CUSTOM STRATEGY PROGRAMMER'S MANUAL

Baldur Gíslason

March 22, 2021

Contents

1 Introduction

This manual describes the syntax and how-tos of the custom strategy development environment on Baldur's Control Systems electronic control units.
The custom strategies are written in a programming language that is specific to this application. It is a compiled language but runs in a virtual machine environment on the ECU rather than being compiled to native ARM machine language. The programming environment allows read access to every real time variable and every configuration item found on the ECU. It also provides a memory stack as well as memory allocation space static to the custom strategy environment. Custom real time variables may be declared in the custom strategy environment and they become first class members of the real time data memory, including the ability to be used as inputs to existing strategies, the ability to be data logged by the controller's built in memory as well as the ability to be sent over the CAN network like predefined real time variables. Not like user defined variables that are created inside the Calibrator application and don't exist on the ECU.
Custom configuration variables, including tables and curves may also be defined and accessed from the Calibrator application as well as the custom strategy code, and these appear in the configuration tree in Calibrator like existing configuration variables.
As of the writing of this document, the compiler and development environment is in a very early state of development, and while stable and reasonably well tested, many features remain to be implemented. It would be highly useful to get feedback from users of the programming environment on what it is lacking the most.
If the controller is connected by USB when it powers up, execution of the custom strategy is delayed by 2 seconds. This is to allow time to recover by performing a firmware flash in case a bug in the custom strategy program that is running renders the ECU inoperable.

2 Syntax

The syntax in many ways resembles the C language, but is not a full implementation of the C language. The language is case sensitive. Statements are terminated by semicolons, arrays are denoted by square brackets, function blocks are enclosed in curly brackets. Variables and constants may be defined outside of functions only, but stack memory is provided for function call arguments. Anything in a line that is preceded by two forward slashes (//) is ignored by the compiler and may be used for comments or to temporarily disable execution of those lines.
The order of operations can be controlled by nesting operations inside brackets () so the nested operation is performed on its own. In the virtual machine, all function calls return a 32 bit signed value, and all function arguments are 32 bit signed as well but types may be specified for clarity of what range of values are to be expected. Function arguments are literal values except for array types they are references. References ignore the type specified but always inherit the type of the passing variable, type may be specified for clarity. All data is stored in little endian byte order. The virtual machine has a single execution thread and callbacks are made from the ECU's main background loop which is interruptable by real time functions such as the handling of timed inputs and outputs. A good indicator of the overall strain on the ECU's microprocessor is the `mainfrequency` real time variable, that indicates how often over the past second the main thread was able to complete a full loop. Currently, variables must be defined outside of functions, stack memory is only available for function arguments, but arguments are writable inside the function. It is also legal to call a function with fewer arguments than the function header defines. When a program starts up, the special function `init` is called. If the `init` function does not register any call backs to functions to be executed later then no further program execution will happen after the `init` function exits.
For best results, computation heavy functions should only execute as frequently as new values are required, delegating them to fixed interval callbacks if infrequent updates are necessary or using variables to keep track of when updates are necessary.
Callbacks that override the assignment of existing variables should in most cases be kept simple as in many cases they execute as frequently as on every loop of the main thread.

2.1 Data types

The following data types are available for variables.

const A named constant, can be given a value as it is defined. Is a 32 bit signed value.

uint8 Unsigned 8 bit integer.

uint16 Unsigned 16 bit integer.

uint32 Unsigned 32 bit integer.

int8 Signed 8 bit integer.

int16 Signed 16 bit integer.

int32 Signed 32 bit integer.

In the variable definition, the variable name may be succeeded by square brackets enclosing a number to mark that the variable is an array, and the number inside the brackets marks the number of elements making up the array. When defining a function argument that is to be a reference to an array, the name should be succeeded by square brackets with nothing inside. Array elements can be addressed individually by specifying the array name succeeded by the element number inside square brackets. The first element in an array is number zero. When a function specification specifies an input array of a const type, it means write access is not required for that array so it may originate from configuration or be a string literal. The array may nonetheless be from a writable origin.

2.1.1 Strings

String literals may be substituted for constant array references as function arguments where such are specified.
A string literal produces a constant (not writable) reference to a null terminated string.
A string literal begins and ends with double quotation marks. Inside the string, the following escape sequences are supported to produce special characters:

\0 Null (ASCII value 0)

\a Bell (ASCII value 7)

\b Backspace (ASCII value 8)

\e Escape (ASCII value 27)

\f Form feed (ASCII value 12)

\v Vertical tab (ASCII value 11)

\n Line feed (ASCII value 10)

\r Carriage return (ASCII value 13)

\t Horizontal tab (ASCII value 9)

**** A literal backslash (ASCII value 92)

\xNN Any character, value specified by two digits of hexadecimal. Eg. \x20 produces ASCII value 32 which is a space.

2.2 Operators

= Assignment operator. Left of the operator is the destination to be written to with the value of what is on the right of the operator. The compiler will throw an error if the destination is not writable by the custom strategy programming environment. Writable variables include static variables declared in the programming environment, stack variables declared inside of a function call, real time variables declared as part of the custom strategies and built in variables part of the programming environment such as callback pointers.
Examples of destinations that are NOT writable by this operator include configuration variables, pre-existing real time variables, declared and built in functions, and any kind of constant with the exception that user defined named constants can be assigned a value at time of definition.

== Equal comparison operator, returns true if values on both sides are equal.

>= Greater than or equal comparison operator, returns true if the value on the left of the operator is greater than or equal to the value on the right.

