RTA-OSEK for PC

User Guide

Contact Details

ETAS Group

www.etasgroup.com

Germany

ETAS GmbH
Borsigstraße 14
70469 Stuttgart

Tel.:+49 (711) 8 96 61-102
Fax:+49 (711) 8 96 61-106

www.etas.de

USA

ETAS Inc.
3021 Miller Road
Ann Arbor, MI 48103

Tel.: +1 (888) ETAS INC
Fax: +1 (734) 997-94 49

www.etasinc.com

Japan

ETAS K.K.
Queen's Tower C-17F,
2-3-5, Minatomirai, Nishi-ku,
Yokohama, Kanagawa
220-6217 Japan

Tel.: +81 (45) 222-0900
Fax: +81 (45) 222-0956

www.etas.co.jp

France

ETAS S.A.S.
1, place des États-Unis
SILIC 307
94588 Rungis Cedex

Tel.: +33 (1) 56 70 00 50
Fax: +33 (1) 56 70 00 51

www.etas.fr

Korea

ETAS Korea Co., Ltd.
4F, 705 Bldg. 70-5
Yangjae-dong, Seocho-gu
Seoul 137-899, Korea

Tel.: +82 (2) 57 47-016
Fax: +82 (2) 57 47-120

www.etas.co.kr

Great Britain

ETAS UK Ltd.
Studio 3, Waterside Court
Third Avenue, Centrum 100
Burton-upon-Trent
Staffordshire DE14 2WQ

Tel.: +44 (0) 1283 - 54 65 12
Fax: +44 (0) 1283 - 54 87 67

www.etas-uk.net

Copyright Notice

Disclaimer

Trademarks

Table of Contents

1 About this Guide

RTA-OSEK for PC consists of tools and libraries that enable the creation of Windows-hosted applications that emulate the behavior of applications running on microcontroller hardware – including the ability to run automotive-style OSEK™ and AUTOSAR applications. This guide describes the tools and libraries used to create and monitor such applications.

RTA-OSEK for PC includes a Windows port of the popular RTA-OSEK kernel. You should consult the documents *RTA-OSEK User Guide* and *RTA-OSEK Reference Guide* for general information on how to use RTA-OSEK.

The document *RTA-OSEK Binding Manual PC* provides specific notes about using the RTA-OSEK kernel component in a Windows environment.

1.1 Who Should Read this Guide?

It is assumed that you are a developer who wants to know how to create and monitor OSEK or AUTOSAR applications on Windows PCs. You should be familiar with programming in C, and should have some knowledge of C++.

1.2 Conventions

Important: Notes that appear like this contain important information that you need to be aware of. Make sure that you read them carefully and that you follow any instructions that you are given.

In this guide you'll see that program code, header file names, C/C++ type names, C/C++ functions and API call names all appear in the `Courier` typeface.

2 Overview

Congratulations on selecting *RTA-OSEK for PC*.

RTA-OSEK for PC is a complete environment for developing OSEK[1] and AUTOSAR[2] applications. Mostly you'll be using it to prototype a new application before migrating it on to the production hardware, but you will also find that it is a good tool for learning how to develop applications for embedded targets.

But you needn't stop there. Because *RTA-OSEK for PC* is a complete and fast implementation of OSEK, you also add inter-application communication using a standard networking solution such as CAN[3]. You can write applications that sit on your CAN network as test or simulation units. You can remotely monitor the state and progress of your applications using the *RTA-OSEK for PC* monitor, *RTA-TRACE* or your PC-based debugger. And of course the development turnaround time is tiny – just recompile and run. No downloading of hex files to an emulator. No programming Flash.

2.1 Terms

RTA-OSEK for PC is typically used in automotive environments where the term *ECU* (Electronic Control Unit) is commonly used to refer to the target hardware on which the application runs. The ECU can be considered as a black box[4] with inputs and outputs that perform a specific set of functions.

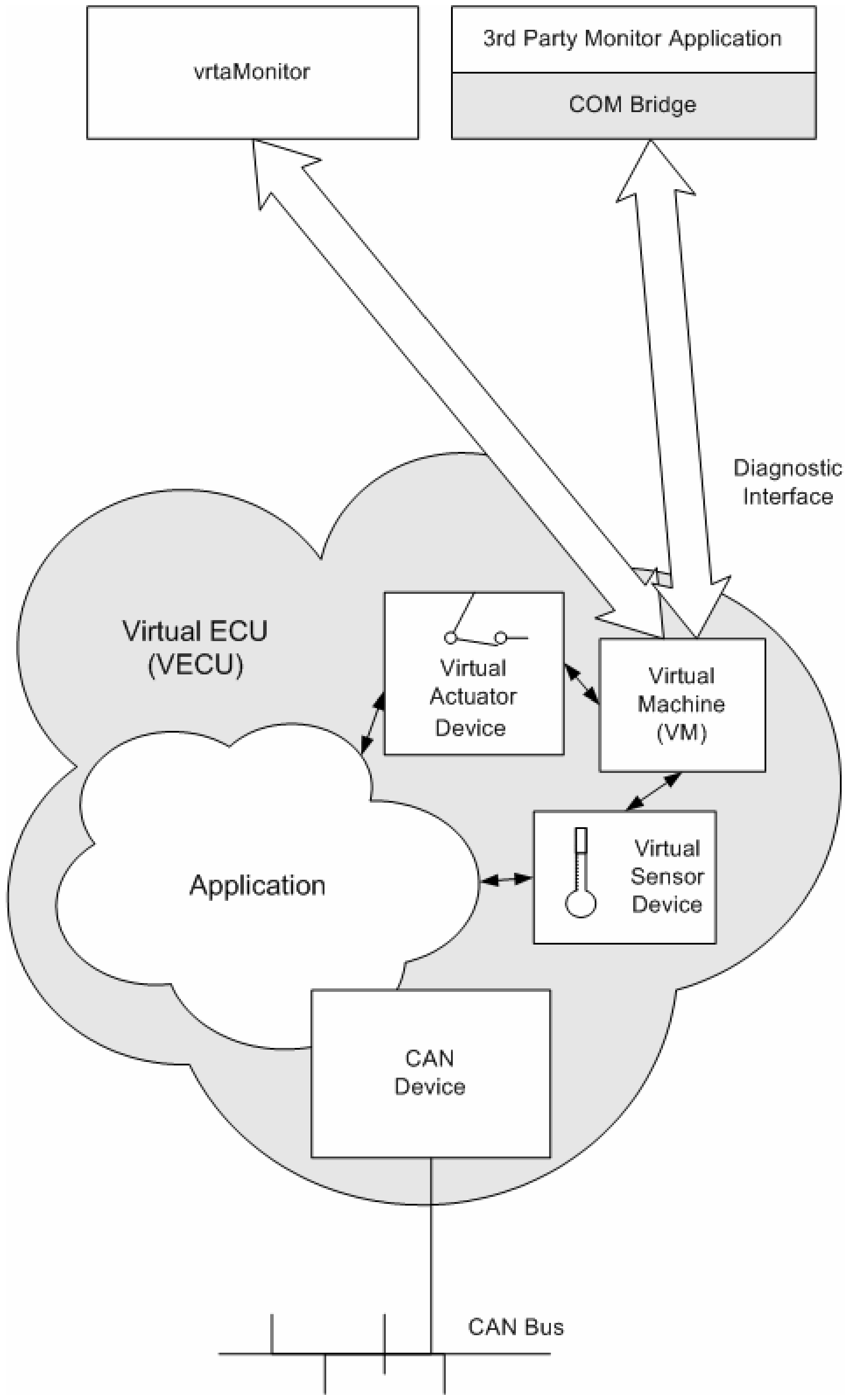
In a typical modern car, ECUs will be found in the engine compartment, the doors, the body and the boot. Many if not all will be running OSEK or AUTOSAR applications.

Other than the PC that you run it on, an application built under *RTA-OSEK for PC* does not need any real hardware. Instead, you create a *Virtual ECU (VECU)* in software that simulates the real-life devices such as switches or sensors that will be present in your ECU. These devices are built around a core *Virtual Machine (VM)* that provides services such as the interrupt controller, application control and diagnostic links.

Within this document we will use the terms VM and VECU extensively. Remember that *VM* represents the 'core' of the simulated hardware, and that *VECU* is the whole 'black-box'.

The VM provides a *diagnostic interface* (via TCP/IP) that allows external programs to interact with a VECU. `vrtaMonitor` is a program provided with *RTA-OSEK for PC* that can monitor and manage VECUs. The *COM Bridge* is a DLL provided with *RTA-OSEK for PC* that allows COM clients to interact with VECUs.

[1] See http://www.osek-vdx.org/
[2] See http://www.autosar.org
[3] See ISO standards 11898-1 and 11898-2
[4] Also available in silver



The string "vrta" is used to prefix executables such as `vrtaMonitor.exe`, `vrtaVM.dll` and many of the supplied source files. "vrta" is an abbreviation for (Virtual) *RTA-OSEK for PC*.

2.2 What do I need?

RTA-OSEK for PC runs under Microsoft Windows 2000 or later (including Windows XP). It requires a Pentium class processor. The actual performance of a VECU will clearly be dependent on the power of the processor, but you will find that a modern PC is capable of running a typical OSEK application many times faster that a typical embedded target.

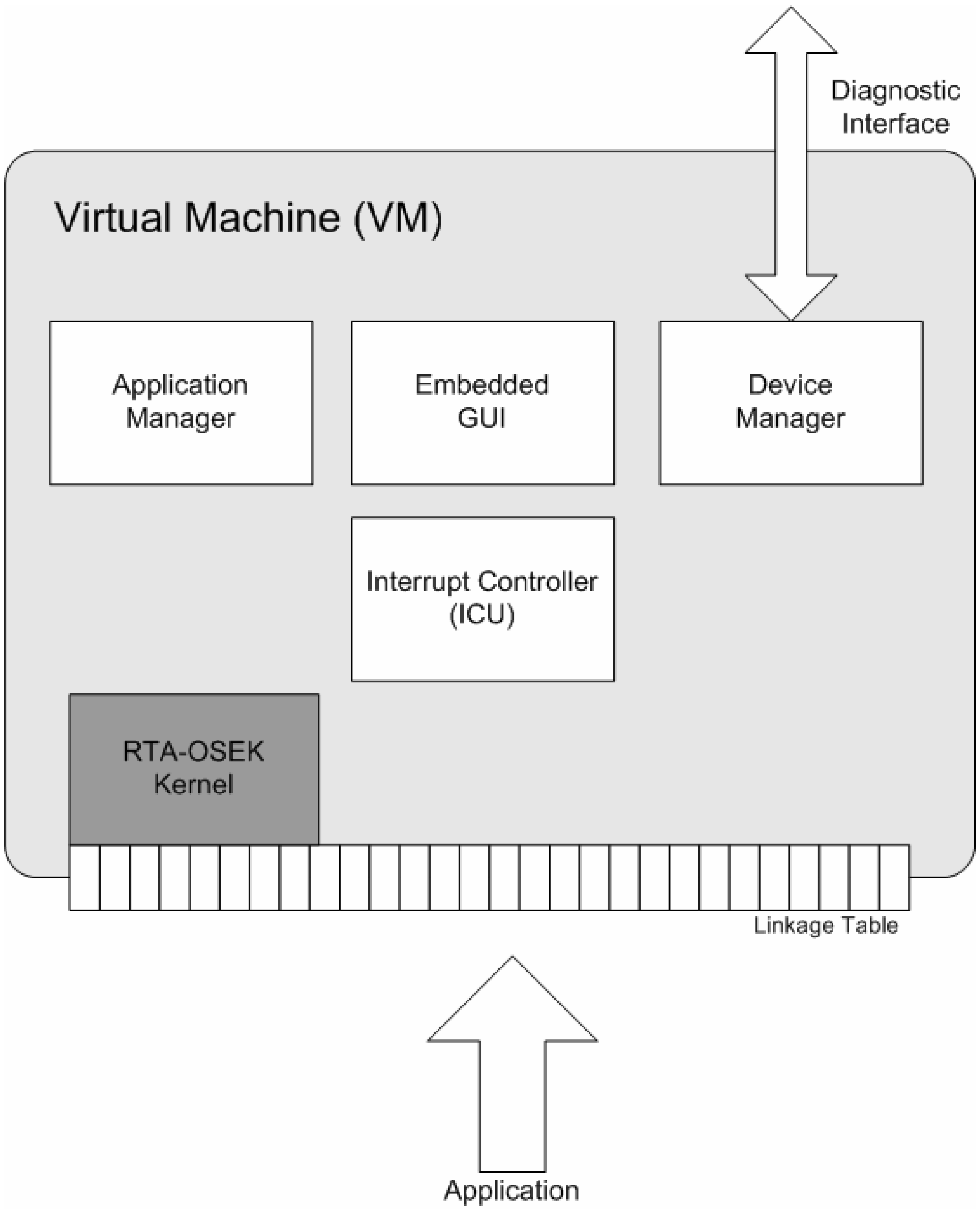
You can use 'any' Windows C++ Compiler to generate Virtual ECUs. *RTA-OSEK for PC* has been tested with the following compiler variants:

- MinGW / GCC. We test against version 3.4.2.
- Microsoft Visual C++ 5.0
- Microsoft Visual Studio 2003
- Borland C++ 5.5.1 / Borland C++ Builder 5
- Borland C++ 5.8.1 / Borland Developer Studio 2006

It is a straightforward process to support different compilers or compiler versions; refer to chapter 14 for details.

You can use the debugger that comes with your compiler to debug your VECU code.

2.3 What is the Virtual Machine?

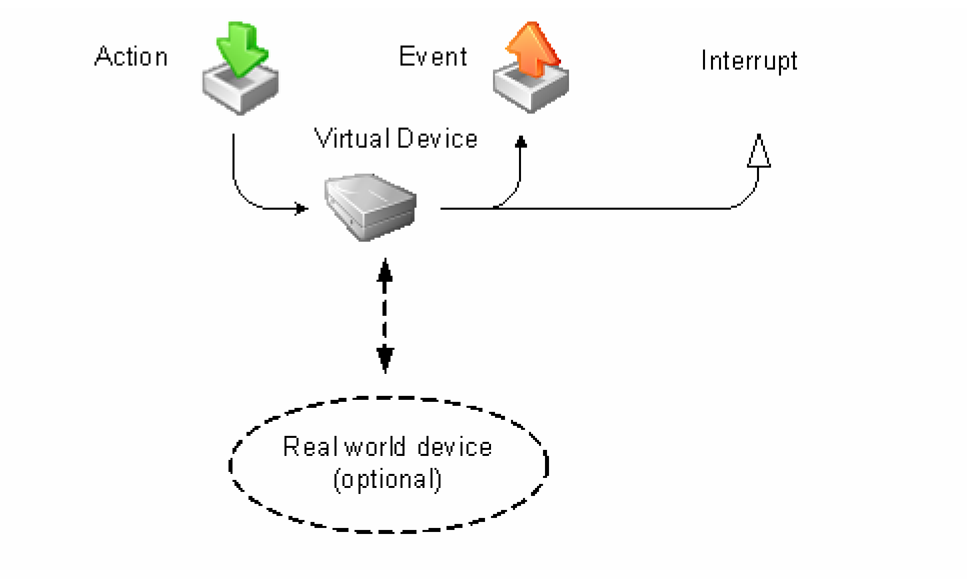


The Virtual Machine is at the core of a Virtual ECU. It manages (virtual) interrupts, startup and shutdown of the application, and routing of messages between devices and the outside world. The VM may also contain the RTA-OSEK kernel code.

The VM can be split into a number of components as explained in the following sub-sections.

2.3.1 Device Manager

Virtual ECUs use *devices* to get things done. Devices have *actions* and *events*. You can tell a device to perform a particular action. A device can inform you of some change in state by raising an event or interrupt.



A device can be a simple representation of a switch or an LED, or it can represent a complex component such as a PCMCIA-based CAN controller.

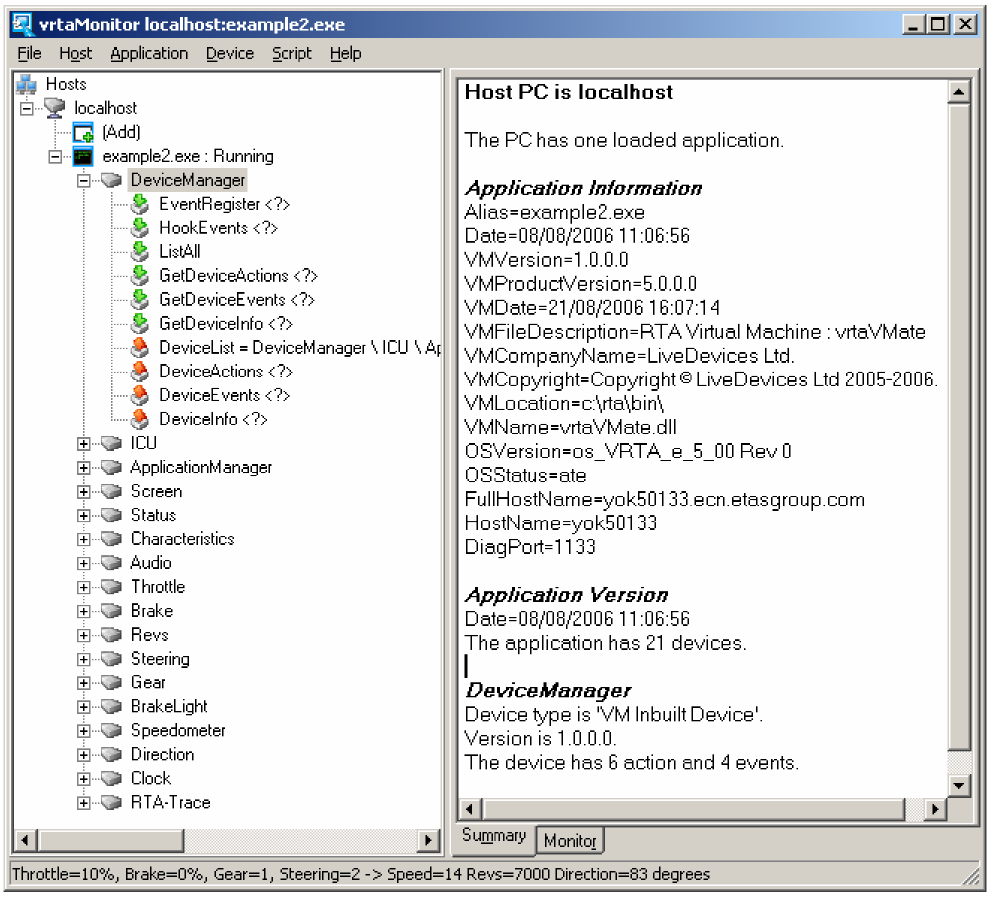
Most operations in a VECU are performed by sending actions or responding to events.

The VM itself contains the three standard devices: the DeviceManager[5], the ICU and the ApplicationManager.

The Device Manager coordinates all devices in the VECU. Each device registers with the Device Manager during initialization of the application. The Device Manager can then be queried to find what devices exist, what actions/events they support and the data used by the actions and events.

These services are not only available within the VECU: the Device Manager includes a Diagnostic Interface that allows external applications, such as `vrtaMonitor` (see chapter 6) to inspect the state of the application's devices.

[5] Yes, the Device Manager is itself also a device. You query it as device zero to find out what other devices are present.


vrtaMonitor localhost:example2.exe
File Host Application Device Script Help
Summary Monitor

Hosts
localhost
(Add)
example2.exe : Running
DeviceManager
EventRegister <?>
HookEvents <?>
ListAll
GetDeviceActions <?>
GetDeviceEvents <?>
GetDeviceInfo <?>
DeviceList = DeviceManager \ ICU \ Ap
DeviceActions <?>
DeviceEvents <?>
DeviceInfo <?>
ICU
ApplicationManager
Screen
Status
Characteristics
Audio
Throttle
Brake
Revs
Steering
Gear
BrakeLight
Speedometer
Direction
Clock
RTA-Trace

Host PC is localhost

The PC has one loaded application.

Application Information
Alias=example2.exe
Date=08/08/2006 11:06:56
VMVersion=1.0.0.0
VMProductVersion=5.0.0.0
VMDate=21/08/2006 16:07:14
VMFileDescription=RTA Virtual Machine : vrtaVMate
VMCompanyName=LiveDevices Ltd.
VMCopyright=Copyright © LiveDevices Ltd 2005-2006.
VMLocation=c:\rta\bin\
VMName=vrtaVMate.dll
OSVersion=os_VRTA_e_5_00 Rev 0
OSStatus=ate
FullHostName=yok50133.ecn.etasgroup.com
HostName=yok50133
DiagPort=1133

Application Version
Date=08/08/2006 11:06:56
The application has 21 devices.

DeviceManager
Device type is 'VM Inbuilt Device'.
Version is 1.0.0.0.
The device has 6 action and 4 events.

Throttle=10%, Brake=0%, Gear=1, Steering=2 -> Speed=14 Revs=7000 Direction=83 degrees

2.3.2 Interrupt Control Unit

The Interrupt Control Unit (ICU) is a device within the VM that simulates multi-level interrupts in the Virtual ECU.

The ICU supports 32 different interrupt vectors (1 to 32) and 33 interrupt priorities (0[6] to 32). Each interrupt vector can be assigned a priority (1 to 32).

The ICU maintains a current Interrupt Priority Level (IPL). An interrupt that has a priority <= the current IPL remains pending until the IPL drops below the assigned priority.

When an interrupt is handled, the IPL is raised to match the interrupt's assigned priority. When the interrupt handler completes, the IPL is taken back to the value that was in effect when the interrupt was taken.

Each interrupt vector can be masked. A masked vector can still become pending, but its interrupt handler will not run unless the vector is unmasked.

After reset, all interrupts are masked.

[6] Interrupt priority zero means 'no interrupt'.

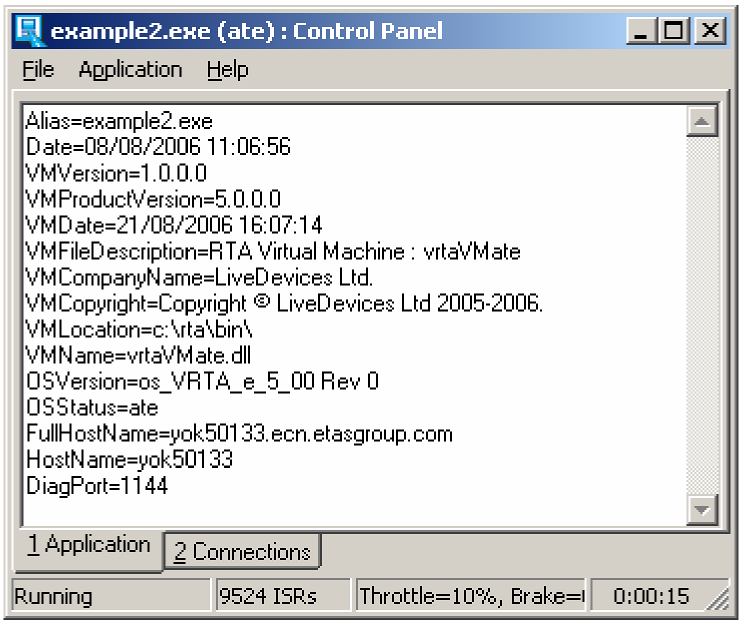
2.3.3 Application Manager

The Application Manager is a device within the VM that manages the state of the overall VECU. It is responsible for controlling the Windows thread in which your application code runs[7]. Its actions can be used to start, pause, resume, reset or terminate your application.

The Application Manager can also provide your application with access to any parameters present on the command-line when the VECU was invoked.

2.3.4 Embedded GUI

When it runs, your VECU probably doesn't show much other than a boring black empty console window. The VM can optionally display a simple GUI window that will give you a bit of confidence that the application is actually alive.



```
Alias=example2.exe
Date=08/08/2006 11:06:56
VMVersion=1.0.0.0
VMProductVersion=5.0.0.0
VMDate=21/08/2006 16:07:14
VMFileDescription=RTA Virtual Machine : vrtaVMate
VMCompanyName=LiveDevices Ltd.
VMCopyright=Copyright © LiveDevices Ltd 2005-2006.
VMLocation=c:\rta\bin\
VMName=vrtaVMate.dll
OSVersion=os_VRTA_e_5_00 Rev 0
OSStatus=ate
FullHostName=yok50133.ecn.etasgroup.com
HostName=yok50133
DiagPort=1144
```

```
Running | 9524 ISRs | Throttle=10%, Brake=I | 0:00:15
```

You can perform some simple operations such as pause/resume/reset from the menu of the GUI. For more complex options, just select menu option Application/Monitor to launch `vrtaMonitor`.

[7] Application code runs in its own thread, not the main Windows thread.

2.3.5 RTA-OSEK Kernel

The Virtual Machine is provided in a DLL file. A VECU gets dynamically linked to the Virtual Machine during initialization.

Advantage is taken of this mechanism to support the different 'build status' variants of OSEK. *RTA-OSEK for PC* actually ships with 9 different Virtual Machine DLLs. 8 of these contain the different flavors of the RTA-OSEK kernel, and one has no RTA-OSEK kernel[8].

VM Name	**Content**
vrtaVM.dll	VM only, no OSEK component
vrtaVMs.dll	VM plus OSEK 's' build.
vrtaVMt.dll	VM plus OSEK 't' build.
vrtaVMe.dll	VM plus OSEK 'e' build.
vrtaVMts.dll	VM plus OSEK 's' build with simple RTA-TRACE support.
vrtaVMtt.dll	VM plus OSEK 't' build with simple RTA-TRACE support.
vrtaVMte.dll	VM plus OSEK 'e' build with simple RTA-TRACE support.
vrtaVMatt.dll	VM plus OSEK 't' build with advanced RTA-TRACE support.
vrtaVMate.dll	VM plus OSEK 'e' build with advanced RTA-TRACE support.

2.3.6 Linkage Table

You've now seen that the VM is packaged in a DLL and will realize that this means that your code has to somehow call into the DLL to make API calls. The simple answer is "don't worry" – the startup code provided with *RTA-OSEK for PC* will ensure that the correct DLL is loaded and that API stubs exist for all VM and OSEK calls. Your application makes RTA-OSEK and other API calls just as it would in normal embedded application code.

[8] It turns out that you can create some very effective 'PC' applications using the *RTA-OSEK for PC* framework without using OSEK at all ;-)

2.4 What is in a Virtual ECU?

A Virtual ECU is the combination of the VM DLL and your application code to create a program that has inputs and outputs that simulate a physical ECU.

Your application code can have an RTA-OSEK element, namely the tasks, ISRs and processes.

It will also have *RTA-OSEK for PC* startup and linkage code[9] that glues your application to the VM.

It will also contain virtual devices that connect to real hardware or simulate it. Virtual devices are very easy to use, as you'll see in the next chapter.

2.5 Managing Virtual ECUs

RTA-OSEK for PC allows you to run several VECUs on the same PC at the same time. This causes something of a problem for external monitor programs, because they need a way to find out what VECUs are running, and how to connect to their diagnostic links.

The solution chosen for *RTA-OSEK for PC* is to have a server program running on the PC that VECUs register with when they start up. Monitor programs then ask the server what VECUs exist, and how to connect to them.

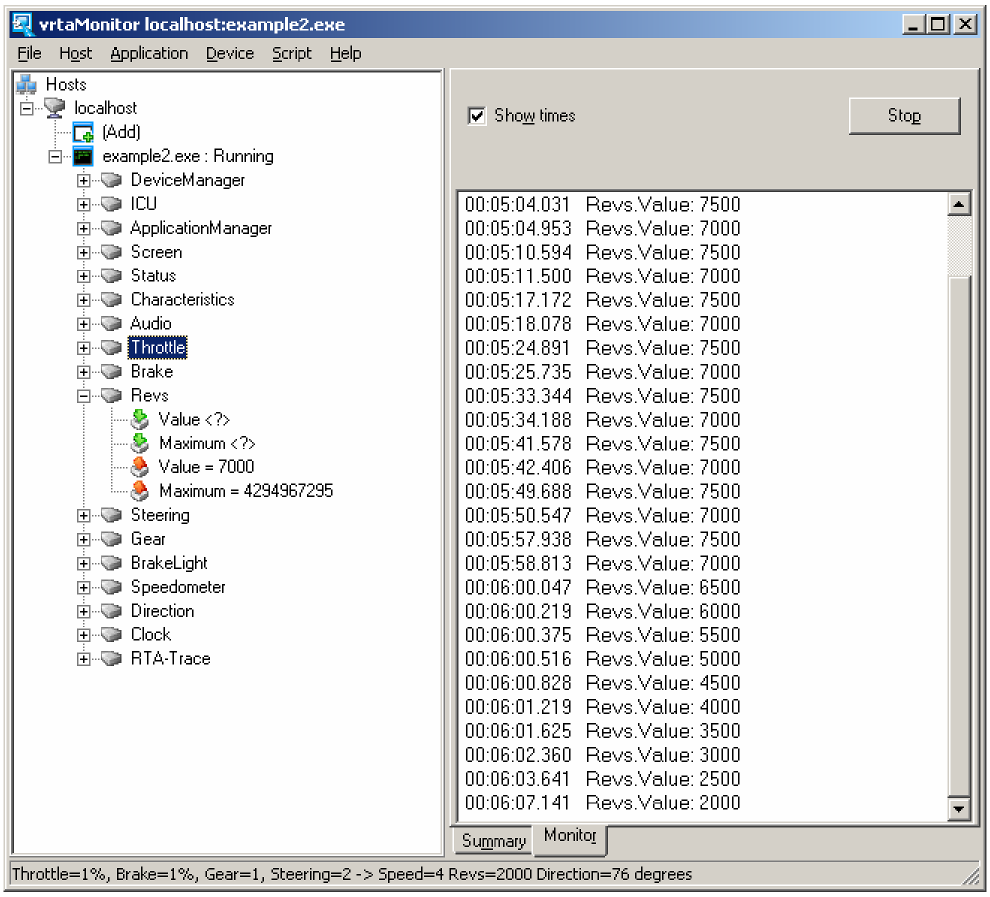
The server program is `vrtaServer.exe`, and it normally runs unnoticed as a Windows service. You do not have to start it yourself – a VECU or monitor will start the server if required.

One useful benefit of having such a server is that monitor programs can also attach to servers and VECUs that are on remote PCs. The monitor has all of the features available when used on a local machine, including the ability to reset, terminate and load VECUs.

[9] All provided with *RTA-OSEK for PC*

2.6 Interacting with a Virtual ECU

The `vrtaMonitor` program is the quickest way to interact with a Virtual ECU. All VECUs have a diagnostic link to which the monitor can connect, so no special action is needed when building your application. The monitor allows you to send actions and view events on local and remote PCs. It is discussed in detail in chapter 6.

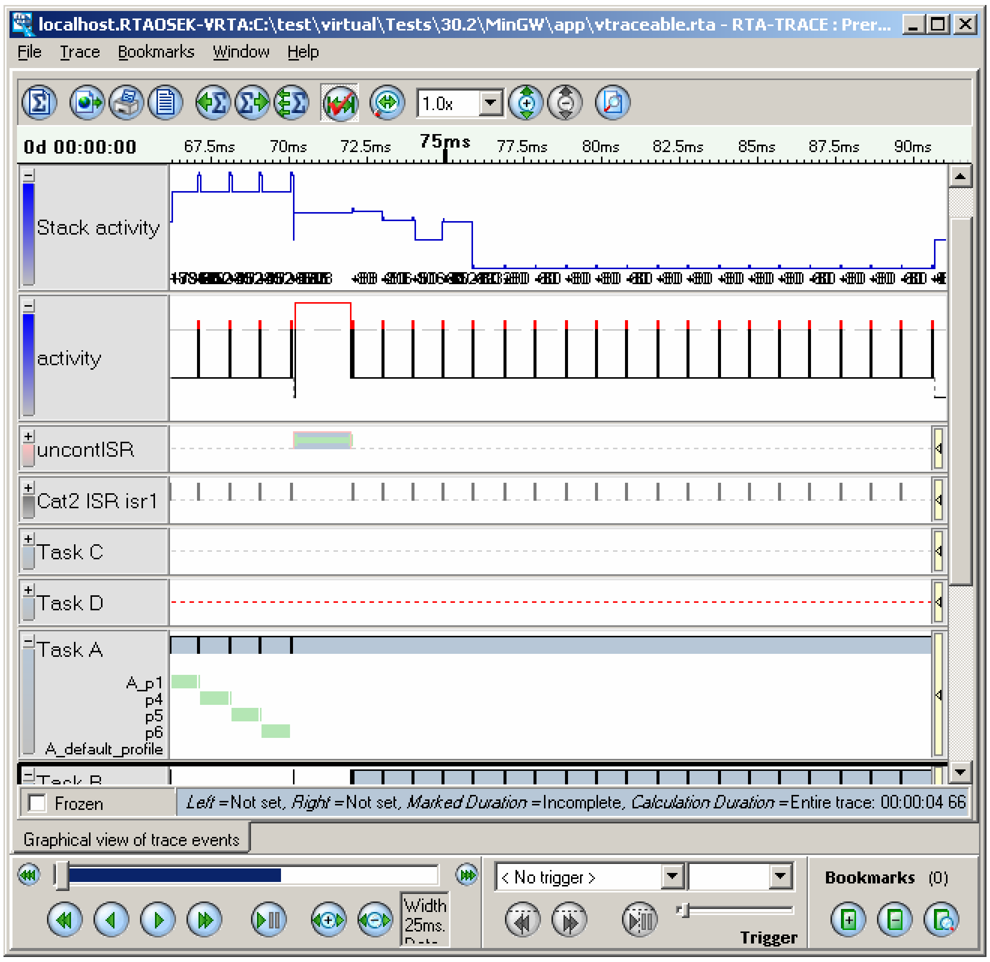

vrtaMonitor localhost:example2.exe
File Host Application Device Script Help
Show times
Stop
Summary
Monitor

```
Hosts
  localhost
    (Add)
    example2.exe : Running
      DeviceManager
      ICU
      ApplicationManager
      Screen
      Status
      Characteristics
      Audio
      Throttle
      Brake
      Revs
        Value <?>
        Maximum <?>
        Value = 7000
        Maximum = 4294967295
      Steering
      Gear
      BrakeLight
      Speedometer
      Direction
      Clock
      RTA-Trace
```

```
00:05:04.031  Revs.Value: 7500
00:05:04.953  Revs.Value: 7000
00:05:10.594  Revs.Value: 7500
00:05:11.500  Revs.Value: 7000
00:05:17.172  Revs.Value: 7500
00:05:18.078  Revs.Value: 7000
00:05:24.891  Revs.Value: 7500
00:05:25.735  Revs.Value: 7000
00:05:33.344  Revs.Value: 7500
00:05:34.188  Revs.Value: 7000
00:05:41.578  Revs.Value: 7500
00:05:42.406  Revs.Value: 7000
00:05:49.688  Revs.Value: 7500
00:05:50.547  Revs.Value: 7000
00:05:57.938  Revs.Value: 7500
00:05:58.813  Revs.Value: 7000
00:06:00.047  Revs.Value: 6500
00:06:00.219  Revs.Value: 6000
00:06:00.375  Revs.Value: 5500
00:06:00.516  Revs.Value: 5000
00:06:00.828  Revs.Value: 4500
00:06:01.219  Revs.Value: 4000
00:06:01.625  Revs.Value: 3500
00:06:02.360  Revs.Value: 3000
00:06:03.641  Revs.Value: 2500
00:06:07.141  Revs.Value: 2000
```

```
Throttle=1%, Brake=1%, Gear=1, Steering=2 -> Speed=4 Revs=2000 Direction=76 degrees
```

An RTA-OSEK application can also be traced using RTA-TRACE. A VECU has to be built with the tracing option enabled, but from then on the VECU will run as normal, only sending trace data out if RTA-TRACE is connected. RTA-TRACE can monitor a VECU from a remote PC, a fact that can be used to minimize the impact of the RTA-TRACE GUI on the execution of the VECU[10].



Left =Not set, *Right* =Not set, *Marked Duration* =Incomplete, *Calculation Duration* =Entire trace: 00:00:04 66

You can also use the PC debugger that comes with your compiler toolchain to debug a VECU at a line-by-line level. Simply ensure that your compile and link options are set correctly, and then load the VECU into the debugger.

[10] If RTA-TRACE runs on a remote PC, then the processor cycles that are used to process its data and draw its graphs do not have to be stolen from the VECU.

2.7 Possible Problem Areas

The following is a short list of problems that you might encounter when you start developing VECU code:

- You must not make any non-VM or non-OSEK API calls from your application thread if virtual interrupts could occur. This includes 'quick hacks' such as using `printf()` to display the content of some piece of data. All non-VM and non-OSEK API calls must be protected by an uninterruptible section (see section 9.4.3). See section 16.2 for more details.

- Your application seems to lock up. Provide `StackFaultHook()` and `ShutdownHook()` handlers and print an error to the screen if they occur. (You can use `printf()` here.)

3 Tutorial

The best way to get an understanding of *RTA-OSEK for PC* is to get something running, so let's set aside a couple of hours and see what we can come up with.

3.1 Prerequisites

For this tutorial we will need a copy of *RTA-OSEK for PC* installed and licensed on your computer. We will make use of *RTA-TRACE* if you have that. We also need a C++ compiler.

3.1.1 RTA-OSEK

If you have followed the instructions in the *Getting Started Guide* then you have already installed the RTA-OSEK v5 tools CD and the *RTA-OSEK for PC* target CD. You will have obtained and installed your license. In this tutorial we will assume that you have installed RTA-OSEK to `c:\rta`,so the route for the RTA executables and DLLs is `c:\rta\bin`. The files that are specific to *RTA-OSEK for PC* are found at `c:\rta\vrta`.

3.1.2 Compiler

This tutorial will use the MinGW compiler that is freely available under the GNU license. See the "Installation" section of "RTA-OSEK for PC Getting Started Guide".

In this tutorial we will assume that you have installed the compiler at `c:\MinGW`.

You now need to tell RTA-OSEK about the compiler. We need to edit the file `c:\rta\vrta\toolinit.bat`. **This is the only configuration file that you will normally need to edit**.

`toolinit.bat` is a batch file that gets run during the build stage for a VECU. It simply sets certain environment variables that tell RTA-OSEK where to find the compiler elements, and sets up some default values. If you look in `toolinit.bat` you will see that it is already set up to recognize a range of compilers. The value of the environment variable `VRTA` is used to determine which compiler to select:

e.g.

```
@echo off
< ...snip... >
if not @%1==@ set VRTA=%1
if @%VRTA%@==@MinGW@ goto MINGW
if @%VRTA%@==@BorlandC@ goto BCPP
if @%VRTA%@==@BDS2006@ goto BDS_2006
if @%VRTA%@==@VisualC5@ goto VCPP5
if @%VRTA%@==@VS2003@ goto VS2003
echo !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
echo Compiler not specified in environment variable VRTA
echo Valid settings are:
echo     MinGW
echo     BorlandC
echo     BDS2006
echo     VisualC5
echo     VS2003
echo !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
goto exit
```

You should edit the MINGW section so that it looks something like this:

```
:MINGW
rem tools installation directory
set CBASE=c:\mingw

rem location of C compiler
set CC=%CBASE%\bin\gcc.exe

rem location of C++ compiler
set AS=%CBASE%\bin\gcc.exe

rem location of linker
set LNK=%CBASE%\bin\g++.exe

rem location of Archiver / librarian
set AR=%CBASE%\bin\ar.exe

rem Set location of C include files
set CBASE_INC=%CBASE%\include

rem Default settings
SET _LIBS=-lwinmm -lws2_32
SET _OBJ=o

goto check
```

Now open a console window and execute `c:\rta\vrta\toolinit MinGW`[11], followed by `%cc% --version`. Your results should be like this:

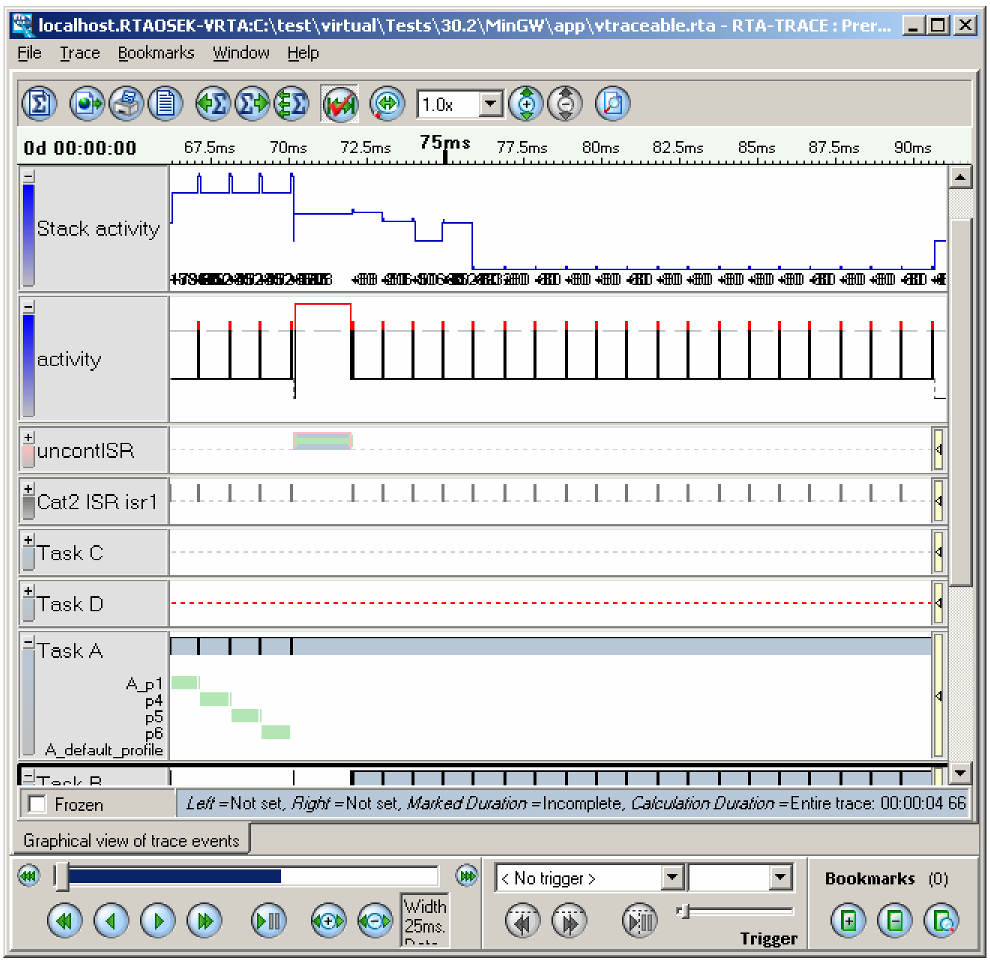
```
C:\>c:\rta\vrta\toolinit MinGW
C:\>%cc% --version
gcc.exe (GCC) 3.4.2 (mingw-special)
Copyright (C) 2004 Free Software Foundation, Inc.
This is free software; see the source for copying
conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE.
C:\>
```

[11] `toolinit.bat` will use the first command-line parameter passed to it if the environment variable `VRTA` is not set.

3.1.3 RTA-TRACE

RTA-TRACE is available as a separate product and provides a very detailed graphical display showing in real-time the execution of all Tasks, ISRs and processes in your application.

RTA-OSEK for PC comes complete with a special high-bandwidth virtual device that can be used to connect to RTA-TRACE. If you have installed RTA-TRACE in the same location as RTA-OSEK then this link will be detected automatically. If not you must copy the file `rtcVRTAlink.dll` from RTA-OSEK's 'bin' to RTA-TRACE's 'bin'.



