these are used to execute a pre-programmed **action** on the device.
- **Query command** - these are used to **read** data from the device in order to learn the value(s) of one of its properties.
- **Update command** - these are used to **send** data to the device in order to modify the value(s) of one of its properties.

Properties are single data values, or arrays of data values, that reflect the state of the device. Many properties are multi-valued. For example, with 6 channels, properties such as output level exist as an array of 6 values, one for each channel. In these cases an "address" qualifier is used to specify which particular value in a property's array of values is the target of the command. In most cases a wildcard address can be used to specify "all". Some properties are read-only, others

may be both read (queried) and written (updated). Details are found in the documentation for the command set.

Device properties include:

- Hardware related control settings for things like gain, phantom power, network setup etc. These are usually read/write properties.
- Hardware related status indications such as audio levels, serial number, programmable input status etc. These are usually read-only properties.
- "Soft" settings such as macro definitions, labels for inputs and outputs and run-time variables used in macros. These are usually read/write properties.

Actions change the state of the device in some way without requiring an explicit data input. An "address" qualifier may be required to specify exactly which property is affected (or referenced) by the action. Details are found in the documentation for the command set.

DR Control Language

This section concentrates on the language as used for 3rd party control of DR receivers over the serial and ethernet/TCP interfaces. To learn more about the use of the control language within macros, refer to the Macros and the DR Control Language topic.

The Control language is formally specified in the ABNF grammar document.

Contents:

Request Messages

Request messages are composed from a number of components, some of which are mandatory, some optional. The order in which these appear in the message is always the same. These components appear as **tokens**, which are simply characters or groups of characters defined in the grammar. The tokens contained in a message may be separated by one or more whitespace characters such as **space** or **tab.** A special meaning is reserved for the whitespace characters **carriage return** and **linefeed** when used to mark the end of a message. Otherwise whitespace has no significance and is ignored.

The building blocks for a request message are:

- **verbose mode** token (optional). An exclamation point character (!) in this position forces a verbose mode response message to be returned.
- **target** token (mandatory). This identifies the target of the command, some property or action. This may be a variable reference. The target token may contain **only** the alphabetic characters [a - z] and [A - Z]. **Exception:** Variable names must be surrounded by the '@' character, and may contain the alphabetic characters [a - z] and [A - Z] and the numeric characters [0 - 9]. No more than one target token may be present in a message. Example: `serial` or `@scene3@`
- **address** token (optional). This modifies the target of the command. It identifies a particular member of a array of values associated with some device property. It consists of an opening "**(**" character, then one or two groups of **digits**, and finally a closing "**)**" character. Only two digit groups are allowed, so at most a two dimensional space may be addressed. In the case where two digit groups are present, they must be separated by a **comma.** The numbers represented by these groups of digits must be **decimal** (base 10) numbers **greater than zero**. Addressing and indexing schemes start with the number 1. Some commands support the use of a **range** of numbers to specify an address, using 2 numbers separated by the colon character (:). In a further twist, some commands may allow one or both of the numbers to be replaced by a **wildcard** value, the character '***'.** This has the meaning that **all** of the members addressed by that index are being referenced. Also, references to **variables containing integer values** may be used to specify an address in place of a literal number. Variable names must be surrounded by the '@' character. No more than one address token may be present in a message. Examples: `(7)` or `(17,6)` or `(*)` or `(3,4:10)` or `(@which@)` or `(@row@,@col@)` or `(*,@col@)`
- **operator** token (optional). This modifies the meaning of the command from a simple action to either a **query** or an **update.** Two tokens are allowed, **?** and **=**. The query operator (?) signals that the request is a query for data, and the update operator (=) signals that the request carries data related to the desired action. No more than one operator token may be present.
- **data format** token (optional). This modifies the data format used in the transaction. Only one token is defined, **$**, which specifies the **hexidecimal encoded** format, and it must be preceded by either the query operator or the update operator. When following the query operator it signals that the data payload in the response message is expected to be encoded in hexidecimal format. When following the update operator it signals that the data which follows in the request message is encoded in hexidecimal format. When absent the default data encoding (described below) is in force. No more than one data format token may be present.
- **argument** token (optional). This contains data related to the request and may be present only when preceded by the update operator (and optionally the data format operator). There are several options for the argument in a request message:
    - **simple** data type. For example: `7` or `"Jury Box"`
    - **array** of simple data types. For example: `{1,0,1,0,1,0,1,0}`
    - **binary data** encoded in hexidecimal format, but **only** if preceded by the data format operator **$**. For example: `02FF1D22`

- **variable** reference. Variable names must be surrounded by the '@' character. For example: `@scene3@`
- **comparison expression** which evaluates to 0 or 1, using the >, <, >=, <=, == and **!=** operators. Variables, constants or arithmetic expressions may be compared. Expressions must be enclosed in parentheses. For example: `(@act@<3)` or `(@name@=="John")` or `(@gain@>=(@last@+2))`
- **logical expression** which evaluates to 0 or 1, using the logical **&&** (AND) or logical || (OR) operators. Variables, comparison expressions or logical expressions may be evaluated. Expressions must be enclosed in parentheses. For example: `(@lights@&&@sound@)` or `(@lights@||(@act@>1))`
- **arithmetic expression** which evaluates to an integer value, using the **+**, **-**, *****, **/** and **%** operators. Variables, constants, or arithmetic expressions may be used as terms. For example: `(@act@+1)` or `(@count@*(@stage@/3))`

See the Macros and the DR Control Language topic for details on variables and expressions. No more than one data token may be present in a message.

- **end of message** token (mandatory). This is a **carriage return** character indicating the end of the request message.

Verbose request

A request message can be prefixed with an exclamation point (!) character to force a "verbose" response message. The verbose response contains the name and address of the target (action or property being addressed), along with the current value of that target if it's a property. This supports certain 3rd party control programming styles where the response message resulting from a request needs to be self-describing so that it can be processed independently of the request message. For example, this query request (verbose mode):

```
!serial?<CR>
```

results in a response like this:

```
OK serial="5000101"<CRLF>
```

rather than a response like this (non-verbose mode)

```
OK "5000101"<CRLF>
```

Regarding verbose update requests, it's important to note that property values returned in the verbose response reflect the actual state of the property *after* the update has been attempted. This makes the verbose response reliable for refreshing 3rd party controller internal state or displays.

