Programming with CAPL

CANalyzer
CANoe

vector
the art of engineering

Programming With CAPL

December 14, 2004

First printing

Vector CANtech, Inc.
Suite 550
39500 Orchard Hill Place
Novi, MI 48375
USA
http://www.vector-cantech.com

Preface IX
About This Book IX
Organization IX
Acknowledgments IX
Tell Us What You Think! X
1 Introduction to CAPL 1
1.1 Evaluation Capability 1
1.2 Simulation Capability 1
1.3 Prerequisites for Using CAPL 2
1.4 Learning Steps 2
1.5 CAPL – Event-Driven Software 3
1.6 CAPL Program Organization 3
1.7 Using the CAPL Browser for Program Development 3
1.8 Program Development Uses the Database Tool – CANdb++ 3
1.9 CAPL Programming Knowledge 4
1.10 CAN Communication Knowledge 4
1.11 THE WARNING – Welcome to the Jungle 4
1.12 Responsibility 5
2 CANalyzer and CANoe 6
2.1 One Key Difference – Level of Node Control 6
2.2 Graphic Panels – The Other Major Difference 7
3 A Brief Introduction to CANalyzer 9
3.1 Value of the Downloadable Demo 9
3.2 First-Time Considerations 9
3.3 How to Start and Stop CANalyzer 10
3.4 The Architecture of CANalyzer 10
3.5 Measurement Setup Window 12
3.5.1 Trace Window 12
3.5.2 Statistics Window 12
3.5.3 Bus Statistics Window 13
3.5.4 Data Window 13
3.5.5 Graphics Window 13
3.5.6 Write Window 13
3.6 Duplicating Analysis Blocks 13
3.7 Controlling Data Flow 13
3.8 Function Blocks 13
3.8.1 Filtering Blocks 14
3.8.2 Message Blocks 15
3.9 Using CAPL Program Blocks in CANalyzer 17
3.9.1 CANalyzer Program Blocks for Transmission 18
3.9.2 Program Blocks for Analysis 19
3.10 CANalyzer – CAPL Programming Environment 20
4 A Brief Introduction to CANoe 21
4.1 Value of the Downloadable Demo 21
4.2 First-Time Considerations 21
4.3 Learning from the Demo 22
4.4 Tool Architecture of CANoe 23
4.5 Using CAPL Program Blocks in CANoe 24
4.5.1 CANoe Program Blocks for Simulation 24
4.5.2 CANoe Program Blocks for Analysis 25
4.6 CANoe – CAPL Programming Environment 25
5 Using Databases with CAPL 26
5.1 Why Use a Database with CAPL? 26
5.1.1 Additional Uses of the Database 26
5.2 Database Association with CAPL 26
5.3 Database Objects Accessible by CAPL 27
5.3.1 Attributes 28
5.3.2 Value Tables 28
5.3.3 Network Nodes 28
5.3.4 Messages 28

5.3.5 Signals 28
5.3.6 Environment Variables 28
6 Using Panels with CAPL (CANoe only) 29
7 The CAPL Browser 30
7.1 Starting the CAPL Browser 30
7.2 Browser Organization 31
7.3 Using the Right Mouse Button 32
7.4 The Events Window 32
7.4.1 Creating an Event Procedure 32
7.5 The Global Variables Window 32
7.6 The Event Procedure Window 33
7.7 Colorized CAPL Syntax 33
7.8 Compiling Code 33
7.9 Fixing Compilation Errors 34
7.10 Debugging Run-Time Errors 34
7.11 Using Other Text Editors 35
7.12 CAPL Source File Format 35
8 CAPL Syntax and Semantics 37
8.1 Major CAPL Differences from C 37
8.2 CAPL Equivalents to C Functions 38
8.3 Notation 38
8.4 Comments 38
8.5 Naming Conventions 39
8.6 CAPL is Case Sensitive 39
8.7 CAPL Keywords 39
8.8 Data Types 40
8.9 Declarations 41
8.9.1 Local Variables are Static 41
8.9.2 Initialization 42
8.10 Type Casting 42
8.11 Arrays 42
8.12 Constants 43
8.12.1 Integer Constants 43
8.12.2 Floating Point Constants 44
8.12.3 Character Constants 44
8.12.4 String Constants 45
8.12.5 Symbolic Constants 45
8.13 Operators 46
8.13.1 Arithmetic Operators 46
8.13.2 Assignment Operators 47
8.13.3 Boolean Operators 48
8.13.4 Relational Operators 49
8.13.5 Bitwise Operators 50
8.13.6 Miscellaneous Operators 50
8.13.7 Unsupported Operators 51
8.14 Control Statements 51
8.14.1 Branching Statements 51
8.14.2 If Statement 52
8.14.3 If-Else Statement 52
8.14.4 Switch Statement 53
8.15 Loops 54
8.15.1 While Statement 54
8.15.2 Do–While Statement 55
8.15.3 For Statement 55
8.15.4 Unconditional Branching 56
8.15.5 Return Statement 57
8.16 The “this” Keyword 57
9 CAPL Functions 59
9.1 User-defined Functions 59
9.2 Function Overloading is Allowed 59

9.3 Function Naming Conventions 60
9.4 CAPL Function Categories 60
9.4.1 Mathematical Functions 60
9.4.2 User Interface Functions 61
9.4.3 Time Functions 62
9.4.4 Message Handling Functions 63
9.4.5 Database Functions 64
9.4.6 Byte Ordering Functions 64
9.4.7 Logging Functions 65
9.4.8 String Handling Functions 66
9.4.9 Measurement Control Functions 66
9.4.10 Statistics Functions 67
9.4.11 CAN Protocol Functions 68
9.4.12 Port Functions 69
9.4.13 Replay Block Functions 69
9.4.14 Environment Variable and Panel Functions (CANoe only) 70
9.4.15 Miscellaneous Functions 71
9.5 CAPL Function Compatibilities 72
10 CAPL Events and Event Procedures 73
10.1 Creating an Event Procedure 74
10.2 Event Procedure Requirements 74
10.3 Some Event Procedures Use the Keyword “this” 75
10.4 The * symbol 75
10.5 Event Priority 76
11 Using Messages 78
11.1 Messages in CAPL 78
11.1.1 Declaring Messages in CAPL with a Database 78
11.1.2 Declaring Messages in CAPL Without a Database 79
11.1.3 Message Selectors 79
11.2 Accessing Data 81
11.2.1 Using Signals (Only Available with a Database) 82
11.2.2 Physical Values and the “phys” Attribute 82
11.2.3 Round-Off Error in Symbolic Signal Access 83
11.3 Message Transmission 83
11.4 Message Reception 83
11.5 Error Frames 86
11.5.1 Error Frame Event 86
12 Using Keyboard Events 87
12.1 Keyboard Events and Procedures 87
12.2 Using the Keyword “this” with the Wildcard Symbol “*” 87
13 Using System Events 89
13.1 Types of System Events 89
13.2 What Happens When a System Event Occurs? 89
14 Using Timers 91
14.1 Declaring a Timer 91
14.2 Starting a Timer 91
14.3 Expiration of a Timer 91
14.4 Resetting a Timer 92
14.5 Periodic Clock 92
14.6 Stopping a Timer Before It Expires 92
14.7 Common Timer Mistake 92
15 Using CAN Protocol Controller Events 94
15.1 CAN Controller States 94
15.2 CAN Controller Events 95
16 Using Environment Variables 96
16.1 Environment Variable Types 97
16.2 Environment Variable Initialization 97
16.3 Declaring an Environment Variable Event 97
16.4 Event Execution 98
16.5 The putValue() and getValue() Functions 98

16.6 The “this” Keyword 98
17 Using File Input/Output Functions 100
17.1 File Input/Output Functions 100
17.1.1 Set-Up for File Input/Output Functions 100
17.1.2 Set-Up for Sequential File Access Functions 101
18 Using the Serial and Parellel Port 103
18.1 Installation of RS232VC DLL 103
18.2 RS232 CAPL Functions 103
18.2.1 RS232SetCommState 103
18.2.2 RS232ByteCallback 104
18.2.3 RS232WriteByte 104
18.2.4 RS232WriteBlock 104
18.2.5 RS232EscapeCommFunc 105
18.2.6 RS232CloseHandle 105
19 Constructing CAPL Programs 106
19.1 CAPL Program Organization 106
19.1.1 Creating Network Nodes 106
19.1.2 P Block Placement and Message Pass-Through 107
20 CAPL Program Examples 108
20.1 The Write() Function 108
20.2 Sending a Periodic Message 108
20.3 Sending a Conditionally Periodic Message 110
20.4 Reading Data in a Received Message 111
20.5 Responding to a Message After a Delay 112
20.6 Sending an Error Frame 113
20.7 Using Panels 113
20.8 Bus Off Condition 114
20.9 Using a CAPL Program to Control Logging 115
20.10 Data Files and CAPL 116
20.10.1 Set 1: File I/O Functions 117
20.10.2 Set 2: ProFile Functions 119
21 Basic Steps in Creating Your First CANoe Application 121
21.1 CANoe Development - Five Step Process 121
21.1.1 Create a New Directory 122
21.1.2 Creating a New CANoe Configuration 122
21.1.3 Bus Parameters 123
21.1.4 Set Up Analysis Functions 124
21.1.5 Working with the Analysis Windows 125
22 Using CANdb++ to Create a Database 126
22.1 CANdb++ Database Development 126
22.2 Starting the CANdb++ Program 127
22.3 Defining Attributes 127
22.4 Defining Value Tables 128
22.5 Defining Nodes 128
22.6 Defining Messages 128
22.7 Defining Signals 129
22.8 Establishing Object Associations 130
22.9 Defining Environment Variables 130
22.10 Saving the Database 131
22.11 Associating a Database 131
23 Using the Panel Editor to Create Panels 132
23.1 What are Panels? 132
23.2 Advantages of Using Panels 133
23.3 Requirements 133
23.3.1 Database Association 133
23.3.2 Environment Variables 133
23.4 Creating a New Panel 134
23.4.1 Properties of the Panel 134
23.4.2 Using Elements 135
23.5 Configuring Elements 135

23.5.1 Associating Elements ... 135
23.5.2 Alarms ... 136
23.6 Using Bitmaps ... 138
23.6.1 Creating a Dynamic Bitmap ... 138
23.7 Associating Panels to CANoe ... 139
24 Introduction To CAPL DLLs ... 140
24.1 DLL – Dynamic Link Library ... 140
24.2 CAPL DLLs ... 140
24.3 Performance ... 140
24.3.1 Using Microsoft Visual C++ to Implement a CAPL DLL ... 141
24.4 CAPL Export Table ... 142
24.5 Project Configuration ... 143
24.5.1 Compiling the DLL ... 143
24.6 Linking the CAPL DLL ... 143
24.6.1 CAN.ini File ... 144
24.6.2 DLL Search Sequence ... 144
24.6.3 Demo DLL and Source Code ... 145
24.6.4 C++ Code ... 146
24.6.5 Troubleshooting CAPL DLL Errors in the CAPL Browser ... 150
24.6.6 CAPL DLL Questions and Answers ... 150
25 Introduction to COM ... 151
25.1 COM – Component Object Model ... 151
25.2 CANoe/CANalyzer as a COM server ... 151
25.3 COM Server Registration ... 151
25.4 Using Microsoft Visual Basic to Access the COM server ... 151
25.4.1 Project Configuration ... 152
25.5 Controlling a Measurement ... 153
25.6 Accessing Communication Data ... 153
25.7 Accessing Environment Variables (CANoe only) ... 155
25.8 Reacting to CANoe/CANalyzer events ... 156
25.9 Calling CAPL Functions ... 157
25.10 Transmitting CAN Messages ... 159
25.11 COM Server Example in Microsoft Visual Basic ... 160
25.11.1 VB Source Code ... 161
26 Appendix A: Introduction to CAN Communications ... 166
26.1 The Distributed Application ... 166
26.2 Distributed Functions ... 166
26.2.1 A Common Example of a Distributed Function ... 168
26.3 Partitioning Requires Addressing ... 168
26.4 Distributed Functions Require a Transfer Dialog ... 169
26.5 A Distributed Function May Require File Transfer Capability ... 170
26.6 The Distributed Application and the OSI Seven-Layer Model ... 171
26.6.1 Physical Layer ... 172
26.6.2 Data Link Layer - Data Transfer Structure ... 172
26.6.3 Network Layer - Address Structure ... 173
26.6.4 Transport Layer - Message Transfer Structure ... 173
26.6.5 Session Layer - Conversation Structure ... 173
26.6.6 Presentation Layer - Data Structure ... 173
26.6.7 Application Layer - Application-to-Network Interface Structure ... 173
26.7 Distributed Applications Require Data Structures ... 173
26.8 Understanding the CAN Protocol ... 174
26.8.1 Key Attributes ... 174
26.8.2 Understanding the Basic Elements of the CAN Message ... 177
26.8.3 The CAN Error Frame ... 179
26.8.4 CAN Error Counting ... 179
26.8.5 The CAN Receive State Machine ... 179
26.8.6 The CAN Transmit State Machine ... 180
26.8.7 CAN Implementation Ingredients – CAN Controller, Software, Physical Layer ... 181
26.8.8 CAN Physical Layer ... 182
26.8.9 Physical Layer Choices ... 182

26.8.10 Basic CAN Physical Layer Principles 182
26.8.11 CAN Controller – Dominant/Recessive Logic Levels 183
26.8.12 Common Dominant/Recessive Logic Implementations 184
26.8.13 The Common High-Speed, Two-Wire Differential Transceiver 185
26.8.14 The Common CAN Transfer Medium - Wire 185
26.9 First-Time CAN Steps 188
26.9.1 Step One – Wiring First 188
26.9.2 Step Two – Add Modules Incrementally 188
26.9.3 Notes on Building Your Own CAN-Based Module 189
27 Appendix B: Abbreviations/Acronyms 190
28 Appendix C: Glossary 191
29 Index 195

Preface

Programming In CAPL introduces the basic concepts of the Vector CAN Access Programming Language (CAPL), the programming language foundation of Vector CANoe and CANalyzer – two of Vector's most popular development tools. CAPL is a rich, robust tool used to extend the power of CANoe and CANalyzer beyond the tool's interfaces and to customize tool functionality to the user's requirements.