> Greater than comparison operator, returns true if the value on the left of the operator is greater than the value on the right.

<= Less than or equal comparison operator, returns true if the value on the left of the operator is less than or equal to the value on the right.

< Less than comparison operator, returns true if the value on the left of the operator is less than the value on the right.

<< Left shift operator. The result is the value on the left shifted left by the value on the right. Effectively the result is a multiplication by 2 to the power of the number on the right. In cases where a multiplication by any number that is a power of 2 is desired, performing a bit shift may execute faster.

>> Right shift operator. The result is the value on the left shifted right by the value on the right. Effectively the result is a division by 2 to the power of the number on the right. In cases where a division by

any number that is a power of 2 is desired, performing a bit shift will definitely result in faster execution.

& Bitwise AND operator. Bits that are 1 in the values passed on both sides of the operator will be 1 in the result.

| Bitwise OR operator. Bits that are 1 in values on either side of the operator will be 1 in the result.

&& Logical AND operator. If values on both sides of the operator are not equal to 0, this operator returns a value of 1.

|| Logical OR operator. If a value on either side or both sides of the operator are not equal to 0, this operator returns a value of 1.

+ Adding operator.

- Subtracting operator. May also preceed a numeric value to indicate it is a negative value.

* Multiplication operator.

/ Division operator. Quotient on the left, dividend on the right.

% Remainder operator. Returns the remainder of an integer division in which the quotient is on the left and the dividend is on the right.

∧ Bitwise XOR operator.

[] Array access operator. Used to access an item from an array of values for either reading or writing. For example foo[0] = 1; or foo = bar[5];

2.3 Built in constants

user_out0 up through user_out15 These constants describe output pins that may be associated with physical outputs on the ECU or as conditions for built in ECU strategies.
These also set corresponding bits in the real time variable `userflags` which is referred by aliases `userflag0` up through `userflag15`.

caninterfacecount Denotes the number of CAN interfaces available on this ECU.

2.4 Built in variables

callback_1hz A function pointer called once per second. The function takes no arguments and no return value is expected.

callback_2hz A function pointer called twice per second. The function takes no arguments and no return value is expected.

callback_5hz A function pointer called 5 times per second. The function takes no arguments and no return value is expected.

callback_10hz A function pointer called 10 times per second. The function takes no arguments and no return value is expected.

callback_20hz A function pointer called 20 times per second. The function takes no arguments and no return value is expected.

callback_25hz A function pointer called 25 times per second. The function takes no arguments and no return value is expected.

callback_50hz A function pointer called 50 times per second. The function takes no arguments and no return value is expected.

callback_100hz A function pointer called 100 times per second. The function takes no arguments and no return value is expected.

callback_200hz A function pointer called 200 times per second. The function takes no arguments and no return value is expected.

callback_250hz A function pointer called 250 times per second. The function takes no arguments and no return value is expected.

callback_500hz A function pointer called 500 times per second. The function takes no arguments and no return value is expected.

callback_1000hz A function pointer called 1000 times per second. The function takes no arguments and no return value is expected.

callback_can1rx A function pointer called each time a frame is received on CAN interface 1.
Arguments:
(uint32 CANID, uint8 DLC, uint32 DataA, uint32 DataB)
No return value is expected.

callback_can2rx A function pointer called each time a frame is received on CAN interface 2.
Arguments:
(uint32 CANID, uint8 DLC, uint32 DataA, uint32 DataB)
No return value is expected.

callback_uart0rx A function pointer called when one or more bytes of data have been received on UART 0. If this callback is not set prior to calling `init_uart()` then receiving will be disabled altogether.
Arguments:
(uint8 count, uint8 data[])

callback_uart3rx Same as `callback_uart0rx` but for UART 3.

callback_din1event A function pointer called whenever an edge is registered on digital input 1, if digital input 1 is in edge sensitive mode (frequency input mode). Equivalent functions exist for other digital inputs that are frequency capable.

userpwmpw1 up through userpwmpwXX depending on controller User defined PWM output pulse width in microseconds. Pulse widths shorter than 100 microseconds are ignored by the output control code and will disable the output temporarily.

userpwmperiod1 up through userpwmperiodXX depending on controller User defined PWM output pulse interval period in microseconds. The output control code has hard coded minimum periods that vary by output and will respect those. The longest period that will be respected in any case is 1 million microseconds (1 second). Since both the pulse width and period are specified in microseconds, the pulse width must be some fraction of the period for the output to work correctly.

2.5 Built in functions

varpicker Used to get the value from the variable a varpicker type configuration value points to, takes one argument which is the varpicker pointer value.
Example: if foo is a configuration variable of varpicker type and it is set to point to analog0, then varpicker(foo) returns the value of analog0.

ternary A function that takes 3 arguments. First argument is evaluated and if its value is non-zero the second argument is evaluated and

its value returned, otherwise the third argument gets evaluated and returned.

wraps8 Takes a signed 32 bit value and returns it as a signed 8 bit value clipped to the appropriate range.

wrapu8 Takes a signed 32 bit value and returns it as an unsigned 8 bit value clipped to the appropriate range.

wraps16 Takes a signed 32 bit value and returns it as a signed 16 bit value clipped to the appropriate range.

wrapu16 Takes a signed 32 bit value and returns it as an unsigned 16 bit value clipped to the appropriate range.

