

The Kvaser *t* Programming Language

Copyright 2011 KVASER AB, Mölndal, Sweden
<http://www.kvaser.com>

Compiler version: 3.1
Last updated 2013-01-30

This document is preliminary and subject to change.

(This page is intentionally left blank.)

Table of Contents

1	<i>t</i> Programming.....	9
1.1	Overview.....	9
1.1.1	Creating a <i>t</i> Program.....	10
1.1.2	Using a <i>t</i> Program.....	10
1.1.3	Version of <i>t</i> compiler.....	10
1.1.4	Inherited Settings Versus <i>t</i> Program Settings.....	11
1.2	Introduction to the <i>t</i> Language for CAN.....	11
1.2.1	Simple Example.....	11
1.2.2	More Complex Example.....	12
1.2.3	Receive Example.....	12
1.3	Elements of a <i>t</i> Program.....	13
1.3.1	The <code>variables</code> Section.....	13
1.3.2	The <code>envvar</code> Section.....	13
1.3.3	Event Hooks.....	13
1.3.4	Function Definitions.....	14
1.3.5	Function Declarations.....	14
1.3.6	CAN Databases (.dbc Files).....	14
1.3.7	Exception Handling.....	15
2	<i>t</i> Language Reference.....	16
2.1	Types.....	16
2.1.1	Predefined Types.....	16
2.1.2	Run-time library types.....	16
2.1.3	Type declarations.....	16
2.1.4	Explicit Type Conversion (typecast).....	17
2.1.5	Arrays.....	17
2.2	Variables and constants.....	18
2.2.1	Variable Definitions.....	18
2.2.2	Variable Initializers.....	19
2.2.3	Constant Definitions.....	19
2.2.4	Reference Variable Definitions.....	20
2.2.5	Reserved Keywords.....	20
2.3	Serialization and Deserialization.....	20
2.4	Environment Variables.....	21
2.5	Functions.....	21

2.5.1	Function definitions	21
2.5.2	Function declarations	22
2.5.3	Function overloading	22
2.6	Statements	23
2.6.1	The <code>if</code> Statement	23
2.6.2	The <code>while</code> Statement	23
2.6.3	The <code>do</code> Statement	24
2.6.4	The <code>for</code> Statement	24
2.6.5	The <code>switch</code> Statement	24
2.6.6	The <code>return</code> Statement	25
2.6.7	The <code>break</code> Statement	25
2.6.8	The <code>continue</code> Statement	25
2.7	Expressions	26
2.8	Blocks	27
2.9	Comments	27
2.10	Hooks	27
2.10.1	Hook Execution Order	28
2.10.2	Hook Details	28
2.11	Using CAN Databases	34
2.12	<code>#include</code>	35
2.13	<code>#error</code>	35
2.14	<code>#warning</code>	35
2.15	Conditional compilation	35
2.15.1	Command line options	35
2.15.2	<code>#if</code> / <code>#else</code> / <code>#endif</code>	36
2.15.3	<code>#ifdef</code> / <code>#ifndef</code>	36
2.16	Predefined Output Functions	37
2.16.1	<code>printf</code>	37
2.17	Predefined File I/O Functions	37
2.17.1	<code>fileOpen</code>	37
2.17.2	<code>fileClose</code>	38
2.17.3	<code>fileGets</code>	38
2.17.4	<code>fileReadBlock</code>	38
2.17.5	<code>filePuts</code>	38
2.17.6	<code>fileWriteBlock</code>	39
2.17.7	<code>fileSeek</code>	39

2.17.8	fileDelete	39
2.18	Predefined XML Functions.....	39
2.18.1	xmlOpen.....	39
2.18.2	xmlClose	40
2.18.3	xmlGet.....	40
2.19	Predefined Math Functions.....	41
2.19.1	sin.....	41
2.19.2	cos.....	41
2.19.3	tan.....	41
2.19.4	asin.....	42
2.19.5	acos.....	42
2.19.6	atan.....	42
2.19.7	abs.....	42
2.19.8	ceil.....	42
2.19.9	floor	42
2.19.10	randomize.....	43
2.19.11	random.....	43
2.19.12	sqrt.....	43
2.19.13	exp.....	43
2.19.14	exp10	43
2.19.15	round.....	44
2.19.16	log.....	44
2.19.17	log10	44
2.20	Predefined String Functions	44
2.20.1	strlen.....	44
2.20.2	strcpy.....	44
2.20.3	strcat.....	45
2.20.4	strcmp.....	45
2.20.5	sprintf.....	45
2.20.6	atoi.....	45
2.20.7	atof.....	46
2.20.8	itoa.....	46
2.21	Predefined CAN Related Functions	46
2.21.1	canWrite.....	46
2.21.2	canGetTimestamp	47
2.21.3	canSetBitRate	47
2.21.4	canSetBusParams	47
2.21.5	canGetBusParams	48

2.21.6	canBusOff.....	48
2.21.7	canBusOn.....	48
2.21.8	canSetBusOutputControl.....	49
2.22	Predefined Timer Related Functions	49
2.22.1	timerStart.....	49
2.22.2	timerCancel.....	50
2.22.3	timerIsPending.....	50
2.22.4	timerSetHandler.....	50
2.23	Predefined Environment Variable Functions.....	51
2.23.1	envvarSetValue.....	51
2.23.2	envvarGetValue.....	51
2.24	Predefined CAN Transport Protocol Related Functions.....	52
2.24.1	canTpOpen.....	52
2.24.2	canTpClose.....	52
2.24.3	canTpTransmit.....	52
2.24.4	canTpAbort.....	53
2.24.5	canTpSetHandler.....	53
2.24.6	canTpSetAttr.....	53
2.24.7	canTpGetAttr.....	54
2.25	Predefined t Program Related Functions	54
2.25.1	scriptLoad.....	54
2.25.2	scriptUnload.....	55
2.25.3	scriptStart.....	55
2.25.4	scriptStop.....	55
2.26	Predefined Logger Related Functions.....	56
2.26.1	filterDropMessage.....	56
2.26.2	loggerStatus.....	56
2.26.3	loggerStart.....	56
2.26.4	loggerStop.....	56
2.27	Predefined Crypto Related Functions.....	57
2.27.1	cryptoHash.....	57
2.27.2	cryptoEncipher.....	57
2.27.3	cryptoDecipher.....	57
2.27.4	cryptoRandom.....	58
2.28	Predefined System Related Functions.....	58
2.28.1	sysGetValue.....	58

2.29	Predefined Customer Data Related Functions	58
2.29.1	customerdataGetLength	58
2.29.2	customerdataGetValue.....	58
2.30	Predefined LED Functions.....	59
2.30.1	ledOpen.....	59
2.30.2	ledClose	59
2.30.3	ledSet.....	59
2.31	Other Predefined Functions.....	59
2.31.1	timeGetDate	59
2.31.2	timeGetLocal.....	60
2.31.3	sizeof.....	60
2.32	Predefined Symbols	60
2.32.1	this.....	60
2.33	Predefined Types	61
2.33.1	Timer	61
2.33.2	CanMessage	61
2.33.3	CanTpMessage.....	61
2.33.4	tm	61
2.33.5	ExceptionData	61
2.33.6	Types for Database Defined Signals	62
2.34	Predefined Constants.....	62
2.34.1	Maths	62
2.34.2	Timer Period Counts	62
2.34.3	CAN Message Flags	63
2.34.4	CAN Bitrates	63
2.34.5	CAN Driver Modes.....	63
2.34.6	CAN status.....	63
2.34.7	CAN Transport Protocols	63
2.34.8	File Open	64
2.34.9	File Seek	64
2.34.10	Logger Status	65
2.34.11	Crypto	65
2.34.12	System.....	65
2.34.13	User parameters	65
2.34.14	LED Constants.....	65
2.34.15	Envvar Constants	66
3	Document revision history	67

1 *t* Programming

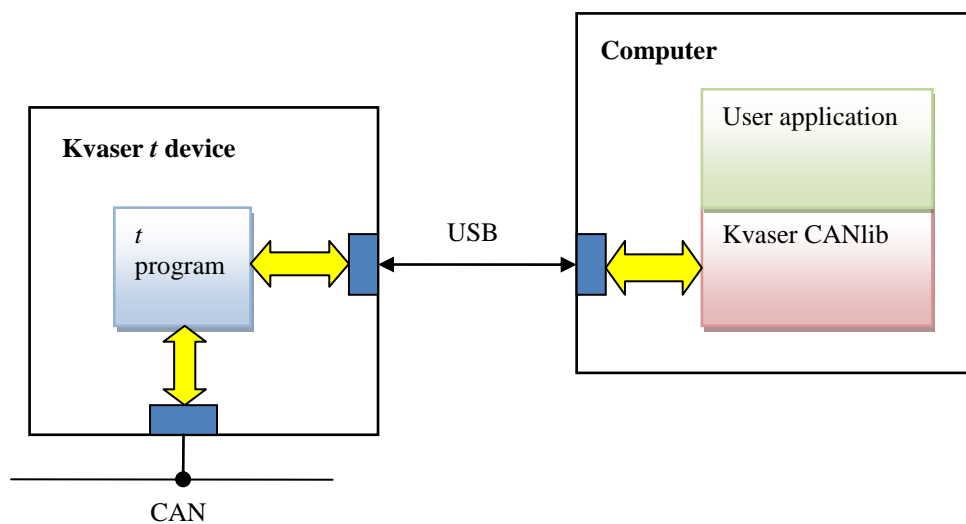
1.1 Overview

The Kvaser *t* programming language is event oriented and modeled after C. It can be used to customize the behavior of the Kvaser Eagle and other Kvaser *t* capable devices.

A *t* program is invoked via hooks, which are entry points that are executed at the occurrence of certain events. These events can be, for example, the arrival of specific CAN messages, timer expiration, or external input.

Like any Kvaser CAN interface, the Kvaser Eagle can be used via CANlib on a PC. The addition of *t* programs running directly on the Kvaser Eagle makes it possible to react much quicker to CAN bus events (for example to speed up file transfer protocols or to simulate missing hardware). The Kvaser Eagle can also operate completely autonomously.

In this document, “Kvaser *t*”, “*t*”, “the *t* language”, and “the *t* programming language” are used as synonyms for the language. A piece of software written using it is a *t* program or possibly a *t* script. No difference in meaning is intended between, for example, “a Kvaser *t* program” and “a *t* script”.



1.1.1 Creating a *t* Program

First of all, decide what you want to happen in your program. Do you want to receive a message, send a message, or a combination of both?

You must decide what will trigger an action in your code. You can trigger on execution start/stop, on a key press, on a timer, or on reception of a message.

If you want to receive CAN messages, use the `on CanMessage` construct. After receiving a message, you may print to a log file, send a message, update a counter, etc.

If you want to send messages, decide how the transmission will be triggered. As mentioned above, there are different ways of triggering in the system.

Use your favorite text editor to create the *t* program.

If you are familiar with C programming this will all be straightforward!

1.1.2 Using a *t* Program

When you have written your *t* program, you need to compile the file using the supplied command line compiler. Any syntactic or semantic errors in the file are diagnosed by the compiler and must be corrected before the compiler will produce a binary format file as output.

This is an example of how to use the *t* language compiler:

```
> scc.exe test.t
```

The command will produce a binary output file named test.txe that can then be loaded and executed.

Running long program segments may adversely affect the general system operation. Do not write code that loops "forever"!

1.1.3 Version of *t* compiler

The compiler version number consists of three parts: MAJOR.MINOR.BUILD. Updates in the MAJOR part indicates significant changes that needs support from new firmware. This also means that older scripts may need to be recompiled before they can be executed on a newer firmware. See the release notes for information about which compiler versions a specific firmware supports.

The current version of the compiler is printed when invoked without any arguments.

1.1.3.1 Running Via CANlib

To test the program, you must download the compiled binary file to your Kvaser Eagle via CANlib. You can also download data files via CANlib.

CANlib provides functions for starting and stopping the *t* program. There are functions for communicating with the *t* program also.

The following example uses the Kvaser *t* utility (a program that wraps CANLib's *t* related functions) to download and start the compiled *t* program:

```
> tutil.exe -channel=0 -slot=0 -load test.txe -start
```

Or, if the *t* program is stored on the SD card, this command starts the program:

```
> tutil.exe -channel=0 -slot=0 -loadlocal test.txe -start
```

For help on how to use the Kvaser *t* utility, use the following command:

```
> tutil.exe -?
```

Strings printed by the *t* program can be captured by the *t* utility with the following command:

```
> tutil.exe -channel=0 -listen -1
```

Where '-1' means "forever".

1.1.3.2 Running in Standalone Mode

To run the *t* program in standalone mode (i.e. the Kvaser Eagle connected to a CAN bus and not a USB bus), you must compile your program(s) and then download the generated binary file(s) with the Kvaser Memorator Tools.