Left =Not set, *Right* =Not set, *Marked Duration* =Incomplete, *Calculation Duration* =Entire trace: 00:00:04 66

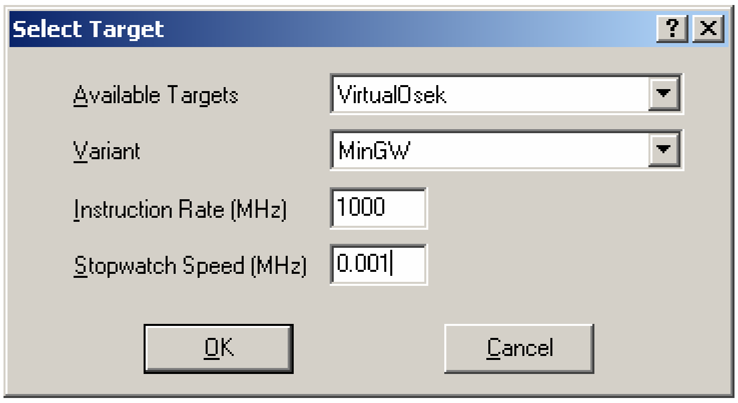
3.2 Creating your first Virtual ECU

Let's go for the traditional 'Hello World' starter application – with an OSEK twist.

3.2.1 File\New…

Start the RTA-OSEK GUI (`c:\rta\bin\RTA-OSEK.exe`).

You are now looking at a rather drab empty grey window which we can brighten up a bit by selecting menu option **File / New…** , then entering the values shown in the dialog below.

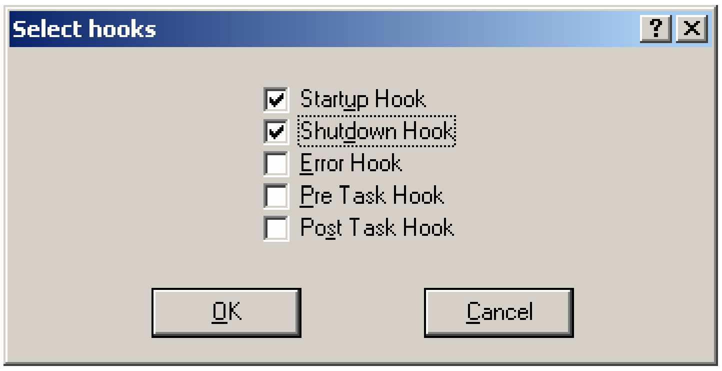


Available Targets: VirtualOsek
Variant: MinGW
Instruction Rate (MHz): 1000
Stopwatch Speed (MHz): 0.001

You now have a new empty application.

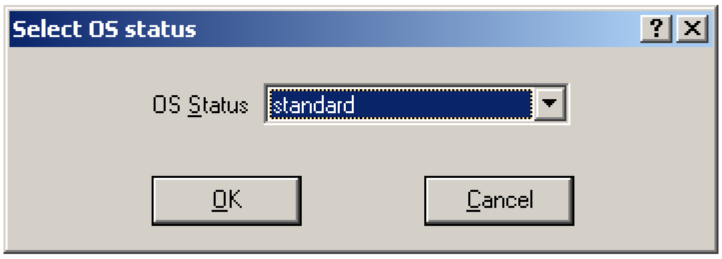
Select menu option **Application / OS configuration** (or use the navigation bar on the left hand side to get to the same place).

Select the **Hooks** Button and select **Startup** and **Shutdown** hooks.



- [✓] Startup Hook
- [✓] Shutdown Hook
- [ ] Error Hook
- [ ] Pre Task Hook
- [ ] Post Task Hook

Select the **OS Status** button to change the OS status to **standard**.



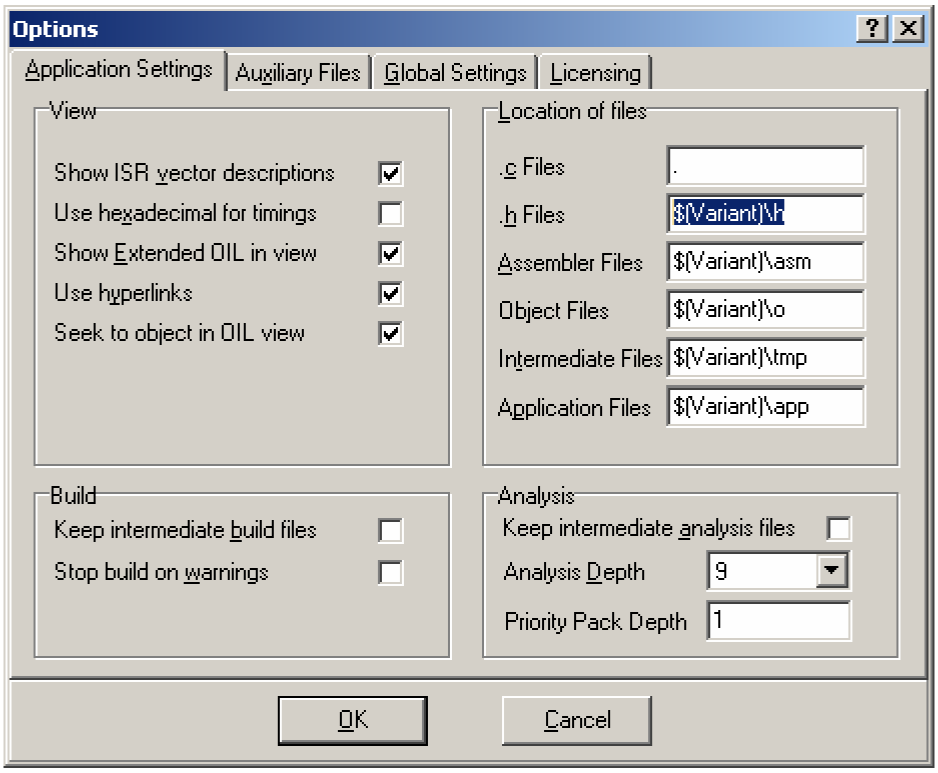
OS Status: standard

If you now look at the hints that RTA-OSEK provides about implementation details (Menu option **View / Implementation** or Ctrl+M) then you will see that we will need to supply the code for these two callback functions.

```
Hook functions
The application must implement:
  OS_HOOK(void) StartupHook(void);
  OS_HOOK(void) ShutdownHook(StatusType s);
```

We are now ready to write some code.

Because we like to keep things tidy, we first tell RTA-OSEK where to look for, and generate, files. Select menu option **File / Options…** and set the location of files as shown here[12,13].



Show ISR vector descriptions ✓
Use hexadecimal for timings ☐
Show Extended OIL in view ✓
Use hyperlinks ✓
Seek to object in OIL view ✓

.c Files: .
.h Files: $(Variant)\h
Assembler Files: $(Variant)\asm
Object Files: $(Variant)\o
Intermediate Files: $(Variant)\tmp
Application Files: $(Variant)\app

Keep intermediate build files ☐
Stop build on warnings ☐

Keep intermediate analysis files ☐
Analysis Depth: 9
Priority Pack Depth: 1

[12] If you really like these settings you can also put them on the Global Setting tab. That way your next project will pick up these values too.

[13] RTA-OSEK supports a number of macros such as "$(Variant)". "$(Variant)" expands out to be the name of the current compiler variant. The compiler variant was selected when you created the application with File / New …. See chapter 14 for more information on compiler variants. In this case, "$(Variant)" will expand out to "MinGW".

The first entry says that `.c` files will be located in the same directory as the project OIL file.

The next entry says that `.h` files will be placed 2 levels down from that in directory $(Variant)\h, which is `.\MinGW\h` in our case.

The remaining entries perform the same role for assembler, object, temporary and application files.

Now give the project a name and save it. Select menu **File / Save As...** and save the project as something like `c:\Play\Tutorial1\Tutorial1.OIL`.

3.2.2 Build / Custom Build

Select menu option **Build / Custom Build** or navigate to it via the GUI tab **2 Builder.**

Press the **Create Templates** button. RTA-OSEK will display the template code for 'main' and offer to save it as `c:\play\Tutorial1\main.c`. Accept its offer.

Press the **Quick Edit** button which will allow you to open `main.c` in your favorite editor[14].

Add in the two callbacks that we need to the existing code so that you end up with the content shown below.

```
/* Template code for 'main' in project: Tutorial1*/
#include "osekmain.h"
OS_MAIN()
{
        StartOS(OSDEFAULTAPPMODE);
        ShutdownOS(E_OK);
}

OS_HOOK(void) StartupHook(void)
{
        vrtaEnterUninterruptibleSection();
        printf("Hello World\n");
        vrtaLeaveUninterruptibleSection();
}

OS_HOOK(void) ShutdownHook(StatusType s)
{
        vrtaEnterUninterruptibleSection();
        printf("Goodbye World\n");
        vrtaLeaveUninterruptibleSection();
        vrtaTerminate();
}
```

Any OSEK-based VECU that you build will use at least one virtual device, so *RTA-OSEK for PC* expects there to be a `.cpp` file (`devices.cpp` by default) that contains your device declarations. *RTA-OSEK for PC* expects it to contain the function `InitializeDevices()` in which you initialize them. At the

[14] Notepad isn't your favourite editor? Use the **Global Settings** tab in the **File\Options** dialog to change it.

moment, we don't have any devices, but we should create a placeholder `devices.cpp`.

Press the **Quick Edit** button again and create `c:\play\Tutorial1\devices.cpp` with the following content[15].

```
void InitializeDevices(void)
{
        // Placeholder
}
```

All that remains is to tell RTA-OSEK to include `devices.cpp` and what files to link. Go back to the RTA-OSEK GUI and press the **Configure** button on the **Custom Build** pane. The **Build Script** tab already contains a call to `rtkbuild.bat`. Add the APP_AOPT and link steps to script as below.

```
set APP_AOPT=-DVRTA_INCLUDE_DEVICES -D$(name)
call $(DestDir_obj)rtkbuild.bat
%LNK% -O2 $(Variant)\o\*.$(objext) %_LIBS% -o"$(DestDir_app)$(name).exe"
```

Press the **Build Now** button and RTA-OSEK will check that you want to create the file `c:\Play\Tutorial1\MinGW\o\rtkbuild.bat`. This is the file that controls the part of the build process that RTA-OSEK knows about or can infer, so say 'Yes'.

RTA-OSEK now creates the low-level files for the OSEK component of your application, and then runs the build script. If you have entered the code as shown above you should see the script complete without errors[16].

If you now look into the directory `C:\Play\Tutorial1\MinGW\o`, you will see 3 object files: `main.o`, `osgen.o` and `osekdefs.o`. The `osgen` and `osekdefs` files contain the code needed to support the OSEK side of your application. So far, so good.

If you've already peeked in the 'app' directory, you will also see `Tutorial1.exe`. If you run this by clicking from within Windows, you might see a quick flash as it runs. A better option is to run it from a command window, where the results will be, as expected[17]:

```
C:\>\Play\Tutorial1\MinGW\app\Tutorial1.exe
Hello World
Goodbye World
```

Now save your project and close RTA-OSEK.

[15] You don't need any `#include` lines for now, because `devices.cpp` will itself be `#included` into another `.cpp` file as part of the build process.
[16] MinGW is alarmingly quiet when compiling source that has no warnings or errors. Other compilers will usually output a bit more progress information during the build so you can see the progress of the build on the RTA-OSEK display.
[17] If you did not install to c:\rta\bin then you may have to add your 'bin' directory to the path.

3.3 Adding devices

In the same way that a real ECU contains hardware such as counters, comparators, switches and the like, a Virtual ECU will also need some devices to work with.

Virtual devices are software components that can declare their name, type, actions and events to the VM's Device Manager, and then respond to action requests, or event queries.

Virtual devices must be declared statically in *RTA-OSEK for PC* so that they can become known to the Device Manager before entering 'main'. Do not attempt to create devices dynamically[18].

RTA-OSEK for PC wraps all of the repetitious 'plumbing' code needed to create a device in a C++ class called `vrtaDevice` that is defined in the file `vrtaDevice.h`. You can use this as a base class from which to create your own devices.

RTA-OSEK for PC also supplies a number of sample devices in the files `vrtaSampleDevices.h` and `vrtaSampleDevices.cpp`. These sample devices are automatically included into your VECU if you use them. The sample devices are documented in detail in chapter 11, but for now let's just see how to use them.

3.3.1 Clocks, counters and compare devices

The vrtaClock device represents a <u>clock source</u> in a real ECU. It ticks independently of the rest of the application at up to 1000 times per second[19]. You can declare a clock named "Clock" that ticks every 5ms like this:

```
#define MSECS_PER_TICK (5)
vrtaClock Clock("Clock", MSECS_PER_TICK);
```

On its own, the clock source is not of much use. You can start it, stop it and change its interval, but it does not maintain a value or raise any interrupts. Think of it as the oscillator in a real ECU.

We need to connect a counter to this clock source. *RTA-OSEK for PC* supplies two sorts of counter; one that counts upwards and one that counts downwards. These lines of code declare two such counters attached to the clock source:

```
vrtaUpCounter CountUp("CountUp", Clock);
vrtaDownCounter CountDown("CountDown", Clock);
```

By default, a counter increments or decrements its current value within the range zero to 2^{32}-1. You can reduce the range of the count using the `SetMin()` and `SetMax()` methods. The normal place to do this is in the

[18] You wouldn't expect a hardware counter to suddenly come into existence on a real ECU either!

[19] The clock device uses Windows multimedia timers, which have a minimum interval of 1ms.

`InitializeDevices()` function we saw earlier. It's also a good place to enable the counter.

```
void InitializeDevices(void)
{
        /* CountUp goes 0...999 then back to 0 */
        CountUp.SetMax(999);
        CountUp.Start();

        /* CountDown goes 1999...1000 then
           back to 1999 */
        CountDown.SetValue(1999);
        CountDown.SetMin(1000);
        CountDown.SetMax(1999);
        CountDown.Start();
}
```

You can read the current value of a counter using `<counter>.GetValue()`. But what we'd really like to do is to raise an interrupt when the counter reaches a certain value.

Enter the `vrtaCompare` virtual device.

When you declare a compare device you can attach it to a counter and specify an interrupt that fires when the value of the counter reaches a specific value. In the example below there are two comparators connected to the same counter. The first one raises interrupt 1 when the counter reaches value 5. The second raises interrupt 2 when the counter value reaches 15.

```
vrtaCompare Compare5("Compare5",CountUp,5,1);
vrtaCompare Compare15("Compare15",CountUp,15,2);
```

Hopefully it is clear from this that you can very easily construct a chain of timing elements of arbitrary complexity by simple combinations of clock, counter and compare devices.

One point worth emphasizing is that `vrtaClock` sources run independently of the application. If you use the `ApplicationManager` device to pause the application thread, the clocks continue to run and any counter/compare devices that are attached can continue to raise interrupts that will be serviced once the application thread resumes.

3.3.2 Sensors

RTA-OSEK for PC provides sample 'sensor' devices that are intended to represent inputs to an ECU. Sensors in the real world could include switches, thermocouples, pressure monitors and so on.

Sensors have a minimum value, a maximum value and a current value in the same way as a counter. The vanilla `vrtaSensor` device values range from zero to 2^{32}-1, but you can change the upper limit for this using the `SetMax()` method.

Two derivatives of `vrtaSensor` are `vrtaSensorToggleSwitch` and `vrtaSensorMultiwaySwitch`. The toggle switch can have a position value zero or 1. The multiway switch can have an upper limit specified in its declaration.

e.g.

```
vrtaSensor Throttle("Throttle");
vrtaSensorToggleSwitch EjectSwitch("EjectSwitch");
vrtaSensorMultiwaySwitch Gear("Gear", 5); // 0 to 5
```

The values of a sensor can be changed remotely from programs such as `vrtaMonitor` and the COM Bridge (see chapter 13). You can read the value of the sensor directly from your application via the `GetValue()` method.

Alternatively you can attach a `vrtaCompare` device to the sensor and raise an event when a certain value is set.

e.g.

```
vrtaCompare Eject("Eject",EjectSwitch,1,3);
```

This compare device causes interrupt 3 to be raised when the `EjectSwitch` position becomes 1.

3.3.3 Actuators

RTA-OSEK for PC 'actuators' represent outputs from an ECU. The base `vrtaActuator` device can be set to a value from zero to 2^{32}-1. It raises an event when its value changes, so external programs such as `vrtaMonitor` can detect changes in the VECU's output.

As previously, you can attach a `vrtaCompare` device to an actuator so that you can raise an interrupt when a set value is reached[20].

Trivial specializations of `vrtaActuator` are `vrtaActuatorLight` (values zero and 1), `vrtaActuatorDimmableLight` ('levels' zero to n) and `vrtaActuatorMultiColorLight` ('colors' zero to n).

[20] By our definition of an actuator, it is an output device so you wouldn't expect it to be able to raise an interrupt: that is really the role of a sensor attached to it. Nevertheless, you may find this capability useful in certain cases.

e.g.

```
vrtaActuator Speedometer("Speedometer");
vrtaActuatorLight BrakeLight("BrakeLight");
vrtaActuatorDimmableLight
InteriorLight("InteriorLight",20);
vrtaActuatorMultiColorLight
FuelIndicator("FuelIndicator",4);
```

3.3.4 IO

The `vrtaIO` virtual device is a more general purpose component. It simulates a block of 32-bit values in an ECU's IO or memory space. You (or an external monitor) can set or inspect values in the individual elements. An event is raised when a value is changed, but there is no ability to generate an interrupt based on the value.

3.3.5 Custom devices

The sample devices provided with *RTA-OSEK for PC* are a good starting point when creating a VECU, but you will want to create your own devices to reflect your own environment. Thanks to C++ inheritance this is a straightforward process that we will cover shortly – see section 3.4.11 and chapter 4.

3.4 Tutorial2

Reopen the Tutorial1 project then use **File / Save As...** to make a copy in a new directory as `c:\Play\Tutorial2\Tutorial2.OIL`. Copy `main.c` and `devices.cpp` from the `Tutorial1` directory to `Tutorial2`.

We are going to create a very artificial VECU to demonstrate the use of virtual devices:

- There will be 4 input switches: Accelerate, Brake, Left and Right.
- There will be a speed setting actuator.
- There will be a direction indicating actuator.
- The Accelerate and Brake switches cause the speed to increase / decrease by one notch on each zero to one transition under interrupt control.
- The Left and Right switches are polled from an OSEK task every 100ms and cause a one degree change in direction each time they are sampled at '1'.
- The speed starts at zero and limits at 100. The speed is preserved if the program resets, but not if the program restarts.
- The direction is from zero to 359 degrees. The direction is preserved over reset AND program restart.
- There will be on-screen feedback of the current speed and direction.

3.4.1 Devices

Edit `devices.cpp` so that it contains the following code.

```
//----------------------------------------
// Device declarations
//----------------------------------------
#include <devices.h>

#define MSECS_PER_POLL (100)
#define ACCEL_ISR (1)
#define BRAKE_ISR (2)
#define POLL_ISR (3)

// Switches
vrtaSensorToggleSwitch Accelerate("Accelerate");
vrtaSensorToggleSwitch Brake("Brake");
vrtaSensorToggleSwitch Left("Left");
vrtaSensorToggleSwitch Right("Right");

// Actuators
vrtaActuator Speed("Speed");
vrtaActuator Direction("Direction");

// Comparators
vrtaCompare
AccelDetect("AccelDetect",Accelerate,1,ACCEL_ISR);
```

```
vrtaCompare
BrakeDetect("BrakeDetect",Brake,1,BRAKE_ISR);

// Clock
vrtaClock ClockSource("ClockSource",MSECS_PER_POLL);
vrtaUpCounter PollCounter("PollCounter", ClockSource);
vrtaCompare PollCompare("PollCompare", PollCounter, 0,
POLL_ISR);

// Data
#define DATA_SIZE (1)
#define DIR_DATA (0)
vrtaIO  PersistentData("PersistentData",DATA_SIZE);

// Status
#include <vrtaLoggerDevice.cpp>
Logger Status("Status");
//------------------------------------------
int status_printf(const char* format, ...)
{
        va_list argptr;
        va_start(argptr, format);
        int ret = Status.printf(format, argptr);
        va_end(argptr);
        return ret;
}
//------------------------------------------
void InitializeDevices(void)
{
        Speed.SetMax(100);
        Direction.SetMax(359);

        PollCounter.SetMin(0);
        PollCounter.SetMax(0);
        PollCounter.Start();

        ClockSource.Start();

        Speed.PersistThroughReset(true);
        PersistentData.PersistThroughReset(true);
        Direction.SetValue(
                PersistentData.GetValue(DIR_DATA)
        );
}
//------------------------------------------
```

Hopefully the declarations for the switches, actuators, clock devices and IO will be clear.

3.4.2 Logger

The Logger device is not included as part of the standard set of sample devices, but you will find it very useful if you want to output diagnostic text from an application. You can make `printf()` style calls to a Logger device and it can output the text to the console window and/or a file. If you create a Logger device with the special name "Status", then the VM's embedded GUI will display its last line in its status bar. `vrtaMonitor` will do the same.

Use a Logger device to output text to the VECU's console. Do not use direct `printf()` calls. The Logger device has the interrupt protection that is needed when making non VM or non-OSEK API calls from your application thread.

Now we come to an interesting issue. Your logger device is a C++ object with a nice set of methods for outputting text and saying where to direct the text to. You can make calls such as `Status.printf("Boo")` so that you can indicate to the outside world what is happening.

But your OSEK application is written in C not C++, so it does not understand the Status object. How can it make use of it?

3.4.3 Interfacing C code with devices

There are two answers to this.

1) Your C code can use the VM API call `vrtaSendAction` to send a string to the device's 'Print' action. You'll see how to do this later, but the code would look something like this[21]:

```
vrtaAction act;
char * pText_to_send = "Boo";
act.devID = status_device_id;
act.devAction = 1;
act.devActionLen = strlen(pText_to_send);
act.devActionData = pText_to_send;
vrtaSendAction(act.devID, &act);
```

2) Write a C / C++ interface function. This is what is done in the code above. The function `status_printf` is a simple wrapper function that is intended to be callable from C code so your code becomes:

```
status_printf("Boo")
```

You must provide a prototype for this function that can be seen by both the C and C++ source code that declares `status_printf` to be a C rather than C++ function. For this reason we need to add the file `devices.h`, with the following contents.

[21] This won't work if the string length is less than 16 bytes because the string has to be copied into the action's embedded data area. This is discussed later.

```
/* Interface between C and C++ */

#ifdef __cplusplus
extern "C" {
#endif

extern int status_printf(const char* format, ...);

#ifdef __cplusplus
}
#endif
```

This file must get `#included` into any C or C++ files that reference `status_printf`.

3.4.4 Device Initialization

The initialization code in our `InitializeDevices()` function sets both the min and max values for the `PollCounter` to zero. You will note that `PollCompare` is also set to match on zero. This has the effect of raising an interrupt every clock tick, because each time the `ClockSource` ticks the `PollCounter` is 'incremented'. Incrementing past its maximum value (0) causes the count to reset to the minimum value (0). Each time the `PollCounter` gets 'incremented' the new value is passed to the `PollCompare` – which matches on zero every time.

The calls to `PersistThroughReset()` tell the `Speed` and `PersistentData` devices to preserve their data if the program resets. Note that a reset is not the same as killing the program and restarting it manually. We do have a requirement that the direction is preserved over a restart that is not satisfied by this – but we will sort that out a bit later.

3.4.5 main()

We also need to modify `main.c` so that the program does not simply start and stop. Enter the code below. You will notice the use of `status_printf()`. `vrtaIsAppFinished()` returns false normally or true if the VECU should terminate. `vrtaIsIdle()` waits for the specified number of milliseconds. During this time the processor is assigned to a different thread. Virtual interrupts can still occur during a call to `vrtaIsIdle()`.

```
/* Template code for 'main' in project: Tutorial2 */
#include "osekmain.h"
#include <devices.h>

OS_MAIN()
{
        StartOS(OSDEFAULTAPPMODE);

        while(!vrtaIsAppFinished()) {
                vrtaIsIdle(5);
        }

        ShutdownOS(E_OK);
}
OS_HOOK(void) StartupHook(void)
{
        status_printf("Hello World");
}

OS_HOOK(void) ShutdownHook(StatusType s)
{
        status_printf("Shutdown occurred");
        vrtaTerminate();
}
```

3.4.6 Trial run 1

Let's check that was all entered correctly. Save your project and perform a build (F9).

Hopefully the build will be successful. If not just go back and check that you have entered everything correctly.

We can run the program by typing in its full path, but you'll probably find it easier to run it directly from the RTA-OSEK GUI.

If you don't already have the custom buttons below, press the Configure button on the Custom Build page. Select the Custom Buttons tab in the configure dialog and make the following entries:

Button 1

Name:	`&Quick Edit`
Content:	`$(EDITOR) $(OPENFILE)`

Button 2

Name:	`&Run`
Content:	`$(DestDir_app)$(name)`

Button 3

Name:	`RTA-&Trace`
Content:	`$(RTABase)\bin\rta-trace.exe "localhost.RTAOSEK-VRTA:$(destdir_app)$(name).rta"`

Button 4

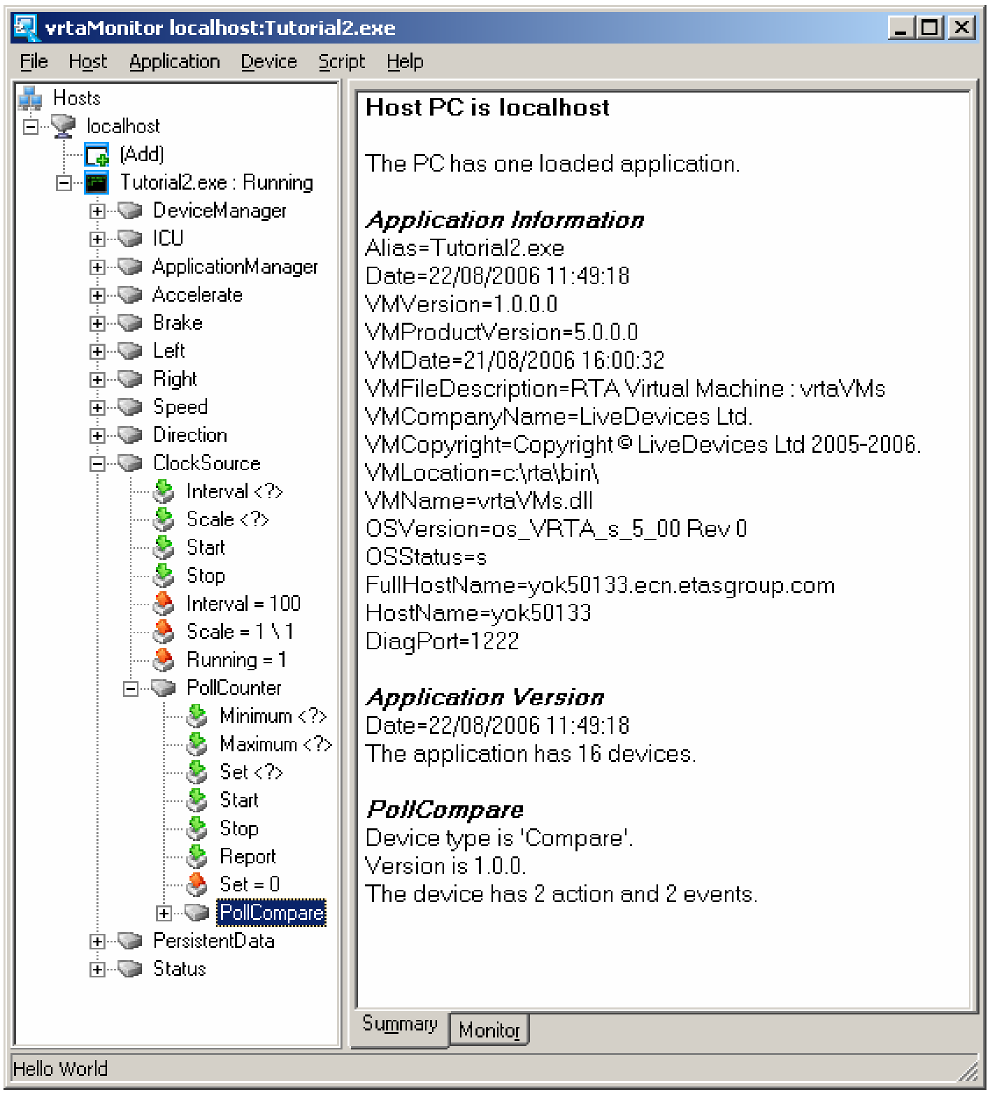
Name:	`RTA-Trace &Server`
Content:	`$(RTABase)\bin\rta-trace-server.exe`

You can now use the Run button to launch `Tutorial2.exe`. The RTA-TRACE entries will come in handy later.

When you run your new VECU, you will get an empty console screen plus an embedded GUI. The GUI shows a few details about your VECU and gives you the ability to pause, resume, reset and terminate it. You should see "Hello World" on the GUI's status bar.

If you press Ctrl+M when in the embedded GUI, then you will launch an instance of `vrtaMonitor` that is connected to the VECU.

Note that it too has "Hello World" in the status bar.



Host PC is localhost

The PC has one loaded application.

Application Information
Alias=Tutorial2.exe
Date=22/08/2006 11:49:18
VMVersion=1.0.0.0
VMProductVersion=5.0.0.0
VMDate=21/08/2006 16:00:32
VMFileDescription=RTA Virtual Machine : vrtaVMs
VMCompanyName=LiveDevices Ltd.
VMCopyright=Copyright © LiveDevices Ltd 2005-2006.
VMLocation=c:\rta\bin\
VMName=vrtaVMs.dll
OSVersion=os_VRTA_s_5_00 Rev 0
OSStatus=s
FullHostName=yok50133.ecn.etasgroup.com
HostName=yok50133
DiagPort=1222

Application Version
Date=22/08/2006 11:49:18
The application has 16 devices.

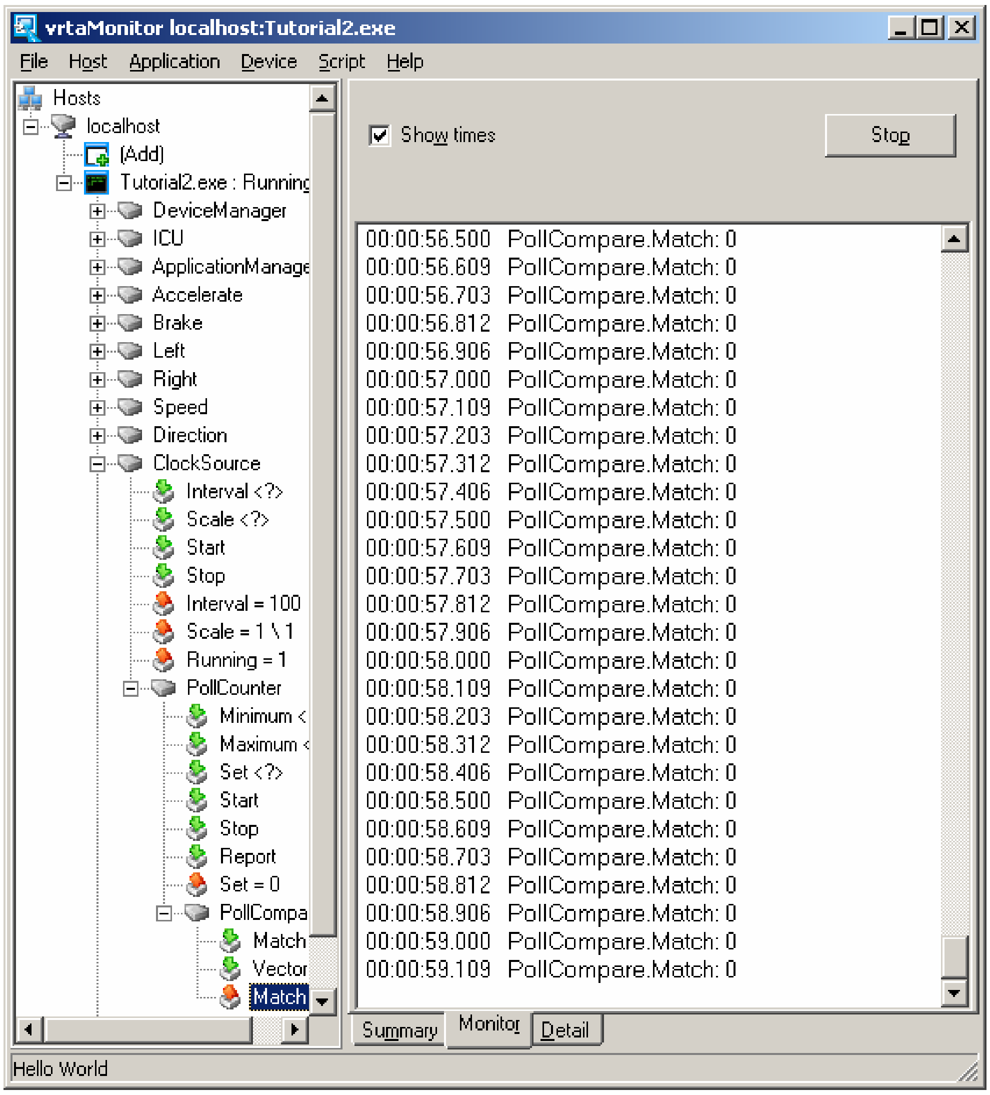
PollCompare
Device type is 'Compare'.
Version is 1.0.0.
The device has 2 action and 2 events.

Take some time to explore the devices in the monitor. You should be able to work out most of the features by checking out the main menu, clicking (and especially right-clicking) on elements in the left hand navigation pane and looking at the tabs on the right hand side. If you get stuck, nip ahead to chapter 6 which describes the monitor.

One thing that is worth pointing out is that the monitor has worked out that the `PollCompare` device 'belongs to' the `PollCounter`, which in turn 'belongs to' the `ClockSource`. It has therefore arranged these devices in a hierarchy. The same applies to the `Accelerate` and `Brake` compare devices.

3.4.7 Summary so far

We haven't really written much code yet, but a large part of the framework is in place. In fact, we can even see that the clock chain is working. Drag the 'Match' Event of the `PollCompare` device from the left hand side over to the right-hand side. This causes the monitor to hook all match events from the device and display them in the **Monitor** tab. If you tick the **Show Times** checkbox you will also see the time that each event was raised in the VECU[22].



```
00:00:56.500  PollCompare.Match: 0
00:00:56.609  PollCompare.Match: 0
00:00:56.703  PollCompare.Match: 0
00:00:56.812  PollCompare.Match: 0
00:00:56.906  PollCompare.Match: 0
00:00:57.000  PollCompare.Match: 0
00:00:57.109  PollCompare.Match: 0
00:00:57.203  PollCompare.Match: 0
00:00:57.312  PollCompare.Match: 0
00:00:57.406  PollCompare.Match: 0
00:00:57.500  PollCompare.Match: 0
00:00:57.609  PollCompare.Match: 0
00:00:57.703  PollCompare.Match: 0
00:00:57.812  PollCompare.Match: 0
00:00:57.906  PollCompare.Match: 0
00:00:58.000  PollCompare.Match: 0
00:00:58.109  PollCompare.Match: 0
00:00:58.203  PollCompare.Match: 0
00:00:58.312  PollCompare.Match: 0
00:00:58.406  PollCompare.Match: 0
00:00:58.500  PollCompare.Match: 0
00:00:58.609  PollCompare.Match: 0
00:00:58.703  PollCompare.Match: 0
00:00:58.812  PollCompare.Match: 0
00:00:58.906  PollCompare.Match: 0
00:00:59.000  PollCompare.Match: 0
00:00:59.109  PollCompare.Match: 0
```

To stop monitoring an event, drag the event onto the **Stop** button.

When you are ready, close the monitor and VECU.

[22] Event times are recorded using the Windows API `GetTickCount()`. This typically has a resolution of around 15ms.

3.4.8 Tasks and ISRs

We now create the OSEK elements that implement the required functionality. Using the 'Basic Data Entry' screen in RTA-OSEK, add the following:

- A task called `DirectionPoll`
- A Category 2 ISR named `isrAccel` with priority 1 and vector 1.
- A Category 2 ISR named `isrBrake` with priority 1 and vector 2.
- A Category 2 ISR named `isrPoll` with priority 2 and vector 3.
- An OSEK Counter named `AlarmCounter` with a tick rate of 100 real time ms driven by `isrPoll`.
- An OSEK Alarm named `Poller` that activates task `DirectionPoll`.
- In the Startup tab, set `Poller` to **Autostart**.
- In the Stimuli tab of the Planner section, set the alarm's arrival pattern to 1 counter tick and start offset 1[23].

Go back to the **Custom Build** pane and press the **Create Templates** button. **DO NOT** overwrite `main.c`, but do allow RTA-OSEK to create the remaining files.

We will now add some code into the templates.

Edit `main.c` so that `OS_MAIN()` looks like this:

```
OS_MAIN()
{
        StartOS(OSDEFAULTAPPMODE);

        show_status();
        while(!vrtaIsAppFinished()) {
                vrtaIsIdle(5);
        }

        ShutdownOS(E_OK);
}
```

Edit `isrPoll.c` thus:

```
/* Template code for 'isrPoll' in project:
Tutorial2 */
#include "isrPoll.h"
#include <devices.h>
ISR(isrPoll)
{
        Tick_AlarmCounter();
}
```

[23] If you leave the start offset as zero, the alarm will take a long time to start because the counter is already at zero in `StartOS()`, so the first offset you can get is 1.

Edit `isrBrake.c` thus:

```
/* Template code for 'isrBrake' in project: Tutorial2 */
#include "isrBrake.h"
#include <devices.h>
ISR(isrBrake)
{
        change_speed(-1);
}
```

Edit `isrAccel.c` thus:

```
/* Template code for 'isrAccel' in project: Tutorial2 */
#include "isrAccel.h"
#include <devices.h>
ISR(isrAccel)
{
        change_speed(+1);
}
```

Edit `DirectionPoll.c` thus:

```
/* Template code for 'DirectionPoll' in project:
Tutorial2 */
#include "DirectionPoll.h"
#include <devices.h>
TASK(DirectionPoll)
{
        if (left_pressed()) {
                change_direction(-1);
        }
        if (right_pressed()) {
                change_direction(+1);
        }
        TerminateTask();
}
```

Update `devices.h` to add the C / C++ interface function prototypes:

```
extern int show_status(void);
extern int left_pressed(void);
extern int right_pressed(void);
extern int change_direction(int amount);
extern int change_speed(int amount);
```

We now add the C / C++ functions to `devices.cpp`. They are clearly quite simple wrappers to the devices.

```
//----------------------------------------
int left_pressed(void)
{
        return Left.Value();
}
//----------------------------------------
int right_pressed(void)
{
        return Right.Value();
}
//----------------------------------------
int show_status(void)
{
        Status.printf("Speed %d, Direction %d",
                Speed.Value(),
                Direction.Value());
}
//----------------------------------------
int change_direction(int amount)
{
        int newvalue = Direction.Value() + amount;
        while (newvalue < 0) {
                newvalue += 360;
        }
        while (newvalue > 359) {
                newvalue -= 360;
        }
        PersistentData.SetValue(DIR_DATA,newvalue);
        Direction.SetValue(newvalue);
        show_status();
}
//----------------------------------------
int change_speed(int amount)
{
        int newvalue = Speed.Value() + amount;
        if ((newvalue >=0) && (newvalue <=100)) {
                Speed.SetValue(newvalue);
                show_status();
        }
}
```

If you try to run this now, you'll find that nothing much appears to respond to the inputs. This is because we need to enable (or unmask) the 3 interrupts that we are using. Add the code below into `InitializeDevices()`. It sends the Unmask action to the VM's ICU device, passing in each interrupt number in turn.

```
vrtaAction action;
action.devID = ICU_DEVICE_ID;
action.devAction = ICU_ACTION_ID_Unmask;
action.devActionLen =
        sizeof(action.devEmbeddedData.uVal);
action.devActionData = NULL;
action.devEmbeddedData.uVal = ACCEL_ISR;
vrtaSendAction(ICU_DEVICE_ID,&action);
action.devEmbeddedData.uVal = BRAKE_ISR;
vrtaSendAction(ICU_DEVICE_ID,&action);
action.devEmbeddedData.uVal = POLL_ISR;
vrtaSendAction(ICU_DEVICE_ID,&action);
```

Nearly there.

For no other reason than to show you some interesting stuff, we are now going to add a 'spring' to the Accelerate and Brake switches so that they flip back to zero after being pushed. We will do this in an *RTA-OSEK for PC* Thread.

3.4.9 Threads

RTA-OSEK for PC allows you to create any number of threads of execution that run independently of your application thread (the OSEK thread). These threads are native Windows threads with a small amount of protection built in. You can make Windows API calls from within a thread without having to protect them from *RTA-OSEK for PC* interrupts. You can access the VECU's devices and even raise interrupts from within a thread, which makes them an excellent choice for interfacing to real hardware.

To see a thread at work add the following declaration just before `InitializeDevices()`:

```
extern void AsyncThread(void); // Forward declaration
```

Then add this to the end of `InitializeDevices()`:

```
vrtaSpawnThread(AsyncThread);
```

The code for the thread can be added at the end of `devices.cpp`. The basic shape of an *RTA-OSEK for PC* thread is usually something like this:

```
void AsyncThread(void)
{
        while (!vrtaIsAppFinished()) {
                vrtaIsIdle(100);      // Sleep 100ms
        }
}
```

You should check `vrtaIsAppFinished()` regularly within a thread so that the VECU can perform an orderly tidy up when asked to terminate. You should use `vrtaIsIdle()` to yield control to other threads if you have no work to do.

In our thread, we want to hook event changes in the `AccelDetect` and `BrakeDetect` compare devices. Whenever the Match event fires for one of these, we know that the associated switch has been pressed. We then reset the switch value to zero. Note that hooking the `Accelerate` and `Brake` Position event will not work because events are raised before the compare devices are informed. If you reset the switch in the event hook, the compare devices only ever see the zero value.

The code that we need to add to `devices.cpp` to implement the thread and hook code is shown below.

```
static vrtaErrType ListenCallback(
        const void *instance,
        const vrtaEvent *event)
{
        // Has 'Accelerate' become 1?
        if (
         (event->devID == AccelDetect.GetID())  &&
         (event->devEmbeddedData.uVal == 1)) {
                // Set directly
                Accelerate.SetValue(0);
        }


        // Has 'Brake' become 1?
        if ((event->devID == BrakeDetect.GetID())  &&
         (event->devEmbeddedData.uVal == 1)){
                // Set via an action
                vrtaAction act;
                act.devAction = 1;
                act.devActionLen = sizeof(unsigned);
                act.devActionData = NULL;
                act.devID = Brake.GetID();
                act.devEmbeddedData.uVal = 0;
                vrtaSendAction(act.devID, &act);
        }
}
//------------------------------------------
void AsyncThread(void)
{
        // Create a listener and associate its callback
        vrtaEventListener tListener =
                vrtaEventRegister(ListenCallback, 0);

        // Hook event 1 of AccelDetect into listener
        vrtaHookEvent(
                tListener,
                AccelDetect.GetID(), 1, true);

        // Hook event 1 of BrakeDetect into listener
        vrtaHookEvent(
                tListener,
                BrakeDetect.GetID(), 1, true);

        // Nothing else to do while app is running
        while (!vrtaIsAppFinished()) {
                vrtaIsIdle(100);
        }

        // Tidy up hooks on exit from thread
        vrtaHookEvent(
                tListener,
                AccelDetect.GetID(), 1, false);
        vrtaHookEvent(
                tListener,
                BrakeDetect.GetID(), 1, false);
}
```

Notice that for illustration purposes, different methods are used to reset the Accelerate and Brake devices.

3.4.10 Trial run 2

Save your project and perform a build (F9).

Because you are getting good at this, you will now have a (nearly) fully working VECU, so let's run it.

We'll need `vrtaMonitor` to feed some inputs and view the output, so start it up (Ctrl+M) from the embedded GUI.