Request message types

- **Action request**

Action commands may or may not have an address operator, and trigger an action which doesn't need input data and doesn't return any output data. Some examples:

```
exit<CR>
```

This is a normal mode request to execute the "exit" action..

```
run(3)<CR>
```

This is a normal mode request to execute the "run" action on address 3.

```
!run(5)<CR>
```

This is a verbose mode request to execute the "run" action on address 5.

- **Query request**

Queries may or may not have an address operator. They specify the query operator **'?'** and trigger the return of the requested data in the response message. They are the means by which a controller can read the value of some property of the device or poll the device to get status information, perhaps to drive indicators. Some examples:

```
id?<CR>
```

This is a normal mode request for the value of the property "id".

```
rxalevel(3)?<CR>
```

This is a normal mode request for the value of the property "rxalevel" at address "(3)".

```
rxalevel(*)?<CR>
```

This is a normal mode request for the value of the property "rxalevel" at all addresses (wildcarded).

```
!rxalevel(3)?<CR>
```

This is a verbose mode request for the value of the property "rxalevel" at address "(3)". A verbose mode response will be received by the controller.

```
@scene@?<CR>
```

This is a normal mode request for the value of the run-time variable "@scene@".

- **Update request**

Updates may or may not have an address operator. They specify the update operator '=' and carry within them a data payload (the argument) for the device to process. They are

the means by which a controller can change the value of device properties such as the gain of an audio channel or control a feature like a pink noise generator. Some examples:

```
lcdbl=1<CR>
```

This is a normal mode request to update the property named "lcdbl" to the value 1.

```
rxalevel(3)=-5<CR>
```

This is a normal mode request to update the property named "rxalevel", at address "(3)" to the value -5.

```
rxalevel(*)={-3,0,0,0,0,0}<CR>
```

This is a normal mode request to update the property named "rxalevel", at address "(*)" (wildcarded, meaning *all*). The new values are supplied in the array of integers in the data payload. Property "rxalevel(1)" will be updated with the 1st integer in the array (-3), "rxalevel(2)" with the 2nd integer in the array (0), and so on. *A complete set of data must be provided when an update is directed at a wildcarded address*. If the "rxalevel" property is an array of 6 items, then an ERROR response will be received if an array of data is sent to the device whose size is not exactly 6. Likewise, if a variable reference is supplied as the argument the variable must contain an array of 6 integers. In the case of two dimensional arrays, the notation (*,*) used in addressing a property implies a responsibility for the controller to send the entire 2D array if data is transferred. Likewise, the notations (2,*) and (*,2) imply a responsibility for the controller to send the appropriate row or column slice of the 2D array.

```
!rxmute(2)=1<CR>
```

This is a verbose mode request to update the property named "rxmute", at address "(2)" to the value 1. A verbose mode response will be received by the controller.

```
@scene@=3<CR>
```

This is a normal mode request to update the run-time variable "@scene@" to the value 3.

Response Messages

Response messages are composed from a number of components, some of which are mandatory, some optional. The order in which these appear in the message is always the same. These components are tokens as described in the discussion of request messages above, and the same whitespace considerations apply.

The building blocks for a response message are:

- **status** token (mandatory). This indicates whether the request was successful or resulted in failure. The status token has two possible values, **OK** and **ERROR** . No more than one status token may be present in a message. Example: `OK`
- **target** token (optional). This is present only in verbose mode response messages, and identifies the target of the command, some property or action. This may be a variable reference. The target token may contain **only** the alphabetic characters [a - z] and [A - Z]. **Exception:** Variable names must be surrounded by the '@' character, and may contain the alphabetic characters [a - z] and [A - Z] and the numeric characters [0 - 9]. No more than one target token may be present in a message. Example: `serial` or `@scene3@`
- **address** token (optional). This is present only in verbose mode response messages, and modifies the target of the command. It identifies a particular member of a array of values associated with some device property. It consists of an opening "**(**" character, then one or two groups of **digits**, and finally a closing "**)**" character. Only two digit groups are allowed, so at most a two dimensional space may be addressed. In the case where two digit groups are present, they must be separated by a **comma.** The numbers represented by these groups of digits must be **decimal** (base 10) numbers **greater than zero**. Addressing and indexing schemes start with the number 1. Some commands support the use of a **range** of numbers to specify an address, using 2 numbers separated by the colon character (:). In a further twist, some commands may allow one or both of the numbers to be replaced by a **wildcard** value, the character '***'.** This has the meaning that **all** of the members addressed by that index are being referenced. Also, references to **variables containing integer values** may be used to specify an address in place of a literal number. Variable names must be surrounded by the '@' character. No more than one address token may be present in a message. Examples: `(7)` or `(17,6)` or `(*)` or `(3,4:10)` or `(@which@)` or `(@row@,@col@)` or `(*,@col@)`
- **operator** token (optional). This is present only in verbose mode response messages, where the update operator (=) signals that the response message carries data representing the value of the target of the command. Only the update operator may appear in a verbose mode response message.
- **data format** token (optional). This modifies the data format used in the transaction. Only one token is defined, **$**, which specifies the **hexadecimal encoded** format. When present it signals that the data payload in the response message is encoded in hexadecimal format. No more than one data format token may be present.
- **data** token (optional). This contains data requested by the controller in a query message or being returned in response to a verbose mode request message. In a verbose mode response the data is preceded by the update operator (and optionally the data format operator). There are only 3 options for the data payload in a response message:
    - **simple** data type. For example: `7` or `"Jury Box"`
    - **array** of simple data types. For example: `{1,0,1,0,1,0,1,0}`
    - **binary data** encoded in hexidecimal format, but **only** if preceded by the data format operator **$**. For example: `02FF1D22` No more than one data token may be present in a message.

Note: Response messages return only data **values**, expressed as simple types, arrays of simple types, or hexadecimal encoded binary data. When variables or expressions are used to convey data in a verbose request message, the variable or expression is **evaluated** and the result is used. For example, this verbose request:

!rxalevel(3)=(2+2)<CR>

will result in this verbose response:

OK rxalevel(3)=4<CRLF>

In other words, verbose response messages return only data **values** for addresses and properties.

- **end of message** token (mandatory). This is a **carriage return / linefeed** character pair indicating the end of the message.

Response message types

- **Action response**

  The response contains a status code indicating success or failure of the request processing. No data token is present. An example:

  OK<CRLF>

  Normal mode request processing succeeded. The action was executed.

  OK run(2)<CRLF>

  Verbose mode request processing succeeded. The name of the action executed ("run") is returned along with the address specified in the request (2).

  OK rxmute(2)=1<CRLF>

  Verbose mode request processing succeeded. The name, address and value of the property affected by the "rxmutetog(2)" action is returned.