About This Book

The target audience of this book is anyone needing to develop advanced, customized CAN networking tool solutions using Vector CANalyzer and CANoe, including engineering students, faculty, practicing engineers, and electronic technicians. This material is suitable for college programs that focus on electrical engineering, computer engineering, computer science, distributed control systems and distributed embedded systems that use the CAN protocol.

This book assumes that the programming experience level of the user will be individuals with little experience in the C programming language, in addition to those with C coding experience.

Additionally, this text assumes that the reader has a functional understanding of the CAN protocol. Readers needing to develop such an understanding are encouraged to peruse Appendix A: Introduction to CAN Communications.

Organization

This book is organized into logically arranged sections that allow the reader to build an understanding of CAPL in a clear and concise manner. There are six major parts to this book:
The first part (Chapters 2 through 7) introduces the reader to CAPL concepts and the CAPL working environment
The next part (Chapters 8 and 9) provides an overview of CAPL programming
The third part (Chapters 10 through 19) details CAPL functions, events and I/O
The fourth part (Chapters 20 through 24) prepares the reader to develop projects where CAPL is used
The fifth part (Chapters 26 through 28) discusses Advanced CAPL Concepts such as DLL usage and the CANoe COM interface
Lastly, the sixth section is comprised of a set of Appendices containing overviews for those engineers and students that are new to either CAN, distributed embedded networking or both.

Acknowledgments

The original creator of CAPL is Dr. Helmut Schelling, who also developed and authored the first compiler and first editor for the CAPL programming language.

Jurgen Kluser incorporated the data structural elements of the CAPL programming language into the Vector CANdb database tool. Additionally, those who participated in continuing the development of the CAPL programming language equally deserve credit, and these individuals include Thomas Riegraf and the CANoe/CANalyzer development teams.

On the authoring side, it is important to recognize several individuals who have made significant contributions to this book, including Jun Lin, Tom Guthrie, and Mike Alexander. Other contributors include Frank Voorburg, Max Sprauer, Brett Emaus, Maureen Emaus, and Todd Emaus.

Bruce Emaus

Tell Us What You Think!

We believe that you, the reader, are the most important person of all, since it is you who will benefit from reading this book. We value your input, and we would like to know what we're doing right, what we could do better, what things you think are important that we haven't covered, and any other comments you might have.

You may fax, e-mail or write us directly to let us know what you did or didn't like about this book – as well as what we should do to make our books better.

When you write, please include the title of this book, as well as your name and phone or fax number. We will carefully review your comments and share them with the authors and editors of this book.

E-mail: CAPL_Books@Vector-CANtech.com

Address: 39500 Orchard Hill Place Novi, MI USA 48375

Fax: 248-449-9704

1 Introduction to CAPL

Based on the C programming language, CAPL, or CAN Access Programming Language, is the programming language used exclusively within the PC-based tool environments of CANalyzer and CANoe. The original design intent behind CAPL (which is pronounced “kapple”) was to meet the CAN-based distributed embedded system developer’s requirements including:

- Maximum control of all test and measurement operations
- Maximum control of system or module simulation – CANoe- or CANalyzer-specific
- Maximum support for one or more communication channels
- Maximum event and message recording and playback control
- Ability to interconnect to other PC applications

The creation of CAPL and its programming environment became the implementation to meet these requirements.

Using CANalyzer or CANoe in combination with CAPL makes it possible to create custom tool applications with user-defined behavior. Potential applications are limited only by imagination, available communication hardware limitations (if applicable), and the speed of the PC.

1.1 Evaluation Capability

The CANalyzer or CANoe tool itself, without CAPL programs, is enough to execute simple measurement and analysis. With CAPL programs involved, measurement and analysis is greatly extended for CAN communication. One area that the tool cannot perform without CAPL is analysis that involves timing. CAPL can make analysis more efficient with the help of timers.

CAPL can be used in an application to:

- Analyze specific messages or specific data
- Analyze data traffic
- Create and modify the tool’s measurement environment
- Design a custom module tester
- Create a black box to simulate the rest of the network
- Create a module simulator
- Create a custom module manufacturing tester
- Create a custom module diagnostic or service tool
- Create programs to perform customized analysis of network logging (playback) files
- Create complex logging filters
- Create a comprehensive message or data content generation tester for module/network validation.
- Program a functional gateway between two different networks
- Evaluate the module network software strategy by generating CAN error frames in simulation to see if modules are working properly

1.2 Simulation Capability

The situation often arises when developing a distributed application that a portion or part of the application is not available for testing. The system environment can be emulated with the help of CAPL, for example, to simulate the data traffic of all remaining network nodes. For this reason, CAPL can also be used to simulate:

- Node or system behavior using readable English instructions and values rather than hexadecimal values
- Event messages, periodic messages, or conditionally repetitive messages
- Human events like button presses on the PC keyboard
- Timed node or network events
- Multiple time events, each with its own programmable behavior
- Normal operation, diagnostic operation, or manufacturing operation
- Changes in physical parameters or symbolic values (for example,, “ON”, “OFF”)
- Module and network faults to evaluate a limited operation strategy
- Simple or complex functions (sin, cos)

1.3 Prerequisites for Using CAPL

To use CAPL effectively you need the following:

- A CANalyzer or CANoe tool – the programmable version is required
- An understanding of the CAPL Browser – where you write your CAPL program
- A database tool – CANdb++ – to create your shared network data variables
- CAPL programming knowledge – available in this book
- A small amount of CAN communication knowledge

For those students and engineers who do not have access to a real CANalyzer or CANoe, consider using the demo version of the tools (available from the Download Center at http://www.vector-cantech.com). Even without real communication hardware, considerable experience can be gained using the demo version.

In addition, because CAPL is based on C, the user must be familiar with the C or C++ language. Users need not be familiar with programming orientation, mechanics, and its libraries, but on the syntax, operators, expressions and statements. Some common statements in C can also be found in CAPL. For example, the C function **printf()** is known as the **write()** function in both CANalyzer and CANoe to output data to the **Write** window. Both functions use exactly the same C/C++ formatting characters in the argument with the percent sign (%d for integers, %s for a string, and so on.).

1.4 Learning Steps

CAPL represents the programmable portion of CANoe and CANalyzer. Although there are significant differences between the two tools, both share a large set of features. For learning purposes, however, it is easier to learn CANalyzer first, as CANoe is built on the foundation of CANalyzer.

While both tools are valuable in the development of any distributed product or distributed embedded system architecture, CANoe possesses more powerful features than CANalyzer. In general, system and distributed product developers typically use CANoe.

Once you understand the structure of both CANoe and CANalyzer, it is relatively easy to understand where you can insert CAPL program blocks to implement the wide variety of available programmable functions. In general, a developer has the choice of using CAPL programs for simulation or for measurement analysis operations. As shown in Figure 1 below, once you understand the basic tool operation, where to insert program blocks, and how to use a database, mastering the CAPL programming language is the last step to becoming a power user of CANoe or CANalyzer.

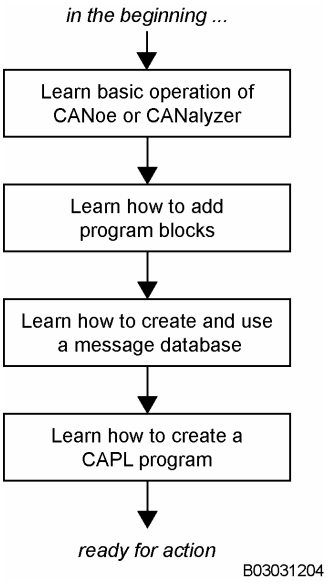

Figure 1 – General Steps to Learn CAPL

Because CAN communication systems use a large number of CAN messages with an even larger number of network data variables (or signals, as many prefer to call them), a database that can be used to manage this large volume of information is extremely valuable, not only for observing the communication information on the bus but also because the same information can be used inside the programming environment of CANalyzer and CANoe. Learning how to use the user-configurable database will simplify your development effort.

1.5 CAPL – Event-Driven Software

CAPL is an event-based procedural language that includes a large number of special-purpose functions to help the distributed embedded system developer. As explained in Figure 2, CAPL software is essentially event-driven.

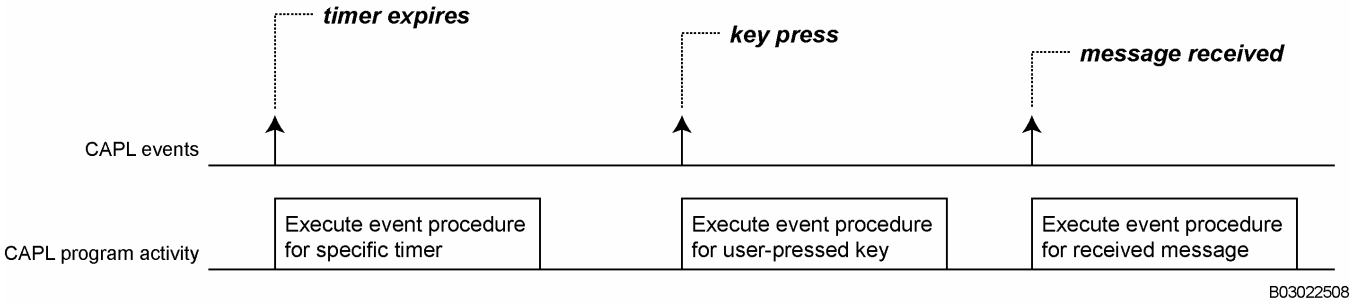

Figure 2 – CAPL Software Execution is Event-Driven

CAPL software can be created to respond to the following:

- Time events (software timers)
- I/O events (keyboard, serial port, parallel port)
- CAN communication events (messages, errors)

1.6 CAPL Program Organization

A CAPL program is organized into sections called event procedures. The CAPL Browser is used to parse a program into event procedures for display, modification and compilation. Because CAPL is an event-driven language, CAPL will only executes one event procedure at a time. If there are no events, you can say that the CAPL program is “idle”.

Extensive information regarding event procedures can be found in CAPL Events and Event Procedures – Chapter 10.

1.7 Using the CAPL Browser for Program Development

CAPL software is developed inside a complete development environment called the CAPL Browser, which is provided as an integral component of the programmable version of CANalyzer or CANoe. The CAPL Browser essentially combines a text editor and a compiler with an easy-to-use graphical user interface.

Because the overall structure of the CAPL Browser is based on the concept of events and is specifically organized around different event classes, the software development process is much simpler than the traditional organization of C programs.

The CAPL Browser also directly interconnects to databases, created by the CANdb++ Editor, to simplify the insertion of network data variable names. Common message databases can be shared by CAPL programmers to raise the consistency level across a development project at the system level.

To learn more about the CAPL Browser, see The CAPL Browser – Chapter 7.

1.8 Program Development Uses the Database Tool – CANdb++

The typical development of CAPL application software most likely uses CANdb++, the CAN database editor, to create and define the system’s network message set or database. CANdb++, an integral component of all versions of CANalyzer or CANoe, is a simple database editor and generator. The easy-to-use graphical user interface makes CANdb++ easy to learn; most of its basic capabilities can be easily implemented without a manual. For more information on databases, see Using CANdb++ to Create a Database – Chapter 22.

With CANdb++, you can perform all of the following tasks:

- Define network nodes
- Define messages and its properties
- Define and position data items or signals within a CAN message
- Define the physical units of a signal, if appropriate
- Define symbolic values for a signal, if appropriate
- Define common variables (known as attributes), if appropriate

Once a network message set is available, it can be shared by all development activities. This means that CAPL programmers, CAN module software engineers, CAN module hardware design engineers, test engineers, and technicians can all use the same database file when interconnecting tools to the bus.

