
RTA-OS3.0

Virtual ECU User Guide

Copyright

The data in this document may not be altered or amended without special notification from ETAS GmbH. ETAS GmbH undertakes no further obligation in relation to this document. The software described in it can only be used if the customer is in possession of a general license agreement or single license. Using and copying is only allowed in concurrence with the specifications stipulated in the contract. Under no circumstances may any part of this document be copied, reproduced, transmitted, stored in a retrieval system or translated into another language without the express written permission of ETAS GmbH.

©Copyright 2008 ETAS GmbH, Stuttgart.

The names and designations used in this document are trademarks or brands belonging to the respective owners.

Document: 10384(VRTA)-VEUG-1.0.0

Contents

1	Welcome to the RTA-OS3.0 Virtual ECU!	20
1.1	Related Documents	20
1.2	About You	21
1.3	Document Conventions	21
1.4	References	22
2	Introduction to the Virtual ECU	23
2.1	What do I need?	23
2.2	What is the Virtual Machine?	25
2.2.1	Device Manager	25
2.2.2	Interrupt Control Unit	27
2.2.3	Application Manager	27
2.2.4	Embedded GUI	28
2.2.5	Linkage Table	28
2.3	What is in a Virtual ECU?	28
2.4	Managing Multiple Virtual ECUs	29
2.5	Interacting with a Virtual ECU	29
2.6	Debugging	29
2.7	Possible Problem Areas	31
3	Tutorial	33
3.1	Prerequisites	33
3.1.1	RTA-TRACE	33
3.2	Creating your first Virtual ECU: Part1	33
3.2.1	Configuring RTA-OS3.0	34
3.2.2	Building RTA-OS3.0	35

3.2.3	Writing Application Code	36
3.2.4	Writing VECU Code	37
3.2.5	Building the Virtual ECU	37
3.2.6	Building the Virtual ECU	38
3.3	Adding Devices	38
3.3.1	Clocks, Counters and Compare Devices	39
3.3.2	Sensors	40
3.3.3	Actuators	41
3.3.4	IO	42
3.3.5	Custom Devices	42
3.4	Creating your first Virtual ECU: Part2	42
3.4.1	Devices	43
3.4.2	Logger	44
3.4.3	Interfacing C code with C++ Devices	45
3.4.4	Device Initialization	46
3.4.5	The main() program	46
3.4.6	Trial Run 1	47
3.4.7	Summary so far	49
3.4.8	Adding Tasks and ISRs	50
3.4.9	Threads	53
3.4.10	Trial run 2	55
3.4.11	Linking to Real Hardware	56
3.4.12	Non-volatile Data	58
3.4.13	RTA-TRACE	60
3.5	Summary	61

4	ECUs and Virtual Devices	63
4.1	Registering the Device	63
4.2	Handling actions	64
4.3	Handling State Queries	64
4.4	Raising Events	65
4.5	Raising Interrupts	65
4.6	Parent/Child relationships	66
4.7	Threads	68
5	Interacting with VECUs	69
5.1	Running vrtaServer	69
5.1.1	Security issues	72
5.2	Using vrtaMonitor	73
5.2.1	Actions	75
5.2.2	Events	75
5.3	Multiple instances of a VECU	77
5.4	Scripting using vrtaMonitor	78
5.4.1	Example Scripts	78
6	RTA-TRACE Integration	81
6.1	How it works	81
6.1.1	The Virtual ECU	81
6.1.2	RTA-TRACE-Server	82
6.2	Tuning process and thread priorities	82
6.3	Controlling the trace at run-time	83
7	Windows Notes	84
7.1	Real-Time Behavior	84

7.2	Calling the C/C++ Runtime and Windows	84
7.3	Virtual Machine Location	85
8	Migrating from a VECU to Real Hardware	86
8.1	XML file	86
8.1.1	Target and variant	86
8.1.2	Interrupts	86
8.1.3	Number of tasks	87
8.2	Hardware Drivers	87
8.3	Initialization	87
8.4	Interrupts	88
8.5	Register Sets	88
9	Virtual Machine API Reference	89
9.1	General notes	89
9.1.1	API Header Files	89
9.1.2	Linkage	89
9.2	Common Data Types	89
9.2.1	vrtaActEVID	89
9.2.2	vrtaAction	89
9.2.3	vrtaActionID	90
9.2.4	vrtaBoolean	90
9.2.5	vrtaByte	90
9.2.6	vrtaDevID	91
9.2.7	vrtaDataLen	91
9.2.8	vrtaEmbed	91
9.2.9	vrtaErrType	91

9.2.10	vrtaEvent	92
9.2.11	vrtaEventID	93
9.2.12	vrtaIntPriority	93
9.2.13	vrtaSRID	93
9.2.14	vrtaMillisecond	93
9.2.15	vrtaOptStringlistPtr	93
9.2.16	vrtaStringlistPtr	93
9.2.17	vrtaTextPtr	93
9.2.18	vrtaTimestamp	94
9.3	Data Format Strings	94
9.3.1	Overview	94
9.3.2	Definition	94
9.3.3	Examples	96
9.4	API Functions	97
9.4.1	InitializeDevices()	99
9.4.2	OS_MAIN()	100
9.4.3	vrtaEnterUninterruptibleSection()	102
9.4.4	vrtaEventRegister()	103
9.4.5	vrtaEventUnregister()	105
9.4.6	vrtaGetState()	106
9.4.7	vrtaHookEvent()	108
9.4.8	vrtaInitialize()	110
9.4.9	vrtalsAppFinished()	113
9.4.10	vrtalsAppThread()	114
9.4.11	vrtalsIdle()	115
9.4.12	vrtaLeaveUninterruptibleSection()	116

9.4.13	vrtaLoadVM()	117
9.4.14	vrtaRaiseEvent()	118
9.4.15	vrtaReadHPTime()	119
9.4.16	vrtaRegisterVirtualDevice()	120
9.4.17	vrtaReset()	125
9.4.18	vrtaSendAction()	126
9.4.19	vrtaSpawnThread()	127
9.4.20	vrtaStart()	128
9.4.21	vrtaTerminate()	129
10 Standard Devices (vrtaStdDevices.h)		130
10.1	Action and Event Descriptions	130
10.2	Device Manager	131
10.2.1	Action: EventRegister	131
10.2.2	Action: HookEvents	131
10.2.3	Action: ListAll	132
10.2.4	Action: GetDeviceActions	132
10.2.5	Action: GetDeviceEvents	132
10.2.6	Action: GetDeviceInfo	132
10.2.7	Event: DeviceList	132
10.2.8	Event: DeviceActions	132
10.2.9	Event: DeviceEvents	133
10.2.10	Event: DeviceInfo	133
10.3	Interrupt Control Unit	133
10.3.1	Action: Raise	134
10.3.2	Action: Clear	134

10.3.3	Action: Mask	134
10.3.4	Action: Unmask	135
10.3.5	Action: GetPending	135
10.3.6	Action: GetIPL	135
10.3.7	Action: SetIPL	135
10.3.8	Event: Pending	135
10.3.9	Event: Start	135
10.3.10	Event: Stop	136
10.3.11	Event: IPL	136
10.3.12	Event: EnabledVecs	136
10.4	Application Manager	136
10.4.1	Action: Start	136
10.4.2	Action: Terminate	136
10.4.3	Action: Pause	137
10.4.4	Action: Restart	137
10.4.5	Action: Reset	137
10.4.6	Action: GetInfo	137
10.4.7	Action: TestOption	137
10.4.8	Action: ReadOption	137
10.4.9	Action: ReadParam	138
10.4.10	Event: Started	138
10.4.11	Event: Paused	138
10.4.12	Event: Restarted	138
10.4.13	Event: Reset	138
10.4.14	Event: Terminated	138
10.4.15	Event: Info	138

10.4.16	Event: Option	139
10.4.17	Event: OptionText	139
10.4.18	Event: ParamText	139
10.4.19	Event: State	139
11 Sample Devices (vrtaSampleDevices.h)		140
11.1	Device Descriptions	140
11.1.1	Methods	140
11.1.2	Actions and Events	141
11.2	vrtaClock	142
11.2.1	Method: vrtaClock()	142
11.2.2	Method: SetInterval()	142
11.2.3	Method: SetScale()	143
11.2.4	Method: Start()	143
11.2.5	Method: Stop()	144
11.2.6	Action: Interval	144
11.2.7	Action: Scale	144
11.2.8	Action: Start	144
11.2.9	Action: Stop	144
11.2.10	Event: Interval	145
11.2.11	Event: Scale	145
11.2.12	Event: Running	145
11.3	vrtaUpCounter	146
11.3.1	Method: vrtaUpCounter()	146
11.3.2	Method: Min()	146
11.3.3	Method: Max()	147

11.3.4	Method: Value()	147
11.3.5	Method: SetMin()	147
11.3.6	Method: SetMax()	148
11.3.7	Method: SetVal()	148
11.3.8	Method: Start()	149
11.3.9	Method: Stop()	149
11.3.10	Action: Minimum	150
11.3.11	Action: Maximum	150
11.3.12	Action: Set	150
11.3.13	Action: Start	150
11.3.14	Action: Stop	150
11.3.15	Action: Report	150
11.3.16	Event: Set	151
11.4	vrtcDownCounter	152
11.4.1	Method: vrtcDownCounter()	152
11.4.2	Method: Min()	152
11.4.3	Method: Max()	153
11.4.4	Method: Value()	153
11.4.5	Method: SetMin()	153
11.4.6	Method: SetMax()	154
11.4.7	Method: SetVal()	154
11.4.8	Method: Start()	155
11.4.9	Method: Stop()	155
11.4.10	Action: Minimum	156
11.4.11	Action: Maximum	156
11.4.12	Action: Set	156

11.4.13	Action: Start	156
11.4.14	Action: Stop	156
11.4.15	Action: Report	156
11.4.16	Event: Set	157
11.5	vrtaSensor	158
11.5.1	Method: vrtaSensor()	158
11.5.2	Method: GetMax()	158
11.5.3	Method: Value()	159
11.5.4	Method: SetMax()	159
11.5.5	Method: SetVal()	159
11.5.6	Action: Value	160
11.5.7	Action: Maximum	160
11.5.8	Event: Value	160
11.5.9	Event: Maximum	160
11.6	vrtaSensorToggleSwitch	161
11.6.1	Method: vrtaSensorToggleSwitch()	161
11.6.2	Method: Value()	161
11.6.3	Method: SetVal()	161
11.6.4	Action: Position	162
11.6.5	Event: Position	162
11.7	vrtaSensorMultiwaySwitch	163
11.7.1	Method: vrtaSensorMultiwaySwitch()	163
11.7.2	Method: GetMax()	163
11.7.3	Method: Value()	164
11.7.4	Method: SetMax()	164
11.7.5	Method: SetVal()	164

	11.7.6	Action: Value	165
	11.7.7	Action: Maximum	165
	11.7.8	Event: Value	165
	11.7.9	Event: Maximum	165
11.8		vrtaActuator	166
	11.8.1	Method: vrtaActuator()	166
	11.8.2	Method: GetMax()	166
	11.8.3	Method: Value()	167
	11.8.4	Method: SetMax()	167
	11.8.5	Method: SetVal()	167
	11.8.6	Action: Value	168
	11.8.7	Action: Maximum	168
	11.8.8	Event: Value	168
	11.8.9	Event: Maximum	168
11.9		vrtaActuatorLight	169
	11.9.1	Method: vrtaActuatorLight()	169
	11.9.2	Method: Value()	169
	11.9.3	Method: SetVal()	169
	11.9.4	Action: Value	170
	11.9.5	Event: Value	170
11.10		vrtaActuatorDimmableLight	171
	11.10.1	Method: vrtaActuatorDimmableLight()	171
	11.10.2	Method: GetMax()	171
	11.10.3	Method: Value()	172
	11.10.4	Method: SetMax()	172
	11.10.5	Method: SetVal()	172

11.10.6	Action: Value	173
11.10.7	Action: Maximum	173
11.10.8	Event: Value	173
11.10.9	Event: Maximum	173
11.11	vrtaActuatorMultiColorLight	174
11.11.1	Method: vrtaActuatorMultiColorLight()	174
11.11.2	Method: GetMax()	174
11.11.3	Method: Value()	175
11.11.4	Method: SetMax()	175
11.11.5	Method: SetVal()	175
11.11.6	Action: Value	176
11.11.7	Action: Maximum	176
11.11.8	Event: Value	176
11.11.9	Event: Maximum	176
11.12	vrtaCompare	177
11.12.1	Method: vrtaCompare()	177
11.12.2	Method: GetMatch()	178
11.12.3	Method: SetMatch()	178
11.12.4	Method: IncrementMatch()	179
11.12.5	Method: SetVector()	179
11.12.6	Action: Match	179
11.12.7	Action: Vector	180
11.12.8	Event: Match	180
11.13	vrtaIO	181
11.13.1	Method: vrtaIO()	181
11.13.2	Method: SetValue()	181

11.13.3	Method: SetValue()	181
11.13.4	Method: GetValue()	182
11.13.5	Method: GetValues()	182
11.13.6	Action: Value	183
11.13.7	Action: Values	183
11.13.8	Action: GetValue	183
11.13.9	Action: GetValues	183
11.13.10	Event: Value	183
11.13.11	Event: Values	184
11.14	Rebuilding from Source Code	184
12 Command Line		185
12.1	<VirtualECU>.exe	185
12.2	vrtaServer	186
12.3	vrtaMonitor	187
12.3.1	Global Options	188
12.3.2	Sequential Options	188
12.3.3	Command Files	190
13 Virtual ECU Server Library		191
13.1	Using the DLL	191
13.2	Using the Source Code	191
13.3	Virtual ECU Aliases	191
13.4	Types	192
13.4.1	VesLibEcuInfoType	192
13.4.2	VesLibEcuAliasType	192
13.5	The API Call Template	192

13.6	VesLibAttachToECU()	194
13.7	VesLibCreateAlias()	195
13.8	VesLibExit()	196
13.9	VesLibFindECUs()	197
13.10	VesLibFreeAlias()	199
13.11	VesLibFreeMemory()	200
13.12	VesLibGetAliases()	201
13.13	VesLibGetInfo()	203
13.14	VesLibInitialize()	204
13.15	VesLibListAliases()	205
13.16	VesLibListLoadedECUs()	206
13.17	VesLibLoadECU()	207
13.18	VesLibSelectServer()	209
14 COM Bridge Tutorial		211
14.1	Example	211
14.1.1	CVcServer	211
14.2	CVcECU	213
14.2.1	CVcDevice, CVcAction and CVcEvent	215
14.3	Tutorial	216
14.3.1	Setting up the project	217
14.3.2	Connecting to vrtaServer	217
14.3.3	Connecting to the VECU	218
14.3.4	Initializing the devices	219
14.3.5	Reacting to events	220
14.3.6	Sending actions	221

14.3.7	Summary	221
15 COM Bridge Reference		223
15.1	CVcServer	224
15.2	ICVcServer	225
15.2.1	Enum: IVcServer_DisplayMode	225
15.2.2	Enum: IVcServer_StartMode	225
15.2.3	Enum: IVcServer_Status	225
15.2.4	Method: AttachECU()	226
15.2.5	Method: Connect()	227
15.2.6	Method: CreateAlias()	228
15.2.7	Method: Disconnect()	229
15.2.8	Method: FindECUs()	230
15.2.9	Method: FreeAlias()	231
15.2.10	Method: GetAliases()	232
15.2.11	Method: GetInfo()	233
15.2.12	Method: ListAliases()	234
15.2.13	Method: ListLoadedAliases()	235
15.2.14	Method: LoadECU()	236
15.2.15	Method: ServerStatus()	237
15.3	CVcECU	238
15.4	ICVcECU	239
15.4.1	Enum: IVcECU_Status	239
15.4.2	Method: Connect()	240
15.4.3	Method: Disconnect()	241
15.4.4	Method: DoAction()	242

15.4.5	Method: GetDeviceByID()	243
15.4.6	Method: GetDeviceByName()	244
15.4.7	Method: GetDeviceCount()	245
15.4.8	Method: Hook()	246
15.4.9	Method: QueryEvent()	247
15.4.10	Method: QueryFormat()	248
15.4.11	Method: ReplyFormat()	249
15.4.12	Method: SendAction()	250
15.4.13	Method: SendFormat()	251
15.5	ICVcECUEvents	252
15.5.1	Method: OnEventChange()	252
15.6	CVcDevice	253
15.7	ICVcDevice	254
15.7.1	Method: DeviceID()	254
15.7.2	Method: DoAction()	255
15.7.3	Method: GetActionByID()	256
15.7.4	Method: GetActionByName()	257
15.7.5	Method: GetActionCount()	258
15.7.6	Method: GetEventByID()	259
15.7.7	Method: GetEventByName()	260
15.7.8	Method: GetEventCount()	261
15.7.9	Method: Hook()	262
15.7.10	Method: Name()	263
15.7.11	Method: QueryEvent()	264
15.7.12	Method: QueryFormat()	265
15.7.13	Method: ReplyFormat()	266

15.7.14	Method: SendAction()	267
15.7.15	Method: SendFormat()	268
15.8	ICVcDeviceEvents	269
15.8.1	Method: OnEventChange()	269
15.9	CVcAction	270
15.10	ICVcAction	271
15.10.1	Method: ActionID()	271
15.10.2	Method: Do()	272
15.10.3	Method: Name()	273
15.10.4	Method: Send()	274
15.10.5	Method: SendFormat()	275
15.11	CVcEvent	276
15.12	ICVcEvent	277
15.12.1	Method: EventID()	277
15.12.2	Method: Hook()	278
15.12.3	Method: Name()	279
15.12.4	Method: Query()	280
15.12.5	Method: QueryFormat()	281
15.12.6	Method: ReplyFormat()	282
15.13	ICVcEventEvents	283
15.13.1	Method: OnEventChange()	283
16	Glossary	284
17	Contacting ETAS	286
17.1	Technical Support	286
17.2	General Enquiries	287

1 **Welcome to the RTA-OS3.0 Virtual ECU!**

The RTA-OS3.0 Virtual ECU is a complete environment for developing AUTOSAR OS applications hosted on a Windows PC. 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 use RTA-OS3.0 to develop applications for embedded targets.

But you needn't stop there. Because RTA-OS3.0 for the Virtual ECU is a complete and fast implementation of OSEK, you also add inter-application communication using a standard networking solution such as CAN. 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 supplied Virtual ECU monitor program, or use ETAS's 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.

The guide is structured as follows:

Chapter 2 introduces you to the Virtual ECU, covering what tools are provided, which standards are supported by the kernel and gives a brief overview of kernel features.

Chapter 3 takes you through a tutorial that explains how to build, run and monitor a Virtual ECU.

Chapters 4 to 8 provide detailed information about how to work with a Virtual ECU, including how to interface to RTA-TRACE and how to migrate from a Virtual ECU to an embedded target.

Chapters 9 to 11 provide a technical reference for creating a Virtual ECU and interacting with it from your application code. The sample devices that are automatically provided for you are also documented here.

Chapters 12 to 15 provide a reference to [vrtaMonitor](#) and [vrtaServer](#) - the tools that you can use to interact with a Virtual ECU at runtime, and discuss how to write your own tools to interact with a Virtual ECU and how write your own Windows applications to interact with the ECU provide a complete technical reference to all aspects of Virtual ECU development.

Chapter 16 explains commonly used terms.

1.1 **Related Documents**

This guide does not tell you how to configure and use RTA-OS3.0, this information is provided in the *RTA-OS3.0 User Guide*. A complete technical reference

to RTA-OS3.0 can be found in the *RTA-OS3.0 Reference Guide*. Both of these documents can be found in the Documents folder of your RTA-OS3.0 installation. For a default installation, this will be C:\ETAS\RTA-OS3.0\Documents but your local installation may differ.

Specific technical details about the implementation of RTA-OS3.0 for the Virtual ECU is contained in the associated *RTA-OS3.0 VRTA Port Guide*.


1.2 About You

You are a trained software engineer who wants to build real-time applications using a pre-emptive operating system. You should have knowledge of the C and C++ programming languages, including the compilation and linking of C and C++ code for PC-based applications with a PC-hosted development environment. Advanced features of the RTA-OS3.0 Virtual ECU will require knowledge about Windows COM Component Object Model) applications.

You should also be familiar with common use of the Microsoft Windows® 2000, Windows® XP or Windows® Vista operating systems, including installing software, selecting menu items, clicking buttons, navigating files and directories.

1.3 Document Conventions

The following conventions are used in this guide:

Choose File > Open .	Menu options are printed in bold, blue characters.
Click OK .	Button labels are printed in bold characters
Press <Enter>.	Key commands are enclosed in angle brackets.
The "Open file" dialog box appears	The names of program windows, dialog boxes, fields, etc. are enclosed in double quotes.
Activate(Task1)	Program code, header file names, C type names, C functions and RTA-OS3.0. Component API call names all appear in the courier typeface.
See Section 1.3 .	Hyperlinks through the document are shown in red letters .
	Functionality that is provided in RTA-OS3.0 but it may not be portable to another AUTOSAR OS implementation is marked with the ETAS logo.