- ERROR<CRLF>

  Request processing failed, the action was not executed. The status code ERROR is returned to the controller, which **should** recognize this status code and take appropriate action. The most common reasons why a action command request might fail are:

  - the target of the command is misspelled or nonexistent
  - the address is out of range

- **Query response**

  The response contains a status code indicating success or failure of the request processing. A data payload is present. Examples:

  ```
  OK 22<CRLF>
  ```

  Normal mode request succeeded. The requested data (22) is returned in the message.

  ```
  OK rxalevel(3)=0<CRLF>
  ```

  Verbose mode request processing succeeded. The name/address of the property queried is returned along with its current value of 0.

  ```
  ERROR<CRLF>
  ```

  Request processing failed. The status code ERROR is returned to the controller, which **should** recognize this status code and take appropriate action. The most common reasons why a query request might fail are:

  - the target of the command is misspelled or nonexistent
  - the address is out of range

- **Update response**

  The response contains a status code indicating success or failure of the request processing. A data payload is present only it the request was made in verbose mode. Examples:

  ```
  OK<CRLF>
  ```

  Normal mode request processing succeeded.

  ```
  OK rxalevel(3)=5<CRLF>
  ```

  Verbose mode request processing succeeded. The name/address of the property updated is returned along with its new value of 5.

  ```
  ERROR<CRLF>
  ```

  Request processing failed. The status code ERROR is returned to the controller, which **should** recognize this status code and take appropriate action. The most common reasons why an update request might fail are:

  - the target of the command is misspelled or nonexistent
  - the address is out of range
  - the data sent in the update request message is ill formed or out of range

Data types

The following data types are supported:

- **quoted string**
- **integer**
- **array of integer**
- **floating point**
- **array of floating point**
- **binary**

Variables are capable of storing values of all types **except** the binary type.

It is worth noting that many properties are naturally thought of as **boolean** types. These items use the **integer** type and but are limited to values **0** ("false" or "disabled") and **1** ("true" or "enabled"). Likewise, logical and comparison expressions evaluate to integer values, either **0** or **1**.

Data type formats

- **Quoted string type**

  A quoted string token consists of an opening **double quote** character ("), followed by zero or more characters, and finally a closing double quote character. Quoted strings may contain any printable ASCII character **except** the double quote character (") used to enclose them, and the backslash character (\) used as an "escape" character. To embed double quote characters or backslashes within a string they must be "escaped" by preceding them with a **backslash** character: `\"` or `\\`. The special escaped forms `\r` (carriage return), `\n` (newline) and `\t` (tab) are also recognized. Non-printable ASCII characters may be expressed using the hexadecimal escaped form `\xHH` where `HH` is any 2-digit hexadecimal number. Whitespace is ok within a quoted string, and is preserved. Quoted string tokens may not exceed 127 characters in length, exclusive of the enclosing double quote characters. Strings may be empty. Some examples:

  ```
  "Chairman"

  ""

  "The \"Lost\" Sheep"

  "serial?\r"

  "id?\x0D"
  ```

- **Integer type**

  An integer token consists of an optional **sign** character (either "+" or "-") and a series of one or more decimal **digit** characters (0,1,2,3,4,5,6,7,8,9 and 0). No other characters are permitted. The maximum number of characters in an integer token, including any sign character, is 15. This limitation is general and has nothing to do with the range of **values** allowed for a particular property. Some examples:

  ```
  999
  ```

  ```
  -22
  ```

- **Array of integer**

  An array of integer token consists of a series of one or more **comma delimited** integer tokens enclosed in matching **braces.** Whitespace separating braces, integer tokens, and commas is ignored. The maximum size of an integer array is 64 items. Arrays may **not** be empty. Some examples:

  ```
  {0,-10,22,0,0,50}
  ```

  ```
  {1750}
  ```

- **Floating point**

  A floating point token consists of an optional **sign** character (either "+" or "-"), a series of zero or more decimal **digit** characters (0,1,2,3,4,5,6,7,8,9 and 0), a **mandatory decimal point** character ("."), and finally another a series of zero or more decimal **digit** characters. Exponential notation is not supported. The maximum number of characters in a floating point token, including any sign character and the mandatory decimal point character, is 15. This limitation is general and has nothing to do with the range of **values** allowed for a particular property. Some examples:

  ```
  0.6242
  ```

  ```
  -172.0
  ```

- **Array of floating point**

  An array of floating point token consists of a series of one or more **comma delimited** floating point tokens enclosed in matching **braces.** Whitespace separating braces, floating point tokens, and commas is ignored. The maximum size of an floating point array is 64 items. Arrays may **not** be empty. Some examples:

  ```
  {0.0,-10.6,-22.651,0.0,0.0}
  ```

  ```
  {-1.000175}
  ```

- **Binary**

  A binary data token consists of a series of **hexadecimal** digit character **pairs** (0...9 and A...F). Each pair encodes one byte of binary data. The maximum size of an encoded binary block is 96 bytes in a request message to a device (which requires 192 hex digits total when encoded). Binary data tokens may be used **only** when the data format for the message has been set to **hexadecimal encoded** by the presence of the data format token **$.** An example

  ```
  $00A7C2990014
  ```

  The hexadecimal encoded data format and the binary data type are used by certain software programs for special purposes. "Third party" remote control applications don't need to use the binary type, it is described here for completeness.

Macros and the DR Control Language

Macros are simply a series of instructions - expressed using the DR Control language. The elements of the control language are:

- Commands - These are the familiar native commands of the DR receiver, as documented in the "Command Set" in the reference manual or in the Wireless Designer Help system. Ultimately, the purpose of the macro will be to issue commands to the device in order to make it "do" something, or to read out its current settings for use by external controllers.
- Variables - These are user defined global storage, used to pass data within a macro, or between Macros. Variables make it possible for Macros to have a "memory" of past actions, or to capture data for use within another macro, at some other time. Arithmetic, comparison, and logical operations can be performed with variables.
- Expressions - These are used to compute logical or arithmetical results using variables or constant values. Expressions make it possible to perform arithmetic, create loops, or make decisions using conditional statements.
- Loops - These are "while-do" statements of the sort seen in many other programming languages. Loops make it possible for a particular command to be run multiple times as long as the state of some device property or the value of some variable meets a specified condition.
- Conditionals - These are "if-then-else" statements of the sort seen in many other programming languages. Conditionals make it possible for a macro to choose between alternative actions on the basis of the current state of some device property or the value of some user defined variable.