Expand the `Accelerate` device and drag the Position event to the right hand side. Double click the Position action (not the event) and enter '1' as the data value. Press OK. You will see the position value change to one and then zero in the monitor window. The status bar will show that the speed has increased. Double-click on the Position action a few times and you will see the speed go up further.

Do the same for the `Brake` device and notice the speed decrease.

Now expand the `Left` device and send its Position action value '1'. The direction will go East to West until you change it back to zero.

The same applies to the `Right` device, though obviously that will make the direction go West to East.

If you now reset the VECU (for example using the Application menu in `vrtaMonitor`), the VECU console and GUI will flash off and then return. The speed and direction will have persisted across the reset.

3.4.11 Non-volatile data

We have seen that the speed and direction values persist over a VECU reset. But we don't yet have a way to keep data between completely different runs of the VECU.

We'd really like to have something that looks like Flash memory in a real ECU. In this release of *RTA-OSEK for PC*, there is no direct support for nonvolatile memory so we will have to make our own version.

We will do this by creating a `vrtaFlash` device that inherits from `vrtaIO`. We can then change the type of `PersistentData` to `vrtaFlash` and the job is done.

The code to do this is shown below. Just replace the current declaration for `PersistentData` with these lines.

```
// Add some persistence to vrtaIO to simulate nvram.
// In this version we store the data in the file
// "VECU.flash". The file gets read in when the device
// starts, and written when the device is stopped.
#include <stdio.h>
class vrtaFlash : public vrtaIO {

protected:
    void Starting(void) {
        vrtaIO::Starting();
        FILE *f = fopen("VECU.flash","rb");
        if (f) {
            char buffer[100];
            if (GetPersistentDataSize() ==
             fread(buffer,1,GetPersistentDataSize(), f)
        ) {
            memcpy(
                GetPersistentData(),
                buffer,
                GetPersistentDataSize());
        }
        fclose(f);
        }
    }

    void Stopping(void) {
        FILE *f = fopen("VECU.flash","wb");
        fwrite(
            GetPersistentData(),
            1,
            GetPersistentDataSize(),
            f);
        fclose(f);
        vrtaIO::Stopping();
    }
public:
    // Constructor
    vrtaFlash(const vrtaTextPtr name, unsigned elements)
        : vrtaIO(name, elements) {};

};
vrtaFlash    PersistentData("PersistentData",DATA_SIZE);
```

You will now find that if you rebuild your VECU that it will remember the direction it was last pointing in when it restarts.

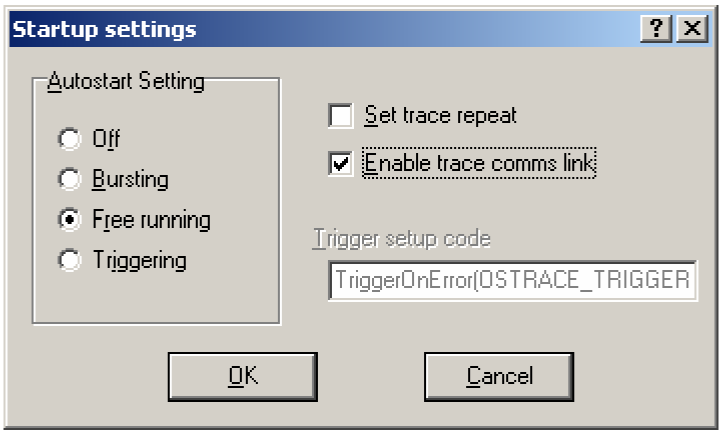
In a real-life application you might want to move away from a hard-coded file name, to avoid conflicts where two applications try to access the same file.

3.4.12 RTA-TRACE

If you have RTA-TRACE installed, you can get a detailed view of the internal operation of your VECU.

Close any instances of the VECU and monitors, and select the **3 RTA-TRACE** tab in the RTA-OSEK GUI.

Use the **Configuration** pane to set the trace type to **Advanced**. Set the **autostart** values to **Free running / enable trace link**.


Startup settings

Autostart Setting
- Off
- Bursting
- [✓ Free running]
- Triggering

Set trace repeat ☐

Enable trace comms link ☑

Trigger setup code: TriggerOnError(OSTRACE_TRIGGER

Now try to rebuild.

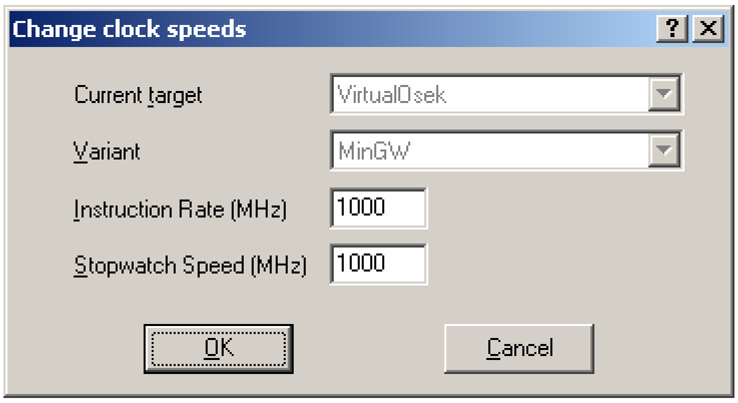
You should get warnings about missing stopwatch and overrun functions. In build status 's', RTA-OSEK does not collect or police timing information, so we have not yet needed to provide any time-related code.

RTA-TRACE of course does need to know about the time at which things happen, so we must give it the help it needs. Add the code below to `devices.cpp`.

```
//----------------------------------------
OS_NONREENTRANT(StopwatchTickType) GetStopwatch(void)
{
/*
    vrtaReadHPTime(x) returns the current time
    in 'xticks', where there are 'x' xticks per
    second. So the code below returns time in the
    units defined as the stopwatch rate in the
    RTA-OSEK GUI.
*/
    return vrtaReadHPTime(
         (1000000000 / OS_NS_PER_CYCLE) /
         OS_CYCLES_PER_SWTICK
    );
}

#ifdef OSTRACE_ENABLED
//----------------------------------------
OS_NONREENTRANT(StopwatchTickType)
osTraceStopwatch(void)
```

```
{
/*
    Ensure that the trace stopwatch is the same
    as the system stopwatch
*/
  return GetStopwatch();
}
#endif

#if defined(OS_ET_MEASURE)
//-----------------------------------------
OS_NONREENTRANT(StopwatchTickType)
GetStopwatchUncertainty(void)
{
    return (StopwatchTickType) 1U;
    /* Not really relevant for this target */
}
#endif
//-----------------------------------------
OS_HOOK(void) OverrunHook(void) {
        status_printf("Overrun occurred");
}
//-----------------------------------------
```

You may remember that the project was set up initially with a 1kHz stopwatch. Using the high-performance counter means that we can now do a lot better than that, so select the RTA-OSEK menu **Target / Timing data** and change the clocks to something more sensible.



Change clock speeds

Current target	VirtualOsek
Variant	MinGW
Instruction Rate (MHz)	1000
Stopwatch Speed (MHz)	1000

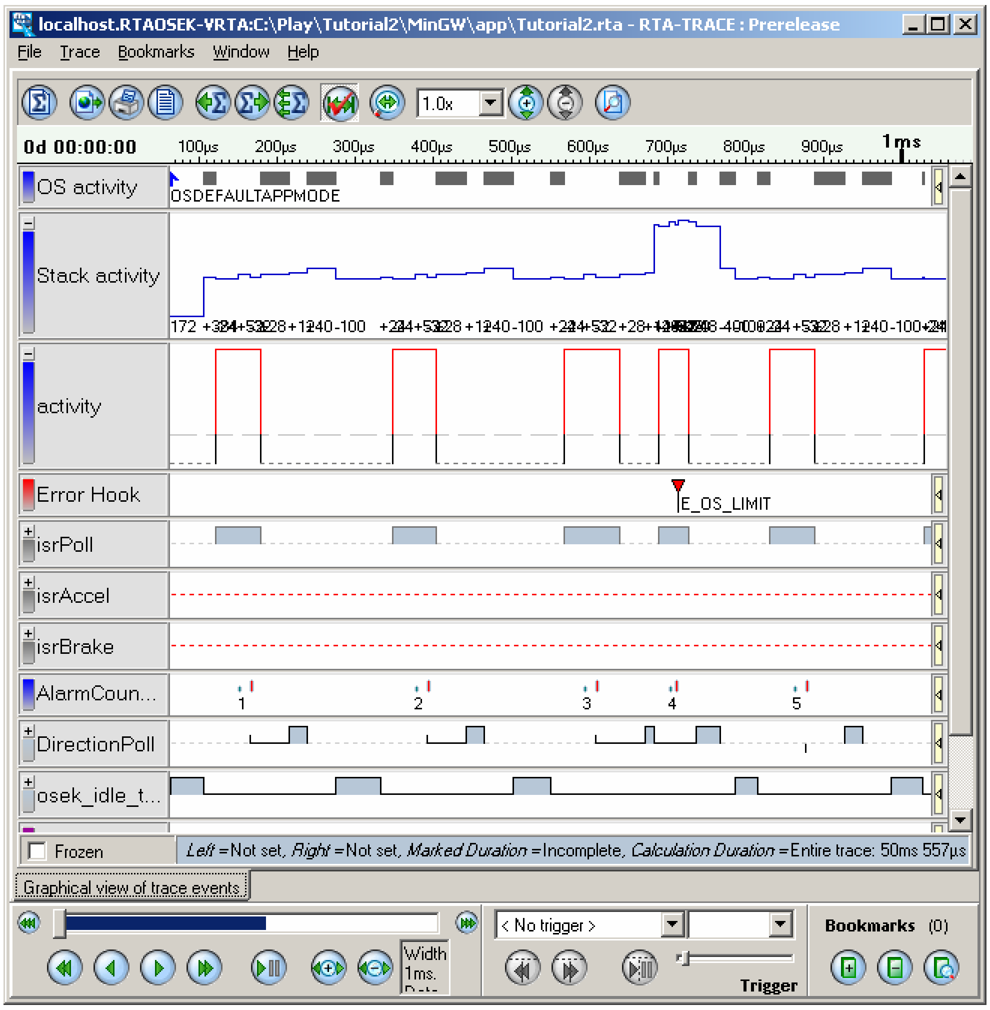
Finally we must add some code in the idle task to pump the trace data out of the application. Edit `main.c` so that it now looks like this:

```
/* Template code for 'main' in project: Tutorial2 */
#include "osekmain.h"
#include <devices.h>

OS_MAIN()
{
        StartOS(OSDEFAULTAPPMODE);

        show_status();
        while(!vrtaIsAppFinished()) {
#ifdef OSTRACE_ENABLED
                CheckTraceOutput();
                UploadTraceData();
#endif
                vrtaIsIdle(5);
        }

        ShutdownOS(E_OK);
}
OS_HOOK(void) StartupHook(void)
{
        status_printf("Hello World");
}

OS_HOOK(void) ShutdownHook(StatusType s)
{
        status_printf("Shutdown occurred");
        vrtaTerminate();
}
```

Now rebuild the VECU (F9).

Start the RTA-TRACE Server from the custom button you set up earlier, and then press the RTA-TRACE button. RTA-TRACE will start up, read the VECU's configuration information and then launch it automatically for you. What service!

Your mileage will vary, but we get a trace looking like the one below.



Left =Not set, *Right* =Not set, *Marked Duration* =Incomplete, *Calculation Duration* =Entire trace: 50ms 557µs

Hmm… that's interesting. The trace is showing `E_OS_LIMIT` during the fourth instance of `isrPoll`[24]. We weren't expecting that!

It's over to you now. You have all the tools and knowledge that you need to sort this one out yourself[25].

[24] Depending upon the speed of your machine, this may occur in a different position.
[25] Yes indeed, an 'exercise for the reader'.

3.4.13 Addendum

Just to prove that you can link to real hardware, you might like to change the implementation of the `AsyncThread` as shown below. It just peeks at the state of the arrow keys on your keyboard and makes calls to your switches.

```
//---------------------------------------
void AsyncThread(void)
{
    vrtaEventListener tListener =
                vrtaEventRegister(ListenCallback, 0);
    vrtaHookEvent(
            tListener, AccelDetect.GetID(), 1, true);
    vrtaHookEvent(
            tListener, BrakeDetect.GetID(), 1, true);

    while (!vrtaIsAppFinished()) {
        if (GetAsyncKeyState(VK_LEFT) < 0) {
             if (Left.GetValue() == 0) {
                   Left.SetValue(1);
             }
        } else {
             if (Left.GetValue() == 1) {
                   Left.SetValue(0);
             }
        }
        if (GetAsyncKeyState(VK_RIGHT) < 0) {
             if (Right.GetValue() == 0) {
                   Right.SetValue(1);
             }
        } else {
             if (Right.GetValue() == 1) {
                   Right.SetValue(0);
             }
        }
        if (GetAsyncKeyState(VK_UP)  < 0) {
             Accelerate.SetValue(1);
        }
        if (GetAsyncKeyState(VK_DOWN)  < 0) {
             Brake.SetValue(1);
        }
       vrtaIsIdle(100);
    }

    vrtaHookEvent(
        tListener, AccelDetect.GetID(), 1, false);
    vrtaHookEvent(
        tListener, BrakeDetect.GetID(), 1, false);
}
```

4 ECUs and Virtual Devices

At its most basic level, a virtual device is simply a software component that has a name and provides functions that can be called to send it a command (action) or query its status (event).

The 'action' callback function gets passed information through a `vrtaAction` structure.

The 'state' callback function gets passed information through a `vrtaEvent` structure.

Virtual devices can be written from scratch using C code, but we recommend that they are implemented as C++ classes that derive from the `vrtaDevice` class that is defined in `vrtaDevice.h`.

Many examples of how to write such classes are provided with *RTA-OSEK for PC*, most notably in `vrtaSampleDevices.h/.cpp`. This chapter covers some of the issues that you should understand when writing your own devices.

4.1 Registering the device

You have to tell the VM that your device exists by calling the `vrtaRegisterVirtualDevice()` API. This must be done before `vrtaStart()` is called. (It is done automatically for you if you are using a class that derives from `vrtaDevice`.)

When registering a device, you supply the following information:

- **name**: This is the name that external monitor programs will see when accessing the device. Each device in a VECU must have a different name.
  e.g. "LeftWindowSwitch", "EjectorSeatTrigger"

- **info**: This is a string containing information about the device in the form "<tag1>=<value1>\n<tag2>=< value 2>…". As a minimum the string should contain Type and Version tags. This information is used by external monitor programs.
  e.g. "Type=Thruster\nVersion=1.2.3\n"

- **list of events**: This is a string in the same format as above that lists the events that the device supports and the data format for each event. The tags are the event names and the values are the data format descriptions. These are explained in detail in section 9.3.
  e.g. "Value=%u,%u(%u)\nValues=%a\n"

- **list of actions**: This is a string in the same format as above that lists the actions that the device supports and the data format for each action. The tags are the action names and the values are the data format descriptions. These are explained in detail in section 9.3.
  e.g. "Value=%u,%u\nValues=%a\nGetValue=%u\nGetValues\n"

- **action callback function**: This is a reference to the C function that will be called when an action is sent to the device. See the section below on how to handle action requests.
- **state callback function**: This is a reference to the C function that will be called when a status query is sent to the device. See the section below on how to handle status queries.

4.2 Handling actions

The action callback function that you register gets called when code in the VECU calls `vrtaSendAction()`, or when an external monitor sends data via the diagnostic interface.

The callback can be invoked from *any* thread; therefore the callback must take care of any reentrancy issues.

Often you will raise an event as a result of receiving an action.

If your device is written as a C++ class that inherits from `vrtaDevice`, the action callback is translated into a call to your `OnAction` method. The basic form of the `OnAction` method is shown below.

```
//-----------------------------------------------------
vrtaErrType mydev::OnAction(const vrtaAction *action)
//-----------------------------------------------------
{
  switch (action->devAction) {
        case 1:
              /* respond to action 1 */
              RaiseEvent(...);
              break;

        default:
              return ErrorAction(action);
  }
  return OKAction(action);
}
```

4.3 Handling state queries

The state callback function that you register gets called when code in the VECU calls `vrtaGetState()`, or when an external monitor queries the device via the diagnostic interface.

The callback can be invoked from *any* thread; therefore the callback must take care of any reentrancy issues.

If your device is written as a C++ class that inherits from `vrtaDevice`, the event callback is translated into a call to your `AsyncGetState` method. The basic form of the `AsyncGetState` method is shown below.

```
//-----------------------------------------------------
vrtaErrType mydev::AsyncGetState(vrtaEvent *event)
//-----------------------------------------------------
{
  switch (event->devEvent) {
        case 1:
              /* Update *event */
              break;

        default:
              return ErrorState(event);
  }
  return OKState(event);
}
```

By convention, `AsyncGetState` returns the value of the most recent `RaiseEvent` for the event in question so that the state of a device can be tracked by either 'hooking' the events or polling them.

4.4 Raising events

Any code can raise a device event directly via `vrtaRaiseEvent()`, but normally it is only code within the device that raises its events. The `vrtaDevice` class provides a `RaiseEvent()` method that can be used by classes that inherit from it.

e.g.

```
//----------------------------------------------------
void mydev::NewValue(unsigned val)
//----------------------------------------------------
{
  m_Val = val;

  vrtaEvent event;
  ReadState(&event,1);
  RaiseEvent(event);
}
```

4.5 Raising interrupts

Interrupts get raised by sending action `ICU_ACTION_ID_Raise` to the ICU device using code like this:

```
//--------------------------------------------------------
void RaiseInterrupt(unsigned vector)
{
  vrtaAction      action;
  action.devID                 = ICU_DEVICE_ID;
  action.devAction             = ICU_ACTION_ID_Raise;
  action.devActionLen          =
        sizeof(action.devEmbeddedData.uVal);
  action.devEmbeddedData.uVal = vector;
  action.devActionData         = NULL;
  SendAction(ICU_DEVICE_ID, action);
}
```

The `vrtaDevice` class provides a `RaiseInterrupt()` method that can be used by classes that inherit from it.

e.g.

```
//--------------------------------------------------------
void mydev::NewValue(unsigned val)
//--------------------------------------------------------
{
  m_Val = val;
  if (val == m_Match) {
        vrtaEvent event;
        ReadState(&event,1);
        if (m_Vector) {
              RaiseInterrupt(m_Vector);
        }
        RaiseEvent(event);
  }
}
```

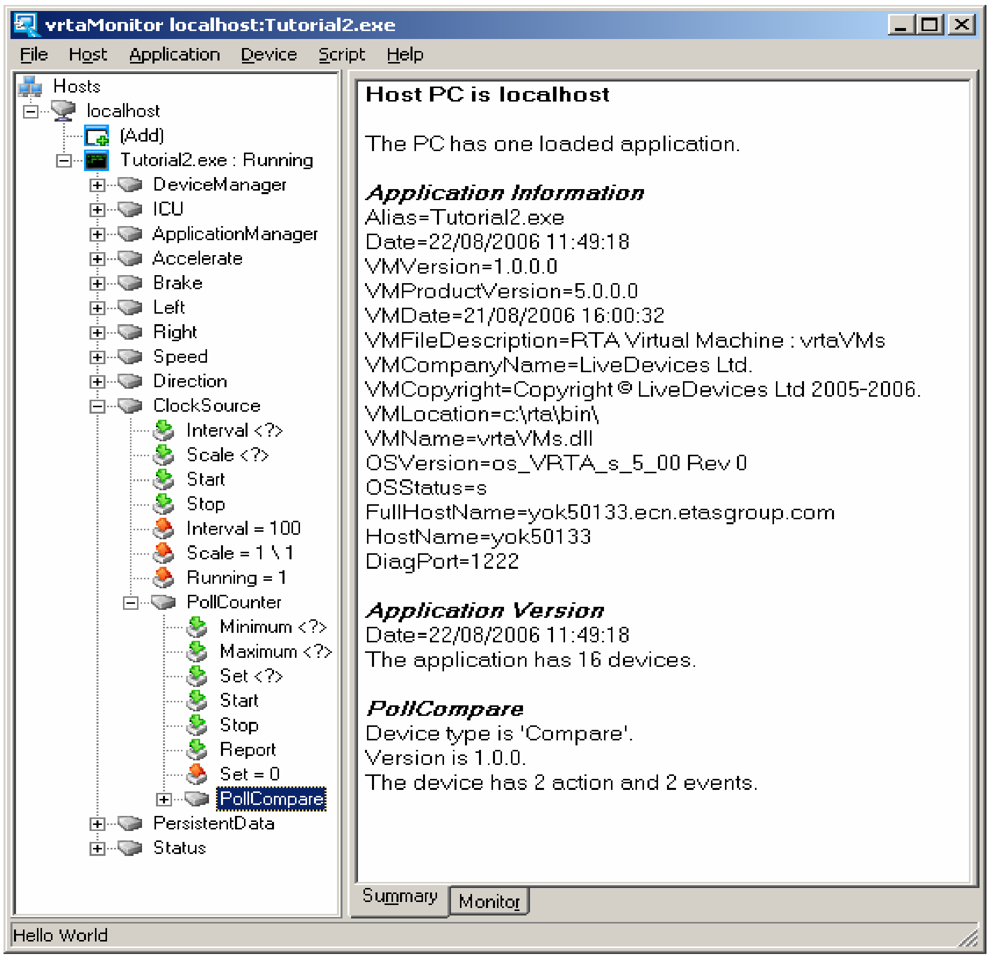
4.6 Parent / Child relationships

Sometimes you want to create a device that somehow ‘belongs to’ another device.

An example is the `vrtaCounter` device that ‘belongs’ to a `vrtaClock`.

You can tell external programs such as `vrtaMonitor` about this relationship by implementing an event called “_Parent”[26] which returns the device ID of the device that it belongs to. The program can then represent this relationship visually and will normally hide the _Parent event from view. This feature is demonstrated in the screenshot below – where `PollCompare`’s _Parent is set to `PollCounter` and `PollCounter`’s _Parent is set to `ClockSource`.

[26] By convention this is the highest numbered event in the device.



Host PC is localhost

The PC has one loaded application.

Application Information
Alias=Tutorial2.exe
Date=22/08/2006 11:49:18
VMVersion=1.0.0.0
VMProductVersion=5.0.0.0
VMDate=21/08/2006 16:00:32
VMFileDescription=RTA Virtual Machine : vrtaVMs
VMCompanyName=LiveDevices Ltd.
VMCopyright=Copyright © LiveDevices Ltd 2005-2006.
VMLocation=c:\rta\bin\
VMName=vrtaVMs.dll
OSVersion=os_VRTA_s_5_00 Rev 0
OSStatus=s
FullHostName=yok50133.ecn.etasgroup.com
HostName=yok50133
DiagPort=1222

Application Version
Date=22/08/2006 11:49:18
The application has 16 devices.

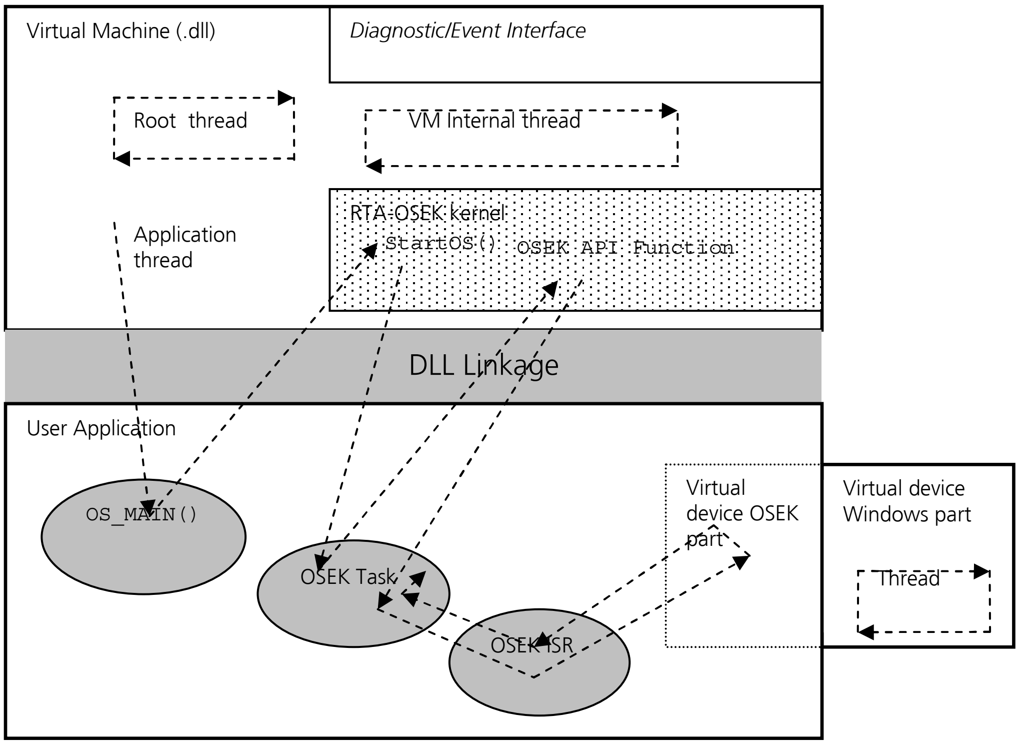
PollCompare
Device type is 'Compare'.
Version is 1.0.0.
The device has 2 action and 2 events.

4.7 Threads

A virtual device can spawn an *RTA-OSEK for PC* thread to perform operations asynchronously from the main application thread. Such threads can, with appropriate interlocks, access the device data and methods. They can cause events and interrupts to be raised.

The `vrtaDevice` class provides the `SpawnThread()` method that can be used by classes that inherit from it.

The diagram below illustrates how threads are used within a VECU. The application thread is the Windows thread that runs the VECU application code, including ISRs and RTA-OSEK tasks. This is the thread that calls `OS_MAIN()`. The root thread is the thread created by Windows when the VECU was loaded. This is the thread that executes `main()`. Virtual device drivers may be called by the application thread but may also contain private threads.



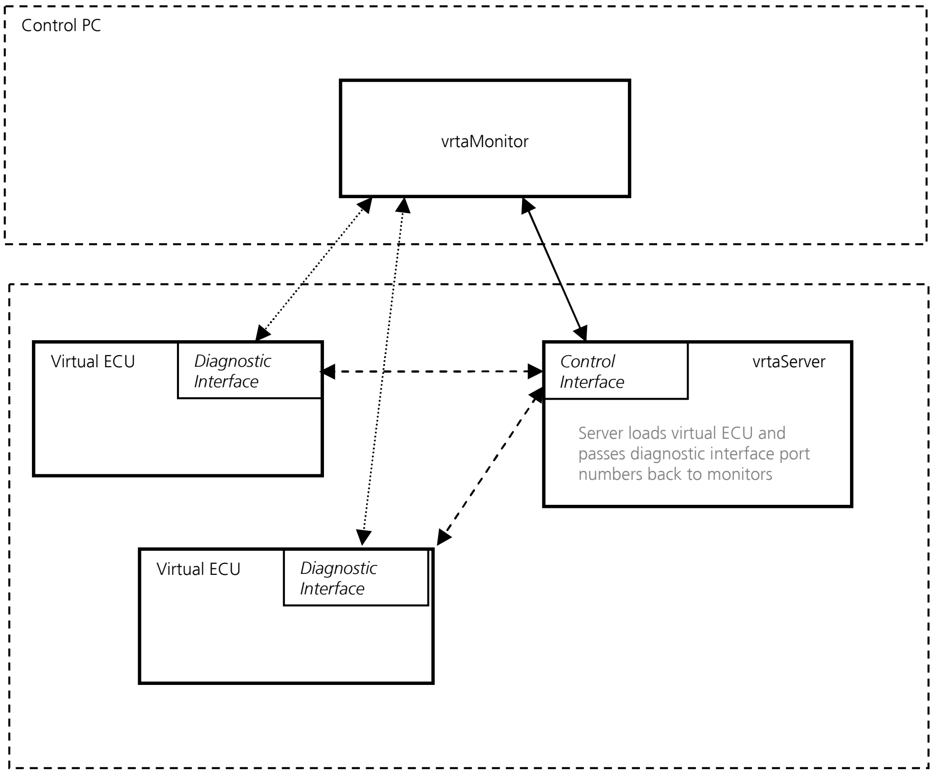
5 vrtaServer

`vrtaServer.exe` is a small program that runs unobtrusively on your PC coordinating the loading and locating of your Virtual ECUs. It normally runs as a Windows service.

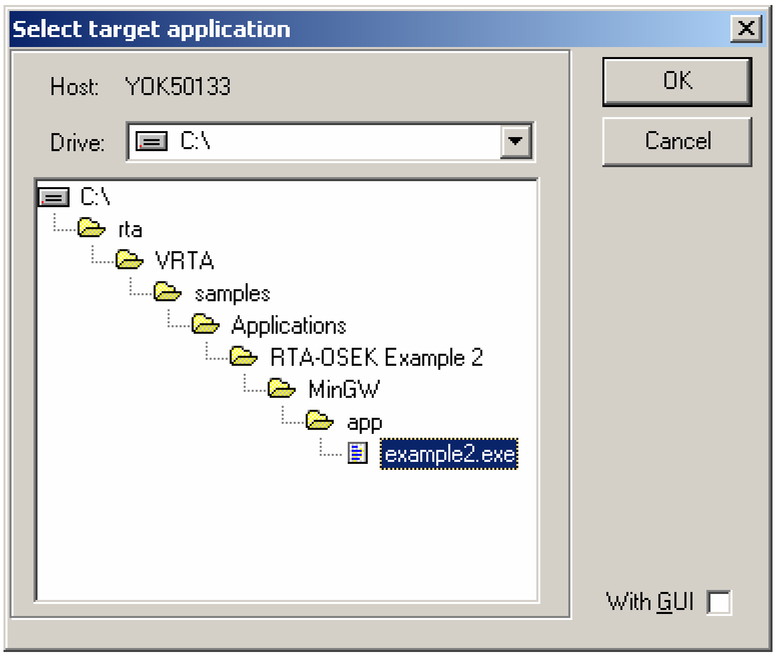
A VECU informs `vrtaServer` when it starts or terminates. External programs such as `vrtaMonitor` can then ask `vrtaServer` what VECUs are loaded, and attach to a VECU via its diagnostic interface.

An important point to note is that `vrtaMonitor` can be on a different PC to the server and its ECUs, so you can perform remote monitoring and control of a bank of test PCs.

The figure below shows a `vrtaMonitor` attached to two VECUs. When the VECUs load, they register with `vrtaServer` (dashed lines). `vrtaMonitor` then queries `vrtaServer` to find out what VECUs are loaded on the local machine and the TCP port numbers of their diagnostic interfaces (solid line). `vrtaMonitor` then communicates with the VECUs via their diagnostic interfaces (dotted lines).



Monitor programs also use `vrtaServer` to locate and load VECUs. This is necessary because the monitor may be running on a remote PC without access to files on the host PC. In the load dialog below, `vrtaMonitor` is accessing a remote PC, so the directory structure that you see reflects that on the remote PC.



5.1 Multiple instances of a VECU

Each VECU that loads is assigned a name (or alias) by `vrtaServer`. Normally this is just the file name of the VECU with the path information stripped off. This 'user-friendly' alias is the name that gets shown in a monitor program.

If the same VECU is loaded twice, or if an ECU with the same name but in a different directory is loaded, `vrtaServer` has to generate a different alias. Typically it will do this by adding _2, _3 etc. to the default alias.

`vrtaServer` keeps a count of the ECUs and monitors that know about the different aliases. It 'frees' an alias when no programs are using it.

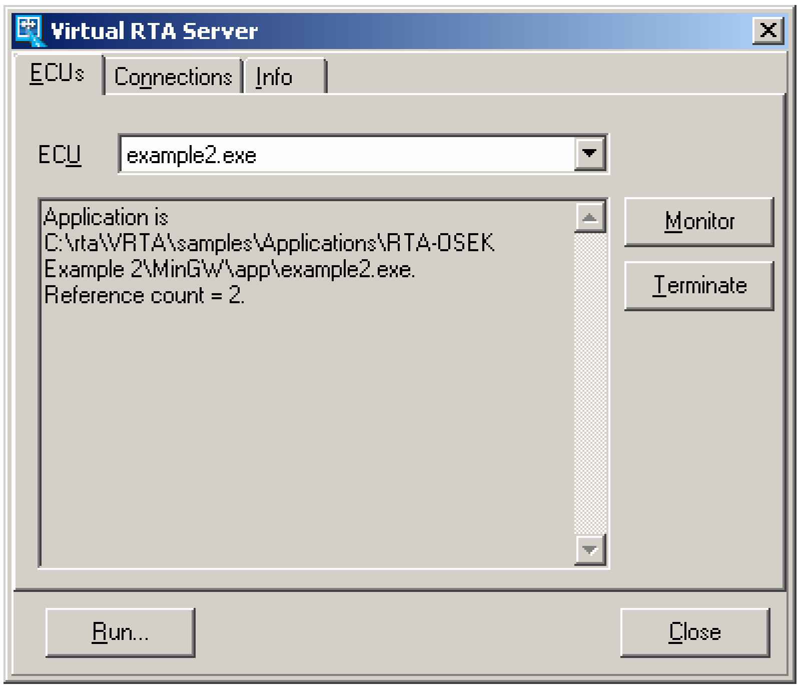
5.2 Server status information

`vrtaServer` normally runs as a Windows service under the 'SYSTEM' account. It therefore does not normally have any user-visible element. If it encounters problems, it logs them with the Windows Event Viewer.

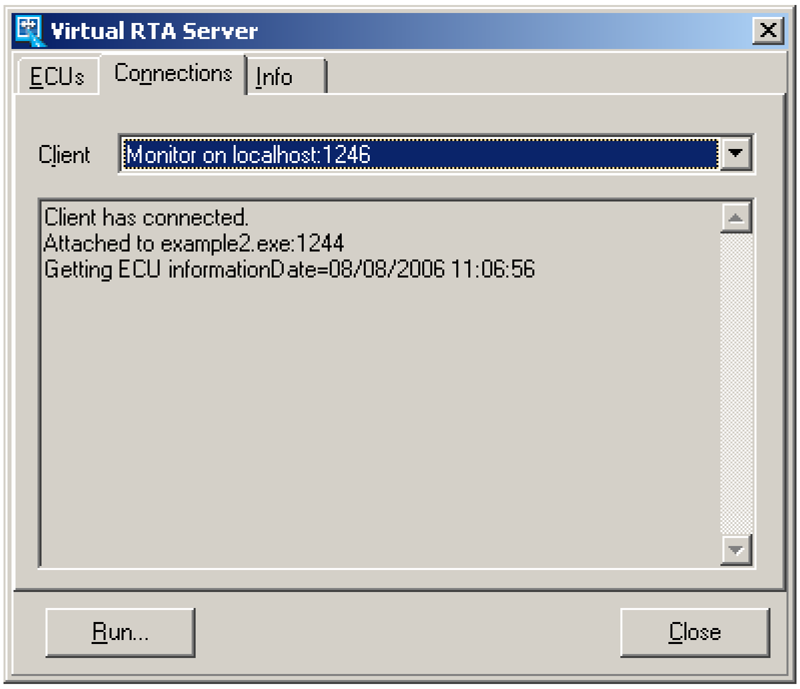
You can alternatively run `vrtaServer` as a Windows system-tray application. Close any VECUs or monitors on your PC then run the command '`vrtaServer -stop`' to stop the service[27]. Then run '`vrtaServer -standalone`'. You will see a tabbed dialog-style window appear along with a new icon in your system tray.

This dialog will give you information about the VECUs and monitors that are connected to `vrtaServer`.

[27] '`vrtaServer -start`' would start it again.



```
Application is
C:\rta\VRTA\samples\Applications\RTA-OSEK
Example 2\MinGW\app\example2.exe.
Reference count = 2.
```



```
Client has connected.
Attached to example2.exe:1244
Getting ECU informationDate=08/08/2006 11:06:56
```

Note that closing this dialog window does not cause `vrtaServer` to quit – it just minimizes back to the system tray. Select the **Close Server** menu item from the system tray icon to quit the server (or `'vrtaServer -stop'`).

5.3 Security issues

`vrtaServer` does not allow a monitor program to copy any programs or data to the host PC.

`vrtaServer` does not allow a remote user to modify files via the load dialog.

However as we have seen, `vrtaServer` allows remote monitor programs to launch VECUs on its host PC.

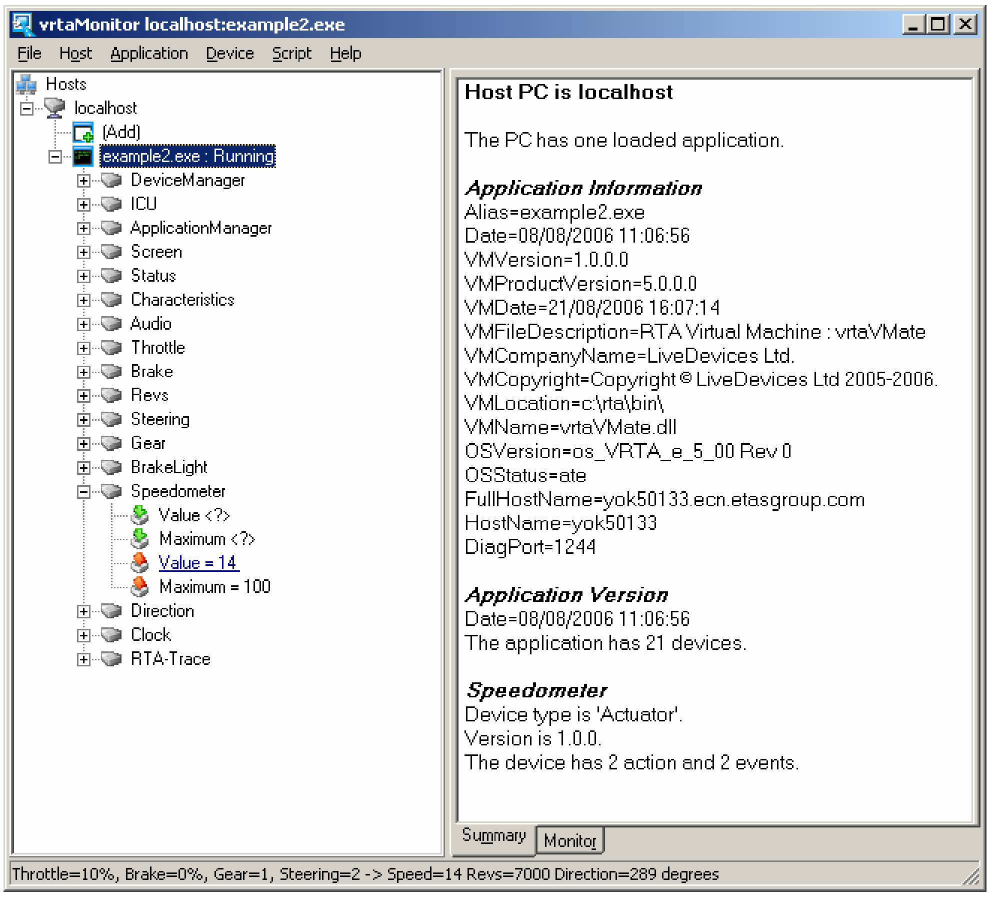
In a controlled test environment, this is probably the behavior that you want. However if you are worried about malicious abuse of this feature, you should configure your firewall to block external access to `vrtaServer`[28].

The default TCP ports used by `vrtaServer` (and RTA-TRACE) are `26000`, `31765` and `17185`.

[28] You could alternatively use the –p<n> command-line option to force `vrtaServer` to listen on a non-standard TCP port number. A casual visitor would find it difficult to guess what port to connect to.

6 vrtaMonitor

The program `vrtaMonitor.exe` can be used to inspect and control virtual ECUs on local and remote PCs.



Host PC is localhost

The PC has one loaded application.

Application Information
Alias=example2.exe
Date=08/08/2006 11:06:56
VMVersion=1.0.0.0
VMProductVersion=5.0.0.0
VMDate=21/08/2006 16:07:14
VMFileDescription=RTA Virtual Machine : vrtaVMate
VMCompanyName=LiveDevices Ltd.
VMCopyright=Copyright © LiveDevices Ltd 2005-2006.
VMLocation=c:\rta\bin\
VMName=vrtaVMate.dll
OSVersion=os_VRTA_e_5_00 Rev 0
OSStatus=ate
FullHostName=yok50133.ecn.etasgroup.com
HostName=yok50133
DiagPort=1244

Application Version
Date=08/08/2006 11:06:56
The application has 21 devices.

Speedometer
Device type is 'Actuator'.
Version is 1.0.0.
The device has 2 action and 2 events.

Throttle=10%, Brake=0%, Gear=1, Steering=2 -> Speed=14 Revs=7000 Direction=289 degrees

`vrtaMonitor` can connect to multiple PCs (Hosts), and multiple VECUs within each host.

You can interact with `vrtaMonitor` in a number or ways including:

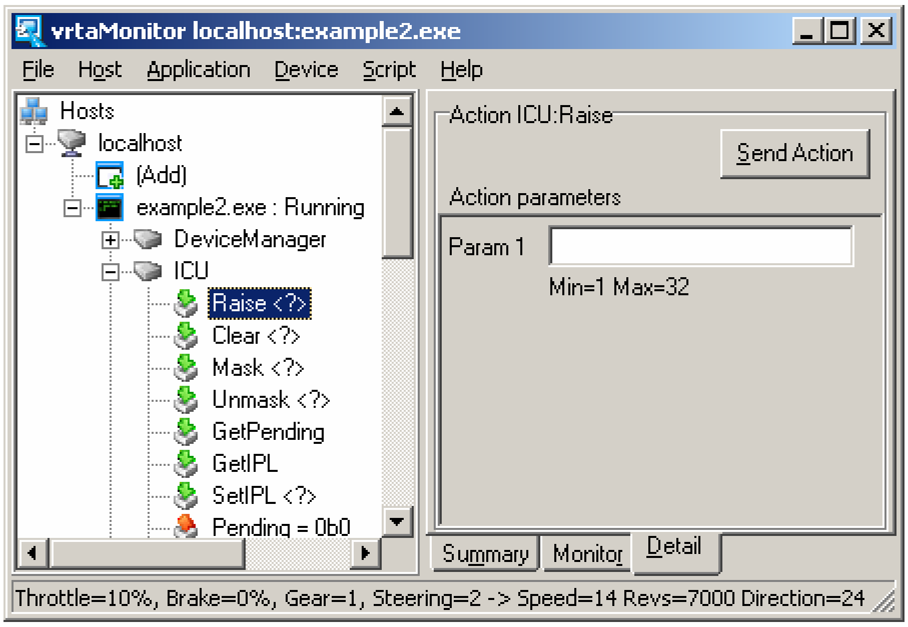
- The application's main menu
- Context menus (right-click on an element in the tree view)
- Shortcut keys (e.g. Ctrl+L to load an ECU)
- Double-clicking on an element in the tree view
- Pressing Enter on an element in the tree view

For example, you can connect to a different PC by right-clicking the **Hosts** element and selecting **Add Host...** from the context menu. You can do the same thing through the application's **File** menu.

6.1 Actions

You can send an action ( ) to a virtual device in the following ways.

- Double-click on an action in the tree view. If the action does not require any data then the action is sent immediately (e.g. **ApplicationManager / Pause**). If the action requires input data (e.g **ICU / Raise**) then `vrtaMonitor` asks you to enter it the first time round, then re-sends the same value on subsequent double-clicks. (If you want to change the data that gets sent, select the **Params...** option from the context or main **Device** menu or press Ctrl+Alt+S).
- Press Ctrl+S when an action is selected. This is the same as a double-click above.
- Right-click the action and select **Send** from the context menu.
- Select the main menu item **Device / Current Action / Send**.
- Go to the **Detail** tab on the right-hand side. You can enter data (where needed) and send it by pressing the **Send Action** button.