2.5.1 Table lookup functions

A number of table lookup functions are provided, they all follow the same basic format but differ in the data types they operate on.

tablelookup16 Perform a lookup with linear interpolation on an unsigned 16 bit table using unsigned 16 bit table axis.
Arguments:
(const uint16 table[], const uint16 xaxis[], const uint16 yaxis[], uint8 xdims, uint8 ydims, uint16 xvalue, uint16 yvalue)

tablelookups16 Perform a lookup with linear interpolation on a signed 16 bit table using unsigned 16 bit table axis.
Arguments:
(const int16 table[], const uint16 xaxis[], const uint16 yaxis[], uint8 xdims, uint8 ydims, uint16 xvalue, uint16 yvalue)

tablelookup8 Perform a lookup with linear interpolation on an unsigned 8 bit table using unsigned 16 bit table axis.
Arguments:
(const uint8 table[], const uint16 xaxis[], const uint16 yaxis[], uint8 xdims, uint8 ydims, uint16 xvalue, uint16 yvalue)

tablelookups8 Perform a lookup with linear interpolation on a signed 8 bit table using unsigned 16 bit table axis.
Arguments:
(const int8 table[], const uint16 xaxis[], const uint16 yaxis[], uint8 xdims, uint8 ydims, uint16 xvalue, uint16 yvalue)

2.5.2 Curve lookup functions

curvelookup16 Perform a lookup with linear interpolation on an unsigned 16 bit curve using unsigned 16 bit table axis.
Arguments:
(const uint16 curve[], const uint16 axis[], uint8 dims, uint16 input)

curvelookups16 Perform a lookup with linear interpolation on a signed 16 bit curve using unsigned 16 bit table axis.
Arguments:
(const int16 curve[], const uint16 axis[], uint8 dims, uint16 input)

curvelookup16 Perform a lookup with linear interpolation on an unsigned 8 bit curve using unsigned 16 bit table axis.
Arguments:
(const uint8 curve[], const uint16 axis[], uint8 dims, uint16 input)

curvelookup16 Perform a lookup with linear interpolation on a signed 8 bit curve using unsigned 16 bit table axis.
Arguments:
(const int8 curve[], const uint16 axis[], uint8 dims, uint16 input)

2.5.3 String/buffer handling functions

memcopy Copies data from one array to another.
Arguments:
(uint8 destination[], const uint8 source[], uint16 bytecount)
Source and destination can be of different types, a direct copy is performed without caring about data types, so an array of 8 bytes may be copied into an array of 2 32 bit values for example. Only restrictions are both have to be array types and the destination must be writable.
Care should be taken as limited boundary checking is performed in case the specified byte count exceeds the memory allocated to the destination or source variable.
Returns 1 on success, 0 if either variable reference is invalid. Returns 0 if destination is not writable.

offsetcopy Copies data from one array to another, but source and/or destination may be offset.
(uint8 destination[], uint16 dstoffset, const uint8 source[], uint16 srcoffset, uint16 bytecount)
Source and destination can be of different types, a direct copy is performed without caring about data types, so an array of 8 bytes may be copied into an array of 2 32 bit values for example. Only restrictions are both have to be array types, and the destination must be writable. Source and destination may be the same array with different offsets.
Care should be taken as limited boundary checking is performed in case the specified byte count exceeds the memory allocated to the destination or source variable.
Returns 1 on success, 0 if either variable reference is invalid. Returns 0 if destination is not writable.

strcopy Copies a null terminated string from one array to another.
Arguments:
(uint8 destination[], const uint8 source[], uint16 bytecount)

Source and destination can be of different types, a direct copy is performed without caring about data types, so an array of 8 bytes may be copied into an array of 2 32 bit values for example. Only restrictions are both have to be array types, and the destination has to be writable. In this case bytecount specifies the maximum length of the string, excluding the terminating null byte.
Returns 1 on success, 0 if either variable reference is invalid. Returns 0 if destination is not writable.

strlen Parses a null terminated string and returns its length.
Arguments:
(const uint8 string[], uint16 maxlength)
maxlength specifies the maximum number of bytes to parse, in case the given array does not actually contain a null terminated string. If no null byte is found then the return value will be equal to maxlength and the string should be considered invalid.

strformat Formats a string with values from other sources such as other strings or integers.
Arguments:
(uint8 destination[], uint16 maxlength, const uint8 formatstring[], ...)
The number of arguments after the format string depends on the contents of the format string. It may not be zero, use `strcopy` if you wish to copy a string with no formatting. The format string accepts the following special sequences:

%s Specifies that the next argument is another string

%d Specifies that the next argument is an integer and should be formatted as base 10 decimal. Two variations of this are supported. % Nd where N is a number specifying the minimum number of characters to be used up by the output and if the number is small the slack is taken up by spaces. %0Nd works the same way except the slack is taken up by zeroes.

%x Specifies that the next argument is an integer and should be formatted as hexadecimal. The same variations are supported to prefix with spaces or zeroes.

