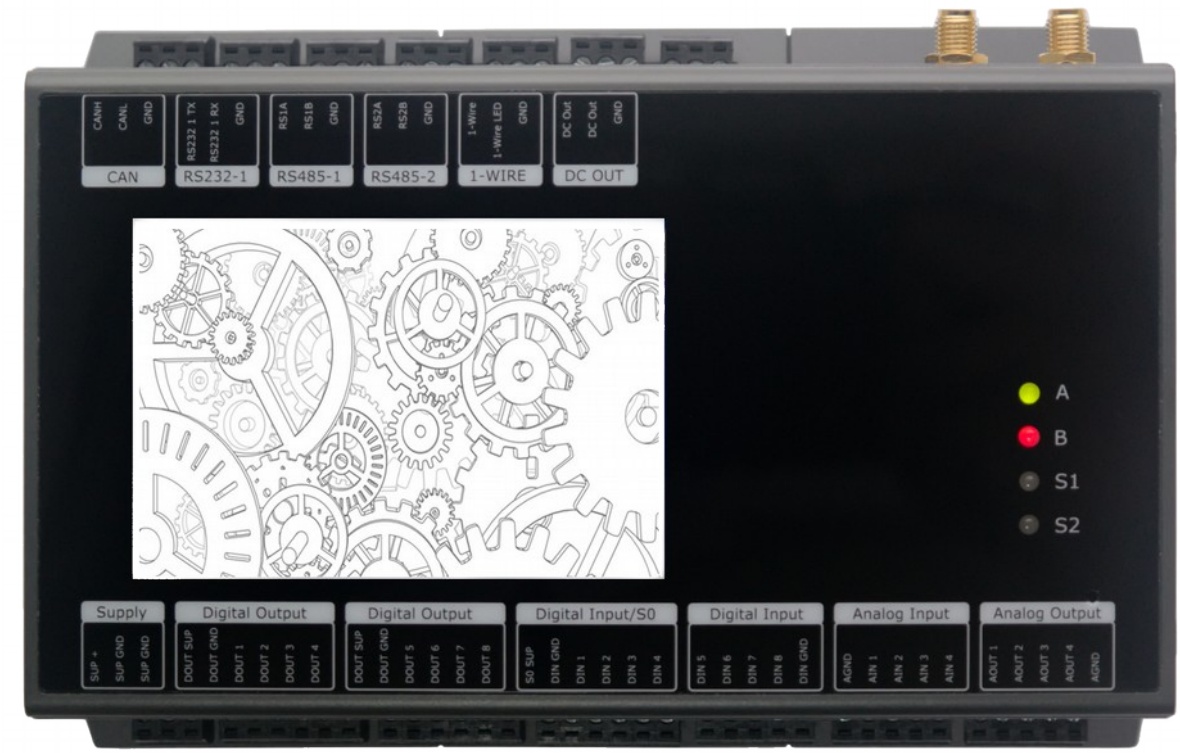
logic IO
The M2M Enabler

RTCU Platform SDK

V1.10



Reference Manual

1 Table of contents

2 Introduction

The **RTCU M2M Platform** is designed to be as open, adaptable and programmable as possible. For most applications the power and flexibility offered with the RTCU IDE is sufficient to realize the most advanced M2M applications, but the need for a deeper customization may arise is certain specialized vertical applications.

Under the NX32L Architecture a mechanism to expand the platform on the Linux system level is offered named Platform Extensions.
With Platform Extensions it is possible to develop two types of extensions to complement the RTCU IDE application:

Program
A program is an application that operates autonomously in a process separate from the RTCU run-time process. Basically such a program has access to most resources and functionality offered by the Linux operating system.
Specialized applications to meet the requirement for advanced protocols, specialized gateways or simple optimized calculation services can be realized.
The recommended way to communicate with a program is to use standard TCP/IP sockets.

Module
Modules are extension libraries that are loaded by the RTCU runtime to natively expand the API with any new functionality that may not be part of the standard API as supplied by Logic IO.

The purpose of this document is to supply all the necessary information and tools allowing development of Platform Extensions using the **RTCU M2M Platform SDK**.

3 Setting up the build environment

3.1 Installation

To set up the system to be able to compile new extension programs and modules, please install a 64 bit Cygwin, using setup-x86_64.exe from https://cygwin.com/. In addition to the default package selection, the "make" package must be selected to be able to build the examples.

Extract the SDK archive to a suitable location with no spaces in the path and change the `cygwin` variable in the file `use.bat` in the SDK to point at the correct path to the Cygwin root folder.

Note: The installation requires a 64-bit Windows system to work.

Note: If a path with spaces is used, building the extensions will fail with error messages such as `\library\build\module.mk: No such file or directory`.

3.2 Starting the environment

To start the build environment, double-click `start_sdk.bat`, which opens a command prompt with the correct paths set.

To build a module or a program, navigate to its folder and run `make`, which will build it unless it is already built and no changes to the source code have been made.

To force a rebuild, either run `make clean` followed by `make` or run `make -B`.

3.3 rtcuexttag.exe

The tool rtcuexttag.exe is used to add tags to the programs and modules, to mark them as valid extension programs and modules.

In addition to the built-in tags, it is possible to define additional tags that will be shown in the IDE when installing the extension and which can be read from the VPL application.

```
rtcuexttag <filename> [options] [metadata]

  options:
    -v                - Show verbose output
    -h                - Show how to use the tool
    -out <filename>   - Store the result in <filename>
```

The metadata is a list of space separated key-value pairs, each defining a single tag. It is possible to use quotes to use values containing spaces.

The examples in the SDK uses the makefiles to define the custom tags, such as the version and description, resulting in a command line that looks like this:

```
rtcuexttag.exe empty.so -out empty.rmx version 1.0 description "An empty
```

```
module"
```

This results in a module, `empty.rmx`, with a version tag of 1.0 and a description tag of “An empty module”.