Commands, loops and conditionals are statements, and can stand alone as a macro "line" or instruction. Variables and expressions play a supporting role, with variables commonly used in expressions and both often found in update commands as the "argument". Loops and conditionals contain both expressions defining their "condition" and commands to be executed as their "actions" if the condition is met.

Macros are "run" (executed) in response to some triggering event, such as a serial command or the pressing of a push button connected to a programmable logic input pin. Applications such as room combining, courtroom sound systems, and teleconferencing rely on Macros to make system setup changes "on the fly" in response to button panel activity or serial commands from 3rd party control systems.

Macros may include up to 64 "lines", each line containing one or more instructions, or statements. Multiple statements must be separated by a ';' (semicolon) character. These maximum length of a macro line is 115 characters. However, a long statement can be divided between multiple macro lines using "line continuation". An underscore character '_' at the end of a line indicates a *continuing* statement. A **maximum of 8 lines** can be joined together this way into a single statement. Execution occurs when the statement is complete, which is indicated by a final line which is *not* continued. For example:

```
bar(1,*)={-70,-70,-70,-70,-70,-70,-70,-70,-70,-70,-70,-70,-70,-70,-70,-70,_
-70,-70,-70,-70,-70,-70,-70,-70,-70,-70,-70,-70,-70,-70,-70,-70,-70,-70,_
-70,-70,-70,-70,-70,-70,-70,-70,-70,-70,-70,-70,-70,-70,-70,-70,-70,-70}
```

Here an array consisting of 48 elements is spread across three lines so that the line length limit is not exceeded. Note the underscore character at the end of the first two lines, which indicates that the statement is to be continued on the next line. The third and final line is not continued so the entire statement is executed at that point.

Note: the following must *not* be interrupted by a line continuation:

- **variables**, for example: `@foo@` bracketed by the "at" character (**@**)
- **quoted strings**, for example: `"East Room #1"` bracketed by the double quote character (**"**)
- **action** expressions, for example: `` `rxalevel(3)=@result@` `` bracketed by the backtick character (**`**)

The continuation will fail if this is done. Line continuation is available in firmware versions 1.4.4 and higher.

Commands

Briefly, commands executed within Macros consist of a mix of elements in the following order:

- **verbose mode** token - an exclamation point character (!) in this position causes a verbose mode response message to be sent to the communications port designated as the "macro verbose response" port. This allows the output of commands executed in a macro to be monitored by 3rd party control applications.
- **target** - this identifies a property of the device like "id", or an action to be performed, like "run a macro". It may be a variable referenced by name. Mandatory.
- **address** - this modifies the target of the command, by identifying a particular member of a set of things - **which** input gain, or **which** macro to run. The documentation for a command specifies whether or not an address a required, and the form it should take.
- **operator** - if present, modifies the meaning or intent of the command. The query operator '?' signals that the command is a query for the value of some device property. The update (assignment) operator '=' signals that the command includes data (the argument) which is to be assigned to some device property. If no operator is present then the command is for a some action to be performed. The documentation for a command will specify which operators, if any, may be used.
- **argument** - This is the value or data payload needed for "update" commands. The documentation for a command will specify what kind of argument, if any, is required.

For example:

- `rxalevel(3)?`
  The target "rxalevel" means receiver audio level value, the address "3" means channel 3, the '?' operator means it's a query for the value of the target property.

- `rxalevel(3)=2`
  The target "rxalevel" means receiver audio level value, the address "3" means channel 3, the '= ' operator means it's an update of the target property, the argument "2" is the new value.
- `run(3)`
  The target "run" means "run a macro", the address 3 means macro number 3, the absence of an operator means it's an order to perform the target action.

Special built-in commands for use in Macros:

- `exit` - this command forces the macro processor to stop executing the macro. It's normally used within an conditional or loop instruction. For example:
  `if(@foo@<10)then`rxmute(*)=@bar@`else`exit`
- `sendstr` - this command allows an arbitrary string to be sent over the RS232 or TCP port of a device. The port is specified in the address value. The string to be sent is given as a quoted string argument or a string value stored in a variable. For example, `sendstr(2)=@foo@` could be used in a macro in any device to send the string stored in variable @foo@ to port 2 (the TCP port local to that device).
  Note: The maximum length of a command sent using `sendstr` is 255 characters, which is the maximum length of a *string variable*. However, if using a *quoted string* (e.g. "FF0122;") as the argument, it must be remembered that the maximum length of quoted strings is only 127 characters. Strings longer than 127 characters must be built up dynamically using variables and the string format and concatenation operators. For example:

  ```
  @data@=`rxameter(*)?`
  @crlf@="\r\n"
  @str@=format(@data@,"%d")
  @str@=@str@:@crlf@
  sendstr(1,2)=@str@
  ```

  Here an array value is dynamically captured in a variable, then formatted as a string in variable `@str@`. A line ending is concatenated, and the string is then sent.

Verbose mode commands in Macros: If a "verbose mode" command is executed within a macro, the response can be directed to a communications port rather than suppressed. Verbose mode commands are those prefixed with the '!' (bang) character, for example: `!rxalevel(3)=0`. This can be useful for 3rd party control panel applications that rely on feedback from command execution to update their visual controls. If connected to the "macro verbose response" port the application can stay synchronized with the device state when a macro is run. Either the RS232 port or the TCP ports may be used for this purpose, selected with the `macvrport` command. This command can be issued from within any macro, but the "run on powerup" macro is the natural choice if it needs to be configured only once.

Variables - Part 1

Variables are user defined, and consist of a variable "name" enclosed within a pair of '@' characters. The name may include any of the printable characters except '@'. The maximum length allowed for variable names is 15 characters (not including the enclosing '@' characters). Examples: `@stage2@` or `@mic_muted@`. Variable names are case sensitive! For example, `@foo@` is not the same variable as `@Foo@` or `@FOO@`.

Variable types: Variables can store the following data "types":

- **integer** such as `42`
- **array of integer** such as `{1,2,3,4}`
- **string** such as `"East hallway"`
- **boolean** really an integer, interpreted as "false" when the integer value is `0` and as "true" for any other value
- **array of boolean** really an array of integer, with values of either `1` or `0`, such as `{1,0,1,0}`

Initializing variables: A variable is "initialized" the first time (after powerup) that the macro command processor encounters an instruction assigning a value to it. For instance, running a macro containing the instruction `@foo@=42` will cause the variable `@foo@` to be created if it doesn't already exist. The variable `@foo@` is said to be "initialized" at this point with integer value 42. The instruction `@bar@={1,2,3,4,5}`, on the other hand, would initialize `@bar@` with an array of integer value with 5 elements. **Important:** variables are "volatile", which means that they are lost when power is removed from the device. When the device is powered back up, the variables will be initialized all over again as described above. The "run on powerup" macro is a good place for initializing variables shared between multiple Macros.