1.1.4 Inherited Settings Versus *t* Program Settings

As already stated, there are two ways of starting a *t* program - a configuration downloaded by the Kvaser Memorator Tools or a CANlib application like `tutil.exe`. The bus parameters for the CAN bus must be set before the bus can be used. This can be done either via CANlib, the Kvaser Memorator Tools, or directly in a *t* program.

The simple rule is: the last one to set the bus parameters "wins".

1.1.4.1 Start-up Sequence in Standalone Mode

When in standalone mode, the *t* program starts automatically when the device powers on.

In this case, the bus settings from the log configuration are used. The program does not need to setup the bus or go bus-on since these steps have already been handled before the program starts.

However, the *t* program may change the bus settings. The following is the start-up sequence in standalone mode:

1. Kvaser Eagle is connected to the CAN bus; power on
2. Kvaser Eagle reads configuration
3. Kvaser Eagle sets the bus parameters and goes bus-on
4. *t* program loads, starts, and optionally changes the bus parameters

1.1.4.2 Start-up with CANlib Application

When using CANlib to start a *t* program, the CANlib program or the *t* program must set the bus parameters and activate a channel on the bus. If the CANlib program sets the bus parameters, the *t* program can still re-configure the bus settings.

1.2 Introduction to the *t* Language for CAN

If you have programmed in C or C++, most of the *t* language will look very familiar. However, the event based nature of *t* means that a *t* program looks slightly different. Some C language features are missing, such as unsigned integers, union types and pointers, while some features from other languages have been incorporated.

1.2.1 Simple Example

```
on start {
    canSetBitrRate(canBITRATE_1M);
    canSetBusOutputControl(canDRIVER_SILENT);
    canBusOn();
}

on stop {
    canBusOff();
}

on CanMessage 54321x {
    printf("Hello, User!\n");
}
```

This program simply prints the text "Hello, User!" to the standard output when the program detects a CAN message with the extended identifier 54321 (decimal) is received.

1.2.2 More Complex Example

```

variables {
    const int can_channel = 0;
    const int can_bitrate = 1000000;
    Timer sendTimer;
    int value;
}

on start {
    printf("Hello again! It's the Second Example speaking..\n");
    printf("Send message 1000 and I will send another one.\n");
    value = 0;

    canSetBitrate(can_channel, can_bitrate);
    canSetBusOutputControl(can_channel, canDRIVER_NORMAL);
    canBusOn(can_channel);

    sendTimer.timeout = 1000; // Milliseconds
    timerStart(sendTimer, FOREVER);
}

on stop {
    canBusOff(can_channel);
}

on CanMessage<1> 1000 {
    CanMessage msg;
    msg.id = 123;
    msg.dlc = 8;
    msg.flags = 0;
    msg.data = "\x11\x22\x33\x44\x55\x66\x77\x88";
    canWrite(msg);
}

on Timer sendTimer {
    CanMessage msg;
    msg.id = 1234;
    msg.dlc = 2;
    msg.flags = canMSG_EXT;
    msg.data[0] = value;
    msg.data[1] = value >> 8;
    value++;
    canWrite(msg);
}

```

This program will send one CAN message with standard identifier 123 (decimal) each time a CAN message with identifier 1000 is received on channel 1. The program will also send a CAN message with extended identifier 1234 once every second.

1.2.3 Receive Example

```

on start {
    printf("Hello again! It's the Third Example speaking..\n");
    printf("Receive a message..\n");
}

on CanMessage<*> [*] {
    printf("Pling! Message %d received, with %d bytes of data:",

```

```
        this.id, this.dlc);
    for(int i = 0; i < this.dlc; i++) {
        printf("%02x ", this.data[i]);
    }
    printf("\n");
}
```

This program will receive any CAN message and print the message's data to the standard output.

1.3 Elements of a *t* Program

A *t* program consists of a sequence of constructs. The order is irrelevant, except that a construct is not visible before it is declared. None of the constructs are mandatory, and there can be any number of any construct. One or more CAN database files can also be used.

The possible constructs are:

- **variables** section, contains definitions of global variables, constants, and types
- **envvar** section, contains declarations of communication variables
- event hook, connects a desired code response to a specific event
- function definition, a piece of code that can be called from elsewhere in the program
- function declaration, establishes a function's interface without providing the function definition

1.3.1 The variables Section

All variables, constants, and types that need to be global (accessible from multiple hooks or functions) must be defined in a **variables** section. This is equivalent to defining something in global scope in a C program.

Example

```
variables {
    const int LENGTH = 80;
    const char filename[13] = "file.dat";
    int count = 0;
    char text[LENGTH] = filename;
    float x, y;
}
```

1.3.2 The envvar Section

Environment variables are used to communicate between different *t* programs or with a PC using CANlib. Environment variables are defined in the **envvar** section, and just like variables in the **variables** section, environment variables are global.

Example

```
envvar {
    int option;
    char message[8];
    float angle;
}
```

1.3.3 Event Hooks

All events that a *t* program will react to are specified using various **on** event hooks. In effect, these are functions which are called when the specific events occur. Among other events, the program can react to the start of the program, the arrival of a CAN message, and the elapse of a timer.

Example

```
on start {
    count = 0;
}

on CanMessage 100 {
    length = this.dlc;
    message = this.data;
}
```

1.3.4 Function Definitions

You can define your own functions in *t*. This can greatly enhance the usability of the language. Each function definition is placed at the outermost level in the program file, in the same way as a **variables** section.

Example

```
void send (int id, const byte data[])
{
    CanMessages msg;
    msg.flags = canMSG_EXT;
    msg.id = id;
    msg.dlc = data.count;
    msg.data = data;
    canWrite(msg);
}
```

1.3.5 Function Declarations

Sometimes using a function before the function is defined is necessary or preferable. For that reason, the language supports function declarations. The function declaration prepares the compiler by describing the function's interface.

Example

```
void report(char text[]);
float getAngle(void);
```

1.3.6 CAN Databases (.dbc Files)

While writing a *t* program without CAN database files is possible, using a CAN database file is a good idea. By using a CAN database file, the compiler can generate **typedef** and **struct** members to match the signals in the database, as well as automatically handle any specified scaling of the signals.

Example

```
on CanMessage StartRamps {
    ramp_amp = this.Amplitude.Phys;
    ramp_offset = this.Offset.Phys;
}
```

When compiling this program, we need to add the database containing "StartRamps" like this:

```
> scc.exe -dbase=my_database.dbc my_prog.t
```

When working with multiple databases, two databases might define the same message. To clarify which definition should be used, a logical name should be assigned to the database. For example, if we have two databases with "rpm" as shown below:

database1.dbc:

```
...
BO_ 43 rpm: 8 Vector__XXX
SG_ value : 0|16@1+ (1,0) [0|5000] "RPM" Vector__XXX
```

database2.dbc:

```
...
BO_ 143 rpm: 8 Vector__XXX
SG_ value : 0|16@1+ (1,0) [0|20000] "RPM" Vector__XXX
```

Then we need to establish a unique logical name for the database during compilation:

```
> scc.exe -dbase=my@database1.dbc -dbase=their@database2.dbc my_prog.t
```

Now the logical name can be used in the program:

```
on CanMessage my_rpm {
    // ID = 43, RPM: 0 - 5000
}

on CanMessage their_rpm {
    // ID = 143, RPM: 0 - 20000
}
```

Note that the compiler does not take max/min values from the database into account when generating code for signal access. This means that a read/write from/to a signal outside the signal's bounds will NOT cause an exception or saturation.

Also note that the *t* language does not support unsigned integer or double types. This means unsigned to signed conversions could produce the wrong result. Also, signals defined as type double in the database cannot be used.

For more information on how to compile with a database file, see:

```
> scc.exe -?
```

1.3.7 Exception Handling

The *t* program environment has a built in exception handler that will print some useful information and then stop the program where the exception occurred. For examples on how to write your own handler, see section 2.10.2.8.

If an exception hook is not provided, the program will be stopped.

2 *t* Language Reference

This document describes the *t* language as implemented in version 2.0 of the *t* compiler (scc.exe).

2.1 Types

t has a small number of predefined types, which can be used as building blocks for more complex types. The run-time library defines additional complex types and more can be automatically generated when using database files.

2.1.1 Predefined Types

The following types are predefined:

Type	Meaning
float	A 32-bit floating point number
int	A 32-bit signed integer
char	An 8-bit character (signed integer)
byte	An 8-bit unsigned integer

The **char** and **byte** types are always treated as **int** in expressions (after sign extending for **char**).

When an operator can take **float** arguments, if one of the operands is an **int** and the other operand is a **float**, the **int** is converted to **float** before being used.

For operators that only take **int** arguments, any **float** is converted to an **int** before being used. Automatic type conversion from **float** to **int** is also done for array indexing and **switch** statements.

2.1.2 Run-time library types

The run-time library defines the following types:

Type	Use
FileHandle	Opaque type for dealing with files
XmlHandle	Opaque type for dealing with XML files
CanTpHandle	Opaque type for dealing with CAN transport protocol sessions
EnvVar	Opaque type for dealing with environment variables
Timer	Partly opaque type for dealing with timers - see section 2.33.1
CanMessage	Both sent and received CAN messages are of this type - see section 2.33.2
CanTpMessage	Used to report CAN transport protocol events - see section 2.33.3
tm	Calendar data and time - see section 2.33.4
ExceptionData	Contains information about an exception - see section 2.33.5

2.1.3 Type declarations

New data types are created through type declarations. Type declarations are indicated by **typedef**. Currently, **typedef** only supports the structure complex data type (similar to C structures and Pascal records) which is indicated using **struct**.

A structure is composed of a list of variable declaration. Each variable declared is considered a member of the structure.

A structure member behaves like any other variable of the same type with one exception. Currently, a structure member that is a predefined type cannot be passed by reference unless the member is an array. Arrays are always passed by reference.

A structure can only be assigned values by assigning a value to each contained member. Assigning values to the whole structure is currently not supported.

Syntax

```
typedef struct '{'  
    struct_body  
}' ident ';' ;
```

struct_body is one or more member declarations using the same syntax as variable declarations. *ident* is the name of the type being defined.

Example

```
variables {  
    typedef struct {  
        byte data[4];  
    } Temporary;  
  
    typedef struct {  
        int control[3];  
        Temporary temp;  
        CanMessage msg;  
    } aacinfo;  
}  
...  
  
aacinfo aac;  
aac.control[0] = 0x251;  
aac.temp.data = 0;
```

2.1.4 Explicit Type Conversion (typecast)

In addition to the automatic type conversion described in section 2.1.1, explicit type casts can be performed for the following data types: **float**, **int**, **char**, and **byte**. The syntax is the same as C.

Syntax

```
(' type_name ') expression
```

type_name is the desired resulting data type and *expression* is the value being converted.

Example

```
int p          = (int)M_PI;  
int y          = x * (int)round(f(t));  
float seconds  = (float)milliseconds / 1000;  
int delta      = (char)msg.data[2];  
log((byte)++seqno);
```

2.1.5 Arrays

t supports arrays of any data type. Arrays are always range checked on use. That means, any attempt to use an array to access memory outside the array's range will cause an exception. The number of elements in an array can be accessed using the array variable's `count` member.

An array element behaves like any other variable of the same type with one exception. Currently, an array element that is a predefined type cannot be passed by reference. If the array element is a structure, the element can be passed by reference since structures are always passed by reference.

Array assignment is supported for arrays of predefined data types. If a scalar value is assigned to the array, elements in the array will be set to the scalar value. If the assignment is another array, all data will be copied into the destination array. If the assignment is from a smaller array, old data is left at the end of the destination array. If the assignment is from a larger array, any extra data at the end is ignored.

Wherever an array can be used, so can a slice of the array. A slice is simply a reference to a subset of elements in the array. A slice's subset can be created in three different ways:

- the `'..'` range operator specifies a start and end index
- the `'.'` range operator specifies a start index and a total element count
- the `'+'` operator specifies a start index

Example

```
vector[i]          = t / 3;
counts[0 .. 5] = 0;           // Elements 0, 1, ..., 5
data[n * 8, 8] = msg.data;    // Elements 8n, 8n + 1, ..., 8n + 7
total = my_sum(v + (f - 1));  // Elements v + f - 1, v + f, ...
                               // v is an array, f is a number
i          = messages.count;
```

The parenthesis are necessary for the calculation of *total* in the example above. If the parenthesis were removed, the compiler would consider $v + f$ a temporary array and the -1 would be an attempt at a negative offset. If $f - 1$ produces a negative value during execution, the resulting negative offset will be treated like any other out-of-range access (i.e. an exception will be thrown).