Having a database containing all of your system's network data variables simplifies the process of CAPL development. This language tool, as well as symbolic access to the different variables defined in the database, allows software to be developed quicker and with a higher degree of consistency. The CAPL Browser interface also allows you to select user-defined variables directly from the database during CAPL implementation to avoid typing mistakes, and in some cases, just to see what variables have been defined.

1.9 CAPL Programming Knowledge

The CAPL programming language is based on C. Those familiar with C will need minimal learning time to begin using CAPL effectively. The syntax, most standard functions, and some mathematical functions of ANSI C have all been incorporated into CAPL, which extends the C foundation with extra network-specific functions and data types.

The CAPL Browser only compiles one file at a time, so no linking is necessary. Since the development process requires no header files, library functions, etc., CAPL programs are generally simpler to develop than C programs, which require third-party editors, compilers, and debuggers. Syntax and semantic errors are found at compile time, with run-time errors announced in a special window called the **Write** window inside the CANalyzer and CANoe environment.

1.10 CAN Communication Knowledge

When developing a CAN-based module or a portion of a distributed embedded system that uses the CAN protocol, it is helpful to have some basic understanding of the CAN communication system. Based on your level of involvement, there may be no need to learn the details of the CAN protocol. However, basic knowledge about CAN communication is quite valuable.

Available resource material may include information from the web, application notes, perhaps a book on the CAN protocol, a copy of the Bosch CAN Protocol 2.0B specification, or a nearby expert willing to be your encyclopedia.

A chapter at the end of this book also gives you some reference on the Appendix A: Introduction to CAN Communications – Chapter 26.

1.11 THE WARNING – Welcome to the Jungle

Welcome to the world of distributed systems. A button you press at your location might influence the system's behavior elsewhere – perhaps your button press has turned on a light at a remote location. This doesn't seem to be a problem, but what if the light was actually a powerful search beacon and, unfortunately, someone was standing next to it? Did you unintentionally just cause an accident?

When using, designing, testing or servicing a product or system that is distributed, you must be aware that you are potentially a member of the overall system. Your actions can influence the behavior of a network-based distributed embedded system.

Depending on the application, the consequences of improper actions could cause serious operational malfunction, damage to equipment, and physical injury to yourself and others. Imagine working in an engineering development lab that is designing and evaluating distributed air bag systems – you could be one button press from a serious accident.

Only those persons who understand the warnings contained in this book and agree to the conditions of use may operate this product.

WARNING: A potentially hazardous operating condition is present when the following two conditions are concurrently true:

- **The system or product is physically interconnected to the real world**
- **The functions and operations of the real distributed embedded system are controllable or influenced by the use of a network-based communication system.**

WARNING: A potentially hazardous operating condition may result from the activity or non-activity of some distributed embedded system functions and operations, which may result in serious physical harm or death or cause damage to equipment, devices, or the surrounding environment.

Note: By using distributed embedded system technology or using Vector tools, the operator may potentially:

- Cause a change in the operation of the system, module, device, circuit, or output
- Turn on or activate a module, device, circuit, output, or function
- Turn off or deactivate a module, device, circuit, output, or function
- Inhibit, turn off, or deactivate normal operation
- Modify the behavior of a distributed product
- Activate an unintended operation
- Place the system, module, device, circuit, or output into an unintended mode

1.12 Responsibility

Only those persons who understand the surrounding circumstances and accept the responsibility should be allowed access to the tools and technology. When participating in the development, testing, or service of any distributed system, you must understand that as a network member your involvement may impact the safety of the system.

YOU ARE RESPONSIBLE – AND ASSUME ALL RISKS AND LIABILITY FROM THE OPERATION OF THESE PRODUCTS.

2 CANalyzer and CANoe

The CANalyzer and CANoe tools were developed to meet the essential needs of the CAN-based module or system developer by combining a comprehensive set of measurement and simulation capabilities.

Both CANalyzer and CANoe can interface to multiple CAN networks (or other common small area network protocols), and provide accurate time-stamped measurements for all communication transfers, including both acknowledged messages and communication errors. Recording and playback operations are standard. Users can record the messages from one system and e-mail them to another engineer for playback and analysis.

Both tools basically operate like a multi-channel oscilloscope, a multi-channel logic analyzer, and a custom alphanumeric display unit - all using an integrated database.

In addition, both tools are capable of creating any message generation pattern, much like a programmable function generator, with complete control of all network data variables (or signals).

As shown in Figure 3, both CANoe and CANalyzer share a major portion of the same network analysis interface.

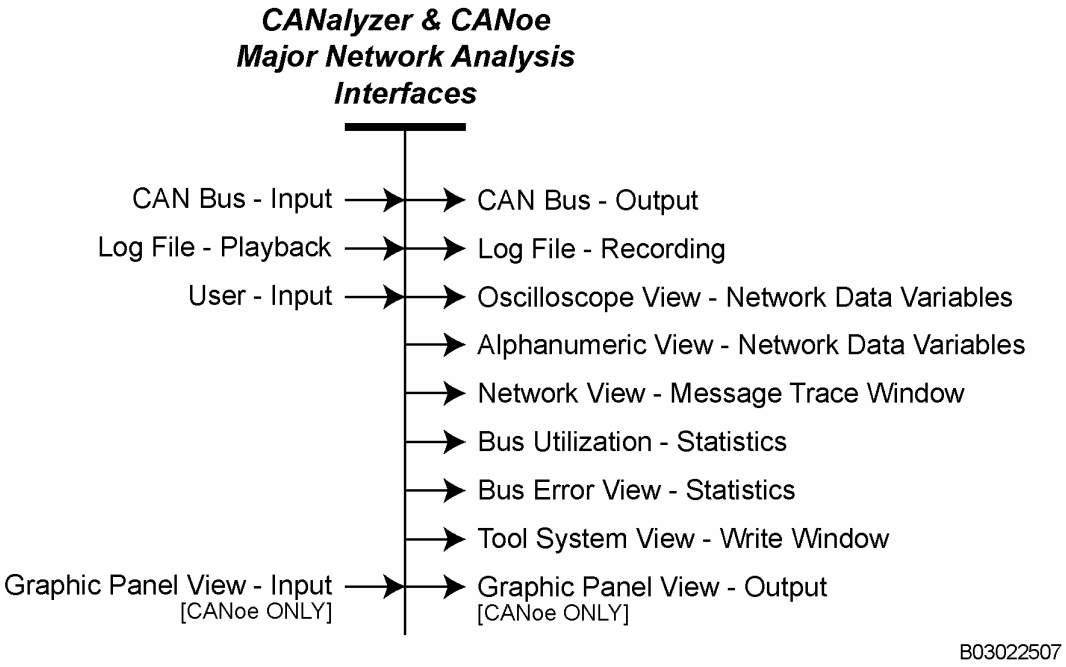

Figure 3 – CANalyzer & CANoe Major Network Analysis Interfaces

Because one tool is usually a better choice over the other for most applications, you need to understand the key differences between CANalyzer and CANoe before starting a project.

2.1 One Key Difference – Level of Node Control

One key difference between CANalyzer and CANoe is in the level of node control. Essentially, a single CANalyzer tool can act as a single network member, but CANoe has no limit as to the number of modules with which it may substitute.

As shown in Figure 4, CANalyzer supports the control of a single node (a single tester, or a single module simulation), while CANoe supports the control of a collection of multiple nodes (any number of module simulations or any number of testers).

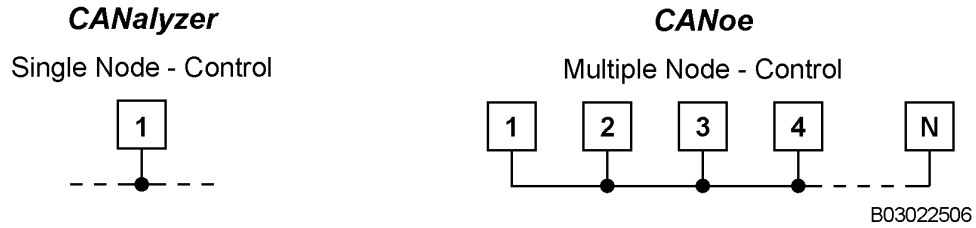

Figure 4 - Level of Node Control Distinguishes Between CANalyzer and CANoe

In CANoe, each node may be enabled to evaluate a simulation, or each node may be disabled to allow connection of a real module to the “remaining network simulation”. This can be done in real time for any number of nodes and for one or more communication networks.

As shown in Figure 5, the ability to interconnect a real module to CANoe that represents “all the other remaining network members” provides a significant testing advantage in distributed product architecture.

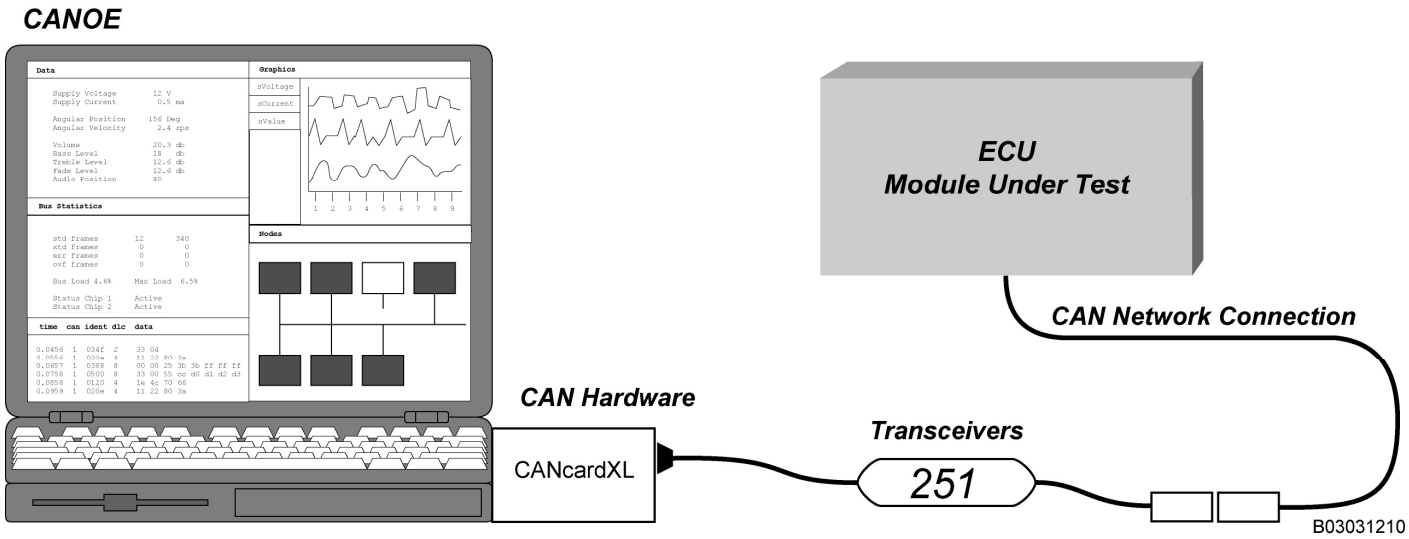

Figure 5 – Using CANoe to Simulate the Rest of the System

The limitations when using CANoe depend on both the speed of the available PC and the amount of CAN hardware that can be placed on a single PC. While laptops are typically limited to 4 CAN network connections (2 PCMCIA cards with 2 CAN channels each), desktop configurations with up to 32 CAN channels have been created for special applications.

2.2 Graphic Panels – The Other Major Difference

The second and quite distinctive difference from CANalyzer is that CANoe supports “graphic panels” for both inputs and outputs. This allows the user to construct “higher-level application” behavior to simulate actual inputs and outputs. For example, let's assume that your new project requires you to build a tester. Traditionally, you would typically choose between two alternatives:

- Build a custom electronic module - design all the hardware and software yourself
- Build a semi-customized PC-based system - choose some PC-compatible hardware and design the software yourself

However, another choice is now available - you could construct the entire tester in CANoe and write the entire application in CAPL.

CANoe allows you to construct tester panel interfaces to give inputs and outputs. You can add the necessary CAPL software to interconnect your switch presses to the corresponding CAN transmit messages that you wish the tester to

send. It is also easy to connect incoming CAN receive messages to your front panel graphic output devices. In addition, moving meters, blinking lights, and numerical display graphics are easy to create (see Figure 6).