%.Nd Prints a decimal number, but places a decimal point at the Nth place, and prefixes with zeroes as necessary. So if the input number is 66666, and the format string is %.3d it prints 66.666, and if the number is 66 it prints 0.066

Usage example:

```
uint8 buffer[50];

function mycallback() {
  strformat(buffer, 49, "Engine speed is %d RPM\n", rt.enginespeed);
}
```

2.5.4 Miscellaneous library functions

crc8 Computes CRC8 checksum of an array of bytes.
Arguments:
(const uint8 data[], uint8 offset, uint8 length, uint8 seed, uint8 poly)
data is a reference to an array of bytes. offset is an offset into the array to start from, usually 0. length is the number of bytes to process. seed is the initial value of the crc register. poly is the value convolutionally XOR'ed with the value on each iteration.

sqrt Computes square root of a 32 bit unsigned integer value and returns a 16 bit unsigned value.
Arguments:
(uint32 value)

2.5.5 Communication functions

can1tx Transmit data on first CAN bus interface.
Arguments:
(uint32 CANID, uint8 DLC, const uint8 data[])
CANID is the frame identifier, up to 29 bits long.
DLC is the data length descriptor, a maximum of 8 bytes.
data is a pointer to an array.
Return value is 1 if transmission was successful, 0 if all transmit buffers are full and transmission was not possible.

can2tx Transmit data on the second CAN bus interface if the system is equipped with two or more CAN interfaces. Otherwise same syntax as can1tx.

init_uart Initialises serial port UART. If data is to be received, the appropriate callback must be assigned before calling this function.
Arguments:
(uint8 uart, uint32 baudrate)
uart specifies the UART to initialise. Valid options are 0 or 3. Some controllers may not have any external pins connected to the UARTs, but all have UART 0 available on a 4 pin header internally which can be connected to a Bluetooth module or a level shifter circuit for wired connections.
baudate specifies the data rate. Valid options are 1200, 4800, 9600, 19200, 38400, 57600 and 115200

uart0tx Transmits data on UART 0. A maximum of 64 bytes may be transmitted at once.
Arguments:
(uint8 count, const uint8 data[])

uart3tx Same as `uart0tx` but for UART 3

2.5.6 System functions

getbit Get the value of a system logic bit as described by the `map_inputs` value map in the configuration file. Best used in conjunction with a configuration item that is of enum type and associates with map_inputs for its values.
Arguments:
(uint8 bit)
Returns the value of the bit selected, either 0 or 1.

setbit Sets the value of an output pin, it is best to only use the named constants for user program output pins (user_out0 up through user_out15) as arguments to this function. Note that the bit numbers are not compatible with those used by the getbit function.
Arguments:
(uint8 bit, uint8 booleanstate)
Has no defined return value.

startlogging Starts recording a data log to the internal logging memory of the ECU. Takes one argument which is the type of log to be recorded if the ECU supports multiple types of logs.
Arguments:
(uint8 logtype)
The following log types are valid in general, but not all controllers support all types of logs.
1: Standard log, a snapshot of the real time variable struct taken at fixed intervals with the possibility of a variable interval in burst mode.
2: Event log, a log record written for every input and output event in an event based ECU such as the LPC4/LPC8 and DID1. Not available on the MPC1 and DSL1.
3: Structured log, a snapshot of the real time variable struct taken at fixed intervals but with the possibility of recording selected variables at faster rates in 4 groups of 16 variables.
4: CAN receive log, writes a log entry for every CAN frame received on any CAN interface. Transmitted frames are not recorded.
Returns no defined value.
The log status may be observed on the logging, logrecs and logstatus variables.

stoplogging If a data log is being recorded to the controller's internal memory, this will halt logging. Takes no arguments, returns no value.

3 Custom real time variables

Real time variables are variables that can be logged, displayed in Calibrator and used as inputs to standard ECU functions.
Currently these are described in JSON format, but a graphical interface to edit this descriptor is planned.
The descriptor is a JSON array of objects, so it starts and ends with square brackets and inside it the objects are surrounded by curly braces, separated by commas.
Example:

```
[
  {
    "id": "steeringangle",
    "type": 2,
    "unit": "degrees",
    "sign": 1
  },
  {
    "id": "yawrate",
    "type": 2,
    "unit": "degrees/second",
    "digits": 1,
    "scale": 0.1,
    "sign": 1,
    "descr": "Computed yaw rate"
  },
  {
    "id": "yawrate_desired",
    "type": 2,
    "unit": "degrees/second",
    "digits": 1,
    "scale": 0.1,
    "sign": 1,
    "descr": "Desired yaw rate according to steering angle"
  },
  {
    "id": "yawrate_error",
    "type": 2,
    "unit": "degrees/second",
    "digits": 1,
    "scale": 0.1,
    "sign": 1,
    "descr": "Yaw rate error, positive numbers for oversteer, negative
    ↪ numbers for understeer"
  },
  {
    "id": "oversteer_ignition_retard",
    "type": 2,
```

```
    "unit": "degrees crank",
    "digits": 1,
    "scale": 0.01,
    "descr": "Ignition timing retard applied by oversteer reaction
    ↪ strategy"
  }
]
```

3.1 Properties

A number of properties can be described in a real time variable descriptor, most of them optional. Identifier is the only mandatory field but if a type is not specified then the default type is a bit which must be accompanied by an address property.

id Identifier of variable. Case sensitive. Apart from ASCII letters and numbers the only other character allowed is the underscore. Must not begin with a number and must not contain any spaces.

type Variable type. The following types are permitted:

- **0** Bit. Must be accompanied by an address making it an alias for a single bit of another variable.
- **1** 8 bit integer, sign specified separately.
- **2** 16 bit integer, sign specified separately.
- **3** 32 bit integer, sign specified separately.
- **5** 32 bit IEEE float. No library functions are currently available in the programming environment for handling these but they are understood by Calibrator.
- **6** 64 bit IEEE double. No library functions are currently available in the programming environment for handling these but they are understood by Calibrator.
- **8** Enumerator translating an integer value to text. An address can be specified to associate with a subset of bits on another variable, if address is omitted it takes the form of an 8 bit integer variable.
- **10** Time stamp, 32 bits long comprised of multiple values.
  Bits 5..0 seconds.
  Bits 11..6 minutes.
  Bits 16..12 hours in 24h format
  Bits 21..17 day of month
  Bits 25..22 month
  Bits 31..26 years since 2015

sign If omitted, the variable is unsigned. If specified with a value of 1 the variable is signed.