4 Program extensions

The programs extensions makes it possible to run custom binary code in the RTCU device, by building a stand-alone program and then transferring it to the RTCU device and running it.

The program extensions are stand alone programs, that are built normally and then run through rtcuexttag.exe, which adds a number of tags to the module, marking it as a valid program extension.

To communicate with the VPL application, it is possible to use sockets, files and other similar methods for communicating between processes.

The program extension runs in its own separate process so if it crashes it will not take the entire system down with it.

The program extensions are especially suitable when a program for solving the task already exists or for long running tasks, e.g. for providing services for the VPL application or via external interfaces.

4.1 Execution environment for program extensions

The program starts with the current folder set to the location of the program, typically the `SYSTEM\EXT\` folder on the internal drive.

To navigate to the SD Card or an USB drive, use relative paths, using the drive letter(in lower case) to change the drive, e.g. `..\..\..\a\` to access the SD Card. Note that the drives must be opened from VPL first.

Absolute paths should be avoided as the paths may be changed without notice.

Note that on the internal drive, all file and folder names are case sensitive and all names on it are with uppercase letters.

4.2 Examples

The following two examples of extension programs are available in the SDK:

4.2.1 EchoServer

This program is used to demonstrate a way for a VPL application to communicate with the extension program.

EchoServer sets up a socket and begins listening on it. When the VPL application connects to the socket, it sends "Hello, World!" to the application, waits 5 seconds and closes the socket.

This could be expanded to e.g. send a command to the program which could then respond with the result of the command.

4.2.2 HttpServer

This program demonstrates how an extension program can be used to make data from the VPL application available as a web-page, without having to create the entire server or the HTML page from VPL.

HttpServer is a very simple HTTP server, based on the Simple web-server[1]. When started, it starts serving the files in the `B:\WEB` folder.

The VPL application creates the `B:\WEB` folder and populates it with an `index.htm` file.

The application then starts the server and begins updating the file `B:\WEB\STATUS.TXT` with the current supply voltage as well as the serial number of the device.

`Index.htm` then uses JavaScript to fetch the `STATUS.TXT` file at a fixed interval and updates the page with the current values.

This could e.g. be expanded to VPL creating one or more simple files (perhaps in the JSON format) that are updated when a basic status changes, which is then made available via the web-page.

1https://github.com/labcoder/simple-webserver/blob/master/server.c

5 Extension modules

The extension modules makes it possible to run custom C functions directly from the VPL runtime, similar to how the ordinary platform support functions work.

This makes it possible to add powerful new functions to the VPL applications to e.g. manipulate strings, perform complex calculations or to use custom protocols to communicate over external interfaces.

This is a much more advanced form of extending the runtime than the extension programs and is very likely to cause the system to crash in the case of an error.

5.1 Execution environment for module extensions

The current folder for the extension modules starts out as the location of the internal drive, but as it is shared for the entire process, it can be changed by other extension modules.

The recommended way for finding the path to a file is to use `vplFsMapFile`. It can map both VPL files and devices such as serial ports to the system paths, which can then be used with the normal file operations.

Note that on the internal drive, all file and folder names are case sensitive and all names on it must use uppercase letters.

5.2 Examples

The following example modules are available in the SDK, located in the `modules` folder:

5.2.1 Empty

The `Empty` module is a module with no functions. It is meant as a template for creating new modules.

5.2.2 HelloWorld

The `HelloWorld` module contains one function, `HelloWorld`, which returns the string "Hello World".

5.2.3 dt_mod

The `dt_mod` module shows how different data types can be moved between VPL and C.

It contains a number of functions with different parameters.

The functions `stringToUpper` and `numberToHex` are helper functions. `stringToUpper` sets the

ACCESS variable out to the uppercase version of the string str. numberToHex returns a string containing the value of the given DINT as a hexadecimal value with optional padding to 8 characters.

The functions sintTest, intTest, dintTest, floatTest and boolTest all takes a value and uses it to create two values, one stored in the access value and the other returned from the function.

sbTest populates the given sb_data STRUCT_BLOCK variable with 7 elements.

get_buf and free_buf allocates and frees dynamic memory, to show one possible way to use pointers. Please note that allocating large amounts of memory may cause the system to run out of memory and crash, so care should be taken.

fbTest is an example of a FUNCTION_BLOCK with some variables.

5.2.4 Example

The Example module contains a more complete numberToHex function that the dt_mod module.

It takes a DINT value and the number of characters to pad the number to and returns a string with the padded hexadecimal value.

It is used for the examples in the Creating Functions section.

5.2.5 NTP

The NTP module uses NTP to provide the current time for the wanted timezone.

The module consists of two C-files.

ntp.c contains the NTP client implementation, based on a standalone ntp client[2], changed to be a single function, ntp_get_time.

ntp_mod.c contains the code for the module. It has the function get_time which takes the host name from VPL and passes it on to the ntp_get_time function. If a timezone is specified, it then uses localtime to convert the time into the local time before converting the time to a linsec and returning it.

5.2.6 ext_lib

The ext_lib module is an example of how to use external, static libraries.

The mxml folder contains the mini-XML library[3]. It can be built by calling sh build.sh from the mxml folder, which generates the library libmxml.a.

The ext_lib makefile has the variable STATIC_LIBS set to point at libmxml.a, which causes it to be linked with the rest of the module.

The module itself has a single function that parses an XML file and prints an attribute from it to the device output.

The XML file is generated by the VPL application before it calls the function to parse it.

2https://github.com/lettier/ntpclient/blob/master/source/c/main.c
3https://github.com/michaelrsweet/mxml

5.2.7 Wireless M-Bus

The Wireless M-Bus example module `mbus` is a complex example that also provides the starting point for an API for controlling the Radiocrafts RC1180-MBUS Wireless M-Bus Module that is available on some versions of the NX-400.