Note: *t* only supports one-dimensional arrays. The effect of multiple dimensions can be had by using arrays of structures containing arrays.

2.2 Variables and constants

A variable or constant is defined in a **variables** section (which makes the object globally visible) or in a block (which makes the object local to that block). Like in C++, variables and constants can be defined anywhere within a block, not just at the beginning of a block. Also like C++, a variable can be defined in a for-loop statement. In this case, the variable's scope is limited to the loop.

Example

```
{
  int total = 3;
  x = y * 3 + total;
  const float angle = M_PI * 0.75;
  float length = vector * sin(angle);
  for(int n = 0; n < total; n++) {
    ...
  }
  ...
}
```

A variable initializer defined in a **variables** section is executed during program initialization.

2.2.1 Variable Definitions

Syntax

```
datatype ident [ '[' size ']' ] [ '=' initializer ]
[ ',' ident [ '[' size ']' ] [ '=' initializer ] ... ] ';' ;
```

Where *datatype* is either a predefined type or a user-defined type, *size* is a constant expression, and *initializer* is appropriate for the associated data type.

Note: No default initialization is performed for any variable. Default initialization for references is "BAD".

2.2.1.1 Static Variables

Inside a block, a variable's storage class can be specified as **static**. Like in C, this means that the variable's value is retained upon exiting and re-entering the block. If the static variable uses an initializer, the initializer will only be executed on the first pass of the variable definition.

Example

```
int engineSpeed, engineType = 3;
float height = length * tan(angle);
char text[20] = "Hello World!", data[2] = {7, Max_size - 2};
static int count = 0;
```

2.2.2 Variable Initializers

Scalar variable initializers are not restricted, whether defined in a **variables** section or a block. However, an array initializer must be a constant expression.

Arrays can be initialized using a list of expressions. Arrays of type **char** or **byte** can also use a special string initializer. Initializing with a string is equivalent to putting all the separate characters in sequence in a normal array initializer, except that the string always contains an implicit '\0' at the end of the array. The string initializer can be replaced by a defined **char** array constant.

The compiler will indicate an error if the initializer is too large to fit in the array. But if the initializer is smaller than the array, the remaining elements will be undefined.

Example

```
float angles[2] = {M_PI * 0.5, M_PI * 1.3};
char texts[10] = "\x02Hi\x05Hello";
```

Note: Variables of type structure cannot use initializers.

2.2.3 Constant Definitions

A constant is defined in a **variables** section (making the constant visible globally) or inside a block (making the constant local to the block).

The constant expression (*constexpr*) which defines the constant's value may only use other constants. However, the expression can be arbitrarily complex, since the expression is evaluated at compile time.

Only **float**, **int**, and array of **char** constants are allowed.

Syntax

```
const ( int | float ) ident '=' constexpr
    [ ',' ident '=' constexpr ... ] ';'
const char ident '[' ']' '=' string ';'

```

Example

```
const float pi = 3.141592;
const int loops = 100, maxloops = loops * 3;
const char filename[] = "report.txt";
```

2.2.4 Reference Variable Definitions

It is possible to create a reference to a variable, or part of a variable, with the same result as if it had been passed as a reference argument to a function (see 2.5.1). That is, any access using the reference will operate directly on the original variable, or part of variable. Reference variables are mainly useful to simplify code.

The references themselves are not assignable, and can thus only be set using an initializer. This also means that it is impossible to ever have a reference to something that is out of scope.

The compiler will automatically infer the type of the reference from its initializer.

Syntax

```
auto ident '=' '&' expr ';' ;
```

Example

```
auto fileno = &filename[6..7];  
auto values = &table[n].row[i].col;  
auto speed = &msg.data[2];
```

2.2.5 Reserved Keywords

The following keywords are reserved and may not be used as identifiers:

```
__internal, __pragma_internal, action, and, auto, bool, break,  
byte, CanMessage, CanTpMessage, case, catch, char, class, const,  
continue, default, delete, do, double, else, enum, envvar,  
exception, export, extern, false, float, for, goto, if, init,  
input, int, key, long, monitor, mutex, namespace, new, not, on,  
or, output, postfilter, prefilter, private, protected, public,  
register, return, semaphore, short, short, signal, signed,  
sizeof, start, startup, static, stop, stopped, string, struct,  
switch, thread, throw, timer, true, try, typedef, union, union,  
unsigned, using, va_arg, va_end, va_list, va_start, variables,  
while, void, volatile, xor
```

2.3 *Serialization and Deserialization*

When using data files, or when communicating with another program, it is often useful to be able to convert data from structure form to an array of bytes, and the other way around. This is called serialization and deserialization, respectively. In *t* this is done by simply assigning an arbitrary structure to an appropriately sized array of **char** / **byte**, and vice versa.

The result of serialization is what is sometimes called a packed array with the contents from the struct. That is, there is no padding between values for alignment. Also, any compiler/runtime internal parts of the structure in question have been removed. Values of type **int** and **float** are stored in little endian format (i.e. with the least significant byte first).

To find out what how many bytes are needed for a serialized structure, use `sizeof()` (see 2.31.3).

Example

```
char buf[sizeof(CanMessage)];  
CanMessage incoming_msg, outgoing_msg;  
...  
buf = incoming_msg;  
fileWriteBlock(outfile, buf);  
...  
fileReadBlock(infile, buf);  
outgoing_msg = buf;
```

2.4 Environment Variables

An environment variable can only be defined in an **envvar** section. Like a variable defined in a **variables** section, an environment variable is visible globally. Further, an environment variable is visible to other *t* programs as well as a connected PC.

The lifespan of an environment variable is from when the first *t* program using the environment variable loads until the last *t* program using the environment variable unloads. To access environment variables on a PC, use the Kvaser CANlib script API for environment variables.

Unlike ordinary variables, environment variables cannot be initialized or accessed directly in the *t* program. Rather, consider them handles and use `envvarSetValue` and `envvarGetValue` to access their contents.

Syntax

```
int    ident ';'
float ident ';'
char  ident '[' size ']' ';'

```

ident is the name of the environment variable and *size* is a constant expression. *size* is limited to ENVVAR_MAX_SIZE. The three data types shown are the only environment variable data types supported.

Example

```
envvar {
    int CANidToPC;
}

on CanMessage * {
    envvarSetValue(CANidToPC, this.id);
}

```

2.5 Functions

Functions can call each other and themselves, freely.

2.5.1 Function definitions

Syntax

```
datatype ident '(' formal_parameters ')' fun_def
```

The function's return type is specified by *datatype*. *formal_parameters* is a sequence of:

```
[ const ] datatype [ '&' ] ident [ '[' ']' ]
[ ',' [ const ] datatype [ '&' ] ident [ '[' ']' ] ... ]
```

When the formal parameter's name (*ident*) is preceded by '&', the parameter is passed by reference. If the formal parameter's name is succeeded by '[]', the parameter is an array.

The data types **int**, **float**, **char**, and **byte** are passed by value. To pass an **int**, **float**, **char**, or **byte** by reference; place an ampersand, '&', in front of the parameters name in the function declaration and in front of the argument when calling the function. Arrays and structures are always passed by reference.

Array and structure parameters may have their types preceded by the keyword **const**, which means the formal parameter may not be modified by the function. Passing a non-constant argument via a constant formal parameter is legal. However, passing a constant argument via a non-constant formal parameter is an error that is enforced by the compiler.

Though not supported in user code, the run time library makes use of a special case where the last parameter may be '...' (three consecutive periods). This indicates that the function takes a variable number of arguments.

Example

```
int testfunction (int inparam)
{
    return inparam;
}

void testfun2 (int &p, int q)
{
    p = 34;
    q = 23;
}
...
int i = 1, j = 2;
testfun2(&i, j);
```

After the call to *testfun2*, *i* will be 34 and *j* will be 2.

2.5.2 Function declarations

Function declarations describe a function's interface allowing the function to be used before the function definition occurs. The function return type and parameter types must match those specified in the function definition.

Syntax

```
datatype ident '(' formal_parameters ')' ';' ;'
```

formal_parameters are defined as in 2.5.1.

Example

```
int testfunction(int inparam);
void testfun2(int &p, int q);
```

2.5.3 Function overloading

A function name in *t* need not be unique. Like in C++, a function may be overloaded by defining several different functions with the same name but different formal parameters. However, each definition of the overloaded function must have the same return type.

When parsing a function call, the compiler will find the set of overloaded functions that could fit the arguments, with or without implicit casting. Then the compiler will begin a closer examination of the matches starting with the first argument. For every argument, any function that does not match as well as the closest matching function found will be excluded from further testing. If the compiler resolves to a single candidate, that candidate is chosen. Otherwise, an error is reported.

Exact matches are always preferred. Arrays, structures, and references need exact matches except for:

- **byte** and **char**, which are interchangeable (but with preference for an exact match)
- **CanMessage** and **CanMessage_X** (database defined message), where a formal parameter of type **CanMessage** can accept a **CanMessage_X**, but not the reverse.

A string constant matches both a constant **char** array and a constant **byte** array, with preference for the former.

For numeric types, automatic casting is done as necessary by order of preference (higher number preferred):

Actual parameter	Formal parameter	Preference
int	int / byte / char	2
	float	1
float	float	2
	int / byte / char	1

Note: There are no **byte/char** actual parameters, since these types are always implicitly cast to **int** when used. Thus, the compiler will not allow attempts to overload functions based on these types.

This means that functions can not differ only in what integer type a parameter has. While **byte/char** are considered identical for overloading purposes, **int** versus either of them is disallowed if there are no other distinguishing (as in non-castable) parameters.

Example

```
int f(int a, int b);
int f(int a);
int f(float a, byte b[]);
...
f(2, "Hi");      // '2' will be cast to float.
f(M_PI);         // M_PI will be cast to int.
```

2.6 Statements

2.6.1 The `if` Statement

If the conditional expression (*cond_expr*) evaluates to a nonzero value, the condition is considered to be true and the *code_or_block_1* is executed. Otherwise, *code_or_block_2* is executed, if available.

Syntax

```
if '(' cond_expr ')'
    code_or_block_1
[ else
    code_or_block_2 ]
```

Example

```
if (speed >= 90) {
    printf("OK!");
} else {
    printf("Speed up!");
}
```

2.6.2 The `while` Statement

As long as *cond_expr* evaluates to a nonzero value, the conditional expression is considered true and *code_or_block* is executed. *code_or_block* is executed after evaluating *cond_expr*. If the first evaluation of *cond_expr* evaluates to false, *code_or_block* is never executed.

Syntax

```
while '(' cond_expr ')'
    code_or_block
```

Example

```
while (speed++ < 90) {
    printf("Speed up!");
}
```

2.6.3 The `do` Statement

While *cond_expr* evaluates to a nonzero value the expression is considered true and *code_or_block* is executed. *code_or_block* is executed before *cond_expr* is evaluated. The **do** statement guarantees that *code_or_block* is always executed at least once, unlike the **while** statement described above.

Syntax

```
do
    code_or_block
while '(' cond_expr ') ' ';'
```

Example

```
do {
    printf("Speed up!");
} while (speed++ < 90);
```

2.6.4 The `for` Statement

First, *assignment1* is executed. For as many iterations as *cond_expr* evaluates to true, *code_or_block* is executed. The statement '*assignment2*' is executed after each iteration of the loop. If the first evaluation of *cond_expr* evaluates to false, *assignment2* and *code_or_block* are never executed.

Like in C++, *assignment1* may contain variable definitions. Variables defined in this manner will only have scope local to the loop.

Syntax

```
for '(' [ assignment1 ] ';' cond_expr ';' [ assignment2 ] ') '
    code_or_block
```

Example

```
for(int x = 0; x < array.count; x++) {
    array[x] = x * 2;
}
```

Note: In *t*, *cond_expr* cannot be empty.

2.6.5 The `switch` Statement

For selection between multiple choices, the **switch** statement can be useful. Only constant integer case expressions are allowed (**float** data types are cast to **int**). A **default** case will be executed if *expr* does not evaluate to any of the defined constant case expressions.

Like C/C++, the keyword **break** must be placed at the end of a case, or execution will fall through to the next case.

Syntax

```
switch '(' expr ') ' '{'
    cases
    '}'
```

Example


```
switch (x) {
case 1:
    printf("One\n");
    break;
case 2:
    printf("Two\n");
    break;
default:
    printf("Unknown\n");
    break;
}
```

2.6.6 The `return` Statement

The `return` statement terminates the function currently executing. Program control is returned back to the function from which the terminated function was called. A function that declares a return type of `void` cannot supply an *expr* as shown below. A function that declares a return type that is not `void` must supply an *expr*.

Syntax

```
return expr ';'
return ';' 
```

2.6.7 The `break` Statement

A `break` statement terminates the nearest enclosing `while`, `do`, `for`, or `switch` statement. Execution resumes at the statement immediately following the terminated statement.