Caution! Notes like this contain important instructions that you must follow carefully in order for things to work correctly.

1.4 References

OSEK is a European automotive industry standards effort to produce open systems interfaces for vehicle electronics. For details of the OSEK standards, please refer to:

<http://www.osek-vdx.org>

AUTOSAR (AUTomotive Open System ARchitecture) is an open and standardized automotive software architecture, jointly developed by automobile manufacturers, suppliers and tool developers. For details of the AUTOSAR standards, please refer to:

<http://www.autosar.org>

2 Introduction to the Virtual ECU

RTA-OS3.0 is commonly used in automotive environments where the term ECU (Electronic Control Unit) is used to refer to the target hardware on which the application runs. The ECU can be considered as a black box 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 etc. Many if not all will be running OSEK or AUTOSAR applications.

The RTA-OS3.0 VRTA port is different. It does not need any real hardware, other than the host PC that you run it on. 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 physical ECU. These devices are built around a core Virtual Machine (VM) that provides services such as the interrupt controller, application control and diagnostic links as shown in Figure 2.1.

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', a little like a physical ECU which has a microcontroller as a 'core' and additional hardware either as microcontroller peripherals or as special discrete devices inside the ECU itself.

The VM provides a diagnostic interface (via TCP/IP) that allows external programs to interact with a VECU. The VRTA port ships with a program called **vrtaMonitor** that can monitor and manage VECUs. You can also build your own applications to interact with VECUs using a Component Object Model (COM) interface called the COM Bridge. The COM bridge allows COM clients to interact with VECUs and is supplied as DLL called `vrtaMSCOM.dll` that ships with your RTA-OS3.0 VRTA port

The string `vrta` is used to prefix executables such as `vrtaMonitor.exe`, `vrtaVM.dll` and the source files generated by **rtaosgen**. "vrta" is an abbreviation for Virtual RTA-OS3.0.

2.1 What do I need?

The VECUs that you build run under Microsoft Windows 2000 or later (including Windows XP and Windows Vista). They require 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 AUTOSAR OS R3.0 application many times faster than a typical embedded target.

You will need a Windows C++ Compiler to generate Virtual ECUs (and the associated debugger that comes with your compiler to debug your Virtual

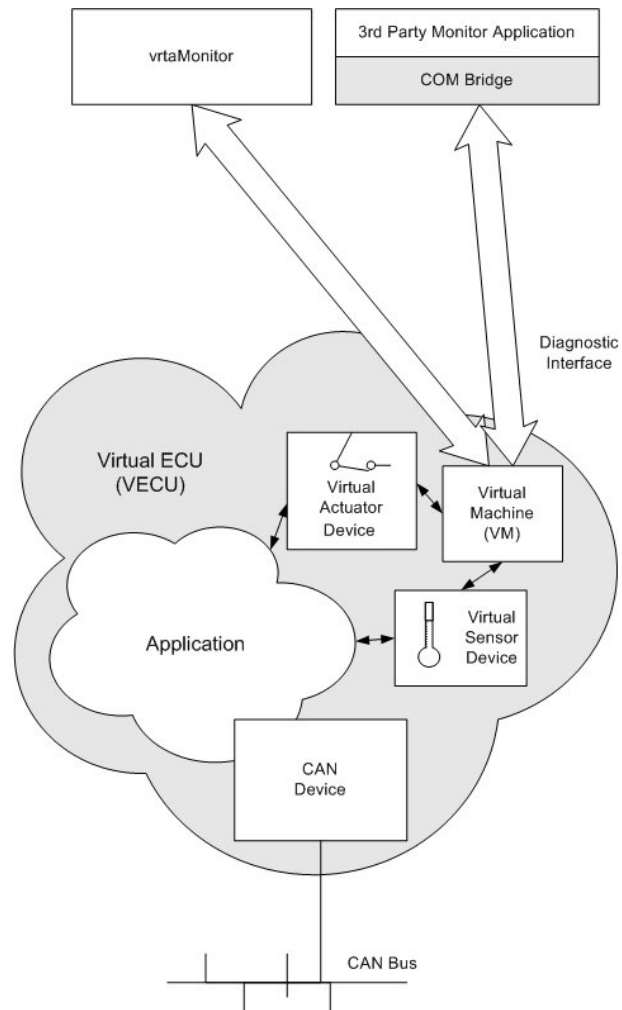


Figure 2.1: Virtual ECU Architecture

ECU code). Supported compilers are listed in the *RTA-OS3.0 VRTA Port Guide* and correspond to target variants for VRTA.

If you want to use an unsupported compiler then you should discuss your requirements with ETAS¹. Contact details can be found in Chapter 17.

2.2 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. Figure 2.2 shows the components that make up the VM. The components are explained in the following subsections.

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



Applications built for a Virtual ECU must be able to find `vrtavm.dll` on the Windows DLL search path (See Section for details). It is highly recommended that you add `C:\ETAS\RTA-OS3.0\Bin` to your Windows `PATH` environment variable to ensure that every Virtual ECU application can find the Virtual Machine.

2.2.1 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 as shown in Figure 2.3.

A device can be a simple representation of a switch or an LED, or it can rep-

¹While ETAS cannot guarantee that RTA-OS3.0 for VRTA will work any compiler you choose, 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** are always aligned on a 32 bit boundary and a **double** is always aligned on a 64 bit boundary.

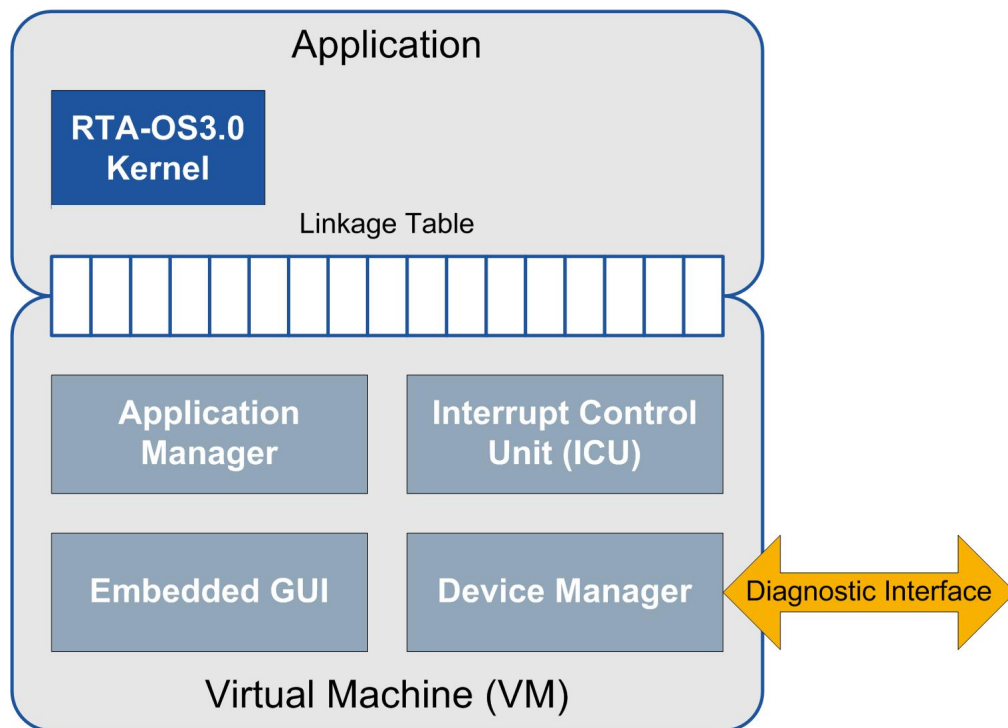


Figure 2.2: VM Architecture

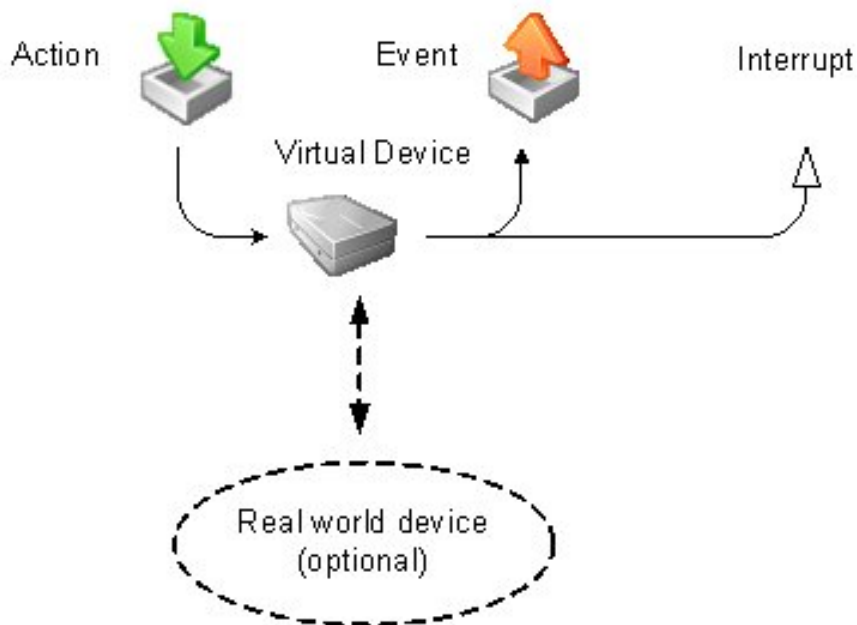


Figure 2.3: Virtual Devices

resent a complex component such as a PCMCIA-based CAN controller. Most operations in a VECU are performed by sending actions to or responding to events from devices.

The VM itself contains the three standard devices: the DeviceManager², 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 Section 5.2) to inspect the state of the application's devices.

2.2.2 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 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 \leq 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 of a VECU (i.e. when you start the <VECU>.exe program) all interrupts are masked.

2.2.3 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. 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.

²The Device Manager is itself also a device. You query it as device zero to find out what other devices are present.

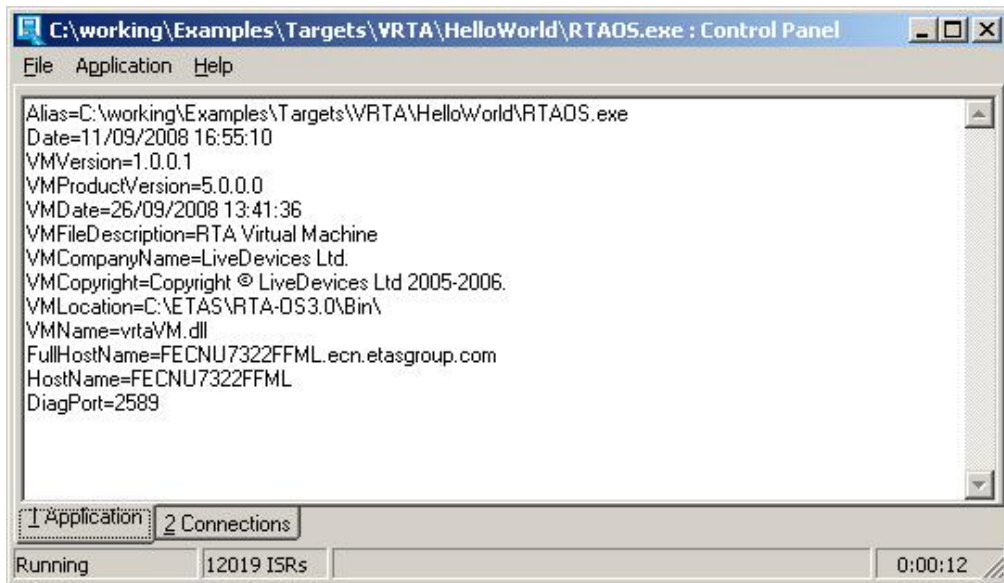


Figure 2.4: Embedded GUI

2.2.4 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 like that shown in Figure 2.4 that will give you a bit of confidence that the application is actually alive.

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.

2.2.5 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-OS3.0 for VRTA will ensure that the DLL is loaded and that API for the VM is made available.

2.3 What is in a Virtual ECU?

A Virtual ECU is the combination of the VM DLL and a Windows executable containing your application code. Your application will usually contain:

- a program that creates the inputs and outputs that represent a physical ECU using virtual devices that simulate real devices, or in some cases connect to real hardware.

- startup and linkage code that glues your application to the VM (this is generated for you automatically by **rtaosgen**).
- an RTA-OS3.0 element, namely the tasks and ISRs that you want to run in the virtual environment, built for the VRTA target.

2.4 Managing Multiple Virtual ECUs

You can 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.

To solve this problem, a server program called **vrtaServer** is provided that can run on the PC. VECUs can register with the server when they start up. The **vrtaMonitor** can then ask the **vrtaServer** which VECUs exist, and how to connect to them.



vrtaServer must be running in order for **vrtaMonitor** can connect to a VECU.

If you want to use **vrtaServer** on a regular basis then it should be installed as a Windows service so 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.

2.5 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. Figure 2.5 shows the **vrtaMonitor** GUI - it is discussed in detail in Section 5.2.

2.6 Debugging

You can use the PC debugger that comes with your compiler tool chain 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.

If you want to see how real-time interaction occurs then you can integrate an RTA-OS3.0 application with RTA-TRACE as shown in Figure 2.6. A VECU has to be built with RTA-TRACE enabled, but from then on the VECU will run as normal, only sending trace data out if RTA-TRACE is connected. RTA-TRACE

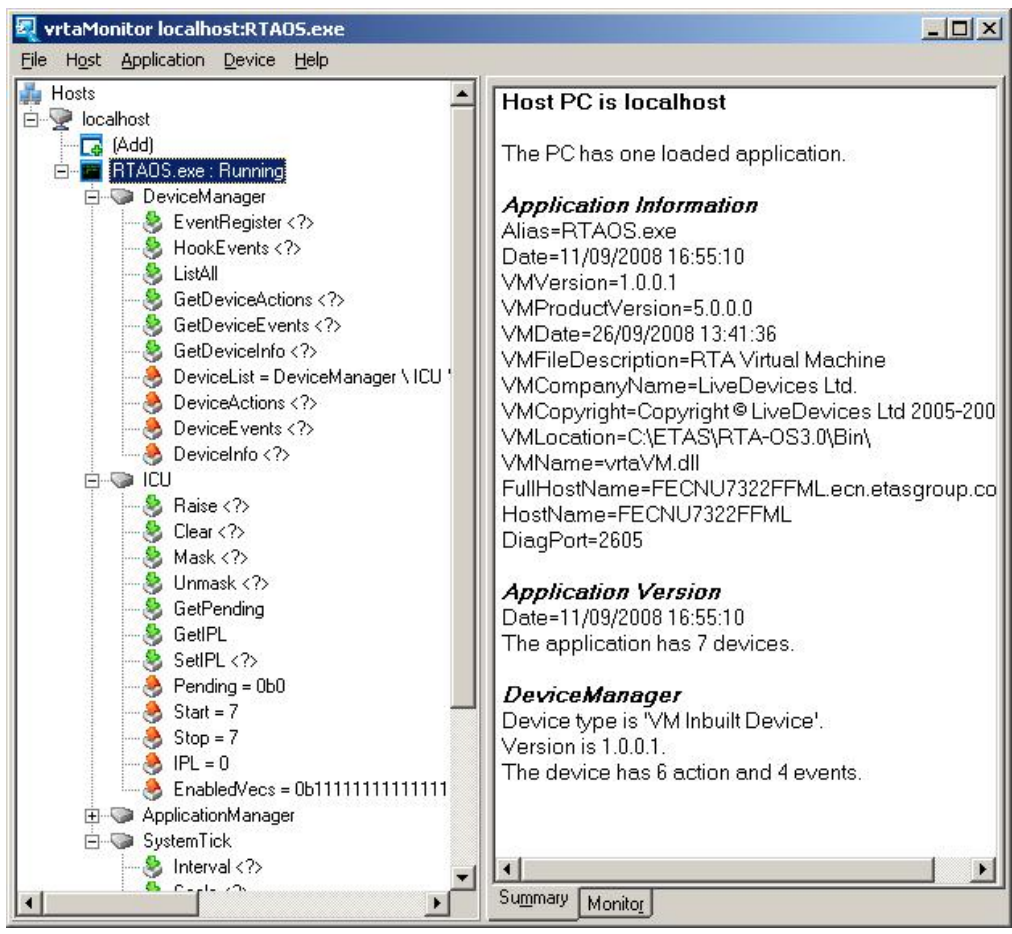


Figure 2.5: vrtMonitor

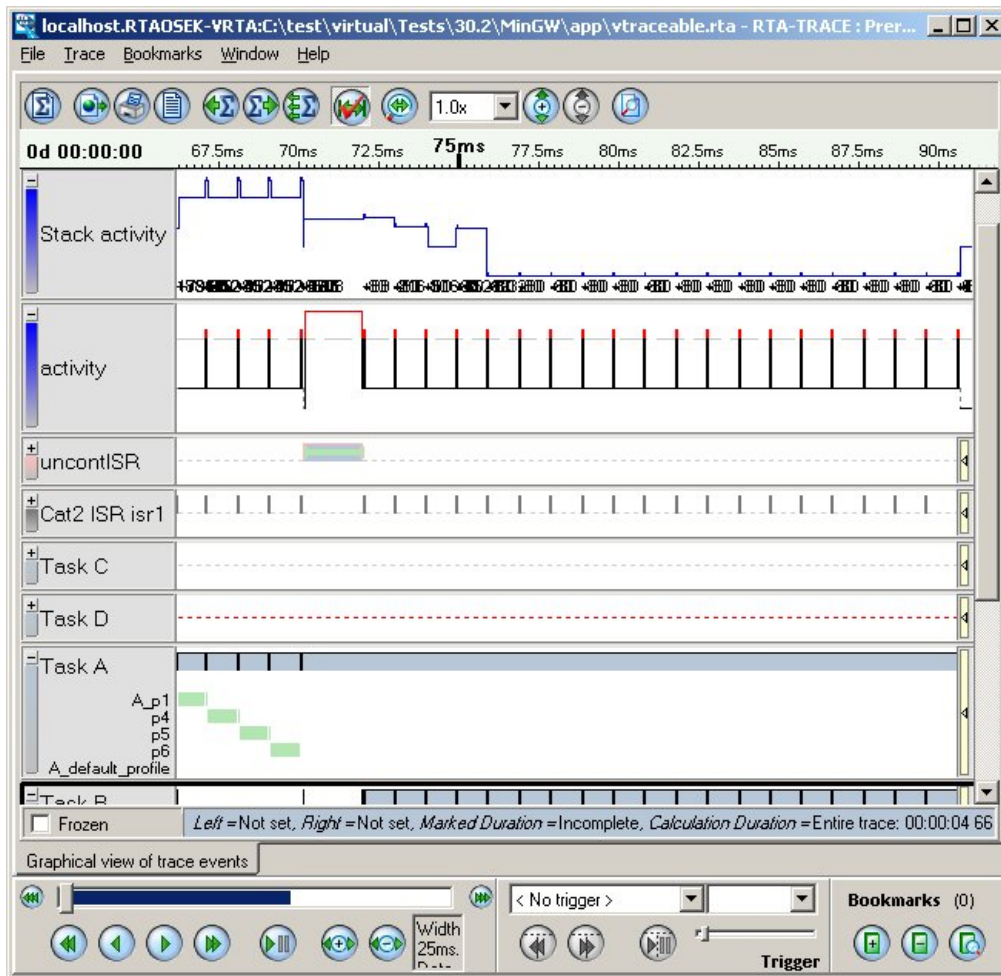


Figure 2.6: Tracing a Virtual ECU with RTA-TRACE

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.

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-AUTOSAR 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-AUTOSAR API calls must be protected by an uninterruptible section (see Section 9.4.3). See Section 7.2 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.)

- You can run your VECU but you can't see any tracing information? You need to have **vrtaServer** installed as a Windows service.
- Nothing appears to be happening? Check that the vrtaVM.dll is on the Windows DLL search path (see Section 7.3 for details).
- You cannot connect to a VECU using **vrtaMonitor**? Make sure that **vrtaServer** is running, either as a Windows service or as a standalone application.

3 Tutorial

The best way to get an understanding of RTA-OS3.0 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-OS3.0 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.

RTA-OS3.0

If you have followed the instructions in the *RTA-OS3.0 Getting Started Guide* then you have already installed the RTA-OS3.0 tools and the VRTA port plug-in. You will have obtained and installed your license. In this tutorial we will assume that you have installed RTA-OS3.0 to C:\ETAS\RTA-OS3.0, so the route for the RTA executables and DLLs is C:\ETAS\RTA-OS3.0\Bin. The files that are specific to RTA-OS3.0 for PC are found at C:\ETAS\RTA-OS3.0\Targets\VRTA_n.n.n.