For a detailed description of this example, please see the documentation in `modules/mbus/doc/index.html`.

5.2.8 Audio

The Audio example module demonstrates the usage of the `vplAudioStreamRegister` function and how modules can be used to handle streaming audio.

The application waits for an incoming phone call and then it plays the file `B:\SND.WAV` to the caller while recording any incoming audio.

When the call is terminated, it stops the recording and is ready for the next call.

To make sure that there is always sound to play, the file is repeated indefinitely.

The module has two callback functions:

- test_stream_capt reads data from the file `B:\SND.WAV` and passes it on to the stream, which then sends it to the caller.
- test_stream_playb receives the caller audio from the stream and records it to the file `B:\REC_10.WAV` on the internal drive.

To reduce the latency, performance critical applications should not generate the filenames and open the files in the callbacks, but e.g. do this from a separate function.

5.3 Creating modules

The module extensions are shared libraries, that are built normally and then run through `rtcuexttag.exe`, which adds a number of tags to the module, marking it as a valid module extension.

The adjustable tags, such as version and the description are defined in the makefile.

To be able to load the module, it must export the functions moduleInit and moduleNotify.

5.3.1 moduleInit

This function is called by the system when `extModuleLoad` is called.

Any initialization the module might require must be performed here, including installing functions and function blocks.

See `vplInstallFunctions` for information on how to install modules.

Returns:

- 0 Success
- 1 Failed to initialize module.

Declaration

```
int MODDECL moduleInit(void)
```

5.3.2 moduleNotify

The function `moduleNotify` is called whenever an event happens that the module should handle. This could e.g. be power events such as reset, where the module may need to release some resources.

Parameters:

	Name		Description
In	event	-	The event ID
In	parm	-	The event parameter

Returns:

None.

Declaration

```
void MODDECL moduleNotify(int event, int parm)
```

5.4 Creating Functions

This section describes how to create a new function, using the `Example` module as the basis.

5.4.1 VPL function

To create a new custom function, start by creating the VPL function that will call the C function, and have it call the `MODCALL` function, with the expected name of the module and function. Make sure the function uses the `ALIGN` keyword, to make it align the variables correctly on the stack.

```
FUNCTION ALIGN numberToHex:STRING;
VAR_INPUT
   v:DINT;
   padding:SINT := 0;
END_VAR;
VAR
   err:INT;
END_VAR;
   numberToHex := STRING(INT(MODCALL("mod_example",
"numberToHex", err)));
   IF err <> 0 THEN
      numberToHex := "Error";
   END_IF;
END_FUNCTION;
```

From modules/example/example_vpl/mod_example.vpl.

Build the VPL project to generate, among other files, the `.LST` file, which is needed to define the C function.

5.4.2 C Function

If creating a new module, create a new C file by copying the example file `modules\empty\empty.c`, otherwise the function should just be added to the existing file.

Create a new C function with the wanted name and a signature matching the signature of `vpl_function_call`.

```
static int32 MODDECL numberToHex(HANDLE* pCPU, void* pbase)
{
      ...
}
```

From modules/example/mod_example.c.

Add the name of the function(used in `MODCALL`) and the function to the function table, ftable. This will make the function be called when the VPL function calls MODCALL.

```
static vplFunctionEntry ftable[] =
{
      { "numberToHex", numberToHex },
      { NULL, NULL }
};
```

From modules/example/mod_example.c.

5.4.3 Variable handling

Open the `.LST` file from the VPL project and find the newly created VPL function:

```
;*** FUNCTION numberToHex ***
;*** var
;numberToHex          off=0000 key=791722 sz=02 ty=std ival=0
;err                  off=0002 key=791745 sz=02 ty=std ival=0
;*** var_input
;v                    off=0004 key=749631 sz=04 ty=std ival=0
;padding              off=0008 key=791735 sz=01 ty=std ival=0
;*** var_input (access)
```

From modules/example/example_vpl/mod_example.lst.

Below the line with the function name, is the variables of the functions.

Each line contains one variable, ordered by what kind of variable it is, i.e. input, input access, output or local variables.

The first variable has the name of the function and will be used from VPL for the return value from the function.

The second variable is the local variable `err`, which is used for the call to `MODCALL`. It should not be modified, as it is already used by `MODCALL`.

After the name of each variable is a number of features about the variable:

- `off` is the offset of the variable from the start. The first variable, that has the same name as the function, is the return code and has the offset of 0.
- `sz` is the size of the variable. SINTs and BOOLs are 1 byte, INTs and STRINGs are two bytes and DINTs and FLOATs are 4 bytes.

Function blocks are similar to functions, except that they do not have a return code and that they supports output variables.

Based on this information, create a struct that contains the same variables in the same order, using data types of the specified size. The struct must use the `ALIGN_ATTR` attribute.

```
typedef struct {
   /*** var */
   int16       retval;
   int16       error;
   /*** var_input */
   int32       v;
   int8        padding;

} ALIGN_ATTR tdef_numbertohex;
```

From modules/example/mod_example.c.

In the C function, create a struct pointer to your newly created struct, cast the `pbase` pointer to be a pointer to the struct and assign it to the struct pointer.

```
tdef_numbertohex *data =(tdef_numbertohex*)pbase;
```

From modules/example/mod_example.c.

For very simple functions, the pointer can instead be cast in-place to avoid having to create the struct pointer.

5.4.3.1 Variable types

Depending on the kind and type of the variables, they must be handled differently.

See the `dt_mod` example for examples for many of the different combinations.

Input variables

The number types and BOOLs can be used directly from the struct.

To get a string from an input variable, use the `vplStringGet` function, which takes the string id from VPL and returns a pointer to the string.

Output variables

Output variables using the number types and BOOLs can be set just by assigning the value to the variable.