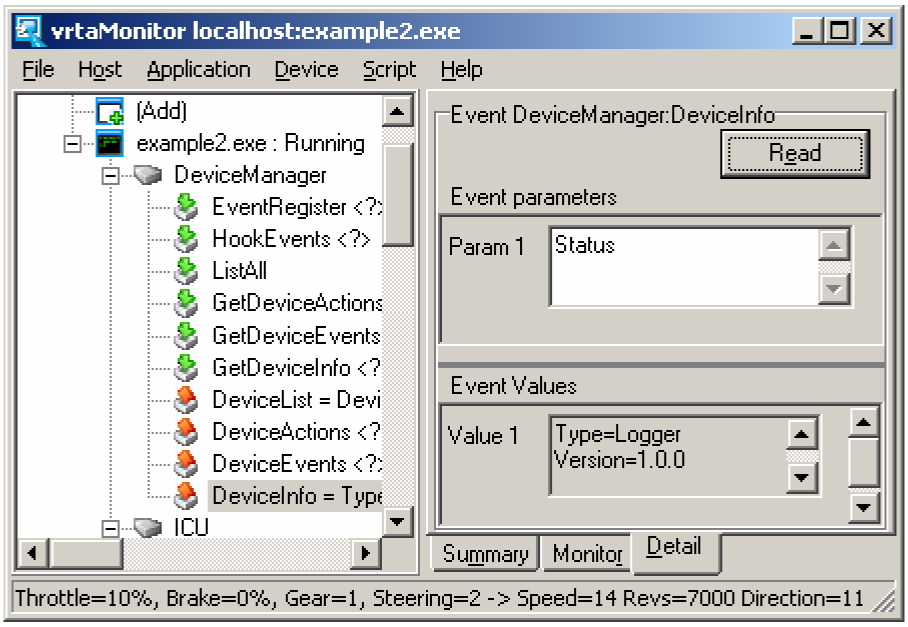
Throttle=10%, Brake=0%, Gear=1, Steering=2 -> Speed=14 Revs=7000 Direction=24

6.2 Events

6.2.1 Query

You can query the state of any event (●) in a virtual device in the following ways.

- Double-click on an event in the tree view. If the event does not require any data then the current value of the event is read immediately (e.g. **ICU / Pending**). If the event requires input data (e.g **DeviceManager /DeviceInfo**) then `vrtaMonitor` asks you to enter it the first time round, then re-sends the same value on subsequent double-clicks.
(If you want to change the data that gets sent, select the **Params...** option from the context or main **Device** menu or press Ctrl+Alt+R).
- Press Ctrl+R when an event is selected. This is the same as a double-click above.
- Right-click the action and select **Read** from the context menu.
- Select the main menu **item Device / Current Event / Read**.
- Go to the **Detail** tab on the right-hand side. You can enter data (where needed) and query the event by pressing the **Read** button.



Throttle=10%, Brake=0%, Gear=1, Steering=2 -> Speed=14 Revs=7000 Direction=11

It is also possible to ask `vrtaMonitor` to query all of the events of a device automatically every second or so and update the values displayed in the tree and detail views.

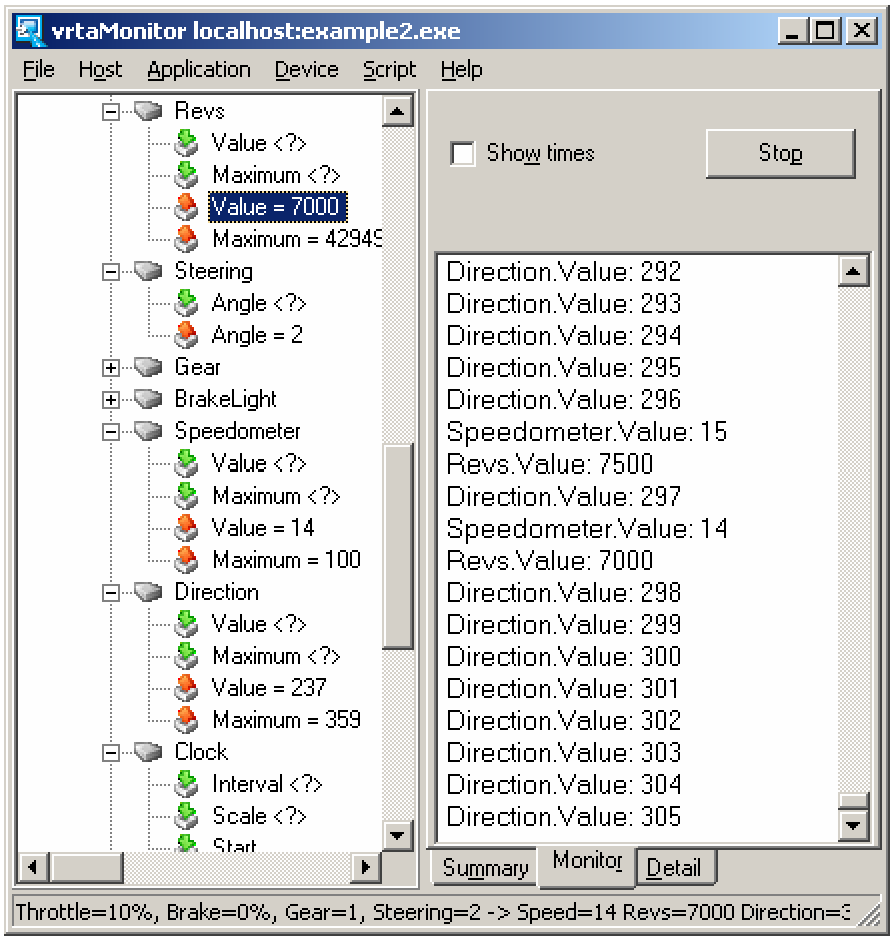
You do this by selecting **Auto Refresh** (Ctrl+A) for the device[29].

[29] A second Ctrl+A will turn auto-refresh off again.

6.2.2 Monitor

You can alternatively specify that you want to *monitor* an event rather than just querying it.

In this case, the VECU notifies `vrtaMonitor` whenever a monitored event is 'raised'[30]. All notifications are displayed in the monitor tab:


vrtaMonitor localhost:example2.exe
File Host Application Device Script Help
Show times
Stop
Summary Monitor Detail

```
Revs
  Value <?>
  Maximum <?>
  Value = 7000
  Maximum = 4294
Steering
  Angle <?>
  Angle = 2
Gear
BrakeLight
Speedometer
  Value <?>
  Maximum <?>
  Value = 14
  Maximum = 100
Direction
  Value <?>
  Maximum <?>
  Value = 237
  Maximum = 359
Clock
  Interval <?>
  Scale <?>
  Start
```

```
Direction.Value: 292
Direction.Value: 293
Direction.Value: 294
Direction.Value: 295
Direction.Value: 296
Speedometer.Value: 15
Revs.Value: 7500
Direction.Value: 297
Speedometer.Value: 14
Revs.Value: 7000
Direction.Value: 298
Direction.Value: 299
Direction.Value: 300
Direction.Value: 301
Direction.Value: 302
Direction.Value: 303
Direction.Value: 304
Direction.Value: 305
```

```
Throttle=10%, Brake=0%, Gear=1, Steering=2 -> Speed=14 Revs=7000 Direction=3
```

You can set up a monitor for an event using the normal application menu or context menu mechanisms, but the simplest way is just to drag the event from the tree view onto one of the right-hand side tab pages.

You can also drag a complete device across: this causes all of its events to be monitored.

To stop monitoring an event, just drag it (or its device) to the **Stop** button. Pressing the button on its own cancels all event monitors.

[30] Events are typically raised when some value in the virtual device changes.

6.2.3 Scripting using vrtaMonitor

The `vrtaMonitor` command-line options can be used to support a limited form of scripting capability for VECUs. (See section 15.3 for a complete list of command line options.)

The basic scripting operations include:

- Loading a VECU
- Attaching to an existing VECU
- Sending an action to a device
- Monitoring an event
- Pausing for a set amount of time
- Waiting for a termination condition

Scripting options can be entered directly on the `vrtaMonitor` command-line, but you will probably find it easier to use a command-file using the form `vrtaMonitor @commands.txt`.

A command file is a simple text file with one option per line. A line is treated as a comment if it starts with a semi-colon, forward-slash, space or tab character. Command-files can be nested up to 5 times.

The command-line options are documented in detail in chapter 15, but a few useful examples are presented here. They assume that you have built Tutorial2.exe and that you run '`vrtaMonitor @commands.txt`' from the directory `c:\Play\Tutorial2\MinGW\app`.

Enter the text below into `commands.txt`.

```
-k
-log=log.txt
-t1
Tutorial2.exe
```

The –k option tells `vrtaMonitor` to stop further processing of the script options if one of the following events occurs.

Event
Failure to connect to `vrtaServer`.
Cannot attach to an alias specified via `-alias`.
Cannot auto-load a specified Virtual ECU.
Closed as a result of a `-t` timeout.
Failed to load VECU.
Closed as a result of `-f`.
Failed to send an action or receive an event.

The –log option causes logging information to be written to the file log.txt.

The –t1 option tells `vrtaMonitor` to run for 1 second (after processing its command-line options) before quitting.

The `Tutorial2.exe` parameter tells `vrtaMonitor` to load and run the VECU `Tutorial2.exe` (without showing its embedded GUI).

By contrast, if you use the option –d *before* naming the VECU then the VECU will load (and its devices become accessible), but the application thread will not be started.

Similarly if you use the –g option then the VECU will show its embedded GUI:

```
-k
-log=log.txt
-t1
-d
-g
```

You can attach the monitor to a VECU that is already loaded by specifying its *alias*:

```
-k
-log=log.txt
-alias=Tutorial2.exe
```

You can send an action to a device. The commands below will attach to an existing VECU and then terminate it.

```
-k
-log=log.txt
-alias=Tutorial2.exe
-send=ApplicationManager.Terminate
```

You can also monitor events. The example below runs `Tutorial2.exe`, monitors event `PollCompare.Match` for 5000ms then terminates. The file `log.txt` contains the results from the monitor window.

```
-k
-log=log.txt
-t10
Tutorial2.exe
-mon=PollCompare.Match
-wait=5000
-send=ApplicationManager.Terminate
-quit
```

6.2.4 Plug-Ins

`vrtaMonitor` supports much more comprehensive scripting features by using plug-in scripting DLLs. These are intended to allow you to write scripts in high-level languages such a Java and Ruby. The *RTA-OSEK for PC* installer does not currently provide any scripting DLLs. They may become available at a later date as a purchased add-on.

7 Migration Guide

Assume that you have developed an application using *RTA-OSEK for PC* and now need to migrate the application to your real hardware.

We will assume that you have obtained a version of RTA-OSEK that will run on your target hardware. You also need a compatible compiler and some way to load the compiled code onto your hardware.

This chapter covers the migration issues that you should expect to face.

7.1 OIL file

Your VECU application is described by a combination of the C source code, build scripts and the *RTA-OSEK* project file. The project file contains the project information encoded in OSEK's OIL syntax and is called something like *myproject.oil*.

If you are migrating to a non *RTA-OSEK* system, then you will have to refer to its documentation to discover how to encode the extra information that you might need.

Migrating to an *RTA-OSEK* based implementation is relatively easy. The elements that you might have to modify in your OIL file are described next[31].

7.1.1 Target and variant

Although you can modify a target variant within the RTA-OSEK GUI, you cannot change the target type itself.

The target and variant are specified in the OIL file in lines like this:

```
//RTAOILCFG OS_TARGET "VirtualOsek";
//RTAOILCFG TYPE = "MinGW";
```

To change to a different target modify the target to something like this:,

```
//RTAOILCFG OS_TARGET "PIC18/IAR 16 task";
```

You can delete the variant line (`//RTAOILCFG TYPE`) because the variant will be set to the target's default variant when you next load the OIL file. You can use the GUI if you need to select a non-default variant.

7.1.2 Interrupts

RTA-OSEK for PC simulates an interrupt controller with 32 interrupt sources, each with one of 32 priorities and attached to a vector number 1 to 32. It allows you to decide in software how to map your (virtual) hardware to an interrupt source.

The interrupt controller in your target hardware will have similar capabilities, but will probably have different vectors and available priorities.

[31] At the time of writing, there is no automatic way to switch to different targets within the RTA-OSEK GUI, so some editing of the OIL file is unfortunately necessary. If this worries you, contact our support department and we may be able to perform this work for you.

Therefore you will need to assign new target-based vector numbers and priorities for each ISR in your application[32].

This has to be done by editing the ISR definitions in your OIL file, The priority goes in the ISR's `//RTAOILCFG PRIORITY` line, and the vector goes in the `//RTAOILCFG ADDRESS` line.

```
ISR isr1 {
         CATEGORY = 2;
         //RTAOILCFG PRIORITY = 22;
         //RTAOILCFG ADDRESS = 0x11;
         //RTAOILCFG OS_EXECUTION_BUDGET OS_UNDEFINED;
         //RTAOILCFG OS_BEHAVIOUR OS_SIMPLE;
         //RTAOILCFG OS_USES_FP FALSE;
         //RTAOILCFG OS_STACK {OS_UNDEFINED };
  };
```

As an option, you can simply delete the PRIORITY and ADDRESS lines for each ISR. When you next load the OIL file into the *RTA-OSEK* GUI they will appear as 'undefined' and you can set the priority and vector in the normal way.

7.1.3 Number of tasks

RTA-OSEK for PC ECUs can use up to 1024 tasks (1025 if you count the idle task). However there are very few ports of RTA-OSEK that support this many tasks because the run-time overhead is just too great. Most RTA-OSEK ports support 16 or 32 tasks. Although this seems to be a small, you will find that this is more than you need in most applications – particularly if you make use of features such as *RTA-OSEK* processes that allow you to pack multiple executable-code elements into tasks and ISRs.

If the RTA-OSEK port can handle fewer tasks than your VECU currently uses, you will have to do some re-engineering of your application.

7.2 Hardware drivers

In the VECU, most if not all of your hardware is simulated via virtual devices. These obviously need to be replaced in your real application.

By a fortuitous coincidence, the inability of C code to interact directly with the C++ code has already meant that you will have written some C / C++ interface functions such as the one below[33].

```
//----------------------------------------
int left_pressed(void)
{
        return Left.Value();
}
```

You simply provide different implementations for each of these interface functions so that they map onto your target hardware.

[32] Unless you were far-sighted enough to configure your VECU to use the same interrupt model as in your target ECU.
[33] Refer to the tutorial chapter for a fuller explanation.

e.g.

```
//---------------------------------------
int left_pressed(void)
{
        return inp(0x1001);
}
```

As long as you take care that each interface function has the same behavior as in the virtual device, this part of the migration should be straightforward.

7.3 Initialization

You will need to add code to your application to initialize your ECU's hardware. In particular you may need to configure the interrupt controller and any clock/compare devices. In effect you will need to provide a hardware specific version of the `InitializeDevices()` function used in *RTA-OSEK for PC* applications. You must refer to the user-guide for your hardware for the best way to do this.

7.4 Interrupts

Depending on your interrupt controller and the ECU hardware, you may need to add code to your ISRs that tells the hardware that the interrupt source has been serviced.

You can use RTA macros to implement conditional compilation of ISR code so that it can adapt to different platforms.

e.g.

```
ISR(isrAccel)
{
  change_speed(+1);

#ifdef OS_TARGET_VRTA
  // Nothing needed to clear interrupt for VRTA
#endif

#ifdef OS_TARGET_HC12X
  outp(0x12,99);
#endif
}
```

7.5 Building the application

If you want to build your application from within the *RTA-OSEK* GUI then you will have to modify the custom-build script that you are using. You should refer to the script that is supplied in the standard example program provided with the *RTA-OSEK* version for your target hardware.

8 RTA-TRACE Reference

Running RTA-TRACE on an *RTA-OSEK for PC* application is generally much easier than in other applications because the trace communications mechanism is fast, efficient and 'built-in'.

For most applications you simply enable RTA-TRACE support in the *RTA-OSEK* GUI, enable the trace comms link and call `CheckTraceOutput()` and `UploadTraceData()` regularly.

e.g.

```
OS_MAIN()
{
        StartOS(OSDEFAULTAPPMODE);

        while(!vrtaIsAppFinished()) {
#ifdef OSTRACE_ENABLED
                CheckTraceOutput();
                UploadTraceData();
#endif
                vrtaIsIdle(5);
        }

}
```

8.1 How it works

8.1.1 The VECU

When you build a trace-enabled VECU, code is added to your application to implement the trace communication APIs `TraceCommInit()`, `osTraceCommInitTarget()`, `UploadTraceData()` and `osTraceCommDataReady()` plus a virtual device named "RTA-Trace".

Whenever a block of trace data is ready to be sent, `UploadTraceData()` passes its address and size to the RTA-Trace device. The device then simply raises a 'Trace' event with this data attached.

The event can be 'hooked' by observers within the VECU or outside it (e.g. vrtaMonitor), and each of them will get a notification when the event is raised.

Thanks to the design of virtual devices, this mechanism is quick and efficient. Once the call to `RaiseEvent` returns, the trace buffer can resume being filled, so for most purposes emptying of the buffer appears to be instantaneous and an *RTA-OSEK for PC* application can generate accurate traces without being affected by 'communication-interval gaps' that affect other ports.

8.1.2 RTA-TRACE-Server

The *RTA-TRACE* communications driver `rtcVRTAlink.dll` adds the ability for *RTA-TRACE-Server* to communicate directly with a RTA-Trace device on a VECU.

When you select a `.rta` file for a VECU from within the *RTA-TRACE* GUI, this DLL checks to see if there is already a VECU running with the same path and name as the `.rta` file, but with a `.exe` extension. If so, it will attach to the VECU and hook the Trace event of its RTA-Trace device.

e.g. For `C:\Play\Tutorial2\MinGW\app\Tutorial2.rta` the VECU must be `C:\Play\Tutorial2\MinGW\app\Tutorial2.exe`.

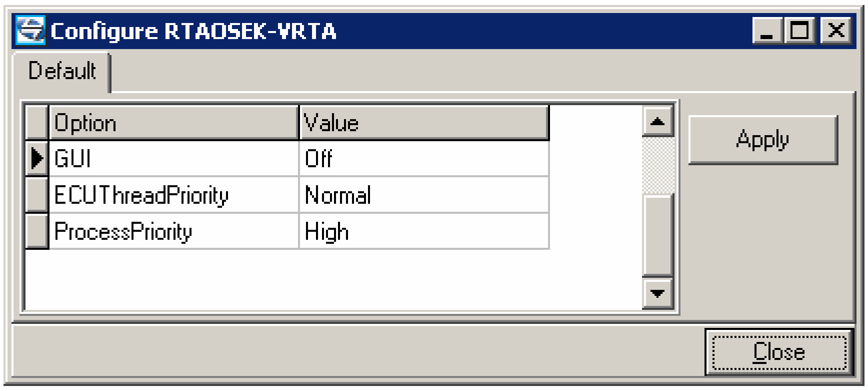
Important: The `.rta` file for a VECU must be in the same directory as the executable.

If the VECU is not already running, the DLL will load and start it before hooking the event.

8.2 Tuning process and thread priorities

The quality of the trace data that you see depends heavily on the interaction between different processes in your PC. If there are other processor-intensive applications running at the same time as tracing then you are likely to see irregularities in the trace that correspond to the moments where other applications are running[34].

You may find it useful to adjust the process or thread priority for the VECU for best results. This can be done via the RTAOSEK-VRTA configuration dialog that is accessible via *RTA-TRACE-Server*.



Option	Value
GUI	Off
ECUThreadPriority	Normal
ProcessPriority	High

The ECUThreadPriority value affects the priority of the application (OSEK) thread within the VECU.

The ProcessPriority value affects the priority of the complete VECU.

The astute reader will guess that the GUI value determined whether a VECU is started with its embedded GUI visible when started by `rtcVRTAlink.dll`.

[34] Note that this can include the RTA-TRACE GUI, which has to perform a very large amount of processing to keep up with the trace data being fed to it. You may find it better to run the RTA-TRACE GUI on a different PC to the one that is hosting the VECU.

8.3 Controlling the trace at run-time

The RTA-Trace virtual device has a few other tricks up its sleeve. In addition to its Trace event, it has 4 actions that you can use to affect the run-time trace behavior.

- **State**. This action can be sent the values Stop, FreeRunning, Bursting and Triggering. As long as your application is calling `CheckTraceOutput()` regularly, this action will cause the appropriate target API (`StopTrace()`, `StartFreeRunningTrace()`, `StartBurstingTrace()` or `StartTriggeringTrace()`) to get called.
- **Repeat**. This is sent On or Off to set the `SetTraceRepeat()` value in the VECU. Again, you must call `CheckTraceOutput()` regularly for this to be acted upon.
- **ECUThreadPriority**. This action can be used to change the application thread's priority in the same way as described in section 8.2.
- **ProcessPriority**. This action can be used to change the VCU's process priority in the same way as described in section 8.2.

8.4 Rolling your own

If you want to write the RTA-TRACE communications link yourself instead of using the inbuilt version, simply define the macro `OS_OVERRIDE_vrtaTraceDevice` when compiling `osgen.cpp`. This macro will prevent the trace communication APIs `TraceCommInit()`, `osTraceCommInitTarget()`, `UploadTraceData()` and `osTraceCommDataReady()` and the RTA-Trace device from being added.

9 Virtual Machine API Reference

This chapter gives a detailed description of the *RTA-OSEK for PC* Virtual Machine API calls, listed in alphabetical order.

9.1 General notes

9.1.1 API Header Files

The file `vrtaCore.h` must be included to use the API calls listed in this chapter. `vrtaCore.h` contains prototype declarations for the API calls described here. It also file #includes the files `vrtaTypes.h` and `vrtaVM.h`.

9.1.2 Linkage

Unless specified otherwise all Virtual Machine API calls use C linkage (i.e. no C++ name mangling) and so may be called from C or C++ source.

9.1.3 The API Call Template

Each API call is described in this chapter using the following standard format:

The title gives the name of the API call.

A brief description of the API call is provided.

Function declaration:

Interface in C syntax.

Parameters:

Parameter	Input/Output	Description
`Parameter Name`	Input/Output	Description.

Description:

Explanation of the functionality of the API call.

Return values:

Value	Description
`Return values.`	Description of return value.

Notes:

Usage restrictions and notes for the API call.

See also:

List of related API calls.

9.2 Common Data Types

These data types are all declared in `vrtaTypes.h`.

9.2.1 vrtaDevID

A scalar value that contains the ID of a virtual device. Devices 0, 1 and 2 are the standard VM devices DeviceManager, ICU and ApplicationManager respectively.

9.2.2 vrtaActionID

A scalar value that contains the ID of an action in a virtual device. Actions IDs start at 1.

9.2.3 vrtaEventID

A scalar value that contains the ID of an event in a virtual device. Events IDs start at 1.

9.2.4 vrtaActEvID

A scalar value that contains the ID of an action or event in a virtual Device.

9.2.5 vrtaISRID

A scalar value that contains the number of an ISR. ISRs range from 1 to 32, but a `vrtaISRID` can sometimes be set to zero to mean 'no ISR'.

9.2.6 vrtaIntPriority

A scalar value that contains an interrupt priority. Priorities from zero (no ISR) to 32.

9.2.7 vrtaErrType

This scalar value gets used as a status return type by many of the API functions. It can take one of the values:

• RTVECUErr_NONE	: No error / success
• RTVECUErr_Dev	: Device fault. Typically invalid device ID
• RTVECUErr_ID	: ID fault. Typically invalid action or event ID
• RTVECUErr_VAL	: Value fault. Typically value is out of range.

• RTVECUErr_Conn	: Connection fault. Occurs with remote monitor applications if the link to the VECU fails.

9.2.8 vrtamillisecond

A scalar representing an *interval* in milliseconds.

9.2.9 vrtaTimestamp

A scalar representing the *current ECU time* in milliseconds. This is normally the number of milliseconds since (just before) `OS_MAIN()` was called[35].

9.2.10 vrtaBoolean

The scalar type `vrtaBoolean` is used to represent boolean values. In this document a `vrtaBoolean` type is taken to be "true" when it is non-zero and "false" when it is zero.

9.2.11 vrtaByte

Represents a single byte of data passed into or out of a device. Normally part of an array of bytes.

9.2.12 vrtaDataLen

A scalar that represents the size (in bytes) of some data being passed into or out of a device. The maximum value that this can take is given by the value of the macro `vrtaDataLenMax`. (Currently 0xffff.)

9.2.13 vrtaEmbed

`vrtaEmbed` is a 'C' union containing the following fields:

• int	iVal
• unsigned	uVal
• double	dVal
• vrtaByte	bVal[16]

`vrtaEmbed` is used to support data passing operations in and out of virtual devices via `vrtaAction` and `vrtaEvent` (described in a moment). Both of these data structures embed an instance of `vrtaEmbed` within themselves.

Whenever the amount of data passed in or out of an action or event will fit inside an instance of `vrtaEmbed`, then the data *must be* passed in it.

[35] If a timestamp is taken before `OS_MAIN()`, then the time recorded is the number of milliseconds since the VM was loaded. This allows events to be timed where a VECU is started in slave mode.

In all common situations the data passed easily fits within the `vrtaEmbed` instance, so low-overhead code such as this is common:

```
thisEvent.devEmbeddedData.uVal = 32;
```

It is only where larger amounts of data need to be passed that we need to worry about allocating data buffers and data ownership issues.

9.2.14 vrtaAction

The `vrtaAction` structure is used to pass data value(s) to a specific action in a virtual device. The fields in `vrtaAction` are:

type and name	description
`vrtaDevID devID`	The ID of the device containing the action.
`vrtaActionID devAction`	The ID of the action.
`vrtaDataLen devActionLen`	The number of bytes of data. This can be zero. If its value is from 1 to 16 inclusive, then the data is present in the `devEmbeddedData` union. If it is more than 16 then `devEventData` contains the address of the data (see note below).
`const void * devActionData`	If `devEventLen` is from zero to 16 inclusive then `devEventData` must either be NULL or the address of `devEmbeddedData`. If there are more than 16 bytes of data then the storage for the input data must be provided by the creator of the `vrtaAction` and `devActionData` must point to this storage. Where `vrtaSendAction()` is called to send data to an action, any data referenced by `devActionData` must not change during the call.
`vrtaEmbed devEmbeddedData`	The union that contains the data where it is no larger than 16 bytes.
`vrtaTimestamp devTimeStamp`	This field is updated by the VM just before passing the action to the device.

9.2.15 vrtaEvent

The `vrtaEvent` structure is used to pass state information about a specific event in a virtual device. The fields in `vrtaEvent` are:

type and name	Description
`vrtaDevID devID`	The ID of the device containing the event.
`vrtaEventID devEvent`	The ID of the event.
`vrtaEventID devEventLen`	The number of bytes of data. This can be zero. If its value is from 1 to 16 inclusive, then the data is present in the `devEmbeddedData` union. If it is more than 16 then `devEventData` contains the address of the data (see note below).
`const void * devEventData`	If `devEventLen` is from zero to 16 inclusive then `devEventData` must either be NULL or the address of `devEmbeddedData`. If there are more than 16 bytes of data then the storage for the input data must be provided by the creator of the `vrtaEvent` and `devEventData` must point to this storage. See the note below on data ownership.
`vrtaEmbed devEmbeddedData`	The union that contains the data where it is no larger than 16 bytes.
`vrtaTimestamp devTimeStamp`	This field is updated by the VM to show one of: • The time that a query was made into `vrtaGetState()`. • The time that the data was passed to `vrtaRaiseEvent()`.

Ownership of `devEventData:`

- Where the current value of an event is being queried, data can be passed *into* the event via `vrtaGetState()`. The caller of `vrtaGetState()` ensures that any data referenced by `devEventData` does not change during the call.
- Where the current value of an event is being queried via `vrtaGetState()`, the data that is passed *out of* the event via `devEventData` may not change from the time that the call returns up to <u>the next time that `vrtaGetState()` is called from the same thread</u>. This can clearly be very complicated to achieve. However this is almost never necessary in real applications. Most return data fits within

`devEmbeddedData`, and in most other cases the data does not change anyway[36].

- Where an event is being raised via `vrtaRaiseEvent()`, the device may supply data to its listeners. Listeners must take a copy of any data that they need, so the caller of `vrtaRaiseEvent()` only has to ensure that the data does not change during the call.

9.2.16 vrtaTextPtr

A pointer to a simple ASCIIZ string.

e.g.

```
vrtaTextPtr tp = "Hello World";
```

9.2.17 vrtaStringlistPtr

A pointer to an ASCIIZ string that comprises zero or more \n separated list items.

e.g.

```
vrtaStringlistPtr lp = "One\nTwo\nThree";
```

9.2.18 vrtaOptStringlistPtr

A pointer to an ASCIIZ string that comprises zero or more \n separated option items each with the form <name>=<value>.

e.g.

```
vrtaOptStringlistPtr op = "Name=Bill\nAge=51\nWife=
Melinda";
```

[36] For example the DeviceEvents event in the DeviceManager often returns string data that is larger than 16 bytes. The strings that it returns are all allocated during initialization and do not change during the application, so no special protection is necessary.

9.3 Data format Strings

9.3.1 Overview

Virtual device actions and events commonly have some data associated with them.

For example the ICU's Raise action has to be passed an integer in the range 1 to 32. The ApplicationManager's State event supplies a value that represents the application thread state (Loaded|Running|Paused|Terminating|Resetting).

Within the VECU, all data is handled in native machine format, i.e. an 'int' for an integer value. It is your responsibility in VECU code to send data of the correct type between actions and events. As long as you trust your code to pass the right type of data, you can choose to omit range checks within your devices[37].

While this is a reasonable assumption in the C and C++ code that gets compiled into the VECU, it clearly does not hold where external programs such as `vrtaMonitor` access the data. For this reason a device must supply a description of the data that is used by its actions and events. This information is used by remote programs to format the data sent to a device and interpret the data it sends back. The VM performs size and range checking on data from remote programs so that it does not have to be done by each device itself.

Example format strings are:

String	Description
%d	32-bit signed integer
%d:;1;32	Integer that can take values 1 through 32.
%b	32-bit integer, normally represented in binary by a monitor program.
Loaded|Running|Paused|Terminating|Resetting	32-bit value with values zero through 4 that is normally represented by a monitor program as one of the | separated strings
%d,%d,%d	A structure comprising 3 32-bit signed integers.

[37] By all means add range checking code if you wish. Our design aim is to allow (but not force) devices to have a very small run-time overhead.

9.3.2 Definition

A data format string consists of one or more data-item descriptors. If there are multiple data-item descriptors then they are separated by `,`'s. The data-item descriptors are as follows (text inside " `[]` " is optional):

Data-item descriptor	Description
`%d[:<cons>]`	The data-item is a signed number. By default this is stored in 32 bits (a C `int`). The range is determined by the data-item size unless there is a constraint.
`%u[:<cons>]`	The data-item is an unsigned number. By default this is stored in 32 bits (a C `unsigned`). The range is determined by the data-item size unless there is a constraint.
`%f[:<cons>]`	The data-item is floating-point number. By default this is stored in 64 bits (a C `double`). The range is determined by the data-item size unless there is a constraint.
`%x[:<cons>]`	The data-item is an unsigned number that should represented in hexidecimal. By default this is stored in 32 bits (a C `unsigned`). The range is determined by the data-item size unless there is a constraint.
`%b[:<cons>]`	The data-item is an unsigned number that should represented in binary. By default this is stored in 32 bits (a C `unsigned`). The range is determined by the data-item size unless there is a constraint.
`<a>|<b>|…[:<cons>]`	The data-item is an unsigned number that should represented as a series of enumeration values. Enumeration value `<a>` corresponds to the number 0, enumeration value `<b>` to the number 1, and so on. By default this is stored in 32 bits (a C `unsigned`).
`%s[:<size>]`	The data-item is an ASCII string – which may or may not have a trailing '`\0`'. If no `<size>` value is given then the size of the string is inferred from the length of the action or event data. If a `<size>` value is given then it specifies the size of the string (including a trailing '`\0`' if there is one).
`%a[:<size>]`	The data-item is an array of bytes. If no `<size>` value is given then the size of the array is inferred from the length of the action or event data. If a `<size>` value is given then it specifies the size of the array.

A data-item descriptor may optionally include a constraint `<cons>`. A constraint has the form: `[<bits>[!<width>]][;<min>;<max>]`. Where:

`<bits>`	is the number of bits used to store a numeric value. This can be 8, 16, 32 or 64.

`<width>`	is the number of bytes between the start of the data-item and the start of the next data-item.
`<min>` and `<max>`	are the minimum and maximum values that may be stored in a numeric data-item.

For example:

`"%d"`

There is a single signed number that will be stored in 32 bits (a C `int`).

`"%u:;1;10"`

There is an unsigned number in the range 1 to 10 inclusive that will be stored in 32 bits (a C `unsigned`).

`"%x:16,%b:8"`

There is an unsigned number that will be stored in 16 bits (a C `unsigned short`) and should be displayed in hexidecimal. This is immediately followed by an unsigned number that will be stored in 8 bits (a C `unsigned char`) and should be displayed in binary.

`"%s:10,%u:8!4,%u:64;1;100"`

There is a 10 character string. This is followed by an unsigned number stored in 8 bits. There are then 3 bytes of padding; since the next data-item is stored 4 bytes after the start of the 8 bit value. This padding is followed by an unsigned number in the range 1 to 100 inclusive stored in 64 bits

9.4 API Functions

9.4.1 InitializeDevices()

Device initialization hook function.

Function declaration:

```
void InitializeDevices(void)
```

Parameters:

Parameter	Input/Output	Description
`<none>`		

Description:

You must provide the `InitializeDevices()` hook function in your application code. It is called by the application thread immediately before it calls `OS_MAIN()`. `InitializeDevices()` is normally used to execute code that carries out initialization of virtual devices. By the time that `InitializeDevices()` is called, each virtual device will have been sent a Reset action to inform it that the application thread is about to start.

Return values:

Value	Description
`<none>`	

Notes:

This function has C++ linkage and so must be implemented in a C++ source module, typically `devices.cpp`.

See also:

`OS_MAIN()`, `vrtaStart()`

9.4.2 OS_MAIN()

The entry-point for the application thread.

Function declaration:

```
OS_MAIN()
```

Parameters:

Parameter	Input/Output	Description
`<none>`		

Description:

The `OS_MAIN()` function is provided by the Virtual ECU application code and is the entry-point for the application thread.

Typically an application will make some initialization calls and then start the OS kernel via `StartOS()`. On the return from `StartOS()` the code that executes is deemed to be the 'idle task'. That is, it is the code that runs when no task or ISR is active.

Normally an OSEK application does not 'return' from `OS_MAIN()` because this has no meaning in a typical embedded system. Sometimes it might call `ShutdownOS()`, which has implementation-dependent behavior.

For *RTA-OSEK for PC*, the application can return from `OS_MAIN()` or call `ShutdownOS()` to cause the application to finish. This might be because the ECU simulation is complete for example.

If the Virtual ECU has been loaded in autostart mode (the default) then the complete Virtual Machine will terminate automatically.

If, however, the VECU was been loaded in slave mode then the Virtual Machine will wait for a Terminate action to be received by the Application Manager. This allows the state of the VECU's devices to be queried after the application thread has terminated.

Return values:

Value	Description
`<none>`	

Notes:

`OS_MAIN()` normally has the structure shown below in an *RTA-OSEK for PC* application. The call to `vrtaIsAppFinished()` should be made so that external requests to terminate the program get recognized promptly. If you don't quit `OS_MAIN()` when such a request occurs then the VM will forcibly terminate the application thread after a few seconds.

```
OS_MAIN()
{
        initialize_something();
        StartOS(OSDEFAULTAPPMODE);

        while(!vrtaIsAppFinished()) {
                vrtaIsIdle(5);
        }

        ShutdownOS(E_OK);
}
```

See also:

`InitializeDevices()`, `vrtaStart()`,`vrtaIsAppFinished()`, `vrtaIsIdle()`

9.4.3 vrtaEnterUninterruptibleSection()

Enter a critical section that cannot be interrupted.

Function declaration:

```
void vrtaEnterUninterruptibleSection(void)
```

Parameters:

Parameter	Input/Output	Description
`<none>`		

Description:

This function enters a critical section. Only one thread at a time may be in the critical section. Calling `vrtaEnterUninterruptibleSection()` will block the calling thread if another thread is already in the critical section.

If the application thread calls `vrtaEnterUninterruptibleSection()` then it cannot be interrupted until it leaves the critical section.

Important: if the application thread needs to call any Windows API function or non-reentrant C/C++ runtime library function then it must call `vrtaEnterUninterruptibleSection()` before making the call and `vrtaLeaveUninterruptibleSection()` afterwards. Windows API functions and non-reentrant C/C++ runtime library functions cannot cope with the stack manipulation that occurs when an *RTA-OSEK for PC* interrupt executes.

Return values:

Value	Description
`<none>`	

Notes:

See also:

`vrtaLeaveUninterruptibleSection()`

9.4.4 vrtaEventRegister()

Register an event handler.

Function declaration:

```
vrtaEventListener vrtaEventRegister(
    vrtaEventCallback eCallback, const void *tag)
```

Parameters:

Parameter	Input/Output	Description
`eCallback`	Input	A pointer to an event handling function.
`tag`	Input	A caller provided value that will be passed as an argument to `eCallback`.

Description:

This API call registers an event hook callback routine with the VM. The `vrtaEventListener` handle is needed when calling `vrtaHookEvent()` so that it can identify `eCallback` as the function to call when the specified event is raised.

`eCallback` is of type `vrtaEventCallback` which is defined as follows:

```
typedef vrtaErrType (*vrtaEventCallback)(
    const void *instance, const vrtaEvent *event);
```

When `eCallback` is called its `instance` argument will be set to the `tag` argument passed to `vrtaEventRegister()` and its `event` argument will contain the event raised[38]. `eCallback` should always return `RTVECUErr_NONE`.

The content of the `vrtaEvent` structure pointed to by `event` is only valid for the duration of the call to `eCallback`. If you need to use this data after `eCallback` has returned then you must take a copy of the data.

Return values:

Value	Description
`<a handle>`	A handle for the event handler.

Notes:

The event hook callback function gets called during the execution of `vrtaRaiseEvent()`. Your function must be thread-safe because it is quite normal for devices to raise events from threads that are independent of the application thread. Any event that gets provoked from an external monitor application *will* be in a different thread.

See also:

`vrtaEventUnregister()`, `vrtaHookEvent()`, `vrtaRaiseEvent()`

[38] See the description of `vrtaGetState()` for more information about the `vrtaEvent` type.

9.4.5 vrtaEventUnregister()

Unregister an event handler.

Function declaration:

```
vrtaErrType vrtaEventUnregister(
    vrtaEventListener listener)
```

Parameters:

Parameter	Input/Output	Description
`listener`	Input	An event-handler handle returned by `vrtaEventRegister()`.

Description:

This API call unregisters an event handler previously registered with `vrtaEventRegister()`. Any events that have been hooked by the event handler are unhooked.

Return values:

Value	Description
`RTVECUErr_NONE`	The API call was successful.
`RTVECUErr_VAL`	The `listener` argument is invalid.

Notes:

This API cannot be called from within an event handler.

See also:

`vrtaEventRegister()`, `vrtaHookEvent()`, `vrtaRaiseEvent()`

9.4.6 vrtaGetState()

Query the current state (value) of an event.

Function declaration:

```
vrtaErrType vrtaGetState(vrtaDevID id, vrtaEvent *ev)
```

Parameters:

Parameter	Input/Output	Description
`id`	Input	The ID of the virtual device to be queried.
`ev`	Input/Output	A pointer to the structure that specifies the event and its data.

Description:

This API is used to obtain the current value of an event supported by a virtual device. Virtual devices raise events at appropriate times and these can be multicast to interested receivers. However, sometimes it is useful to be able to discover the "current value" of an event. This makes most sense for events that contain data. For example, one may wish to discover the current interrupt priority (IPL) level of the Virtual Machine's interrupt control unit rather than waiting for an event to be raised when the IPL changes. Events which do not contain data but simply indicate that something has happened can still be queried, but it is not really useful so to do.

When calling `vrtaGetState()`, you must set the correct device and event IDs in `ev`. If the event needs to be passed some data as part of the query (e.g. the name of the device for the DeviceManager's DeviceAction event), then the data must be set up before the call. If no data is needed, set the `devEventLen` field to zero.

On successful return from `vrtaGetState()`, the data in `ev` now references the current value of the event.

Return values:

Value	Description
`RTVECUErr_NONE`	The API call was successful.
`RTVECUErr_Dev`	The specified device ID is invalid.
`RTVECUErr_ID`	The specified event ID is invalid.
`RTVECUErr_VAL`	The data provided in the event is invalid (out of range?).

Notes:

During the call of `vrtaGetState()` the VM passes the `vrtaEvent` structure to the device. The device determines the event's value (possibly using the input data) and either copies the event data into the `devEmbeddedData` field or places the data in storage that it has allocated and sets the `devEventData` field to point to this storage. The queried device is responsible for managing any storage that it allocates. Refer to section 9.2.15 for details.

See also:

`vrtaRegisterVirtualDevice()`, `vrtaRaiseEvent()`,
`vrtaEventRegister()`, `vrtaEventUnregister()`,
`vrtaHookEvent()`

9.4.7 vrtaHookEvent()

Hook or unhook an event so that an event handler is or is not called when the event is raised.

Function declaration:

```
vrtaErrType vrtaHookEvent(vrtaEventListener listener,
    vrtaDevID dev, vrtaEventID ev, vrtaBoolean capture)
```

Parameters:

Parameter	Input/Output	Description
`listener`	Input	An event-handler handle returned by `vrtaEventRegister()`.
`dev`	Input	The ID of the device.
`ev`	Input	The ID of the event.
`capture`	Input	True to hook an event, false to unhook an event.

Description:

If `capture` is true then this call 'hooks' the event so that the event handler associated with `listener` is called when the event gets raised.

If `capture` is false then this call unhooks one or more events previously hooked with this API call. The behavior of the call when `capture` is false depends on the values of `dev` and `ev` as follows:

`dev`	`ev`	Result
0	0	All event hooks are removed from `listener`.
Non-zero	0	All event hooks for events owned by the specified device are removed from `listener`.
Non-zero	Non-zero	The event hook for the specified event is removed from `listener`.

Return values:

Value	Description
`RTVECUErr_NONE`	The API call was successful.
`RTVECUErr_Dev`	The specified device ID is invalid.
`RTVECUErr_ID`	The specified event ID is invalid.
`RTVECUErr_VAL`	The `listener` argument is invalid or called from inside an event handler.

Notes:

This API may not be called from inside an event handler.

See also:

`vrtaEventRegister()`, `vrtaEventUnregister()`, `vrtaRaiseEvent()`

9.4.8 vrtaInitialize()

Initialize the Virtual Machine.

Function declaration:

```
void vrtaInitialize(int argc,
    char* argv[], const vrtaVectorTable* vecTable)
```

Parameters:

Parameter	Input/Output	Description
`argc`	Input	The number of command line arguments on the Virtual ECU's command line.
`argv`	Input	The array of command line arguments from the Virtual ECU's command line.
`vecTable`	Input	A pointer to the interrupt vector table.

Description:

This API call is used to initialize the Virtual Machine – it must be called after `vrtaLoadVM()` and before `vrtaStart()`.

For an RTA-OSEK VECU, this API is called automatically for you.

The `argc` and `argv` arguments should be taken directly from the `argc` and `argv` arguments of the Virtual ECU's `main()` entry-point.

`vectTable` points to a `vrtaVectorTable` structure containing the interrupt vector table. `vrtaVectorTable` is defined as follows:

```
#define RTVECU_NUM_VECTORS   (32)
typedef struct {
    unsigned      numVectors;
    vrtaIntVector vectors[RTVECU_NUM_VECTORS];
} vrtaVectorTable;
```

The `numVectors` field must be 32. This is checked by the Virtual Machine during `vrtaInitialize()` and a fatal error generated if the value is not 32. `vectors[]` is an array of 32 interrupt vectors. The Virtual Machine's interrupt control unit (ICU) numbers interrupt vectors from 1 to 32 (0 is used to mean no interrupt). ICU interrupt vector number 1 corresponds to `vectors[0]`, ICU interrupt vector number 2 corresponds to `vectors[1]`, and so on up to ICU interrupt vector number 32 which corresponds to `vectors[31]`.