String variable length: The maximum length of an string variable is **255** characters.

Array variable length: The maximum length of an array variable is **64** elements. The length of a particular array variable can be determined using the **length** operator, which returns the number of elements in the array. For example:

- `@foo@=length(@bar@)`

Accessing elements in array variables: Individual elements of a variable of type **array of integer** or **array of boolean** can be accessed using a "subscript" notation. For example, `@foo@[2]` accesses the 2nd element in the array. The location of the element must be a an integer value between **1** and **64** enclosed within a pair of '[' and ']' characters. The location may also be specified as a variable with an integer value, or even by an arithmetic expression. For example:

- `@foo@[2]`
- `@foo@[@bar@]`
- `@foo@[@bar@[3]]`
- `@foo@[@bar@+1]`

Important: It is an error to address array elements that don't exist, such as in `@foo@[5]`, when `@foo@` has only 4 elements. It is also an error to use the subscript notation with any variable whose type is not an array of integer or array of boolean (e.g. strings, simple integers).

Expressions

Arithmetic expressions: Basic **integer** arithmetic operations may be performed using variables and constant values. Addition ('+'), subtraction ('-'), multiplication ('*'), division ('/') and modulo division ('%') operations are supported, but remember that the variable needs to hold an integer value for the result to make sense. Use pairs of parentheses, '(' and ')', to group terms in complicated expressions to get the proper result. The **value** of an arithmetic expression is the integer result of the operations. For example:

- `@foo@+2`
  Value is the sum of 2 and @foo@
- `(2+@foo@[2])*4`
  Value is the sum of 2 and the 2nd element of array @foo@, multiplied by 4
- `@foo@-2+@bar@`
  Value is @bar@ added to the difference between @foo@ and 2
- `@foo@/(@bar@+1)`
  Value is @foo@ divided by the sum of @bar@ and 1
- `@foo@*@foo@`
  Value is the square of @foo@
- `@foo@=@foo@+1`
  Value is @foo@ incremented by 1

Concatenation expressions: String values can be concatenated (joined) by using the concatenation operator '**:**'. The value of a concatenation expression is another string value. For example:

- `@foo@:" Two"`
  If @foo@ holds the string value "Act" then the value is "Act Two".
- `"Scene: ":@foo@`
  If @foo@ holds the string value " Two" then the value is "Scene Two".
- `@foo@:@bar@`
  If @foo@ holds the string value "Act" and @bar@ holds the value " Two" then the value is "Act Two".
- `"Next: ":@foo@:@bar@`
  If @foo@ holds the string value "Act" and @bar@ holds the value " Two" then the value is "Next: Act Two".

Concatenation expressions must include at least one variable reference. The concatenation of two literal string values is not supported, an error will result. Literal values or variables used in concatenation expressions **must** hold string values, or an error will result. If the result of a string

concatenation exceeds the maximum string variable length of 255 characters it will be truncated to that length.

String formatting expressions: Formatted string values can be generated by using the format operator "format". This operator requires two comma separated parameters enclosed in parentheses, given in the form 'format(variable,"format specifier")' where:

- **variable** is a variable reference (name) containing a string or integer value to be formatted.
- **format specifier** is a double-quoted literal string value containing a C language style format specifier and optional literal text. Only format specifiers **"%d"** (decimal integer), **"%x"** (lowercase hexadecimal integer), **"%X"** (uppercase hexadecimal integer) and **"%s"** (string) are supported. C language style flags and width sub-specifiers may be applied to integer values. **Note:** When the "%d" format specifier is used and the variable holds an array of integers, the array will be formatted in the array syntax (comma separated values enclosed in curly braces).

The value of a format expression is a string value which can be assigned to a variable, or used in a concatenation expression. For example:

- `@result@=format(@foo@,"Title: %s")`
  If @foo@ holds the string value "Macbeth" then the value assigned to @result@ is "Title: Macbeth".
- `@result@=format(@foo@,"run(%d)")`
  If @foo@ holds the integer value 7 then the value assigned to @result@ the value is "run(7)".
- `@result@=@title@:format(@foo@," - Act %d")`
  If @title@ holds the string value "Macbeth" and @foo@ holds the integer value 2 then the value assigned to @result@ the value is "Macbeth - Act 2".
- `@result@=format(@foo@,"rxmute(*)=%d")`
  If @foo@ holds the array of integer value {1,2,3,4,5,6,7,8} then the value the value assigned to @result@ is "rxmute(*)={1,2,3,4,5,6,7,8}".
- `@result@=format(@foo@[3],"%d")`
  If @foo@ holds the array of integer value {1,2,3,4,5,6,7,8} then the value the value assigned to @result@ is "3".
- `@result@=format(@foo@,"AA %03d\r")`
  If @foo@ holds the integer value 7 then the value the value assigned to @result@ is "AA 007\r" where '\r' indicates an ASCII carriage return (hex 0D) and the specifier '%03d' is the decimal integer specifier modified to force the value 7 to be formatted as a zero padded 3 digit number (sub-specifiers are flag '0', width '3' - consult a C language reference to learn more).

The type of the variable value and the C language style format specifier must agree or an error will result. If the result of a format expression exceeds the maximum string variable length of 255 characters it will be truncated to that length.