unit Text specifying the unit of the variable. Default is empty.

digits Specifies the number of digits after the decimal point when variable is logged or displayed in Calibrator. Default 0.

scale A floating point multiplier used when displaying or logging the variable in Calibrator.

offset An integer number added to the raw number prior to scaling, can be negative even if the value is unsigned.

inverse If this option is specified with a value of 1, the result is inverted (1 over result division) before displaying.

expression A mathematical expression to fomat the value prior to displaying, where preformatted value is represented by lower case letter `x` but other variables or config items may be referenced as well. See Calibrator documentation for format of expression. Makes variable ineligible for use in varpicker options so it will not be available for use in ECU control strategies.

descr A text description of the variable displayed in Calibrator variable lists.

address Byte offset inside the real time variable memory space, or in the case of a single bit or enumerator variable the name of the parent variable and the bit offset inside of it. For single bit, `bitmask.0` specifies the lowest significant bit of variable bitmask. For enums, `bitmask.1.3` specifies a range of bits starting at bit 1 ending at bit 3 on the variable bitmask.

variations An array of conditions and following each condition an object whose contents is included in the parent object if the condition is true, overriding existing fields. Each condition contains a configuration variable, a comparison operator and a value. For each condition there should be a corresponding inverted condition to undo whatever changes that condition made if it no longer applies. If the corresponding object includes an endif clause, then the condition list will not be evaluated further once the object has been applied, otherwise there are no limits on how many conditions can apply. Example:

```
"variations": [
  [ "fuelstrategy", "=", "Fuel mass" ], { "unit": "mg/cycle", "endif":
    ↪ true },
  [ "fuelstrategy", "=", "Air mass (VE)"], { "unit": "%" }
]
```

4 Custom configuration variables

Due to the way configuration as stored by the ECU can differ a lot from how it's displayed in Calibrator, the configuration definition is split into 2 parts. One part is the definition of the displayed configuration, arranged by category, and another part is the definition of how the configuration is stored on the ECU.

4.1 Configuration flash storage

The configuration stored in ECU flash is described by an array of JSON objects. A graphical interface to edit this is planned but until then this is how it works.
In the example below, a few different configuration types are defined. Two number arrays, one two dimensional function, one single dimensional function, a hidden parent value, an enum attached to the parent value and a single value scalar. The variables described mimic a fictional stability control system. Not to be taken as an example of how to implement a stability control system.

```
[
  {
    "id": "stabilcontrolbits",
    "type": 1,
    "descr": "A byte that will not be displayed anywhere by itself, but
    ↪ configuration objects refer to it. This description won't appear
    ↪ anywhere in the software because this variable is effectively
    ↪ hidden"
  },
  {
    "id": "oversteer_ignretard_enable",
    "type": 8,
    "options": [ "Disabled", "Enabled" ],
    "descr": "If enabled, the control strategy will retard ignition
    ↪ timing if oversteer is detected"
  },
  {
    "id": "oversteer_ignretard_max",
    "type": 2,
    "scale": 0.01,
    "digits": 1,
    "max": 50,
    "unit": "degrees",
    "descr": "This option specifies the maximum amount the ignition
    ↪ timing may be retarded to reduce torque during oversteer events",
    "applies": [ "oversteer_ignretard_enable", "=", "Enabled" ]
```

```
  },
  {
    "id": "desiredyawtable",
    "type": 2,
    "unit": "degrees/second",
    "digits": 1,
    "scale": 0.1,
    "sign": 1,
    "array": 96,
    "rows": 12,
    "cols": 8,
    "descr": "Desired yaw rate as a function of vehicle speed and
    ↪ steering angle"
  },
  {
    "id": "allowable_oversteer",
    "type": 2,
    "unit": "degrees/second",
    "digits": 1,
    "scale": 0.1,
    "array": 12,
    "descr": "What amount of oversteer is permitted before the control
    ↪ strategy takes action"
  },
  {
    "id": "steeringanglebreakpoints",
    "type": 2,
    "unit": "degrees",
    "digits": 1,
    "scale": 0.1,
    "sign": 1,
    "array": 8,
    "input": "steeringangle",
    "descr": "Breakpoints for steering angle on table axis"
  },
  {
    "id": "roadspeedbreakpoints",
    "type": 2,
    "unit": "km/h",
    "digits": 1,
    "scale": 0.1,
    "array": 12,
    "input": "roadspeed",
    "descr": "Breakpoints for vehicle speed on table axis"
  }
]
```

4.1.1 Properties

A number of properties are defined for each variable, most of which are optional. Identifier is the only mandatory field but if a type is not specified then the default type is a bit which must be accompanied by an address property. The properties differ slightly from the real time variables but there are similarities.

id Identifier of variable. Case sensitive. Apart from ASCII letters and numbers the only other character allowed is the underscore. Must

not begin with a number and must not contain any spaces.

type Variable type. The following types are permitted:

- **0** Bit. Must be accompanied by an address making it an alias for a single bit of another variable.
- **1** 8 bit integer, sign specified separately.
- **2** 16 bit integer, sign specified separately.
- **3** 32 bit integer, sign specified separately.
- **5** 32 bit IEEE float. No library functions are currently available in the programming environment for handling these but they are understood by Calibrator.
- **6** 64 bit IEEE double. No library functions are currently available in the programming environment for handling these but they are understood by Calibrator.
- **7** Text variable for storing comments or descriptions.
- **8** Enumerator translating an integer value to text. An address can be specified to associate with a subset of bits on another variable, if address is omitted it takes the form of an 8 bit integer variable.
- **9** Varpicker variable. Presented in Calibrator as a drop down list of real time variables and stores a memory pointer to the chosen variable.