Compiler

This tutorial will use the MinGW compiler that is freely available under the GNU license. See the "Installation" section of *RTA-OS3.0 VRTA Port Guide*.

In this tutorial we will assume that you have installed the compiler at C:\Program Files\MinGW.513.

3.1.1 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-OS3.0 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-OS3.0 then this link will be detected automatically. If not, then you must copy the file `rtcVRTAlink.dll` from RTA-OS3.0's Bin directory to RTA-TRACE's Bin directory.

3.2 Creating your first Virtual ECU: Part1

Let's go for the traditional 'Hello World' starter application - with an AUTOSAR OS twist. The following steps are required:

1. Configure RTA-OS3.0

2. Build the RTA-OS3.0 kernel library
3. Write the application code
4. Write the VECU code
5. Build the Virtual ECU

3.2.1 Configuring RTA-OS3.0

Start [rtaoscfg](#), the RTA-OS3.0 GUI¹.

You are now looking at an empty gray window which we can brighten up a bit by selecting menu option **File → New Project**. You can accept the default project values.

Basic Configuration

We need to add enough basic RTA-OS3.0 configuration to generate an OS kernel.

Step 1 In **OS Configuration → General** on the “General” tab:

Item	Setting
Scalability Class	SC1
Status	STANDARD
Enable Stack Monitoring	FALSE
Enable Time Monitoring	FALSE

Step 2 In **OS Configuration → General** on the “Hooks” tab click “Clear All”.

Step 3 In **OS Configuration → General** on the “Error Hook” tab click “Clear All”.

Step 4 In **OS Configuration → General → Target**:

Item	Setting
Target Selection → Name	VRTA
Target Selection → Variant	Set the compiler you will be using to build the application.
Clock Speeds → Instruction Cycle Rate (Hz)	1000000 (i.e. 1MHz).
Clock Speeds → Stopwatch Speed (Hz)	1000(i.e. 1kHz).

This has configured the target-specific aspects we need for RTA-OS3.0 for this tutorial.

¹For a default installation of RTA-OS3.0 this will be located in C:\ETAS\RTA-OS3.0\Bin.

Step 5 In **OS Configuration → General → Optimizations** on the “AUTOSAR” set “Use RES_SCHEDULER” to FALSE.

Step 6 In **OS Configuration → Application modes** Create a mode called OSDEFAULTAPPMODE.

These settings create a basic, empty, application for VRTA. 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.rtaos. If you click on “Check Now” then it should report “No errors”. If it doesn’t then you’ll need to check that you’ve done the preceding steps correctly.

Hooks

We’ll use AUTOSAR OS R3.0’s StartupHook() and ShutdownHook() to say ‘Hello World!’ and ‘Goodbye World!’ so these need to be enabled in the RTA-OS3.0 configuration.

Go back to **OS Configuration → General** and the “Hooks” tab and make the following changes:

Item	Setting
Call Startup Hook	TRUE
Call Shutdown Hook	TRUE

3.2.2 Building RTA-OS3.0

Select the **Builder → Setup** workspace and the “Samples” tab and make the following changes.

Item	Setting
Sample Header Files	Check all the boxes.

You can now build the RTA-OS3.0 kernel. To build from **rtaoscfg** go to **Builder → Build** and click “Run code-generator”. Building will create the following RTA-OS3.0 files in the same directory as your RTA-OS3.0 project file:

Filename	Contents
Os.h	The main include file for the OS.
Os_Cfg.h	Declarations of the objects you have configured. This is included by Os.h.
Os_MemMap.h	AUTOSAR memory mapping configuration used by RTA-OS3.0 to merge with the system-wide MemMap.h file.
RTA0S.<lib>	The RTA-OS3.0 library for your application. The extension <lib> depends on your target.
RTA0S.<lib>.sig	A signature file for the library for your application. The extension <lib> depends on your target.

You will also see a set of files whose name are prefixed vrta that have been created to support VRTA.

3.2.3 Writing Application Code

You now need to implement your RTA-OS3.0 application. Code Example 3.1 shows the code you need.

```

#include <Os.h>
OS_MAIN(){
    StartOS(OSDEFAULTAPPMODE);
}

FUNC(boolean,OS_APPL_CODE) Os_Cbk_Idle(void){
    ShutdownOS(E_OK);
    return TRUE; /* Never reached */
}

FUNC(void,OS_APPL_CODE) StartupHook(void){
    vrtaEnterUninterruptibleSection();
    printf("Hello World!\n");
    vrtaLeaveUninterruptibleSection();
}

FUNC(void,OS_APPL_CODE) ShutdownHook(StatusType s){
    vrtaEnterUninterruptibleSection();
    printf("Goodbye World!\n");
    vrtaLeaveUninterruptibleSection();
    vrtaTerminate();
}

```

Code Example 3.1: The complete application

Save this as a file called `Application.c` in the same directory as `Tutorial1.rtaos`

3.2.4 Writing VECU Code

Any AUTOSAR OS R3.0-based VECU that you build will need to initialize your virtual devices. The VM doesn't know how to initialize your devices, but it knows when it needs to do this, so it makes a callback to your code. The callback is named `InitializeDevices()` and it is a C++ function.



`InitializeDevices()` must always be present in any VECU application.

Code Example 3.2 shows a 'dummy' implementation.

```
#include "vrtaCore.h"
void InitializeDevices(void){
    // Placeholder
}
```

Code Example 3.2: Virtual Device Initialization

Save this as a file called `VirtualDevices.cpp` in the same directory as `Tutorial1.rtaos`

3.2.5 Building the Virtual ECU

All that remains is build the Virtual ECU:

- Compile the `Application.c` file, the `VirtualDevices.cpp` file. Make sure that the working directory (`.`) and `Samples\Includes` on the include path.
- Link two object files with `RTAOS.lib` and your compiler's Windows and User32 libraries to create an executable called `RTAOS.exe`.

The following batch file shows how this can be done with the MinGW compiler:

```
rem Define locations
set CBASE=C:\Progra~1\MinGW.513
set CC=%CBASE%\bin\gcc.exe
set CPP=%CBASE%\bin\g++.exe
set LNK=%CBASE%\bin\g++.exe

rem Set options
set COPTS=-c -g -I. -ISamples\Includes -I%CBASE%\include
set CPPOPTS=-c -g -I. -ISamples/Includes
```

```

rem Compile application and virtual device code
%CC% %COPTS%    Application.c
%CPP% %CPPOPTS% VirtualDevices.cpp

rem Link the application
%LNK% -O2 *.o RTA0S.lib -lwinmm -lws2_32 -o"RTA0S.exe"

```

3.2.6 Building the Virtual ECU

If everything has compiled and linked correctly then you can run your first Virtual ECU! Open a Windows command shell (DOS box) and run the program RTA0S.exe. Figure 3.1 shows what you should see.

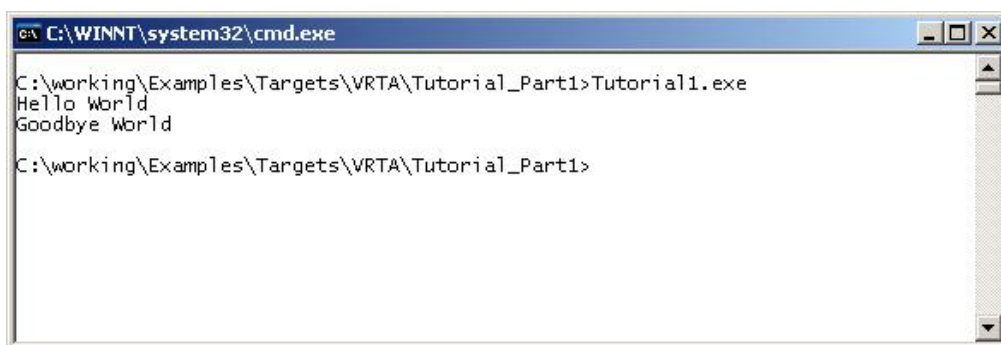


Figure 3.1: A successful run of you first Virtual ECU

If you are having problems, then you can find a working version of this part of the tutorial in the Examples directory of the target installation in the directory Tutorial_Part1.

3.3 Adding Devices

Before we look at making the Virtual ECU more interesting, it is useful to look at how virtual devices are provided.

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 functions 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 so that they can become known to the Device Manager before entering the main program (i.e. OS_MAIN). Do not attempt to create devices dynamically².

²You wouldn't expect a hardware counter to suddenly come into existence on a real ECU either!

All of the repetitious ‘plumbing’ code needed to create a device is provided in a C++ class called `vrtaDevice` that is defined in the file `vrtaDevice.h` which is generated when you build the RTA-OS3.0 kernel library. You can use this as a base class from which to create your own devices.

When the RTA-OS3.0 library is built a number of sample devices are automatically generated for you and placed in the library. You can access these devices through the header file `vrtaSampleDevices.h`. These sample devices are automatically included into your VECU if you use them and are documented in detail in Chapter 11.

The following sections shown you how you might uses these devices in application code.

3.3.1 Clocks, Counters and Compare Devices

The `vrtaClock` device represents a clock source in a real ECU. It can be thought of as the oscillator. It ticks independently of the rest of the application at up to 1000 times per second³. Code Example 3.3 shows you how to declare a clock named “Clock” that ticks every 5ms like this:

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

Code Example 3.3: Declaring a clock

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. We need to connect a counter to this clock source. Two types of counter are provided:

1. up-counting
2. down-counting

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 `InitializeDevices()` function as shown in Code Example 3.4. `InitializeDevices()` is also a good place to enable the counter.

³The clock device uses Windows multimedia timers, which have a minimum interval of 1ms.

```

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();
}

```

Code Example 3.4: Declaring counters

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. The `vrtaCompare` virtual device is provided to do exactly this job

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. Code Example 3.5 shows 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);

```

Code Example 3.5: Comparing counter values and raising interrupts

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.

 *Note 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

Sample 'sensor' devices are provided which 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. A basic `vrtaSensor` device has 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` which can have a position value zero or one.
- `vrtaSensorMultiwaySwitch` which has multiple positions, with an upper limit specified in its declaration.

Code Example 3.6 shows how these three types of sensor are declared.

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

Code Example 3.6: Declaring sensors

The values of a sensor can be changed remotely using **vrtaMonitor**. You can read the value of the sensor directly from your application using the `GetValue()` method. Alternatively you can attach a `vrtaCompare` device to the sensor and raise an event when a certain value is set as shown in Code Example 3.7 which raises interrupt 3 when the `EjectSwitch` position is set to 1.

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

Code Example 3.7: Generating an interrupt from a sensor value

3.3.3 Actuators

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

As with sensors, you can attach a `vrtaCompare` device to an actuator so that you can raise an interrupt when a set value is reached⁴.

Three derivatives of `vrtaActuator` are:

- `vrtaActuatorLight` with values zero and one.
- `vrtaActuatorDimmableLight` with 'levels' zero to n.

⁴By 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.

- vrtaActuatorMultiColorLight ('colors' zero to n).

Code Example 3.8 shows how these three types of sensor are declared.

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

Code Example 3.8: Declaring actuators

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 which are generated automatically by **rtaosgen** 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 is shown in Chapter 4.

3.4 Creating your first Virtual ECU: Part2

Reopen the Tutorial1.rtaos project then use **File → Save As** to make a copy in a new directory as C:\Play\Tutorial2\Tutorial2.rtaos. Copy Application.c and VirtualDevices.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 unit on each zero to one transition under interrupt control.
- The Left and Right switches are polled from an AUTOSAR 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 `VirtualDevices.cpp` so that it contains the code shows in Code Example 3.9.

```
#include "vrtaCore.h"
#include "vrtaSampleDevices.h"
#include "vrtaLoggerDevice.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
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));
}

```

Code Example 3.9: Extending VirtualDevices.cpp

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. The Logger device is provided by the header file `vrtaloggerDevice.h`.

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 place the text. You can make calls such as `Status.printf("Boo")` so that you can indicate to the outside world what is happening.

But your RTA-OS3.0 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 C++ Devices

There are two answers to this.

Using `vrtaSendAction`

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 be similar too that shown in Code Example 3.10⁵.

```
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);
```

Code Example 3.10: Using the `vrtaSendAction` to print

Using C/C++ Interfacing

The alternative, and simpler, model is to write a C / C++ interface function. This is what is done in Code Example 3.9. 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 `VirtualDevices.h` which will

⁵This 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.

export the C definitions for use in your C program. `VirtualDevices.h` needs to contain the code shown in Code Example 3.11.

```
/* Interface between C and C++ */
#ifdef __cplusplus
extern "C" {
#endif

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

#ifdef __cplusplus
}
#endif
```

Code Example 3.11: Interfacing C++ Devices to your C Application

The file `VirtualDevices.h` must be **#included** into C or C++ files that define or reference `status_printf`. For this tutorial application this means you need to have a **#include** `"VirtualDevices.h"` in `Application.c` and `VirtualDevices.cpp`.

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 The main() program

We need to modify `Application.c` so that the program does not simply start and stop by modifying what happens in `Os_Cbk_Idle`. We can also use the new `status_printf` function to display our output. Code Example 3.12 shows the how Code Example 3.1 is has been modified

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()`.

```
#include <Os.h>
#include "VirtualDevices.h"

OS_MAIN(){
    StartOS(OSDEFAULTAPPMODE);
}

FUNC(boolean,OS_APPL_CODE) Os_Cbk_Idle(void){
    while(!vrtaIsAppFinished()){
        vrtaIsIdle(5);
    }
    ShutdownOS(E_OK);
    return TRUE; /* Never reached */
}

FUNC(void,OS_APPL_CODE) StartupHook(void){
    status_printf("Hello World!\n");
}

FUNC(void,OS_APPL_CODE) ShutdownHook(StatusType s){
    status_printf("Goodbye World!\n");
    vrtaTerminate();
}
```

Code Example 3.12: Modifying the idle mechanism and using `status_printf`

3.4.6 Trial Run 1

Let's check that was all entered correctly. Save your project and perform a build. 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 running `RTAOS.exe`

When you run your new VECU, you will get an empty console screen plus an embedded GUI. The GUI is shown Figure 3.2. It shows a few details about your VECU and gives you the ability to suspend <F6>, resume <F7>, reset <Shift+F8> and terminate <F8> it. All these options are available from the **Application** menu. You should also 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 as shown in Figure 3.3. Note that it too has "Hello World" in the status bar.

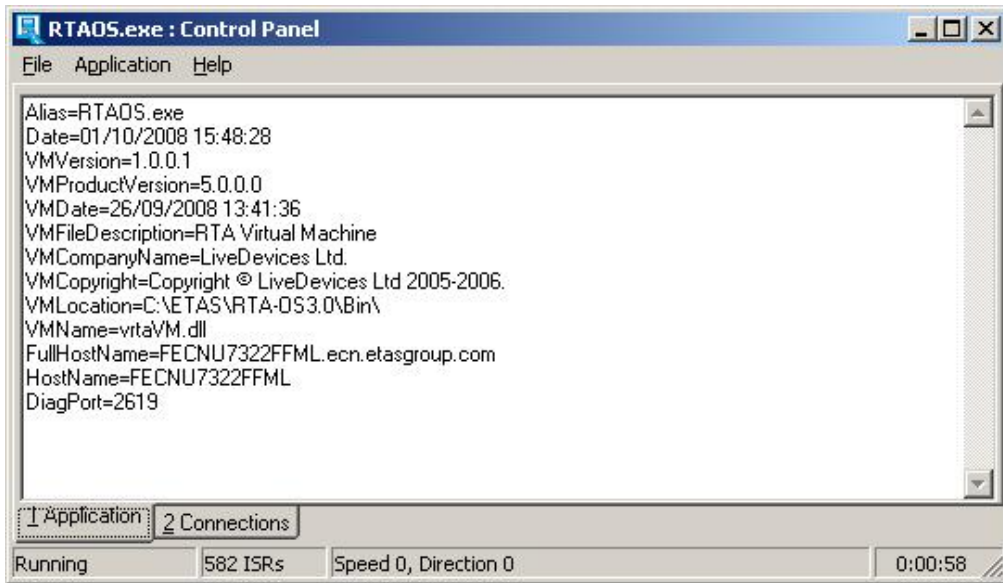


Figure 3.2: Trial Run 1 - Embedded GUI

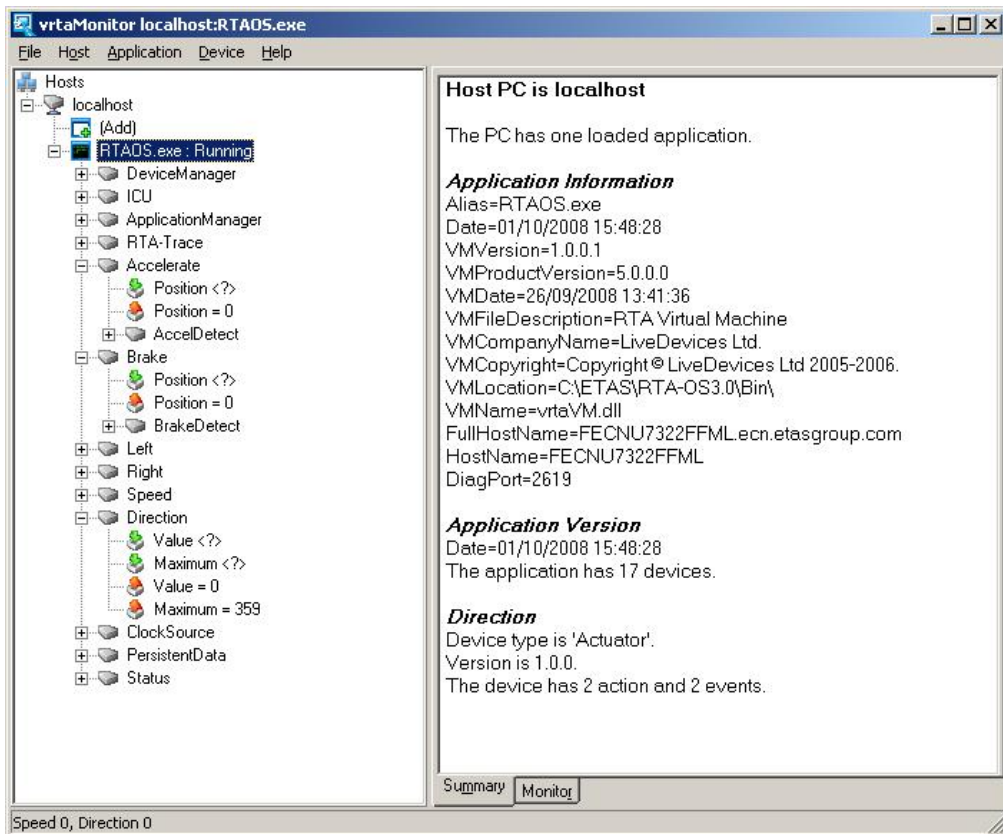


Figure 3.3: Trial Run 1 - vrtavMonitor

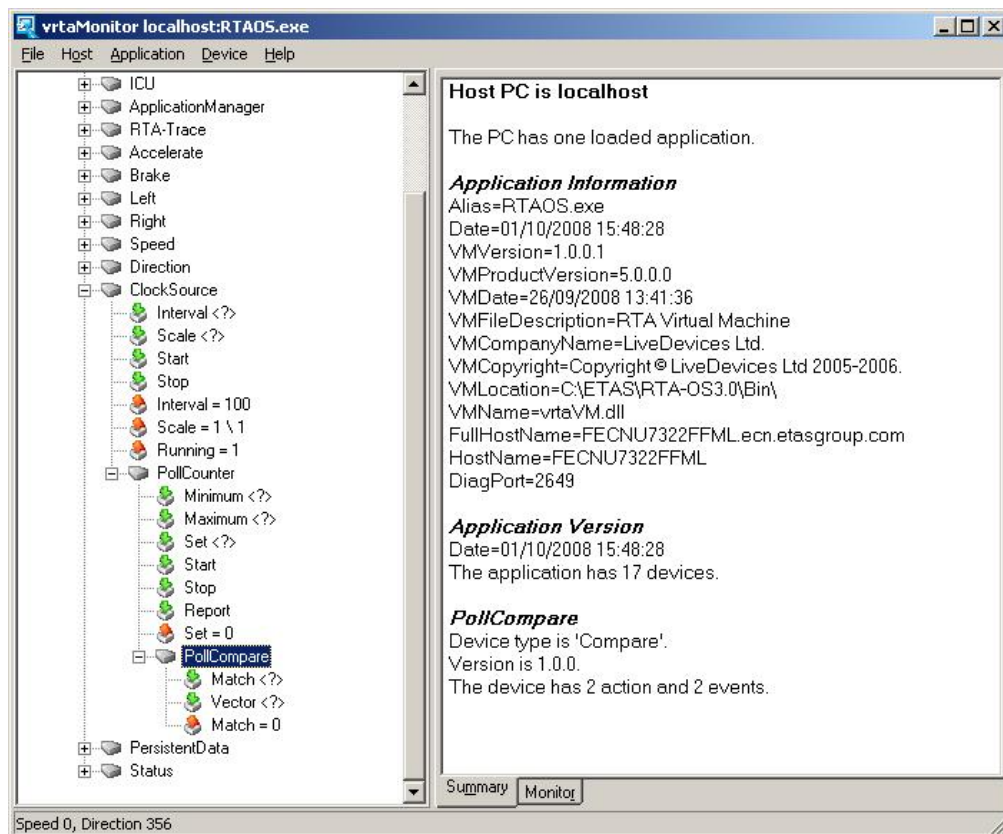


Figure 3.4: Display of hierarchical devices

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 then Section 5.2 describes **vrtaMonitor** in more detail.

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 as shown in Figure 3.4. 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 VECU⁶.

To stop monitoring an event, drag the event onto the “Stop” button. When you are ready, close the monitor and VECU.

3.4.8 Adding Tasks and ISRs

We now create the OS elements that implement the required functionality. Use `rtaoscfg` to add the following:

- A Task called `DirectionPoll` with Priority 1, Activations 1 and Preemptability FULL
- A Category 2 ISR named `isrAccelerate` 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.
- A Counter named `AlarmCounter` with:
 - “Minimum Cycle” of 1
 - “Maximum Value” of 65535
 - “Ticks Per Base” of 1
 - “Seconds Per Tick” of 0.1 (i.e. it is 100ms)
 - “Type” of SOFTWARE
- An Alarm named `Poller` attached to `AlarmCounter` with Action/Activate Task `DirectionPoll`, Auto start in `OSDEFAULTAPPMODE` with:
 - “Type” of RELATIVE
 - “Alarm Time” of 1
 - “Cycle Time” of 1

We will now need to implement the tasks and ISRs. Edit `Application.c` to add the three Category 2 ISRs and the task as shown in Code Example 3.13.