Syntax

```
break ';' 
```

Example

```
for (x = 0; x < array.count; x++) {
    if (array[x] == searched_value) {
        break;
    }
}
```

2.6.8 The `continue` Statement

A `continue` statement causes the current iteration of the nearest enclosing `while`, `do`, or `for` statement to terminate. Unlike the `break` statement, the `continue` statement terminates only the current iteration.

Syntax

```
continue ';' 
```

Example

```
for (x = 0; x < array.count; x++) {
    if (array[x] != searched_value) {
        continue;
    }
    count++;
    //... do some processing
}
```

2.7 Expressions

The following operators are defined. They are listed in the order of decreasing priority.

Operator	Meaning	Priority
(expr)	Parenthesis can be used to group expressions	13
++ variable	Increase the variable by one, then use the resulting value	12
-- variable	Decrease the variable by one, then use the resulting value	12
variable ++	Use the variable's value, then increase it by one	12
variable --	Use the variable's value, then decrease it by one	12
function call	Not an operator; inserted to show its relative priority	11
!	Logical NOT	10
~	Bitwise NOT	10
unary -	Define/change the sign of the operand	10
unary +	Define/change the sign of the operand	10
(type)	Cast to a given type	9
*	Multiply	8
/	Divide	8
%	Modulo	8
Binary +	Addition	7
Binary -	Subtraction	7
&	Bitwise AND	6
	Bitwise OR	5
^	Bitwise XOR	5
<<	Left shift	4
>>	Right shift	4
==	Equal	3
!=	Not equal	3
>=	Greater than or equal	3
<=	Less than or equal	3
<	Less than	3
>	Greater than	3
&&	Logical AND	2
	Logical OR	1

The operators `*`, `/`, `%`, `+`, `-`, `&`, `|`, `^`, `<<`, and `>>` may be used in a more compact form when combined with the assignment operator `=`. The operation and the assignment are combined, like `a += b` which is equal to `a = a + b`.

Note: The `&&` (logical AND) and `||` (logical OR) operators use short-circuit evaluation like C/C++.

Example

```
// There will never be an exception below, since the
// part of the expression to the right of && will not be
// evaluated if the part to the left of it is untrue.
```

```

    if (++i < values.count) && (values[i] != 0) {
        return 1;
    }

```

2.8 Blocks

To group code, enclose the statement(s) in curly braces, { and }. Such a group is called a block.

Variables and constants defined in a block have scope local to that block.

Example

```

{
    int buf[80];    // Invisible outside this block!
    get_report(buf);
    printf("%s\n", buf);
}

```

2.9 Comments

C++ style comments can be used to make the *t* program more readable.

Example

```

// The end of this line is ignored by the compiler
/* and so is everything in here */

```

2.10 Hooks

Hooks are a central part of the *t* language. They are triggered when the specified external or internal event occurs. The language treats hooks as functions, most of which have an implicit parameter called *this*. The *this* parameter has different types for different hooks.

Like a function whose return type is **void**, it is possible to exit from a hook using a `return` statement.

The following hooks are currently defined:

Hook	Triggered when
on start	Program execution is started.
on stop	Program execution is stopped.
on key <i>key</i>	A key is "pressed".
on Timer <i>timer</i>	A timer expires. Several timers can be forced to use the same handler.
on envvar <i>variable</i>	The specified environment variable is updated.
on CanMessage <i>message</i>	A CAN message arrives. Allows for symbols from a database, numbers, constant expressions and wildcards.
on CanTpMessage <i>event</i>	A CAN transport protocol message event occurs. Named events can be set up per transport protocol handle for: - transmission completed - start of arrival - arrival completed
on exception	An exception has occurred.

2.10.1 Hook Execution Order

The execution order of the hooks is independent of the order the hooks appear in the program file, except for hooks of the same type. Hooks of the same type are executed in the order they appear in the file.

When the program is loaded, the following happens:

1. The variables inside any **variables** section are initialized.

When the program execution is started, the following happens:

1. The **on start** hook is run.

When the program execution is stopped, the following happens:

1. The **on stop** hook is run.

Note: In the current implementation, both the **init** and **start** hooks run when the program starts.

2.10.2 Hook Details

2.10.2.1 start

Syntax

```
on start block
```

Description

The **on start** code block is run when program execution starts.

Example

```
on start {  
    printf("Program starting...\n");  
}
```

2.10.2.2 stop

Syntax

```
on stop block
```

Description

The **on stop** code block is run when program execution is stopped.

2.10.2.3 key

Syntax

```
on key key block
```

Description

The **on key** code block is executed when the keyboard key(s) specified by the *key* argument is pressed. If more than one **on key** block is defined with the same *key*, the code contained in all the blocks will run in sequence.

As a special case, an asterisk '*' can be used as the *key* argument. This means that each time a key is pressed, the code block will be executed if a more specific **on key** for the "pressed" key does not exist.

Another special case is an asterisk in brackets. Using "[*]" as the *key* argument is the same as '*' except the code block will always be executed even if there is a more specific **on key** for the key pressed. The more specific **on key** will then be executed as normal.

Within the block, you can use `this` to refer to the key that triggered the hook.

Example

```
on key 'A' {
  // handle 'A' keys
}

on key * {
  // Handle any key, except 'A' and '*'
}

on key '*' {
  // Handle key '*'
}

on key [*] {
  // Handle any key
  printf("%c' pressed\n", this);
}
```

2.10.2.4 Timer

Syntax

```
on Timer timer block
```

Description

The **on Timer** block is automatically bound (by name) to a previously defined `Timer` variable. A timer may also be manually bound using `timerSetHandler()`. When using an array of timers, all timers will automatically bind by name to the same **on Timer** handler, but you can manually bind the timers to different **on Timer** blocks.

The **on Timer** code is run when the timer specified by the *timer* argument expires. If a timer expires without being bound to an **on Timer** block, the expiration will be ignored and no exception will be thrown.

By default, timers in *t* are single-shot. This means the timer expires only once and can then be re-enabled if required. Since the timer handler could have been delayed for various reasons, restarting the timer inside of the handler is likely to cause drift in the interval over time.

A timer can be setup as periodic. In this case, the timer can execute at a certain interval a number of times, or forever. While the timer handler's invocation could still be delayed, the next interval will be suitably shortened to compensate.

Within the *block*, `this` refers to the actual timer that expired and gives access to the timeout value and a user settable timer id.

The minimum timeout for a timer is 1 millisecond and only one timer can be active per millisecond in the program. This means that if two timers expire at the exact same millisecond, one of them will be delayed 1 millisecond.

Example

```
variables {
  Timer msgTimer;
}

on start {
  msgTimer.timeout = 100; // 100 ms
  msgTimer.id      = 42;
  timerStart(msgTimer);
}
```

```

}

on Timer msgTimer {
    timerStart(msgTimer);
    printf("Timer %d expired\n", this.id);
}

```

2.10.2.5 **envvar**

Syntax

```
on envvar envvar block
```

Description

The **on envvar** hook is called when someone has updated an environment variable. In this context, updated includes the setting of the same value as before.

To get the value of *envvar*, use `envvarGetValue()`.

Example

```

envvar {
    char Msg[128];
    int Value;
}

on envvar Value {
    char tmp_msg[128];
    int v;
    envvarGetValue(Msg, tmp_msg);
    envvarGetValue(Value, &v);
}

```

2.10.2.6 **CanMessage**

Syntax

```

on [ filter ] CanMessage [ channel ] message [ mask ] block
on [ filter ] CanMessage [ channel ] errorframe block

```

where

```

filter := prefilter
         | postfilter
channel := '<' '*' '>'
         | '<' constant_expression '>'
message := ident
         | number [ x ] [ r ]
         | '(' constant_expression ')' [ x ] [ r ]
         | '*'
         | '[' '*' ']'
mask := '&' number
         | '&' '(' constant_expression ')'

```

Description

The **on CanMessage** code block is executed when the CAN message specified arrives.

Using `postfilter` is equivalent to not specifying a *filter*. In this case the `on CanMessage` hook is invoked after any trigger block filtering (see section 2.26.1). With `prefilter`, the hook is invoked before the trigger block filtering.

If no *channel* is specified, the CAN channel associated with the program is implicitly used. A `*` signifies reception from any channel.

For *message*, the *number* and *constant_expression* correspond to the identifier of the CAN message. They can be followed by one or more of the following letters:

- `X` or `x` The identifier is an extended (29-bit) identifier.
- `R` or `r` The message is a remote frame.

When *ident* is used, the string is expected to correspond to a message definition in a CAN message database specified when compiling (see section 1.3.6).

As a special case, an asterisk `*` can be used as the *message*. This means that each time a CAN message arrives the code block will be executed if a more specific `on CanMessage` block does not exist for the received message identifier.

Another special case is an asterisk in brackets. Using `"[*]"` is the same as `*` except the code block will be executed even if a more specific `on CanMessage` block for the received message identifier. The more specific `on CanMessage` block will then be executed as normal.

When specified, *mask* is used to decide which bits in the incoming message identifier should be checked against the *message*. Any bits not set in the *mask* are irrelevant to the comparison.

Within the *block*, `this` is used to reference the CAN message that triggered the action.

Error frames are only received when using the special `on CanMessage errorframe` hook. In this case, the contents of `this` are undefined except for the `canMSG_ERROR_FRAME` flag.

Example

```

on CanMessage<*> 0xF00400x & 0xf00fff {
    // Handle all messages with 29-bit id's, where
    // (id & 0xf00f00) == 0xF00400 (hex)
}

on CanMessage 123rx {
    // Handle remote request of 29-bit identifier 123 (decimal)
    // on the channel associated with the program.
}

on CanMessage (msgId) {
    // Handle messages with message id defined by the constant
    // msgId on the channel associated with the program.
}

on prefilter CanMessage TrigData {
    // Handle the TrigData (database name) message, before the
    // trigger mechanism, on the channel associated with the

    // program
}

on CanMessage<0> (engine_msg + 3) x {
    // Handle engine message 4 on channel 0 here.
}

on CanMessage * {
    // Handle any message, except 0xF00400x, 123rx, ...
    // on the channel associated with the program.
}

```

```

    }

    on CanMessage<1> [*] {
        // Handle any message on channel 1.
    }

```

2.10.2.7 CanTpMessage

Syntax

```
on CanTpMessage name block
```

Description

The **on CanTpMessage** block is executed when an event connected to *name* occurs. The connection is established using `canTpSetHandler()`.

Within the *block*, `this` refers to a `CanTpMessage` where a result code and a user settable timer id are always present. For a successful reception event, the data member is also populated.

Example

```

variables {
    CanTpHandle cantp;
}

on start {
    canTpOpen(cantp, 4, 6, "ISO-15765",
              iso15765_Fixed | iso15765_Physical);
    canTpSetAttr(cantp, iso15765_RxTimeout, 1000);
    canTpSetHandler(cantp, "cantpHandler",
                   iso15765_RxIndication);
}

on CanTpMessage "cantpHandler" {
    if (this.result != canTp_OK) {
        printf("CanTp receive error: %d\n", this.result);
    } else {
        printf("Received: %s\n", this.data);
    }
}

```

2.10.2.8 exception

Syntax

```
on exception block
```

Description

The **on exception** code block is executed when an unrecoverable situation within a function occurs.

block can contain variable definitions; such definitions are local to *block*. If more than one **on exception** code block is defined, each code block will execute in sequence.

Within *block*, `this` references the error that triggered the exception (see section 2.33.5). Only read access is allowed, since `this` is constant in an **on exception** block.

Execution of the code directly after the function call that triggered the exception will be aborted.