Each interrupt vector is defined as follows:

```
typedef struct {
    vrtaIntHandler  handler;
    vrtaIntPriority priority;
    vrtaAppTag      tag;
} vrtaIntVector;
```

The `handler` field points to the interrupt handler to be run when the corresponding interrupt arrives. `priority` is the priority of the corresponding interrupt – this must be a number in the range 1 to 32 inclusive (1 is the lowest priority and 32 is the highest priority). `tag` is application data that that is passed to the interrupt handler when it is called.

An interrupt handler has the following definition:

```
typedef void (*vrtaIntHandler)(vrtaAppTag tag,
    vrtaIntPriority oldIPL);
```

When an interrupt handler is called its `tag` argument is set to the `tag` argument in the corresponding interrupt vector and its `oldIPL` argument is set to the priority of the interrupted code – zero for code not running in an interrupt handler or the priority of the interrupt for code running in an interrupt handler.

A trivial example of starting a Virtual ECU might look like:

```
void IntHandler(vrtaAppTag tag,
                vrtaIntPriority oldIPL) {
    /* Handle interrupt. */
}

vrtaVectorTable IntVectors = {
    RTVECU_NUM_VECTORS,
    {{IntHandler, 1, (vrtaTag) 1},
     {IntHandler, 2, (vrtaTag) 2},
     < ...snip... >
     {IntHandler, 32, (vrtaTag) 32}}
};

OS_MAIN() {
    /* The application thread starts here. */
     < ...snip... >
}

void main(int argc, char * argv[]) {
    vrtaLoadVM();
    vrtaInitialize(argc, argv, &IntVectors);
    vrtaStart();
    /* Control returns here when the application
     * thread terminates. */
}
```

Return values:

Value	Description
`<none>`	

Notes:

Important: if a Virtual ECU is using OSEK this API should not be called explicitly as it is called by the `main()` entry-point in the support file `vrtaOSEKSupp.cpp` that is `#included` via `osgen.cpp`.

See also:

`vrtaLoadVM()`, `vrtaStart()`

9.4.9 vrtaInitializeOS()

Initialize the OS kernel within the Virtual Machine DLL.

Function declaration:

```
void vrtaInitializeOS(
    vrtaLinkageTablePtr * osLinkageTablePtrPtr,
    vrtaLinkageTablePtr * traceLinkageTablePtrPtr)
```

Parameters:

Not documented.

Description:

This API call is used from inside RTA-OSEK support files to initialize the OSEK kernel within the Virtual Machine DLL. **This API call is not for application use.**

Return values:

Not documented.

Notes:

See also:

9.4.10 vrtaIsAppFinished()

Determine if the application thread has terminated or is about to terminate.

Function declaration:

```
vrtaBoolean vrtaIsAppFinished(void)
```

Parameters:

Parameter	Input/Output	Description
`<none>`		

Description:

This API call returns true if the application thread has terminated or is about to terminate, or false otherwise. This API call may be used by the application thread to discover if it is about to be forcibly terminated – e.g. because another thread has called `vrtaTerminate()` or a Terminate action has been sent to the ApplicationManager Device. This API may also be used in *RTA-OSEK for PC* threads to discover if they should terminate themselves.

Return values:

Value	Description
`true`	The application thread has terminated or is about to terminate.
`false`	The application thread has not terminated and is not about to terminate.

Notes:

See also:

`vrtaSpawnThread()`

9.4.11 vrtaIsAppThread()

Determine if the calling thread is the application thread.

Function declaration:

```
vrtaBoolean vrtaIsAppThread(void)
```

Parameters:

Parameter	Input/Output	Description
`<none>`		

Description:

This API call returns true if the calling thread is the application thread or false if the calling thread is not the application thread.

Return values:

Value	Description
`true`	The calling thread is the application thread.
`false`	The calling thread is not the application thread.

Notes:

See also:

9.4.12 vrtaIsIdle()

Yield the processor whilst idle.

Function declaration:

`void vrtaIsIdle(vrtamillisecond msecs)`

Parameters:

Parameter	Input/Output	Description
`msecs`	Input	The number for milliseconds to yield for.

Description:

This API call tells the Virtual Machine that the calling thread will be idle for the specified number of milliseconds. Ideally a thread that is idle should call this API rather than busy-waiting. Doing so allows the VM to suspend the thread so that Windows can use the processor to run another thread.

The application thread will still respond to interrupts and run the corresponding ISRs while inside a call of `vrtaIsIdle()`. For example, if at time `t` the application thread calls `vrtaIsIdle(100)` and at time `t+10` an interrupt arrives, the corresponding ISR will be run by the application thread at, or shortly after, `t+10`.

Return values:

Value	Description
`<none>`	

Notes:

See also:

9.4.13 vrtaLeaveUninterruptibleSection()

Leave a critical section.

Function declaration:

`void vrtaLeaveUninterruptibleSection(void)`

Parameters:

Parameter	Input/Output	Description
`<none>`		

Description:

This function leaves a critical section previously entered by calling `vrtaEnterUninterruptibleSection()`.

Return values:

Value	Description
`<none>`	

Notes:

See also:

`vrtaEnterUninterruptibleSection()`

9.4.14 vrtaLoadVM()

Load the Virtual Machine DLL.

Function declaration:

```
void vrtaLoadVM(void)
```

Parameters:

Parameter	Input/Output	Description
`<none>`		

Description:

This API call loads the correct VM DLL and prepares its API for use.

Return values:

Value	Description
`<none>`	

Notes:

This API must be called before any other Virtual Machine API is called.

The correct sequence of API calls to start a Virtual ECU running is: `vrtaLoadVM()`, `vrtaInitialize()` and `vrtaStart()`.

Important: if a Virtual ECU is using OSEK this API does not need to be called explicitly as it is called by the `main()` entry-point in the support file `vrtaOSEKSupp.cpp` that is `#included` by `osgen.cpp`.

See also:

`vrtaInitialize()`, `vrtaStart()`

9.4.15 vrtaNoneUserThread()

Generate a fatal error when an RTA-OSEK API call is made by a thread other than the application thread.

Function declaration:

```
void vrtaNoneUserThread(const char * funcName)
```

Parameters:

Not documented.

Description:

This API call is used from inside RTA-OSEK support files to generate a fatal error when an RTA-OSEK API call is made by a thread other than the application thread. **This API call is not for application use.**

Return values:

Not documented.

Notes:

9.4.16 vrtaOSGetIPL()

Get the interrupt priority level within the VM.

Function declaration:

```
vrtaIntPriority vrtaOSGetIPL(void)
```

Parameters:

Not documented.

Description:

This API call is used from inside RTA-OSEK support files to get the interrupt priority level within the *RTA-OSEK for PC* Virtual Machine. **This API call is not for application use.**

Return values:

Not documented.

Notes:

9.4.17 vrtaOSSetIPL()

Set the interrupt priority level within the VM.

Function declaration:

```
vrtaIntPriority vrtaOSSetIPL(vrtaIntPriority newIPL)
```

Parameters:

Not documented.

Description:

This API call is used from inside RTA-OSEK support files to set the interrupt priority level within the *RTA-OSEK for PC* Virtual Machine. **This API call is not for application use.**

Return values:

Not documented.

Notes:

9.4.18 vrtaRaiseEvent()

Raise an event.

Function declaration:

```
vrtaErrType vrtaRaiseEvent(vrtaDevID dev,
    const vrtaEvent *ev)
```

Parameters:

Parameter	Input/Output	Description
`dev`	Input	The ID of the virtual device raising the event.
`ev`	Input	A pointer to a structure that contains the event to be raised.

Description:

This API is used by a virtual device to raise an event.

Return values:

Value	Description
`RTVECUErr_NONE`	The API call was successful.
`RTVECUErr_Dev`	The specified device ID is invalid.
`RTVECUErr_ID`	The specified event ID is invalid.
`RTVECUErr_VAL`	The data provided in the event is invalid.

Notes:

The VM calls any event handlers that have hooked the event during `vrtaRaiseEvent()` and passes them the `vrtaEvent` structure as an argument.

See also:

`vrtaRegisterVirtualDevice()`, `vrtaGetState()`,
`vrtaEventRegister()`, `vrtaEventUnregister()`,
`vrtaHookEvent()`

9.4.19 vrtaReadHPTime()

Read the PC's high-performance timer.

Function declaration:

```
unsigned vrtaReadHPTime(unsigned desired_ticks_per_s)
```

Parameters:

Parameter	Input/Output	Description
`desired_ticks_per_s`	Input	The number of "ticks" required per second.

Description:

This API call returns the number of caller defined "ticks" that have elapsed since the Virtual ECU started. The value is derived by reading the PC's high-performance timer; therefore the resolution of the value returned depends on the details of the PC. In practice this appears to be a low multiple of the CPU clock speed on a typical PC.

Return values:

Value	Description
`<number of ticks>`	The number of "ticks" that have elapsed since the Virtual ECU started. The number of "ticks" in a second is set by the API call argument `desired_ticks_per_s`.

Notes:

See also:

9.4.20 vrtaRegisterVirtualDevice()

Register a virtual device.

Function declaration:

```
vrtaDevID vrtaRegisterVirtualDevice(
    const vrtaTextPtr name,
    const vrtaOptStringlistPtr info,
    const vrtaOptStringlistPtr events,
    const vrtaOptStringlistPtr actions,
    const vrtaActionCallback aCallback,
    const vrtaStateCallback sCallback,
    const void *tag)
```

Parameters:

Parameter	Input/Output	Description
`name`	Input	A unique name for the virtual device.
`info`	Input	A '`\n`' separated string containing information about the virtual device.
`events`	Input	A '`\n`' separated string containing descriptions of the events supported by the virtual device.
`actions`	Input	A '`\n`' separated string containing descriptions of the actions supported by the virtual device.
`aCallback`	Input	A pointer to an action callback function called to handle actions sent to the virtual device.
`sCallback`	Input	A pointer to a state callback function called to handle `vrtaGetState()` requests.
`tag`	Input	Application data passed to the `aCallback` and `sCallback` functions.

Description:

This API call is used to register a virtual device.

`name`: the `name` argument specifies a unique name for the virtual device. If the name is not unique the Virtual Machine will generate a fatal error.

`info`: the `info` argument describes the virtual device. This should be a string of the form: "`Type=<type>\nVersion=<version>\n`" where `<type>` is the type of device e.g. "`clock`", "`actuator`" or "`CAN Channel`", and `<version>` is the version of the device. The Virtual Machine does not prescribe the values of `<type>` and `<version>` as these are simply information items that can be obtained by querying the Virtual Machine's device manager (e.g. with `vrtaMonitor`).

`events`: the `events` argument describes the events supported by the virtual device. Each event supported is described by a string of one of the following forms:

(a) "`<name>`"

(b) "`<name>=<format>`"

(c) "`<name>=<format>(<format>)`".

`<name>` is the name of the event and `<format>` is a data format string as defined in section 9.3. Where device supports multiple events then the event descriptions are separated using '`\n`'.

The (a) form of event description describes an event that does not have any associated data. This would be used for an event that simply happens at some point in time. The (b) form describes an event that contains data as described by `<format>`. The (c) form describes an event that contains data as described by `<format>` before the "`()`" and input data when queried by `vrtaGetState()` as described by `<format>` inside the "`()`".

The first event in the list has event ID 1, the second has event ID 2, and so on.

`actions`: the `actions` argument describes the actions supported by device. Each action supported is described by a string of one of the following forms:

(a) "`<name>`"

(b) "`<name>=<format>`"

`<name>` is the name of the action and `<format>` is a data format string as defined in section 9.3. Where device supports multiple actions then the event descriptions are separated using '`\n`'.

The (a) form of action description describes an action that does not have any associated data. The (b) form describes an action that contains data as described by `<format>`.

The first action in the list has action ID 1, the second has action ID 2, and so on.

`aCallback`: the `aCallback` argument points to an action callback function that is called to handle actions sent to the virtual device. The action callback function has the type `vrtaActionCallback` with the following definition:

```
typedef vrtaErrType (*vrtaActionCallback)(
    void *instance, const vrtaAction *action);
```

When the action callback function is called its `instance` argument is set the value of the `tag` argument passed to `vrtaRegisterVirtualDevice()` and the `action` argument points to the `vrtaAction` structure containing the action sent to the device (see `vrtaSendAction()` for a description of the contents of the `vrtaAction` structure in section 9.2.14).

The action callback function should determine what action is to be carried out by examining the `devAction` field of `action`. It should then extract any data required from the `devEmbeddedData` or `devActionData` fields of `action` (again see `vrtaSendAction()`).

The action callback function should return `RTVECUErr_NONE` on success, `RTVECUErr_ID` if the action ID in the `devAction` field of `action` is invalid, or `RTVECUErr_VAL` if the data in `action` is invalid.

The `vrtaAction` structure pointed to by `action` and any storage pointed to by the `devActionData` field of `action` are only valid for the duration of the action callback function. If the application wishes to use this data after the action callback function has returned it must copy the data into its own storage.

In addition to handling actions described in the `actions` argument passed to `vrtaRegisterVirtualDevice()` a virtual device will also been sent a special Reset command with action ID zero. In this case the action data will be a copy of a `vrtaDevResetInfo` structure defined as follows:

```
enum vrtaResetTypes {
    vrtaDevStart,
    vrtaDevStop,
    vrtaDevWriteToPersistentStorage,
    vrtaDevReadFromPersistentStorage};
```

```
typedef struct {
    vrtaDataLen * vPSLen;
    vrtaByte **   vPSAddr;
    vrtaByte      vResetType;
} vrtaDevResetInfo;
```

The `vResetType` field describes the reason for the "reset" action as follows:

`vResetType` value	Reason for "reset" action
`vrtaDevStart`	The application thread is about to start running.
`vrtaDevStop`	The Virtual Machine is about to terminate.
`vrtaDevWriteToPersistentStorage`	The Virtual ECU is about to be reset. The virtual device may wish to arrange for data to be propagated across the reset. If it does it should set `*vPSLen` to the number of bytes of data to propagate and `*vPSAddr` to point to the data to propagate.
`vrtaDevReadFromPersistentStorage`	The Virtual ECU has been reset. The virtual device may have arranged to propagate data from before the reset. If it did then `*vPSLen` will contain the number of bytes of data propagated and `*vPSAddr` will point to the data propagated. The virtual device must copy the data from `*vPSAddr` before the action callback handler returns.

`sCallback`: the `sCallback` argument points to a state callback function that is called when `vrtaGetState()` is called to query one of the events supported by the virtual device. The state callback function has the type `vrtaStateCallback` with the following definition:

```
typedef vrtaErrType (*vrtaStateCallback)(
    void *instance, vrtaEvent *state);
```

When the state callback function is called its `instance` argument is set to the value of the `tag` argument passed to `vrtaRegisterVirtualDevice()` and the `state` argument points to the `vrtaEvent` structure containing the event to be queried. There may be incoming data in the `vrtaEvent` structure. See section 9.2.15 for a description of the contents of the `vrtaEvent` structure.

The state callback function should determine what event is to be queried by examining the `devEvent` field of `state`. It should then extract any input data required from the `devEmbeddedData` or `devEventData` fields of `state` and store the result of the query in the `devEmbeddedData` or `devEventData` fields.

The function should return `RTVECUErr_NONE` on success, `RTVECUErr_ID` if the event ID in the `devEvent` field of `state` is invalid, or `RTVECUErr_VAL` if the data in `state` is invalid.

Return values:

Value	Description
`<device ID>`	The ID of the virtual device.

Notes:

The callback functions can be called from different threads, so they must be thread-safe.

See also:

`vrtaGetState()`, `vrtaRaiseEvent()`, `vrtaEventRegister()`, `vrtaEventUnregister()`, `vrtaHookEvent()`, `vrtaSendAction()`

9.4.21 vrtaReset()

Reset the Virtual Machine.

Function declaration:

```
void vrtaReset(void)
```

Parameters:

Parameter	Input/Output	Description
`<none>`		

Description:

This API call instructs the VM to reset. It does this by creating a new Windows process and running a new copy of the VECU in it. Certain information such as command line options and connections to external programs are propagated to the new process. This creates the effect of an ECU being reset and starting execution from its reset vector. Since connections to external programs are propagated to the new process, programs communicating with the VECU (such as `vrtaServer` or `vrtaMonitor`) continue to be able to communicate with the VECU after reset and don't notice the handover of processes.

After `vrtaReset()` has been called the application thread is allowed approximately 10 seconds to terminate cleanly (either by returning from `OS_MAIN()` or calling `vrtaTerminate()`). If the application thread does not terminate within 10 seconds it is forcibly terminated.

Once the application thread has terminated the VM sends a Reset action to each virtual device to inform it that the VECU is about to reset.

Next the VM saves certain state information (such as connections to external programs) in a temporary file.

The call of `vrtaStart()` that started the application thread then returns.

Once this has happened the new Windows process starts running and the `main()` entry-point of the VECU is called by the C/C++ start-up code. This entry-point carries out the normal initialization sequence of calling `vrtaLoadVM()`, `vrtaInitialize()` and `vrtaStart()`. However, when `vrtaInitialize()` is called the VM determines that it has been reset and restores state information from the temporary file created by the original Windows process.

Return values:

Value	Description
`<none>`	

Notes:

See also:

`vrtaStart()`, `vrtaTerminate()`

9.4.22 vrtaSendAction()

Send an action to a virtual device.

Function declaration:

```
vrtaErrType vrtaSendAction(vrtaDevID id,
    const vrtaAction *a)
```

Parameters:

Parameter	Input/Output	Description
`id`	Input	The ID of the virtual device to which the action should be sent.
`a`	Input	A pointer to a structure that contains the action to be sent.

Description:

This API call causes the data in the `vrtaAction` structure to be sent to the virtual device.

Return values:

Value	Description
`RTVECUErr_NONE`	The API call was successful.
`RTVECUErr_Dev`	The specified device ID is invalid.
`RTVECUErr_ID`	The specified action ID is invalid.
`RTVECUErr_VAL`	The data provided in the action is invalid.

Notes:

The action callback function of the target device is called by the same Windows thread that calls `vrtaSendAction()`.

See also:

`vrtaRegisterVirtualDevice()`

9.4.23 vrtaSpawnThread()

Create a new *RTA-OSEK for PC* thread.

Function declaration:

```
void vrtaSpawnThread(void (*func)(void))
```

Parameters:

Parameter	Input/Output	Description
`func`	Input	The entry function for the new thread.

Description:

This API call creates a new *RTA-OSEK for PC* thread. This API is a wrapper around the Windows `CreateThread()` function that allows the VM to keep track of the number of threads running in the VECU.

A thread created with `vrtaSpawnThread()` should terminate itself as soon as it discovers that the VECU is about to terminate. This is normally done by polling `vrtaIsAppFinished()` regularly. If a thread does not do this it will continue running until forcibly terminated as the VECU process terminates.

Return values:

Value	Description
`<none>`	

Notes:

You will normally call this API during the initialization of your devices, before `OS_MAIN()` starts.

If you call it from within the application thread after `OS_MAIN()` starts, you must ensure that it cannot be interrupted.

See also:

`vrtaIsAppFinished()`, `vrtaEnterUninterruptibleSection()`, `vrtaLeaveUninterruptibleSection()`

9.4.24 vrtaStart()

Start the application thread.

Function declaration:

```
void vrtaStart(void)
```

Parameters:

Parameter	Input/Output	Description
`<none>`		

Description:

This API call requests the VM to start the application thread.

If the VECU has been loaded in autostart mode (default) then the application thread is started as soon as `vrtaStart()` is called. If the VECU has been loaded in slave mode then the application thread is not started until a Start action is sent to the ApplicationManager device.

The VM sends a Reset action to each virtual device just before the application thread starts to inform them that the application thread starting.

This is followed by a call to the application-provided function `InitializeDevices()` that should carry out any necessary virtual device initialization. Finally the main Virtual ECU application entry-point function `OS_MAIN()` is called.

`vrtaStart()` does not return until the VM terminates (e.g. because the application thread or another thread calls `vrtaTerminate()` or a Terminate action is sent to the ApplicationManager device).

Return values:

Value	Description
`<none>`	

Notes:

The correct sequence of API calls to start a Virtual ECU running is: `vrtaLoadVM()`, `vrtaInitialize()` and `vrtaStart()`.

Important: if a Virtual ECU is using OSEK this API should not be called explicitly as it is called by the `main()` entry-point in the support file `vrtaOSEKSupp.cpp` that is `#included` by `osgen.cpp`.

See also:

`InitializeDevices(), OS_MAIN(), vrtaLoadVM(), vrtaInitialize()`

9.4.25 vrtaTerminate()

Terminate the Virtual Machine.

Function declaration:

```
void vrtaTerminate(void)
```

Parameters:

Parameter	Input/Output	Description
`<none>`		

Description:

This API call instructs the VM to terminate. If this API is called by the application thread then it never returns. If this API is called by any other thread then it does return.

If `vrtaTerminate()` is called by a thread other than the application thread then the application thread is allowed approximately 10 seconds to terminate cleanly (either by returning from `OS_MAIN()` or calling `vrtaTerminate()`). If the application thread does not terminate within 10 seconds it is forcibly terminated.

Once the application thread has terminated the VM sends a Reset action to each virtual device to inform it that the VECU is about to terminate.

Finally the call of `vrtaStart()` that started the application thread returns.

Return values:

Value	Description
`<none>`	

Notes:

See also:

`vrtaStart(), vrtaReset()`

10 Standard Device Reference

Most of the functionality of the Virtual Machine is accessed through the VM's three standard devices which behave in just the same way as devices that you create in your application code.

This chapter describes the purpose of these 3 devices and the actions and events they support.

10.1 vrtaStdDevices.h

The header file `vrtaStdDevices.h` contains definitions of the device, action and event IDs used by the internal devices. `vrtaStdDevices.h` is automatically included if you include `vrtaCore.h`.

10.2 Action and Event Descriptions

Each action or event supported by an internal device is described by a standard table, as below, followed by text to explain the purpose of the action or event.

ID	Data Format
`YYYY`	`ZZZZ`

The table contains the ID of the action or event (`YYYY`), and the format of the action or event data (`ZZZZ`).

For actions, `ZZZZ` will be a data format string describing the format of the data in the action (e.g. "`%s`" for string data).

For events that do not require any input data to be supplied when they are queried, `ZZZZ` will be a data format string describing the format of the data in the event (e.g. "`%u`" for a single unsigned numeric value).

For events that do require input data to be supplied when they are queried, `ZZZZ` will be a data format string followed by a second data format string enclosed in "`()`". The first data format string describes the format of the data in the event. The data second data format string describes the format of the input data required when the event is queried (e.g. "`%s(%u)`" for an event that contains string data and requires a single unsigned numeric value as input data when queried).

Refer to section 9.3 for a description of data format strings.

10.3 Device Manager

The Device Manager (DM) is the internal device that manages all devices. The DM is identified as follows:

Device ID Constant	Name
`DM_DEVICE_ID`	DeviceManager

10.3.1 Action: EventRegister

Action ID Constant	Data Format
`DM_ACTION_ID_EventRegister`	`%s`

This action is only available for use via the diagnostic interface and is only used by external monitor programs.

This action registers a TCP/IP port wishing to hook events. The data is a string containing a list of '`\n`' separated values. The first value is the IP address of the listener - e.g. "192.168.0.100" or "localhost". The second value is the TCP port number in decimal - e.g. "2034".

This action causes the diagnostic interface to open a network connection to the specified TCP/IP port and then associate a 'listener' instance with it. Each diagnostic connection can have at most one such listener.

The listener is removed if the connection breaks or if another EventRegister action is received.

10.3.2 Action: HookEvents

Action ID Constant	Data Format
`DM_ACTION_ID_HookEvents`	`%s`

This action is only available for use via the diagnostic interface and is only used by external monitor programs.

This action specifies which events to hook for the diagnostic interface connection on which the action is received. The data is a '`\n`' separated list of items of the form `<device>=<event1>,<event2>,…,<eventN>`. Where `<device>` is a device ID in decimal and `<eventi>` is an event ID in decimal. Such an item means hook the events with IDs `<event1>`…`<eventN>` for the device with ID `<device>` - e.g. "`3=1,3,6\n5=2\n7=2,3`".

The data forms the complete list of events to hook for the diagnostic connection. Subsequent HookEvents actions replace the events hooked rather than add to them. An empty list means that no events will be hooked.

10.3.3 Action: ListAll

Action ID Constant	Data Format
`DM_ACTION_ID_ListAll`	`<none>`

This action causes the DM to raise a DeviceList event.

10.3.4 Action: GetDeviceActions

Action ID Constant	Data Format
`DM_ACTION_ID_GetDeviceActions`	`%s`

This action causes the DM to raise a DeviceActions event for the device named by the action data.

10.3.5 Action: GetDeviceEvents

Action ID Constant	Data Format
`DM_ACTION_ID_GetDeviceEvents`	`%s`

This action causes the DM to raise a DeviceEvents event for the device named by the action data.

10.3.6 Action: GetDeviceInfo

Action ID Constant	Data Format
`DM_ACTION_ID_GetDeviceInfo`	`%s`

This action causes the DM to raise a DeviceInfo event for the device named by the action data.

10.3.7 Event: DeviceList

Event ID Constant	Data Format
`DM_EVENT_ID_DeviceList`	`%s`

The data is a '`\n`' separated list of all of the devices registered with the Virtual Machine.

10.3.8 Event: DeviceActions

Event ID Constant	Data Format
`DM_EVENT_ID_DeviceActions`	`%s(%s)`

The data is a '`\n`' separated list of all of the actions supported by the named device in the same form as used for specifying the list of actions supported by a virtual device in the `vrtaRegisterVirtualDevice()` call. If the event is raised in response to a GetDeviceActions action then the device is named by the action data. If the event is queried the device is named by the event input data.

10.3.9 Event: DeviceEvents

Event ID Constant	Data Format
`DM_EVENT_ID_DeviceEvents`	`%s(%s)`

The data is a '`\n`' separated list of all of the events supported by the named device in the same form as used for specifying the list of events supported by a virtual device in the `vrtaRegisterVirtualDevice()` call. If the event is raised in response to a GetDeviceEvents action then the device is named by the action data. If the event is queried the device is named by the event input data.

10.3.10 Event: DeviceInfo

Event ID Constant	Data Format
`DM_EVENT_ID_DeviceInfo`	`%s(%s)`

The data is information about the named device in the same form as used for specifying virtual device information in the `vrtaRegisterVirtualDevice()` call. If the event is raised in response to a GetDeviceInfo action then the device is named by the action data. If the event is queried the device is named by the event input data.

10.4 Interrupt Control Unit

The Interrupt Control Unit (ICU) is the internal device that manages interrupts within the Virtual Machine and arranges for interrupt handlers to be invoked within the application thread.

The ICU implements a multilevel interrupt controller. There are 32 interrupts numbered 1 to 32 inclusive. Interrupt number n corresponds to interrupt vector number n. Each interrupt has a priority in the range 1 to 32 inclusive. The priority of an interrupt is set in the corresponding interrupt vector table entry. See the `vrtaInitialize()` call for a description of the interrupt vector table.

The ICU maintains the current interrupt priority level (IPL). This is a number in the range zero to 32 inclusive. If an interrupt handler is running then the current IPL is equal to the priority of the corresponding interrupt. If non interrupt code is running the current IPL is zero.

Each interrupt has a pending flag and may be masked (disabled) or unmasked (enabled). An interrupt is made pending (i.e. its pending flag is set) by sending a Raise action to the ICU. The ICU invokes the interrupt handler for the highest priority pending and unmasked interrupt that has a priority higher than the current IPL. If an interrupt is pending but is masked its handler will not be invoked until the interrupt is unmasked. If an interrupt is pending but the current IPL is higher or equal to the priority of the interrupt then its handler will not be invoked until the IPL drops below the priority of the interrupt.

An interrupt's pending flag is cleared just before its handler is invoked. Therefore if the handler for an interrupt sends a Raise action to the ICU for the same interrupt the interrupt will become pending again and a second instance of the interrupt handler will run as soon as the first ends.

A higher priority interrupt handler can pre-empt a lower priority interrupt handler.

If two interrupts of the same priority are pending then the one with the lower interrupt vector number is handled first.

When the Virtual ECU starts all interrupts are masked (disabled) and the current IPL is zero.

The ICU is identified as follows:

Device ID Constant	Name
`ICU_DEVICE_ID`	ICU

10.4.1 Action: Raise

Action ID Constant	Data Format
`ICU_ACTION_ID_Raise`	`%d:;1;32`

This action makes the specified interrupt number pending (i.e. sets the interrupt's pending flag).

10.4.2 Action: Clear

Action ID Constant	Data Format
`ICU_ACTION_ID_Clear`	`%d:;1;32`

This action clears the specified interrupt number's pending flag. Note that it is not necessary to send this action to the ICU in an interrupt handler to clear the pending flag of the interrupt being handled since an interrupt's pending flag is cleared just before its handler is invoked.

10.4.3 Action: Mask

Action ID Constant	Data Format
`ICU_ACTION_ID_Mask`	`%d:;1;32`

This action masks (disables) the specified interrupt number.

10.4.4 Action: Unmask

Action ID Constant	Data Format
`ICU_ACTION_ID_Unmask`	`%d:;1;32`

This action unmasks (enables) the specified interrupt number.

10.4.5 Action: GetPending

Action ID Constant	Data Format
`ICU_ACTION_ID_GetPending`	`<none>`

This action causes a Pending event to be raised.

10.4.6 Action: GetIPL

Action ID Constant	Data Format
`ICU_ACTION_ID_GetIPL`	`<none>`

This action causes an IPL event to be raised.

10.4.7 Action: SetIPL

Action ID Constant	Data Format
`ICU_ACTION_ID_SetIPL`	`%d:;0;32`

This action sets the IPL to the specified value.

10.4.8 Event: Pending

Event ID Constant	Data Format
`ICU_EVENT_ID_Pending`	`%b`

This event contains a list of all of the currently pending interrupts. Bit n is set in the event if interrupt number n is pending (where bit 1 is the least significant bit).

The event is raised in response to a GetPending action or when the list of pending interrupts changes.

10.4.9 Event: Start

Event ID Constant	Data Format
`ICU_EVENT_ID_Start`	`%d`

This event is raised just before an interrupt handler is invoked. The number of the interrupt is specified in the event data.

10.4.10 Event: Stop

Event ID Constant	Data Format
`ICU_EVENT_ID_Stop`	`%d`

This event is raised just after an interrupt handler has ended. The number of the interrupt is specified in the event data.

10.4.11 Event: IPL

Event ID Constant	Data Format
`ICU_EVENT_ID_IPL`	`%d`

This event contains the current IPL. The event is raised in response to a GetIPL action or when the current IPL changes.

10.4.12 Event: EnabledVecs

Event ID Constant	Data Format
`ICU_EVENT_ID_MASKS`	`%b`

This event contains a list of all of the currently enabled (unmasked) interrupts. Bit n is set in the event if interrupt number n is enabled (where bit 1 is the least significant bit).

The event is raised when the list of enabled interrupts changes.

10.5 Application Manager

The Application Manager (AM) is the internal device that manages the Virtual ECU application

The AM is identified as follows:

Device ID Constant	Name
`AM_DEVICE_ID`	ApplicationManager

10.5.1 Action: Start

Action ID Constant	Data Format
`AM_ACTION_ID_Start`	`<none>`

This action starts the application thread running in a Virtual ECU that was loaded in *slave* mode.

10.5.2 Action: Terminate

Action ID Constant	Data Format
`AM_ACTION_ID_Terminate`	`<none>`

This action tells the Virtual Machine to terminate. It has the same effect as the `vrtaTerminate()` call.

10.5.3 Action: Pause

Action ID Constant	Data Format
`AM_ACTION_ID_Pause`	`<none>`

This action tells the Virtual Machine to suspend execution of the application thread.

10.5.4 Action: Restart

Action ID Constant	Data Format
`AM_ACTION_ID_Restart`	`<none>`

This action tells the Virtual Machine to restart execution of the application thread after it has previously been suspended.

10.5.5 Action: Reset

Action ID Constant	Data Format
`AM_ACTION_ID_Reset`	`<none>`

This action tells the Virtual Machine to reset. It has the same effect as the `vrtaReset()` call.

10.5.6 Action: GetInfo

Action ID Constant	Data Format
`AM_ACTION_ID_GetInfo`	`<none>`

This action causes an Info event to be raised.

10.5.7 Action: TestOption

Action ID Constant	Data Format
`AM_ACTION_ID_TestOption`	`%s`

This action causes an Option event to be raised to signal if the named command-line option exists. The option prefix ('-' or '/') is not specified.

10.5.8 Action: ReadOption

Action ID Constant	Data Format
`AM_ACTION_ID_ReadOption`	`%s`

This action causes an OptionText event to be raised containing the full text of the command-line option that starts with the specified string.

10.5.9 Action: ReadParam

Action ID Constant	Data Format
`AM_ACTION_ID_ReadParam`	`%u`

This action causes a ParamText event to be raised containing the full text of the specified command-line parameter. The first command line parameter (the executable name) is number zero, the second parameter is number one, and so on.

10.5.10 Event: Started

Event ID Constant	Data Format
`AM_EVENT_ID_Started`	`<none>`

This event is raised to indicate that the application thread has started.

10.5.11 Event: Paused

Event ID Constant	Data Format
`AM_EVENT_ID_Paused`	`<none>`

This event is raised to indicate that the application thread has been suspended.

10.5.12 Event: Restarted

Event ID Constant	Data Format
`AM_EVENT_ID_Restarted`	`<none>`

This event is raised to indicate that the application thread has been restarted.

10.5.13 Event: Reset

Event ID Constant	Data Format
`AM_EVENT_ID_Reset`	`<none>`

This event is raised to indicate that the Virtual ECU has been reset.

10.5.14 Event: Terminated

Event ID Constant	Data Format
`AM_EVENT_ID_Terminated`	`<none>`

This event is raised to indicate that the Virtual Machine is about to terminate.

10.5.15 Event: Info

Event ID Constant	Data Format
`AM_EVENT_ID_Info`	`%s`

This event contains version information about the Virtual ECU application and the Virtual Machine.

10.5.16 Event: Option

Event ID Constant	Data Format
`AM_EVENT_ID_Option`	`%u(%s)`

This event contains the number 1 if the named option is present on the command line or zero if it is not. If the event is raised in response to a TestOption action then the option is named in the action data. If the event is queried then the option is named in the input data. The option prefix ('-' or '/') is not included in the name.

10.5.17 Event: OptionText

Event ID Constant	Data Format
`AM_EVENT_ID_OptionText`	`%s(%s)`

This event contains an empty string if a command line options starting with the specified prefix does not exist. If a command line option starting with the specified prefix does exist then the event contains the full text of the option. If the event is raised in response to a ReadOption action then the prefix is the action data. If the event is queried then the prefix is the input data.

10.5.18 Event: ParamText

Event ID Constant	Data Format
`AM_EVENT_ID_ParamText`	`%s(%u)`

This event contains the full text of the specified command-line parameter. The first command line parameter (the executable name) is zero, the second parameter is number one, and so on. If the specified command line parameter does not exist the event contains the empty string. If the event is raised in response to a ReadParam action then the parameter number is in the action data. If the event is queried then the parameter number is in the input data.

10.5.19 Event: State

Event ID Constant	Data Format
`AM_EVENT_ID_State`	`Loaded| Running| Paused| Terminating| Resetting`

This event contains the current state of the Virtual ECU.

11 Sample Device Reference

RTA-OSEK for PC includes a collection of sample virtual devices that implement commonly used devices such as clocks, counters, comparators, actuators and sensors. These are contained in the files `vrtaSampleDevices.cpp` and `vrtaSampleDevices.h`. All of the sample virtual devices make use of (i.e. are derived from) the C++ virtual device framework provided by the `vrtaDevice` base class (see the file `vrtaDevice.h`).

11.1 Summary

The following sample devices are provided:

`vrtaClock`	A clock source for counter devices.
`vrtaUpCounter`	A counter that counts upwards.
`vrtaDownCounter`	A counter that counts downwards.
`vrtaSensor`	A generic sensor.
`vrtaSensorToggleSwitch`	A two position "toggle" switch.
`vrtaSensorMultiwaySwitch`	A multi-position switch.
`vrtaActuator`	A generic actuator.
`vrtaActuatorLight`	A light that can be on or off.
`vrtaActuatorDimmableLight`	A light that can be set to different levels of brightness.
`vrtaActuatorMultiColorLight`	A light that can be set to different colors.
`vrtaCompare`	A comparator that can generate an interrupt when other another device reaches a specified value.
`vrtaIO`	An I/O space.

11.2 Compiling the Sample Devices

You must compile and link the file `vrtaSampleDevices.cpp` with your application if you want to use any of the sample devices.

If you are creating an RTA-OSEK application, `vrtaSampleDevices.cpp` is automatically `#included` within `osgen.cpp` unless you define the preprocessor macro `VRTA_EXCLUDE_SAMPLE_DEVICES` when compiling `osgen.cpp`.

11.3 Device Descriptions

Each sample device is described in the same way. The description starts with an introduction to the purpose and operation of the device. This is followed by a description of the C++ methods exported by the device class and then the actions and events supported by the device.

11.3.1 Methods

Each C++ method is described in a standard form as follows:

The title gives the name of the method.

A brief description of the method is provided.

Method declaration:

Interface in C++ syntax.

Parameters:

Parameter	Input/Output	Description
`Parameter Name`	Input/Output	Description.

Description:

Explanation of the functionality of the method.

Return values:

Value	Description
`Return values.`	Description of return value.

11.3.2 Actions and Events

Each action or event supported by a sample device is described by a standard table, as below, followed by text to explain the purpose of the action or event.

ID	Data Format
`XXXX`	`YYYY`

The table contains the numeric ID of the action or event (`XXXX`), and the format of the action or event data (`YYYY`).

11.4 vrtaClock

A `vrtaClock` device provides a time source for `vrtaUpCounter` and `vrtaDownCounter` counter devices. A `vrtaClock` device uses a Windows multi-media timer to provide a source of very-close-to 1 millisecond ticks.

A `vrtaClock` device ticks an attached counter every `T` milliseconds. `T` is calculated by multiplying the `vrtaClock` device's clock tick interval by its scaling factor. The scaling factor has a multiplier and a divisor. If the clock tick interval (set in the constructor or with the `SetInterval()` method) is `interval`, and the scaling factor (set with the `SetScale()` method) is `mult` / `div`, then an attached counter would be ticked every `(interval * mult / div)` milliseconds. By default the scale factor is 1 / 1.

Multiple counters may be attached to the same `vrtaClock` device.

Important: if using a `vrtaClock` device in a VECU, the VECU executable will need to be linked with the Windows multi-media library. (This might be done automatically depending on your compiler.)

11.4.1 Method: vrtaClock()

The constructor.

Method declaration:

```
vrtaClock(const vrtaTextPtr name, unsigned interval)
```

Parameters:

Parameter	Input/Output	Description
`name`	Input	The name of the virtual device.
`interval`	Input	The number of milliseconds in one clock tick.

Description:

This is the constructor used to create an instance of a `vrtaClock` device.

Return values:

Value	Description
`<none>`	

11.4.2 Method: SetInterval()

Set the tick interval.

Method declaration:

```
void SetInterval(unsigned interval)
```

Parameters:

Parameter	Input/Output	Description
`interval`	Input	The number of milliseconds in one clock tick.

Description:

This method is used to change the number of milliseconds in one clock tick.

Return values:

Value	Description
`<none>`	

11.4.3 Method: SetScale()

Set the scaling factor.

Method declaration:

```
void SetScale(unsigned mult, unsigned div)
```

Parameters:

Parameter	Input/Output	Description
`mult`	Input	The multiplier for the scaling factor.
`div`	Input	The divisor for the scaling factor.

Description:

This method sets the scaling factor for the clock.

Return values:

Value	Description
`<none>`	

11.4.4 Method: Start()

Start the clock ticking.

Method declaration:

```
void Start(void)
```

Parameters:

Parameter	Input/Output	Description
`<none>`		

Description:

This method starts the clock device ticking any attached counters.

Return values:

Value	Description
`<none>`	

11.4.5 Method: Stop()

Stop the clock ticking.

Method declaration:

```
void Stop(void)
```

Parameters:

Parameter	Input/Output	Description
`<none>`		

Description:

This method stops the clock device ticking any attached counters.

Return values:

Value	Description
`<none>`	

11.4.6 Action: Interval

Action ID	Data Format
1	`%u`

This action sets the number of milliseconds in a clock tick. The action data is the number of milliseconds in a clock tick. This action is equivalent to the `SetInterval()` method.

11.4.7 Action: Scale

Action ID	Data Format
2	`%u,%u`

This action sets the device's scaling factor. The first number in the action data is the scaling factor multiplier and the second number is the divisor. This action is equivalent to the `SetScale()` method.

11.4.8 Action: Start

Action ID	Data Format
3	`<none>`

This action starts the clock device ticking any attached counters. This action is equivalent to the `Start()` method.

11.4.9 Action: Stop

Action ID	Data Format
4	`<none>`

This action stops the clock device ticking any attached counters. This action is equivalent to the `Stop()` method.

11.4.10 Event: Interval

Event ID	Data Format
1	`%u`

This event is raised when the clock device's tick interval changes. The event data is the new tick interval.

11.4.11 Event: Scale

Event ID	Data Format
2	`%u,%u`

This event is raised when the clock device's scaling factor changes. The event data is the new scaling factor multiplier followed by the new divisor.

11.4.12 Event: Running

Event ID	Data Format
3	`%u:;0;1`

This event is raised when the clock is started or stopped. The event data is 1 if the clock is now running (i.e. has been started) or zero if the clock is not now running (i.e. has been stopped).

11.5 vrtaUpCounter

A `vrtaUpCounter` is a counter device that is driven by a `vrtaClock` device. It has a minimum value, a maximum value and a current value. When a `vrtaUpCounter` device is ticked by a `vrtaClock` device and its current value is less than its maximum value then its current value is incremented. When the `vrtaUpCounter` is ticked and its current value is equal to its maximum value then its current value is set back to its minimum value. The cyclic period of a `vrtaUpCounter` is thus (maximum – minimum) + 1.

By default the minimum value is zero, the maximum value is 4294967295, and the current value starts at zero.

11.5.1 Method: vrtaUpCounter()

The constructor.

Method declaration:

```
vrtaUpCounter(const vrtaTextPtr name, vrtaClock &clock)
```

Parameters:

Parameter	Input/Output	Description
`name`	Input	The name of the virtual device.
`clock`	Input	The `vrtaClock` device that will be used to drive the counter.

Description:

This is the constructor used to create an instance of a `vrtaUpCounter` device.

Return values:

Value	Description
`<none>`	

11.5.2 Method: Min()

Get the minimum value.

Method declaration:

```
unsigned Min(void)
```

Parameters:

Parameter	Input/Output	Description
`<none>`		

Description:

This method is used to get the minimum value of the counter.

Return values:

Value	Description
`<a value>`	The minimum value of the counter.

11.5.3 Method: Max()

Get the maximum value.

Method declaration:

```
unsigned Max(void)
```

Parameters:

Parameter	Input/Output	Description
`<none>`		

Description:

This method is used to get the maximum value of the counter.

Return values:

Value	Description
`<a value>`	The maximum value of the counter.

11.5.4 Method: Value()

Get the current value.

Method declaration:

```
unsigned Value(void)
```

Parameters:

Parameter	Input/Output	Description
`<none>`		

Description:

This method is used to get the current value of the counter.

Return values:

Value	Description
`<a value>`	The current value of the counter.