sign If omitted, the variable is unsigned. If specified with a value of 1 the variable is signed.

unit Text specifying the unit of the variable. Default is empty.

digits Specifies the number of digits after the decimal point when variable is logged or displayed in Calibrator. Default 0.

scale A floating point multiplier used when displaying or logging the variable in Calibrator.

offset An integer number added to the raw number prior to scaling, can be negative even if the value is unsigned.

inverse If this option is specified with a value of 1, the result is inverted (1 over result division) before displaying.

descr A text description of the variable displayed in the description box when a user presses the F1 key while in edit mode.

array An integer value that indicates this configuration item is an array of multiple values, and denotes the number of values it contains. Using the rows/cols properties the displayed values may be restricted to a smaller count.

cols Indicates the number of columns in a table, or values being used in case of a single dimensional function that is to be displayed horizontally. Can be an integer value or it can be the identifier of another configuration variable that specifies the number being used.

rows Indicates the number of rows in a table, or values being used in case of a single dimensional function that is to be displayed vertically. Can be an integer value or it can be the identifier of another configuration variable that specifies the number being used.

input Identifier of a specific real time variable. Indicates that the value(s) in this configuration item are related to a specific real time variable, in which case the Calibrator application will display a button to grab the current value of the real time variable and place it at the selected field in the editor and in the tree view the identifier of the real time variable will be displayed next to a function if the axis the function is based on has this defined relationship to the real time variable.

describedby Identifier of a Varpicker configuration item that specifies the real time variable this configuration item relates to. The format (sign, scale, offset, digits, unit, max/min) is then inherited from the real time variable selected by the Varpicker.

min Minimum value the user may input.

max Maximum value the user may input.

maxwidth In case of a Varpicker object, this property specifies the maximum byte width of the variables it presents. Set this to 2 if functions that are to be used with the Varpicker object use 16 bit variables to store the data related to the Varpicker object. If omitted, any width is allowed.

relative Used in conjunction with a `describedby` property to indicate that regardless of the Varpicker source being signed or unsigned, this variable is always signed as it describes a value relative to the origin variable. In this case the offset specified by the origin variable will not be applied either, only the scale.

verbose Used with Varpicker types only, if a value of 1 is given this specifies that the Varpicker should display the type of each variable in the list and not just the name.

hex If a value of 1 is given, this specifies that the values should be displayed in hexadecimal format, using the 0x notation to clarify that they are hexadecimal values.

options When an enum type is used, this property specifies the number to text relationship. Can be a reference by name to a predefined enum or map such as `enum_enabled` or `map_inputs` or it can be an array of text values that are enumerated automatically, or it can

be an array of arrays, where the subarrays contain a number as the first member and the associated text as the second member.

address Byte offset inside the real time variable memory space, or in the case of a single bit or enumerator variable the name of the parent variable and the bit offset inside of it. For single bit, `bitmask.0` specifies the lowest significant bit of variable bitmask. For enums, `bitmask.1.3` specifies a range of bits starting at bit 1 ending at bit 3 on the variable bitmask.

variations An array of conditions and following each condition an object whose contents is included in the parent object if the condition is true, overriding existing fields. Each condition contains a configuration variable, a comparison operator and a value. For each condition there should be a corresponding inverted condition to undo whatever changes that condition made if it no longer applies. If the corresponding object includes an endif clause, then the condition list will not be evaluated further once the object has been applied, otherwise there are no limits on how many conditions can apply. Example:

```
"variations": [
  [ "fuelstrategy", "=", "Fuel mass" ], { "unit": "mg/cycle", "endif":
    ↪ true },
  [ "fuelstrategy", "=", "Air mass (VE)"], { "unit": "%" }
]
```

beforechange A Calibrator script (see Calibrator user's manual) that is executed only when exiting edit mode and the value of this configuration item has been changed. Executed before pushing the changes to the ECU.

onupdate A Calibrator script that is executed whenever the value of this configuration item has been changed for any reason, by the editor or by downloading values from the ECU. Executed before the changes are pushed to the ECU. Executed regardless of whether ECU is connected or not.

onchange A Calibrator script that is executed after a new value has been pushed to the ECU.

onmerge A Calibrator script that is executed after the value of the configuration item has been changed by merging with the configuration downloaded from the ECU.

4.2 Configuration user interface

The configuration user interface is described by a JSON array whose layout represents the layout displayed in the Calibrator configuration tree.
Example:

```
[
{
  "name": "My strategies",
  "type": 255,
  "value": [
    {
      "name": "Retard ignition on excessive oversteer",
      "id": "oversteer_ignretard_enable",
      "type": 3
    },
    {
      "name": "Maximum timing to remove",
      "id": "oversteer_ignretard_max",
      "applies": [ "oversteer_ignretard_enable", "=", "Enabled" ]
    },
    {
      "name": "Desired yaw rate",
      "id": "desiredyawtable",
      "type": 5,
      "xaxis": "steeringanglebreakpoints",
      "yaxis": "roadspeedbreakpoints"
    },
    {
      "name": "Permitted oversteer rate",
      "id": "allowable_oversteer",
      "type": 5,
      "xaxis": "roadspeedbreakpoints"
    },
    {
      "name": "Road speed breakpoints",
      "id": "roadspeedbreakpoints",
      "type": 4
    },
    {
      "name": "Steering angle breakpoints",
      "id": "steeringanglebreakpoints",
      "type": 4
    }
  ]
}
]
```