Example


```

// Example of an exception handler that notifies the user
// by blinking the power LED.

variables {
    Timer ex_timer;
    LedHandle ex_led;
    int ex_led_state[2] = {LED_STATE_OFF, LED_STATE_ON};
}

on exception {
    ledOpen(ex_led, LED0);
    ex_timer.timeout = 777; // ms
    ex_timer.id = 0;
    timerStart(ex_timer, FOREVER);
}

on Timer ex_timer {
    ledSet(ex_led, ex_led_state[this.id]);
    this.id ^= 1;
}

```

Example

```

void dump_exception(const ExceptionData e);
void dump_ints(const int data[]);

on exception {
    dump_exception(this);
}

// Exception data dump in the same format as the built-in one.
// (See sample files for a version that uses no local data.)
void dump_exception (const ExceptionData e)
{
    printf("Execution error %d at %d after %d cycles.\n",
           e.error, e.pc, e.cycle);
    if (e.line) {
        printf("Execution stopped after/at line %d.\n", e.line);
    } else {
        printf("Line number information excluded from code.\n");
    }
    printf("Stack (%d):\n", e.stack.count);

    int fp = e.locals;
    for(int sp = e.stack.count - 1; sp >= 0; sp--) {
        int frame_size = e.stack[fp - 1];
        if (sp >= fp + frame_size) {
            printf("%4d: 0x%08x %11d\n", sp, e.stack[sp], e.stack[sp]);
        } else {
            if (frame_size) {
                printf("Locals (%d @ 0x%08x):\n",
                       frame_size * 4, e.stack_base + fp * 4);
                dump_ints(e.stack[fp, frame_size]);
            }
            sp -= frame_size;
            sp--;
            fp = e.stack[sp--];
            if (sp >= 0) {
                printf("Return address: %d\n", e.stack[sp]);
            }
        }
    }
}

```

```

    }
  }
}

printf("Globals (%d):\n", e.globals.count);
dump_ints(e.globals);
}

// Dump integers with address, hex and ASCII.
void dump_ints (const int data[])
{
  char buf[80];
  for(int i = 0; i < data.count; i += 4) {
    int j;
    for(j = 0; (j < 4) && (i + j < data.count); j++) {
      sprintf(buf + j * 9, "%08x ", data[i + j]);
    }
    buf[j * 9 .. 4 * 9 - 1] = ' ';
    buf[4 * 9, 16] = '.';
    for(j = 0; (j < 16) && (i + j / 4 < data.count); j++) {
      byte b = data[i + j / 4] >> (8 * (j % 4));
      if ((b >= 32) && (b < 127)) {
        buf[4 * 9 + j] = b;
      }
    }
    buf[4 * 9 + j] = '\0';
    printf("%04x: %s\n", i * 4, buf);
  }
}

```

2.11 Using CAN Databases

When you connect one or more CAN database files to an interface, all messages in the databases become available as types, prefixed by the string:

```
CanMessage_
```

For example, if a message *SpeedData* is defined in the database, you can define a variable *spd* like this:

```
CanMessage_SpeedData spd;
```

Assuming the message *SpeedData* consists of the signal *Speed*, you can then use

```
spd.Speed.Raw
```

or

```
spd.Speed.Phys
```

to access the *Speed* signal in the message. The former will access the “raw” value of the signal - the actual contents of the CAN message regardless of the scaling defined in the database. The latter will access the scaled value according to the database definition.

Floating point values of 16, 32 and 64 bit size are supported (IEEE754-2008), as specified in the database. Since *t* only has a 32 bit floating point type, automatic conversion is done on fetch/store.

2.12 `#include`

t allows a source code file to refer to another code file by including the file. This will behave exactly as if the contents of the included file had been inserted at that point in the source code.

Syntax

```
#include " file_name "
```

2.13 `#error`

To enforce compilation errors, for example while using conditional compilation (see 2.15), a special directive can be used. The supplied text will be output by the compiler as an error message.

Example

```
#error too large buffer requested.
```

2.14 `#warning`

To enforce compilation warnings, for example while using conditional compilation (see 2.15), a special directive can be used. The supplied text will be output by the compiler as a warning message.

Example

```
#warning buffer may be too small
```

2.15 *Conditional compilation*

Unlike C/C++, *t* does not have a pre-processor. However, the ability to do conditional compilation, even controlled by options passed to the compiler on the command line, is still available.

2.15.1 Command line options

Both integer and string values can be passed on the command line. They end up as constant declarations in a compiler generated **variables** section, situated just before the first line of user code. There is no limit to the number of constants set up this way.

Syntax (on the compiler command line)

```
-Dconstant_integer_name=number  
-Dconstant_string_name[]=text
```

Example

```
scc -Dcount=10 -Dsize=80 "-Dgreeting[]=Hello World!" program.t
```

Will cause the following to be compiled as if it had been at the top of source code file:

```
variables {  
    const int count = 10;  
    const int size = 80;  
    const char greeting[] = "Hello World!";  
}
```

The text given for a named constant is copied into the **variables** section as is, without any checking whatsoever. Any errors will be caught by the compiler and the reported line number will be consistent with the example **variables** section above.

2.15.2 #if / #else / #endif

Conditional compilation in *t* is done as part of the language itself. Thus, any named integer constant in scope is at the disposal of a conditional. No expressions are allowed in the conditionals themselves, but the full constant expression calculation capabilities of *t* can be used earlier to set things up.

As in other Boolean expressions, zero is considered false and any other value true.

Nesting of conditional compilation directives is not supported.

Syntax

```
#if ( ident | integer )
...
[ #else ]
...
#endif
```

Example

```
variables {
    const int dump_data = DEBUG && VERBOSE;
}

#if dump_data
void dumpToFile(int data[])
{
    ...
}
#else
void dumpToFile(int data[]) { /* Do nothing */ }
#endif
```

2.15.3 #ifdef / #ifndef

Conditional compilation in *t* is done as part of the language itself. Thus, any identifier in scope is considered defined.

Nesting of conditional compilation directives is not supported.

Syntax

```
#ifdef ident
...
[ #else ]
...
#endif

#ifndef ident
...
[ #else ]
...
#endif
```

Example

```
#ifndef output
void output(char *text)
{
    printf("%s\n", text);
}
#endif
```

```

#ifdef DEBUG
variables {
    char dbgbuf[1024];
}
#endif

```

2.16 Predefined Output Functions

2.16.1 printf

Syntax

```
void printf(const char fmt[], ...)
```

Description

Prints the string *fmt* to the standard output. The string *fmt* may contain formatting characters just like the C language `printf()`.

If the standard output is not displayed anywhere, you will not see the output from the `printf` calls.

The format string can contain regular characters and conversion specifiers. Regular characters are simply printed. Conversion specifiers begin with a percent symbol and are shown below:

Conversion specifier	Meaning
%d	Prints argument as a signed decimal number
%c	Prints argument as a character
%s	Prints argument as a string
%x	Prints argument as a hexadecimal number
%u	Prints argument as an unsigned decimal number
%f, %g	Prints argument as a floating-point number
%%	Prints a percent sign

%d, %u, %x, and %s can take a number directly after the %, which specifies the number of characters the result should use (right justified). %d, %u, and %x can have an initial zero that specifies that the fill characters to the left should be zeroes instead of the default space.

2.17 Predefined File I/O Functions

2.17.1 fileOpen

Syntax

```

int fileOpen(FileHandle file, const char filename[])
int fileOpen(FileHandle file, const char filename[], int flags)

```

Description

Opens the file with the given *filename* using the specified *flags*. The handle *file* can then be used to refer to the file.

The possible flags are:

- `OPEN_READ` Only read is allowed from the file
- `OPEN_WRITE` Read and write is allowed

- `OPEN_APPEND` Like `OPEN_WRITE`, but an automatic seek to the end is done
- `OPEN_TRUNCATE` Like `OPEN_WRITE`, but any previous contents are deleted

No value in *flags* is equivalent to `OPEN_WRITE`.

Returns

A negative error code is returned on failure.

2.17.2 fileClose**Syntax**

```
void fileClose(FileHandle file)
```

Description

Closes the file whose handle is *file* (previously returned from `fileOpen`).

2.17.3 fileGets**Syntax**

```
int fileGets(const FileHandle file, char buffer[])
int fileGets(const FileHandle file, char buffer[],
             int bufferSize)
```

Description

Reads a line from *file*. The result is placed in *buffer*. You may specify the maximum length to read.

Returns

The number of characters read is returned when successful. Otherwise, a negative error code is returned.

2.17.4 fileReadBlock**Syntax**

```
int fileReadBlock(const FileHandle file, char buffer[])
int fileReadBlock(const FileHandle file, char buffer[],
                 int bufferSize)
```

Description

Reads a block from *file*. The result is placed in *buffer*. You may specify the length to read.

Returns

The number of characters read is returned when successful. Otherwise, a negative error code is returned.

2.17.5 filePuts**Syntax**

```
int filePuts(const FileHandle file, const char buffer[])
int filePuts(const FileHandle file, const char buffer[],
             int bufferSize)
```

Description

Writes the string *buffer* to *file*. You may specify the maximum length to write.

Returns

The number of characters written is returned on success. Otherwise, a negative error code is returned.

2.17.6 fileWriteBlock

Syntax

```
int fileWriteBlock(const FileHandle file, const char buffer[])
int fileWriteBlock(const FileHandle file, const char buffer[],
                   int bufferSize)
```

Description

Writes the block *buffer* to *file*. You may specify the length to write.

Returns

The number of characters written is returned on success. A negative error code is returned on failure.

2.17.7 fileSeek

Syntax

```
int fileSeek(const FileHandle file, int offset, int whence)
```

Description

Repositions the offset of the *file*. The *whence* argument determines how *offset* is used to reposition. If *whence* is `SEEK_CUR`, the new position is *offset* bytes from the current position. If *whence* is `SEEK_SET`, the new position is *offset* bytes from the start of the file. If *whence* is `SEEK_END`, the new position is *offset* bytes from the end of the file.

Note: `fileSeek` will not extend a file. For example, a `fileSeek` beyond the end of the file will set the position to the end of file.

Returns

If the operation was successful, the resulting offset location, in bytes, from the beginning of the file is returned. Otherwise, a negative error code is returned.

2.17.8 fileDelete

Syntax

```
int fileDelete(const char filename[])
```

Description

Deletes the file whose name is passed in *filename*.

Returns

A negative error code is returned on failure.

2.18 Predefined XML Functions

The XML parser is not a full-featured one, but the parser can handle data organized in structure/array form and attributes. Unless otherwise specified, parsing is restarted from the beginning of the data for every access.

The XML parser is not capable of accessing extra data that comes after a sub-tag. But this should never happen in an XML file that describes an actual data structure. However, this practice is common in XHTML, for example to add a line break (“`
`”).

Also, the XML parser cannot deal with comments, processing instructions, entities, and other things that are not necessary to describe a data structure.

2.18.1 xmlOpen

Syntax

```
int xmlOpen(XmlHandle handle, const char filename[])
```

Description

Opens the file with the given *filename* for XML access using `OPEN_READ` mode. The *handle* can then be used to refer to the file.

Returns

A negative error code is returned on failure.

2.18.2 xmlClose**Syntax**

```
void xmlClose(XmlHandle handle)
```

Description

Closes the XML access *handle*.

2.18.3 xmlGet**Syntax**

```
int xmlGet(XmlHandle handle, char buf[], ...)  
int xmlGet(XmlHandle handle, char buf[], int &position, ...)
```

Description

Finds a specific item or attribute in the XML data referred to by *handle*. The supplied *buf* will be used to store the result. Optionally, the *position* in the XML data where searching for the tag is to begin can be specified. The *position* will then be updated to point to where the tag was found.

The field/attribute to search for is specified using a sequence of strings, or string and number pairs.

- Strings are checked against the XML tags hierarchically.
As a special case, the very last string may instead refer to an attribute.
- Numbers signify indexing into a sequential list of the same tag type.
A missing index is equivalent to 0.
- At the very end, `XML_ATTR` can be used to specify that the final string is an attribute to access.
- `XML_DATA` may optionally be used at the end to specify that the tag data should be accessed.
- An asterisk ('*') may be used to match an arbitrary tag/attribute string.
Indexing works as expected, with any tag/attribute at the same hierarchical level matching.
If `XML_TAG` is used at the end, the name of the tag will be fetched instead of the tag's data.

Returns

Returns the number of bytes fetched on success. A negative error code is returned on failure.

Example file

```
<Whatever>Aberforth  
  <Misc>Bellatrix  
    <Someone>Cedric</Someone>  
  </Misc>  
  <Data>  
    <Something>Dedalus</Something>  
    <Something>Ellerby</Something>  
    <Nothing/>  
  </Data>  
  <Data>  
    <Something>Fulbert</Something>
```



```
    <SomethingElse what="Gellert" who="Hagrid" />
  </Data>
</Whatever>
```

Example *t* code

```
variables {
  char name1[80];
  char name2[80];
}
on start {
  name1[0] = 0;
  name2[0] = 0;
  XmlHandle file;
  if (xmlOpen(file, "params.xml") >= 0) {
    xmlGet(file, name1, "Whatever", "Data", "Something", 1);
    xmlGet(file, name2, "Whatever", "Data", 1, "SomethingElse",
           "who", XML_ATTR);
    xmlClose(file);
  }
}
```

2.19 Predefined Math Functions

The trigonometric functions for sine, cosine, and tangent of an angle, assume the argument is expressed in radians. In the same way, the return values of the arc functions are expressed in radians.

So, when your angle is expressed in degrees, this is the formula:

$$\text{radians} = \text{degrees} * \text{M_PI} / 180$$

2.19.1 `sin`

Syntax

```
float sin(float x)
```

Returns

Returns the sine of *x*. The return value will be between -1.0 and 1.0, inclusive.

An exception is thrown if *x* is infinite.