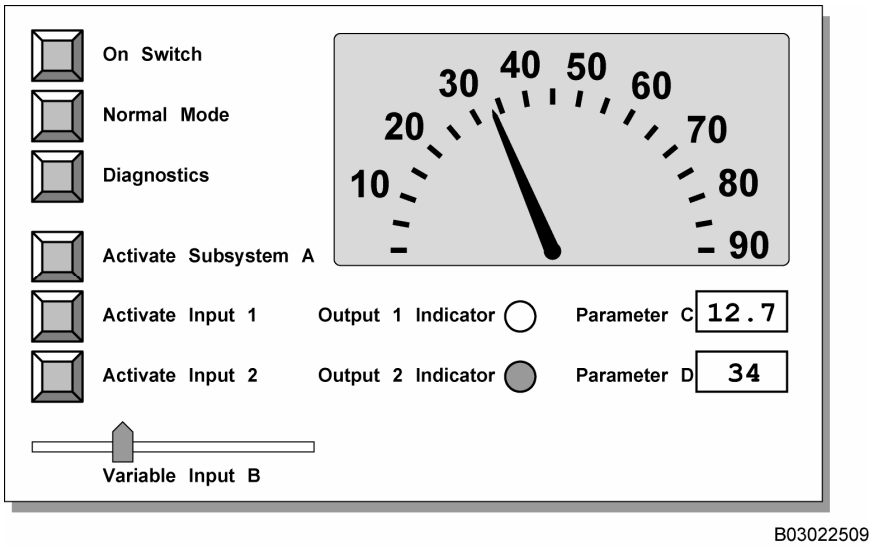

Figure 6 - Example of CANoe graphics used for both Front Panel Input and Output

Bit-mapped graphics and digital photos, as shown in Figure 7, of actual product front panels can be easily animated for use.

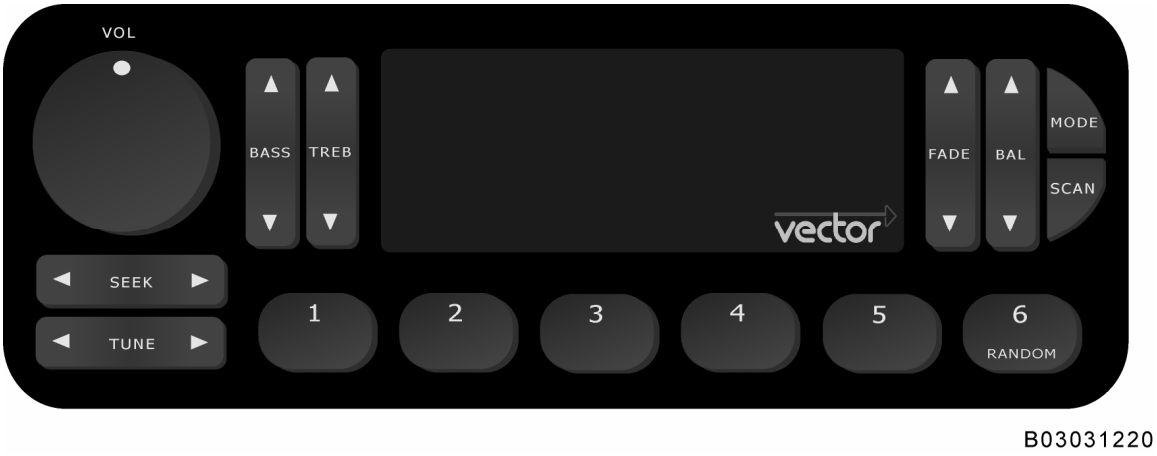

Figure 7 – Example of User-Designated Bitmapped Graphics

3 A Brief Introduction to CANalyzer

This section focuses on quickly learning the basics of CANalyzer, its tool architecture, and several of its key operational features.

To get some initial experience, we will use a configuration demo that comes with the tool.

Not too many details are presented – those things are better left for your later encounters. We just want to get up the curve a short distance at this time.

3.1 Value of the Downloadable Demo

Those who might initially experience CANalyzer by obtaining the downloadable demo version from the Internet (Available at http://www.vector-cantech.com/download) will find this brief overview helpful.

The Internet downloadable demo is a usable, working copy of the programmable version of CANalyzer. While the demo does not require hardware association and no connectivity to the real CAN bus, you can use it to perform the following operations:

- Learn the basic operation of CANalyzer
- Create your own application
- Create your own message set and define specific message data content
- Reuse all your work, configurations, and program blocks in a real project

3.2 First-Time Considerations

While you learn CANalyzer, please remember the following important guidelines:

- Do not save
- Use “cancel”
- Avoid changing the CANalyzer configuration

You should look everywhere, play with as many things you feel comfortable with, but DO NOT SAVE ANY CHANGES until you gain some more experience. Because CANalyzer uses a collection of related files, you may wish to keep these as relatively clean copies for later use.

If you have a full version of CANalyzer, running a configuration demo may require hardware connections. Connect your hardware as shown in Figure 8.

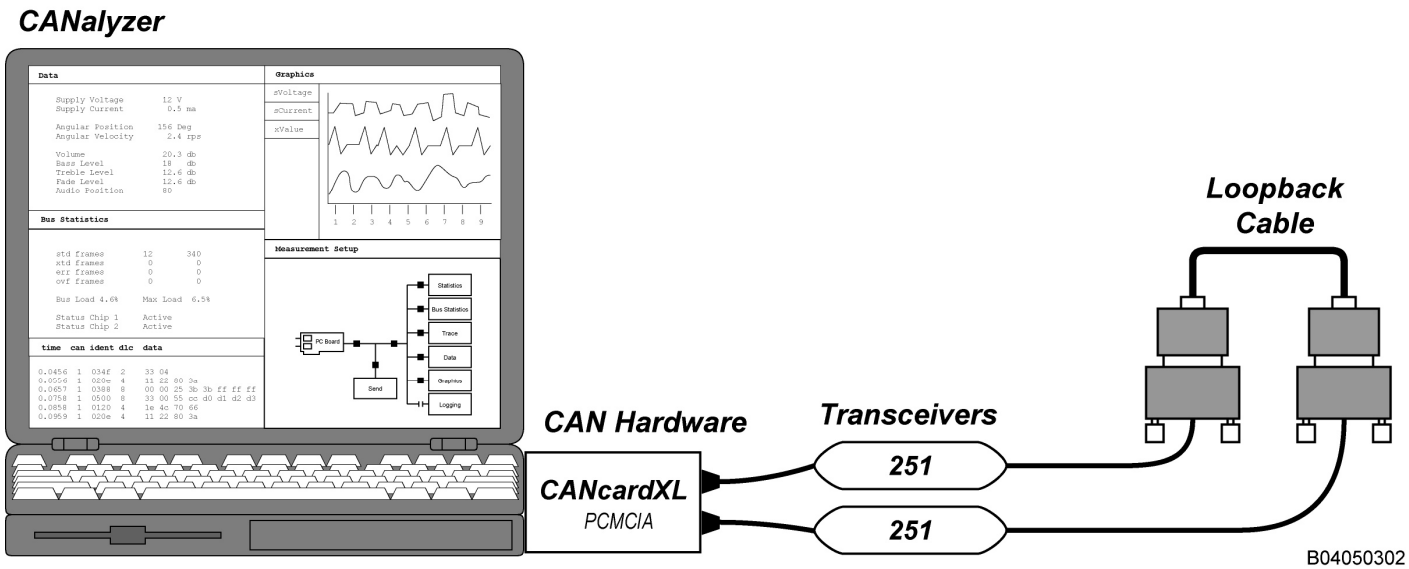

Figure 8 – CANalyzer Hardware Connections (CANcardXL Setup)

3.3 How to Start and Stop CANalyzer

Find and launch the CANalyzer application. There will be several CANalyzer configuration demo choices on the Windows' Start Menu. Start with the one called **Motbus**, a very useful demo program of a Motor Bus (a simple automotive application) and shows most of the tool's capabilities.

Once CANalyzer has filled the front screen of your computer, examine the upper toolbar to locate a pair of grouped buttons – the "lightning bolt" and the "stop sign" (see Figure 9). Then:

- Click on the "lightning bolt" to start CANalyzer
- Click on the "stop sign" to stop CANalyzer

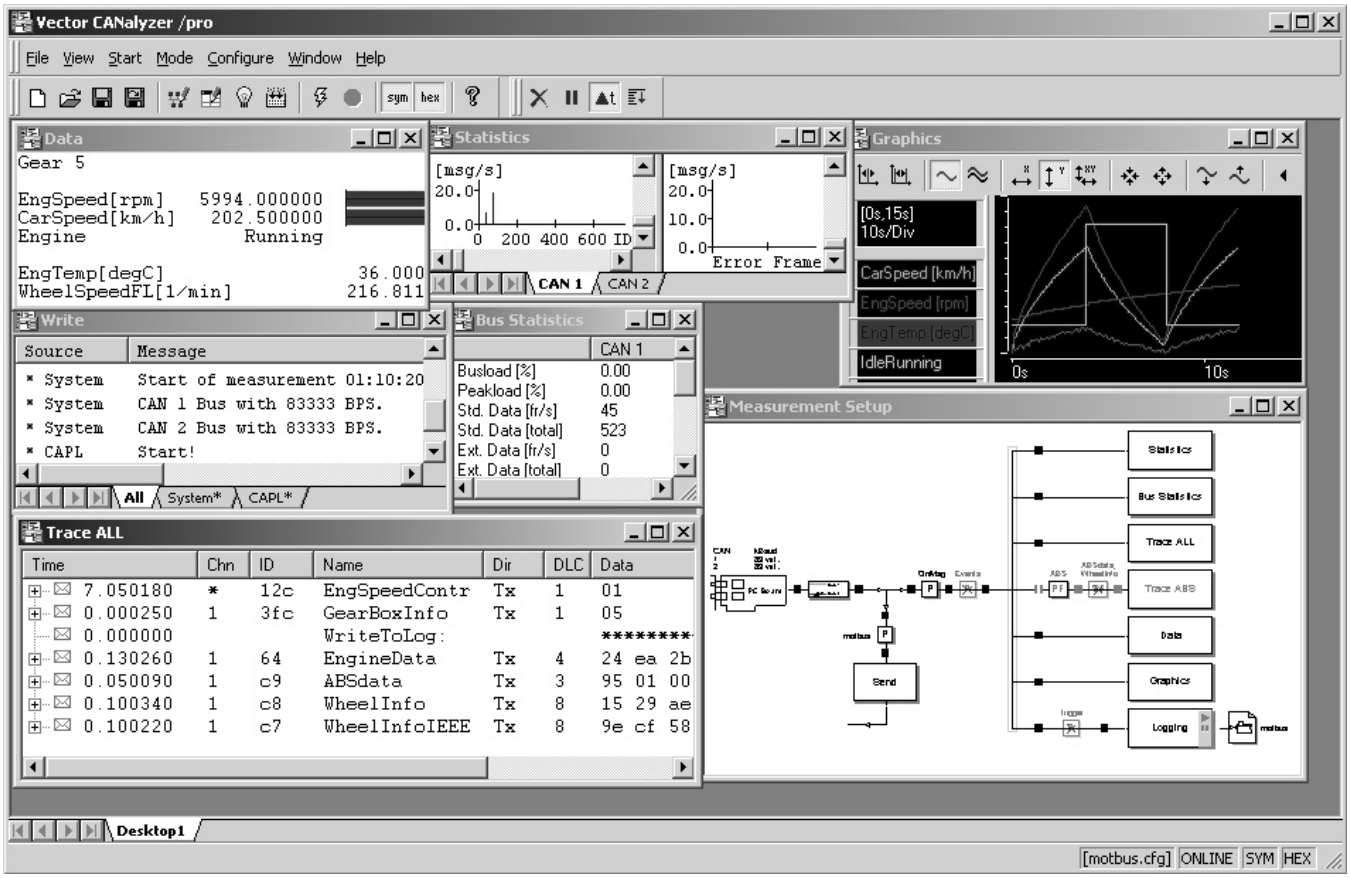

Figure 9 – The CANalyzer Motbus Interface

3.4 The Architecture of CANalyzer

The first time the "lightning bolt" is pressed to start CANalyzer, all sorts of activity begins. This gives the appearance that this tool is somewhat complicated and perhaps a bit overwhelming. But do not worry; CANalyzer is easy to use if you do two things first:

- Learn the tool architecture
- Learn to right-click the mouse button – it does a lot.

Figure 10 shows the basic tool architecture of CANalyzer. The diagram is essentially the same as CANalyzer's **Measurement Setup** window.