To set the string in an output variable, use the `vplStringMake` function to store the string.

Return values

The return code is set to the return code from the function.

To return a string, create a new string id by calling the `vplStringMake` function.

The number types and BOOL can just return the value directly.

ACCESS Variables

`ACCESS` variables are a bit more complex.

`ACCESS` number and BOOL variables are pointers to the actual VPL variable, and as the variables are not necessarily aligned, the data must be copied to and from the variables using the `write16b`/`write32b` and `read16b`/`read32b` functions.

`ACCESS` STRING variables are still string IDs.

To retrieve the string from an `ACCESS` STRING variable, use the `vplStringGet` function.

To change the string in an `ACCESS` STRING variable, use the `vplStringUpdate` function.

STRUCT_BLOCKs

Struct blocks ACCESS variables can be represented as struct pointers within the variable struct. The structure of the STRUCT_BLOCK can also be read from the `.LST` file. See the `dt_mod` example for an example of how ACCESS Struct blocks can be handled.

5.4.4 Finishing the function

Once the variable handling is in place, all that remains is to implement the custom functionality.

Then it is just a matter of rebuilding the module by running `make`, installing it using the IDE and to test if it works by loading and calling it from the VPL application.

If it does not work as expected, the `vplDebug` function can be used to send debug messages to the IDE from strategic places in the code, to help with the debugging.

If the system crashes before any debug message can be printed from the function, it is likely a problem with how the variables are handled.

5.5 Troubleshooting extension modules

There can be many reasons for an extension module to not work as expected:

5.5.1 extModuleLoad returns -4:

The following are typical causes for return code -4:

5.5.1.1 The module is missing the functions moduleNotify and moduleInit.

Please make sure that these functions have been implemented.

5.5.1.2 The module fails to load.

It may be that the module is depending on some libraries that are not available.

The tool `objdump` can be used to get a list of the dependencies:

```
arm-cortexa5-linux-uclibcgnueabihf-objdump -p hello.rmx | grep NEEDED
```

The list of libraries can then be checked against the default libraries available in the SDK library folders:

- `sysroot/lib`
- `sysroot/usr/lib`
- `library/lib`

If it needs a library that is not available in one of these folders by default, it is necessary to link to the library statically.

5.5.1.3 The module has not been built correctly.

It is possible that a wrong compiler has been used for compiling the module. To check if this is the case, compare it against a known good module, e.g. one of the examples.
This can be done with multiple tools:

- `file`:
  - Call `file hello.rmx`
  - Should show "`ELF 32-bit LSB shared object, ARM, EABI5 version 1 (SYSV), dynamically linked, not stripped, with debug_info`" or something similar.
- `readelf`
  - Call `arm-cortexa5-linux-uclibcgnueabihf-readelf -h hello.rmx`
  - `Machine` must be `ARM`.
  - `Flags` must be `0x5000400, Version5 EABI, hard-float ABI`.

If what it shows does not match, it suggests that the wrong compiler has been used.

Please verify that the build environment is set correctly up.

5.5.2 extModuleLoad returns -5:

The `moduleInit` function returned a value different from 0.

Please check the source for the module to determine the reason for this.

6 VPL interface

The program and module extensions are controlled by the VPL application, using the functions below. For more information on how to use the functions from VPL, see the RTCU IDE Online Help.

6.1 MODCALL

`MODCALL` is used from VPL to call C functions in the module extensions.

For examples of how this can be used, please see the examples in the `modules` folder.

The `dt_mod` example is especially interesting as it shows how to handle many different types of variables.

Parameters:

Direction		Description
In	-	The name of the module to call
In	-	The name of the function in the module.
Out		The result code from calling the function. A variable, e.g. called `err`, must be used for this which may not be touched from within the function call, as it is automatically set from the system. The possible values for this parameter are: • 0 = Function was successfully invoked. • 1 = The symbolic function name was not found. • 2 = Invalid module and/or function name. • 3 = Not supported.

Returns:
The return value from the C function, if the out parameter is set to 0.
Depending on the wanted return type, it is often necessary to cast it before it can be returned from the VPL function.
STRINGs require an extra cast to first convert the return value to an INT, i.e. `STRING(INT(MODCALL(…)))`.

6.2 extModuleLoad

`extModuleLoad` is used to load module extensions. It is called with the path to the extension, and will call the `moduleInit` function in the module.
Please see the Troubleshooting extension modules section for information on common solutions when this function fails.

6.3 extProgramStart

`extProgramStart` starts the program extension, which will then run in a separate process.

6.4 extProgramSignal

`extProgramSignal` sends a signal to a running program extension, either to stop it or just to notify the program that it must do something.

6.5 extProgramStatus

`extProgramStatus` checks the status of the program, to determine if it is still running and to release the program handle.

6.6 extTagEnumerate and extTagRead

`extTagEnumerate` and `extTagRead` are used to read the tags written by `rtcuexttag.exe`, to make the application able to choose which module to load or to validate that it is the expected module.

7 Module API Functions

This section describes all the functions etc. that can be called from extension modules. To use the functions, the module must link against the libmodule.so library.

These functions can NOT be called from extension programs.

7.1 Structures

7.1.1.1 vplFunctionEntry

Function table entry.

Parameters:

	Name		Description
	fname	-	The name of the function. Used in VPL to identify the function
	func	-	Pointer to the function in module.

Declaration:

```
struct vplFunctionEntry {
      const char* fname
      vpl_function_call func
}
```

7.2 Definitions and Macros

7.2.1 Declarations

7.2.1.1 MODDECL

Define to include in the declaration of functions, in case the calling conventions change.
Declaration:

7.2.2 Data types

These defines provide platform independent data types.

7.2.2.1 int8

Signed 8 bit integer, corresponding to SINT in VPL.
Declaration:

```
signed char
```

7.2.2.2 int16

Signed 16 bit integer, corresponding to INT in VPL.