4.2.1 Properties

The properties of the configuration user interface objects are again different from the configuration flash objects. Again most of the properties are optional, name is always mandatory and identifier is mandatory unless it's a widget panel or a subcategory.

name Displayed name

type A type specifier, different from the real time or flash variable type. The following types are valid:

- **0** This is the default if no type is specified, and specifies a single numerical value.
- **3** Enumerator type, used in conjunction with flash item type 8.
- **4** List of values that is not a function of something, typically used for variables that are used as function axis.
- **5** A function, with one or two axis. Internally can contain numerical values, text values, enumerators or Varpickers.
- **6** Text type, used in conjunction with flash item type 7.
- **7** Varpicker type, used in conjunction with flash item type 9.
- **9** Widget panel. Can have widgets that edit configuration items as well as display data. See Calibrator user's manual. Note that this type is currently not compatible with Calibrator's configuration comparison and merge functions so if the configuration items being edited here do not have another representation they will not show up in the difference dialog when connecting to an ECU or when comparing configuration files.
- **255** A subcategory, requires a value that is a JSON array of more objects.

id Identifier to connect to a configuration flash object, unless type is 255 or 9.

context A documentation and real time variable context identifier, inherits to subcategories. There is currently no way to input the context data from inside Calibrator but this is a valid option so it is included for sake of complete coverage.

xaxis In case of a function (type 5) this property specifies the X axis, and can be either the name of the configuration item that contains the axis values or it can be a JSON array of values, numerical or text.

yaxis Same as `xaxis` but for the Y axis.

xvariable Can be used with a function type to explicitly specify a real time variable that the X axis relates to in case the variable is different from the variable implied by the X axis.

yvariable Same as `xvariable` but for the Y axis.

colwidth A floating point multiplier that changes the widths of the columns used to present the data.

novisual If a value of 1 is given, this property specifies that it makes no sense to display the values of this function/list visually.

value Contains the default value of a configuration item or the array containing the child items of a category.

applies An array of conditions, each condition starting with a configuration variable, the second item being the condition, either equals or not equals ("=" or "!=") and the third item being the value, so the array must contain items in multiples of three. If not all of the conditions are met then the configuration item icon becomes grey in the configuration tree, indicating to the user that the values of this configuration item is not significant due to the strategy using it not being enabled.

applies_any Same format as the `applies` property but instead of all conditions having to be true, only one condition has to match to indicate that this item is in use and significant.

5 A coding tutorial

Let's start with simplest possible program. This program turns on an output so if any output pin on the ECU is configured as `User program output 0` that output will be activated. The lowest bit of the `userflags` variable will also be set high, also visible on the `userflag0` alias. This program exits after the initial run of the init function and only runs again if the ECU restarts or if an updated version of the program is pushed to the ECU.

```
function init() {
  setbit(user_out0, 1);
}
```

To make the program continue execution after initialisation, one or more callbacks must be created. The following program sets a callback that runs 5 times per second, checks the value of the userflags0 variable and changes it.

```
function mycallback() {
  if(userflag0) {
    setbit(user_out0, 0);
  } else {
    setbit(user_out0, 1);
  }
}

function init() {
  callback_5hz = mycallback;
}
```

Sometimes you may want to execute a program once every time an input value changes. In this case it is simplest to allocate a variable to keep the old state of the input value and compare every time the program executes.

```
uint8 oldvalue;

function mycallback() {
// Toggle bit whenever din1 changes state from 0 to 1
  if(din1 && (oldvalue == 0)) {
    setbit(user_out0, 0);
  } else {
    setbit(user_out0, 1);
  }
  oldvalue = rt.din1;
}

function init() {
```

```
  callback_5hz = mycallback;
}
```

Let's get to some basic arithmetic.

```
uint16 averagespeed;

function mycallback() {
// This function computes the average value of roadspeed and roadspeed2,
    ↪ using a right shift to perform a divide by 2 operation.
averagespeed = (rt.roadspeed + rt.roadspeed2) >> 1;
}

function init() {
  callback_100hz = mycallback; // Run my call back routine 100 times per
    ↪ second
}
```

To control program flow, two methods are available the `if/else` statements and the `while` loop statement.

```
//An if statement with an optional else if statement and an else
    ↪ statement.
if(rt.enginespeed > 500) {
  engine_running = 2;
} else if(rt.enginespeed) {
  engine_running = 1;
} else {
  engine_running = 0;
}

uint16 somearray[8];
//A while loop, use sparingly.
i = 7;
while(i) {
  somearray[i] = somearray[i] + 1;
  i = i - 1;
}
```

It is also possible to define functions that take arguments and return a value.

```
uint16 absoval;

function absvalue(a, b) {
  return(sqrt(a * a + b * b));
}

function mycallback() {
  absoval = absvalue(rt.roadspeed, rt.roadspeed2);
}
```