CANALYZER TOOL ARCHITECTURE

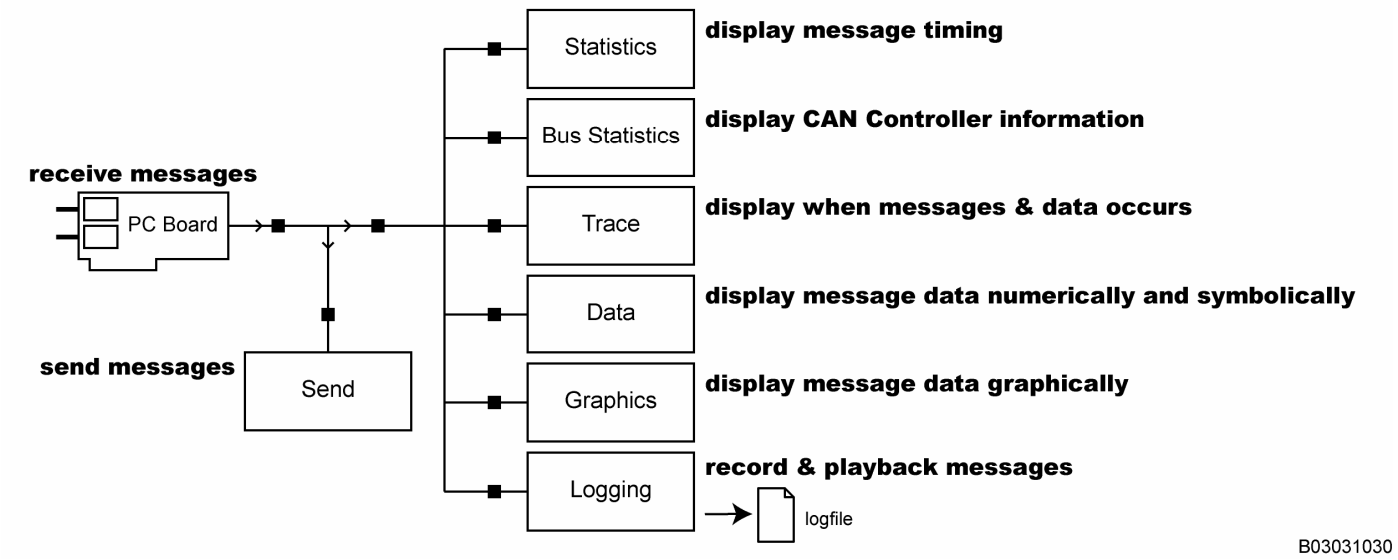

Figure 10 – Basic Tool Architecture of CANalyzer

Figure 10 (shown above) illustrates the default structure of the tool. It is completely under your control to change the configuration. In fact, you will soon be naming each configuration with a different file name so you can reuse different configurations based on your needs.

As shown in Figure 11, the organization of CANalyzer provides a comprehensive network analysis interface.

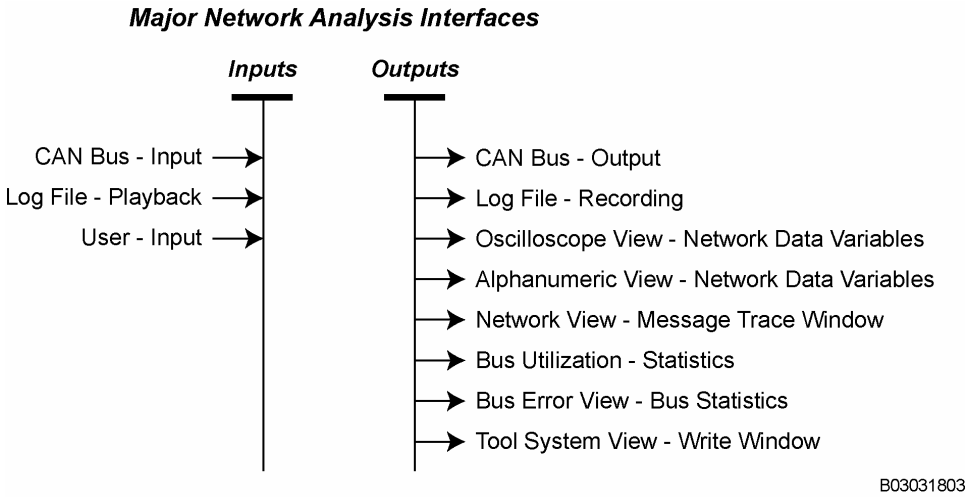

Figure 11 – CANalyzer's Network Analysis Interfaces

Not only can messages be received and transmitted on one or more buses, but recording and playback features are also provided. Recorded files at one location can be emailed for playback at a different location.

An unlimited amount of viewable network information is available, including the following forms:

- Oscillographic
- Numerical
- Alphanumeric or symbolic

3.5 Measurement Setup Window

As shown in Figure 12, the **Measurement Setup** window shows the structural process and information flow of the CANalyzer configuration. This diagram is adjustable, and the layout is saved into the CANalyzer configuration. You can create, modify, read, and save any CANalyzer configuration. Configurations are saved as .cfg files.

From a general point of view, the **Measurement Setup** window shows several blocks with interconnecting lines. Each block represents a major tool activity. Most of these activities also have an associated and separate viewable window. Double left click on the block to see the corresponding window or simply go to the **View** menu to select.

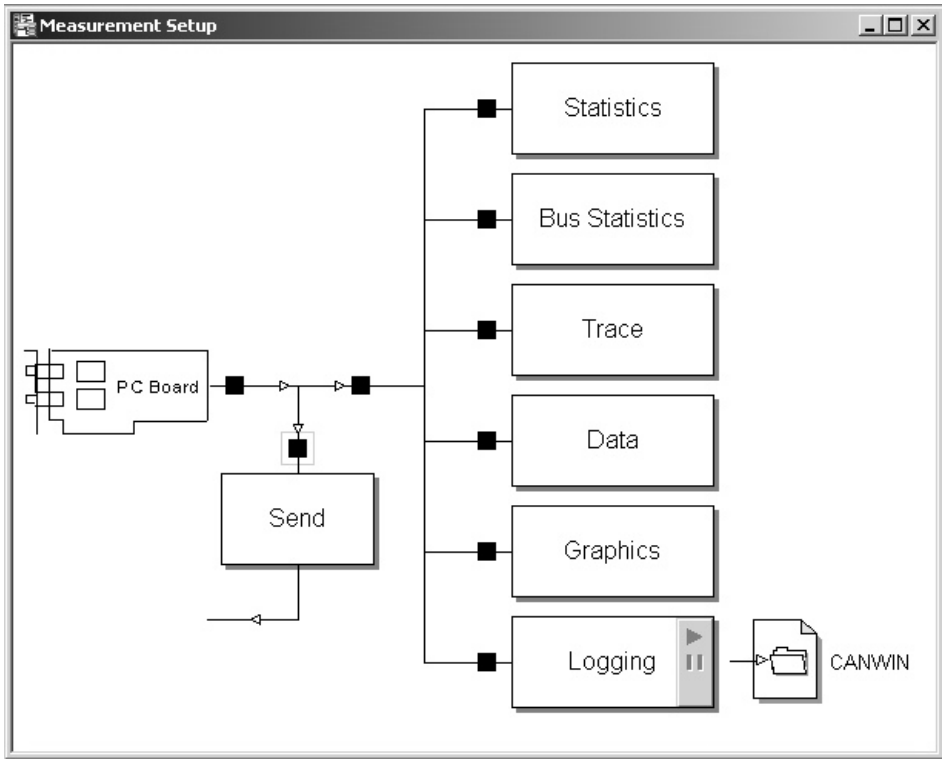

Figure 12 – CANalyzer's Measurement Setup Window

From a communication point of view, the PC Board represents the CAN hardware that handles the reception process, and the Send Block represents the transmission process. All remaining blocks - Statistics, Bus Statistics, Trace, Data, Graphics and Logging - are network analysis processes.

3.5.1 Trace Window

The **Trace** window shows the data content of the messages detected on the bus and other information about them. Each message is time-stamped, and the identifier, message type, data length, and any associated data bytes are shown. Several CANalyzer toolbar buttons are associated with this window.

Experiment with the toolbar buttons to see the effects. You may notice that identifiers can be shown either as decimal or hex or symbolic. Notice that this use of the CAN message identifier names, rather than using numbers, can be accomplished by using an associated database file. This **Motbus** demo uses a pre-defined database file that is specific for this configuration. We will soon learn more about databases in another chapter.

3.5.2 Statistics Window

The **Statistics** window shows the statistical information for individual messages. The window displays the average time between message transmissions during a measurement. It is shown using a line histogram indicated with the message rate on the Y axis and the message identifier on the X axis. If the Statistics block is configured, a statistical report can be generated at the end of the measurement.

3.5.3 Bus Statistics Window

The **Bus Statistics** window shows the statistical information for the entire network, including the total composite message rate, total number of messages, and the average and maximum bus utilization. In addition, the state of the CAN controller is shown as either active, passive, or bus off.

3.5.4 Data Window

The **Data** window shows the current content of message data and is configurable. The user selects which data items are to be shown, their respective positions in the window, and how the data content will be represented.

The **Motbus** configuration shows data in physical units and in bar graph form, but several other formats are also available. The integrated database is used to create and name data items within messages, much like the way CAN message identifiers are named, and it also allows further definition of physical units and symbolic values.

3.5.5 Graphics Window

The **Graphics** window is similar to the **Data** window. They both display message data, but instead of displaying numeric data, the **Graphics** window actually plot those numeric message data into a graph. This window is also configurable and has a wide assortment of associated toolbar controls for graph evaluation. Multiple signals can be graphed with independent time axis, signal position, signal unit, and signal color.

3.5.6 Write Window

The **Write** window is a text output window and is used to indicate major CANalyzer operational modes, such as measurement start/end times and detected hardware warning conditions. This window also allows a CAPL program to output text information during program execution. The CAPL **write()** function is used to output text to this window. Clearing the window and copying its contents to the clipboard are also possible. By using the right mouse button, the window can be further defined and configured. There may be some dependencies, based on whether CANalyzer is running or stopped.

3.6 Duplicating Analysis Blocks

Several of these analysis blocks can be duplicated.

For example, if you would like to use two separate **Graphics** windows, point to the Graphics block, hold the right mouse button, and select `Insert graphic window`. Now you have created two **Graphics** windows. You can name each block differently. Removing an inserted block is also possible by again using the appropriate right mouse selection.

3.7 Controlling Data Flow

Looking at the **Measurement Setup** window, data flows from the left (PC Board) to the right (analysis blocks). This is illustrated by the arrows on the line paths. Data flow connections to the various analysis blocks may be toggled between being connected or disconnected by double clicking on the small black squares located on the line paths, also known as hotspots.

Note: Modifying the **Measurement Setup** window is not allowed if the CANalyzer measurement is running.

3.8 Function Blocks

The **Measurement Setup** window not only displays the direction of data flow, but it also allows users to insert a variety of function blocks so as to provide more analysis and simulation options to the user. For example, suppose that you only want the Trace block to monitor certain messages. Instead of disconnecting the entire Trace block, a function block can be inserted into the **Measurement Setup** window by right-clicking on the hotspots along the line paths of the data flow plan.

Because data flow follows a certain path in the **Measurement Setup** window, placing a function block in a specific location can have varying impact. In the example shown in Figure 13 (below), a filter block is placed directly to the left

of the Trace block. As illustrated in the left image, filtering only affects the Trace block. If the same filter block is removed and placed at the location in the image on the right, filtering will affect all the analysis blocks.

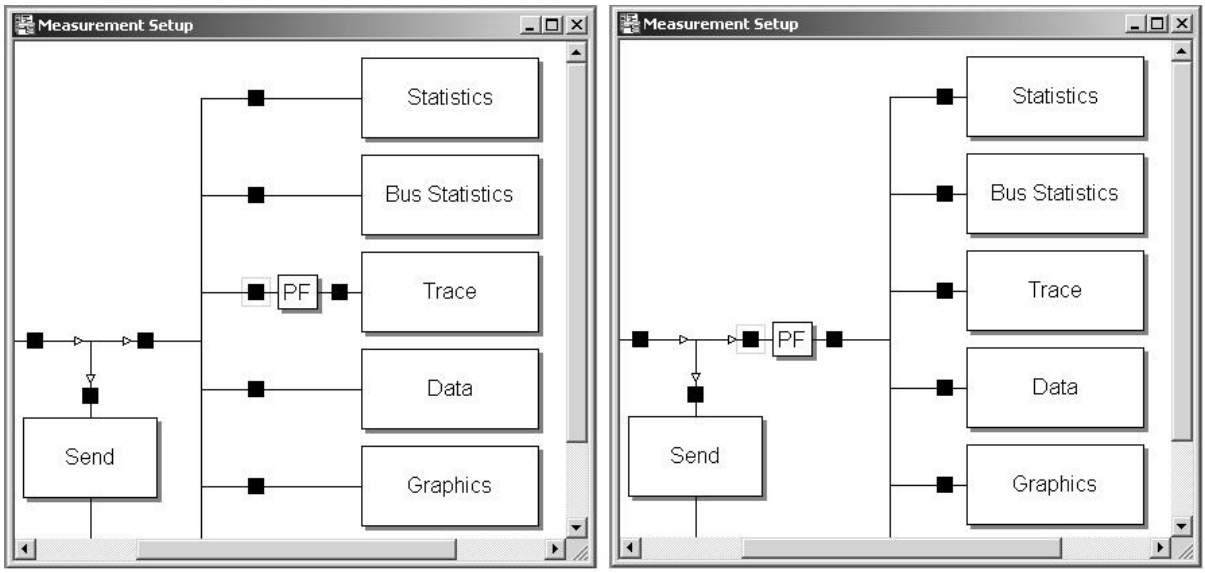

Figure 13 – Block Placement Effects

Function blocks are divided into two groups, one group for data analysis and the other for message simulation. Both groups are equally important to CANalyzer to make the tool powerful to use and user-friendly.