Declaration:
```
signed short
```

7.2.2.3 int32

Signed 32 bit integer, corresponding to DINT in VPL.
Declaration:
```
signed long
```

7.2.2.4 uint8

Unsigned 8 bit integer.
Declaration:
```
unsigned char
```

7.2.2.5 uint16

Unsigned 16 bit integer.
Declaration:
```
unsigned short
```

7.2.2.6 uint32

Unsigned 32 bit integer.
Declaration:
```
unsigned long
```

7.2.2.7 HANDLE

Handle to internal system data.
Declaration:
```
void
```

7.2.3 SOS Flags

Flags for the SOS objects.

These flags can be combined bitwise.

7.2.3.1 SOS_FLAG_NONE

No special flags.
Declaration:
```
0x00
```

7.2.3.2 SOS_FLAG_RO

Read-only object. The object can not be written by external systems, such as the RTCU IDE.
Declaration:

```
0x01
```

7.2.3.3 SOS_FLAG_CRYPT

The object is encrypted. Number and boolean objects can not be encrypted.
Declaration:

```
0x02
```

7.2.4 Notification events

Event IDs for notifications sent to moduleNotify

7.2.4.1 EVENT_HALT

The system is halting.
Declaration:

```
1
```

7.2.4.2 EVENT_RESET

The system is resetting.
Declaration:

```
2
```

7.2.4.3 EVENT_SHUTDOWN

The system is shutting down.
Declaration:

```
3
```

7.2.4.4 EVENT_POWERFAIL

External power has disappeared/has appeared again. Parm is 1 when the power disappears and 0 when it appears again.
Declaration:

```
4
```

7.2.4.5 EVENT_POWERSAVE

The system is entering/leaving pmWaitEvent. Parm is 1 when entering pmWaitEvent and 0 when leaving.
Declaration:

```
5
```

7.2.5 Alignment macros

Macros for working with unaligned data

7.2.5.1 ALIGN_ATTR

Attribute to set alignment.
Declaration:
```
__attribute__ ((packed, aligned(2)))
```

7.2.5.2 PACKED_ATTR

Declaration:
```
__attribute__ ((packed))
```

7.2.5.3 read16b

Macro for unaligned read of a 16 bit value.

Parameters:

Name		Description
addr	-	The address to read from

Returns:
The value read from the address

Declaration:
```
((uint16)(((((uint8*)(addr))[1]<<8) | ((uint8*)(addr))[0])))
```

7.2.5.4 read24b

Macro for unaligned read of a 24 bit value.

Parameters:

Name		Description
addr	-	The address to read from

Returns:
The value read from the address

Declaration:
```
((uint32)((((uint8*)(addr))[2]<<16) | (((uint8*)(addr))[1]<<8) | (((uint8*)
(addr))[0])))
```

7.2.5.5 read32b

Macro for unaligned read of a 32 bit value.

Parameters:

Name		Description
addr	-	The address to read from

Returns:

The value read from the address

Declaration:
```
((uint32)((((uint8*)(addr))[3]<<24) | (((uint8*)(addr))[2]<<16) | (((uint8*)
(addr))[1]<<8) | (((uint8*)(addr))[0])))
```

7.2.5.6 write16b

Macro for unaligned write of a 16 bit value.

Parameters:

Name		Description
addr	-	The address to write to
val	-	The value to write

Declaration:
```
{ ((uint8*)(addr))[0] = ((val)&0xff); ((uint8*)(addr))[1] = ((val)>>8)&0xff; }
```

7.2.5.7 write24b

Macro for unaligned write of a 24 bit value.

Parameters:

Name		Description
addr	-	The address to write to
val	-	The value to write

Declaration:
```
{ ((uint8*)(addr))[0] = ((val)&0xff); ((uint8*)(addr))[1] = ((val)>>8)&0xff;
((uint8*)(addr))[2] = ((val)>>16)&0xff; }
```

7.2.5.8 write32b

Macro for unaligned write of a 32 bit value.

Parameters:

Name		Description
addr	-	The address to write to
val	-	The value to write

Declaration:
```
{ ((uint8*)(addr))[0] = ((val)&0xff); ((uint8*)(addr))[1] = ((val)>>8)&0xff;
((uint8*)(addr))[2] = ((val)>>16)&0xff; ((uint8*)(addr))[3] =
((val)>>24)&0xff; }
```

7.2.6 Enumerations

Enumeration data types.

7.2.6.1 vpl_io_signals

IO Signals that can be controlled using vplSetIOSignal()
Note that not all signals are available on all targets.

	Name		Description
	RF_OFF	-	Control power to the optional RF module. Set to 0 to turn the power on.
	RF_CFG	-	Control the CFG pin on the RF module.
	SER0_EN	-	Enable serial port 0 if available.
	SER1_EN	-	Enable serial port 1 if available.
	SER2_EN	-	Enable serial port 2 if available.
	SER3_EN	-	Enable serial port 3 if available.
	SER4_EN	-	Enable serial port 4 if available (reserved)
	SER5_EN	-	Enable serial port 5 if available (reserved)
	SER6_EN	-	Enable serial port 6 if available (reserved)
	SER7_EN	-	Enable serial port 7 if available (reserved)
	SER8_EN	-	Enable serial port 8 if available (reserved)

7.3 Type definitions

7.3.1.1 vpl_function_call

Function footprint definition.

Parameters:

	Name		Description
In	pCPU	-	The system handle.
In	pBase	-	Pointer to structure with function parameters.

Returns:
Return value to pass on to VPL.

Declaration:

```
typedef int32(MODDECL * vpl_function_call) (HANDLE *pCPU, void *pbase)
```

7.3.1.2 capture_handle

Audio capture callback, for use with vplAudioStreamRegister().
When this callback is called, the module must fill the buffer with the data to play and store the data size in size.