```
ISR(isrPoll){
    IncrementCounter(AlarmCounter);
}

ISR(isrBrake){
    change_speed(-1);
}
```

⁶Event times are recorded using the Windows API `GetTickCount()`. This typically has a resolution of around 15ms.

```

ISR(isrAccelerate){
    change_speed(+1);
}

TASK(DirectionPoll){
    if (left_pressed()) {
        change_direction(-1);
    }
    if (right_pressed()) {
        change_direction(+1);
    }
    TerminateTask();
}

```

Code Example 3.13: Adding OS Objects

We need to get access to the VECUs devices. Add the lines shown in Code Example 3.14 to `VirtualDevices.h`. These provide the C/C++ interface function prototypes that allow your RTA-OS3.0 application to get access to the devices you'll create in the VECU.:

```

#ifdef __cplusplus
extern "C" {
#endif

// The existing logger device
extern int status_printf(const char* format, ...);

// The new interfacing functions
extern void show_status(void);
extern int left_pressed(void);
extern int right_pressed(void);
extern void change_direction(int amount);
extern void change_speed(int amount);

#ifdef __cplusplus
}
#endif

```

Code Example 3.14: Exporting C function for the `VirtualDevices.cpp`

We now add the code shown in Code Example 3.15 to `VirtualDevices.cpp` provide the C/C++ functions themselves. They are clearly quite simple wrappers to the devices.

```

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

int right_pressed(void){
    return Right.Value();
}

void show_status(void){
    Status.printf("Speed %d, Direction %d",
                  Speed.Value(),
                  Direction.Value());
}

void 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();
}

void change_speed(int amount){
    int newvalue = Speed.Value() + amount;
    if ((newvalue >=0) && (newvalue <=100)) {
        Speed.SetValue(newvalue);
        show_status();
    }
}

```

Code Example 3.15: Accessing the C++ devices

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 shown in Code Example 3.16 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;

// Unmask the accelerator interrupt
action.devEmbeddedData.uVal = ACCEL_ISR;
vrtaSendAction(ICU_DEVICE_ID,&action);

// Unmask the brake interrupt
action.devEmbeddedData.uVal = BRAKE_ISR;
vrtaSendAction(ICU_DEVICE_ID,&action);

// Unmask the poll interrupt
action.devEmbeddedData.uVal = POLL_ISR;
vrtaSendAction(ICU_DEVICE_ID,&action);

```

Code Example 3.16: Unmasking the interrupt sources

Nearly there!

3.4.9 Threads

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, but we will do this in a separate thread.

You can create any number of threads of execution that run *independently* of your main application thread (the RTA-OS3.0 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-OS3.0 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.

The basic shape of an RTA-OS3.0 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.

To see a thread at work we need to add the thread and hook code is shown in Code Example 3.17 to `VirtualDevices.cpp` before `InitializeDevices()`:

```

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);
    }

    return RTVECUErr_NONE;
}

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);
}

```

Code Example 3.17: Implementing a Thread and Hooking events

We now need to make the thread run, so at the of InitializeDevices() add the line:

```
vrtaSpawnThread(AsyncThread);
```

This tells the VM to spawn the thread called AsyncThread.

3.4.10 Trial run 2

Save your project and perform a build. 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 using <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

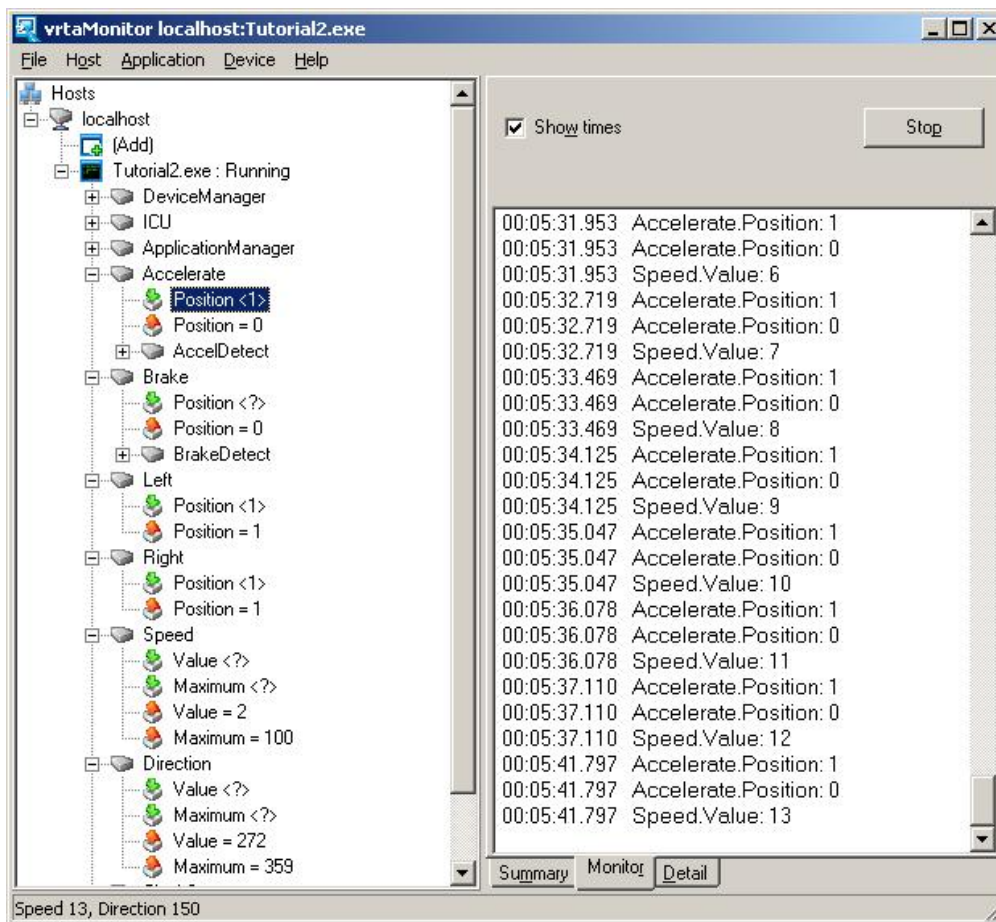


Figure 3.5: Trial Run 2 - Seeing acceleration with vrtaMonitor

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”’s action a few times and you will see the speed go up further as shown in Figure 3.5.

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 Value will count down and wrap at 0 degrees until you change it back to zero.

The same applies to the “Right” device, though obviously that will make the Value count up and wrap at 359 degrees.

3.4.11 Linking to Real Hardware

Just to prove that you can link to real hardware, you might like to change the implementation of the AsyncThread as shown in Code Example 3.18. It just

peeks at the state of the arrow keys on your keyboard and makes calls to your switches.

```
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);

    // NOW! Peek at keys to control Accelerator, Brake and
    // Direction
    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);
}

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

```

Code Example 3.18: Interfacing with 'real' hardware

3.4.12 Non-volatile Data

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

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. 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` in `VirtualDevices.cpp`:

```
vrtaIO PersistentData("PersistentData",DATA_SIZE);
```

with the lines shown in Code Example 3.19.

```

#include <stdio.h>
class vrtaFlash : public vrtaIO {

protected:
    void Starting(void) {

```

```

vrtaIO::Starting();
FILE *f = fopen("VirtualECU.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("VirtualECU.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);

```

Code Example 3.19: Adding persistence to vrtaIO to simulate NVRAM

You will now find that if you rebuild your VECU that it will remember the Value of the direction it was last pointing in when it restarts. The data is stored in the file called `VirtualECU.flash` which is read when the device starts and written when the device stops. 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.

3.4.13 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 **vrtaMonitor**, open the **RTA-TRACE Configuration → Configuration** in **rtaoscfg** and do the following configuration:

Item	Setting
Enable Tracing	TRUE
Use Compact IDs	TRUE
Use Compact Time	TRUE
Enable Stack Recording	FALSE
Run-Time Target Triggering	FALSE
Auto-Initialize Comms	TRUE
Set Trace Auto-repeat	FALSE
Buffer Size	2000
Autostart Type	FREE_RUNNING

Now try to rebuild.

You should get warnings about a missing `Os_Cbk_GetStopwatch` function. RTA-TRACE needs to know about the time at which things happen, so we must give it the help it needs. Code Example 3.20 shows the code you need to add to `Application.c`.

```
void Os_GetStopwatch(void){
    /* vrtaReadHPTime(x) returns the current time based on a
       required number
       of 'xticks', where there are 'x' xticks per second. We
       use the macro
       OSSWTICKSPERSECOND that defines the number of stopwatch
       ticks we expect
       per second.
    */
    return (Os_StopwatchTickType)vrtaReadHPTime(
        OSSWTICKSPERSECOND);
}
```

Code Example 3.20: Implementing `Os_Cbk_GetStopwatch`

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 **rtaoscfg** menu **General → Target → Clock Speeds** and change both value to something more sensible like 100MHz.

Finally we must add some code to `Os_Cbk_Idle` to send trace data from the application to RTA-TRACE. Edit `Os_Cbk_Idle` in `Application.c` so that looks

like Code Example 3.21.

```
boolean Os_Cbk_Idle(void){
    show_status();
    while(!vrtaIsAppFinished()) {
        vrtaIsIdle(5);
#ifdef OS_TRACE
        Os_CheckTraceOutput();
#endif
    }
    ShutdownOS(E_OK);
    return TRUE; /* Never reach here */
}
```

Code Example 3.21: Implementing Os_Cbk_GetStopwatch

Now rebuild the VECU.

What you now need to do is to install **vrtaServer** as a Windows service. You can do this easily by running **vrtaServer -install** from a command prompt.

Now start RTA-TRACE, connect to localhost and use the RTA0SEK-VRTA connection. Open the RTA0S.rta file. 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 shown in Figure 3.6.

If you are having problems, then you can find a working version of this part of the tutorial in the Examples directory of the target installation in the directory Tutorial_Part2.

3.5 Summary

If you have worked through this tutorial then you should have a good understanding of how to work with a Virtual ECU. In particular you should now have an idea of how you can:

- create virtual devices and initialize them.
- access virtual devices from your C (RTA-OS3.0) application.
- print debugging data to the embedded GUI and to the status bar of **vrtaMonitor**.
- interact with your virtual devices using
 - **vrtaMonitor**
 - an asynchronous thread to simulate some behavior

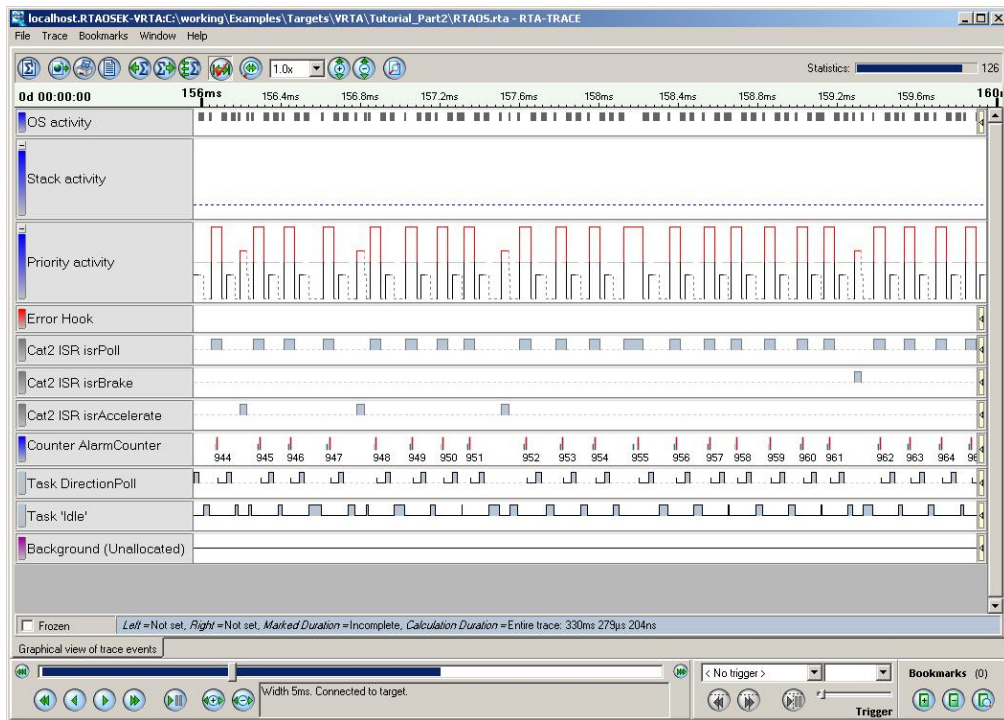


Figure 3.6: Tracing the tutorial VECU

- real hardware (like your keyboard!)
- how to integrate RTA-TRACE with your virtual ECU.

The remainder of this guide provides a technical reference to building and interacting with a Virtual ECU.

4 ECUs and Virtual Devices

At its most basic level, a virtual device is simply an object 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 in the files generated by RTA-OS3.0 for VRTA, 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. This is done automatically 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>=<value2>`. 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 for-

mat 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 Section 4.2 for 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 Section 4.3 for how to handle status queries.

4.2 Handling actions

The action callback function that you register is 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. Code Example 4.1 shows the basic form of the `OnAction` method.

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

Code Example 4.1: Handling a device 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 in Code Example 4.2.

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

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

Code Example 4.2: Responding to a state request

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. Code Example 4.3 show you how to raise an event.

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

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

Code Example 4.3: Raising an event

4.5 Raising Interrupts

Interrupts are raised by sending action `ICU_ACTION_ID_Raise` to the ICU device using code like that shown in Code Example 4.4.

```
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);
}

```

Code Example 4.4: Raising an interrupt

The `vrtaDevice` class provides a `RaiseInterrupt()` method that can be used by classes that inherit from it. Code Example 4.5 shows how to raise an interrupt if `val` has reached a specified value.

```

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);
    }
}

```

Code Example 4.5: Using `RaiseInterrupt()`

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`¹ 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. You have seen this used already in the screenshot in Figure where `PollCompare`’s `_Parent` is set to `PollCounter` and `PollCounter`’s `_Parent` is set to `ClockSource`.

¹By convention this is the highest numbered event in the device.

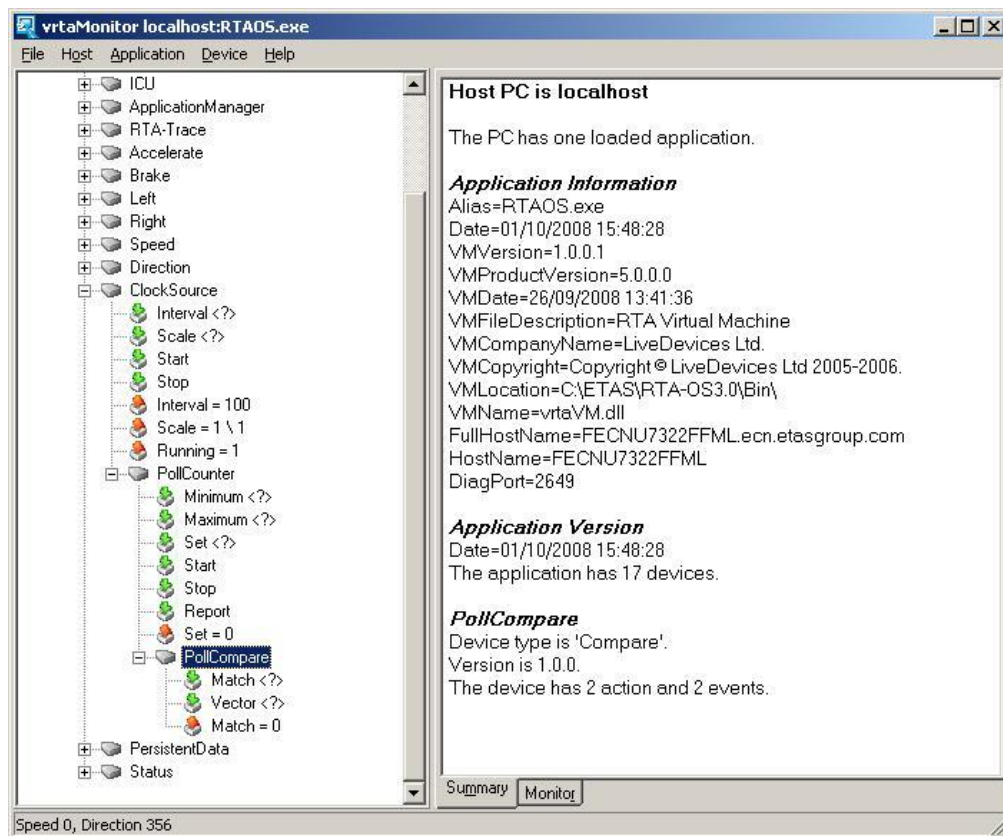


Figure 4.1: Viewing devices using **vrtaMonitor**

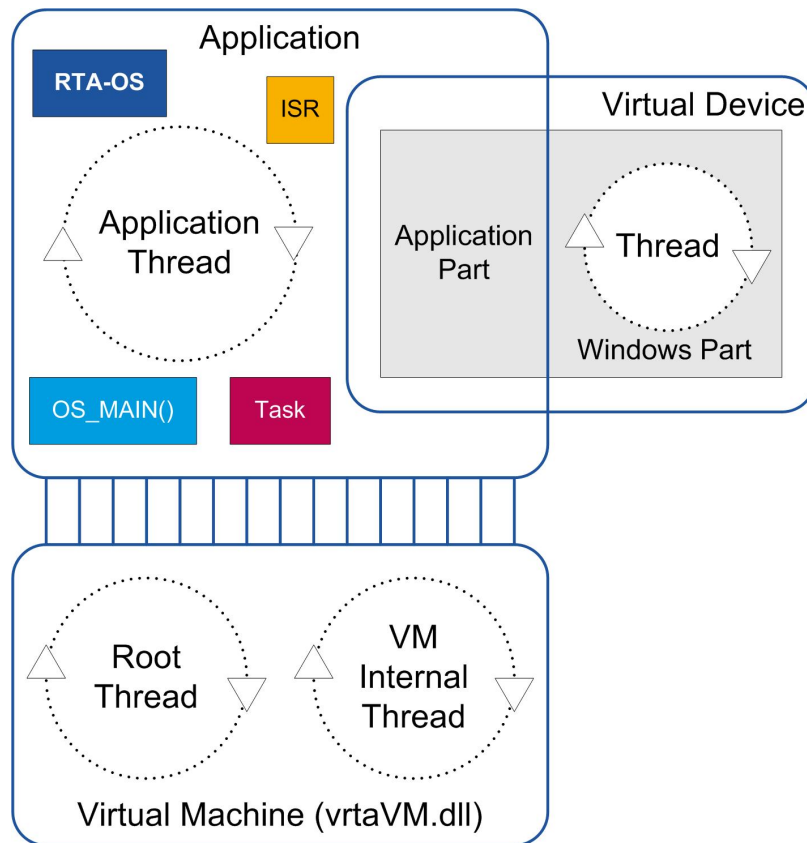


Figure 4.2: Threads in a Virtual ECU application

4.7 Threads

A virtual device can spawn an RTA-OS3.0 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.