Comparison expressions: Comparisons can be made between variables and constant values. An integer variable can be compared to a literal integer value or another integer variable. A string variable can be compared to a another string variable, or tested for equality with a string literal. Equality ('== '), inequality ('!= '), less than ('<'), greater than('>'), less than or equal ('<= ') and greater than or equal ('>= ') comparisons are supported. For string types the comparisons are"lexical" in nature - they are compared character by character, and characters are judged by their ASCII code value. So "aaa" is equal to "aaa" but not equal to "aaaa", and "abc" is "less than" "bbc" because 'a' comes before 'b' in the ASCII table. The **value** of a comparison expression is a boolean type: "true" if the condition is met or "false" if the condition is not met. For example:

- `@foo@<10`
  True if @foo@ is less than 10
- `@foo@[2]<10`
  True if the 2nd element of array @foo@ is less than 10
- `@foo@>@bar@`
  True if @foo@ is greater than @bar@ (works for integer or string values)
- `"test"==@foo@`
  True if @foo@ contains the string value "test"
- `@foo@ != 17`
  True if @foo@ is not equal to 17
- `@foo@==@bar@`
  True if @foo@ is identical to @bar@ (works for string, integer and array of integer types)
- `@foo@[2]==@bar@`
  True if the 2nd element of array @foo@ is identical to @bar@ (works only if @foo@ is an array of integer (or boolean) type and @bar@ is an integer (or boolean) type.)
- `@foo@!=@bar@`
  True if @foo@ is not identical to @bar@ (works for string, integer and array of integer types)

Since an arithmetic expression has an integer value, it can substitute for an integer within a comparison expression. For example:

- `@foo@<(@bar@+10)`
  True if @foo@ is less than the sum of @bar@ and 10
- `((@foo@*2)+1)==@bar@`
  True if 1 added to the product of @foo@ and 2 is equal to @bar@

Note that the arithmetic expression must be enclosed with parentheses for this to work properly.

Logical expressions: Variables can be used in logical expressions. Any variable with an integer value can be treated as a boolean type, either "true" or "false". The convention is that the integer value zero (0) means "false" and any nonzero value means "true". Variables that are used in this way can then be compared using the "logical" operations "AND" ('&&'), "OR" ('||') and "NOT" ('!'). The **value** of a logical expression is also a "boolean" type. For example:

- `@foo@`
  True if @foo@ is nonzero
- `!@foo@`
  True if NOT @foo@ is nonzero (that is, @foo@ is zero)
- `@foo@&&@bar@`
  True if both @foo@ and @bar@ are nonzero
- `@foo@||@bar@`
  True if either @foo@ or @bar@ are nonzero (or both)
- `!@foo@&&!@bar@&&@sam@`
  True if NOT @foo@ is nonzero and NOT @bar@ is nonzero and @sam@ is nonzero (that is, @foo@ and @bar@ are zero, and @sam@ is nonzero)

Going further, more complicated expressions can be formed by grouping terms using parentheses to get the proper result. Also, since a comparison expression has a boolean value, it can substitute for a boolean variable within a logical expression. For example:

- `@foo@&&(@bar@||@sam@)`
  True if @foo@ is nonzero and either @bar@ or @sam@ are nonzero
- `(@foo@||@bar@)&&(@sam@<50)`
  True if @foo@ or @bar@ are nonzero and @sam@ is less than 50.
- `(@foo@!="stop")&&!@bar@`
  True if @foo@ does not contain the string value "stop" and @bar@ is zero

Note that the comparison expression must be enclosed with parentheses for this to work properly.

Variables Part 2

Assigning a value to a variable: A value can be assigned to a variable in several ways. Whatever the method, the assignment instruction begins with the variable name followed by the assignment operator '=', like this: `@foo@=`. Next comes the expression for the value to be stored in the variable. The possibilities are:

- **Literal value:** A "literal" integer or string value can be assigned. The value is expressed using the normal syntax. For example:
  - `@foo@=142`
  - `@foo@={4,4,4,4}`
  - `@foo@="hello"`
- **Query output:** The value of some device "property" can be assigned to a variable by capturing the output of the appropriate "query" command. The command is expressed using the normal syntax, enclosed within a pair of '`' (backtick) characters. Don't forget the '?' at the end of the query! For example:
  - `@foo@=\`rxmute(*)?\``
  - `@foo@=\`rxalevel(3)?\``
  - `@foo@[2]=\`rxalevel(3)?\``
  - `@foo@=\`serial?\``

- **Another variable:** The value of another variable can be assigned. For example:
  - `@foo@=@bar@`
- **Arithmetic expression:** The value of an arithmetic expression can be assigned. For example:
  - `@foo@=(@bar@+!@sam@)`
  - `@foo@=((@bar@/2)+1)`

  Since the value of a arithmetic expression is an integer type @foo@ will hold result as such. **Note:** It is not mandatory to enclose the entire arithmetic expression in parentheses when assigning its value to a variable. Use parentheses to group terms in complicated expressions to get the proper result.

- **Concatenation expression:** The value of a concatenation expression can be assigned. For example:
  - `@foo@=@bar@:@sam@`
  - `@foo@=@bar@:"hello"`
  - `@foo@=@bar@:"hello":@sam@`

  Since the value of a concatenation expression is an string type @foo@ will hold result as such. **Note:** Do *not* enclose a concatenation expression in parentheses when assigning its value to a variable.

- **String formatting expression:** The value of a string formatting expression can be assigned. For example:
  - `@foo@=format(@bar@,"-- %s --")`
  - `@foo@=format(@bar@,"rxalevel(*)=%d")`
  - `@foo@=@bar@:format(@sam@,"%d")`

  Since the value of a string formatting expression expression is an string type @foo@ will hold result as such. **Note:** Do *not* enclose a string formatting expression in parentheses when assigning its value to a variable.

- **Comparison expression:** The value of a comparison can be assigned. For example:
  - `@foo@=(@bar@!="stop")`
  - `@foo@=(@bar@<(@sam@/2))`

  Since the value of a comparison expression is a boolean type @foo@ will hold either the integer value 1 (true) or 0 (false). **Note:** the overall comparison expression must be enclosed in parentheses as shown for the assignment to work. Use parentheses to group terms in complicated expressions to get the proper result.

- **Logical expression:** The value of a logical expression can be assigned. For example:
  - `@foo@=(@bar@&&!@sam@)`
  - `@foo@=(@bar@&&(@sam@||(@scene@<5)))`

  Since the value of a logical expression is a boolean type @foo@ will hold either the integer value 1 (true) or 0 (false). **Note:** the overall logical expression must be enclosed in parentheses as shown for the assignment to work. Use parentheses to group terms in complicated expressions to get the proper result.

- **Length of array variable:** The length of an array variable can be assigned with the **length** operator. For example:
    - `@foo@=length(@bar@)`

  @foo@ will hold an integer value: the number of elements in variable @bar@.

Variables as command arguments: Once variables are initialized, they can be directly assigned to some device property. For example:

First, initialize the variable in a macro:

```
@foo@={0,0,0,0,0,0}
```

and then somewhere else in the same macro, or in another macro...

```
rxalevel(*)=@foo@
```

The "argument" to the `rxalevel` command is now a variable name, rather than a literal integer array, such as `{0,0,0,0,0,0,0,0}`. Now the value of `@foo@` can be controlled by one macro, and used by one or more other Macros when setting the receiver audio levels. Variables can also be used within an integer array. For example:

First, initialize the variable in a macro:

```
@foo@=-3
```

and then somewhere else in the same macro, or in another macro...

```
rxalevel(*)={0,@foo@,0,0,0,0}
```

In this case the 2nd array element takes on the value of `@foo@`. Any or all of the array members may be represented by a variable.