Parameters:

	Name		Description
In	id	-	The ID of the stream to provide data for.
In	buffer	-	The buffer to place the data in.
In	buffer_size	-	The maximum size of data to write to the buffer.
Out	size	-	The actual amount of data written to the buffer.

Returns:

Value		Description
0	-	Success.
<0	-	Error, currently ignored.

Declaration:

```
typedef int(* capture_handle) (int16 id, char *buffer, uint32 buffer_size,
uint32 *size)
```

7.3.1.3 playback_handle

Audio playback callback, for use with vplAudioStreamRegister().
When this callback is called, the module must use the data provided in the buffer and return.

Parameters:

	Name		Description
In	id	-	The ID of the stream to provide data for.
In	buffer	-	The buffer containing the audio data.
In	size	-	The amount of data in the buffer.

Returns:

Value		Description
0	-	Success.
<0	-	Error, currently ignored.

Declaration:

```
typedef int(* playback_handle) (int16 id, char *data, uint32 size)
```

7.4 Functions

The following functions can be called from the Extension Module to interact with the system.

7.4.1 Basic functions

7.4.1.1 vplInstallFunctions

This function must be called from moduleInit to register the functions in the system, making it possible to call the functions from VPL.
It is possible to call this function multiple times, to install functions under multiple module names or to dynamically choose which functions to install.

Parameters:

	Name		Description
In	modname	-	The name of the module to install the function for. The first character must be a letter or underscore, the remaining characters may also contain digits. Maximum length is 20 characters.
In	ftable	-	Pointer to the Function table of the module. The table must be terminated with an empty entry, where both fname and func are NULL.

Returns:

Value		Description
0	-	Success.
1	-	A function with the same name already exists.
2	-	Invalid module name.
3	-	Modules are not supported.

Declaration:

```
uint16 MODDECL vplInstallFunctions(char modname[16], const vplFunctionEntry
*ftable)
```

7.4.1.2 vplDebug

Send a debug message to Device output in a connected IDE.
The text can be 480 characters long. This function can be used for debugging the module, by printing out strategic locations and variables.

Parameters:

	Name		Description
In	message	-	The text to send.

Returns:

None.

Declaration:
```
void MODDECL vplDebug(const char *message)
```

7.4.2 String functions

The following functions are used to move string variables between VPL and C.

7.4.2.1 vplStringGet

Reads a string from VPL and returns a pointer to it.
A pointer to an empty string is returned if the string handle is not valid.

Parameters:

	Name		Description
In	cpu	-	The handle to the system.
In	stringid	-	The handle to the string.

Returns:
A pointer to the string.

Declaration:
```
const char* MODDECL vplStringGet(HANDLE *cpu, uint16 stringid)
```

7.4.2.2 vplStringMake

Create a new VPL string from the provided C string.

Parameters:

	Name		Description
In	cpu	-	The handle to the system.
In	text	-	The text to store in the new VPL string.

Returns:
Handle to the new string.

Declaration:
```
uint16 MODDECL vplStringMake(HANDLE *cpu, const char *text)
```

7.4.2.3 vplStringUpdate

Used to update existing VPL strings by releasing the old string and returning a handle to the new string that should be used instead.
Mainly for use with ACCESS variables.

Parameters:

	Name		Description
In	cpu	-	The handle to the system.
In	strid	-	The handle to the old VPL string. It is invalid after this call.
In	text	-	The text to store in the new VPL string.

Returns:
Handle to the new VPL string.

Declaration:
```
uint16 MODDECL vplStringUpdate(HANDLE *cpu, uint16 strid, const char *text)
```

7.4.3 Time functions

The following functions are used to work with the time.

7.4.3.1 vplClockGet

Retrieves the current system time.

Returns:
The current time as linsec.

Declaration:
```
int32 MODDECL vplClockGet(void)
```

7.4.3.2 vplClockTimeToLinsec

Converts a time provided as the individual elements to a linsec.

Parameters:

	Name		Description
In	year	-	The year. (1980..2048)
In	month	-	The month. (1..12)
In	day	-	The day. (1..31)
In	hour	-	The hour. (0..23)
In	minute	-	The minute. (0..59)
In	second	-	The second. (0..59)

Returns:
The time as linsec.

Declaration:
```
int32 MODDECL vplClockTimeToLinsec(int16 year, int8 month, int8 day, int8 hour,
int8 minute, int8 second)
```

7.4.3.3 vplClockLinsecToTime

Convert linsec to the individual time elements.

Parameters:

	Name		Description
In	linsec	-	The linsec time
Out	year	-	The year.
Out	month	-	The month.
Out	day	-	The day.
Out	hour	-	The hour.
Out	minute	-	The minute.
Out	second	-	The second.

Returns:
None.

Declaration:
```
void MODDECL vplClockLinsecToTime(int32 linsec, int16 *year, int8 *month, int8
*day, int8 *hour, int8 *minute, int8 *second)
```

7.4.3.4 vplClockTMToLinsec

Convert a struct tm to linsec.

Parameters:

	Name		Description
In	time	-	Pointer to a struct tm with the time.

Returns:
The time as linsec.

Declaration:
```
int32 MODDECL vplClockTMToLinsec(struct tm *time)
```

7.4.3.5 vplClockLinsecToTM

Convert linsec to a struct tm.

Parameters:

	Name		Description
In	linsec	-	The linsec time
Out	time	-	Pointer to a struct tm with the time.

Returns:
None.

Declaration:
```
void MODDECL vplClockLinsecToTM(int32 linsec, struct tm *time)
```

7.4.4 SOS functions

The following functions are used to work with the System Object Storage, which can be used for storing settings for the module.

The functions provide the same functionality as the VPL functions, but with the added support for using encrypted objects and objects that can not be changed from the IDE.