Figure 4.2 illustrates how threads are used within a VECU. The application thread is the Windows thread that runs the application code, including ISRs and RTA-OS3.0 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 Interacting with VECUs

vrtaServer is a small program that can be installed as a Windows service and will run unobtrusively on your PC coordinating the loading and locating of your Virtual ECUs.

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. This means 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.

Figure 5.1 shows a **vrtaMonitor** attached to three 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 shown in Figure 5.2, **vrtaMonitor** is accessing a remote PC, so the directory structure that you see reflects that on the remote PC.

5.1 Running **vrtaServer**

To install **vrtaServer** as a service you need to run **vrtaServer -install** from the command line. If you decide that you want to uninstall the service then this is done using **vrtaServer -uninstall**. **vrtaServer** 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.



*If you are using Windows Vista then you must run **vrtaServer -install** from a Windows Command Prompt that has administrator rights. To obtain a Windows Command Prompt with administrator rights, right-click on the Command Prompt icon and select "Run as administrator". When the dialogue shown in Figure 5.3 appears click **Continue** and then run **vrtaServer -install** at the prompt.*

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 service¹. Then run 'vrtaServer -standalone'. You will see a tabbed dialog-style window appear along with a new icon in your system tray as shown in Figure 5.4.

¹'vrtaServer -start' would start it again.

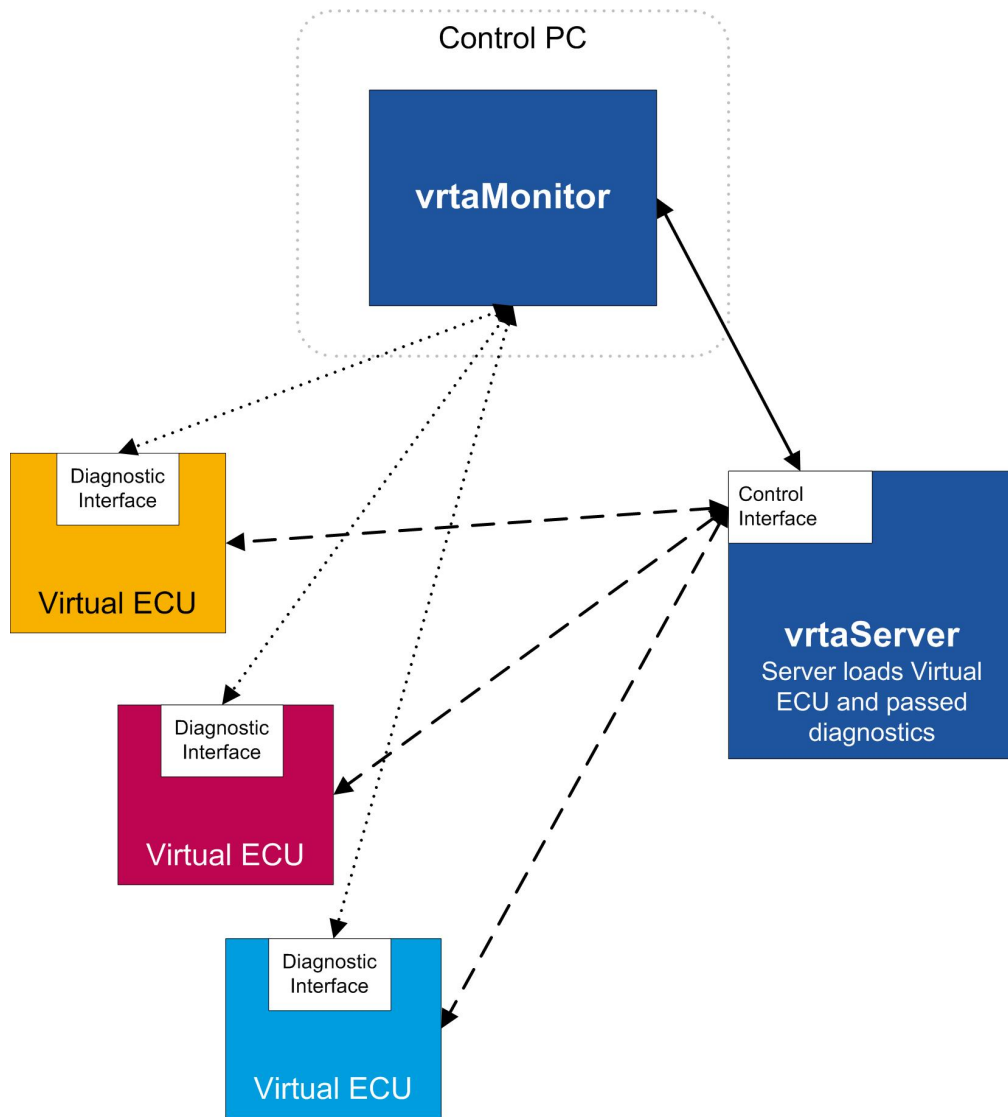


Figure 5.1: Interaction between **vrtaServer**, VECUs and **vrtaMonitor**

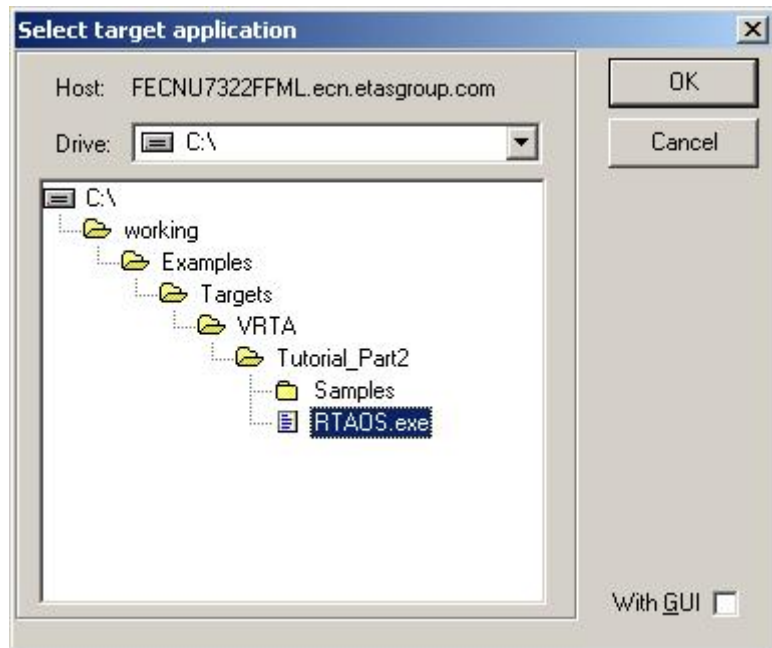


Figure 5.2: Accessing a remote PC with **vrtaMonitor**

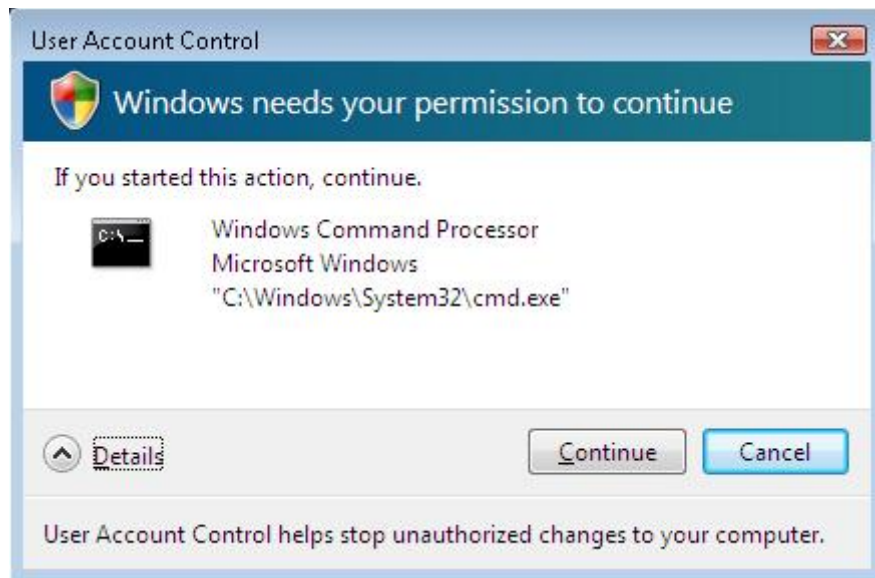


Figure 5.3: Running the Windows Command Prompt in Windows Vista

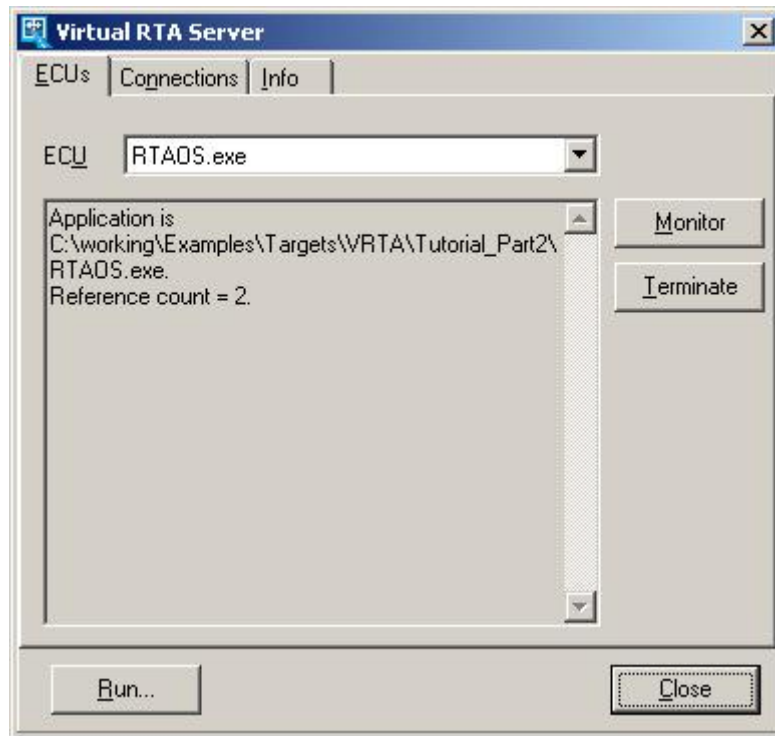


Figure 5.4: Viewing ECUs with **vrtaServer**

The dialog will also give you information about the VECUs and monitors that are connected to **vrtaServer** by selecting the “Connections” tab as shown in Figure 5.5.

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 use **vrtaServer -stop** from the command line).

5.1.1 Security issues

vrtaServer allows remote monitor programs to launch VECUs on its host PC. This means that there are security issues of which you need to be aware.

vrtaServer does *not* allow a monitor program to copy any programs or data to the host PC or allow a remote user to modify files via the load dialog.

In a controlled test environment, on a secure network, **vrtaServer** should not pose any serious security threat. However if you are worried about malicious abuse of this feature, you should configure your firewall to block external access to **vrtaServer** and ensure that **vrtaServer** and any VECUs run on the same host PC. The default TCP ports used by **vrtaServer** (and RTA-TRACE) are 26000, 31765 and 17185.

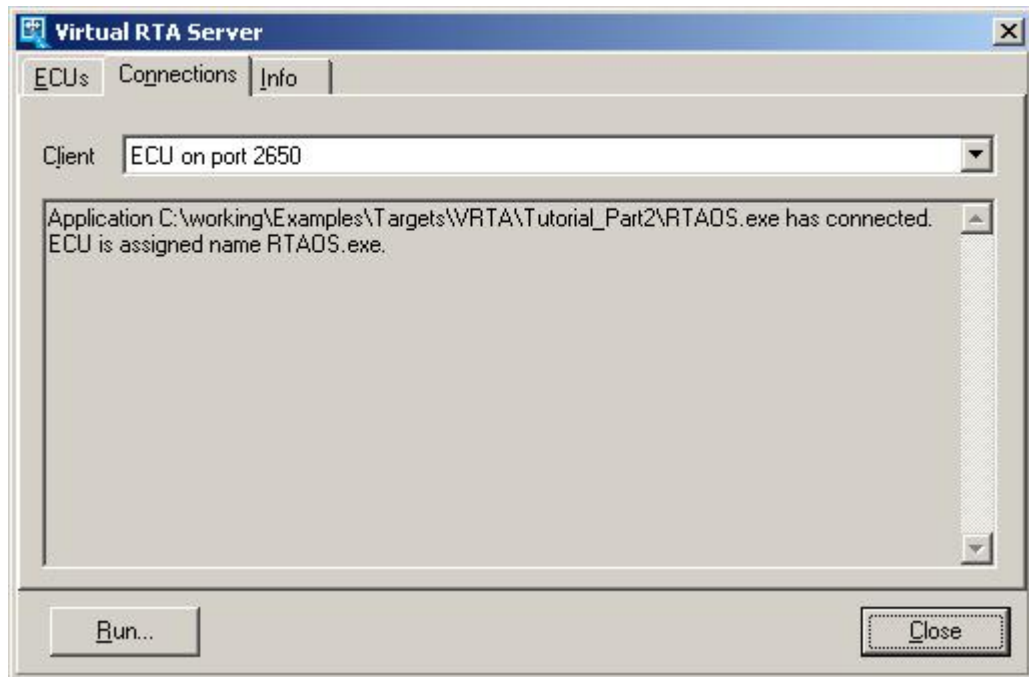


Figure 5.5: Viewing connections with **vrtaServer**

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 a connection should be made, though a port-scan would find the connection easily.

5.2 Using **vrtaMonitor**

The program **vrtaMonitor** can be used to inspect and control virtual ECUs on local and remote PCs. Figure 5.6 shows **vrtaMonitor** connected to a VECU called `example2.exe` running on the `localhost` (i.e. the same PC that is running the **vrtaMonitor**).

vrtaMonitor can connect to multiple PCs (Hosts), and multiple VECUs within each host. You can interact with **vrtaMonitor** in a number of 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.

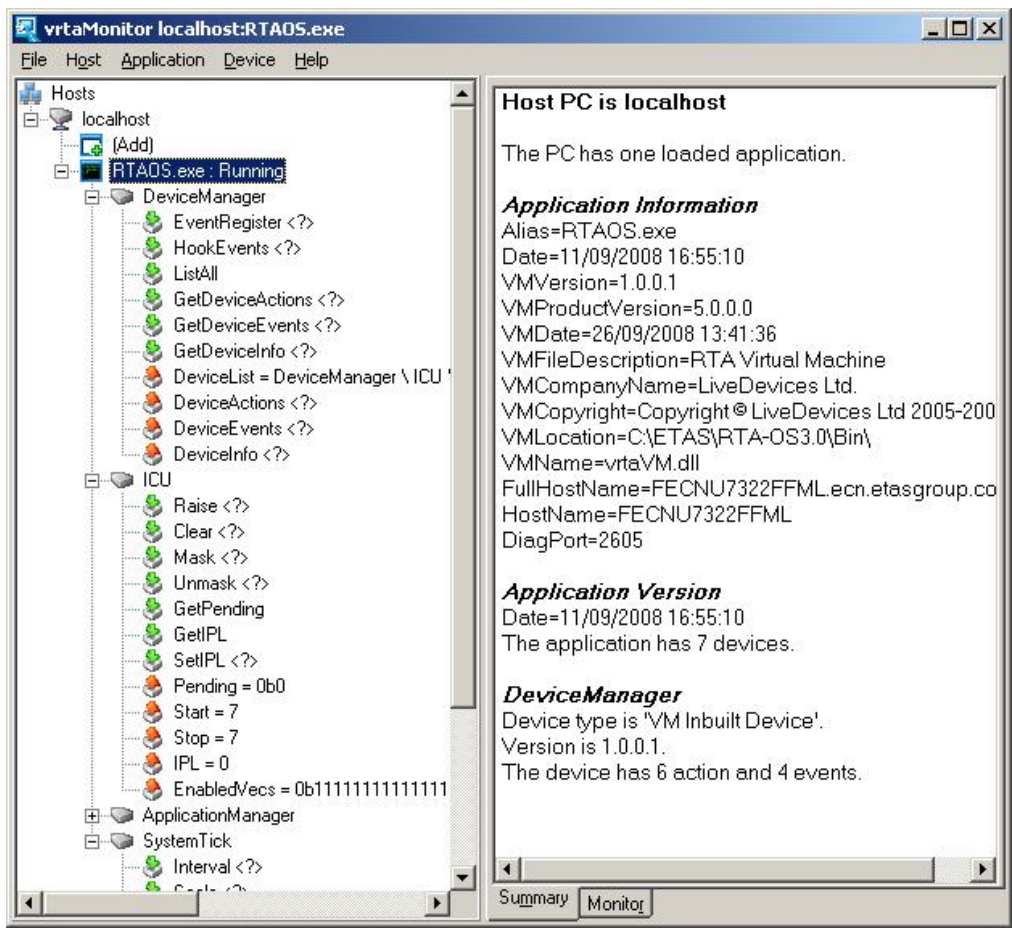


Figure 5.6: Using **vrtMonitor**

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.

5.2.1 Actions

Actions are shown in **vrtaMonitor** using the following icon:



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.

Figure 5.7 shows the actions dialogue.

5.2.2 Events

Events are shown in **vrtaMonitor** using the following icon:



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

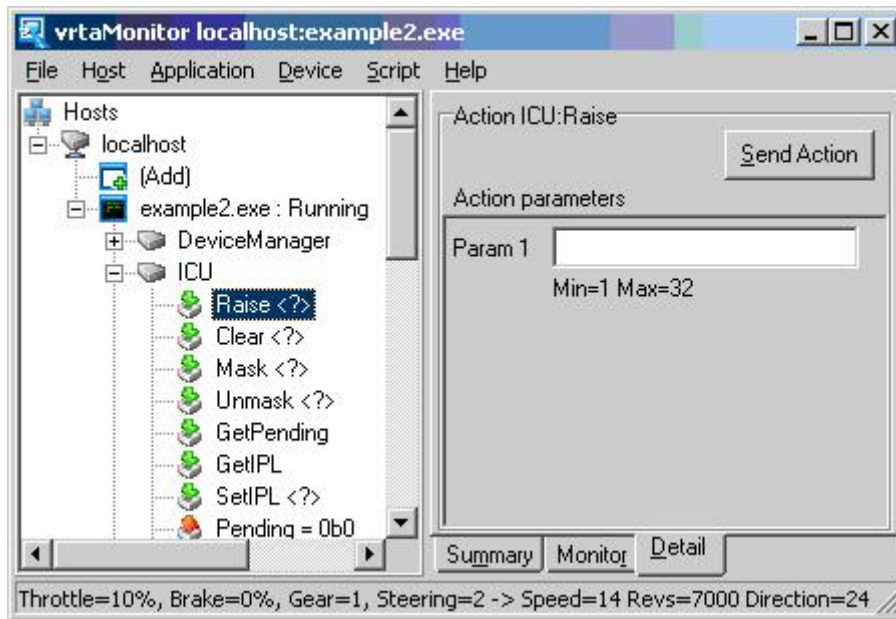


Figure 5.7: vrtaMonitor actions dialogue

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.

Figure 5.8 shows the events dialogue.

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. A second <Ctrl+A> will turn auto-refresh off again.

Monitor

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

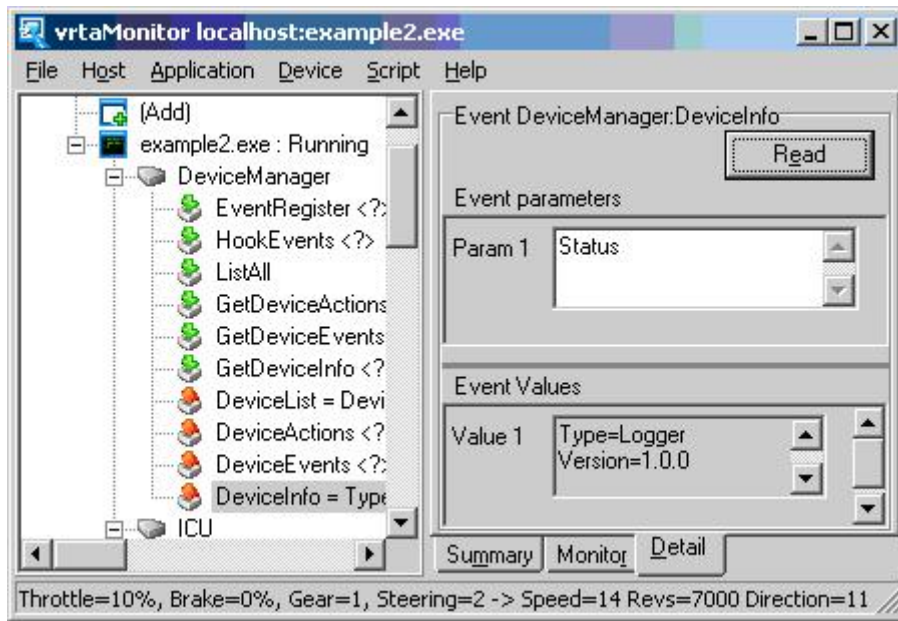


Figure 5.8: vrtaMonitor events dialogue

In this case, the VECU notifies **vrtaMonitor** whenever a monitored event is 'raised'. Events are typically raised when some value in the virtual device changes. All notifications are displayed in the monitor tab:

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.