11.5.5 Method: SetMin()

Set the minimum value.

Method declaration:

```
void SetMin(unsigned v)
```

Parameters:

Parameter	Input/Output	Description
`v`	Input	The new minimum value for the counter.

Description:

This method is used to set the minimum value of the counter. If the current value of the counter is smaller than the new minimum value then the current value is set to the new minimum value.

Return values:

Value	Description
`<none>`	

11.5.6 Method: SetMax()

Set the maximum value.

Method declaration:

```
void SetMax(unsigned v)
```

Parameters:

Parameter	Input/Output	Description
`v`	Input	The new maximum value for the counter.

Description:

This method is used to set the maximum value of the counter. If the current value of the counter is greater than the new maximum value then the current value is set to the minimum value.

Return values:

Value	Description
`<none>`	

11.5.7 Method: SetVal()

Set the current value.

Method declaration:

```
void SetVal(unsigned v)
```

Parameters:

Parameter	Input/Output	Description
`v`	Input	The new value for the counter.

Description:

This method is used to set the current value of the counter. If the new current value of the counter is smaller than the minimum value or greater than the maximum value then the current value is set to the minimum value.

Return values:

Value	Description
`<none>`	

11.5.8 Method: Start()

Start the counter counting.

Method declaration:

```
void Start(void)
```

Parameters:

Parameter	Input/Output	Description
`<none>`		

Description:

This method is used to start the counter counting when ticked by the attached `vrtaClock` device.

Return values:

Value	Description
`<none>`	

11.5.9 Method: Stop()

Stop the counter counting.

Method declaration:

```
void Stop(void)
```

Parameters:

Parameter	Input/Output	Description
`<none>`		

Description:

This method is used to stop the counter counting when ticked by the attached `vrtaClock` device.

Return values:

Value	Description
`<none>`	

11.5.10 Action: Minimum

Action ID	Data Format
1	`%u`

This action sets the minimum value of the counter. It is the equivalent of the `SetMin()` method.

11.5.11 Action: Maximum

Action ID	Data Format
2	`%u`

This action sets the maximum value of the counter. It is the equivalent of the `SetMax()` method.

11.5.12 Action: Set

Action ID	Data Format
3	`%u`

This action sets the current value of the counter. It is the equivalent of the `SetVal()` method.

11.5.13 Action: Start

Action ID	Data Format
4	`<none>`

This method is used to start the counter counting when ticked by the attached `vrtaClock` device. It is the equivalent of the `Start()` method.

11.5.14 Action: Stop

Action ID	Data Format
5	`<none>`

This method is used to stop the counter counting when ticked by the attached `vrtaClock` device. It is the equivalent of the `Stop()` method.

11.5.15 Action: Report

Action ID	Data Format
6	`<none>`

This action causes a Set event to be raised.

11.5.16 Event: Set

Event ID	Data Format
`1`	`%u`

This event contains the current value of the counter. It is raised in response to a Report action.

11.6 vrtaDownCounter

A `vrtaDownCounter` is a counter device that is driven by a `vrtaClock` device. It has a minimum value, a maximum value and a current value. When a `vrtaDownCounter` device is ticked by a `vrtaClock` device and its current value is greater than its minimum value then its current value is decremented. When the `vrtaDownCounter` is ticked and its current value is equal to its minimum value then its current value is set back to its maximum value. The cyclic period of a `vrtaDownCounter` is thus (maximum – minimum) + 1.

By default the minimum value is zero, the maximum value is 4294967295, and the current value starts at zero.

11.6.1 Method: vrtaDownCounter()

The constructor.

Method declaration:

```
vrtaDownCounter(const vrtaTextPtr name,
    vrtaClock &clock)
```

Parameters:

Parameter	Input/Output	Description
`name`	Input	The name of the virtual device.
`clock`	Input	The `vrtaClock` device that will be used to drive the counter.

Description:

This is the constructor used to create an instance of a `vrtaDownCounter` device.

Return values:

Value	Description
`<none>`	

11.6.2 Method: Min()

Get the minimum value.

Method declaration:

```
unsigned Min(void)
```

Parameters:

Parameter	Input/Output	Description
`<none>`		

Description:

This method is used to get the minimum value of the counter.

Return values:

Value	Description
`<a value>`	The minimum value of the counter.

11.6.3 Method: Max()

Get the maximum value.

Method declaration:

```
unsigned Max(void)
```

Parameters:

Parameter	Input/Output	Description
`<none>`		

Description:

This method is used to get the maximum value of the counter.

Return values:

Value	Description
`<a value>`	The maximum value of the counter.

11.6.4 Method: Value()

Get the current value.

Method declaration:

```
unsigned Value(void)
```

Parameters:

Parameter	Input/Output	Description
`<none>`		

Description:

This method is used to get the current value of the counter.

Return values:

Value	Description
`<a value>`	The current value of the counter.

11.6.5 Method: SetMin()

Set the minimum value.

Method declaration:

```
void SetMin(unsigned v)
```

Parameters:

Parameter	Input/Output	Description
`v`	Input	The new minimum value for the counter.

Description:

This method is used to set the minimum value of the counter. If the current value of the counter is smaller than the new minimum value then the current value is set to the new minimum value.

Return values:

Value	Description
`<none>`	

11.6.6 Method: SetMax()

Set the maximum value.

Method declaration:

```
void SetMax(unsigned v)
```

Parameters:

Parameter	Input/Output	Description
v	Input	The new maximum value for the counter.

Description:

This method is used to set the maximum value of the counter. If the current value of the counter is greater than the new maximum value then the current value is set to the minimum value.

Return values:

Value	Description
<none>	

11.6.7 Method: SetVal()

Set the current value.

Method declaration:

```
void SetVal(unsigned v)
```

Parameters:

Parameter	Input/Output	Description
v	Input	The new value for the counter.

Description:

This method is used to set the current value of the counter. If the new current value of the counter is smaller than the minimum value or greater than the maximum value then the current value is set to the minimum value.

Return values:

Value	Description
<none>	

11.6.8 Method: Start()

Start the counter counting.

Method declaration:

```
void Start(void)
```

Parameters:

Parameter	Input/Output	Description
`<none>`		

Description:

This method is used to start the counter counting when ticked by the attached `vrtaClock` device.

Return values:

Value	Description
`<none>`	

11.6.9 Method: Stop()

Stop the counter counting.

Method declaration:

```
void Stop(void)
```

Parameters:

Parameter	Input/Output	Description
`<none>`		

Description:

This method is used to stop the counter counting when ticked by the attached `vrtaClock` device.

Return values:

Value	Description
`<none>`	

11.6.10 Action: Minimum

Action ID	Data Format
1	`%u`

This action sets the minimum value of the counter. It is the equivalent of the `SetMin()` method.

11.6.11 Action: Maximum

Action ID	Data Format
2	`%u`

This action sets the maximum value of the counter. It is the equivalent of the `SetMax()` method.

11.6.12 Action: Set

Action ID	Data Format
3	`%u`

This action sets the current value of the counter. It is the equivalent of the `SetVal()` method.

11.6.13 Action: Start

Action ID	Data Format
4	`<none>`

This method is used to start the counter counting when ticked by the attached `vrtaClock` device. It is the equivalent of the `Start()` method.

11.6.14 Action: Stop

Action ID	Data Format
5	`<none>`

This method is used to stop the counter counting when ticked by the attached `vrtaClock` device. It is the equivalent of the `Stop()` method.

11.6.15 Action: Report

Action ID	Data Format
6	`<none>`

This action causes a Set event to be raised.

11.6.16 Event: Set

Event ID	Data Format
`1`	`%u`

This event contains the current value of the counter. It is raised in response to a Report action.

11.7 vrtaSensor

A `vrtaSensor` device models a sensor. That is, a device which takes input from one source, stores that input and then allows the input to be read by an application.

`vrtaSensor` represents a generic sensor; `vrtaSensorToggleSwitch` and `vrtaSensorMultiwaySwitch` are derived from `vrtaSensor` and represent more specialized sensors.

A sensor has a current value and a maximum value. The current value of the sensor can be set to a value between zero and the maximum value inclusive. Events are raised whenever the current value or maximum value changes.

When a sensor is created the current value is zero and the maximum value is 4294967295.

11.7.1 Method: vrtaSensor()

The constructor.

Method declaration:

```
vrtaSensor(const vrtaTextPtr name)
```

Parameters:

Parameter	Input/Output	Description
`Name`	Input	The name of the virtual device.

Description:

This is the constructor used to create an instance of a `vrtaSensor` device.

Return values:

Value	Description
`<none>`	

11.7.2 Method: GetMax()

Get the maximum value.

Method declaration:

```
unsigned GetMax(void)
```

Parameters:

Parameter	Input/Output	Description
`<none>`		

Description:

This method is used to get the maximum value of the sensor.

Return values:

Value	Description
`<a value>`	The maximum value of the sensor.

11.7.3 Method: Value()

Get the current value.

Method declaration:

```
unsigned Value(void)
```

Parameters:

Parameter	Input/Output	Description
`<none>`		

Description:

This method is used to get the current value of the sensor.

Return values:

Value	Description
`<a value>`	The current value of the sensor.

11.7.4 Method: SetMax()

Set the maximum value.

Method declaration:

```
void SetMax(unsigned v)
```

Parameters:

Parameter	Input/Output	Description
`V`	Input	The new maximum value for the sensor.

Description:

This method is used to set the maximum value of the sensor. If the current value of the sensor is greater than the new maximum value then the current value is set to zero.

Return values:

Value	Description
`<none>`	

11.7.5 Method: SetVal()

Set the current value.

Method declaration:

```
void SetVal(unsigned v)
```

Parameters:

Parameter	Input/Output	Description
`v`	Input	The new value for the sensor.

Description:

This method is used to set the current value of the sensor. If the new current value of the sensor is greater than the maximum value then the current value is not set.

Return values:

Value	Description
`<none>`	

11.7.6 Action: Value

Action ID	Data Format
1	`%u`

This action sets the current value of the sensor. It is the equivalent of the `SetVal()` method.

11.7.7 Action: Maximum

Action ID	Data Format
2	`%u`

This action sets the maximum value of the sensor. It is the equivalent of the `SetMax()` method.

11.7.8 Event: Value

Event ID	Data Format
1	`%u`

This event contains the current value of the sensor. It is raised whenever the value of the sensor changes.

11.7.9 Event: Maximum

Event ID	Data Format
2	`%u`

This event contains the maximum value of the sensor. It is raised whenever the maximum value of the sensor changes.

11.8 vrtaSensorToggleSwitch

A `vrtaSensorToggleSwitch` is a special form of a sensor that has only two possible values, zero and one, corresponding to "off" and "on".

When a `vrtaSensorToggleSwitch` is created its current value is zero.

11.8.1 Method: vrtaSensorToggleSwitch()

The constructor.

Method declaration:

```
vrtaSensorToggleSwitch(const vrtaTextPtr name)
```

Parameters:

Parameter	Input/Output	Description
`name`	Input	The name of the virtual device.

Description:

This is the constructor used to create an instance of a `vrtaSensorToggleSwitch` device.

Return values:

Value	Description
`<none>`	

11.8.2 Method: Value()

Get the current value.

Method declaration:

```
unsigned Value(void)
```

Parameters:

Parameter	Input/Output	Description
`<none>`		

Description:

This method is used to get the current value of the sensor.

Return values:

Value	Description
`<a value>`	The current value of the sensor.

11.8.3 Method: SetVal()

Set the current value.

Method declaration:

```
void SetVal(unsigned v)
```

Parameters:

Parameter	Input/Output	Description
`v`	Input	The new value for the sensor.

Description:

This method is used to set the current value of the sensor. If the new current value of the sensor is greater than 1 then the current value is not set.

Return values:

Value	Description
`<none>`	

11.8.4 Action: Position

Action ID	Data Format
1	`%u:;0;1`

This action sets the current value (position) of the sensor. It is the equivalent of the `SetVal()` method.

11.8.5 Event: Position

Event ID	Data Format
1	`%u:;0;1`

This event contains the current value (position) of the sensor. It is raised whenever the value of the sensor changes.

11.9 vrtaSensorMultiwaySwitch

A `vrtaSensorMultiwaySwitch` is a special form of a sensor that represents a switch with a number of possible positions. The number of positions is set when the device is created (but can be changed later).

When a `vrtaSensorMultiwaySwitch` is created its current value is zero.

11.9.1 Method: vrtaSensorMultiwaySwitch()

The constructor.

Method declaration:

```
vrtaSensorMultiwaySwitch(const vrtaTextPtr name,
    unsigned ways)
```

Parameters:

Parameter	Input/Output	Description
`name`	Input	The name of the virtual device.
`ways`	Input	The number of positions the switch may take minus 1.

Description:

This is the constructor used to create an instance of a `vrtaSensorMultiwaySwitch` device. The sensor may have a value in the range zero to `ways` inclusive.

Return values:

Value	Description
`<none>`	

11.9.2 Method: GetMax()

Get the maximum value.

Method declaration:

```
unsigned GetMax(void)
```

Parameters:

Parameter	Input/Output	Description
`<none>`		

Description:

This method is used to get the maximum value of the sensor.

Return values:

Value	Description
`<a value>`	The maximum value of the sensor.

11.9.3 Method: Value()

Get the current value.

Method declaration:

```
unsigned Value(void)
```

Parameters:

Parameter	Input/Output	Description
`<none>`		

Description:

This method is used to get the current value of the sensor.

Return values:

Value	Description
`<a value>`	The current value of the sensor.

11.9.4 Method: SetMax()

Set the maximum value.

Method declaration:

```
void SetMax(unsigned v)
```

Parameters:

Parameter	Input/Output	Description
`v`	Input	The new maximum value for the sensor.

Description:

This method is used to set the maximum value of the sensor (i.e. to override the value of the `ways` argument used in the constructor). If the current value of the sensor is greater than the new maximum value then the current value is set to zero.

Return values:

Value	Description
`<none>`	

11.9.5 Method: SetVal()

Set the current value.

Method declaration:

```
void SetVal(unsigned v)
```

Parameters:

Parameter	Input/Output	Description
`v`	Input	The new value for the sensor.

Description:

This method is used to set the current value of the sensor. If the new current value of the sensor is greater than the maximum value then the current value is not set.

Return values:

Value	Description
`<none>`	

11.9.6 Action: Value

Action ID	Data Format
1	`%u`

This action sets the current value of the sensor. It is the equivalent of the `SetVal()` method.

11.9.7 Action: Maximum

Action ID	Data Format
2	`%u`

This action sets the maximum value of the sensor. It is the equivalent of the `SetMax()` method.

11.9.8 Event: Value

Event ID	Data Format
`1`	`%u`

This event contains the current value of the sensor. It is raised whenever the value of the sensor changes.

11.9.9 Event: Maximum

Event ID	Data Format
`2`	`%u`

This event contains the maximum value of the sensor. It is raised whenever the maximum value of the sensor changes.

11.10 vrtaActuator

A `vrtaActuator` device models an actuator. That is, a device which has its value set by an application and then signals that value to entities outside of the ECU. `vrtaActuator` represents a generic actuator; `vrtaActuatorLight`, `vrtaActuatorDimmableLight` and `vrtaActuatorMultiColorLight` are derived from `vrtaActuator` and represent more specialized actuators.

An actuator has a current value and a maximum value. The current value of the actuator can be set to a value between zero and the maximum value inclusive. Events are raised whenever the current value or maximum value changes.

When an actuator is created the current value is zero and the maximum value is 4294967295.

11.10.1 Method: vrtaActuator()

The constructor.

Method declaration:

```
vrtaActuator(const vrtaTextPtr name)
```

Parameters:

Parameter	Input/Output	Description
`name`	Input	The name of the virtual device.

Description:

This is the constructor used to create an instance of a `vrtaActuator` device.

Return values:

Value	Description
`<none>`	

11.10.2 Method: GetMax()

Get the maximum value.

Method declaration:

```
unsigned GetMax(void)
```

Parameters:

Parameter	Input/Output	Description
`<none>`		

Description:

This method is used to get the maximum value of the actuator.

Return values:

Value	Description
`<a value>`	The maximum value of the actuator.

11.10.3 Method: Value()

Get the current value.

Method declaration:

```
unsigned Value(void)
```

Parameters:

Parameter	Input/Output	Description
`<none>`		

Description:

This method is used to get the current value of the actuator.

Return values:

Value	Description
`<a value>`	The current value of the actuator.

11.10.4 Method: SetMax()

Set the maximum value.

Method declaration:

```
void SetMax(unsigned v)
```

Parameters:

Parameter	Input/Output	Description
`v`	Input	The new maximum value for the actuator.

Description:

This method is used to set the maximum value of the actuator. If the current value of the actuator is greater than the new maximum value then the current value is set to zero.

Return values:

Value	Description
`<none>`	

11.10.5 Method: SetVal()

Set the current value.

Method declaration:

```
void SetVal(unsigned v)
```

Parameters:

Parameter	Input/Output	Description
`v`	Input	The new value for the actuator.

Description:

This method is used to set the current value of the actuator. If the new current value of the actuator is greater than the maximum value then the current value is not set.

Return values:

Value	Description
`<none>`	

11.10.6 Action: Value

Action ID	Data Format
1	`%u`

This action sets the current value of the actuator. It is the equivalent of the `SetVal()` method.

11.10.7 Action: Maximum

Action ID	Data Format
2	`%u`

This action sets the maximum value of the actuator. It is the equivalent of the `SetMax()` method.

11.10.8 Event: Value

Event ID	Data Format
1	`%u`

This event contains the current value of the actuator. It is raised whenever the value of the actuator changes.

11.10.9 Event: Maximum

Event ID	Data Format
2	`%u`

This event contains the maximum value of the actuator. It is raised whenever the maximum value of the actuator changes.

11.11 vrtaActuatorLight

A `vrtaActuatorLight` is a special form of an actuator that represents a light. A `vrtaActuatorLight` has two possible values zero and one, representing “off” and “on”.

When a `vrtaActuatorLight` is created its current value is zero.

11.11.1 Method: vrtaActuatorLight()

The constructor.

Method declaration:

```
vrtaActuatorLight(const vrtaTextPtr name)
```

Parameters:

Parameter	Input/Output	Description
`name`	Input	The name of the virtual device.

Description:

This is the constructor used to create an instance of a `vrtaActuatorLight` device.

Return values:

Value	Description
`<none>`	

11.11.2 Method: Value()

Get the current value.

Method declaration:

```
unsigned Value(void)
```

Parameters:

Parameter	Input/Output	Description
`<none>`		

Description:

This method is used to get the current value of the actuator.

Return values:

Value	Description
`<a value>`	The current value of the actuator.

11.11.3 Method: SetVal()

Set the current value.

Method declaration:

```
void SetVal(unsigned v)
```

Parameters:

Parameter	Input/Output	Description
`v`	Input	The new value for the actuator.

Description:

This method is used to set the current value of the actuator. If the new current value of the actuator is greater than 1 then the current value is not set.

Return values:

Value	Description
`<none>`	

11.11.4 Action: Value

Action ID	Data Format
`1`	`%u`

This action sets the current value of the actuator. It is the equivalent of the `SetVal()` method.

11.11.5 Event: Value

Event ID	Data Format
`1`	`%u`

This event contains the current value of the actuator. It is raised whenever the value of the actuator changes.

11.12 vrtaActuatorDimmableLight

A `vrtaActuatorDimmableLight` is a special form of an actuator that represents a light whose brightness can be set. The number of possible brightness levels is set when the actuator is created (but can be changed later).

When a `vrtaActuatorDimmableLight` is created its current value is zero.

11.12.1 Method: vrtaActuatorDimmableLight()

The constructor.

Method declaration:

```
vrtaActuatorDimmableLight(const vrtaTextPtr name,
    unsigned levels)
```

Parameters:

Parameter	Input/Output	Description
`name`	Input	The name of the virtual device.
`levels`	Input	The number of brightness levels minus 1

Description:

This is the constructor used to create an instance of a `vrtaActuatorDimmableLight` device. The actuator may have a value in the range zero to `levels` inclusive.

Return values:

Value	Description
`<none>`	

11.12.2 Method: GetMax()

Get the maximum value.

Method declaration:

```
unsigned GetMax(void)
```

Parameters:

Parameter	Input/Output	Description
`<none>`		

Description:

This method is used to get the maximum value of the actuator.

Return values:

Value	Description
`<a value>`	The maximum value of the actuator.

11.12.3 Method: Value()

Get the current value.

Method declaration:

```
unsigned Value(void)
```

Parameters:

Parameter	Input/Output	Description
`<none>`		

Description:

This method is used to get the current value of the actuator.

Return values:

Value	Description
`<a value>`	The current value of the actuator.

11.12.4 Method: SetMax()

Set the maximum value.

Method declaration:

```
void SetMax(unsigned v)
```

Parameters:

Parameter	Input/Output	Description
`v`	Input	The new maximum value for the actuator.

Description:

This method is used to set the maximum value of the actuator (i.e. to override the value of the `levels` argument used in the constructor). If the current value of the actuator is greater than the new maximum value then the current value is set to zero.

Return values:

Value	Description
`<none>`	

11.12.5 Method: SetVal()

Set the current value.

Method declaration:

```
void SetVal(unsigned v)
```

Parameters:

Parameter	Input/Output	Description
`v`	Input	The new value for the actuator.

Description:

This method is used to set the current value of the actuator. If the new current value of the actuator is greater than the maximum value then the current value is not set.

Return values:

Value	Description
`<none>`	

11.12.6 Action: Value

Action ID	Data Format
1	`%u`

This action sets the current value of the actuator. It is the equivalent of the `SetVal()` method.

11.12.7 Action: Maximum

Action ID	Data Format
2	`%u`

This action sets the maximum value of the actuator. It is the equivalent of the `SetMax()` method.

11.12.8 Event: Value

Event ID	Data Format
1	`%u`

This event contains the current value of the actuator. It is raised whenever the value of the actuator changes.

11.12.9 Event: Maximum

Event ID	Data Format
2	`%u`

This event contains the maximum value of the actuator. It is raised whenever the maximum value of the actuator changes.

11.13 vrtaActuatorMultiColorLight

A `vrtaActuatorMultiColorLight` is a special form of an actuator that represents a light whose color can be set. The number of possible colors is set when the actuator is created (but can be changed later).

When a `vrtaActuatorMultiColorLight` is created its current value is zero.

11.13.1 Method: vrtaActuatorMultiColorLight()

The constructor.

Method declaration:

```
vrtaActuatorMultiColorLight(const vrtaTextPtr name,
    unsigned colors)
```

Parameters:

Parameter	Input/Output	Description
`name`	Input	The name of the virtual device.
`colors`	Input	The number of colors minus 1

Description:

This is the constructor used to create an instance of a `vrtaActuatorMultiColorLight` device. The actuator may have a value in the range zero to `colors` inclusive.

Return values:

Value	Description
`<none>`	

11.13.2 Method: GetMax()

Get the maximum value.

Method declaration:

```
unsigned GetMax(void)
```

Parameters:

Parameter	Input/Output	Description
`<none>`		

Description:

This method is used to get the maximum value of the actuator.

Return values:

Value	Description
`<a value>`	The maximum value of the actuator.

11.13.3 Method: Value()

Get the current value.

Method declaration:

```
unsigned Value(void)
```

Parameters:

Parameter	Input/Output	Description
`<none>`		

Description:

This method is used to get the current value of the actuator.

Return values:

Value	Description
`<a value>`	The current value of the actuator.

11.13.4 Method: SetMax()

Set the maximum value.

Method declaration:

```
void SetMax(unsigned v)
```

Parameters:

Parameter	Input/Output	Description
`v`	Input	The new maximum value for the actuator.

Description:

This method is used to set the maximum value of the actuator (i.e. to override the value of the `colors` argument used in the constructor). If the current value of the actuator is greater than the new maximum value then the current value is set to zero.

Return values:

Value	Description
`<none>`	

11.13.5 Method: SetVal()

Set the current value.

Method declaration:

```
void SetVal(unsigned v)
```

Parameters:

Parameter	Input/Output	Description
`v`	Input	The new value for the actuator.

Description:

This method is used to set the current value of the actuator. If the new current value of the actuator is greater than the maximum value then the current value is not set.

Return values:

Value	Description
`<none>`	

11.13.6 Action: Value

Action ID	Data Format
1	`%u`

This action sets the current value of the actuator. It is the equivalent of the `SetVal()` method.

11.13.7 Action: Maximum

Action ID	Data Format
2	`%u`

This action sets the maximum value of the actuator. It is the equivalent of the `SetMax()` method.

11.13.8 Event: Value

Event ID	Data Format
1	`%u`

This event contains the current value of the actuator. It is raised whenever the value of the actuator changes.

11.13.9 Event: Maximum

Event ID	Data Format
2	`%u`

This event contains the maximum value of the actuator. It is raised whenever the maximum value of the actuator changes.

11.14 vrtaCompare

A `vrtaCompare` device represents a comparator that may be attached to any of the following devices

```
vrtaUpCounter
vrtaDownCounter
vrtaSensor
vrtaSensorToggleSwitch
vrtaSensorMultiwaySwitch
vrtaActuator
vrtaActuatorLight
vrtaActuatorDimmableLight
vrtaActuatorMultiColorLight
```

It will generate an interrupt when the current value of the attached device reaches a specified match value.

Multiple `vrtaCompare` devices may be attached to the same device.

11.14.1 Method: vrtaCompare()

The constructor.

Method declaration:

```
vrtaCompare(const vrtaTextPtr name,
    vrtaComparable &source,
    unsigned match,
    unsigned vector)
```

Parameters:

Parameter	Input/Output	Description
`name`	Input	The name of the virtual device.
`source`	Input	The device to which to attach.
`match`	Input	The match value.
`vector`	Input	The interrupt vector number to be generated or zero for no interrupt.

Description:

This is the constructor used to create an instance of a `vrtaCompare` device. The `vrtaCompare` device will raise interrupt number `vector` when the current value of the device specified by `source` reaches the value `match`. (Note that `vrtaCompare` will not enable the interrupt vector. This must be done by sending an Unmask action to the ICU device.)

Return values:

Value	Description
`<none>`	

11.14.2 Method: GetMatch()

Get the match value.

Method declaration:

```
unsigned GetMatch(void)
```

Parameters:

Parameter	Input/Output	Description
`<none>`		

Description:

This method is used to get the current match value of the device.

Return values:

Value	Description
`<a value>`	The current match value.

11.14.3 Method: SetMatch()

Set the match value.

Method declaration:

```
void SetMatch(unsigned val)
```

Parameters:

Parameter	Input/Output	Description
`val`	Input	The new match value.

Description:

This method is used to set the match value.

Return values:

Value	Description
`<none>`	

11.14.4 Method: IncrementMatch()

Increment the match value.

Method declaration:

```
unsigned IncrementMatch(unsigned val)
```

Parameters:

Parameter	Input/Output	Description
`val`	Input	The amount by which the match value should be incremented.

Description:

This method is used to increment the match value.

Return values:

Value	Description
`<a value>`	The new match value.

11.14.5 Method: SetVector()

Set the interrupt vector number.

Method declaration:

```
void SetVector(unsigned val)
```

Parameters:

Parameter	Input/Output	Description
`val`	Input	The new interrupt vector number.

Description:

This method is used to set the interrupt vector number. If the interrupt vector number is set to zero then no interrupted will be generated.

Return values:

Value	Description
`<none>`	

11.14.6 Action: Match

Action ID	Data Format
`1`	`%u`

This action sets the match value. It is the equivalent of the `SetMatch()` method.

11.14.7 Action: Vector

Action ID	Data Format
`2`	`%u`

This action sets the interrupt vector number. It is the equivalent of the `SetVector()` method.

11.14.8 Event: Match

Event ID	Data Format
`1`	`%u`

This event contains the match value. It is raised whenever the current value of the attached device reaches the match value of the `vrtaCompare` device.

11.15 vrtaIO

A `vrtaIO` device represents an array of 32-bit I/O cells that may be written and read by an application.

11.15.1 Method: vrtaIO()

The constructor.

Method declaration:

```
vrtaIO(const vrtaTextPtr name, unsigned elements)
```

Parameters:

Parameter	Input/Output	Description
`name`	Input	The name of the virtual device.
`elements`	Input	The number of I/O cells to be used.

Description:

This is the constructor used to create an instance of a `vrtaIO` device. The `vrtaIO` device will contain an array of `elements` I/O cells. The I/O cells will have offsets in the range zero to `elements` – 1 inclusive.

Return values:

Value	Description
`<none>`	

11.15.2 Method: SetValue()

Set the value of an I/O cell.

Method declaration:

```
void SetValue(unsigned offset, unsigned value)
```

Parameters:

Parameter	Input/Output	Description
`offset`	Input	The offset of the I/O cell to be set.
`value`	Input	The value to write.

Description:

This method is used to set the value of an I/O cell.

Return values:

Value	Description
`<none>`	

11.15.3 Method: SetValues()

Set the value of multiple I/O cells.

Method declaration:

```
void SetValues(unsigned offset,
    const unsigned *values,
    unsigned number)
```

Parameters:

Parameter	Input/Output	Description
`offset`	Input	The offset of the first I/O cell to be set.
`values`	Input	An array of values to write.
`number`	Input	The number of values to write.

Description:

This method is used to set the values of multiple I/O cells. The `number` values from `values[]` are written to the array of I/O cells starting at `offset`.

Return values:

Value	Description
`<none>`	

11.15.4 Method: GetValue()

Get the value of an I/O cell.

Method declaration:

```
unsigned GetValue(unsigned offset) const
```

Parameters:

Parameter	Input/Output	Description
`offset`	Input	The offset of the I/O cell get.

Description:

This method is used to get the value of an I/O cell.

Return values:

Value	Description
`<a value>`	The value of the specified I/O cell.

11.15.5 Method: GetValues()

Get the values of all I/O cells.

Method declaration:

```
const unsigned *GetValues(void) const
```

Parameters:

Parameter	Input/Output	Description
`<none>`		

Description:

This method is used to get the values of all I/O cells.

Return values:

Value	Description
`<a pointer>`	A pointer to the array of I/O cells.

11.15.6 Action: Value

Action ID	Data Format
1	`%u,%u`

This action sets the value of an I/O cell. The first number in the action data is the I/O cell offset and the second number is the value to write. This action is equivalent to the `SetValue()` method.

11.15.7 Action: Values

Action ID	Data Format
2	`%a`

This action sets the values of multiple I/O cells. The action data is an array of values to write to the I/O cell array starting at offset zero. This action is equivalent to the `SetValues()` method with the `offset` argument set to zero.

11.15.8 Action: GetValue

Action ID	Data Format
3	`%u`

This action causes a Value event to be raised for the offset specified in the action data.

11.15.9 Action: GetValues

Action ID	Data Format
4	`<none>`

This action causes a Values event to be raised.

11.15.10 Event: Value

Event ID	Data Format
1	`%u,%u(%u)`

This event contains the value of an I/O cell. The first number is the offset of the I/O cell. The second number is the value of the I/O cell. If the event is raised in response to a Value action then the I/O cell offset is in the action data. If the event is queried then the I/O cell offset is in the input data.

11.15.11 Event: Values

Event ID	Data Format
2	`%a`

This event contains the values of all of the I/O cells.

12 Virtual ECU Server Library Reference

The Virtual ECU Server library is a library that provides a C programming API for communication with `vrtaServer`. This chapter describes, in alphabetic order, the API calls provided by this library.

12.1 Using the Source Code

The server library is provided in source code form in the file `interfaces\VESLib\veslib.cpp`. You will also need the support file `interfaces\vrtaClientSupport.cpp`. Any source files that wish to use the library should include the `veslib.h` header file.

12.2 Using the DLL

The server library is also provided as a DLL called `veslib.dll`. An import library is not provided for the DLL since the format of import libraries varies between compilers. If you wish to use `veslib.dll` you will need to use the Windows `LoadLibrary()` function to load the DLL and then the Windows `GetProcAddress()` function to get pointers to the library API functions within the DLL. Prototypes for all of the library API functions can be found in `veslib.h`.

Any source files that wish to use the library should include the `veslib.h` header file.

`veslib.dll` was produced by compiling `veslib.cpp` with the C++ macro `VESLIB_DLL` defined.

12.3 Virtual ECU Aliases

`vrtaServer` makes use of *aliases* to keep track of Virtual ECUs. When a Virtual ECU registers with `vrtaServer` it is assigned an alias. The default alias for a VECU is simply the name of its executable. However if multiple VECUs with the same executable name register with `vrtaServer` then a numeric suffix is applied to the executable name to generate a unique alias. For example, if a VECU with the executable `vecu.exe` is loaded then it will be assigned the alias "`vecu.exe`". If a second instance of `vecu.exe` is loaded then it will be assigned the alias "`vecu.exe_2`".

It is also possible for a VECU's alias to be set on the command line of the VECU by using the "`-alias`" command line option. Again `vrtaServer` will apply numeric suffixes to the specified aliases to ensure that all aliases are unique.

12.4 VesLibEcuInfoType

The `VesLibEcuInfoType` type is used to identify Virtual ECU executables to the server library. `VesLibEcuInfoType` has the definition:

```
typedef struct {
    char path[VESLIB_MAX_PATH];
} VesLibEcuInfoType;
```

The `path` field should contain the full path of the Virtual ECU executable.

12.5 VesLibEcuAliasType

The `VesLibEcuAliasType` type is used to contain Virtual ECU aliases. `VesLibEcuAliasType` has the definition:

```
typedef struct {
    char name[VESLIB_MAX_PATH];
} VesLibEcuAliasType;
```

12.6 The API Call Template

Each API call is described in this chapter using the following standard format:

The title gives the name of the API call.

A brief description of the API call is provided.

Function declaration:

Interface in C syntax.

Parameters:

Parameter	Input/Output	Description
`Parameter Name`	Input/Output	Description.

Description:

Explanation of the functionality of the API call.

Return values:

Value	Description
`Return values.`	Description of return value.

Notes:

Usage restrictions and notes for the API call.

See also:

List of related API calls.

12.7 VesLibAttachToECU()

Attach to a loaded Virtual ECU.

Function declaration:

```
VesLibStatusType VesLibAttachToECU(
    VesLibEcuAliasType * alias, int * port)
```

Parameters:

Parameter	Input/Output	Description
`alias`	Input	A pointer to a `VesLibEcuAliasType` structure specifying the alias of a loaded Virtual ECU.
`port`	Output	A pointer to a variable that on successful return will contain the TCP port number of the Virtual ECU's diagnostic port.

Description:

This function is used to connect to a Virtual ECU that has already been loaded. On successful return `*port` contains the port number of the Virtual ECU's diagnostic interface.

Return values:

Value	Description
VESLIB_STATUS_OK	The API call succeeded.
VESLIB_STATUS_SERVER_COMMS	The library cannot communicate with `vrtaServer`.
VESLIB_STATUS_NO_ECU	The alias does not exist.
VESLIB_STATUS_ECU_NOT_LOADED	The alias is not loaded.

Notes:

See also:

`VesLibListLoadedECUs()`, `VesLibLoadECU()`

12.8 VesLibCreateAlias()

Create an alias for a Virtual ECU.

Function declaration:

```
VesLibStatusType VesLibCreateAlias(
    VesLibEcuInfoType * ecu,
    VesLibEcuAliasType * alias)
```

Parameters:

Parameter	Input/Output	Description
`ecu`	Input	A pointer to a `VesLibEcuInfoType` structure that identifies the Virtual ECU executable.
`alias`	Output	A pointer to a `VesLibEcuAliasType` structure that will contain the new alias.

Description:

This API call creates an alias for a Virtual ECU. `ecu` identifies a Virtual ECU executable. `alias` points to a `VesLibtEcuAliasType` structure allocated by the caller. On successful return `*alias` contains a new alias for the Virtual ECU. The reference count for the alias will have been set to 1.

Return values:

Value	Description
VESLIB_STATUS_OK	The API call succeeded.
VESLIB_STATUS_SERVER_COMMS	The library cannot communicate with `vrtaServer`.
VESLIB_STATUS_NO_ECU	The Virtual ECU executable does not exist.

Notes:

See also:

`VesLibFreeAlias()`, `VesLibLoadEcu()`

12.9 VesLibExit()

Shutdown the library.

Function declaration:

```
VesLibStatusType VesLibExit(void)
```

Parameters:

Parameter	Input/Output	Description
`<none>`		

Description:

This API call is used to disconnect from `vrtaServer` and release any resources allocated within `VesLibInitialize()` and subsequent API calls (but it does not release any memory that should have been freed via `VesLibFreeMemory()`).

`VesLibInitialize()` can be called subsequently to re-attach to `vrtaServer`.

Return values:

Value	Description
VESLIB_STATUS_OK	The API call succeeded.

Notes:

See also:

`VesLibSelectServer(), vrtaLibInitialize()`

12.10 VesLibFindECUs()

Find out what Virtual ECU executables are present.

Function declaration:

```
VesLibStatusType VesLibFindECUs(char * dir,
    char * * results)
```

Parameters:

Parameter	Input/Output	Description
`dir`	Input	The path of the directory to be searched for Virtual ECU executables.
`results`	Output	A pointer to a variable which on successful return will point to a list of Virtual ECU executables.

Description:

This API call is used to discover the Virtual ECU executables present on the local PC (or remote PC if `VesLibSelectServer()` has been used to select `vrtaServer` running on a remote PC). `dir` contains the path of the directory to be searched – either as an absolute path or a path relative to the directory containing the `vrtaServer` executable. On successful return `*results` points to a '`\n`' separated list of the Virtual ECUs executables found in the specified directory plus directory information that allows remote navigation of the directories available on `vrtaServer`'s PC.

On successful return the '`\n`' separated list pointed to by `*results` contains the following items in the order specified below:

1. The current path.
2. A comma separated list of drives that are readable.
3. The subdirectories of `dir` (one per line), including `".."` for a non-root directory.
4. A blank line.
5. A list of files in `dir` that may be valid Virtual ECU applications.

Return values:

Value	Description
VESLIB_STATUS_OK	The API call succeeded.
VESLIB_STATUS_SERVER_COMMS	The library cannot communicate with `vrtaServer`.

Notes:

`*results` points to memory allocated by the server library. The memory should be released by calling `VesLibFreeMemory()`.

See also:

`VesLibFreeMemory()`

12.11 VesLibFreeAlias()

Free a Virtual ECU alias.

Function declaration:

```
VesLibStatusType VesLibFreeAlias(
    VesLibEcuAliasType * alias)
```

Parameters:

Parameter	Input/Output	Description
`alias`	Input	A pointer to a `VesLibEcuAliasType` structure specifying an alias.

Description:

This API call decrements the reference count of the specified Virtual ECU alias. An alias is removed when its reference count reaches zero. If the application that is using the library terminates without freeing aliases then `vrtaServer` will automatically decrement the reference counts of aliases appropriately.

Return values:

Value	Description
VESLIB_STATUS_OK	The API call succeeded.
VESLIB_STATUS_SERVER_COMMS	The library cannot communicate with `vrtaServer`.
VESLIB_STATUS_NO_ECU	The alias does not exist.

Notes:

See also:

`VesLibCreateAlias()`, `VesLibGetAliases()`, `VesLibListAliases()`, `VesLibListLoadedECUs()`

12.12 VesLibFreeMemory()

Free memory allocated by the server library.

Function declaration:

```
void VesLibFreeMemory(void * results)
```

Parameters:

Parameter	Input/Output	Description
`results`	Input	A pointer the memory to be freed

Description:

This API call is used to free memory returned from other library functions.

Return values:

Value	Description
<none>	

Notes:

See also:

`VesLibFindECUs()`, `VesLibGetAliases()`, `VesLibListAliases()`, `VesLibListLoadedECUs()`, `VesLibGetInfo()`

12.13 VesLibGetAliases()

Get a list of the aliases that exist for a Virtual ECU executable.

Function declaration:

```
VesLibStatusType VesLibGetAliases(
    VesLibEcuInfoType * ecu,
    VesLibEcuAliasType * * results,
    int * count)
```

Parameters:

Parameter	Input/Output	Description
`ecu`	Input	A pointer to a `VesLibEcuInfoType` structure that identifies the Virtual ECU executable.
`results`	Output	A pointer to a variable which on successful return will point to a list of `VesLibEcuAliasType` structures containing the aliases for the specified Virtual ECU executable.
`count`	Output	A pointer to a variable which on successful return will contain the number of aliases in `*results`.

Description:

This API call gets a list of all the aliases that exists for a specified Virtual ECU executable. The aliases may have been created explicitly with `VesLibCreateAlias()` or have been created when the Virtual ECU registered with `vrtaServer`. `ecu` identifies the Virtual ECU executable. On successful return `*results` points to an array of `VesLibEcuAliasType` structures containing all aliases for the Virtual ECU executable and `*count` contains the number of aliases in the array. If no alias exists for the specified Virtual ECU executable, one will be created. The reference count of each alias returned is incremented by 1.

Return values:

Value	Description
VESLIB_STATUS_OK	The API call succeeded.
VESLIB_STATUS_SERVER_COMMS	The library cannot communicate with `vrtaServer`.
VESLIB_STATUS_NO_ECU	The Virtual ECU executable does not exist.

Notes:

`*results` points to memory allocated by the library. The memory should be released by calling `VesLibFreeMemory()`.

See also:

`VesLibFreeMemory()`, `VesLibFreeAlias()`, `VesLibLoadEcu()`, `VesLibListAliases()`

12.14 VesLibGetInfo()

Get version information about a Virtual ECU.

Function declaration:

```
VesLibStatusType VesLibGetInfo(
    VesLibEcuAliasType * alias, char * * results)
```

Parameters:

Parameter	Input/Output	Description
`alias`	Input	A pointer to a `VesLibEcuAliasType` structure specifying an alias.
`results`	Output	A pointer to a variable which on successful return will point to version information.

Description:

This API call returns version information about the specified Virtual ECU alias. On successful return `*results` points to a '`\n`' separated list containing version number information as a series of "`key=value`" pairs.

Return values:

Value	Description
VESLIB_STATUS_OK	The API call succeeded.
VESLIB_STATUS_SERVER_COMMS	The library cannot communicate with `vrtaServer`.
VESLIB_STATUS_NO_ECU	The alias does not exist.

Notes:

`*results` points to memory allocated by the library. The memory should be released by calling `VesLibFreeMemory()`.

See also:

12.15 VesLibInitialize()

Initialize the library.

Function declaration:

```
VesLibStatusType VesLibInitialize(void)
```

Parameters:

Parameter	Input/Output	Description
`<none>`		

Description:

This API call is used to prepare the library for use. This API must be called before all other API calls except for `VesLibSelectServer()`.

By default the server library communicates with `vrtaServer` running on the local PC. This can be changed by calling `VesLibSelectServer()`.

If `vrtaServer` is not already running on the selected PC then the server library will attempt to start `vrtaServer` as a service on the selected PC when `VesLibInitialize()` is called. This will only succeed if `vrtaServer` has been installed as a service on the selected PC.

Return values:

Value	Description
VESLIB_STATUS_OK	The API call succeeded.
VESLIB_STATUS_SERVER_START	`vrtaServer` cannot be started.
VESLIB_STATUS_SERVER_COMMS	The library cannot communicate with `vrtaServer`.

Notes:

See also:

`VesLibSelectServer()`, `vrtaLibExit()`

12.16 VesLibListAliases()

Get a list of all aliases that exist.

Function declaration:

```
VesLibStatusType VesLibListAliases(
    VesLibEcuAliasType * * results,
    int * count)
```

Parameters:

Parameter	Input/Output	Description
`results`	Output	A pointer to a variable which on successful return will point to a list of `VesLibEcuAliasType` structures containing all aliases.
`count`	Output	A pointer to a variable which on successful return will contain the number of aliases in `*results`.

Description:

This API call gets a list of all aliases that have been created. The aliases may have been created explicitly with `VesLibCreateAlias()` or have been created when Virtual ECUs registered with `vrtaServer`. On successful return `*results` points to an array of `VesLibEcuAliasType` structures containing all aliases that have been created and `*count` contains the number of aliases in the array. The reference count of each alias returned is incremented by 1.