7.4.4.1 vplSosDataCreate

Create a data object in the SOS table.

Parameters:

	Name		Description
In	name	-	The name of the object.
In	data	-	The data of the object.
In	len	-	The number of bytes in the data.
In	flags	-	The flags for the object.
In	desc	-	A short description of the object. (max 80 characters + null terminator)
In	limit	-	The maximum number of bytes allowed in the data.

Returns:

Value		Description
0	-	Success.
-1	-	Error.
-2	-	Illegal parameter.

Declaration:
```
int MODDECL vplSosDataCreate(const char *name, char *data, int len, int flags,
char *desc, int limit)
```

7.4.4.2 vplSosDataUpdate

Update a data object in the SOS table.

Parameters:

	Name		Description
In	name	-	The name of the object.
In	data	-	The data of the object.
In	len	-	The number of bytes in the data.

Returns:

Value		Description
0	-	Success.
-1	-	Error.
-2	-	Illegal parameter.
-10	-	Object is not found.
-11	-	Object is not a data value.

Declaration:

```
int MODDECL vplSosDataUpdate(const char *name, char *data, int len)
```

7.4.4.3 vplSosDataGet

Read a data object in the SOS table.

Parameters:

	Name		Description
In	name	-	The name of the object.
In	max_len	-	The maximum number of bytes that can be read.
Out	data	-	The data of the object.
Out	len	-	The number of bytes in the data.
Out	flags	-	The flags for the object.
Out	desc	-	A short description of the object. (max 80 characters + null terminator)
Out	limit	-	The maximum number of bytes allowed in the data.

Returns:

Value		Description
0	-	Success.
-1	-	Error.

-2	-	Illegal parameter.
-10	-	Object is not found.
-11	-	Object is not a data value.
-12	-	Data buffer is too small.

Declaration:

```
int MODDECL vplSosDataGet(const char *name, char *data, int max_len, int *len,
int *flags, char *desc, int *limit)
```

7.4.4.4 vplSosStringCreate

Create a string object in the SOS table.

Parameters:

	Name		Description
In	name	-	The name of the object.
In	data	-	The string data of the object.
In	flags	-	The flags for the object.
In	desc	-	A short description of the object. (max 80 characters + null terminator)
In	limit	-	The maximum string length allowed.

Returns:

Value		Description
0	-	Success.
-1	-	Error.
-2	-	Illegal parameter.

Declaration:

```
int MODDECL vplSosStringCreate(const char *name, const char *data, int flags,
char *desc, int limit)
```

7.4.4.5 vplSosStringUpdate

Update a string object in the SOS table.

Parameters:

	Name		Description
In	name	-	The name of the object.
In	data	-	The string data of the object.

Returns:

	Value		Description
	0	-	Success.
	-1	-	Error.
	-2	-	Illegal parameter.
	-10	-	Object is not found.
	-11	-	Object is not a string value.

Declaration:

```
int MODDECL vplSosStringUpdate(const char *name, const char *data)
```

7.4.4.6 vplSosStringGet

Read a string object in the SOS table.

Parameters:

	Name		Description
In	name	-	The name of the object.
In	max_len	-	The maximum number of bytes that can be read.
Out	data	-	The data of the object.
Out	flags	-	The flags for the object.
Out	desc	-	A short description of the object. (max 80 characters + null terminator)
Out	limit	-	The maximum number of bytes allowed in the string.

Returns:

	Value		Description
	0	-	Success.
	-1	-	Error.
	-2	-	Illegal parameter.
	-10	-	Object is not found.
	-11	-	Object is not a string value.
	-12	-	Data buffer is too small.

Declaration:

```
int MODDECL vplSosStringGet(const char *name, char *data, int max_len, int
*flags, char *desc, int *limit)
```

7.4.4.7 vplSosFloatCreate

Create a float object in the SOS table.

Parameters:

	Name		Description
In	name	-	The name of the object.
In	data	-	The float data of the object.
In	flags	-	The flags for the object.
In	desc	-	A short description of the object. (max 80 characters + null terminator)
In	min_data	-	The manimum value allowed.
In	max_data	-	The maximum value allowed.

Returns:

Value		Description
0	-	Success.
-1	-	Error.
-2	-	Illegal parameter.

Declaration:

```
int MODDECL vplSosFloatCreate(const char *name, float data, int flags, char
*desc, float min_data, float max_data)
```

7.4.4.8 vplSosFloatUpdate

Update a float object in the SOS table.

Parameters:

	Name		Description
In	name	-	The name of the object.
In	data	-	The value to store.

Returns:

Value		Description
0	-	Success.
-1	-	Error.
-2	-	Illegal parameter.
-10	-	Object is not found.
-11	-	Object is not a float value.

Declaration:

```
int MODDECL vplSosFloatUpdate(const char *name, float data)
```

7.4.4.9 vplSosFloatGet

Read a float object in the SOS table.

Parameters:

	Name		Description
In	name	-	The name of the object.
Out	data	-	The value of the object.
Out	flags	-	The flags for the object.
Out	desc	-	A short description of the object. (max 80 characters + null terminator)
Out	min_data	-	The manimum value allowed.
Out	max_data	-	The maximum value allowed.

Returns:

Value		Description
0	-	Success.
-1	-	Error.
-2	-	Illegal parameter.
-10	-	Object is not found.
-11	-	Object is not a float value.
-12	-	Data buffer is too small.

Declaration:

```
int MODDECL vplSosFloatGet(const char *name, float *data, int *flags, char
*desc, float *min_data, float *max_data)
```

7.4.4.10 vplSosIntCreate

Create an integer object in the SOS table.

Parameters:

	Name		Description
In	name	-	The name of the object.
In	data	-	The value to store.
In	flags	-	The flags for the object.
In	desc	-	A short description of the object. (max 80 characters + null terminator)
In	min_data	-	The manimum value allowed.
In	max_data	-	The maximum value allowed.