5.3 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.

5.4 Scripting using **vrtaMonitor**

The **vrtaMonitor** command-line options can be used to support a limited form of scripting capability for VECUs.

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 12, but a few useful examples are presented here.

5.4.1 Example Scripts

Load and run an application for a specified time

```
-k  
-log=log.txt  
-t1  
VirtualECU.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 `VirtualECU.exe` parameter tells **vrtaMonitor** to load and run the VECU `VirtualECU.exe` (without showing its embedded GUI).

Load the application and start the 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
VirtualECU.exe
```

Attach **vrtaMonitor** to a loaded VECU

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

Send an action to a device

These commands below will attach to an existing VECU and then terminate it:

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

Monitor Events

The example below runs `VirtualECU.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
VirtualECU.exe
```

```
-mon=PollCompare.Match  
-wait=5000  
-send=ApplicationManager.Terminate  
-quit
```


6 RTA-TRACE Integration

RTA-TRACE, available as a separate product provides a very detailed graphical display showing in real-time the execution of all Tasks, ISRs and processes in your RTA-OS3.0 application.

VRTA is supplied with a special high-bandwidth virtual device that can be used to connect Virtual ECU to RTA-TRACE. If you have installed RTA-TRACE in the same location as RTA-OS3.0 then this link will be detected automatically. If not you must copy the file `rtcVRTAlink.dll` from RTA-OS3.0's Bin directory to RTA-TRACE's Bin directory.

Using RTA-TRACE with an RTA-OS3.0 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-OS3.0 GUI, enable the trace communication link and call `Os_CheckTraceOutput()` regularly, for example from the `Os_Cbk_Idle` callback. Code Example 6.1 shows the basic model.

```
boolean Os_Cbk_Idle(void) {
    while(!vrtaIsAppFinished()) {
        #ifdef OSTRACE_ENABLED
            Os_CheckTraceOutput();
        #endif
        vrtaIsIdle(5);
    }
}
```

Code Example 6.1: Using RTA-TRACE

6.1 How it works

6.1.1 The Virtual ECU

When you build a trace-enabled Virtual ECU, code is added to your application to implement the RTA-TRACE communication APIs `Os_Cbk_TraceCommInitTarget()` and `Os_Cbk_TraceCommDataReady()` plus a virtual device named "RTA-TRACE".

Whenever a block of trace data is ready to be sent, `Os_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-OS3.0 for PC application can generate accurate traces without being affected by 'communication-interval gaps' that affect other ports.

6.1.2 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, RTA-TRACE will start the Virtual ECU, attach itself and hook the Trace event of its RTA-TRACE device to collect trace data.



*This functionality only works if the RTA-TRACE configuration file and the Virtual ECU have identical names (with `.rta` and `.exe` extensions respectively) and are located in the same directory. If the RTA-TRACE file is called `ProjectName.rta` then the Virtual ECU must be called `ProjectName.exe`. The **vrtaServer** must be also be installed as a service.*

6.2 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 running¹.

You may find it useful to adjust the process or thread priority for the VECU for best results. This can be done via the RTA-TRACE-Server configuration dialog that is accessible via RTA-TRACE-Server as shown in Figure 6.1.

GUI shows whether a VECU has its embedded GUI visible when started by `rtcVRTAlink.dll`. This is set automatically when the VECU is started.

ECUThreadPriority affects the priority of the application (AUTOSAR OS) thread within the VECU.

ProcessPriority affects the priority of the complete VECU.

¹Note 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.

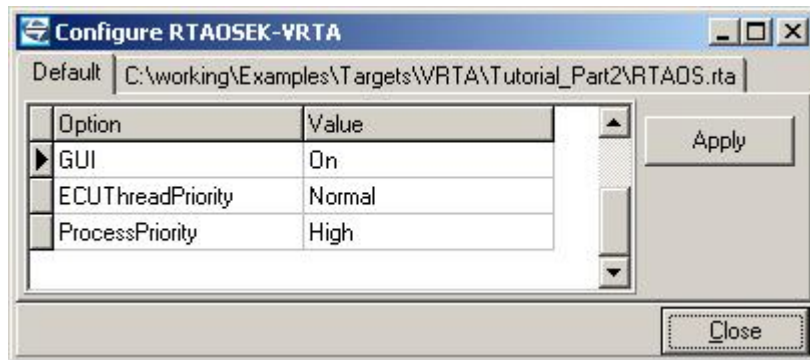


Figure 6.1: RTA-TRACE Server priority Control

6.3 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 `Os_CheckTraceOutput()` regularly, this action will cause the appropriate target API (`Os_StopTrace()`, `Os_StartFreeRunningTrace()`, `Os_StartBurstingTrace()` or `Os_StartTriggeringTrace()`) to get called.

Repeat. This is sent On or Off to set the `Os_SetTraceRepeat()` value in the VECU. Again, you must call `Os_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 6.2.

ProcessPriority. This action can be used to change the VECU's process priority in the same way as described in Section 6.2.

7 Windows Notes

Although RTA-OS3.0 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 may be useful.

7.1 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 Windows process priority of the Virtual ECU - see the `-priority=<n>` command line options.

7.2 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 descriptions of `vrtaEnterUninterruptibleSection()` and `vrtaLeaveUninterruptibleSection()` calls in Chapter 9 for further details.

7.3 Virtual Machine Location

When a Virtual ECU is started it tries to load the VM DLL (vrtaVM.dll). The VECU first tries to load the VM DLL using the normal Microsoft DLL search rules¹. That is, it searches the following locations in the specified order:

1. The directory containing the VECU.
2. The 32-bit Windows system directory (C:\<windowsdir>\system32).
3. The 16-bit Windows system directory (C:\<windowsdir>\system).
4. The Windows directory (C:\<windowsdir>).
5. The current directory.
6. The directories listed in the PATH environment variable.

If the VECU fails to find the vrtaVM.dll then the application will fail to start. You can fix this by making sure that vrtaVM.dll can be found on the DLL search path.



It is highly recommended that you add all the installation directory for RTA-OS3.0 executables (by default C:\ETAS\RTA-OS3.0\Bin to your Windows PATH environment variable to ensure that vrtaVM.dll can always be found.

¹[http://msdn.microsoft.com/en-us/library/ms682586\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/ms682586(VS.85).aspx)

8 Migrating from a VECU to Real Hardware

Assume that you have developed a Virtual ECU application using RTA-OS3.0 and now need to migrate the application to your real hardware. We will assume that you have obtained a version of RTA-OS3.0 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.

8.1 XML file

Your VECU application is described by a combination of the C source code, build scripts and the RTA-OS3.0 project file. The project file contains references to XML configuration files that describe your RTA-OS3.0 configuration using AUTOSAR XML and is called something like `myproject.rtaos`.

If you are migrating to a non-RTA-OS3.0 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-OS3.0 based implementation is relatively easy because you can switch between different targets within the RTA-OS3.0 GUI. Changing targets removes any target-specific configuration from the XML file(s), for example interrupt vectors and priorities, so you will need to configure these attributes for the embedded target.

8.1.1 Target and variant

The target and variant will need to be changed. You can do this by opening your XML configuration in `rtaoscfg` and selecting another target¹.

8.1.2 Interrupts

RTA-OS3.0 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. Therefore you will need to assign new target-based vector numbers and priorities for each ISR in your application².

¹You must have another RTA-OS3.0 target installed to be able to do this.

²Unless you were far-sighted enough to configure your VECU to use the same interrupt model as in your target ECU.

8.1.3 Number of tasks

RTA-OS3.0 for VRTA ECUs can use up to 1024 tasks. Your embedded RTA-OS3.0 port may support this many tasks, but you may find that fewer tasks are supported. If the RTA-OS3.0 port can handle fewer tasks than your VECU currently uses, you will have to do some re-engineering of your application.

8.2 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 shown in Code Example 8.1 (see Section 3.4.3 for a fuller explanation).

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

Code Example 8.1: C/C++ Interface Routine

This means that the framework for interacting with your devices is already there. You simply provide different implementations for each of these interface functions so that they map onto your target hardware as shown in Code Example 8.2

```
int left_pressed(void)
{
    return input(0x1001);
}
```

Code Example 8.2: Re-implementing the routine in C

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.

8.3 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-OS3.0 for PC applications. You must refer to the user-guide for your hardware for the best way to do this.

8.4 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 could use C's macro language to implement conditional compilation of ISR code so that it can adapt to different platforms as shown in Code Example ??.

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

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

    #ifdef TARGET_HC12X
        output(0x12,99);
    #endif
}
```

Code Example 8.3: Conditional Compilation

8.5 Register Sets

RTA-OS3.0 can be configured to save and restore processor register sets for tasks and ISRs that can preempt other tasks or ISRs which use the same register set.

If you have used this feature in your Virtual ECU (for example, to save and restore floating point context) then you will have written code to perform the save and restore of x86 registers. You will need to replace this code with an appropriate implementation for your chosen embedded target. Please read the *RTA-OS3.0 User Guide* for further details.

9 Virtual Machine API Reference

This chapter gives a detailed description of the 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 `#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.

9.2 Common Data Types

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

9.2.1 `vrtaActEvID`

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

9.2.2 `vrtaAction`

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

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

9.2.3 `vrtaActionID`

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

9.2.4 `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.

9.2.5 `vrtaByte`

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

9.2.6 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.

9.2.7 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`.)

9.2.8 vrtaEmbed

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

Field Type	Field Name
int	<code>iVal</code>
unsigned	<code>uVal</code>
double	<code>dVal</code>
<code>vrtaByte</code>	<code>bVal[16]</code>

`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.

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.

9.2.9 vrtaErrType

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

Value	Description
<code>RTVECUErr_NONE</code>	No error / success
<code>RTVECUErr_Dev</code>	Device fault. Typically invalid device ID.
<code>RTVECUErr_ID</code>	ID fault. Typically invalid action or event ID.
<code>RTVECUErr_VAL</code>	Value fault. Typically value is out of range.
<code>RTVECUErr_Conn</code>	Connection fault. Occurs with remote monitor applications if the link to the VECU fails.

9.2.10 vrtaEvent

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

Declaration (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 *the next time that vrtaGetState() is called from the same thread*. 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 anyway¹.

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.

9.2.11 `vrtaEventID`

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

9.2.12 `vrtaIntPriority`

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

9.2.13 `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'.

9.2.14 `vrtaMillisecond`

A scalar representing an *interval* in milliseconds.

9.2.15 `vrtaOptStringlistPtr`

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

Example

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

9.2.16 `vrtaStringlistPtr`

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

Example

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

9.2.17 `vrtaTextPtr`

A pointer to a simple ASCIIZ string

Example

¹For 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.

```
vrtaTextPtr tp = 'Hello World';
```

9.2.18 vrtaTimestamp

A scalar representing the *current ECU time* in milliseconds. This is normally the number of milliseconds since (just before) `OS_MAIN()` was called².

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 devices³.

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.

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 comma separated. The data-item descriptors are as follows (text inside [] is optional):

²If 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.

³By 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.

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 be represented in hexadecimal. 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[:con>]	The data-item is an unsigned number that should be 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> ...[:cons]	The data-item is an unsigned number that should be represented as a series of enumeration values. Enumeration value <a> corresponds to the number 0, enumeration value 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:

Constraint Item	Description
bits	The number of bits used to store a numeric value. This can be 8, 16, 32 or 64.
width	The number of bytes between the start of the data-item and the start of the next data-item.
min	The minimum and maximum values that may be stored in a numeric data-item.
max	The minimum and maximum values that may be stored in a numeric data-item.

9.3.3 Examples

Example format strings are:

String	Description
%d	A single signed number that will be stored in 32 bits (a C int).
%d::1;32	An integer that can take values 1 through 32.
%u::1;10	An unsigned number in the range 1 to 10 inclusive that will be stored in 32 bits (a C unsigned).
%b	A 32-bit integer, normally represented in binary by a monitor program.
%x:16,%b:8	An unsigned number that will be stored in 16 bits (a C unsigned short) and should be displayed in hexadecimal. 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.
A B C D E	A 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.
%s:10,%u:8!4,%u:64;1;100	A composite format string: <ul style="list-style-type: none"> • a 10 character string, followed by • an unsigned number stored in 8 bits, padded to 4 bytes (i.e. 3 bytes of padding as the next data-item is stored 4 bytes after the start of the 8 bit value), followed by • an unsigned number in the range 1 to 100 inclusive stored in 64 bits.

9.4 API Functions

Each API call is described in this section 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

Name	Mode	Description
Parameter Name	Input/Output	Description

Description

Explanation of the API call functionality.

Return values

Value	Description
Return value	Description

Notes

Usage restrictions and notes for the API call.

See also

List of related API calls.

Fields that are not relevant for the API call are omitted.

9.4.1 InitializeDevices()

Device initialization hook function.

Function declaration

```
void InitializeDevices(void)
```

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.

Notes

This function has C++ linkage and so must be implemented in a C++ compilation unit.

See also

[OS_MAIN](#)
[vrtaStart](#)

9.4.2 OS_MAIN()

The entry-point for the application thread.

Function declaration

```
OS_MAIN()
```

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().

If the Virtual ECU has been loaded in auto start 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.

Notes

OS_MAIN() normally has the structure shown below in an RTA-OS3.0 application. You should also implement an Os_Cbk_idle that polls vrtaIsAppFinished() so that external requests to terminate the program get recognized promptly. If you don't shutdown the application cleanly when a termination request is made then the VM will forcibly terminate the application thread after a few seconds.

Example

```
#include <Os.h>
OS_MAIN(){
  /* Initialize target hardware */
  StartOS(OSDEFAULTAPPMODE);
}

FUNC(boolean, OS_APPL_CODE) Os_Cbk_Idle(void) {

  while(!vrtaIsAppFinished()) {
    /* Yield for 5 milliseconds. Note that interrupts will
       still be recognized if they occur */
    they occur
    vrtaIsIdle(5);
  }
}
```

```
ShutdownOS(E_OK);  
}
```

See also

[InitializeDevices](#)
[vrtaStart](#)
[vrtalsAppFinished](#)
[vrtalsIdle](#)

9.4.3 vrtaEnterUninterruptibleSection()

Enter a critical section that cannot be interrupted.

Function declaration

```
void vrtaEnterUninterruptibleSection(void)
```

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.



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-OS3.0 for VRTA interrupt executes.

See also

[vrtaLeaveUninterruptibleSection](#)

9.4.4 vrtaEventRegister()

Register an event handler.

Function declaration

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

Parameters

Name	Mode	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 raised⁴. 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.

⁴See the description of vrtaGetState() for more information about the vrtaEvent type.

See also

[vrtaEventUnregister](#)

[vrtaHookEvent](#)

[vrtaRaiseEvent](#)

9.4.5 vrtaEventUnregister()

Unregister an event handler.

Function declaration

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

Parameters

Name	Mode	Description
listener	Input	An event-handler handle returned by vrtaEventRegister().

Description

This API call un-registers 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

Name	Mode	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 7.2.15 for details.

See also

- [vrtaEventRegister](#)
- [vrtaEventUnregister](#)
- [vrtaHookEvent](#)
- [vrtaRaiseEvent](#)
- [vrtaRegisterVirtualDevice](#)

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

Name	Mode	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	Whether to hook the event or not. 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

Name	Mode	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-OS3.0 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.

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

Example

```
#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 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. */
}
```

Notes

If a Virtual ECU is using RTA-OS3.0 then this API should not be called explicitly as it is called automatically by the main() program.

See also

[vrtaLoadVM](#)
[vrtaStart](#)

9.4.9 `vrtalsAppFinished()`

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

Function declaration

```
vrtBoolean vrtIsAppFinished(void)
```

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 `vrtTerminate()` or a Terminate action has been sent to the ApplicationManager Device. This API may also be used in *RTA-OS3.0* for *VRTA* 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.

See also

[vrtSpawnThread](#)

9.4.10 vrtaIsAppThread()

Determine if the calling thread is the application thread.

Function declaration

```
vrtaBoolean vrtaIsAppThread(void)
```

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.

9.4.11 vrtalsIdle()

Yield the processor whilst idle.

Function declaration

```
void vrtalsIdle  
    (vrtamillisecond msec)
```

Parameters

Name	Mode	Description
msec	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 `vrtalsIdle()`. For example, if at time t the application thread calls `vrtalsIdle(100)` and at time $t+10$ an interrupt arrives, the corresponding ISR will be run by the application thread at, or shortly after, $t+10$.

9.4.12 `vrtaLeaveUninterruptibleSection()`

Leave a critical section.

Function declaration

```
void vrtaLeaveUninterruptibleSection(void)
```

Description

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

See also

[vrtaEnterUninterruptibleSection](#)

9.4.13 vrtaLoadVM()

Function declaration

void vrtaLoadVM(**void**)

Description

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

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().



If a Virtual ECU is using RTA-OS3.0 this API does not need to be called explicitly as it is called by the main program.

See also

[vrtaInitialize](#)

[vrtaStart](#)

9.4.14 vrtaRaiseEvent()

Raise an event.

Function declaration

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

Parameters

Name	Mode	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

[vrtaEventRegister](#)
[vrtaEventUnregister](#)
[vrtaGetState](#)
[vrtaHookEvent](#)
[vrtaRegisterVirtualDevice](#)

9.4.15 vrtaReadHPTime()

Read the PC's high-performance timer.

Function declaration

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

Parameters

Name	Mode	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.

9.4.16 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

Name	Mode	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.

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.

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'
- <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`).

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:

Events	Format
'<name>'	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.
'<name>=<format>'	Describes an event that contains data as described by <format>.
'<name>=<format>(<format>)'	Describes an event that contains data as described by <format> before the '()' and input data when queried by <code>vrtaGetState()</code> as described by <format> inside the '()'.

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

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

Action	Format
'<name>'	Describes an action that does not have any associated data.
'<name>=<format>'	Describes an action that contains data as described by <format>.

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

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 Section 9.2.2 for a description of the contents of the vrtaAction structure).

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 be 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.

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 9.2.10 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

[vrtaEventRegister](#)
[vrtaEventUnregister](#)
[vrtaGetState](#)
[vrtaHookEvent](#)
[vrtaSendAction](#)

9.4.17 vrtaReset()

Reset the Virtual Machine.

Function declaration

void vrtaReset(**void**)

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.

See also

[vrtaStart](#)

[vrtaTerminate](#)

9.4.18 vrtaSendAction()

Send an action to a virtual device.

Function declaration

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

Parameters

Name	Mode	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 event 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](#)

9.4.19 vrtaSpawnThread()

Create a new thread.

Function declaration

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

Parameters

Name	Mode	Description
func	Input	The entry function for the new thread.

Description

This API call creates a new 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.

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](#)

9.4.20 vrtaStart()

Start the application thread.

Function declaration

```
void vrtaStart(void)
```

Description

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

If the VECU has been loaded in auto start 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).

Notes

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



If Virtual ECU is using RTA-OS3.0 then this API should not be called explicitly as it is called by the main program.

See also

[InitializeDevices](#)
[OS_MAIN](#)
[vrtaLoadVM](#)
[vrtaInitialize](#)

9.4.21 vrtaTerminate()

Terminate the Virtual Machine.

Function declaration

void vrtaTerminate(**void**)

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.

See also

[vrtaStart](#)

[vrtaReset](#)

10 Standard Devices (vrtaStdDevices.h)

Most of the functionality of the Virtual Machine is accessed through the VM's three standard devices:

1. Device Manager
2. Interrupt Control Unit (ICU)
3. Application Manager

These standard devices behave in 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.

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`.

10.1 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 braces (). 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.


10.2 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

10.2.1 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.

10.2.2 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 <event_i> is an event ID in decimal. Such an i.e. 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.

10.2.3 Action: ListAll

Action ID Constant	Data Format
DM_ACTION_ID_ListAll	<none>

This action causes the DM to raise a DeviceList event.

10.2.4 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.

10.2.5 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.

10.2.6 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.

10.2.7 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.

10.2.8 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.

10.2.9 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.

10.2.10 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.

10.3 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

10.3.1 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).

10.3.2 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.

10.3.3 Action: Mask

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

This action masks (disables) the specified interrupt number.

10.3.4 Action: Unmask

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

This action unmask (enables) the specified interrupt number.

10.3.5 Action: GetPending

Action ID Constant	Data Format
ICU_ACTION_ID_GetPending	<none>

This action causes a Pending event to be raised.