3.8.1 Filtering Blocks

There are many levels of filtering possible along the data flow plan of the **Measurement Setup** window. The highest level is filtering on specific CAN channels. A CAN channel is normally connected to a single network node or the entire network. When blocking a CAN channel using a Channel Filter, as shown in Figure 14, there are two types of information that is ignored. The first type are the messages from that CAN channel. The second type is the statistical data reported by the micro-controller inside the Vector interface card (e.g., CANcardXL, CANboardXL, and CANcaseXL). This statistical information includes busload percentage, message frames per second on that channel, error frames, CAN controller state, etc.

Note: All messages received on a blocked channel using the Channel Filter will still be acknowledged by that channel's CAN controller in the Vector interface card.

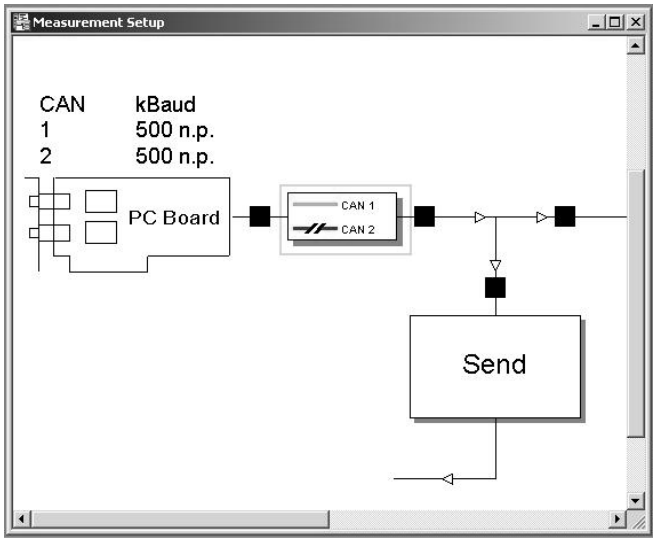

Figure 14 – Channel Filter

If the channel filtering approach is too broad, and filtering has to be done at the message level, you have need of the Filter block. The Filter block is used to filter messages for transmission and reception, depending on where the block is inserted. But, before you start inserting messages into the block, you need to separate the messages into two groups: messages to be blocked and messages to be passed. The **Filtertype** option is selected via radio buttons at the bottom left of the Filter configuration dialog (double click on the block to open). With a Pass Filter (**PF**), as illustrated in Figure 13, the user can specify the list of messages to transmit or receive, while the Stop Filter (**SF)** stops the message list from transmitting or receiving. Both blocks are used to reduce data traffic volume as well as unwanted messages.

In addition to entering a message identifier to filter or select a defined message from a database, the Filter block has also provided many conditions for filtering a message. You can filter a message based on its state:

- Transmit
- Receive
- Transmit Request

If that is not enough, you can filter upon the CAN channel the message belong and the type of message:

- Error frame
- Data frame
- Remote frame

All of the above message characteristics can be defined, and you can use more than one Filter block if needed.

The last level of filtering is available to the user that finds message filtering based on message characteristics too generic. Often, a user must base filtration on signal values or data byte values from the data field of a message frame. The concept is not that signals are being filtered; signals are used as keys to filter messages. For example, you have a **Trace** window monitoring heavy bus traffic, and you are unable to observe a particular signal value because of the complexity of message traffic. This scenario is very common, and the best solution is to use a Trigger block (**T**) . The Trigger block allows filtering based on signal values. The best aspect of using the Trigger block is you do not have to manually pause or stop the **Trace** window from updating once the desire condition has occurred.

The Trigger block has a built-in timer that allows messages to be displayed in the **Trace** window within a time interval. Messages received outside the specified time interval will not be shown. The same idea is also used for logging bus traffic into a file. You need to know when to start logging, and when to stop logging based a message or signal condition.

3.8.2 Message Blocks

CANalyzer contains four function blocks that are capable of transmitting messages:

- Generator block
- Interactive Generator block
- Replay block
- CAPL Program block

Each of the above blocks is specifically suited to a particular method for message transmission. For example, the Generator block is ideal for simple transmission activities. Complex transmission activities, requiring signal changes or complex transmission strategies, are better served by the use of a CAPL program.

3.8.2.1 Generator Block

The oldest method for message transmission in CANalyzer is to use the Generator block (**G**). The Generator block (Figure 15) allows you to define messages to be transmitted, or select the messages from a database. You can also choose the CAN channel on which to transmit each message, and set the data byte values that each message carries.

In order for the Generator block to transmit messages, triggers must be defined. Transmission trigger methods may involve a keystroke, a periodic rate, and/or upon receiving a specific message.

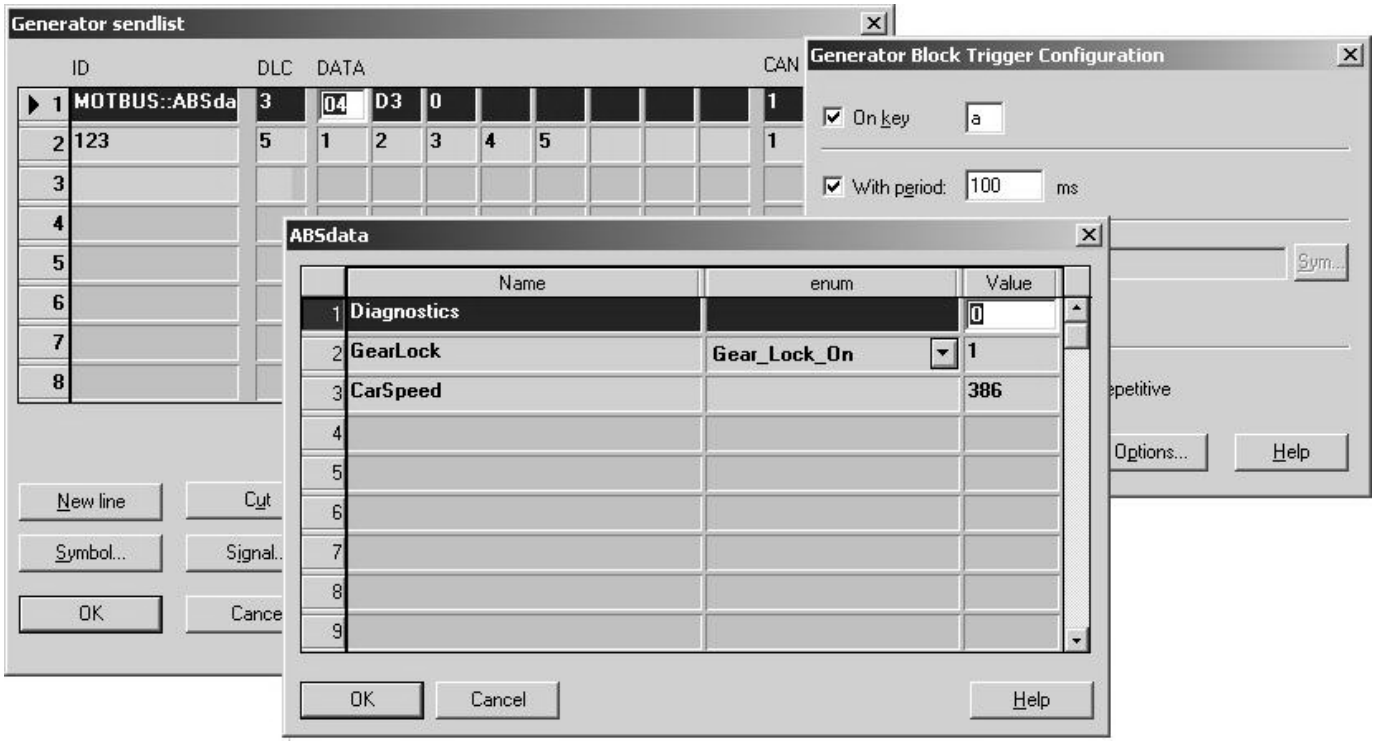

Figure 15 – Generator Block Setup

3.8.2.2 Interactive Generator Block

There are two limitations to the Generator block that limit its effectiveness in complex tasks. The block is misleading for some people because it requires multiple windows for setting up the transmit message list. The second problem is the block settings have to be set before the CANalyzer measurement starts. No changes can be made if the measurement is running.

Fortunately, CANalyzer has another transmission block that eliminates both practical limitations: the Interactive Generator block (**IG**). The **IG** block combines the configuration windows of the Generator block into one window; therefore, everything can be setup in one spot. In addition, changes can be made with the **IG** block while the CANalyzer measurement is running.

Note: You must be very careful when using the **IG** block. Any mistakes, such as an accidental keystroke or mouse-click, may impact the physical CAN bus that CANalyzer is connected to.

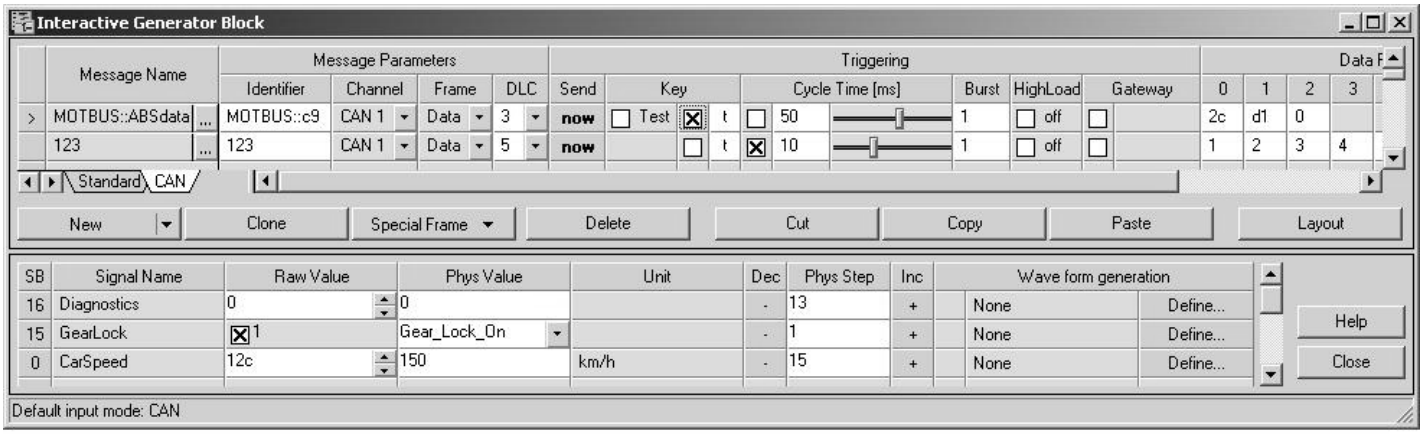

Figure 16 – Interactive Generator Block Setup

3.8.2.3 Replay Block

The third function block that is capable of transmitting message onto the CAN bus is the Replay block (**R**) . The Replay block offers the ability to replay onto the CAN bus or analysis windows a sequence of messages that have been recorded by the Logging block. There are many settings in the Replay block for message transmission, but among the most useful is to allow a CAPL program to have full control of the block activities. Inside a CAPL program, you can start, stop, pause, and resume a Replay block from replaying.