Let's say we wanted to convert an analog input value into a frequency output.

```
// Sample a snapshot for any variable we are going to evaluate multiple
// times in the same function in case the variable changes value
// in between evaluations.
uint16 sample;

function cb20hz() {
```

```
  sample = rt.analog1;
  if(sample < 10) {
    // Set minimum frequency to 1Hz
    // The period is specified in microseconds.
    userpwmperiod1 = 1000000;
  } else {
    // Set output period somewhere in the range of 1 second to
    // 2.442 milliseconds (1Hz to 410Hz) as a 12 bit analog input
    // has an integer value ranging from 0 to 4095
    userpwmperiod1 = 10000000 / sample;
  }
  // Ensure the pulse width is half of the period
  userpwmpw1 = userpwmperiod1 >> 1;
  // Now we just have to configure that output as user program
  // controlled PWM output 1.
  // This is done in the configuration and
  // not in the programming environment.
}

function init() {
  callback_20hz = cb20hz;
}
```

A text formatting example, this one prints a string on UART 0 every second

```
uint8 txbuf[64];
uint8 len;

function callback20hz() {
  len = strformat(txbuf, 63, "Hello world. My supply voltage is %.3d
    ↪ volts\n", rt.supplyvoltage);
  uart0tx(txbuf, len);
}


function init() {
  init_uart(0, 115200);
  callback_1hz = callback20hz;
}
```

5.1 Complete example programs

5.1.1 Example using configuration and real time variables

This program performs lookups using the tables defined in Custom real time variables and Custom configuration variables chapters.

```
uint16 aos;

function mycallback() {
  // Convert analog voltage to steering angle zeroed in the centre, with
    ↪  a range of 186 degrees in either direction.
  rt.steeringangle = (rt.analog4 - 2047) / 11;
  // Assign value from CAN bus connected yaw rate sensor to yawrate
    ↪ variable.
  rt.yawrate = rt.can1src0;
```

```
  // This is a signed 16 bit table, so using the tablelookups16 function
    ↪  to access it.
  rt.yawrate_desired = tablelookups16(conf.desiredyawtable, conf.
    ↪ steeringanglebreakpoints, conf.roadspeedbreakpoints, 8, 12, rt.
    ↪ steeringangle, rt.roadspeed);
  if(rt.yawrate_desired >= 0) { // Turning right or going straight
    rt.yawrate_error = rt.yawrate_desired - rt.yawrate;
  } else { // Turning left
    // Flip the order when the numbers go negative, result should be
    ↪ positive number for oversteer
    // and negative number for understeer.
    rt.yawrate_error = rt.yawrate - rt.yawrate_desired;
  }

  if(conf.oversteer_ignretard_enable == 0) {
    rt.oversteer_ignition_retard = 0;
    return();
  }

  aos = curvelookup16(conf.allowable_oversteer, conf.
    ↪ roadspeedbreakpoints, 12, rt.roadspeed);

  if(rt.yawrate_error > aos) { // We have oversteer
    if((rt.oversteer_ignition_retard + 10) < conf.oversteer_ignretard_max
    ↪ ) {
      // Retard roughly 0.1 degree on every cycle. Remember that this is
      // an example of how to use the programming environment,
      // not an example of how to implement stability control.
      rt.oversteer_ignition_retard = rt.oversteer_ignition_retard + 10;
    }
  } else {
    rt.oversteer_ignition_retard = 0;
  }
}

function init() {
  // Run the callback as often as we can expect to have new information
  // from the yaw rate sensor.
  callback_50hz = mycallback;
}
```

5.1.2 Serial port communications example

This program communicates with a cheap simple dash display via Bluetooth.

```
uint8 txbuf[32];
uint32 tmp;

function callback20hz() {
// send coolant temp, 0-255 for -20 to 235C
  txbuf[4] = (rt.coolanttemp - 2531) / 10;
// send RPM 0-10000 = 0-20000 RPM in 6 cyl setting. Take a snapshot of
    ↪ variable prior to splitting it into bytes for thread safety
  tmp = rt.enginespeed >> 1;
  txbuf[5] = tmp >> 8;
  txbuf[6] = tmp;
// send supply volt, 0-255 = 0-25.5V
```

```
  txbuf[3] = rt.supplyvoltage / 100;
// send oil temp, 0-255 = -20 - 235C
  txbuf[8] = (rt.oiltemp - 2531) / 10;
// send oil pressure, 0-255 = 0-255 PSI, divide millibars by 69 to
    ↪ convert to PSI.
  txbuf[9] = rt.oilpress / 69;
// send boost 0-255 = 0-255 PSI
  tmp = rt.map;
  if(tmp > 1000) {
    txbuf[10] = (tmp - 1000) / 69;
  } else {
    txbuf[10] = 0;
  }
// send AFR, 0-9990 = 0-999:1 no scaling necessary to display lambda 1.0
    ↪  as 100
  tmp = rt.lambda;
  txbuf[14] = tmp;
  txbuf[13] = tmp >> 8;
// send EGT 0-9999 = 0-9999C
  tmp = (rt.egt1 - 2731) / 10;
  txbuf[12] = tmp;
  txbuf[11] = tmp >> 8;
// send speed 0-255 km/h, 24 should be 0
  txbuf[24] = 0;
  tmp = rt.roadspeed / 10;
  txbuf[23] = tmp;
// increment a counter on a byte that isn't used to get around checksum
    ↪ irregularities
  txbuf[20] = txbuf[20] + 1;

  txbuf[25] = crc8(txbuf, 0, 25, 0, 0x31);
  uart0tx(txbuf, 27);
}


function init() {
// Assign the constant bytes that don't change
  txbuf[0] = 0xFF;
  txbuf[1] = 0x1B;
  txbuf[2] = 0xA0;
  txbuf[26] = 0xFE;
// No data is to be received to no receive callback is registered prior
    ↪ to initialising UART
  init_uart(0, 9600);
  callback_20hz = callback20hz;
}
```