10.3.6 Action: GetIPL

Action ID Constant	Data Format
ICU_ACTION_ID_GetIPL	<none>

This action causes an IPL event to be raised.

10.3.7 Action: SetIPL

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

This action sets the IPL to the specified value.

10.3.8 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.

10.3.9 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.

10.3.10 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.

10.3.11 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.

10.3.12 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.

10.4 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

10.4.1 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 *<i>slave</i>* mode.

10.4.2 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.

10.4.3 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.

10.4.4 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.

10.4.5 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.

10.4.6 Action: GetInfo

Action ID Constant	Data Format
AM_ACTION_ID_GetInfo	<none>

This action causes an Info event to be raised.

10.4.7 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.

10.4.8 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.

10.4.9 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.

10.4.10 Event: Started

Event ID Constant	Data Format
AM_EVENT_ID_Started	<none>

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

10.4.11 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.

10.4.12 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.

10.4.13 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.

10.4.14 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.

10.4.15 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.

10.4.16 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.

10.4.17 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.

10.4.18 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.

10.4.19 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 Devices (vrtaSampleDevices.h)

The Virtual ECU 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 kernel library and exported through `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`).

The following sample devices are provided:

Device	Description
<code>vrtaClock</code>	A clock source for counter devices.
<code>vrtaUpCounter</code>	A counter that counts upwards.
<code>vrtaDownCounter</code>	A counter that counts downwards.
<code>vrtaSensor</code>	A generic sensor.
<code>vrtaSensorToggleSwitch</code>	A two position 'toggle' switch.
<code>vrtaSensorMultiwaySwitch</code>	A multi-position switch.
<code>vrtaActuator</code>	A generic actuator.
<code>vrtaActuatorLight</code>	A light that can be on or off.
<code>vrtaActuatorDimmableLight</code>	A light that can be set to different levels of brightness.
<code>vrtaActuatorMultiColorLight</code>	A light that can be set to different colors.
<code>vrtaCompare</code>	A comparator that can generate an interrupt when other another device reaches a specified value.
<code>vrtaIO</code>	An I/O space.

11.1 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.

11.1.1 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 of the parameter.

Description

Explanation of the functionality of the method.

Return values

Value	Description
Value	Description of the return value.

11.1.2 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).

11.2 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.



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.)

11.2.1 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

None.

11.2.2 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

None.

11.2.3 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

None.

11.2.4 Method: Start()

Start the clock ticking.

Method declaration

```
void Start(void)
```

Parameters

None.

Description

This method starts the clock device ticking any attached counters.

Return values

None.

11.2.5 Method: Stop()

Stop the clock ticking.

Method declaration

void Stop(**void**)

Parameters

None.

Description

This method stops the clock device ticking any attached counters.

Return values

None.

11.2.6 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.

11.2.7 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.

11.2.8 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.

11.2.9 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.

11.2.10 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.

11.2.11 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.

11.2.12 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).

11.3 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.

11.3.1 Method: vrtaUpCounter()

The constructor.

Method declaration

```
void 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

None.

11.3.2 Method: Min()

Get the minimum value.

Method declaration

```
unsigned Min(void)
```

Parameters

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.

11.3.3 Method: Max()

Get the maximum value.

Method declaration

unsigned Max(**void**)

Parameters

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.

11.3.4 Method: Value()

Get the current value.

Method declaration

unsigned Value(**void**)

Parameters

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.

11.3.5 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

None.

11.3.6 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

None.

11.3.7 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

None.

11.3.8 Method: Start()

Start the counter counting.

Method declaration

```
void Start(void)
```

Parameters

None.

Description

This method is used to start the counter counting when ticked by the attached vrtaClock device.

Return values

None.

11.3.9 Method: Stop()

Stop the counter counting.

Method declaration

```
void Stop(void)
```

Parameters

None.

Description

This method is used to stop the counter counting when ticked by the attached vrtaClock device.

Return values

None.

11.3.10 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.

11.3.11 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.

11.3.12 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.

11.3.13 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.

11.3.14 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.

11.3.15 Action: Report

Action ID	Data Format
6	<none>

This action causes a Set event to be raised.

11.3.16 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.

11.4 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.

11.4.1 Method: vrtaDownCounter()

The constructor.

Method declaration

```
void 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

None.

11.4.2 Method: Min()

Get the minimum value.

Method declaration

```
unsigned Min(void)
```

Parameters

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.

11.4.3 Method: Max()

Get the maximum value.

Method declaration

unsigned Max(**void**)

Parameters

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.

11.4.4 Method: Value()

Get the current value.

Method declaration

unsigned Value(**void**)

Parameters

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.

11.4.5 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

None.

11.4.6 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

None.

11.4.7 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

None.

11.4.8 Method: Start()

Start the counter counting.

Method declaration

```
void Start(void)
```

Parameters

None.

Description

This method is used to start the counter counting when ticked by the attached vrtaClock device.

Return values

None.

11.4.9 Method: Stop()

Stop the counter counting.

Method declaration

```
void Stop(void)
```

Parameters

None.

Description

This method is used to stop the counter counting when ticked by the attached vrtaClock device.

Return values

None.

11.4.10 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.

11.4.11 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.

11.4.12 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.

11.4.13 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.

11.4.14 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.

11.4.15 Action: Report

Action ID	Data Format
6	<none>

This action causes a Set event to be raised.

11.4.16 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.

11.5 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.

11.5.1 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

None.

11.5.2 Method: GetMax()

Get the maximum value.

Method declaration

unsigned GetMax(**void**)

Parameters

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.

11.5.3 Method: Value()

Get the current value.

Method declaration

unsigned Value(**void**)

Parameters

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.

11.5.4 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

None.

11.5.5 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

None.

11.5.6 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.

11.5.7 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.

11.5.8 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.

11.5.9 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.

11.6 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.

11.6.1 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

None.

11.6.2 Method: Value()

Get the current value.

Method declaration

unsigned Value(**void**)

Parameters

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.

11.6.3 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

None.

11.6.4 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.

11.6.5 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.

11.7 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.

11.7.1 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

None.

11.7.2 Method: GetMax()

Get the maximum value.

Method declaration

```
unsigned GetMax(void)
```

Parameters

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

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 (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

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

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 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.

11.8.1 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

None.

11.8.2 Method: GetMax()

Get the maximum value.

Method declaration

```
unsigned GetMax(void)
```

Parameters

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.

11.8.3 Method: Value()

Get the current value.

Method declaration

unsigned Value(**void**)

Parameters

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.

11.8.4 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

None.

11.8.5 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

None.

11.8.6 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.

11.8.7 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.

11.8.8 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.

11.8.9 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.

11.9 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.

11.9.1 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

None.

11.9.2 Method: Value()

Get the current value.

Method declaration

```
unsigned Value(void)
```

Parameters

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.

11.9.3 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

None.

11.9.4 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.

11.9.5 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.

11.10 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.

11.10.1 Method: `vrtaActuatorDimmableLight()`

The constructor.

Method declaration

```
<p class=CodeNormal>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

None.

11.10.2 Method: `GetMax()`

Get the maximum value.

Method declaration

```
unsigned GetMax(void)
```

Parameters

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

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 (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

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

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 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.

11.11.1 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

None.

11.11.2 Method: GetMax()

Get the maximum value.

Method declaration

```
unsigned GetMax(void)
```

Parameters

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.

11.11.3 Method: Value()

Get the current value.

Method declaration

unsigned Value(**void**)

Parameters

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.

11.11.4 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

None.

11.11.5 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

None.

11.11.6 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.

11.11.7 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.

11.11.8 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.

11.11.9 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.

11.12 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.

11.12.1 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

None.

11.12.2 Method: `GetMatch()`

Get the match value.

Method declaration

```
unsigned GetMatch(void)
```

Parameters

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.

11.12.3 Method: `SetMatch()`

Set the match value.

Method declaration

```
void SetMatch(unsigned val)
```

Parameters

Parameter	Input/Output	Description
<code>val</code>	Input	The new match value.

Description

This method is used to set the match value.

Return values

None.

11.12.4 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.

11.12.5 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

None.

11.12.6 Action: Match

Action ID	Data Format
1	%u

This action sets the match value. It is the equivalent of the SetMatch() method.

11.12.7 Action: Vector

Action ID	Data Format
2	%u

This action sets the interrupt vector number. It is the equivalent of the SetVector() method.

11.12.8 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.

11.13 vrtaIO

A vrtaIO device represents an array of 32-bit I/O cells that may be written and read by an application.

11.13.1 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

None.

11.13.2 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

None.

11.13.3 Method: SetValues()

Set the value of multiple I/O cells.

Method declaration

```
void SetValue(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

None.

11.13.4 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.

11.13.5 Method: GetValues()

Get the values of all I/O cells.

Method declaration

```
const unsigned *GetValues(void) const
```

Parameters

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.

11.13.6 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.

11.13.7 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 SetValue() method with the offset argument set to zero.

11.13.8 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.

11.13.9 Action: GetValues

Action ID	Data Format
4	<none>

This action causes a Values event to be raised.

11.13.10Event: 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.

11.13.11 Event: Values

Event ID	Data Format
2	%a

This event contains the values of all of the I/O cells.

11.14 Rebuilding from Source Code

`vrtaSampleDevices.h` is provided automatically for you and the sample devices themselves are integrated into the RTA-OS3.0 library.

If you prefer, you can rebuild the sample devices from source code by generating the source code for the devices and compiling it yourself.

To do this, you should call `rtaosgen` with the `--samples:[Devices]` option to generate sample source code and follow any instructions you are given.

12 Command Line

This chapter provides a list of the command line options that are supported by Virtual ECU executables, **vrtaServer** and **vrtaMonitor**.

12.1 <VirtualECU>.exe

The command line options listed below can be used when a Virtual ECU executable (the executable you create when building an application using VRTA) 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.

Option	Description														
-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. RTA0S.exe). The option -alias=name causes the token name to be used as the Virtual ECU's alias (if possible).														
-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=int option can be used; where int is: <table border="1"><thead><tr><th>int</th><th>Windows Priority Class</th></tr></thead><tbody><tr><td>0</td><td>IDLE_PRIORITY_CLASS</td></tr><tr><td>1</td><td>BELOW_NORMAL_PRIORITY_CLASS</td></tr><tr><td>2</td><td>NORMAL_PRIORITY_CLASS</td></tr><tr><td>3</td><td>ABOVE_NORMAL_PRIORITY_CLASS</td></tr><tr><td>4</td><td>HIGH_PRIORITY_CLASS</td></tr><tr><td>5</td><td>REALTIME_PRIORITY_CLASS</td></tr></tbody></table> 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.	int	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
int	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														

Option	Description
-silent	<p>Select silent or GUI mode.</p> <p>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).</p> <p>If the -silent option is not used then the Virtual ECU is loaded in GUI mode. In <i>GUI</i> mode the Virtual ECU displays its own GUI.</p>
-slave	<p>Select slave or auto start mode.</p> <p>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.</p> <p>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.</p> <p>If the -slave option is not used then the Virtual ECU loads in auto start mode. In auto start mode the application thread starts immediately after the vrtaStart() Virtual Machine API has been called.</p> <p>In auto start mode the Virtual Machine terminates immediately after the application thread returns from OS_MAIN().</p>

12.2 vrtaServer

vrtaServer supports the command line options listed below.

Option	Description
-install	<p>Install as a service.</p> <p>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.</p>

Option	Description
-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.
-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.
-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.
-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.
-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.
-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.

12.3 vrtaMonitor

vrtaMonitor supports two sets of command line options:

Global options affect the overall operation of **vrtaMonitor**. It does not matter where the global options appear in the command line.

Sequential options are processed in the order they appear on the command line.

12.3.1 Global Options

The following options affect the overall operation of **vrtaMonitor**.

Option	Description
-f<filename>	Close when <filename> appears. The -f<filename> options tells vrtaMonitor to run until the file <filename> appears and then to terminate.
-k	Terminate with specific error level. The -k option causes vrtaMonitor to terminate with a specific error level when certain events occur.
-log=<file>	Write to a log file. The -log=<file> options causes vrtaMonitor to log activity to the file <file>.
-scripter=<name>	Select a scripting engine. The -scripter=<name> option selects the scripting engine called rtaScript<name>.dll.
-t<n>	Close after <n> seconds. The -t<n> options tells vrtaMonitor to run for <n> seconds and then terminate.

12.3.2 Sequential Options

The following options are processed in the order they appear on the command line.

Option	Description
<VirtualECU>.exe	Auto-load the named <VirtualECU>.exe. 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.

Option	Description
-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.
-d	Load but not start a Virtual ECU. The -d options causes the next Virtual ECU auto-loaded to be loaded but not started.
-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).
-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.
-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.
-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.
-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.
-quit	Terminate. The -quit option causes vrtaMonitor to terminate.

Option	Description
-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.
-script=<file>	Use a script file. The -script=<file> option causes vrtaMonitor to run the script in the file <file>. Please contact ETAS if you need information on monitor scripts.
-send=<dev>.<act>	Send an 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.
-send=<dev>.<act>(<str>)	Send an action with data. 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.
-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.
-wait=<n>	Wait. The -wait=<n> option causes vrtaMonitor to wait for approximately <n> milliseconds before processing the next command line option.

12.3.3 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.

13 Virtual ECU Server Library

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.

13.1 Using the DLL

The server library is 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.

13.2 Using the Source Code

If required, you can rebuild the library by generating the source code for the devices and compiling it yourself.

To do this, you should call **rtaosgen** with the `--samples:[Interfaces]` option to generate sample source code and follow any instructions you are given. 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.

The `VesLib.dll` supplied with VRTA was produced by compiling `VesLib.cpp` with the C++ macro `VESLIB_DLL` defined.

13.3 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.

13.4 Types

13.4.1 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.

13.4.2 VesLibEcuAliasType

The VesLibEcuAliasType type is used to contain Virtual ECU aliases. VesLibEcuAliasType has the definition:

```
typedef struct {  
    char name[VESLIB_MAX_PATH];  
} VesLibEcuAliasType;
```

13.5 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	Mode	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.

13.6 VesLibAttachToECU()

Attach to a loaded Virtual ECU.

Function declaration

```
VesLibStatusType VesLibAttachToECU  
                (VesLibEcuAliasType * alias,  
                 int * port)
```

Parameters

Parameter	Mode	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

None.

See also

[VesLibListLoadedECUs](#)
[VesLibLoadECU](#)

13.7 VesLibCreateAlias()

Create an alias for a Virtual ECU.

Function declaration

```
VesLibStatusType VesLibCreateAlias  
    (VesLibEcuInfoType * ecu,  
     VesLibEcuAliasType * alias)
```

Parameters

Parameter	Mode	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 VesLibEcuAliasType 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

None.

See also

[VesLibFreeAlias](#)
[VesLibLoadECU](#)

13.8 VesLibExit()

Shutdown the library.

Function declaration

```
VesLibStatusType VesLibExit(void)
```

Parameters

Parameter	Mode	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 the vrtaServer.

Return values

Value	Description
VESLIB_STATUS_OK	The API call succeeded.

Notes

None.

See also

[VesLibInitialize](#)

[VesLibSelectServer](#)

13.9 VesLibFindECUs()

Find out what Virtual ECU executables are present.

Function declaration

```
VesLibStatusType VesLibFindECUs  
    (char * dir,  
     char * * results)
```

Parameters

Parameter	Mode	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](#)

13.10 VesLibFreeAlias()

Free a Virtual ECU alias.

Function declaration

```
VesLibStatusType VesLibFreeAlias(  
    VesLibEcuAliasType * alias)
```

Parameters

Parameter	Mode	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

None.

See also

[VesLibCreateAlias](#)

[VesLibGetAliases](#)

[VesLibListAliases](#)

[VesLibListLoadedECUs](#)

13.11 VesLibFreeMemory()

Free memory allocated by the server library.

Function declaration

```
void VesLibFreeMemory(void * results)
```

Parameters

Parameter	Mode	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

None.

See also

[VesLibFindECUs](#)
[VesLibGetAliases](#)
[VesLibGetInfo](#)
[VesLibListAliases](#)
[VesLibListLoadedECUs](#)

13.12 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	Mode	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.
ESLIB_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](#)

[VesLibLoadEECU](#)

[VesLibListAliases](#)

13.13 VesLibGetInfo()

Get version information about a Virtual ECU.

Function declaration

```
VesLibStatusType VesLibGetInfo(  
    VesLibEcuAliasType * alias,  
    char * *           results)
```

Parameters

Parameter	Mode	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

None.

13.14 VesLibInitialize()

Initialize the library.

Function declaration

```
VesLibStatusType VesLibInitialize(void)
```

Parameters

Parameter	Mode	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	<code>vrtaServer</code> cannot be started.
VESLIB_STATUS_SERVER_COMMS	The library cannot communicate with <code>vrtaServer</code> .

Notes

None.

See also

[VesLibExit](#)

[VesLibSelectServer](#)

13.15 VesLibListAliases()

Get a list of all aliases that exist.

Function declaration

```
VesLibStatusType VesLibListAliases(  
    VesLibEcuAliasType * * results,  
    int * count)
```

Parameters

Parameter	Mode	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](#)
[VesLibGetAliases](#)
[VesLibLoadECU](#)

13.16 VesLibListLoadedECUs()

Get a list of the Virtual ECUs that have been loaded.

Function declaration

```
VesLibStatusType VesLibListLoadedECUs(  
    VesLibEcuAliasType * * results,  
    int * count)
```

Parameters

Parameter	Mode	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](#)

13.17 VesLibLoadECU()

Load a Virtual ECU.

Function declaration

```
VesLibStatusType VesLibLoadECU(  
    VesLibEcuAliasType * alias,  
    VesLibStartMode     startMode,  
    VesLibDisplayMode   displayMode,  
    char *              cmd,  
    int *               port)
```

Parameters

Parameter	Mode	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
<code>VESLIB_STATUS_OK</code>	The API call succeeded.
<code>VESLIB_STATUS_SERVER_COMMS</code>	The library cannot communicate with <code>vrtaServer</code> .
<code>VESLIB_STATUS_NO_ECU</code>	The alias does not exist.
<code>VESLIB_STATUS_ECU_LOADED</code>	The alias is already loaded.

Notes

None.

See also

[VesLibCreateAlias](#)

[VesLibListLoadedECUs](#)

[VesLibAttachToECU](#)

13.18 VesLibSelectServer()

Select the vrtaServer to use.

Function declaration

```
VesLibStatusType VesLibSelectServer(  
    const char *host,  
    int        port)
```

Parameters

Parameter	Mode	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

None.

See also

[VesLibInitialize](#)

[VesLibExit](#)

14 COM Bridge Tutorial

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 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`.

A reference to the COM bridge is provided in Chapter 15.



This chapter assumes familiarity with COM programming and is a tutorial on the use of the Virtual ECU COM bridge only.

14.1 Example

The section shows code snippets written in Microsoft Visual Basic that demonstrate how to use the COM Bridge.

14.1.1 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(ourexex, 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

```

14.2 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

```

14.2.1 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

```

14.3 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 'VirtualCar' VECU that

ships with RTA-OS3.0 for PC. You should be able to find this in the examples directory for the VRTA port plug-in. 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.

14.3.1 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'.

14.3.2 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:

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.

14.3.3 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.

14.3.4 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

```

14.3.5 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 vrta-Monitor to change the Brake and Throttle values, and your application will respond automatically.