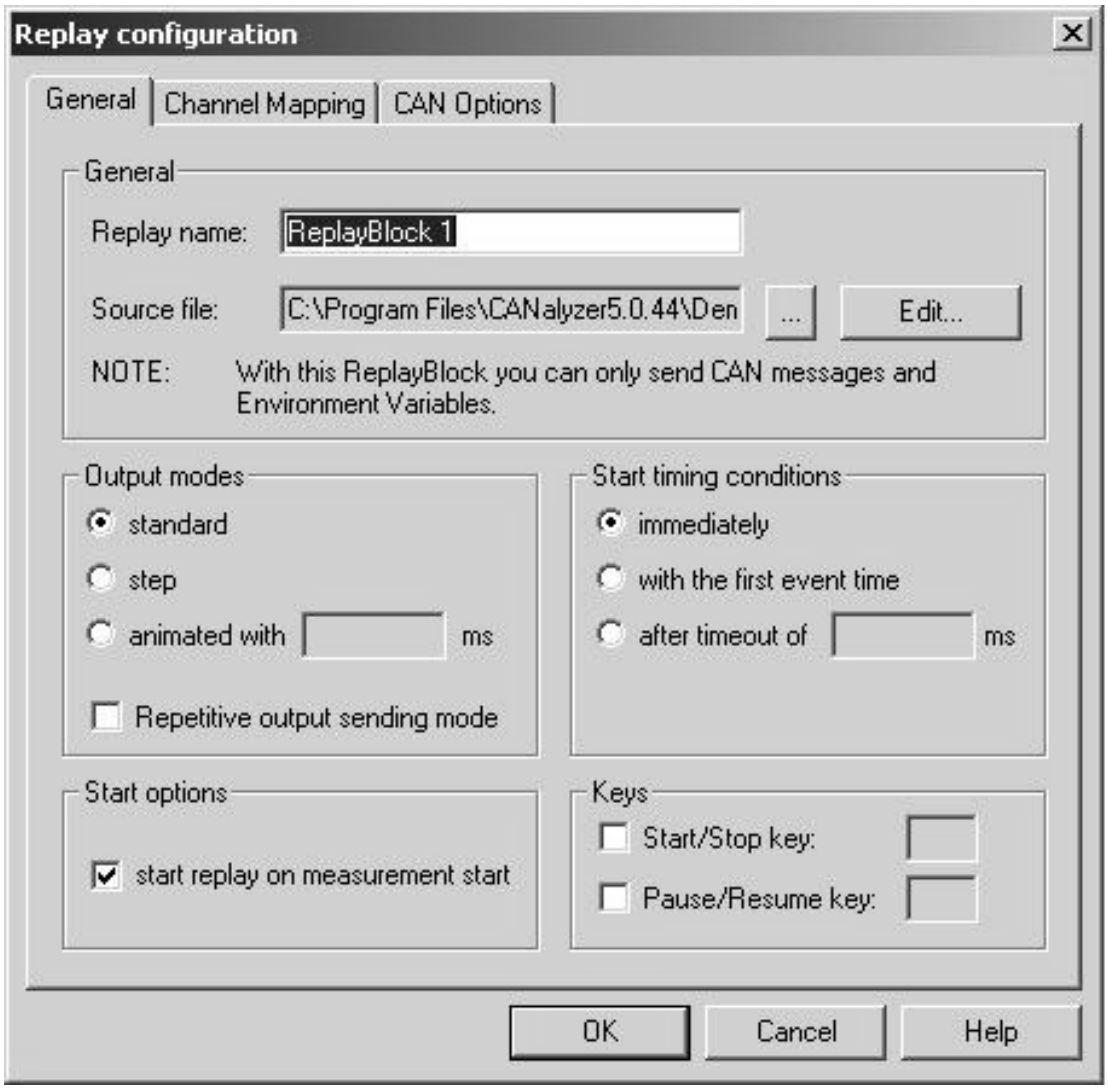

Figure 17 – Replay Block Setup

3.8.2.4 CAPL Program Block

If a CAPL program has the ability to control a Replay block, it certainly can transmit messages, receive messages, and filter messages at all levels. In CANalyzer there is a function block called a CAPL Node or Program block (**P**) that is used to associate a CAPL program to simulate node behavior or perform complex data evaluation. As mentioned before, a CAPL program is developed, edited, compiled, and saved using the CAPL Browser.

3.9 Using CAPL Program Blocks in CANalyzer

Within CANalyzer, CAPL program insertion can be placed into two major application areas – transmission and analysis - as shown in Figure 18.

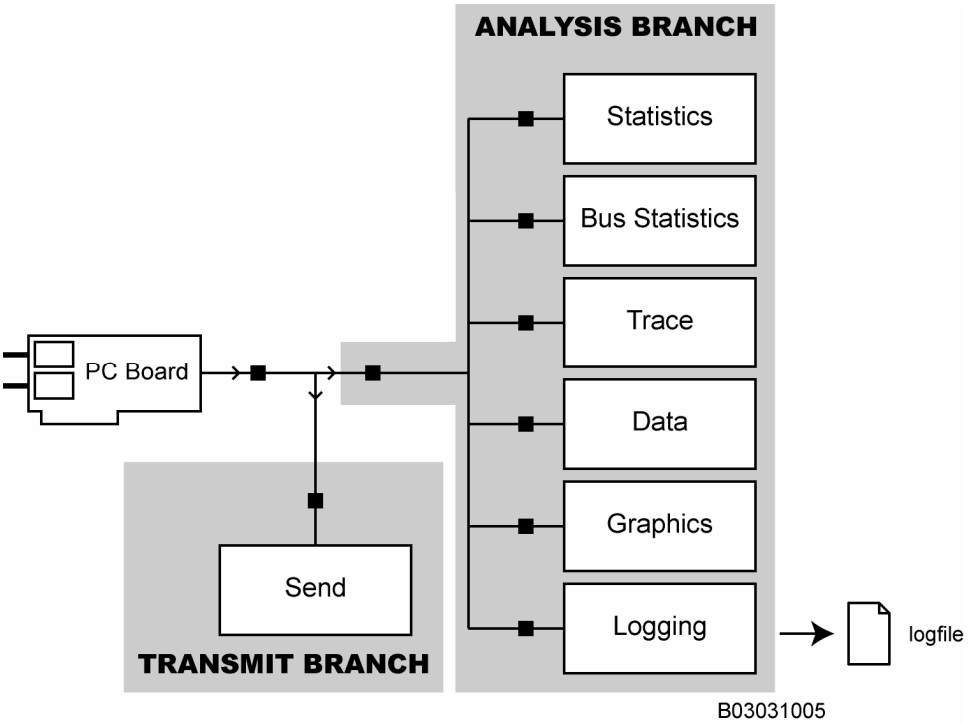

Figure 18 – CAPL Program Location Areas in CANalyzer

A CAPL program is used whenever an application requires any programmable process. It is the specific application that influences where a particular **P** block is inserted.

3.9.1 CANalyzer Program Blocks for Transmission

Insertion of a CAPL program block, or **P** block, into the CANalyzer's Transmit Branch, as shown in Figure 19, can provide the following capabilities:

- Simulation of a network module
- Programmable message generation
- Operation as a network or module tester

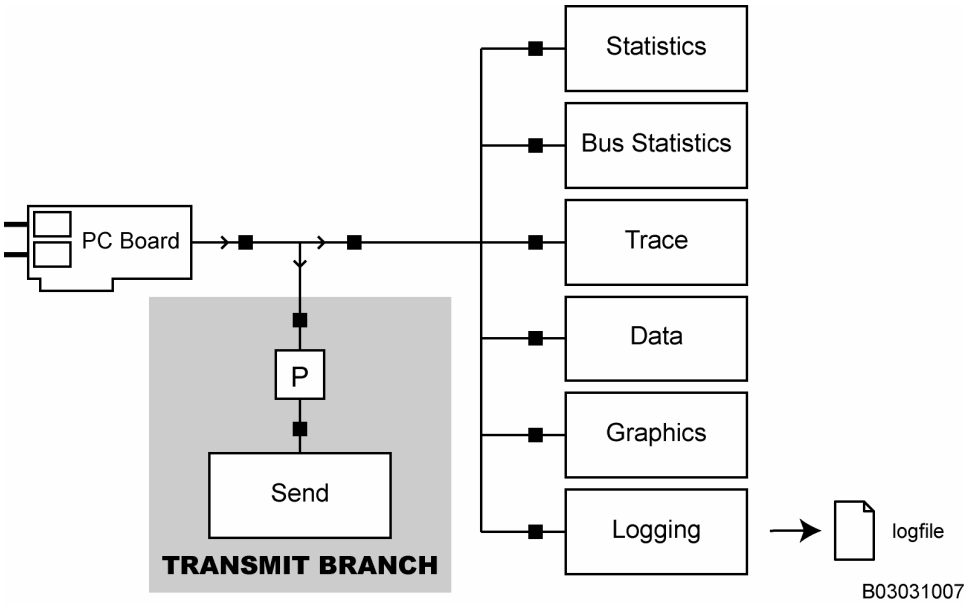

Figure 19 – Inserting a CAPL Program into CANalyzer's Transmit Branch

3.9.2 Program Blocks for Analysis

Depending upon the analysis or measurement requirements, CAPL program blocks may be inserted into the Analysis Branch. CAPL program blocks can be placed either in front of the entire Analysis Branch (so as to affect all analysis blocks, as shown in Figure 20), or in front of any specific analysis block.

Note: Message transmissions done in the Analysis Branch will not go onto the physical CAN bus.

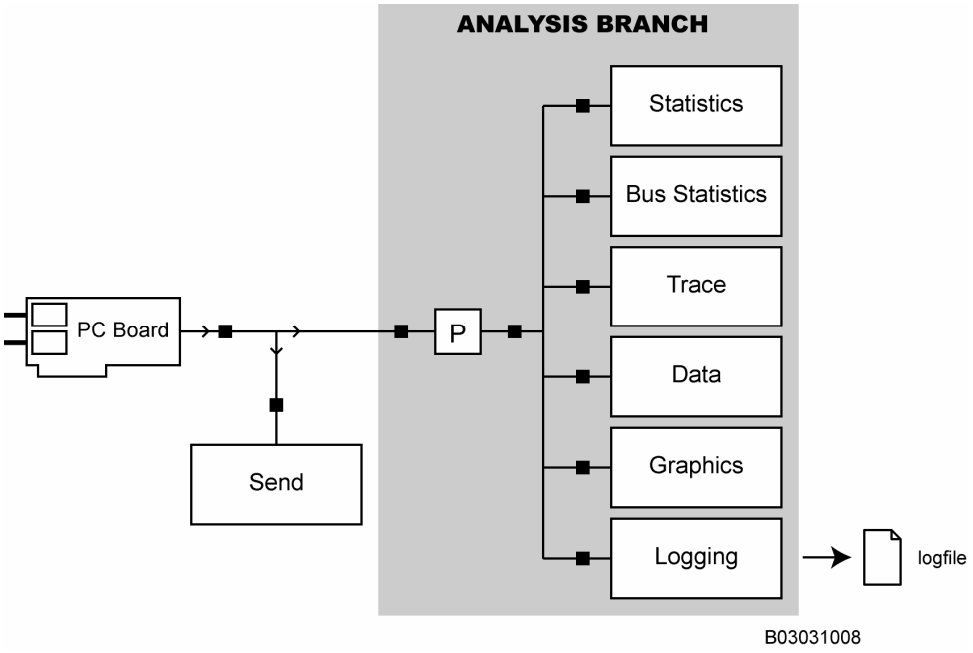

Figure 20 – Inserting the CAPL Program into CANalyzer for all Analysis Blocks

In another example, if a developer is interested in making specific numerical computations before logging information into a file, the appropriate location for the CAPL program block is immediately in front of the logging activity, as shown in Figure 21.

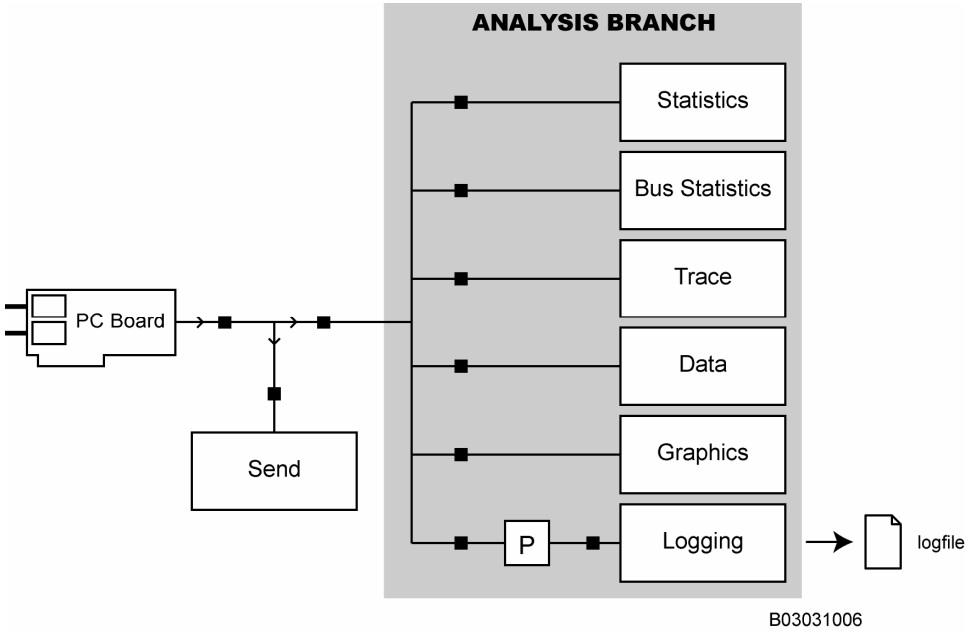

Figure 21 – Inserting a CAPL Program into a Specific Analysis Branch

3.10 CANalyzer – CAPL Programming Environment

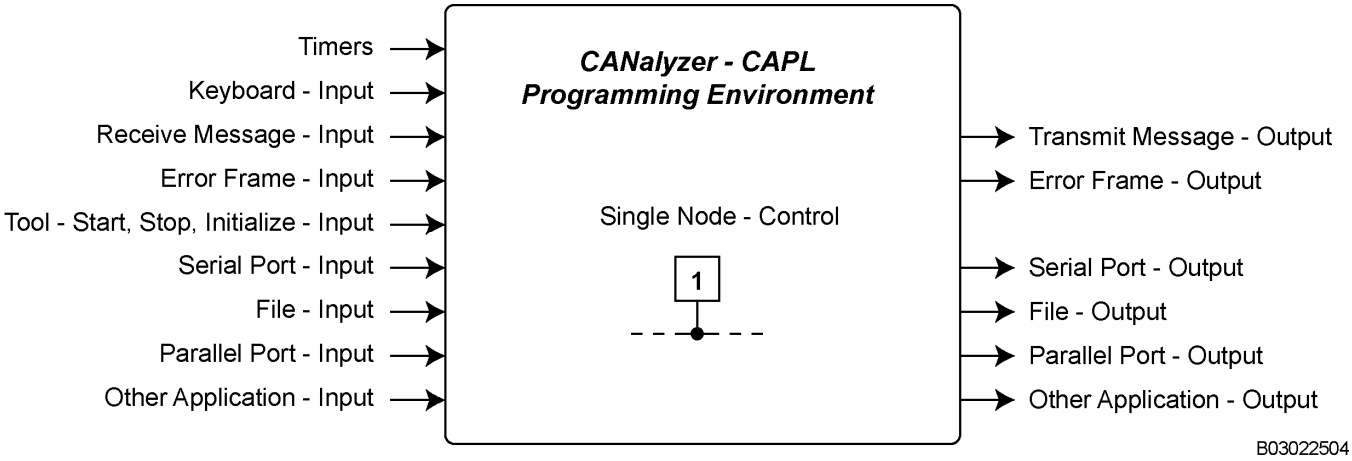

Figure 22 – CANalyzer – CAPL Programming Environment

The CAPL programming environment, shown in Figure 22, provides a wide range of inputs and outputs.