2.19.2 `cos`

Syntax

```
float cos(float x)
```

Returns

Returns the cosine of *x*. The return value will be between -1.0 and 1.0, inclusive.

An exception is thrown if *x* is infinite.

2.19.3 `tan`

Syntax

```
float tan(float x)
```

Returns

Returns the tangent of x .

An exception is thrown if x is infinite.

2.19.4 asin

Syntax

```
float asin(float x)
```

Returns

Returns the arc sine of x ; that is the value whose sine is x .

An exception is thrown if x falls outside the range -1.0 to 1.0 . Otherwise, the function returns the arc sine in radians. The returned value is mathematically defined to be between $-M_PI/2$ and $M_PI/2$, inclusive.

2.19.5 acos

Syntax

```
float acos(float x)
```

Returns

Returns the arc cosine of x ; that is the value whose cosine is x .

An exception is thrown if x falls outside the range -1.0 to 1.0 . Otherwise, the function returns the arc cosine in radians. The returned value is mathematically defined to be between 0 and M_PI , inclusive.

2.19.6 atan

Syntax

```
float atan(float x)
```

Returns

Returns the arc tangent of x ; that is the value whose tangent is x .

The function returns the arc tangent in radians. The returned value is mathematically defined to be between $-M_PI/2$ and $M_PI/2$, inclusive.

2.19.7 abs

Syntax

```
float abs(float x)
```

Returns

Returns the absolute value of the argument.

2.19.8 ceil

Syntax

```
float ceil(float x)
```

Returns

Returns the smallest integer value not less than x .

2.19.9 floor

Syntax

```
float floor(float x)
```

Returns

Returns the largest integer value not greater than x .

2.19.10 randomize**Syntax**

```
void randomize()  
void randomize(int seed)
```

Description

Initializes the random number generator with a physically random (see section 2.27.4) seed or with a specified one.

The sequence of numbers returned by `random()` after specifying a given *seed* is always the same.

2.19.11 random**Syntax**

```
int random(int x)
```

Description

The WELL512 pseudo random number generator is used to generate a random number modulo x . Unless `randomize()` is used, the seed at startup is physically random.

While the random numbers given by WELL512 are of very good quality, `cryptoRandom()` should be used if cryptographic security is required.

Ref: *F. Panneton, P. L'Ecuyer, and M. Matsumoto, "Improved Long-Period Generators Based on Linear Recurrences Modulo 2", ACM Transactions on Mathematical Software, 32, 1 (2006), 1-16*

Returns

Returns an integer random number in the interval $0 .. x-1$, for $x > 0$.

The full 32 bit number is given for $x == 0$. For $x < 0$, the results are undefined.

2.19.12 sqrt**Syntax**

```
float sqrt(float x)
```

Returns

Returns the square root of x .

An exception is thrown if x is negative.

2.19.13 exp**Syntax**

```
float exp(float x)
```

Returns

Returns the exponential value of the floating-point parameter x . That is, e to the power of x .

2.19.14 exp10**Syntax**

```
float exp10(float x)
```

Returns

Returns 10 to the power of *x*.

2.19.15 round**Syntax**

```
float round(float x)
```

Returns

Returns *x* rounded to the nearest integer. Numbers half-way between two integers are rounded away from zero.

2.19.16 log**Syntax**

```
float log(float x)
```

Returns

Returns the natural logarithm of *x*.

An exception is thrown if *x* is negative.

2.19.17 log10**Syntax**

```
float log10(float x)
```

Returns

Returns the base 10 logarithm of *x*.

An exception is thrown if *x* is negative.

2.20 Predefined String Functions

Strings in *t* are null-terminated character arrays like in C. This means the last byte contains the character code 0 which represents the ASCII NUL character. The language also takes advantage of the fact that an array's size is known.

If no maximum lengths are specified for these functions, the lengths of the relevant array arguments are used instead. Accesses outside the arrays cannot be caused by these functions.

2.20.1 strlen**Syntax**

```
int strlen(const char s[])  
int strlen(const char s[], int length)
```

Returns

Returns the length of the string *s*, not including any terminating null character. A maximum *length* can be specified.

2.20.2 strcpy**Syntax**

```
int strcpy(char dest[], const char src[])
int strcpy(char dest[], const char src[], int length)
```

Description

Copies the string *src* (adding a terminating null character, if there is room) to the string *dest*. Only the portion that will fit in *dest* is copied. A maximum *length* can be specified.

Returns

Returns the number of characters that were copied.

2.20.3 strcat**Syntax**

```
int strcat(char dest[], const char src[])
int strcat(char dest[], const char src[], int length)
```

Description

Appends the string *src* to the string *dest*, overwriting the terminating null character at the end of *dest*, and adding a terminating null character to the end. Only the portion that will fit in *dest* is copied. A maximum *length* can be specified.

Returns

Returns the number of characters that were appended.

2.20.4 strcmp**Syntax**

```
int strcmp(const char s1[], const char s2[])
int strcmp(const char s1[], const char s2[], int length)
```

Returns

Returns an integer less than zero if *s1* is found to be less than *s2*. Returns an integer equal to 0 if *s1* matches *s2*. Returns an integer greater than zero if *s1* is found to be greater than *s2*. A maximum *length* can be specified.

2.20.5 sprintf**Syntax**

```
int sprintf(char str[], const char fmt[], ...)
```

Description

Behaves like `printf()` but puts the result into the string *str*. The receiving string must be sufficiently long.

Returns

Returns the number of characters put into *str* on success. A negative error code is returned on failure.

2.20.6 atoi**Syntax**

```
int atoi(const char s[])
int atoi(const char s[], int base)
```

Description

Discards any leading white space in the string *s*, then interprets the rest as an integer in ASCII format. Interpreting stops when the end of the string is reached or when a character that cannot be interpreted as part of the integer is encountered. For example, the character is not of the base being used.

Unless specified, *base* defaults to 10.

A base of zero, or one that is too large to be represented using ‘a’ – ‘z’, will cause an exception to be thrown.

Returns

Returns the integer found in the string *s*.

Example

```
code = atoi("11001101011110101", 2);
data = atoi("-84820473");
```

2.20.7 atof

Syntax

```
float atof(const char s[])
```

Description

Discards any leading white space in the string *s*, then interprets the rest as a floating point number in ASCII format. Interpreting stops when the end of the string is reached or a character that cannot be interpreted as part of the floating point number is encountered. Decimal point is accepted, as is ‘e’ notation for exponent.

Returns

Returns the float found in the string *s*.

Example

```
angle    = atof("-2.3425561134");
distance = atof("3.5e5");
```

2.20.8 itoa

Syntax

```
int itoa(int number, char buffer[], int base)
int itoa(int number, char buffer[], int base, int bufferSize)
```

Description

Converts *number* into the specified *base* and places the result in *buffer*. A buffer size can be specified.

Base 10 numbers will be handled as signed. All other bases as unsigned.

Normally bases higher than 10 will have the alphabetic digits in lower case. Negating the base forces alphabetic digits into upper case instead.

Returns

Returns the number of characters put into *str* on success. A negative error code is returned on failure.

2.21 Predefined CAN Related Functions

2.21.1 canWrite

Syntax

```
int canWrite(const CanMessage msg)
int canWrite(int channel, const CanMessage msg)
```

Description

This function queues the CAN message *msg* for transmission. The message is sent on the channel that the program is connected to if *channel* is not specified.

The following members in *msg* must be filled out:

id	The identifier of the message.
dlc	The length of the message. If this value is greater than 8, the value will still be passed on to the CAN driver, which will make an attempt to send using the value. More than 8 data bytes will never be sent, though.
flags	A combination of CAN message flags. See section 2.34.2 for further information.
data	The data in the message.

Other members are ignored.

Returns

Returns a negative error code on failure (queue full is always a possibility).

2.21.2 canGetTimestamp**Syntax**

```
int canGetTimestamp(const CanMessage msg, int scale)
int canGetTimestamp(const CanMessage msg, int scale,
                    int &remainder)
```

Description

Provides access to the timestamp that every received CanMessage contains.

The optional *remainder* will contain the remainder of the integer division of the timestamp by *scale*.

Returns

Returns the result of an integer division of the timestamp (microseconds) from *msg* by *scale*.

The function considers *scale* to be an unsigned integer. A *scale* value of 0 is a special case interpreted as one larger than the maximum unsigned integer. The result is the function will return the high integer of the timestamp in response.

2.21.3 canSetBitRate**Syntax**

```
int canSetBitrate(int bitrate)
int canSetBitrate(int channel, int bitrate)
```

Description

This function sets the bitrate on the CAN bus. The operation is performed on the CAN channel the program is connected to if *channel* is not specified. The various parameters are as follows:

bitrate	The desired bit rate in bits per second.
---------	--

Returns

Returns a negative error code on failure.

2.21.4 canSetBusParams**Syntax**

```

int canSetBusParams(int bitrate, int tseg1, int tseg2, int sjw,
                   int samples)
int canSetBusParams(int channel, int bitrate, int tseg1,
                   int tseg2, int sjw, int samples)

```

Description

This function sets the bit rate on the CAN bus. The function operates on the CAN channel the program is connected to if *channel* is not specified. The various parameters are as follows:

bitrate	The desired bit rate in bits per second.
tseg1	The desired number of quanta in time segment 1 (the number of quanta before the sampling point minus 1).
tseg2	The desired number of quanta in time segment 2 (the number of quanta after the sampling point).
sjw	The synchronization jump width is the maximum number of quanta that the CAN controller will add to or subtract from a bit in order to resynchronize the clock. Valid values are 1,2,3 or 4.
samples	The number of samples to be used per bit. Can be 1 or 3.

Returns

Returns a negative error code on failure.

2.21.5 canGetBusParams**Syntax**

```

int canGetBusParams(int &freq, int &tseg1, int &tseg2, int &sjw,
                   int &samples, int &mode)

int canGetBusParams(int channel, int &freq, int &tseg1,
                   int &tseg2, int &sjw, int &samples,
                   int &mode)

```

Description

Gets the current bus parameters including bus output control mode. The function operates on the CAN channel the program is connected to if *channel* is not specified.

Returns

Returns a negative error code on failure.

2.21.6 canBusOff**Syntax**

```

int canBusOff()
int canBusOff(int channel)

```

Description

This function takes the CAN channel off-bus. The function operates on the CAN channel the program is connected to if *channel* is not specified.

Returns

Returns a negative error code on failure.

2.21.7 canBusOn**Syntax**


```
int canBusOn()  
int canBusOn(int channel)
```

Description

This function takes the CAN channel on-bus. The function operates on the CAN channel the program is connected to if *channel* is not specified.

Returns

Returns a negative error code on failure.

2.21.8 canSetBusOutputControl

Syntax

```
int canSetBusOutputControl(int driverType)  
int canSetBusOutputControl(int channel, int driverType)
```

Description

This function sets the driver type of the CAN channel that the program is connected to if *channel* is not specified.. See section 2.34.5 for valid *driverType* values.

Returns

Returns a negative error code on failure.

2.22 Predefined Timer Related Functions

2.22.1 timerStart

Syntax

```
void timerStart(Timer t)  
void timerStart(Timer t, int periods)
```

Description

Starts the timer *t*. The number of *periods* for which the timer should run automatically can be specified.

A single-shot timer can be reset inside the associated **on Timer** handler to get periodic behavior. However, this will not guarantee a consistent period because any delay in handling the timer will not affect the next period.

When a timer is set into periodic mode (`FOREVER` can be used to specify a limitless number of periods), there may still be delays for any specific handler invocation. But in this case, the delay will automatically be subtracted from the next delay. For *N* periods, the total time will be *N* times the timer's timeout (plus whatever the delay is for the *N*th invocation).

A timer that has a timeout of zero will not be started. This also means that a periodic timer whose timeout is set to zero will stop after the timer's next invocation.

If the timer was already active, the timer will be deactivated first.

Example

```
variables {  
    Timer refresh;  
    int rpm_value = 0;  
}  
  
on start {  
    msgTimer.timeout = 100;           // 100 ms  
    timerStart(refresh, FOREVER);  
}
```

```
on Timer refresh {
    rpm_value++;
    if (rpm_value >= 8000) {
        rpm_value = 0;
    }
}
```

2.22.2 timerCancel

Syntax

```
int timerCancel(Timer t)
```

Description

Stops the timer *t*, if the timer is running.

Returns

Returns a negative error code if the timer was not running.

2.22.3 timerIsPending

Syntax

```
int timerIsPending(const Timer t)
```

Returns

Returns 0 if the timer is not active (i.e. has not been started with `timerStart()`). If the timer is active, the number of milliseconds until the timer is due will be returned. If the timer has expired but the associated handler has not yet executed, the function will return 1).