Variables in command addresses: The value of a variable can be assigned to one or all of the "address" elements of command. For example:

First, initialize the variable in a macro:

```
@foo@=2
```

and then use it somewhere else in the same macro, or in another macro...

```
rxalevel(@foo@)=0
```

In this case the value of variable `@foo@` determines *which* receiver channel is assigned a gain of 0. This is a powerful device control technique. Here's another example:

```
rxalevel(@foo@[2])=0
```

In this case the variable `@foo@` contains an array of integers, and the value of the 2nd element of that array is used as the address.

More on variable initialization: To prevent trouble with "uninitialized" variables shared between multiple Macros, it is often a good idea to initialize them all at once in a macro that is guaranteed to run before any other macro. This is one of the purposes of the "run on powerup" macro, which provides a good place for one-time variable initializations. It will be run exactly once, at powerup. For preset specific initializations, a "preset run on recall" macro can be designated for each preset in the device. This macro will be run every time the preset is recalled.

Number of variables: The maximum number of variables that may be initialized in a device is **32** .

Loops

A loop is the familiar "while-do" statement used to execute one or more instructions - based on the current value of a comparison, a logical expression, or a combination of both. This expression is the "condition" and must be enclosed in parentheses. The value of the condition is a boolean type, and whether or not the loop continues or terminates depends on whether the condition is "true" or "false". The "actions" are one or more simple commands or update commands, enclosed in '`' (backtick) characters. Multiple instructions must be separated by a ';' (semicolon) character. The special keywords "while" and "do" must be lowercase. For example:

- `while(@foo@<=6)do`rxalevel(@foo@)=0;@foo@=@foo@+1`
  While @foo@ is less than or equal to 10, set the audio level of receiver channel @foo@ to zero, then increment @foo@ and repeat. Terminate when the value of @foo@ is 7. The starting value of @foo@ has been set somewhere else.
- `@foo@=1;while(@foo@<=10)do`rxalevel(@foo@)=0;@foo@=@foo@+1`
  Same as above except @foo@ is initialized on the same macro line as the loop instruction. Note the separating semicolon.
- `@foo@=rxalevel(@bar@);while (@foo@<0)do`@foo@=rxalevel(@bar@);@foo@=@foo@+1;rxalevel(@bar@)=@foo@`
  Initialize @foo@ to the audio level value for some receiver channel specified by @bar@. While this level is less than zero, increment it by one. Terminate when the gain is zero.
- `@foo@=rxalevel(@bar@);while (@foo@<0&&@x@!=1)do`@foo@=rxalevel(@bar@);@foo@=@foo@+1;rxalevel(@bar@)=@foo@;@x@=rxmute(3)`
  Same as above - *except* that the loop terminates if receiver channel 3 is muted.

And so on. The maximum length of a command(s) supplied as the "action" is 63 characters. Commands given as an "action" of the loop must be enclosed within backtick characters - this is the character usually found on the same key as the tilde ('~'), in the upper left corner of your keyboard.

The use of loops allows device properties and variables that are arrays of values to be conveniently and efficiently accessed. A single loop instruction can replace many individual instructions.

Conditionals

A conditional is the familiar "if-then-else" statement used to control *what* happens - based on the current value of a comparison, a logical expression, or a combination of both. This expression is the "condition" and must be enclosed in parentheses. The value of the condition is a boolean type, and the action (if any) to be taken depends on whether the condition is "true" or "false". The "actions" are one or more simple commands or update commands, enclosed in '`' (backtick) characters. Multiple instructions must be separated by a ';' (semicolon) character. The special keywords "if", "then" and "else" must be lowercase. For example:

- `if (@foo@) then `run(3)``
  If @foo@ is nonzero then run macro 3, otherwise do nothing.
- `@foo@=`rxmute(5)?`;if (@foo@ == 5) then `rxalevel(3)=0` else `rxalevel(3)=@bar@``
  If receive channel 5 is muted, then set channel 3 input gain to 0, otherwise set it to the value of @bar@. Variable @foo@ is initialized with the mute status first.
- `if (@foo@||@bar) then `run(3)` else `rxalevel(*)={0,0,0,0,0,0}``
  If @foo@ is nonzero or @bar@ is nonzero then run macro 3, otherwise set all receiver audio levels to 0.

And so on. The maximum length of a command(s) supplied as an "action" is 63 characters. Commands given as an "action" of the conditional must be enclosed within backtick characters - this is the character usually found on the same key as the tilde ('~'), in the upper left corner of your keyboard.

The use of conditional logic gives the control language its power. Variables shared between Macros can drive behavior that is adapted to current circumstances - which preset is active, the state of some programmable input, or the current value of a device setting like output gain or mute status.

Loops and Conditionals combined

Loops and conditionals may be combined (nested) in certain ways:

- **Conditional inside a loop instruction** - for example:
  `while(@foo@<5)do`if(@bar@)then\`rxmute(@foo@)=1\`else\`rxmute(@foo@)=0\`;@foo@=@foo@+1``

- **Loop inside a conditional instruction** - for example:
  ```
  if(@bar@)then`while(@foo@<5)do\`rxmute(@foo@)=1;@foo@=@foo@+1\`else`exi
  t`
  ```

Important: note that in both cases the "action" part of the "inner" instruction is treated specially - the backtick characters ('`') which enclose it need to be "escaped" with a backslash character ('\') since they are nested within another pair of backticks. This also applies to variable assignments made by invoking a built-in command. Do this:

```
if(@foo@)then`@bar@=\`rxalevel(*)?\``
```

not this:

```
if(@foo@)then`@bar@=`rxalevel(*)?``.
```

Otherwise the macro processor will get confused about which backtick character goes with which.

No other combination is possible. Loops may not be contained within another loop instruction, nor may a conditional be contained within another conditional instruction.