Return values:

Value	Description
VESLIB_STATUS_OK	The API call succeeded.
VESLIB_STATUS_SERVER_COMMS	The library cannot communicate with `vrtaServer`.

Notes:

`*results` points to memory allocated by the library. The memory should be released by calling `VesLibFreeMemory()`.

See also:

`VesLibFreeMemory()`, `VesLibFreeAlias()`, `VesLibLoadEcu()`, `VesLibGetAliases()`

12.17 VesLibListLoadedECUs()

Get a list of the Virtual ECUs that have been loaded.

Function declaration:

```
VesLibStatusType VesLibListLoadedECUs(
    VesLibEcuAliasType * * results,
    int * count)
```

Parameters:

Parameter	Input/Output	Description
`results`	Output	A pointer to a variable which on successful return will point to a list of `VesLibEcuAliasType` structures containing the aliases of loaded Virtual ECUs.
`count`	Output	A pointer to a variable which on successful return will contain the number of aliases in `*results`.

Description:

This API call is used to discover the aliases of Virtual ECUs that have been loaded (i.e. the Virtual ECU executables are running). On successful return `*results` points to an array of `VesLibEcuAliasType` structures containing the aliases of all loaded Virtual ECUs and `*count` contains the number of aliases in the array. The reference count of each alias returned is incremented by 1.

Return values:

Value	Description
VESLIB_STATUS_OK	The API call succeeded.
VESLIB_STATUS_SERVER_COMMS	The library cannot communicate with `vrtaServer`.

Notes:

`*results` points to memory allocated by the library. The memory should be released by calling `VesLibFreeMemory()`.

See also:

`VesLibFreeMemory()`, `VesLibFreeAlias()`, `VesLibLoadEcu()`

12.18 VesLibLoadECU()

Load a Virtual ECU.

Function declaration:

```
VesLibStatusType VesLibLoadECU(
    VesLibEcuAliasType * alias,
    VesLibStartMode startMode,
    VesLibDisplayMode displayMode,
    char * cmd,
    int * port)
```

Parameters:

Parameter	Input/Output	Description
`alias`	Input	A pointer to a `VesLibEcuAliasType` structure specifying the alias of the Virtual ECU executable to load.
`startMode`	Input	The start mode for the Virtual ECU.
`displayMode`	Input	The display mode for the Virtual ECU.
`cmd`	Input	The command line for the Virtual ECU.
`port`	Output	A pointer to a variable that on successful return will contain the TCP port number of the Virtual ECU's diagnostic port.

Description:

This API call is used to load and connect to a Virtual ECU alias specified by `alias` (i.e. to run the Virtual ECU executable identified by `alias`). If `startMode` is `VesLibSMAuto` then the alias is loaded in autostart mode. If `startMode` is `VesLibSMSlave` then the alias is loaded in slave mode. If `displayMode` is `VesLibDMSilent` then the alias is loaded in silent mode. If `displayMode` is `VesLibDMGui` then the alias is loaded in GUI mode. `cmd` specifies additional command line parameters. On successful return `*port` contains the port number of the Virtual ECU's diagnostic interface.

Return values:

Value	Description
VESLIB_STATUS_OK	The API call succeeded.
VESLIB_STATUS_SERVER_COMMS	The library cannot communicate with `vrtaServer`.
VESLIB_STATUS_NO_ECU	The alias does not exist.
VESLIB_STATUS_ECU_LOADED	The alias is already loaded.

Notes:

See also:

`VesLibCreateAlias()`, `VesLibListLoadedECUs()`, `VesLibAttachToECUs()`

12.19 VesLibSelectServer()

Select the `vrtaServer` to use.

Function declaration:

```
VesLibStatusType VesLibSelectServer(const char *host,
    int port)
```

Parameters:

Parameter	Input/Output	Description
`host`	Input	The host name of the PC running `vrtaServer`.
`port`	Input	The number of the TCP port being used by `vrtaServer`.

Description:

The server library is normally used to communicate with `vrtaServer` running on the local PC. This API call can direct the server library to communicate with `vrtaServer` running on a remote PC by passing the hostname of the remote PC as `host`. If `host` is NULL then the server library will communicate with `vrtaServer` running on the local PC (it internally defaults `host` to "localhost").

Similarly `vrtaServer` is normally found by searching three pre-defined TCP port numbers. If `vrtaServer` is set to use a different port number then you can specify this number with the `port` argument. If `port` is zero then the server library will search for `vrtaServer` on the pre-defined port numbers.

`VesLibSelectServer()` should normally be called before `VesLibInitialize()`. If it is called after `VesLibInitialize()` but before `VesLibExit()` then the function will itself call `VesLibExit()` then `VesLibInitialize()` to reset the connection.

Return values:

Value	Description
VESLIB_STATUS_OK	The API call succeeded.
VESLIB_STATUS_SERVER_START	`vrtaServer` cannot be started.
VESLIB_STATUS_SERVER_COMMS	The library cannot communicate with `vrtaServer`.

Notes:

See also:

`VesLibInitialize()`, `VesLibExit()`

13 COM Bridge Tutorial and Reference

The COM Bridge provides services to COM enabled clients such as Microsoft Visual Basic to allow them to interact with Virtual ECUs. The COM Bridge translates between COM protocols on the client-side and the TCP/IP protocols used by `vrtaServer` and VECUs.

This chapter describes the objects and interfaces provided by the COM Bridge.

Important: This chapter is not intended as a tutorial on COM. It is assumed that the reader is familiar with COM programming.

13.1 Overview

The COM Bridge is implemented as a DLL COM Server. It will be loaded into the process of any client wishing to use it. The DLL is `vrtaMSCOM.dll`.

The COM objects hosted by the COM Bridge support dual interfaces to enable access from as wide a variety of COM clients as possible. The COM Bridge can create worker threads and therefore the objects hosted by the COM Bridge use the multi-threaded apartment (MTA).

13.2 Sample of use

The section shows code snippets written in Microsoft Visual Basic that demonstrate how to use the COM Bridge.

13.2.1 CVcServer

The `CVcServer` object is used to connect to vrtaServer so that you can load or attach to VECUs.

The code below shows how to create a `CVcServer` named `local_Server`, connect to the local vrtaServer, create an alias for the VECU named by `ourexe`, then load it.

```
' Declare a server component
Private local_Server As CVcServer

' Create the server component in form load
Private Sub Form_Load()
    Set local_Server = New CVcServer
    < ... snip ...>
End Sub

' Release the server component in form unload
Private Sub Form_Unload(Cancel As Integer)
    Set local_Server = Nothing
    < ... snip ...>
End Sub
```

```
' A helper function that checks the server component
' return codes
Private Sub CheckServerStatus(location, val)
    If val = STATUS_OK Then
        Exit Sub
    ElseIf val = SERVER_START Then
        ret = "SERVER_START"
    ElseIf val = SERVER_COMMS Then
        ret = "SERVER_COMMS"
    ElseIf val = NO_ECU Then
        ret = "NO_ECU"
    ElseIf val = ECU_LOADED Then
        ret = "ECU_LOADED"
    ElseIf val = ECU_NOT_LOADED Then
        ret = "ECU_NOT_LOADED"
    ElseIf val = ECU_SLAVE Then
        ret = "ECU_SLAVE"
    ElseIf val = ECU_ALIASED Then
        ret = "ECU_ALIASED"
    ElseIf val = NOT_LOADED Then
        ret = "NOT_LOADED"
    Else
        ret = "** UNKNOWN **"
    End If
    AddLine (location + " return status: " + ret)
    AddLine ("Test failed")

    ' Quit program
    Unload fTest
End Sub

Private Sub DoSomething()

    Call CheckServerStatus(
        "Connect to server",
        local_Server.Connect("localhost", 0)
    )

    Dim ouralias As String
    Call CheckServerStatus(
        "Create alias",
        local_Server.CreateAlias(ourexe, ouralias)
    )

    Dim diagport As Long
    Call CheckServerStatus("LoadECU",
        local_Server.LoadECU(
              ouralias, AUTO, GUI, "", diagport)
    )

    < ... snip ...>

    Call CheckServerStatus("FreeAlias",
        local_Server.FreeAlias(ouralias)
    )
    local_Server.Disconnect
End Sub
```

13.2.2 CVcECU

The `CVcECU` object is used to connect to a VECU so that you can interact with its devices.

The code below shows how to create a `CVcECU` named `monitored_ECU`, connect it to a VECU, hook and display an event then terminate the VECU.

```
' Declare an ECU component that has events
Private WithEvents monitored_ECU As CVcECU

' Create the ECU component in form load
Private Sub Form_Load()
    Set monitored_ECU = New CVcECU
    < ... snip ...>
End Sub

' Release the ECU component in form unload
Private Sub Form_Unload(Cancel As Integer)
    Set monitored_ECU = Nothing
    < ... snip ...>
End Sub

' A helper function that checks the ECU component
' return codes
Private Sub CheckECUStatus(location As String, val As
Integer)
    Dim ret As String

    If val = ECU_OK Then
        Exit Sub
    ElseIf val = ECU_DevErr Then
        ret = "ECU_DevErr"
    ElseIf val = ECU_IDErr Then
        ret = "ECU_IDErr"
    ElseIf val = ECU_ValErr Then
        ret = "ECU_ValErr"
    Else
        ret = "** UNKNOWN **"
    End If
    AddLine (location + " return status: " + ret)
    AddLine ("Test failed")
    ' Quit program
    Unload fTest
End Sub

' This gets called each time an event hooked by
' monitored_ECU gets raised in the VECU
Private Sub monitored_ECU_OnEventChange(ByVal dev As
Long, ByVal id As Long, ByVal value As String)
    hook_count = hook_count + 1
    AddLine (
       "** Device " + Str(dev) +
       ", Event " + Str(id) +
       ", Value " + value)
End Sub
```

```
' This waits for at least one event to be hooked
' It is only here for test purposes - normally we
' just allow events to arrive asynchronously
Private Sub WaitOnEvents()
    Do
        hook_count = 0
        PauseTime = 0.5    ' Set duration.
        Start = Timer    ' Set start time.
        Do While Timer < Start + PauseTime
            DoEvents    ' Yield to other processes.
        Loop
    Loop Until (hook_count = 0)
End Sub

Private Sub DoSomething()

    < ... snip ...>
    Call CheckECUStatus("Connect",
        monitored_ECU.Connect("localhost", diagport)
    )
    AddLine ("Loaded ok")

    Call CheckECUStatus("Hook ecu",
        monitored_ECU.Hook(
              a_device_ID,
              an_event_ID,
              1
        )
    )

    WaitOnEvents

    Call CheckECUStatus("Terminate",
        monitored_ECU.DoAction(2, 2)
    )     ' Terminate

    Call CheckECUStatus("Disconnect",
        monitored_ECU.Disconnect
    )

    < ... snip ...>
End Sub
```

13.2.3 CVcDevice, CVcAction and CVcEvent

The `CVcDevice`, `CVcAction` and `CVcEvent` components represent a VECU device, action and event respectively.

They cannot be created via 'New' like `CVcServer` or `CVcECU` because they must be bound to a parent `CVcECU` or `CVcDevice`.

A `CVcDevice` object is obtained by calling the `CVcECU`'s `GetDeviceByName` or `GetDeviceByID` method.

A `CVcAction` object is obtained by calling the `CVcDevice`'s `GetActionByName` or `GetActionByID` method.

A `CVcEvent` object is obtained by calling the `CVcDevice`'s `GetEventByName` or `GetEventByID` method.

The code below shows how to create these objects, hook and display events.

```
Private WithEvents monitored_Device As CVcDevice
Private monitored_Action As CVcAction
Private WithEvents monitored_Event As CVcEvent

' This gets called each time an event hooked by
' monitored_Device gets raised in the VECU
Private Sub monitored_Device_OnEventChange(ByVal id As
Long, ByVal value As String)
    hook_count = hook_count + 1
    AddLine (
        + Event " + Str(id) + ", Value " + value
    )
End Sub

' This gets called each time an event hooked by
' monitored_Event gets raised in the VECU
Private Sub monitored_Event_OnEventChange(ByVal value As
String)
    hook_count = hook_count + 1
    AddLine ("-- Value " + value)
End Sub

Private Sub DoSomething()
    < ... snip ...>

    Set monitored_Device =
        monitored_ECU.GetDeviceByName("Test")

    Set monitored_Action =
        monitored_Device.GetActionByName("f01")

    Set monitored_Event =
        monitored_Device.GetEventByName("f01")

    Call CheckECUStatus("Hook device",
        monitored_Device.Hook(
              monitored_Event.EventID, 1)
    )
    Call CheckECUStatus("Hook event",
        monitored_Event.Hook(1)
    )

    monitored_Action.Send ("1.01")
    monitored_Action.Send ("2.02")
    monitored_Action.Send ("3.03")
    monitored_Action.Send ("4.04")
    monitored_Action.Send ("5.05")

    WaitOnEvents

    < ... snip ...>

End Sub
```

13.3 A short tutorial

This tutorial gives an example of how to use the COM Bridge in Microsoft Visual Basic. You will create a simple application that can monitor and control a VECU. The example is developed using Visual Basic version 5.0, but should be easily transferable to later versions.

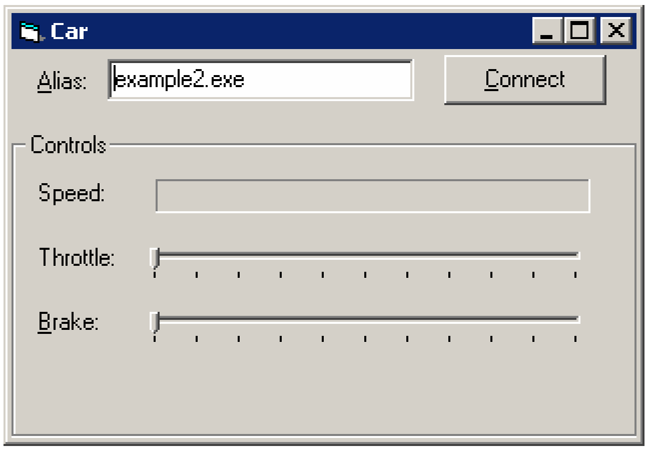
The tutorial creates a program that interacts with the 'Example2' VECU that ships with *RTA-OSEK for PC*. You should be able to find this in a location like `C:\rta\vrta\samples\Applications\RTA-OSEK Example 2`. The application is very simplistic: it will only attach to the VECU if it is already running, and there will be very little error handling done. This is because the main aim of the tutorial is to show you how to interface between Visual Basic and VECUs.

13.3.1 Setting up the project

Firstly create a new empty Visual Basic application, renaming `Form1` to `fCar` with the caption `Car`.

Ensure that you have ProgressBar and Slider components available (you may need to add the Microsoft Windows Common Controls to the Project Components), then use the screenshot below as a reference to:

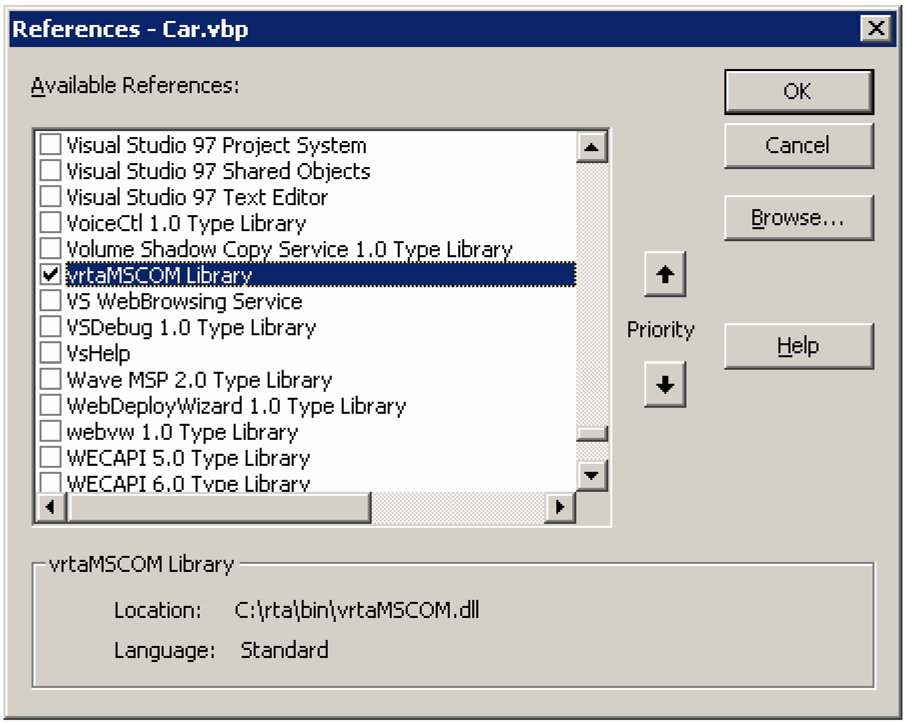
- add a TextBox `eAlias` with the default content 'Example2.exe'.
- add a Button `bConnect` alongside it.
- add a ProgressBar `pSpeed` with a range 0 to 100.
- add a Sliders `sThrottle` and `sBrake`, again with range 0 to 100.



Save the project as ‘Car.vbp’.

13.3.2 Connecting to vrtaServer

You will now add the code to connect to `vrtaServer`. Firstly ensure that the vrtaMSCOM Library is selected in the Project References:



- [ ] Visual Studio 97 Project System
- [ ] Visual Studio 97 Shared Objects
- [ ] Visual Studio 97 Text Editor
- [ ] VoiceCtl 1.0 Type Library
- [ ] Volume Shadow Copy Service 1.0 Type Library
- [✓] vrtaMSCOM Library
- [ ] VS WebBrowsing Service
- [ ] VSDebug 1.0 Type Library
- [ ] VsHelp
- [ ] Wave MSP 2.0 Type Library
- [ ] WebDeployWizard 1.0 Type Library
- [ ] webvw 1.0 Type Library
- [ ] WECAPI 5.0 Type Library
- [ ] WECAPI 6.0 Type Library

vrtaMSCOM Library

Location: C:\rta\bin\vrtaMSCOM.dll

Language: Standard

You now have access to the `CVcServer` object, so add the following lines to the project:

```
Private server As CVcServer
Private Sub Form_Load()
    Set server = New CVcServer
End Sub
Private Sub Form_Unload(Cancel As Integer)
    Set server = Nothing
End Sub
```

As you can see, `server` gets set to a new instance of `CVcServer` as the form loads. It gets released as the form unloads.

Before you can use `server` to access VECUs, you must connect it to `vrtaServer` running on a specified PC. For this example we will assume that `vrtaServer` is on the same PC ("`localhost`").

Add the 'connect' code to the button's Click event:

```
Private Sub bConnect_Click()
 If Not (STATUS_OK = server.Connect("localhost", 0))Then
   MsgBox ("Did not connect to VRTA Server")
   Exit Sub
 End If
 MsgBox ("Connected to VRTA Server")
End Sub
```

If you run the program at this point, then you should see the connection succeed. From this point on, `server` can be used to access and control VECUs on the local PC.

13.3.3 Connecting to the VECU

The `CVcECU` object is used to communicate with a specific VECU. You can obtain the details needed to use such an object via `CVcServer`.

Modify your code to add the declaration for `ecu`:

```
Private server As CVcServer
Private ecu As CVcECU
Private Sub Form_Load()
    Set server = New CVcServer
    Set ecu = New CVcECU
End Sub
Private Sub Form_Unload(Cancel As Integer)
    Set server = Nothing
    Set ecu = Nothing
End Sub
```

In the 'Connect' button event, add:

```
    Dim diagport As Long
    If Not (STATUS_OK = server.AttachECU(eAlias.Text,
diagport)) Then
        server.Disconnect
        MsgBox ("ECU is not running")
        Exit Sub
    End If

    If Not (ECU_OK = ecu.Connect("localhost", diagport))
Then
        server.Disconnect
        MsgBox ("Cannot connect to ECU")
        Exit Sub
    End If
```

The first clause asks `server` for the diagnostic port number of the VECU whose alias is the same as the text in `eAlias`. The connection will fail if there is no such VECU, so you will have to start `Example2.exe` before you can get much further.

The second clause simply binds `ecu` to the VECU by specifying the PC name and diagnostic port number.

Try it and see that it works as expected.

13.3.4 Initializing the devices

The next task is to create objects to link to the VECU's device Throttle, Brake and Speedometer devices.

They should be declared thus:

```
Private dSpeed As CVcDevice
Private aSpeed As CVcAction
Private WithEvents eSpeed As CVcEvent
Private dThrottle As CVcDevice
Private aThrottle As CVcAction
Private WithEvents eThrottle As CVcEvent
Private dBrake As CVcDevice
Private aBrake As CVcAction
Private WithEvents eBrake As CVcEvent
```

Now, when the `ecu` connects the application can read the current value of the Speedometer and initialize ProgressBar `pSpeed`. This is done with the code below, added to the bottom of the 'Connect' handler:

```
    Dim res As String
    Set dSpeed = ecu.GetDeviceByName("Speedometer")
    Set eSpeed = dSpeed.GetEventByName("Value")
    res = ""
    If ECU_OK = eSpeed.Query(res) Then
        pSpeed.value = res
    End If
```

This shows that `dSpeed` gets bound to the Speedometer device, and `eSpeed` gets bound to its Value event. `eSpeed.Query()` takes an in/out String value. This must be empty on entry because the VECU knows that no data should be passed 'in' to this event. `eSpeed.Query()` returns its result in the String `res`. This String can be passed directly in to the ProgressBar's value.

You can also initialize the sliders from the current values from the VECU by adding:

```
    Set dThrottle = ecu.GetDeviceByName("Throttle")
    Set aThrottle = dThrottle.GetActionByName("Value")
    Set eThrottle = dThrottle.GetEventByName("Value")
    res = ""
    If ECU_OK = eThrottle.Query(res) Then
        sThrottle.value = res
    End If
    Set dBrake = ecu.GetDeviceByName("Brake")
    Set aBrake = dBrake.GetActionByName("Value")
    Set eBrake = dBrake.GetEventByName("Value")
    res = ""
    If ECU_OK = eBrake.Query(res) Then
        sBrake.value = res
    End If
```

13.3.5 Reacting to events

You have seen how to read the value of an event to initialize the control values, so it is an easy step to set up a timer to poll for changes.

But we don't want to do that…

It is clearly more efficient to be informed by the VECU that an event has changed, so we simply enable the event hook mechanism and respond to 'OnEventChange'.

Modify the code above to enable the hooks:

```
    Dim res As String

    Set dSpeed = ecu.GetDeviceByName("Speedometer")
    Set eSpeed = dSpeed.GetEventByName("Value")
    res = ""
    If ECU_OK = eSpeed.Query(res) Then
        pSpeed.value = res
        eSpeed.Hook (1)
    End If

    Set dThrottle = ecu.GetDeviceByName("Throttle")
    Set aThrottle = dThrottle.GetActionByName("Value")
    Set eThrottle = dThrottle.GetEventByName("Value")
    res = ""
    If ECU_OK = eThrottle.Query(res) Then
        sThrottle.value = res
        eThrottle.Hook (1)
    End If

    Set dBrake = ecu.GetDeviceByName("Brake")
    Set aBrake = dBrake.GetActionByName("Value")
    Set eBrake = dBrake.GetEventByName("Value")
    res = ""
    If ECU_OK = eBrake.Query(res) Then
        sBrake.value = res
        eBrake.Hook (1)
    End If
```

Also add the event handlers:

```
Private Sub eBrake_OnEventChange(ByVal value As
String)
        sBrake.value = value
End Sub

Private Sub eSpeed_OnEventChange(ByVal value As
String)
        pSpeed.value = value
End Sub

Private Sub eThrottle_OnEventChange(ByVal value As
String)
        sThrottle.value = value
End Sub
```

Easy! The application will now automatically update the sliders and progress bar whenever the associated events change in the VECU. You can use vrtaMonitor to change the Brake and Throttle values, and your application will respond automatically.

13.3.6 Sending actions

The final step links the sliders to the Brake and Throttle.

The code is laughably simple:

```
Private Sub sBrake_Change()
    aBrake.Send (sBrake.value)
End Sub

Private Sub sThrottle_Change()
    aThrottle.Send (sThrottle.value)
End Sub
```

13.3.7 Summary

Clearly in a 'real' application you will wish to go a lot further than this. Points to note are:

- Run-time error checking is necessary
- You may wish to load an ECU rather than simply attach to an existing one. You must remember that the name of the VECU executable that gets passed to vrtaServer in LoadECU must be a path that is visible to the server. You cannot specify a file on machine 'a' if the server is on machine 'b'.
- Data sent between your application and VECU actions/events is in the form of Strings. Multiple values are '\n' separated.

13.4 Method Description

Each interface method is described in a standard form as follows:

The title gives the name of the method.

A brief description of the method is provided.

Method declaration:

Interface in IDL syntax.

Parameters:

Parameter	Input/Output	Description
`Parameter Name`	Input/Output	Description.

Description:

Explanation of the functionality of the method.

Return values:

Value	Description
`Return values`	Description of return value.

13.5 CVcServer

A client must create one or more instances of the `CVcServer` object for each `vrtaServer` with which they wish to communicate.

`CVcServer` provides the capability to load Virtual ECUs, find out what ECUs are loaded / running and discover the information needed to create a `CVcECU` instance.

`CVcServer` implements the interface `ICVcServer`.

13.6 ICVcServer

This is the interface to `vrtaServer`. `ICVcServer`'s constants and API are based on those in the Virtual ECU Server Library (VesLib).

13.6.1 Enum: IVcServer_DisplayMode

This enumeration provides values that determine whether a VECU is started with or without its embedded GUI visible.

IVcServer_DisplayMode Name
`SILENT`
`GUI`

13.6.2 Enum: IVcServer_StartMode

This enumeration provides values that determine whether a VECU is auto-started or started in slave mode

IVcServer_StartMode Name
`AUTO`
`SLAVE`

13.6.3 Enum: IVcServer_Status

This enumeration provides the return values for all `ICVcServer` methods.

IVcServer_Status Name
`STATUS_OK`
`SERVER_START`
`SERVER_COMMS`
`NO_ECU`
`ECU_LOADED`
`ECU_NOT_LOADED`
`ECU_SLAVE`

ECU_ALIASED
NOT_LOADED

13.6.4 Method: AttachECU()

Attach to a loaded Virtual ECU.

Method declaration:

```
HRESULT _stdcall AttachECU(
    [in] BSTR alias,
    [out] int * diagport,
    [out, retval] IVcServer_Status * status
)
```

Parameters:

Parameter	Input/Output	Description
alias	Input	A Virtual ECU alias.
diagport	Output	The TCP port number used by the Virtual ECU's diagnostic interface.
status	Output	The return value.

Description:

This method is used to connect to a Virtual ECU that has already been loaded. On successful return *diagport contains the port number of the virtual ECU's diagnostic interface.

Return values:

Value	Description
STATUS_OK	Success.
SERVER_COMMS	If the object cannot communicate with vrtaServer.
NO_ECU	If the alias does not exist.
ECU_NOT_LOADED	If the alias has not been loaded.

13.6.5 Method: Connect()

Connect to a vrtaServer instance.

Method declaration:

```
HRESULT _stdcall Connect(
    [in, defaultvalue("localhost")] BSTR hostname,
    [in, defaultvalue(0)] long port,
    [out, retval] IVcServer_Status *status
)
```

Parameters:

Parameter	Input/Output	Description
hostname	Input	The hostname of the PC running vrtaServer.
port	Input	The TCP port being used by vrtaServer.
status	Output	The return value.

Description:

This method connects to vrtaServer on the PC named hostname using TCP port number port.

hostname can be a name (e.g. yok50123) or IP address (e.g. 127.0.0.1).

port is normally set to zero, in which case the object will search for vrtaServer in its default location. A non-zero value can be used to force the object to check only the specified port.

All subsequent methods apply to the instance of vrtaServer to which the object is connected.

Return values:

Value	Description
STATUS_OK	Success.
SERVER_START	If vrtaServer cannot be started.
SERVER_COMMS	If the object cannot communicate with vrtaServer.

13.6.6 Method: CreateAlias()

Create an alias for a Virtual ECU.

Method declaration:

```
HRESULT _stdcall CreateAlias(
    [in] BSTR app,
    [out] BSTR * alias,
    [out, retval] IVcServer_Status * status
)
```

Parameters:

Parameter	Input/Output	Description
app	Input	The full path (on the PC running vrtaServer) of a virtual ECU executable.
alias	Output	A new alias for the virtual ECU.
status	Output	The return value.

Description:

This method creates a new alias for a virtual ECU. On successful return, `alias` contains a new alias for the virtual ECU. The reference count for the alias will have been set to 1.

Return values:

Value	Description
`STATUS_OK`	Success.
`SERVER_COMMS`	If the object cannot communicate with `vrtaServer`.
`NO_ECU`	If the Virtual ECU executable does not exist.

13.6.7 Method: Disconnect()

Disconnect from `vrtaServer`.

Method declaration:

```
HRESULT Disconnect( void )
```

Parameters:

Parameter	Input/Output	Description
`<none>`		

Description:

This method is used to disconnect from `vrtaServer`. This method should be called prior to termination of the application, or before connecting to a different instance of `vrtaServer`.

Return values:

Value	Description
`<none>`	

13.6.8 Method: FindECUs()

Find out what Virtual ECU executables are present.

Method declaration:

```
HRESULT _stdcall FindECUs(
    [in] BSTR srchpath,
    [out] BSTR * path,
    [out] BSTR * drives,
    [out] BSTR * subdirs,
    [out] BSTR * apps,
    [out, retval] IVcServer_Status * status,
)
```

Parameters:

Parameter	Input/Output	Description
`srchpath`	Input	The path of the directory to be searched for Virtual ECU executables.
`path`	Output	The current absolute search path.
`drives`	Output	A comma-separated list of the drives that are readable.
`subdirs`	Output	A '`\n`' separated list of the subdirectories of `path`, including `".."` for a non-root directory.
`apps`	Output	A '`\n`' separated list of the names of files in `path` that could be Virtual ECU executables.
`status`	Output	The return value.

Description:

This method is used to discover the virtual ECU executables on `vrtaServer`'s PC.

`srchpath` contains the path of the directory to be searched - either as an absolute path or a path relative to the directory containing `vrtaServer`.

Return values:

Value	Description
`STATUS_OK`	Success.
`SERVER_COMMS`	If the object cannot communicate with `vrtaServer`.

13.6.9 Method: FreeAlias()

Free a Virtual ECU alias.

Method declaration:

```
HRESULT _stdcall FreeAlias(
    [in] BSTR alias,
    [out, retval] IVcServer_Status * status
)
```

Parameters:

Parameter	Input/Output	Description
`alias`	Input	A Virtual ECU alias.
`status`	Output	The return value.

Description:

This method decrements the reference count of the specified virtual ECU alias. An alias is removed when its reference count reaches zero. (If the connection to `vrtaServer` terminates without freeing aliases then `vrtaServer` will automatically decrement the reference counts of aliases appropriately.)

Return values:

Value	Description
`STATUS_OK`	Success.
`SERVER_COMMS`	If the object cannot communicate with `vrtaServer`.
`NO_ECU`	If the alias does not exist.

13.6.10 Method: GetAliases()

Get a list of the aliases that exist for a Virtual ECU executable.

Method declaration:

```
HRESULT _stdcall GetAliases(
    [in] BSTR app,
    [out] BSTR * aliases,
    [out, retval] IVcServer_Status * status
)
```

Parameters:

Parameter	Input/Output	Description
`app`	Input	The full path (on the PC running `vrtaServer`) of a virtual ECU executable.
`aliases`	Output	A '`\n`' separated list of all aliases that exist for the Virtual ECU executable.
`status`	Output	The return value.

Description:

This method gets a list of all aliases that have been created for a virtual ECU executable.

On successful return `aliases` contains a '`\n`' separated list of all aliases that exist for the Virtual ECU executable. If no alias exists then one will be created. The reference count of each alias returned is incremented by 1.

Return values:

Value	Description
`STATUS_OK`	Success.
`SERVER_COMMS`	If the object cannot communicate with `vrtaServer`.
`NO_ECU`	If the Virtual ECU executable does not exist.

13.6.11 Method: GetInfo()

Get version information about a Virtual ECU.

Method declaration:

```
HRESULT _stdcall GetInfo(
    [in] BSTR alias,
    [out] BSTR *info,
    [out, retval] IVcServer_Status * status
)
```

Parameters:

Parameter	Input/Output	Description
`alias`	Input	A Virtual ECU alias.
`info`	Output	Virtual ECU version information.
`status`	Output	The return value.

Description:

This function returns version information about the specified `alias`. On successful return, `info` contains a '`\n`' separated list of "key=value" pairs.

Return values:

Value	Description
`STATUS_OK`	Success.
`SERVER_COMMS`	If the object cannot communicate with `vrtaServer`.
`NO_ECU`	If the alias does not exist.

13.6.12 Method: ListAliases()

Get a list of all aliases that exist.

Method declaration:

```
HRESULT _stdcall ListAliases(
    [out] BSTR * aliases,
    [out, retval] IVcServer_Status * status
)
```

Parameters:

Parameter	Input/Output	Description
`aliases`	Output	A '`\n`' separated list of all aliases that exist
`status`	Output	The return value.

Description:

This method gets a list of all aliases that exist.

On successful return `aliases` contains a '`\n`' separated list of the aliases. The reference count of each alias returned is incremented by 1.

Return values:

Value	Description
`STATUS_OK`	Success.
`SERVER_COMMS`	If the object cannot communicate with `vrtaServer`.

13.6.13 Method: ListLoadedAliases()

Get a list of the Virtual ECUs that have been loaded.

Method declaration:

```
HRESULT _stdcall ListLoadedAliases(
    [out] BSTR * aliases,
    [out, retval] IVcServer_Status * status
)
```

Parameters:

Parameter	Input/Output	Description
`aliases`	Output	A '`\n`' separated list of the aliases of all loaded Virtual ECUs.
`status`	Output	The return value.

Description:

This method gets a list of the aliases for all loaded Virtual ECUs.

On successful return `aliases` contains a '`\n`' separated list of the aliases. The reference count of each alias returned is incremented by 1.

Return values:

Value	Description
`STATUS_OK`	Success.
`SERVER_COMMS`	If the object cannot communicate with `vrtaServer`.

13.6.14 Method: LoadECU()

Load a Virtual ECU.

Method declaration:

```
HRESULT _stdcall LoadECU(
    [in] BSTR alias,
    [in] IVcServer_StartMode startmode,
    [in] IVcServer_DisplayMode displaymode,
    [in] BSTR command,
    [out] int * diagport,
    [out, retval] IVcServer_Status * status
)
```

Parameters:

Parameter	Input/Output	Description
`alias`	Input	A Virtual ECU alias.
`startmode`	Input	The start mode for the Virtual ECU.
`displaymode`	Input	The display mode for the Virtual ECU.
`command`	Input	The command line for the Virtual ECU.
`diagport`	Output	The TCP port number used by the Virtual ECU's diagnostic interface.
`status`	Output	The return value.

Description:

This method is used to load and connect to a virtual ECU specified by `alias`.

If `startmode` is `AUTO` then the alias is loaded in autostart mode.

If `startmode` is `SLAVE` then the alias is loaded in slave mode.

If `displaymode` is `SILENT` then the alias is loaded in silent mode.

If `displaymode` is `GUI` then the alias is loaded in GUI mode.

`command` specifies additional command line parameters for the Virtual ECU.

On successful return `*diagport` contains the port number of the Virtual ECU's diagnostic interface.

Return values:

Value	Description
`STATUS_OK`	Success.
`SERVER_COMMS`	If the object cannot communicate with `vrtaServer`.
`NO_ECU`	If the alias does not exist.
`ECU_LOADED`	If the alias has already been loaded.

13.6.15 Method: ServerStatus()

Check if the server is still connected.

Method declaration:

```
HRESULT ServerStatus(
    [out, retval] IVcServer_Status *status
)
```

Parameters:

Parameter	Input/Output	Description
`status`	Output	The return value.

Description:

This method is used to check if the object is (still) connected to `vrtaServer`.

Return values:

Value	Description
`STATUS_OK`	Success.
`SERVER_COMMS`	If the object cannot communicate with `vrtaServer`.

13.7 CVcECU

A `CVcECU` object represents a connection to a Virtual ECU. It can be connected to (and disconnected from) local and remote Virtual ECUs. It provides access to the Virtual ECU's devices, events and actions.

`CVcECU` implements the interface `ICVcECU`.

13.8 ICVcECU

This is the interface to a Virtual ECU.

13.8.1 Enum: IVcECU_Status

This enumeration provides the return values for all `ICVcECU` methods.

IVcECU_Status Name
`ECU_OK`
`ECU_DevErr`
`ECU_IDErr`
`ECU_ValErr`
`ECU_ConErr`

13.8.2 Method: Connect()

Connect to a Virtual ECU.

Method declaration:

```
HRESULT _stdcall Connect(
    [in] BSTR hostname,
    [in] long port,
    [out, retval] IVcECU_Status * status
)
```

Parameters:

Parameter	Input/Output	Description
`hostname`	Input	The hostname of the PC running the Virtual ECU.
`port`	Input	The TCP port of the Virtual ECU's diagnostic interface.
`status`	Output	The return value.

Description:

This method causes the object to connect to a Virtual ECU whose diagnostic interface is using port `port` and that is loaded on the PC named `hostname`.

`hostname` can be a name (e.g. yok50123) or IP address (e.g. 127.0.0.1).

This method causes the object to disconnect from any existing connection.

Return values:

Value	Description
`ECU_OK`	Success.
`ECU_ValErr`	If the connection cannot be made.

13.8.3 Method: Disconnect()

Disconnect from a Virtual ECU.

Method declaration:

```
HRESULT _stdcall Disconnect(
    [out, retval] IVcECU_Status * status
)
```

Parameters:

Parameter	Input/Output	Description
`status`	Output	The return value.

Description:

This method disconnects from a Virtual ECU. All interfaces supplied via `GetDevicexxx()`, `GetActionxxx()` and `GetEventxxx()` become invalid.

Return values:

Value	Description
`ECU_OK`	Success.
`ECU_ValErr`	If the object is not connected to a Virtual ECU.

13.8.4 Method: DoAction()

Send a data-less action to a virtual device.

Method declaration:

```
HRESULT _stdcall DoAction(
    [in] long dev,
    [in] long id,
    [out, retval] IVcECU_Status * status
)
```

Parameters:

Parameter	Input/Output	Description
`dev`	Input	The device ID.
`id`	Input	The action ID.
`status`	Output	The return value.

Description:

This method sends the action with ID `id` to the device with ID `dev`. Only use this method where the action requires no data.

Return values:

Value	Description
`ECU_OK`	Success.
`ECU_ConErr`	If the connection is invalid.
`ECU_DevErr`	If the device ID is invalid.
`ECU_IDErr`	If the action ID is invalid.
`ECU_ValErr`	If the sent data is invalid. I.e. there should have been some.

13.8.5 Method: GetDeviceByID()

Get an interface to a virtual device by device ID.

Method declaration:

```
HRESULT _stdcall GetDeviceByID(
    [in] long id,
    [out, retval] ICVcDevice ** device
)
```

Parameters:

Parameter	Input/Output	Description
`id`	Input	The device ID.
`device`	Output	The return value.

Description:

This method returns an `ICVcDevice` interface corresponding to the device with the specified device ID. `NULL` is returned if the ID is invalid. The first device has ID zero.

Return values:

Value	Description
`<an interface>`	An `ICVcDevice` interface corresponding to the specified device.

13.8.6 Method: GetDeviceByName()

Get an interface to a virtual device by device name.

Method declaration:

```
HRESULT _stdcall GetDeviceByName(
    [in] BSTR id,
    [out, retval] ICVcDevice ** device
)
```

Parameters:

Parameter	Input/Output	Description
`id`	Input	The device name.
`device`	Output	The return value.

Description:

This method returns an `ICVcDevice` interface corresponding to the device with the specified device name. `NULL` is returned if the name is invalid.

Return values:

Value	Description
`<an interface>`	An `ICVcDevice` interface corresponding to the specified device.

13.8.7 Method: GetDeviceCount()

Get the number of virtual devices.

Method declaration:

```
HRESULT _stdcall GetDeviceCount(
    [out, retval] long * count
)
```

Parameters:

Parameter	Input/Output	Description
`count`	Output	The return value.

Description:

This method gets the number of virtual devices in the Virtual ECU.

Return values:

Value	Description
`<a value>`	The number of virtual devices in the Virtual ECU.

13.8.8 Method: Hook()

Hook or unhook an event.

Method declaration:

```
HRESULT _stdcall Hook(
    [in] long dev,
    [in] long id,
    [in] long value,
    [out, retval] IVcECU_Status * status
)
```

Parameters:

Parameter	Input/Output	Description
`dev`	Input	The device ID.
`id`	Input	The event ID.
`value`	Input	1 to hook an event or zero to unhook an event.
`status`	Output	The return value.

Description:

This method controls whether the event with ID `id` belonging to the device with ID `dev` is hooked or unhooked. If the event is hooked then when the device raises the event a COM event is fired (via the event sink `ICVcECUEvents`). If `value` is one the specified event is hooked, if `value` is zero the specified event is unhooked. By default all events are unhooked.

Return values:

Value	Description
`ECU_OK`	Success.
`ECU_ConErr`	If the connection is invalid.
`ECU_DevErr`	If the device ID is invalid.
`ECU_IDErr`	If the event ID is invalid.

13.8.9 Method: QueryEvent()

Query (poll) the value of an event.

Method declaration:

```
HRESULT _stdcall QueryEvent(
    [in] long dev,
    [in] long id,
    [in, out] BSTR * value,
    [out, retval],IVcECU_Status * status
)
```

Parameters:

Parameter	Input/Output	Description
`dev`	Input	The device ID.
`id`	Input	The event ID.
`value`	Input/Output	Event input data on call and the value of the event on return.
`status`	Output	The return value.

Description:

This method queries the event with ID `id` belonging to the device with ID `dev`. If the event requires input data then this is passed in `value`. On successful return the value of the event is in `value`.

The data passed to and from the object is in string form. It can be converted from/to the native values by reference to the format specifiers that can be obtained via `QueryFormat()` and `ReplyFormat()`.

Return values:

Value	Description
`ECU_OK`	Success.
`ECU_ConErr`	If the connection is invalid.
`ECU_DevErr`	If the device ID is invalid.
`ECU_IDErr`	If the event ID is invalid.
`ECU_ValErr`	If the sent data is invalid.

13.8.10 Method: QueryFormat()

Get the data format for an event's input data.

Method declaration:

```
HRESULT _stdcall QueryFormat(
    [in] long dev,
    [in] long id,
    [out, retval] BSTR * value
)
```

Parameters:

Parameter	Input/Output	Description
`dev`	Input	The device ID.
`id`	Input	The event ID.
`value`	Output	The return value.

Description:

This method returns the input data-format string for the event with ID `id` belonging to the device with ID `dev`. The return value is empty if `dev` or `id` is invalid. This data format string describes the format of the data that should be provided as input to `QueryEvent()`. (Many events have no input data, so this is often empty.)

Return values:

Value	Description
`<a string>`	The input data format string for the specified event.

13.8.11 Method: ReplyFormat()

Get the data format for an event's value.

Method declaration:

```
HRESULT _stdcall ReplyFormat(
    [in] long dev,
    [in] long id,
    [out, retval] BSTR * value
)
```

Parameters:

Parameter	Input/Output	Description
`dev`	Input	The device ID.
`id`	Input	The event ID.
`value`	Output	The return value.

Description:

This method returns the data-format string for the value of the event with ID `id` belonging to the device with ID `dev`. The return value is empty if `dev` or `id` is invalid. This data format string describes the format of the data that is returned by `QueryEvent()`.