2.22.4 timerSetHandler

Syntax

```
int timerSetHandler(Timer t, const char hook[])
```

Description

Sets the **on Timer** hook to use for timer *t*.

Often this function is not needed, since an **on Timer** hook can be specified for every timer. However, this function allows dynamically created timers to be connected to a hook, which can sometimes be useful. Or the function can be used to have a single handler for multiple timers.

The timer structure contains an integer id member that can be used to identify which timer caused the handler to be executed. The contents of that member are user defined.

Returns

Returns a negative error code if the specified handler does not exist.

Example

```
variables {
    Timer t;
    Timer t1;
    Timer t2;
}

on start {
    t.timeout = 1000;
```

```
t.id      = 100;
timerStart(t);

t1.timeout = 2000;
t1.id      = 1;
timerSetHandler(t1, "timer_handler");
timerStart(t1);

t2.timeout = 3000;
t2.id      = 2;
timerSetHandler(t2, "timer_handler");
timerStart(t2);
}

on Timer t {
    printf("timer t, id=%d", this);
}

on Timer "timer_handler" {
    printf("timer t1 or t2, id=%d", this);
}
```

2.23 Predefined Environment Variable Functions

2.23.1 envvarSetValue

Syntax

```
int envvarSetValue(EnvVar envvar, int value)
int envvarSetValue(EnvVar envvar, float value)
int envvarSetValue(EnvVar envvar, const char value[])
int envvarSetValue(EnvVar envvar, const char value[], int length)
```

Description

Sets *envvar* to *value*. The actual update will be queued and therefore delayed until the execution of the current hook has finished. A notification that *envvar* is updated is then propagated to all programs who declare the same environment variable and have a corresponding **on envvar**. These **on envvar** hooks are called. A PC connected to the program can also get a notification if `canSetNotify()` is configured correctly.

Do not set the same environment variable that you are handling in an **on envvar** hook, since that would cause an infinite loop.

The environment variables are queued in a special queue where each environment variable can only reside once. Each time an environment variable is updated, the variable is placed at the end of the queue. This means that if the variable was in the queue already, the variable will be moved back with the new value and the older value is lost.

Returns

Returns the number of bytes put into *envvar*.

2.23.2 envvarGetValue

Syntax

```
int envvarGetValue(const EnvVar envvar, int &value)
int envvarGetValue(const EnvVar envvar, float &value)
int envvarGetValue(const EnvVar envvar, char value[])
```

```
int envvarGetValue(const EnvVar envvar, char value[],
                  int maxlength)
```

Description

Retrieves the last known value of *envvar*. The result is undefined if *envvar* is not initialized.

Returns

Returns the number of bytes fetched from *envvar*.

2.24 Predefined CAN Transport Protocol Related Functions

The built-in CAN transport protocols can be used to simplify programming. Data will be transferred in the background, but certain events can be used to notify the program about what is happening.

2.24.1 canTpOpen

Syntax

```
int canTpOpen(CanTpHandle handle, int rxid, int txid,
              const char protocol[])
int canTpOpen(CanTpHandle handle, int rxid, int txid,
              const char protocol[],
              int addr_mode_rx)
int canTpOpen(CanTpHandle handle, int rxid, int txid,
              const char protocol[],
              int addr_mode_rx, int addr_mode_tx)
```

Description

Sets up a CAN transport (CanTP) session, to be referred to via the *handle*, for the specified *protocol* (currently only ISO-15765 is available). Receive and transmit identifiers are given by *rxid* and *txid*, respectively. Unless receive / transmit address mode is specified (via *addr_mode_rx* and, optionally, *addr_mode_tx*), the session will use Normal Physical 29 bit mode.

The possible address modes for ISO-15765 are:

- 11 / 29 bit
- Physical / Functional
- Normal / Fixed / Extended / Mixed
- Local / Remote.

Returns

Returns a negative error code on failure.

2.24.2 canTpClose

Syntax

```
void canTpClose(CanTpHandle handle)
```

Description

Terminates the CanTP session referred to by *handle*.

2.24.3 canTpTransmit

Syntax

```
int canTpTransmit(const CanTpHandle handle, const byte data[])
int canTpTransmit(const CanTpHandle handle, const byte data[],
                  int length)
```

Description

Transmit *data* using the CanTp session referred to by *handle*. Possibly specifying the maximum *length*. For ISO-15765, transmit is not possible when a receive is under way (a separate CanTp session may be used to accomplish that).

The maximum transfer length for ISO-15765 is 4095 bytes.

Returns

Returns a negative error code on failure.

2.24.4 canTpAbort**Syntax**

```
int canTpAbort(const CanTpHandle handle)
```

Description

Abort any current activity on the CanTp session referred to by *handle*.

Returns

Returns a negative error code on failure.

2.24.5 canTpSetHandler**Syntax**

```
canTpSetHandler(const CanTpHandle handle, const char hook[])
canTpSetHandler(const CanTpHandle handle, const char hook[],
                int event)
```

Description

Set a common handler, *hook*, for the CanTp session referred to by *handle*, or specify that the handler should deal with a specific CanTp *event*.

For ISO-15765, the following events are available:

- TxConfirmation Confirmation that a packet has been transmitted, or a failure code.
- RxIndication Data received, or failure code.
- FfIndication Data is incoming.

Returns

Returns a negative error code on failure.

2.24.6 canTpSetAttr**Syntax**

```
int canTpSetAttr(const CanTpHandle handle, int attr, int value)
```

Description

For the CanTp session referred to by *handle*, set the given *attribute* to the specified *value*.

Most of the attributes are protocol specific, but the following applies to all CAN transport protocols:

- canTp_Id A user supplied id (for use when needed). Ends up in CanTpMessage (this) on events.

For ISO-15765 the following are available (prefixed by iso15765_):

- BS Recieve_bs, supplied in transmitted messages.
- STmin Separation Time minimum, supplied in transmitted messages.

- TxTimeout Transmit timeout in milliseconds. Transmit will fail for longer control flow delays.
- RxTimeout Receive timeout in milliseconds. Reception will fail for longer delays.
- WFTmax Wait For Transmit (number of times allowed).
- Channel CAN channel to use (default channel is used otherwise).
- Mask Mask for accepting received CAN messages (also set by `canToOpen()`).
- Code Id for accepting received CAN messages (also set by `canTpOpen()`).

A CAN id matches when:

```
identifier & Mask == Code & Mask
```

Returns

Returns the value set or a negative error code on failure.

2.24.7 `canTpGetAttr`

Syntax

```
int canTpGetAttr(const CanTpHandle handle, int attr)
```

Description

For the CanTp session referred to by *handle*, get the specified *attribute*.

The attributes are protocol specific. See `canTpSetAttr()`.

Returns

Returns the value of the attribute or a negative error code on failure.

2.25 Predefined *t* Program Related Functions

2.25.1 `scriptLoad`

Syntax

```
int scriptLoad(int slot, int channel, const char filename[])
```

Description

Load a *t* program file called *filename*, using default CAN channel *channel*, into free program slot *slot*. The program should be stored on the SD disk. The actual load of the file is done when the current hook has finished. This means the return code only reflects the execution of the `scriptLoad` command.

Returns

Returns zero on success or a negative error code on failure.

Example

```
// This program loads another t program on start

variables {
    char file[10] = "test2.txe";
    int slot = 1; // a program slot that is free
    int channel = 0; // default CAN channel
}

on start {
    printf("LOAD!");
    int res = scriptLoad(slot, channel, file);
}
```

```
    printf("scriptLoad, result=%d\n", res);  
}
```

Note: Depending on hardware, there can be a limited number of program slots for storing loaded programs.

2.25.2 scriptUnload

Syntax

```
int scriptUnload(int slot)
```

Description

Unload the *t* program loaded in program slot *slot*. The actual unload of the program is performed when the current hook has finished. This means the return code only reflects the execution of the `scriptUnload` command.

Returns

Returns zero on success or a negative error code on failure.

2.25.3 scriptStart

Syntax

```
int scriptStart(int slot)
```

Description

Start an already loaded program that resides in program slot *slot*. The actual start of the program is performed when the current hook has finished. This means the return code only reflects the execution of the `scriptStart` command.

Returns

Returns zero on success.

2.25.4 scriptStop

Syntax

```
int scriptStop(int slot)
```

Description

Stop a started program that resides in program slot *slot*. The actual stop of the program is performed when the current hook has finished. This means the return code only reflects the execution of the `scriptStop` command.

Returns

Returns zero on success or a negative error code on failure.

Example

```
// This program will make sure that there is another program  
// running as long as this program is running  
  
variables {  
    int slot = 2; // a program slot that is free  
    int channel = 0; // default CAN channel  
}  
  
on start {  
    scriptLoad(slot, channel, "test3.txe");
```

```
    scriptStart(slot);  
  }  
  
  on stopped {  
    scriptStop(slot);  
    scriptUnload(slot);  
  }
```

2.26 Predefined Logger Related Functions

2.26.1 filterDropMessage

Syntax

```
int filterDropMessage()
```

Description

This function can only be used in logger mode, in program slot 0 in an **on CanMessage** hook. When a CAN message is filtered using `filterDropMessage()`, the message will not be stored in the log file. Nor will the message be handled by the built-in trigger mechanism.

A CAN message stopped in an **on prefilter CanMessage** will not be handled by any **on CanMessage** hook.

Removes the current message from the logger chain.

Returns

Returns a negative error code on failure.

2.26.2 loggerStatus

Syntax

```
int loggerStatus()
```

Description

This function can only be used in logger mode and in program slot 0.

Returns

Returns the current logger status (see section 2.34.10).

2.26.3 loggerStart

Syntax

```
int loggerStart()
```

Description

This function can only be used in logger mode and in program slot 0. This function makes the Kvaser Eagle start recording CAN frames using the current bus settings and stores a trigger event in the log file. If the Kvaser Eagle is already recording, only a trigger event is stored.

Note: This function will not work if "Log everything" is selected in the Memorator configuration.

Returns

Returns the current logger status (see section 2.34.10).

2.26.4 loggerStop

Syntax


```
int loggerStop()
```

Description

This function can only be used in logger mode and in program slot 0. This function makes the Kvaser Eagle stop recording to the log file. The Kvaser Eagle will not stop listening for CAN frames, but will stop recording them.

Note: This function will not work if "Log everything" is selected in the Memorator configuration.

Returns

Returns the current logger status (see section 2.34.10).

2.27 Predefined Crypto Related Functions

2.27.1 cryptoHash

Syntax

```
int cryptoHash(int type, byte digest[], const byte source[])
int cryptoHash(int type, byte digest[], const byte source[],
               int length)
```

Description

Calculates the *digest* of *source* using a specified *type* of hash algorithm. The *length* of the data to be hashed can be specified.

If *digest* is not large enough to hold the hash, bytes to the right will be discarded.

Returns

Returns the normal size of the hash on success. A negative error code is returned on failure.

2.27.2 cryptoEncipher

Syntax

```
int cryptoEncipher(int type, byte dest[], const byte source[],
                  const byte password[])
int cryptoEncipher(int type, byte dest[], const byte source[],
                  int length, const byte password[])
```

Description

Enciphers *source* by applying a specified *type* of cipher algorithm and a *password*. The *length* of the data to be enciphered can be specified. The enciphered result ends up in *dest*.

Depending on the cipher used, there may be length requirements on *source* and *password*.

Returns

Returns a negative error code on failure.

2.27.3 cryptoDecipher

Syntax

```
int cryptoDecipher(int type, byte dest[], const byte source[],
                  const byte password[])
int cryptoDecipher(int type, byte dest[], const byte source[],
                  int length, const byte password[])
```

Description

Deciphers *source* by applying a specified *type* of cipher algorithm and a *password*. The *length* of the data to be deciphered can be specified. The deciphered result ends up in *dest*.

Depending on the cipher used, there may be length requirements on *source* and *password*.

Returns

Returns a negative error code on failure.

2.27.4 cryptoRandom**Syntax**

```
int cryptoRandom()
```

Description

Makes use of physical randomness in the device to generate a random number. When an actual hardware random number generator is not available, some kind of entropy extraction is used instead.

Returns

A 32 bit random number.

2.28 Predefined System Related Functions**2.28.1 sysGetValue****Syntax**

```
int sysGetValue(int type, int &value)  
int sysGetValue(int type, byte data[])
```

Description

Enables the fetching of various system values. The desired information *type* is specified and the result is returned in *value* or *data*.

Returns

Returns a negative error code on failure. The result ends up in *value* or *data*.

2.29 Predefined Customer Data Related Functions**2.29.1 customerdataGetLength****Syntax**

```
int customerdataGetLength(int user_id, int param_id, int &value)
```

Description

Enables the fetching of customer data values. The desired information *user_id(oem_id)* is specified and the length, in byte, of the contents are returned in *value*.