ABNF Grammar for DR Control Language

```
OCTET          = %x00-FF   ; any 8-bit data
CHAR           = %x01-7F   ; any US-ASCII character except NUL (1 - 127)
UPALPHA        = %x41-5A   ; any US-ASCII uppercase letter "A".."Z"
LOALPHA        = %x61-7A   ; any US-ASCII lowercase letter "a".."z"
DIGIT          = %x31-39   ; any US-ASCII digit "0".."9"
LF             = %x0A      ; US-ASCII LF, linefeed (10)
CR             = %x0D      ; US-ASCII CR, carriage return (13)
SP             = %x20      ; US-ASCII SP, space (32)
HT             = %x09      ; US-ASCII HT, horizontal tab (9)
DQUOTE         = %x22      ; US-ASCII double-quote mark (34)
BSLASH         = %5C       ; US-ASCII backslash (92)
QCHAR          = %x01-21 / %x23-5B / %x5D-7F; any CHAR except DQUOTE and BSLASH
VARCHAR        = %x20-3F / %x41-7E; any printable CHAR except "@"
WS             = SP / HT
SIGN           = "-" / "+"
OFFSET         = 1*DIGIT / "*"
RANGE          = 1*DIGIT ":" 1*DIGIT
ALPHA          = UPALPHA / LOALPHA
ALPHANUM       = ALPHA / DIGIT
HEX            = "A" / "B" / "C" / "D" / "E" / "F" / "a" / "b" / "c" / "d" / "e" / "f" / DIGIT
HEXNUM         = "x" 2HEX
ESCSEQ         = BSLASH ("n" / "r" / "t" / BSLASH / DQUOTE / HEXNUM)
STR_TOK        = ALPHA *(ALPHANUM)
VAR_TOK        = "@" 1*(VARCHAR) "@"
INT_TOK        = *1SIGN 1*DIGIT
FLT_TOK        = *1SIGN *DIGIT "." *DIGIT ; note that bare "." is valid
QSTR_TOK       = DQUOTE *(QCHAR / ESCSEQ) DQUOTE
CRLF           = CR LF
OK_TOK         = %x4F %x4B ; uppercase string "OK"
ERROR_TOK      = %x45 %x52 %x52 %x4F %x52 ; uppercase string "ERROR"
IF_TOK         = "i" "f"
THEN_TOK       = "t" "h" "e" "n"
ELSE_TOK       = "e" "l" "s" "e"
WHILE_TOK      = "w" "h" "i" "l" "e"
DO_TOK         = "d" "o"
FORMAT_TOK     = "f" "o" "r" "m" "a" "t"
LENGTH_TOK     = "l" "e" "n" "g" "t" "h"
input          = request / verb_request
output         = (response / verb_response) *WS CRLF
request        = (query / hquery / vquery / update / vupdate / target / conditional / loop) *WS CR
verb_request   = "!" (query / hquery / vquery / update / vupdate / target) *WS CR
response       = status *WS (argument / hargument)
verb_response  = status *WS (target / variable) "=" *WS (argument / hargument)
status         = OK_TOK / ERROR_TOK
query          = target *WS "?"
hquery         = target *WS "?" *WS "$"
update         = target *WS "=" *WS (argument / hargument / condition / arithmetic)
vquery         = variable *WS "?"
vupdate        = variable *WS "=" *WS (vargument / condition / arithmetic)
conditional    = if_then / if_then_else
if_then        = IF_TOK *WS condition *WS THEN_TOK *WS btq_exp
if_then_else   = IF_TOK *WS condition *WS THEN_TOK *WS btq_exp *WS ELSE_TOK *WS btq_exp
loop           = WHILE_TOK *WS condition *WS DO_TOK *WS btq_exp
btq_exp        = '`' (query / vquery / update / vupdate / target) "`"
format         = FORMAT_TOK *WS "(" *WS variable *WS "," *WS QSTR_TOK *WS ")"
length         = LENGTH_TOK *WS "(" *WS variable *WS ")"
argument       = INT_TOK / FLT_TOK / QSTR_TOK / intarray / fltarray / variable / concatenation / format
hargument      = "$" 1*(*WS 2HEX) ; note that size of data must be > 0
vargument      = INT_TOK / QSTR_TOK / intarray / variable / btq_exp / concatenation / format / length
target         = STR_TOK [*WS arraydims]
arraydims      = "(" *WS arrayoffsets *WS ")"
arrayoffsets   = arraydim [*WS "," *WS arraydim]
arraydim       = RANGE / OFFSET / variable
variable       = VAR_TOK [*WS vardim]
vardim         = "[" *WS (1*DIGIT / variable / arithmetic) *WS "]"
not_variable   = "!" variable
intarray       = "{" *WS intsequence *WS "}"
intsequence    = INT_TOK *( *WS "," *WS INT_TOK)
fltarray       = "{" *WS fltsequence *WS "}"
fltsequence    = FLT_TOK *(*WS "," *WS FLT_TOK)
concatenation  = (QSTR_TOK / variable / format) *WS ":" *WS (QSTR_TOK / variable / format)
condition      = "(" *WS (logical / comparison) *WS ")"
logical        = logical_and / logical_or
logical_and    = (variable / not_variable) *WS "&" "&" *WS (variable / not_variable / comparison)
logical_or     = (variable / not_variable) *WS "|" "|" *WS (variable / not_variable / comparison)
comparison     = lt / gt / lte / gte / eq / ineq
lt             = (INT_TOK / variable) *WS "<" *WS (INT_TOK / QSTR_TOK / variable / arithmetic)
gt             = (INT_TOK / variable) *WS ">" *WS (INT_TOK / QSTR_TOK / variable / arithmetic)
lte            = (INT_TOK / variable) *WS "<" "=" *WS (INT_TOK / QSTR_TOK / variable / arithmetic)
gte            = (INT_TOK / variable) *WS ">" "=" *WS (INT_TOK / QSTR_TOK / variable / arithmetic)
eq             = (INT_TOK / QSTR_TOK / variable) *WS "=" "=" *WS (INT_TOK / QSTR_TOK / variable / arithmetic)
ineq           = (INT_TOK / QSTR_TOK / variable) *WS "!" "=" *WS (INT_TOK / QSTR_TOK / variable / arithmetic)
arithmetic     = "(" *WS (add / sub / mult / div / mod) *WS ")"
add            = (INT_TOK / variable) *WS "+" *WS (INT_TOK / variable / arithmetic)
sub            = (INT_TOK / variable) *WS "-" *WS (INT_TOK / variable / arithmetic)
mult           = (INT_TOK / variable) *WS "*" *WS (INT_TOK / variable / arithmetic)
div            = (INT_TOK / variable) *WS "/" *WS (INT_TOK / variable / arithmetic)
mod            = (INT_TOK / variable) *WS "%" *WS (INT_TOK / variable / arithmetic)
```