Returns:

Value		Description
0	-	Success.
-1	-	Error.
-2	-	Illegal parameter.

Declaration:

```
int MODDECL vplSosIntCreate(const char *name, int data, int flags, char *desc,
int min_data, int max_data)
```

7.4.4.11 vplSosIntUpdate

Update an integer object in the SOS table.

Parameters:

	Name		Description
In	name	-	The name of the object.
In	data	-	The value to store.

Returns:

Value		Description
0	-	Success.
-1	-	Error.
-2	-	Illegal parameter.
-10	-	Object is not found.
-11	-	Object is not a integer value.

Declaration:

```
int MODDECL vplSosIntUpdate(const char *name, int data)
```

7.4.4.12 vplSosIntGet

Read an integer object in the SOS table.

Parameters:

	Name		Description
In	name	-	The name of the object.
Out	data	-	The value of the object.
Out	flags	-	The flags for the object.
Out	desc	-	A short description of the object. (max 80 characters + null terminator)

Out	min_data	-	The manimum value allowed.
Out	max_data	-	The maximum value allowed.

Returns:

Value		Description
0	-	Success.
-1	-	Error.
-2	-	Illegal parameter.
-10	-	Object is not found.
-11	-	Object is not a integer value.

Declaration:

```
int MODDECL vplSosIntGet(const char *name, int *data, int *flags, char *desc,
int *min_data, int *max_data)
```

7.4.4.13 vplSosBoolCreate

Create a boolean object in the SOS table.

Parameters:

	Name		Description
In	name	-	The name of the object.
In	data	-	The value to store.
In	flags	-	The flags for the object.
In	desc	-	A short description of the object. (max 80 characters + null terminator)

Returns:

Value		Description
0	-	Success.
-1	-	Error.
-2	-	Illegal parameter.

Declaration:

```
int MODDECL vplSosBoolCreate(const char *name, int data, int flags, char *desc)
```

7.4.4.14 vplSosBoolUpdate

Update a boolean object in the SOS table.

Parameters:

	Name	Description

In	name	-	The name of the object.
In	data	-	The value to store.

Returns:

Value		Description
0	-	Success.
-1	-	Error.
-2	-	Illegal parameter.
-10	-	Object is not found.
-11	-	Object is not a boolean value.

Declaration:

```
int MODDECL vplSosBoolUpdate(const char *name, int data)
```

7.4.4.15 vplSosBoolGet

Read a boolean object in the SOS table.

Parameters:

	Name		Description
In	name	-	The name of the object.
Out	data	-	The value of the object.
Out	flags	-	The flags for the object.
Out	desc	-	A short description of the object. (max 80 characters + null terminator)

Returns:

Value		Description
0	-	Success.
-1	-	Error.
-2	-	Illegal parameter.
-10	-	Object is not found.
-11	-	Object is not a boolean value.

Declaration:

```
int MODDECL vplSosBoolGet(const char *name, int *data, int *flags, char *desc)
```

7.4.4.16 vplSosDelete

Delete an object in the SOS table.

Parameters:

	Name		Description
In	name	-	The name of the object.

Returns:

Value		Description
0	-	Success.
-1	-	Error.

Declaration:
```
int MODDECL vplSosDelete(const char *name)
```

7.4.5 Audio functions

The following functions are used to work with audio.

The audio format is in stereo, 16bit signed, little endian, at 16kHz.
It is stored interleaved, with first the left channel followed by the right channel, i.e. the byte stream looks like this: "LLRRLLRRLLRR"

7.4.5.1 vplAudioStreamRegister

Register an audio stream handler.
Once a stream handler has been registered, it can be accessed from VPL using the audioRoute function.

Parameters:

	Name		Description
In	id	-	The ID of the stream to modify.
In	capt	-	The capture callback. It is called to read the next audio data.
In	play	-	The playback callback. It is called with the latest audio data, which must be handled.

Returns:

Value		Description
0	-	Success.
-1	-	Invalid ID.
-2	-	The stream is in use.

Declaration:
```
int MODDECL vplAudioStreamRegister(int16 id, capture_handle capt,
playback_handle play)
```

7.4.6 File functions

The following functions are used to work with files.

7.4.6.1 vplFsMapFile

Map a VPL or device file name to the real file name.
Absolute and relative paths are supported. Device paths start with "dev:".
Currently supported devices:
Serial ports: "dev:ser<n>" where n = 0..3

Parameters:

	Name		Description
In	dest	-	The buffer to store the file name in.
In	dest_size	-	The size of the buffer.
In	file_format	-	The name of the file to convert. Supports using sprintf formatting such as "%d".
In	...	-	Formatting arguments for file_format.

Returns:

Value		Description
0	-	Success.
-1	-	Invalid format.
-2	-	Dest too small.
-3	-	Unknown device
-4	-	Invalid filename
-5	-	Invalid path
-6	-	Drive not available
-7	-	Busy
-10	-	Out of memory

Declaration:

```
int MODDECL vplFsMapFile(char *dest, size_t dest_size, const char
*file_format,...)
```

7.4.7 IO functions

The following functions are used to work with input and output signals.

7.4.7.1 vplSetIOSignal

Use this function to set the value of raw IO signals.
See the vpl_io_signals enumeration for a list of the available signals.

Note that not all signals are available on all targets.
Some signals, such as the serial port enable signals, may return success even when they are not present on the specific target.
This is for consistency with other targets that have such a signal.

Parameters:

	Name		Description
In	io	-	The ID of the signal to set
In	val	-	The value to set the pin to. Set it to 1 to assert the signal and to 0 to deassert it.

Returns:

Value		Description
0	-	Success.
-1	-	Invalid signal.

Declaration:

```
int MODDECL vplSetIOSignal(vpl_io_signals io, int val)
```