Returns

Returns a positive error code on failure. Returns 0 if ok.

2.29.2 customerdataGetValue**Syntax**

```
int customerdataGetValue(int user_id, int param_id, byte data[])
```

Description

Enables the fetching of customer data values. The desired information *user_id(oem_id)* is specified and the contents are returned in *data*. *Param_id* must be zero (presently not implemented).

Returns

Returns a positive error code on failure. Returns 0 if ok.

2.30 Predefined LED Functions**2.30.1 ledOpen****Syntax**

```
int ledOpen (LedHandle led, int ledNumber)
```

Description

Opens a handle to a LED and reserves the LED for a program. The LED will not be affected by system events as long as the handle is open. The predefined values for *ledNumber* are LED0, LED1, etc. (see section 2.34.12). *Param_id* must be zero (presently not implemented).

Returns

Returns zero on success or a negative error code on failure.

2.30.2 ledClose**Syntax**

```
void ledClose (LedHandle led)
```

Description

Close a handle to a LED and allow the LED to be controlled by system events again.

2.30.3 ledSet**Syntax**

```
int ledSet (const LedHandle led, int ledState)
```

Description

Set the state for a LED. The two available states are LED_STATE_ON and LED_STATE_OFF. For further information see section 2.34.12.

Returns

Returns zero on success or a negative error code on failure.

2.31 Other Predefined Functions**2.31.1 timeGetDate****Syntax**

```
void timeGetDate(tm t)
void timeGetDate(int &time)
void timeGetDate(char buf[])
void timeGetDate(int time, char buf[])
```

Description

Provides time and date as a **struct** `tm` (see section 2.33.3), as an integer, or as a string. The integer represents time in the standard Unix epoch (i.e. the number of seconds since 00:00:00 UTC on 1 January 1970).

When there is no *time* input parameter, the current clock time is used.

If *buf* is too small, characters to the right will be discarded.

2.31.2 `timeGetLocal`

Syntax

```
int timeGetLocal(int scale)
int timeGetLocal(int scale, int &remainder)
```

Description

Accesses the time since the device was started (synchronized with the CAN message timestamps).

The optional *remainder* will contain the remainder of the integer division of the timestamp by *scale*.

Returns

Returns the result of an integer division of the time since the device was started (microseconds) by *scale*.

The function considers *scale* to be an unsigned integer. A *scale* value of 0 is a special case interpreted as one larger than the maximum unsigned integer. This will cause the function to return the high integer of the time value.

2.31.3 `sizeof`

Syntax

```
sizeof(variable)
sizeof(type)
```

Description

This is really a language/compiler feature rather than a function, and is mainly useful for serialization and deserialization (see 2.3).

Only constant, variable and type names are allowed as “parameter”.

Returns

Returns the “packed” size in bytes, suitable for serialization/deserialization (see 2.3 for details), of the specified type or variable.

Note: To get the number of elements in an array, use *.count* instead (see 2.1.5).

2.32 Predefined Symbols

2.32.1 `this`

The symbol `this` can be used inside a message handler to refer to the message being handled. For example:

```
on CanMessage 123 {
    printf("%d\n" this.id);
}
```

This handler would print ‘123’ to the standard output each time a CAN message with identifier 123 arrived. Inside a `CanMessage` handler, `this` has the type `CanMessage`.

2.33 Predefined Types

2.33.1 Timer

```
typedef struct {
    int timeout;      // Period time in milliseconds
    int id;           // User supplied id (for use when needed)
} Timer;
```

Defines a timer.

2.33.2 CanMessage

```
typedef struct {
    byte channel;     // Channel received message arrived on
    byte flags;       // CAN message flags, for extended id etc
    byte dlc;         // Number of bytes (normally) in CAN message
    int id;           // CAN id of message
    byte data[8];     // CAN message data
} CanMessage;
```

Defines a CAN message.

Note: The time stamp on an incoming message is not directly accessible. `canGetTimestamp()` must be used to access the time stamp.

2.33.3 CanTpMessage

```
typedef struct {
    int id;           // User supplied id (for use when needed)
    int result;       // Error code
    byte data[];      // Received data
} CanTpMessage;
```

The data received by a CanTp handler.

2.33.4 tm

```
typedef struct {
    int tm_sec;       // seconds after the minute
    int tm_min;       // minutes after the hour
    int tm_hour;      // hours since midnight
    int tm_mday;      // day of the month
    int tm_mon;       // months since January
    int tm_year;      // years since 1900
    int tm_wday;      // days since Sunday
    int tm_yday;      // days since January 1
    int tm_isdst;     // Daylight Saving Time flag
} tm;
```

Defines the time format.

2.33.5 ExceptionData

```
typedef struct {
    int current_thread; // # of thread where exception occurred
    int error;          // Exception number
    int line;           // Line (if accessible)
    int pc;             // Program counter
}
```

```

int cycle;           // # of executed cycles
int locals;         // Index to local variables on stack
int stack[];        // The stack
int globals[];      // The global variables
int stack_base;     // Base address of the stack
} ExceptionData;

```

This built-in type defines an exception. When received in an **on exception**, the data is **const**.

2.33.6 Types for Database Defined Signals

```

typedef struct {
    int Raw;
    int Phys;
};

typedef struct {
    float Raw;
    float Phys;
};

```

These two built-in types are used to define signals from database files. If your message is defined in a database file, you can manipulate the message's signals by assigning values to their `.Raw` or `.Phys` components. `Raw` means the value actually transmitted in the CAN messages. `Phys` means the physical value, scaled according to the parameters defined for that signal in the database.

The scaling rules are:

$$\begin{aligned} \text{Physical value} &= (\text{raw value}) * \text{factor} + \text{offset} \\ \text{Raw value} &= ((\text{physical value}) - \text{offset}) / \text{factor} \end{aligned}$$

Suppose you have a database where the message *EngineData* contains the signal *EngineTemp* with a range from -50 to +150 degrees C, the offset is specified as -50 and the factor as 0.01, and the data format is an unsigned 16-bit integer.

```
CanMessage_EngineData.EngineTemp.Phys = 75;
```

The assignment above means that a raw value of $(75 - (-50)) / 0.01 = 12500$ would be assigned to the two bytes that make up the *EngineTemp* in the CAN message *EngineData*.

```
CanMessage_EngineData.EngineTemp.Raw = 75;
```

This assignment would just assign 75 to the two bytes that make up the *EngineTemp* signal making a temperature of $75 * 0.01 + (-50) = -49.25$ degrees C.

2.34 Predefined Constants

2.34.1 Maths

M_PI	3.14159265359
M_E	2.71828182846

2.34.2 Timer Period Counts

FOREVER	
ONCE	

2.34.3 CAN Message Flags

canMSG_RTR	Marks a Remote-Frame. Can also be used for transmitting messages.
canMSG_EXT	For received messages, set if the CAN message has an extended identifier. To send a CAN message with a 29 bit identifier, this flag must be set when calling <code>canWrite()</code> .
canMSG_ERROR_FRAME	
canMSG_TXACK	
canMSG_TXRQ	
canMSGERR_OVERRUN	

You can use these values in conjunction with the `flags` field in a CAN message.

2.34.4 CAN Bitrates

canBITRATE_1M	1 Mbit/s	62.5% sample point
canBITRATE_500K	500 kbit/s	62.5% sample point
canBITRATE_250K	250 kbit/s	62.5% sample point
canBITRATE_125K	125 kbit/s	68.7% sample point
canBITRATE_100K	100 kbit/s	68.7% sample point
canBITRATE_83K	83.333 kbit/s	75% sample point
canBITRATE_62K	62.5 kbit/s	68.7% sample point
canBITRATE_50K	50 kbit/s	68.7% sample point

2.34.5 CAN Driver Modes

canDRIVER_NORMAL	
canDRIVER_SILENT	

2.34.6 CAN status

canSTAT_BUS_OFF	
canSTAT_ERROR_PASSIVE	
canSTAT_ERROR_WARNING	
canSTAT_ERROR_ACTIVE	

2.34.7 CAN Transport Protocols**2.34.7.1 Error Codes**

canTp_OK	No error detected.
canTp_TX_ERR_MASK	Covers all transmit errors.
canTp_RX_ERR_MASK	Covers all receive errors.
canTp_GEN_ERR_MASK	Covers all generic errors.
iso15765_OK	No error detected.
iso15765_TIMEOUT_A	
iso15765_TIMEOUT_Bs	

iso15765_TIMEOUT_Cr	
iso15765_WRONG_SN	
iso15765_INVALID_FS	
iso15765_UNEXP_PDU	
iso15765_WFT_OVRN	
iso15765_BUFFER_OVFLW	

2.34.7.2 Events

iso15765_TxConfirmation	
iso15765_RxIndication	
iso15765_FfIndication	

2.34.7.3 Attributes

canTp_Id	User supplied id (for use when needed).
iso15765_BS	
iso15765_STmin	
iso15765_TxTimeout	
iso15765_RxTimeout	
iso15765_WFTmax	
iso15765_Mask	
iso15765_Code	
iso15765_Channel	

2.34.7.4 Address Modes

iso15765_1lbit	
iso15765_29bit	
iso15765_Physical	
iso15765_Functional	
iso15765_Normal	
iso15765_Fixed	
iso15765_Extended	
iso15765_Mixed	
iso15765_Local	
iso15765_Remote	

2.34.8 File Open

OPEN_READ	
OPEN_WRITE	
OPEN_APPEND	
OPEN_TRUNCATE	

2.34.9 File Seek

SEEK_CUR	
----------	--

SEEK_SET	
SEEK_END	

2.34.10 Logger Status

LOGGER_STATE_UNINITIALIZED	This is returned when the logger mechanism isn't running.
LOGGER_STATE_IDLE	This is returned when the Eagle is in logger mode, but not storing anything to the log file. Usually seen when the Eagle is waiting for a trigger to happen.
LOGGER_STATE_STORING	This is returned when the Eagle is waiting for a trigger and the pre-trigger is activated (i.e. buffering the pre-trigger on the disk).
LOGGER_STATE_LOGGING	This is returned when the Eagle is logging data.
LOGGER_STATE_DISK_FULL	This is returned when the disk is full.
LOGGER_STATE_FAULT	This is returned when the logger mechanisms encounters an error that can't be handle, like the disk is removed.
LOGGER_STATE_UNKNOWN	This should not happen.

2.34.11 Crypto

CRYPTO_MD5	MD5 hash
CRYPTO_SHA1	SHA-1 hash
CRYPTO_AES128	AES-128 cipher
CRYPTO_MD5_SIZE	Size of MD5 digest
CRYPTO_SHA1_SIZE	Size of SHA-1 digest
CRYPTO_HASH_SIZE	Maximum size of supported hashes
CRYPTO_AES128_BLOCK	Data block granularity for AES-128

2.34.12 System

SYS_EAN_HIGH	Vendor ID part of EAN (int)
SYS_EAN_LOW	Product ID part of EAN (int)
SYS_SERIAL_NO	Product serial number (int)
SYS_SD_CID	SD card CID information
SYS_SD_CID_SIZE	Size of SD card CID information

2.34.13 User parameters

USER_PARAM_SIZE (not used)	User parameter size number (int)
USER_PARAM (not used)	User parameter (bytearray)
USER_PARAM_MAX_SIZE	Max size of the user parameters

2.34.14 LED Constants

LED0	The first LED
LED1	The second LED
LED2	The third LED

LED3	The fourth LED
LED4	The fifth LED
LED_STATE_ON	Turn the LED on
LED_STATE_OFF	Turn the LED off

2.34.15 Envvar Constants

ENVVAR_MAX_SIZE	4096
-----------------	------

3 Document revision history

Revision	Date	Compiler Version	Changes
2.0	2011-09-19	2.0	Updated for first release.
2.1	2012-10-02	2.1	Clarified overloading (also new compiler behaviour).
	2012-10-04		Documented conditional compilation.
	2012-10-08		On Timer automatically binds to Timer array
	2012-10-24		Added user parameters and improved sysGetValue
	2012-11-01		Added new user parameters functions.
	2012-11-06		Structures may now contain other structures. #error / #warning can be used to generate compiler output. Corrected note regarding the value returned by sizeof(). Added support for database defined CanMessage signals using 16 and 64 bit floating point numbers (IEEE754-2008).
	2012-11-07		Changed user parameter functions to customer data
	2012-11-16		Argument param_id added to customerdata functions
	2012-12-04		Clarified ceil, floor and round functions. Minor updates to some examples.
	2012-12-07		Added reference variables. Added #ifdef / #ifndef. Added structure serialization / deserialization.
3.0	2012-01-30	3.0	Increased compiler version since some of the previous changes were not backwards compatible.
3.1	2013-09-03	3.1	Increased compiler version. Minor bug fixes.