14.3.6 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
```

14.3.7 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.

15 COM Bridge Reference

The Component Object Model (COM) Bridge provides services to COM-enabled clients such as Microsoft Visual Basic 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.

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).

This chapter describes the objects and interfaces provided by the COM Bridge. A tutorial on the use of the COM bridge is provided in Chapter 14.

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	Mode	Description
Parameter Name	Input/Output	Description.

Description

Explanation of the functionality of the method.

Return values

Value	Description
Return values	Description of return value.

15.1 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.

15.2 ICVcServer

This is the interface to **vrtaServer**. ICVcServer's constants and API are based on those in the Virtual ECU Server Library (VesLib).

15.2.1 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

15.2.2 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

15.2.3 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

15.2.4 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	Mode	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.

15.2.5 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	Mode	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.

15.2.6 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	Mode	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.

15.2.7 Method: Disconnect()

Disconnect from [vrtaServer](#).

Method declaration

```
HRESULT Disconnect( void )
```

Parameters

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

None.

15.2.8 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	Mode	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.

15.2.9 Method: FreeAlias()

Free a Virtual ECU alias.

Method declaration

```
HRESULT _stdcall FreeAlias(  
    [in] BSTR alias,  
    [out, retval] IVcServer_Status * status
```

Parameters

Parameter	Mode	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.

15.2.10 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	Mode	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.

15.2.11 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	Mode	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.

15.2.12 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	Mode	Description
aliases	Output	A <code>Œ</code> 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 `Œ` 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.

15.2.13 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	Mode	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.

15.2.14 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	Mode	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.

15.2.15 Method: ServerStatus()

Check if the server is still connected.

Method declaration

```
HRESULT ServerStatus(  
    [out, retval] IVcServer_Status *status
```

Parameters

Parameter	Mode	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.

15.3 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.

15.4 ICVcECU

This is the interface to a Virtual ECU.

15.4.1 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

15.4.2 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	Mode	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
<code>ECU_OK</code>	Success.
<code>ECU_ValErr</code>	If the connection cannot be made.

15.4.3 Method: Disconnect()

Disconnect from a Virtual ECU.

Method declaration

```
HRESULT _stdcall Disconnect(  
    [out, retval] IVcECU_Status * status
```

Parameters

Parameter	Mode	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.

15.4.4 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	Mode	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.

15.4.5 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	Mode	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.

15.4.6 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	Mode	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.

15.4.7 Method: GetDeviceCount()

Get the number of virtual devices.

Method declaration

```
HRESULT _stdcall GetDeviceCount(  
    [out, retval] long * count
```

Parameters

Parameter	Mode	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.

15.4.8 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	Mode	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
<code>ECU_OK</code>	Success.
<code>ECU_ConErr</code>	If the connection is invalid.
<code>ECU_DevErr</code>	If the device ID is invalid.
<code>ECU_IDErr</code>	If the event ID is invalid.

15.4.9 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	Mode	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.

15.4.10 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	Mode	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.

15.4.11 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	Mode	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.

15.4.12 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	Mode	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.

15.4.13 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	Mode	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.

15.5 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`.

15.5.1 Method: `OnEventChange()`

Event hook callback.

Method declaration

```
HRESULT OnEventChange(  
    [in] long dev,  
    [in] long id,  
    [in] BSTR value
```

Parameters

Parameter	Mode	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

None.

15.6 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`.

15.7 ICVcDevice

This is the interface to a virtual device.

15.7.1 Method: DeviceID()

Get the device's ID.

Method declaration

```
HRESULT _stdcall DeviceID(  
    [out, retval] long * id
```

Parameters

Parameter	Mode	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.

15.7.2 Method: DoAction()

Send a data-less action.

Method declaration

```
HRESULT _stdcall DoAction(  
    [in] long id,  
    [out, retval] IVcECU_Status * status
```

Parameters

Parameter	Mode	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.

15.7.3 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	Mode	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.

15.7.4 Method: GetActionByName()

Get an interface to an action by name.

Method declaration

```
HRESULT _stdcall GetActionByName(  
    [in] BSTR id,  
    [out, retval] ICVcAction ** action
```

Parameters

Parameter	Mode	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.

15.7.5 Method: GetActionCount()

Get the number of actions supported by the device.

Method declaration

```
HRESULT _stdcall GetActionCount(  
    [out, retval] long * count
```

Parameters

Parameter	Mode	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.

15.7.6 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	Mode	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.

15.7.7 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	Mode	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.

15.7.8 Method: GetEventCount()

Get the number of events supported by the device.

Method declaration

```
HRESULT _stdcall GetEventCount(  
    [out, retval] long * count
```

Parameters

Parameter	Mode	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.

15.7.9 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	Mode	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.

15.7.10 Method: Name()

Get the device's name.

Method declaration

```
HRESULT _stdcall Name(  
    [out, retval] BSTR * name
```

Parameters

Parameter	Mode	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.

15.7.11 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	Mode	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.

15.7.12 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	Mode	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.

15.7.13 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	Mode	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.

15.7.14 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	Mode	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.

15.7.15 Method: SendFormat()

Get the data format for an action.

Method declaration

```
HRESULT _stdcall SendFormat(  
    [in] long id,  
    [out, retval] BSTR * value
```

Parameters

Parameter	Mode	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.

15.8 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`.

15.8.1 Method: `OnEventChange()`

Event hook callback.

Method declaration

```
HRESULT OnEventChange(  
    [in] long id,  
    [in] BSTR value
```

Parameters

Parameter	Mode	Description
<code>id</code>	Input	The event ID.
<code>value</code>	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

None.

15.9 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.

15.10 ICVcAction

This is the interface to a virtual device action.

15.10.1 Method: ActionID()

Get the action's ID.

Method declaration

```
HRESULT _stdcall ActionID(  
    [out, retval] long * id
```

Parameters

Parameter	Mode	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.

15.10.2 Method: Do()

Send a data-less action.

Method declaration

```
HRESULT _stdcall Do (  
    [out, retval] IVcECU_Status * status
```

Parameters

Parameter	Mode	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.

15.10.3 Method: Name()

Get the action's name.

Method declaration

```
HRESULT _stdcall Name(  
    [out, retval] BSTR * name
```

Parameters

Parameter	Mode	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.

15.10.4 Method: Send()

Send an action containing data.

Method declaration

```
HRESULT _stdcall SendAction(  
    [in] BSTR value,  
    [out, retval] IVcECU_Status * status
```

Parameters

Parameter	Mode	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.

15.10.5 Method: SendFormat()

Get the data format for an action.

Method declaration

```
HRESULT STDMETHODCALLTYPE SendFormat(  
    [out, retval] BSTR * value
```

Parameters

Parameter	Mode	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.

15.11 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.

15.12 ICVcEvent

This is the interface to a virtual device event.

15.12.1 Method: EventID()

Get the events's ID.

Method declaration

```
HRESULT _stdcall EventID(  
    [out, retval] long * id
```

Parameters

Parameter	Mode	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.

15.12.2 Method: Hook()

Hook or unhook the event.

Method declaration

```
HRESULT _stdcall Hook(  
    [in] long value,  
    [out, retval] IVcECU_Status * status
```

Parameters

Parameter	Mode	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.

15.12.3 Method: Name()

Get the event's name.

Method declaration

```
HRESULT _stdcall Name(  
    [out, retval] BSTR * name
```

Parameters

Parameter	Mode	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.

15.12.4 Method: Query()

Query the value of an event.

Method declaration

```
HRESULT _stdcall QueryEvent(  
    [in, out] BSTR * value,  
    [out, retval] IVcECU_Status * status
```

Parameters

Parameter	Mode	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.

15.12.5 Method: QueryFormat()

Get the data format for an event's input data.

Method declaration

```
HRESULT _stdcall QueryFormat(  
    [out, retval] BSTR * value
```

Parameters

Parameter	Mode	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.

15.12.6 Method: ReplyFormat()

Get the data format for an event's value.

Method declaration

```
HRESULT _stdcall ReplyFormat(  
    [out, retval] BSTR * value
```

Parameters

Parameter	Mode	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.

15.13 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`.

15.13.1 Method: OnEventChange()

Event hook callback.

Method declaration

```
HRESULT OnEventChange(  
    [in] BSTR value
```

Parameters

Parameter	Mode	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

None.

16 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 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-OS3.0 kernel. A Virtual ECU is the RTA-OS3.0 for PC analog of an application running on a real ECU.

Virtual Machine (VM) The Virtual Machine is the component of RTA-OS3.0 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-OS3.0 for PC that allows Virtual ECUs running on the same or a remote PC to be monitored and managed.

VRTA Virtual RTA-OS3.0. used to describe the RTA-OS3.0 port plug-in for the Virtual ECU.

vrtaServer The vrtaServer(.exe) program is an application supplied with RTA-OS3.0 for PC that manages access to Virtual ECU. It runs as a normal application or as a Windows service.

17 **Contacting ETAS**

17.1 **Technical Support**

Technical support is available to all RTA-OS3.0 users with a valid support contract. If you do not have such a contract then please contact ETAS through one of the addresses listed in Section [17.2](#).

The best way to get technical support is by email. Any problems or questions should be sent to: rta.hotline.uk@etas.com

It is helpful if you can provide support with the following information:

- your support contract number.
- your .xml/.rtaos configuration files.
- the error message you received and the file `Diagnostic.dmp` if it was generated.
- the command line that results in an error message.
- the version of the ETAS tools you are using.
- the version of your compiler tool chain you are using.

If you prefer to discuss your problem with the technical support team you can contact them by telephone during normal office hours (0900-1730 GMT/BST). The telephone number for the RTA-OS3.0 support hotline is: +44 (0)1904 562624.

17.2 General Enquiries



Europe

Excluding France, Belgium, Luxembourg, United Kingdom and Scandinavia

ETAS GmbH

Borsigstrasse 14
70469 Stuttgart
Germany

Phone: +49 711 89661-0
Fax: +49 711 89661-300
E-mail: sales.de@etas.com
WWW: www.etas.com

France, Belgium and Luxemburg

ETAS S.A.S.

1, place des États-Unis
SILIC 307
94588 Rungis Cedex
France

Phone: +33 1 56 70 00 50
Fax: +33 1 56 70 00 51
E-mail: sales.fr@etas.com
WWW: www.etas.com

United Kingdom and Scandinavia

ETAS Ltd.

Studio 3, Waterside Court
Third Avenue, Centrum 100
Burton-upon-Trent
Staffordshire DE14 2WQ
United Kingdom

Phone: +44 1283 54 65 12
Fax: +44 1283 54 87 67
E-mail: sales.uk@etas.com
WWW: www.etas.com

USA

ETAS Inc.

3021 Miller Road
Ann Arbor
MI 48103
USA

Phone: +1 888 ETAS INC
Fax: +1 734 997-9449
E-mail: sales.us@etas.com
WWW: www.etas.com

Japan

ETAS K.K.

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

Phone: +81 45 222-0900
Fax: +81 45 222-0956
E-mail: sales.jp@etas.com
WWW: www.etas.com

Korea

ETAS Korea Co. Ltd.

4F, 705 Bldg. 70-5
Yangjae-dong, Seocho-gu
Seoul 137-889
Korea

Phone: +82 2 5747-016
Fax: +82 2 5747-120
E-mail: sales.kr@etas.com
WWW: www.etas.com

P.R.China

ETAS (Shanghai) Co., Ltd.

2404 Bank of China Tower
200 Yincheng Road Central
Shanghai 200120
P.R. China

Phone: +86 21 5037 2220
Fax: +86 21 5037 2221
E-mail: sales.cn@etas.com
WWW: www.etas.com

India

ETAS Automotive India Pvt. Ltd.

No. 690, Gold Hill Square, 12F
Hosur Road, Bommanahalli
Bangalore, 560 068
India

Phone: +91 80 4191 2585
Fax: +91 80 4191 2586
E-mail: sales.in@etas.com
WWW: www.etas.com

Index

A

Action, [63](#)
Actuators, [41](#)
Application Manager, [27](#), [136](#)
 Actions
 GetInfo, [137](#)
 Pause, [137](#)
 ReadOption, [137](#)
 ReadParam, [138](#)
 Reset, [137](#)
 Restart, [137](#)
 Start, [136](#)
 Terminate, [136](#)
 TestOption, [137](#)
 Events
 Info, [138](#)
 Option, [139](#)
 OptionText, [139](#)
 ParamText, [139](#)
 Paused, [138](#)
 Reset, [138](#)
 Restarted, [138](#)
 Started, [138](#)
 State, [139](#)
 Terminated, [138](#)
AUTOSAR OS includes
 Os.h, [36](#)
 Os_Cfg.h, [36](#)
 Os_MemMap.h, [36](#)

C

C++
 Exporting functions to C, [46](#)
C/C++ Interfacing, [45](#)
CAPCOM
 seeCompare Devices, [39](#)
Clocks, [39](#)
Command Line, [185](#)
Compare Devices, [39](#)
Counters, [39](#)
CVcAction, [215](#), [270](#)
CVcDevice, [215](#), [253](#)
CVcECU, [213](#), [238](#)

CVcEvent, [215](#), [276](#)
CVcServer, [211](#), [224](#)

D

Data Format Strings, [94](#)
Debugging, [31](#)
Device Manager, [25](#), [131](#)
 Actions
 EventRegister, [131](#)
 GetDeviceActions, [132](#)
 GetDeviceEvents, [132](#)
 GetDeviceInfo, [132](#)
 HookEvents, [131](#)
 ListAll, [132](#)
 Events
 DeviceActions, [132](#)
 DeviceEvents, [133](#)
 DeviceInfo, [133](#)
 DeviceList, [132](#)
Devices, [63](#)
 Actions, [64](#)
 Events, [65](#)
 Initialization, [37](#)
 Interrupts, [65](#)
 Logger, [44](#)
 Registration, [63](#)
 State, [64](#)
DM, [25](#)

E

ECU, [23](#)
Embedded GUI, [28](#)
Event, [63](#)

I

ICU, [27](#)
ICVcAction, [271](#)
ICVcDevice, [254](#)
ICVcDeviceEvents, [269](#)
ICVcECU, [239](#)
ICVcECUEvents, [252](#)
ICVcEvent, [277](#)
ICVcEventEvents, [283](#)
ICVcServer, [225](#)

InitializeDevices, [37](#), [99](#)
Interrupt Control Unit, [27](#), [133](#)
 Actions
 Clear, [134](#)
 GetIPL, [135](#)
 GetPending, [135](#)
 Mask, [134](#)
 Raise, [134](#)
 SetIPL, [135](#)
 Unmask, [135](#)
 Events
 EnabledVecs, [136](#)
 IPL, [136](#)
 Pending, [135](#)
 Start, [135](#)
 Stop, [136](#)
Interrupts, [65](#), [88](#)

L

Library
 Name of, [36](#)

Linkage Table, [28](#)

N

Non-volatile Data, [58](#)

O

OS_MAIN, [100](#)

Oscillator, [39](#)

P

Problem Solving, [31](#)

R

Real hardware, [86](#)

Register Sets, [88](#)

RTA-TRACE, [29](#), [60](#), [81](#)

rtcVRTALink.dll, [81](#), [82](#)

S

Samples Devices, [140](#)

Security, [72](#)

Sensors, [40](#)

Standard Devices, [130](#)

T

TCP/IP Ports, [72](#)

Threads, [53](#), [68](#)
 VECU Priority, [82](#)

Tutorial, [33](#)
 Prerequisites, [33](#)

V

VECU, [23](#), [28](#)
 Library, [191](#)
 Using RTA-TRACE with, [29](#)

VECU Interaction, [69](#)
 Real hardware, [56](#)
 Threads, [53](#)
 vrtaMonitor, [47](#)

VesLib.dll, [191](#)

VesLibAttachToECU, [194](#)

VesLibCreateAlias, [195](#)

VesLibEcuAliasType, [192](#)

VesLibEcuInfoType, [192](#)

VesLibExit, [196](#)

VesLibFindECUs, [197](#)

VesLibFreeAlias, [199](#)

VesLibFreeMemory, [200](#)

VesLibGetAliases, [201](#)

VesLibGetInfo, [203](#)

VesLibInitialize, [204](#)

VesLibListAliases, [205](#)

VesLibListLoadedECUs, [206](#)

VesLibLoadECU, [207](#)

VesLibSelectServer, [209](#)

Virtual Devices, [25](#), see [Devices](#)

Virtual ECU, [23](#)

Virtual Machine, [23](#), [25](#)
 API, [89](#)
 Application Manager, [27](#)
 Device Manager, [25](#)
 Embedded GUI, [28](#)
 Interrupt Control Unit, [27](#)
 Linkage Table, [28](#)

Visual Basic, [216](#)

VM, [23](#), [25](#)
 API, [89](#)
 Linking to, [28](#)

VRTA, [23](#)

- vrtaActEvID, 89
- vrtaAction, 63, 89
- vrtaActionID, 90
- vrtaActuator, 41, 166
 - Actions
 - Maximum, 168
 - Value, 168
 - Events
 - Maximum, 168
 - Value, 168
 - Methods
 - GetMax, 166
 - SetMax, 167
 - SetVal, 167
 - Value, 167
 - vrtaActuator, 166
- vrtaActuatorDimmableLight, 41, 171
 - Actions
 - Maximum, 173
 - Value, 173
 - Events
 - Maximum, 173
 - Value, 173
 - Methods
 - GetMax, 171
 - SetMax, 172
 - SetVal, 172
 - Value, 172
 - vrtaActuatorDimmableLight, 171
- vrtaActuatorLight, 41, 169
 - Actions
 - Value, 170
 - Events
 - Value, 170
 - Methods
 - SetVal, 169
 - Value, 169
 - vrtaActuatorLight, 169
- vrtaActuatorMultiColorLight, 42, 174
 - Actions
 - Maximum, 176
 - Value, 176
 - Events
 - Maximum, 176
 - Value, 176
 - Methods
 - GetMax, 174
 - SetMax, 175
 - SetVal, 175
 - Value, 175
 - vrtaActuatorMultiColorLight, 174
- vrtaBoolean, 90
- vrtaByte, 90
- vrtaClock, 39, 142
 - Actions
 - Interval, 144
 - Scale, 144
 - Start, 144
 - Stop, 144
 - Events
 - Interval, 145
 - Running, 145
 - Scale, 145
 - Methods
 - SetInterval, 142
 - SetScale, 143
 - Start, 143
 - Stop, 144
- vrtaCompare, 40, 177
 - Actions
 - Match, 179
 - Vector, 180
 - Events
 - Match, 180
 - Methods
 - GetMatch, 178
 - IncrementMatch, 179
 - SetMatch, 178
 - SetVector, 179
 - vrtaCompare, 177
 - Raising Interrupts, 40
 - with Actuators, 41
 - with Sensors, 41
- vrtaDataLen, 91
- vrtaDevice, 39
- vrtaDevice.h, 39

- vrtaDevID, 91
- vrtaDownCounter, 152
 - Actions
 - Maximum, 156
 - Minimum, 156
 - Report, 156
 - Set, 156, 157
 - Start, 156
 - Stop, 156
 - Methods
 - Max, 153
 - Min, 152
 - SetMax, 154
 - SetMin, 153
 - SetVal, 154
 - Start, 155
 - Stop, 155
 - Value, 153
- vrtaEmbed, 91
- vrtaEnterUninterruptibleSection, 102
- vrtaErrType, 91
- vrtaEvent, 92
- vrtaEventID, 93
- vrtaEventRegister, 103
- vrtaEventUnregister, 105
- vrtaGetState, 106
- vrtaHookEvent, 108
- vrtaInitialize, 110
- vrtaIntPriority, 93
- vrtaIO, 42, 181
 - Actions
 - GetValue, 183
 - GetValues, 183
 - Value, 183
 - Values, 183
 - Events
 - Value, 183
 - Values, 184
 - Methods
 - GetValue, 182
 - GetValues, 182
 - SetValue, 181
 - SetValues, 181
 - vrtaIO, 181
- vrtaIsAppFinished, 46, 113
- vrtaIsAppThread, 114
- vrtaIsIdle, 46, 115
- vrtaISRID, 93
- vrtaLeaveUninterruptibleSection, 116
- vrtaLoadVM, 117
- vrtaLoggerDevice.h, 44
- vrtaMillisecond, 93
- vrtaMonitor, 29, 69, 73
 - Monitoring events, 76
 - Scripting, 78
 - Sending actions, 75
 - Viewing events, 75
- vrtaMSCOM.dll, 23, 211
- vrtaOptStringlistPtr, 93
- vrtaRaiseEvent, 118
- vrtaReadHPTime, 119
- vrtaRegisterVirtualDevice, 120
- vrtaReset, 125
- vrtaSampleDevices.h, 39, 140
- vrtaSendAction, 126
- vrtaSensor, 40, 158
 - Actions
 - Maximum, 160
 - Value, 160
 - Events
 - Maximum, 160
 - Value, 160
 - Methods
 - GetMax, 158
 - SetMax, 159
 - SetVal, 159
 - Value, 159
 - vrtaSensor, 158
- vrtaSensorMultiwaySwitch, 41, 163
 - Actions
 - Maximum, 165
 - Value, 165
 - Events
 - Maximum, 165
 - Value, 165
 - Methods
 - GetMax, 163

- SetMax, 164
- SetVal, 164
- Value, 164
- vrtaSensorMultiwaySwitch, 163
- vrtaSensorToggleSwitch, 41, 161
 - Actions
 - Position, 162
 - Events
 - Position, 162
 - Methods
 - SetVal, 161
 - Value, 161
 - vrtaSensorToggleSwitch, 161
- vrtaServer, 29, 69
 - Aliasing VECUs, 77
 - Installing as a Windows service, 69
 - Security, 72
- vrtaSpawnThread, 127
- vrtaStart, 128
- vrtaStringlistPtr, 93
- vrtaTerminate, 129
- vrtaTextPtr, 93
- vrtaTimestamp, 94
- vrtaUpCounter, 146
 - Actions
 - Maximum, 150
 - Minimum, 150
 - Report, 150
 - Set, 150
 - Start, 150
 - Stop, 150
- Events
 - Set, 151
- Methods
 - Max, 147
 - Min, 146
 - SetMax, 148
 - SetMin, 147
 - SetVal, 148
 - Start, 149
 - Stop, 149
 - Value, 147
- vrtaVM.dll, 25
 - Location, 85
 - Search path, 25, 85

W

- Windows
 - Library Functions, 84
 - Real-Time behavior, 84

X

- XML Configuration, 86