Return values:

Value	Description
`<a string>`	The data format string for the value of the specified event.

13.8.12 Method: SendAction()

Send an action containing data to a virtual device.

Method declaration:

```
HRESULT _stdcall SendAction(
    [in] long dev,
    [in] long id,
    [in] BSTR value,
    [out, retval] IVcECU_Status * status
)
```

Parameters:

Parameter	Input/Output	Description
`dev`	Input	The device ID.
`id`	Input	The action ID.
`value`	Input	Action data.
`status`	Output	The return value.

Description:

This method sends the action with ID `id` to the device with ID `dev`. The data passed to the object is in string form. It can be converted from the native values by reference to the format specifier that can be obtained via `SendFormat()`.

Return values:

Value	Description
`ECU_OK`	Success.
`ECU_ConErr`	If the connection is invalid.
`ECU_DevErr`	If the device ID is invalid.
`ECU_IDErr`	If the action ID is invalid.
`ECU_ValErr`	If the sent data is invalid.

13.8.13 Method: SendFormat()

Get the data format for an action.

Method declaration:

```
HRESULT _stdcall SendFormat(
    [in] long dev,
    [in] long id,
    [out, retval] BSTR * value
)
```

Parameters:

Parameter	Input/Output	Description
`dev`	Input	The device ID.
`Id`	Input	The action ID.
`value`	Output	The return value.

Description:

This method returns the data format string for the action with ID `id` belonging to the device with ID `dev`. This shows the format of the data that is passed to `SendAction()`. The return value is empty if `dev` or `id` is invalid.

Return values:

Value	Description
`<a string>`	The data format string for the specified action.

13.9 ICVcECUEvents

This interface is implemented by a client that wishes to receive COM events when a virtual device raises an event. Note that the event hook must have been activated via `ICVcECU.Hook()`. Hooking or unhooking the same event via `ICVcDevice` or `ICVcEvent` does not affect `ICVcECUEvents`.

13.9.1 Method: OnEventChange()

Event hook callback.

Method declaration:

```
HRESULT OnEventChange(
    [in] long dev,
    [in] long id,
    [in] BSTR value
)
```

Parameters:

Parameter	Input/Output	Description
`dev`	Input	The device ID.
`id`	Input	The event ID.
`value`	Input	The event's value.

Description:

This method is called when a hooked event is raised. The format of the data in `value` is the same as the data returned by `ICVcECU` method `QueryEvent()`.

Return values:

Value	Description
`<none>`	

13.10 CVcDevice

A CVcDevice object represents a single device in a Virtual ECU. A CVcDevice object must only be obtained via the CVcECU method GetDevice(). It cannot be created via CoCreateInstance(). This is to maintain the link between the Virtual ECU and the device.

A CVcDevice object provides the ability to access the actions and events of a specific device in a Virtual ECU.

CVcDevice implements the interface ICVcDevice.

13.11 ICVcDevice

This is the interface to a virtual device.

13.11.1 Method: DeviceID()

Get the device's ID.

Method declaration:

```
HRESULT _stdcall DeviceID(
    [out, retval] long * id
)
```

Parameters:

Parameter	Input/Output	Description
id	Output	The return value.

Description:

This method returns the ID of the device. This is the number by which the device is known within its Virtual ECU. The first device has ID zero.

Return values:

Value	Description
<a value>	The ID of the device.

13.11.2 Method: DoAction()

Send a data-less action.

Method declaration:

```
HRESULT _stdcall DoAction(
    [in] long id,
    [out, retval] IVcECU_Status * status
)
```

Parameters:

Parameter	Input/Output	Description
`id`	Input	The action ID.
`status`	Output	The return value.

Description:

This method sends the action with ID `id` to this device. Only use this method where the action requires no data.

Return values:

Value	Description
`ECU_OK`	Success.
`ECU_ConErr`	If the connection is invalid.
`ECU_IDErr`	If the action ID is invalid.
`ECU_ValErr`	If the sent data is invalid. I.e. there should have been some.

13.11.3 Method: GetActionByID()

Get an interface to an action by action ID.

Method declaration:

```
HRESULT _stdcall GetActionByID(
    [in] long id,
    [out, retval] ICVcAction ** action
)
```

Parameters:

Parameter	Input/Output	Description
`id`	Input	The action ID.
`action`	Output	The return value.

Description:

This method returns an `ICVcAction` interface corresponding to the action with the specified action ID. `NULL` is returned if the ID is invalid. The first action has ID 1.

Return values:

Value	Description
`<an interface>`	An `ICVcAction` interface corresponding to the specified action.

13.11.4 Method: GetActionByName()

Get an interface to an action by name.

Method declaration:

```
HRESULT _stdcall GetActionByName(
    [in] BSTR id,
    [out, retval] ICVcAction ** action
)
```

Parameters:

Parameter	Input/Output	Description
`id`	Input	The action name.
`action`	Output	The return value.

Description:

This method returns an `ICVcAction` interface corresponding to the action with the specified name. `NULL` is returned if the name is invalid.

Return values:

Value	Description
`<an interface>`	An `ICVcAction` interface corresponding to the specified action.

13.11.5 Method: GetActionCount()

Get the number of actions supported by the device.

Method declaration:

```
HRESULT _stdcall GetActionCount(
    [out, retval] long * count
)
```

Parameters:

Parameter	Input/Output	Description
`count`	Output	The return value.

Description:

This method returns the number of actions supported by the device.

Return values:

Value	Description
`<a value>`	The number of actions supported by the device.

13.11.6 Method: GetEventByID()

Get an interface to an event by event ID.

Method declaration:

```
HRESULT _stdcall GetEventByID(
    [in] long id,
    [out, retval] ICVcEvent ** event
)
```

Parameters:

Parameter	Input/Output	Description
`id`	Input	The event ID.
`event`	Output	The return value.

Description:

This method returns an `ICVcEvent` interface corresponding to the event with the specified event ID. `NULL` is returned if the ID is invalid. The first event has ID 1.

Return values:

Value	Description
`<an interface>`	An `ICVcEvent` interface corresponding to the specified event.

13.11.7 Method: GetEventByName()

Get an interface to an event by name.

Method declaration:

```
HRESULT _stdcall GetEventByName(
    [in] BSTR id,
    [out, retval] [out, retval] ICVcEvent ** event
)
```

Parameters:

Parameter	Input/Output	Description
`id`	Input	The event name.
`event`	Output	The return value.

Description:

This method returns an `ICVcEvent` interface corresponding to the event with the specified name. `NULL` is returned if the name is invalid.

Return values:

Value	Description
`<an interface>`	An `ICVcEvent` interface corresponding to the specified event.

13.11.8 Method: GetEventCount()

Get the number of events supported by the device.

Method declaration:

```
HRESULT _stdcall GetEventCount(
    [out, retval] long * count
)
```

Parameters:

Parameter	Input/Output	Description
`count`	Output	The return value.

Description:

This method returns the number of events supported by the device.

Return values:

Value	Description
`<a value>`	The number of events supported by the device.

13.11.9 Method: Hook()

Hook or unhook an event.

Method declaration:

```
HRESULT _stdcall Hook(
    [in] long id,
    [in] long value,
    [out, retval] IVcECU_Status * status
)
```

Parameters:

Parameter	Input/Output	Description
id	Input	The event ID.
value	Input	1 to hook an event or zero to unhook an event.
status	Output	The return value.

Description:

This method controls whether the event with ID `id` is hooked or unhooked. If the event is hooked then when the device raises the event a COM event is fired (via the event sink `ICVcDeviceEvents`). If `value` is one the specified event is hooked, if `value` is zero the specified event is unhooked. By default all events are unhooked.

Return values:

Value	Description
ECU_OK	Success.
ECU_ConErr	If the connection is invalid.
ECU_IDErr	If the event ID is invalid.

13.11.10 Method: Name()

Get the device's name.

Method declaration:

```
HRESULT _stdcall Name(
    [out, retval] BSTR * name
)
```

Parameters:

Parameter	Input/Output	Description
name	Output	The return value.

Description:

This method returns the name of the device.

Return values:

Value	Description
`<a value>`	The name of the device.

13.11.11 Method: QueryEvent()

Query (poll) the value of an event.

Method declaration:

```
HRESULT _stdcall QueryEvent(
    [in] long id,
    [in, out] BSTR * value,
    [out, retval] IVcECU_Status * status
)
```

Parameters:

Parameter	Input/Output	Description
`id`	Input	The event ID.
`value`	Input/Output	Event input data on call and the value of the event on return.
`status`	Output	The return value.

Description:

This method queries (polls) the event with ID `id`. If the event requires input data then this is passed in `value`. On successful return the value of the event is in `value`.

The data passed to and from the object is in string form. It can be converted from/to the native values by reference to the format specifiers that can be obtained via `QueryFormat()` and `ReplyFormat()`.

Return values:

Value	Description
`ECU_OK`	Success.
`ECU_ConErr`	If the connection is invalid.
`ECU_IDErr`	If the event ID is invalid.
`ECU_ValErr`	If the sent data is invalid.

13.11.12 Method: QueryFormat()

Get the data format for an event's input data.

Method declaration:

```
HRESULT _stdcall QueryFormat(
    [in] long id,
    [out, retval] BSTR * value
)
```

Parameters:

Parameter	Input/Output	Description
`id`	Input	The event ID.
`value`	Output	The return value.

Description:

This method returns the input data-format string for the event with ID `id`. The return value is empty if `id` is invalid. This data format string describes the format of the data that should be provided as input to `QueryEvent()`. (Many events have no input data, so this is often empty.)

Return values:

Value	Description
`<a string>`	The input data format string for the specified event.

13.11.13 Method: ReplyFormat()

Get the data format for an event's value.

Method declaration:

```
HRESULT _stdcall ReplyFormat(
    [in] long id,
    [out, retval] BSTR * value
)
```

Parameters:

Parameter	Input/Output	Description
`id`	Input	The event ID.
`value`	Output	The return value.

Description:

This method returns the data-format string for the value of the event with ID `id`. The return value is empty if `id` is invalid. This data format string describes the format of the data that is returned by `QueryEvent()`.

Return values:

Value	Description
`<a string>`	The data format string for the value of the specified event.

13.11.14 Method: SendAction()

Send an action containing data.

Method declaration:

```
HRESULT _stdcall SendAction(
    [in] long id,
    [in] BSTR value,
    [out, retval] IVcECU_Status * status
)
```

Parameters:

Parameter	Input/Output	Description
id	Input	The action ID.
value	Input	Action data.
status	Output	The return value.

Description:

This method sends the action with ID `id` to this device. The data passed to the object is in string form. It can be converted from the native values by reference to the format specifier that can be obtained via `SendFormat()`.

Return values:

Value	Description
ECU_OK	Success.
ECU_ConErr	If the connection is invalid.
ECU_IDErr	If the action ID is invalid.
ECU_ValErr	If the sent data is invalid.

13.11.15 Method: SendFormat()

Get the data format for an action.

Method declaration:

```
HRESULT _stdcall SendFormat(
    [in] long id,
    [out, retval] BSTR * value
)
```

Parameters:

Parameter	Input/Output	Description
id	Input	The action ID.
value	Output	The return value.

Description:

This method returns the data format string for the action with ID `id`. This shows the format of the data that is passed to `SendAction()`. The return value is empty if `id` is invalid.

Return values:

Value	Description
`<a string>`	The data format string for the specified action.

13.12 ICVcDeviceEvents

This interface is implemented by a client that wishes to receive COM events when a virtual device raises an event. Note that the event hook must have been activated via `ICVcDevice.Hook()`. Hooking or unhooking the same event via `ICVcECU` or `ICVcEvent` does not affect `ICVcDeviceEvents`.

13.12.1 Method: OnEventChange()

Event hook callback.

Method declaration:

```
HRESULT OnEventChange(
    [in] long id,
    [in] BSTR value
)
```

Parameters:

Parameter	Input/Output	Description
`id`	Input	The event ID.
`value`	Input	The event's value.

Description:

This method is called when a hooked event is raised. The format of the data in `value` is the same as the data returned by `ICVcDevice` method `QueryEvent()`.

Return values:

Value	Description
`<none>`	

13.13 CVcAction

A CVcAction object represents a single action on a specific device in a Virtual ECU. A CVcAction object must only be obtained via the CVcDevice method GetActionxxx(). It cannot be created via CoCreateInstance(). This is to maintain the link between the device and the action.

A CVcAction object provides the ability to send an action to a Virtual ECU.

CVcAction implements the interface ICVcAction.

13.14 ICVcAction

This is the interface to a virtual device action.

13.14.1 Method: ActionID()

Get the action's ID.

Method declaration:

```
HRESULT _stdcall ActionID(
    [out, retval] long * id
)
```

Parameters:

Parameter	Input/Output	Description
id	Output	The return value.

Description:

This method returns the ID of the action. This is the number by which the action is known within its device. The first action has ID 1.

Return values:

Value	Description
<a value>	The ID of the action.

13.14.2 Method: Do()

Send a data-less action.

Method declaration:

```
HRESULT _stdcall Do (
    [out, retval] IVcECU_Status * status
)
```

Parameters:

Parameter	Input/Output	Description
`status`	Output	The return value.

Description:

This method sends the action. Only use this method where the action requires no data.

Return values:

Value	Description
`ECU_OK`	Success.
`ECU_ConErr`	If the connection is invalid.
`ECU_ValErr`	If the sent data is invalid. I.e. there should have been some.

13.14.3 Method: Name()

Get the action's name.

Method declaration:

```
HRESULT _stdcall Name(
    [out, retval] BSTR * name
)
```

Parameters:

Parameter	Input/Output	Description
`name`	Output	The return value.

Description:

This method returns the name of the action.

Return values:

Value	Description
`<a value>`	The name of the action.

13.14.4 Method: Send()

Send an action containing data.

Method declaration:

```
HRESULT _stdcall SendAction(
    [in] BSTR value,
    [out, retval] IVcECU_Status * status
)
```

Parameters:

Parameter	Input/Output	Description
`value`	Input	Action data.
`status`	Output	The return value.

Description:

This method sends the action. The data passed to the object is in string form. It can be converted from the native values by reference to the format specifier that can be obtained via `SendFormat()`.

Return values:

Value	Description
`ECU_OK`	Success.
`ECU_ConErr`	If the connection is invalid.
`ECU_ValErr`	If the sent data is invalid.

13.14.5 Method: SendFormat()

Get the data format for an action.

Method declaration:

```
HRESULT _stdcall SendFormat(
    [out, retval] BSTR * value
)
```

Parameters:

Parameter	Input/Output	Description
`value`	Output	The return value.

Description:

This method returns the data format string for the action. This shows the format of the data that is passed to `Send()`.

Return values:

Value	Description
`<a string>`	The data format string for the specified action.

13.15 CVcEvent

A `CVcEvent` object represents a single event on a specific device in a Virtual ECU. A `CVcEvent` object must only be obtained via the `CVcDevice` method `GetEventxxx()`. It cannot be created via `CoCreateInstance()`. This is to maintain the link between the device and the event.

A `CVcEvent` object provides the ability to query the current value of an event. It can also enable and disable the raising of COM events for events raised by virtual devices.

`CVcEvent` implements the interface `ICVcEvent`.

13.16 ICVcEvent

This is the interface to a virtual device event.

13.16.1 Method: EventID()

Get the events's ID.

Method declaration:

```
HRESULT _stdcall EventID(
    [out, retval] long * id
)
```

Parameters:

Parameter	Input/Output	Description
`id`	Output	The return value.

Description:

This method returns the ID of the event. This is the number by which the event is known within its device. The first event has ID 1.

Return values:

Value	Description
`<a value>`	The ID of the event.

13.16.2 Method: Hook()

Hook or unhook the event.

Method declaration:

```
HRESULT _stdcall Hook(
    [in] long value,
    [out, retval] IVcECU_Status * status
)
```

Parameters:

Parameter	Input/Output	Description
value	Input	1 to hook the event or zero to unhook the event.
status	Output	The return value.

Description:

This method controls whether the event is hooked or unhooked. If the event is hooked then when the device raises the event a COM event is fired (via the event sink `ICVcEventEvents`). If `value` is one the event is hooked, if `value` is zero the event is unhooked. By default all events are unhooked.

Return values:

Value	Description
ECU_OK	Success.
ECU_ConErr	If the connection is invalid.

13.16.3 Method: Name()

Get the event's name.

Method declaration:

```
HRESULT _stdcall Name(
    [out, retval] BSTR * name
)
```

Parameters:

Parameter	Input/Output	Description
name	Output	The return value.

Description:

This method returns the name of the event.

Return values:

Value	Description
<a value>	The name of the event.

13.16.4 Method: Query()

Query the value of an event.

Method declaration:

```
HRESULT _stdcall QueryEvent(
    [in, out] BSTR * value,
    [out, retval] IVcECU_Status * status
)
```

Parameters:

Parameter	Input/Output	Description
`value`	Input/Output	Event input data on call and the value of the event on return.
`status`	Output	The return value.

Description:

This method queries the event. If the event requires input data then this is passed in `value`. On successful return the value of the event is in `value`.

The data passed to and from the object is in string form. It can be converted from/to the native values by reference to the section on data format specifiers.

Return values:

Value	Description
`ECU_OK`	Success.
`ECU_ConErr`	If the connection is invalid.
`ECU_ValErr`	If the sent data is invalid.

13.16.5 Method: QueryFormat()

Get the data format for an event's input data.

Method declaration:

```
HRESULT _stdcall QueryFormat(
    [out, retval] BSTR * value
)
```

Parameters:

Parameter	Input/Output	Description
`value`	Output	The return value.

Description:

This method returns the input data-format string for the event. This data format string describes the format of the data that should be provided as input to `Query()`. (Many events have no input data, so this is often empty.)

Return values:

Value	Description
`<a string>`	The input data format string for the event.

13.16.6 Method: ReplyFormat()

Get the data format for an event's value.

Method declaration:

```
HRESULT _stdcall ReplyFormat(
    [out, retval] BSTR * value
)
```

Parameters:

Parameter	Input/Output	Description
`value`	Output	The return value.

Description:

This method returns the data-format string for the value of the event. This data format string describes the format of the data that is returned by `Query()`.

Return values:

Value	Description
`<a string>`	The data format string for the value of the event.

13.17 ICVcEventEvents

This interface is implemented by a client that wishes to receive COM events when a virtual device raises an event. Note that the event hook must have been activated via `ICVcEvent.Hook()`. Hooking or unhooking the same event via `ICVcECU` or `ICVcDevice` does not affect `ICVcEventEvents`.

13.17.1 Method: OnEventChange()

Event hook callback.

Method declaration:

```
HRESULT OnEventChange(
    [in] BSTR value
)
```

Parameters:

Parameter	Input/Output	Description
`value`	Input	The event's value.

Description:

This method is called when a hooked event is raised. The format of the data in `value` is the same as the data returned by `ICVcEvent` method `Query()`.

Return values:

Value	Description
`<none>`	

14 Compiler Configuration

RTA-OSEK for PC has been designed to work with almost any PC C/C++ compiler. It is pre-configured to work with the following compilers:

- MinGW/gcc (tested with version 3.4.2)
- Microsoft Visual C++ 5.0
- Microsoft Visual Studio 2003
- Borland C++ 5..5.1 / Borland C++ Builder 5
- Borland C++ 5.8.1 / Borland Developer Studio 2006

LiveDevices **cannot guarantee** that *RTA-OSEK for PC* will work with a compiler that is not in the above list but there is a good chance that it will as long as the compiler obeys the following rules:

- The C/C++ `char` type is 8 bits.
- The C/C++ `short` type is 16 bits.
- The C/C++ `int` type is 32 bits.
- The C/C++ `long` type is 32 bits.
- The C/C++ `float` type is 32 bits.
- The C/C++ `double` type is 64 bits.
- The compiler includes header files and libraries to support the Windows API. e.g. the header file `windows.h` is provided.
- Fields within a C `struct` are stored in memory in the order they occur in the structure definition.
- Fields within a C `struct` are aligned on natural boundaries. i.e. a `short` is always aligned on a 16 bit boundary, an `int`, a `long` and a `float` is always aligned on a 32 bit boundary and a `double` is always aligned on a 64 bit boundary.

Each different compiler supported by RTA-OSEK is called a "variant". Configuring *RTA-OSEK for PC* for a new variant consists of three steps:

- Modifying `toolinit.bat` to include details of the variant.
- Creating an RTA-OSEK initialization file for the variant.
- Possibly creating new floating-point wrappers.

The first step is to think of a name for the new variant. This should be reasonably short and should follow C identifier rules (i.e. can contain letters, numbers and '_' and is case sensitive). As an illustration, we will add a new variant that is a variation of the standard MinGW/gcc variant that has debugging enabled. The standard MinGW/gcc variant is called "MinGW"; we will call the new variant "MinGW_DBG".

14.1 Modifying Toolinit.bat

The `toolinit.bat` file is used by RTA-OSEK to setup environment variables containing paths to the compiler, assembler, linker and compiler header file directories. You will need to modify `toolinit.bat` to add support for a new variant.

Near the top of `toolinit.bat` you will see a block of statements like:

```
if not @%1==@ set VRTA=%1
if @%VRTA%@==@MinGW@ goto MINGW
if @%VRTA%@==@BorlandC@ goto BCPP
if @%VRTA%@==@BDS2006@ goto BDS_2006
if @%VRTA%@==@VisualC5@ goto VCPP5
if @%VRTA%@==@VS2003@ goto VS2003
```

These `if` statements branch to the part of `toolinit.bat` that sets up the environment for a specific variant. The variant name is passed into `toolinit.bat` either on the command line or in the `VRTA` environment variable. We need to add an analogous `if` statement for our new variant. For example:

```
if not @%1==@ set VRTA=%1
if @%VRTA%@==@MinGW@ goto MINGW
if @%VRTA%@==@BorlandC@ goto BCPP
if @%VRTA%@==@BDS2006@ goto BDS_2006
if @%VRTA%@==@VisualC5@ goto VCPP5
if @%VRTA%@==@VS2003@ goto VS2003
if @%VRTA%@==@MinGW_DBG@ goto MINGW_DBG
```

The line in bold makes `toolinit.bat` branch to the label `MINGW_DBG` when the variant "MinGW_DBG" is specified.

Further on in `toolinit.bat` you will see blocks of statements like:

```
rem ================= MINGW ================
:MINGW
rem tools installation directory
set CBASE=c:\mingw

rem location of C compiler
set CC=%CBASE%\bin\gcc.exe

rem location of C++ compiler
set AS=%CBASE%\bin\gcc.exe

rem location of linker
set LNK=%CBASE%\bin\g++.exe

rem location of Archiver / librarian
set AR=%CBASE%\bin\ar.exe

rem Set location of C include files
set CBASE_INC=%CBASE%\include
```

```
rem Default settings
SET _LIBS=-lwinmm -lws2_32
SET _OBJ=o

goto check
```

Again we need to add a section for our new variant. Since our new variant is also using MinGW/gcc the environment variable settings will be the same as the standard MinGW variant. Obviously if you were adding a variant for a different compiler the environment variable settings would be different. Have a look at the other standard variants in `toolint.bat` for examples.

Our new section in `toolinit.bat` looks like:

```
rem ============== MINGW Debugging =============
:MINGW_DBG
rem tools installation directory
set CBASE=C:\mingw

rem location of C compiler
set CC=%CBASE%\bin\gcc.exe

rem location of C++ compiler
set AS=%CBASE%\bin\gcc.exe

rem location of linker
set LNK=%CBASE%\bin\g++.exe

rem location of Archiver / librarian
set AR=%CBASE%\bin\ar.exe

rem Set location of C include files
set CBASE_INC=%CBASE%\include

rem Default settings
SET _LIBS=-lwinmm -lws2_32
SET _OBJ=o

goto check
```

The environment variables set up should be self explanatory. The `_LIBS` variable contains the default libraries need when a Virtual ECU executable is linked. Here we have specified the Windows multi-media library (`winmm`) needed by `vrtaClock` devices and the winsock2 library (`ws2_32`). The `_OBJ` variable contains the object file suffix.

One slight oddity is that we set the assembler as `gcc.exe` in the line:

```
set AS=%CBASE%\bin\gcc.exe
```

This is because in a standard RTA-OSEK port the OSEK configuration is generated in an assembly file (`osgen.asm`) and RTA-OSEK uses the `AS` variable to find the assembler to assemble it. In *RTA-OSEK for PC* the OSEK configuration is generated in a C++ file (`osgen.cpp`) so we need to arrange for RTA-OSEK to use the C/C++ compiler to “assemble” `osgen.cpp`.

14.2 Creating a new Variant

The next step is to create a description file for the new variant. The file for a variant called "`XXXX`" is called "`vrta_XXXX.ini`". This file should be in the "`bin`" directory. The "Target .ini files" chapter of the *RTA-OSEK Reference Guide* contains full details of how to write an initialization file.

The initialization file for our new variant will be called "`vrta_MinGW_DBG.ini`" and look like:

```
[globals]
Target=Virtual
numstacks=1
StackName0=Processor Stack

; Tell RTA-OSEK that when the compiler is invoked
; the environment variable %c% should contain the
; name of the C file to be compiled.
c_env="c"
; Tell RTA-OSEK that when the compiler is invoked
; the environment variable %o% should contain the
; name of the object file to be created.
obj_env="o"
; Tell RTA-OSEK that when the assembler is invoked
; the environment variable %c% should contain the
; name of the assembly file to be assembled.
; Remember that in RTA-OSEK for PC osgen is a C++
; file rather than an assembly file.
asm_env="c"
; Insert a compiler command to specify the object
; file.
c_insertopt=" -o%o% "
; The compiler option to set an include directory.
c_include="-I"
; The compiler option to define a macro.
c_define="-D"
; The default compilation options.
c_defopt="-c -g "
; The options used to compile osekdefs.c.
osekdefsc_defopt="-c -g "
; Insert an "assembler" command to specify the
; object file.
a_insertopt=" -o%o% "
; The "assembler" option to set an include
; directory.
a_include="-I"
; The "assembler" option to define a macro.
a_define="-D"
; The default "assembly" options.
a_defopt="-c -g "
; The options used to "assemble" osgen.cpp. We use
; %AOPTS% here because we want the default
; "assembly" options that include the include path
; settings.
osgen_defopt="%AOPTS% "
; The object file suffix.
extobj="o"
; The "assembly" file suffix. Remember that in
; RTA-OSEK for PC osgen is a C++ file rather than
; an assembly file.
extasm="cpp"
; The library file extension.
extlib="a"

; Ensure that the interrupt vectors have the
; correct names (you can put descriptive names here).
[vectors]
0x0=1
0x1=2
0x2=3
0x3=4
0x4=5
0x5=6
0x6=7
0x7=8
0x8=9
0x9=10
0xa=11
0xb=12
0xc=13
0xd=14
0xe=15
0xf=16
0x10=17
0x11=18
0x12=19
0x13=20
0x14=21
0x15=22
0x16=23
0x17=24
0x18=25
0x19=26
0x1a=27
0x1b=28
0x1c=29
0x1d=30
0x1e=31
0x1f=32
```

We have also set the options used to “assemble” `osgen.cpp` to be “`%AOPTS%`”. `AOPTS` will contain the options to “assemble” a general assembly file and will include directives to set up include paths to the *RTA-OSEK for PC* header files. These header files are needed to “assemble” `osgen.cpp`. An alternative would be to use the line:

```
osgen_defopt="-c -g -I. -I%APPL_INC% -I%RTA_INC% -
I%CBASE_INC% "
```

14.3 Floating-point Wrappers

RTA-OSEK saves the processor's floating-point registers for tasks and ISRs that use floating-point and can preempt other FP tasks or ISRs.

Default code is provided in VM to do this. However, it is possible for you to provide your own code to save the floating-point registers if you need to change the default behavior. The *RTA-OSEK Binding Manual PC* has details on how this is done.

The files `osfptgt.c` and `osfptgt.h` contain examples of user-provided floating-point wrappers for the standard variants. If you wish to use your own floating-point wrappers you will need to modify `osfptgt.c` and `osfptgt.h` for your new compiler.

15 Command Line Option Reference

This chapter provides a list of the command line options that are supported by Virtual ECU executables, `vrtaServer` and `vrtaMonitor`.

15.1 Virtual ECU Executables

The command line options listed below can be used when a Virtual ECU executable is run to control the behavior of the Virtual ECU. Note that options other than those listed below may be used with a Virtual ECU executable and they will be ignored by the Virtual Machine but can be recovered by your code by querying the ApplicationManager.

15.1.1 -alias=name (override the default alias)

When a Virtual ECU registers with `vrtaServer` it is normally assigned an alias that is simply the name of the Virtual ECU's executable (e.g. `vecu.exe`). The option `-alias=name` causes the token `name` to be used as the Virtual ECU's alias (if possible).

15.1.2 -lic=<licfile> (select a license file)

The `-lic=<licfile>` option tells a Virtual ECU to look in the license file `<licfile>` for a valid license.

15.1.3 -priority=<n> (set the Windows priority)

By default a Virtual ECU runs at the Window's priority `NORMAL_PRIORITY_CLASS`. If you wish to run a Virtual ECU at a different Windows priority the `-priority=<n>` option can be used; where `<n>` is:

`<n>`	**Windows Priority Class**
0	`IDLE_PRIORITY_CLASS`
1	`BELOW_NORMAL_PRIORITY_CLASS`
2	`NORMAL_PRIORITY_CLASS`
3	`ABOVE_NORMAL_PRIORITY_CLASS`
4	`HIGH_PRIORITY_CLASS`
5	`REALTIME_PRIORITY_CLASS`

Increasing the priority of a Virtual ECU will improve how closely it approximates "real-time" behavior but will negatively affect the performance of other applications running in the same PC.

15.1.4 -silent (select silent or GUI mode)

The `-silent` option causes a Virtual ECU to be loaded in silent mode. In silent mode the Virtual ECU does not display its own (embedded) GUI (it is assumed that `vrtaMonitor` or a similar program will be used to control the Virtual ECU).

If the `-silent` option is not used then the Virtual ECU is loaded in GUI mode. In *GUI* mode the Virtual ECU displays its own GUI.

15.1.5 -slave (select slave or autostart mode)

The `-slave` option causes a Virtual ECU to be loaded in slave mode. In slave mode the application thread is not started immediately after the `vrtaStart()` Virtual Machine API has been called. Instead the Virtual Machine waits until a Start action is sent to the Application Manager before starting the application thread.

In slave mode the Virtual Machine does not terminate immediately after the application thread returns from `OS_MAIN()`. Instead the Virtual Machine waits until a Terminate action is sent to the Application Manager.

If the `-slave` option is not used then the Virtual ECU loads in autostart mode. In autostart mode the application thread starts immediately after the `vrtaStart()` Virtual Machine API has been called.

In autostart mode the Virtual Machine terminates immediately after the application thread returns from `OS_MAIN()`.

15.2 vrtaServer

`vrtaServer` supports the command line options listed below.

15.2.1 -install (install as a service)

The `-install` option causes `vrtaServer` to install itself as a Windows service. Unless the `-silent` option is also specified a confirmation message will be displayed.

15.2.2 -lic=<licfile> (select a license file)

The `-lic=<licfile>` option tells `vrtaServer` to look in the license file `<licfile>` for a valid license.

15.2.3 -p<n> (specify the TCP port)

`vrtaServer` searches a pre-defined set of TCP ports for an empty port on which to listen for connections. The `-p<n>` option forces `vrtaServer` to listen on TCP port `<n>` for connections. `<n>` can be a decimal or hexadecimal (`0x` prefix) number.

15.2.4 -silent (silent install or uninstall)

Installation and un-installation as a Windows service normally generates a confirmation message. The `-silent` option stops the confirmation message being displayed. See `-install` and `-uninstall`.

15.2.5 -standalone (run in standalone mode)

The `-standalone` option causes `vrtaServer` to run as a standalone Windows application rather than as a service. If you want to manually run `vrtaServer` rather than installing it as a service then use the `-standalone` option.

15.2.6 -start (start the vrtaServer service)

If `vrtaServer` is installed as service but has not been started then the `–start` option will cause the service to be started. You do not normally have to start the service yourself – a VECU or vrtaMonitor will start the service if it needs to.

15.2.7 -stop (stop the vrtaServer service)

If `vrtaServer` is installed as service and has been started then the `–stop` option will cause the service to be stopped.

15.2.8 -uninstall (un-install as a service)

The `-uninstall` option causes `vrtaServer` to uninstall itself as a Windows service. If the `–silent` option is not also specified a confirmation message will be displayed.

15.3 vrtaMonitor

`vrtaMonitor` supports two sets of command line options. The *global* options affect the overall operation of `vrtaMonitor`. It does not matter where the global options appear in the command line.

The *sequential* options are processed in the order they appear on the command line.

15.3.1 Command Files

Command line options can be passed to `vrtaMonitor` in a command file. If `vrtaMonitor` encounters a statement of the form `@<file>` on its command line it will start processing command line options from the file `<file>`. Each option in `<file>` must be on a separate line.

15.4 vrtaMonitor Global Options

The following options affect the overall operation of `vrtaMonitor`.

15.4.1 -f<filename> (close when <filename> appears)

The `-f<filename>` options tells `vrtaMonitor` to run until the file `<filename>` appears and then to terminate.

15.4.2 -k (terminate with specific error level)

The `-k` option causes `vrtaMonitor` to terminate with a specific error level when certain events occur.

Error level	Event
1	Cannot connect to `vrtaServer` when auto-connection requested from the command line.
2	Cannot attach to an alias specified via `-alias`.
3	Cannot auto-load a specified Virtual ECU.
4	Closed as a result of a `-t` timeout.
5	Cannot load scripting engine.
6	Cannot run a script.
7	Failed to load.
8	Closed as a result of `-f`.
9	Failed to send an action or receive an event.

15.4.3 -lic=<licfile> (select a license file)

The `-lic=<licfile>` option tells `vrtaMonitor` to look in the license file `<licfile>` for a valid license.

15.4.4 -log=<file> (write to a log file)

The `-log=<file>` options causes `vrtaMonitor` to log activity to the file `<file>`.

15.4.5 -scripter=<name> (select a scripting engine)

The `-scripter=<name>` option selects the scripting engine called `rtaScript<name>.dll`.

15.4.6 -t<n> (close after <n> seconds)

The `-t<n>` options tells `vrtaMonitor` to run for `<n>` seconds and then terminate.

15.5 vrtaMonitor Sequential Options

The following options are processed in the order they appear on the command line.

15.5.1 Virtual ECU Executable Name (auto-load)

If `vrtaMonitor` encounters the name of a Virtual ECU executable on its command line it attempts to load the named Virtual ECU. The Virtual ECU executable may be on the local PC or a remote PC depending on whether or not the `–host` option has been used. The `–d`, `-r`, `-n` and `–g` options affect how the Virtual ECU is auto-loaded.

15.5.2 -alias=<name> (connect to VECU)

The `-alias=<name>` option tells `vrtaMonitor` to try and connect to an existing (loaded) Virtual ECU that has the alias `<name>`. By default `vrtaMonitor` assumes the Virtual ECU is on the local PC. If it is not the –`host` option should be used.

15.5.3 -d (load but not start a Virtual ECU)

The `–d` options causes the next Virtual ECU auto-loaded to be loaded but not started.

15.5.4 -g (load with a GUI)

The `–g` options causes the next Virtual ECU auto-loaded to run with an embedded GUI (i.e. run in GUI mode).

15.5.5 -host=<hostname> (select a remote PC)

The `–host=<hostname>` option selects the remote PC for the `–alias` and auto-load options. `<hostname>` is the host name of the remote PC.

15.5.6 -mon=<dev>.<event> (monitor event)

The `–mon=<dev>.<event>` options tells `vrtaMonitor` to monitor the event called `<event>` from the device called `<dev>` in the Virtual ECU to which `vrtaMonitor` has most recently attached (`-alias`) or auto-loaded.

15.5.7 –n (load without a GUI)

The `–n` options causes the next Virtual ECU auto-loaded to run without an embedded GUI (i.e. run in silent mode). This is the default when auto-loading a Virtual ECU.

15.5.8 -p<n> (select TCP port)

The `–p<n>` option tells `vrtaMonitor` that `vrtaServer` is listening on port `<n>`. By default `vrtaMonitor` looks for `vrtaServer` on a set of pre-defined TCP port numbers. `<n>` may be a decimal or hexadecimal (`0x` prefix) number.

15.5.9 -quit (terminate)

The `–quit` option causes `vrtaMonitor` to terminate.

15.5.10 –r (load and start a VECU)

The `–r` options causes the next Virtual ECU auto-loaded to be loaded and started. This is the default when auto-loading a Virtual ECU.

15.5.11 -script=<file>

The `–script=<file>` option causes `vrtaMonitor` to run the script in the file `<file>`. Please contact LiveDevices if you need information on monitor scripts.

15.5.12 -send=<dev>.<act> (send action)

The `-send=<dev>.<act>` options tells `vrtaMonitor` to send the data-less action called `<act>` to the device called `<dev>` in the Virtual ECU to which `vrtaMonitor` has most recently attached (`-alias`) or auto-loaded.

15.5.13 -send=<dev>.<act>(<str>) (send action)

The `-send=<dev>.<act>(<str>)` options tells `vrtaMonitor` to send the action called `<act>` to the device called `<dev>` in the Virtual ECU to which `vrtaMonitor` has most recently attached (`-alias`) or auto-loaded. The string `<str>` is sent as action data.

15.5.14 -start (start a VECU)

The `-start` option tells `vrtaMonitor` to send a Start action to the Application Manager of the next Virtual ECU auto-loaded.

15.5.15 -wait=<n> (wait)

The `-wait=<n>` option causes `vrtaMonitor` to wait for approximately `<n>` milliseconds before processing the next command line option.

16 Windows Notes

Although *RTA-OSEK for PC* tries very hard to simulate the behavior of a real ECU, ultimately Virtual ECUs are running under Windows alongside other applications. This chapter contains notes about Windows related behavior that developers may find useful.

16.1 Real-Time Behavior

When an embedded application runs on a real ECU the application is the only code using the ECU's processor. As a result the real-time behavior is predictable. However an application running in a Virtual ECU has to share the processor with other applications and Windows itself. As a result it is not possible to completely predict the real-time behavior of applications running in Virtual ECUs.

Despite this, on the whole our experience has shown that applications running in Virtual ECUs exhibit very close to real-time behavior. This is due to the very fast processor speeds of Windows PCs. For example, consider an embedded application that needs to read and then process input from a sensor every 5 milliseconds. On a real ECU it might take almost 5 milliseconds to carry out this activity. However on a Windows PC it may only take 0.5 milliseconds. Thus even if Windows assigns the processor to another application for 3 milliseconds the Virtual ECU application can still carry out the necessary processing in the 5 milliseconds allowed.

If your application is not behaving as you expect for timing reasons you can try the following:

- Shutdown other Windows applications so that more of the processor's time can be dedicated to the Virtual ECU.
- Increase the process priority of the Virtual ECU – see the `-priority=<n>` command line options.

16.2 Calling the C/C++ Runtime and Windows

In order to simulate interrupts in a Virtual ECU, the Virtual Machine has to asynchronously manipulate the stack of the application thread (the thread that calls `OS_MAIN()`). Few C/C++ runtime functions or Windows API functions can cope with the stack being changed asynchronously. Therefore if the application thread needs to call a C/C++ runtime function (including `printf()`) or a Windows API function it must make the call in an uninterruptible section. See the `vrtaEnterUninterruptibleSection()` and `vrtaLeaveUninterruptibleSection()` calls.

16.3 vrtaVMxxx.dll Location

When a Virtual ECU is started it tries to load the appropriate VM DLL (`vrtaVMs.dll` etc.). The VECU first tries to load the VM DLL using the normal DLL search rules. That is, it searches the following locations in the specified order:

1. The directory containing the VECU.
2. The current directory.
3. The 32-bit Windows system directory.
4. The Windows directory.
5. The directories listed in the `PATH` environment variable.

If the VECU fails to find the VM DLL then by default it tries to load it from the directory `c:\rta\bin`.

Therefore, if you have installed *RTA-OSEK for PC* in `c:\rta` (the default location) a VECU will always be able to find the appropriate VM DLL. If you have installed *RTA-OSEK for PC* in a different location then there are three ways of ensuring that VECUs can find VM DLLs:

- Add the directory `<rta>\bin` to the `PATH` environment variable. Where `<rta>` is the RTA-OSEK installation directory.
- Edit the file `<rta>\vrta\inc\vrtaCore.cpp` and set the `DLL_SEARCH` #define to be the directory containing the VM DLLs (i.e. `<rta>\bin`).
- Define the `DLL_SEARCH` macro to be the directory containing the VM DLLs when `vrtaCore.cpp` or `osgen.cpp` is compiled. For example, define the `DLL_SEARCH` macro using a compiler command line option.

Glossary

Action	An action is a command sent to a virtual device.
ApplicationManager (AM)	The virtual device that controls aspects of the application thread.
Application thread	This is the Windows thread that runs the Virtual ECU application code – including OSEK tasks. The application thread is created when the `vrtaStart()` API is called. The application thread's entry point is the function called `OS_MAIN()`.
AUTOSAR	AUTOSAR is a partnership that is seeking to establish an open standard for automotive E/E architecture. See http://www.autosar.org
Autostart mode	In autostart mode the application thread starts immediately after the `vrtaStart()` Virtual Machine API has been called and the Virtual Machine terminates automatically after the application thread returns from `OS_MAIN()`.
Diagnostic interface	A Virtual ECU has a diagnostic interface that can be used by external applications to monitor and manage the Virtual ECU. This diagnostic interface uses TCP/IP. The `vrtaServer` application keeps track of the port numbers used by Virtual ECUs on its local PC.
ECU	Electronic Control Unit
Event	An event is a signal generated (raised) by a virtual device to inform interested parties that something has happened. Events may or may not contain data.
GUI mode	In GUI mode the Virtual ECU displays its own GUI.
Interrupt Control Unit (ICU)	The virtual device that controls aspects of the application's virtual interrupts.
OSEK^{TM}	OSEK is a registered trademark of Siemens AG. It was founded in May 1993 as a joint project in the German automotive industry. Its aims are to provide an "industry standard for an open-ended architecture for distributed control units in vehicles." OSEK is an abbreviation for "Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug", which translates as Open Systems and the Corresponding Interfaces for Automotive Electronics.

	See http://www.osek-vdx.org/
Silent mode	In silent mode the Virtual ECU does not display its own GUI.
Slave mode	In slave mode the application thread is not started immediately after the `vrtaStart()` Virtual Machine API has been called. Instead the Virtual Machine waits until a Start action is sent to the Device Manager before starting the application thread. In *slave* mode the Virtual Machine does not terminate immediately after the application thread returns from `OS_MAIN()`. Instead the Virtual Machine waits until a Terminate action is sent to the Device Manager.
Virtual devices	Virtual devices are software components within Virtual ECUs that simulate hardware devices. Virtual devices include: clocks, counters, sensors, actuators and CAN controllers.
Virtual ECU (VECU)	A Virtual ECU is composed of an application, the Virtual Machine, virtual devices and possibly an RTA-OSEK kernel. A Virtual ECU is the *RTA-OSEK for PC* analog of an application running on a real ECU.
Virtual Machine (VM)	The Virtual Machine is the component of *RTA-OSEK for PC* that simulates an ECU. The Virtual Machine simulates interrupts, manages virtual devices, manages the application thread and manages communication with external applications. The Virtual Machine is supplied as a DLL. Virtual ECU start-up code links to the Virtual Machine DLL at runtime.
vrtaMonitor	The `vrtaMonitor(.exe)` program is an application supplied with *RTA-OSEK for PC* that allows Virtual ECUs running on the same or a remote PC to be monitored and managed.
VRTA	Virtual RTA-OSEK : a short form used for the *RTA-OSEK for PC* product.
vrtaServer	The `vrtaServer(.exe)` program is an application supplied with *RTA-OSEK for PC* that manages access to Virtual ECU. It runs as a normal application or as a Windows service.

Index