Remember that one big difference between CANalyzer and CANoe is the number of modules that can be controlled. While a single CANalyzer tool can act as a single network member, CANoe has no limit on the number of modules on the network.

4 A Brief Introduction to CANoe

This chapter focuses on quickly learning the basics of CANoe, its tool architecture, how it can be used to create a system-level model, and several key operational features. Few details are presented in this chapter, but will be introduced in later chapters. For the purpose of this chapter we will only introduce a few concepts

To get some initial experience, we will use a demo application that comes with the tool.

Because a portion of the network analysis interface is common between CANalyzer and CANoe, some material in this chapter is partially repeated from the last chapter.

4.1 Value of the Downloadable Demo

Those who may initially experience CANoe by obtaining the downloadable demo version from the Internet (available at http://www.vector-cantech.com/download) will find this brief overview helpful.

The downloadable demo is a usable, working copy of the programmable version of CANoe. While the demo does not run on a real CAN bus or work with real CAN hardware, it can provide experience for you to perform the following tasks:

- To learn the basic operation of CANoe
- To create your own multiple-node application
- To create your own message set and to define specific message data content
- To reuse your work, configurations, and program blocks in a real project

4.2 First-Time Considerations

While you begin to learn CANoe using the demo, please remember the following guidelines:

- Do not save
- Use “cancel”
- Avoid changing the CANoe configuration

You should look everywhere, try out as many functions and features as you feel comfortable with, but DO NOT SAVE ANY CHANGES until you gain some more experience. CANoe uses a collection of related files; therefore, you may wish to keep these as relatively clean copies for later use.

If you have a full version of CANoe, running a demo may require hardware connections. Connect your hardware as illustrated in Figure 18.

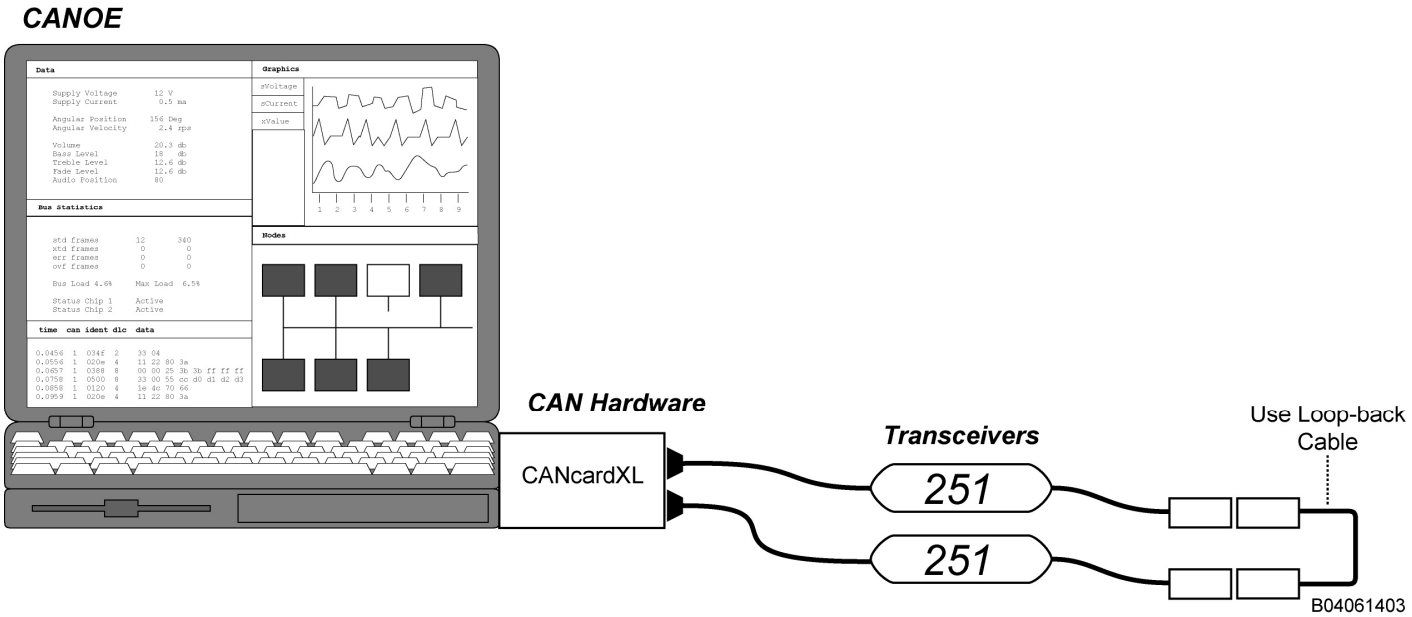

Figure 23 – CANoe Hardware Connections

4.3 Learning from the Demo

Now we are ready to launch the CANoe application. The program you are about to launch is a demonstration of a simple application and shows many of the tool's capabilities. Follow these easy steps to begin:

1. Locate in the Window's Start Menu
2. Start the demo entitled **CANoe Easy**.

CANoe begins by filling the front screen of your computer with a large collection of windows. All of the windows should look familiar, because they are the same as CANalyzer windows, with the exception of two windows: the **Simulation Setup** and the **Measurement Setup** windows. Since CANoe supports multiple node simulation, therefore, the **Measurement Setup** window in CANalyzer has been split into two separate windows in CANoe to explain the CAN network system approach. The **Measurement Setup** window in CANoe is used for analysis only, and the **Simulation Setup** window covers the Transmit Branch of CANalyzer. With the new **Simulation Setup** window in CANoe, the network topology has been changed to allow every simulated node to have direct influence on the simulated CAN bus.

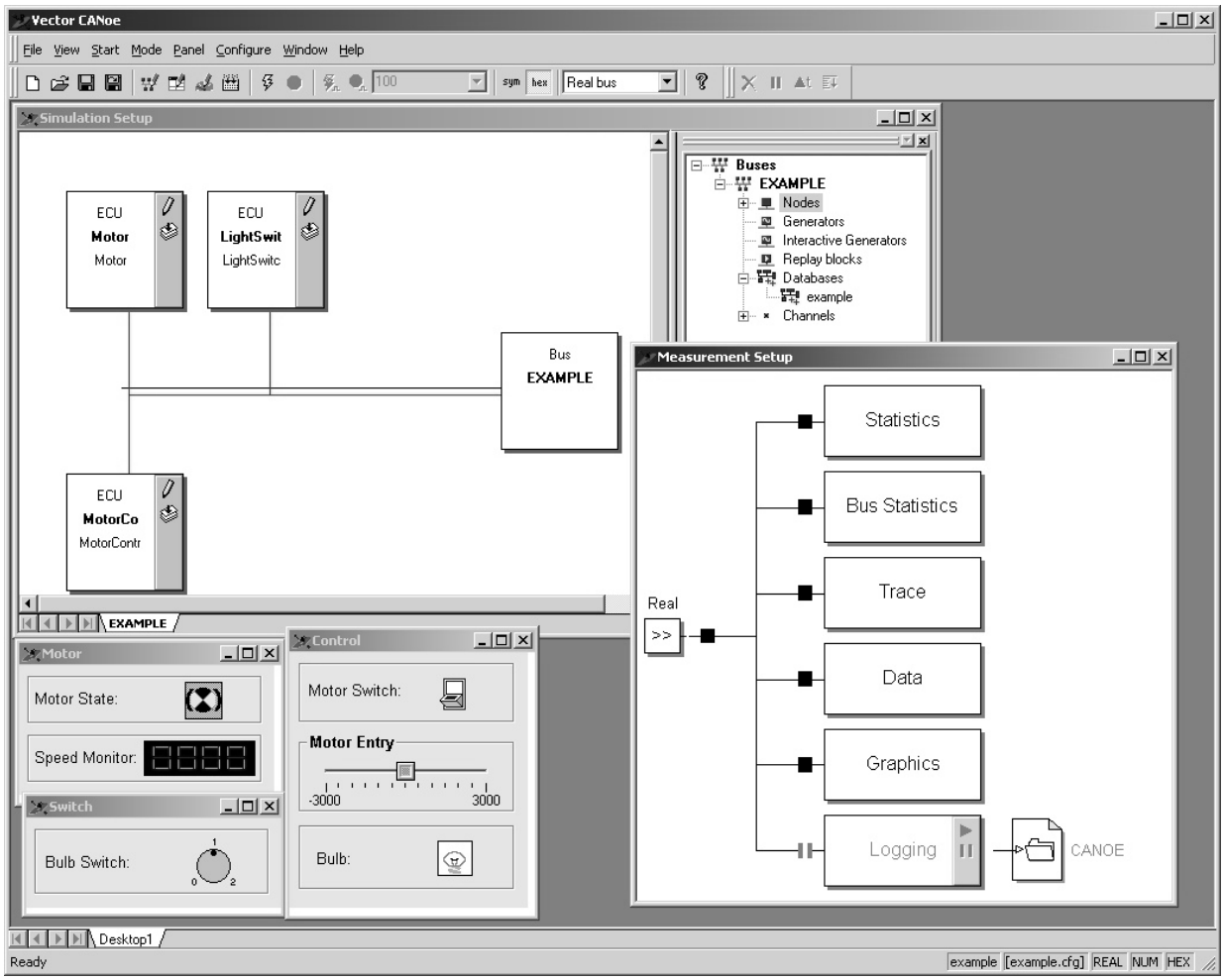

Figure 24 – Demo Configuration - CANoe Easy

As you look at the CANoe Easy demo, you can see three nodes defined in the **Simulation Setup** window: **Motor**, **LightSwitch**, and **MotorControlandBulb**. Each node can represent a module or an ECU function. A user-defined graphical interface is made for each node's function. These interfaces are called panels, which usually represent the inputs and outputs of an individual node. In the demo, if you want to turn on the motor, you can click on the Motor Switch on the Control panel and then set the motor speed. The usefulness of the panels is for you to click on an I/O element and have CANoe perform a specific task, such as sending a message onto the bus which is done by environment variable associations. Each I/O element on the panel is associated with an environment variable defined in a database. When you click or type in a new value on the I/O object, the corresponding environment variable event procedure executes in the CAPL program. You will learn more about panels and environment variables in subsequent chapters.

To run CANoe, look at the upper toolbar and find a pair of grouped buttons – the “lightning bolt” (in yellow) and the “stop sign” (deactivated at the moment). Follow the steps below:

1. Click on the “lightning bolt” to start CANoe.
2. Click on the “stop sign” to stop CANoe.

When you start CANoe, you will see two messages sent in the **Trace** window: **MotorState** and **LightState**. A message in the **Write** window tells you to press ‘1’ or ‘0’ any time on the keyboard to switch between more or less information to show in the window. These key presses are only effective if CANoe is running, and if they are programmed in one of the three CAPL programs (network nodes). If you are looking for a specific window, click on **View** from the main menu to navigate to the desired window.

4.4 Tool Architecture of CANoe

Figure 25 shows the basic tool architecture of CANoe. The right portion of the diagram is similar to the architectural model for CANalyzer (**Measurement Setup** window). Just as in CANalyzer, the CANoe configuration is also adjustable and completely under your control to be modified. In addition to analysis, CANoe also provides a separate window for simulation activities. You also have full control over the simulation window (**Simulation Setup** window).

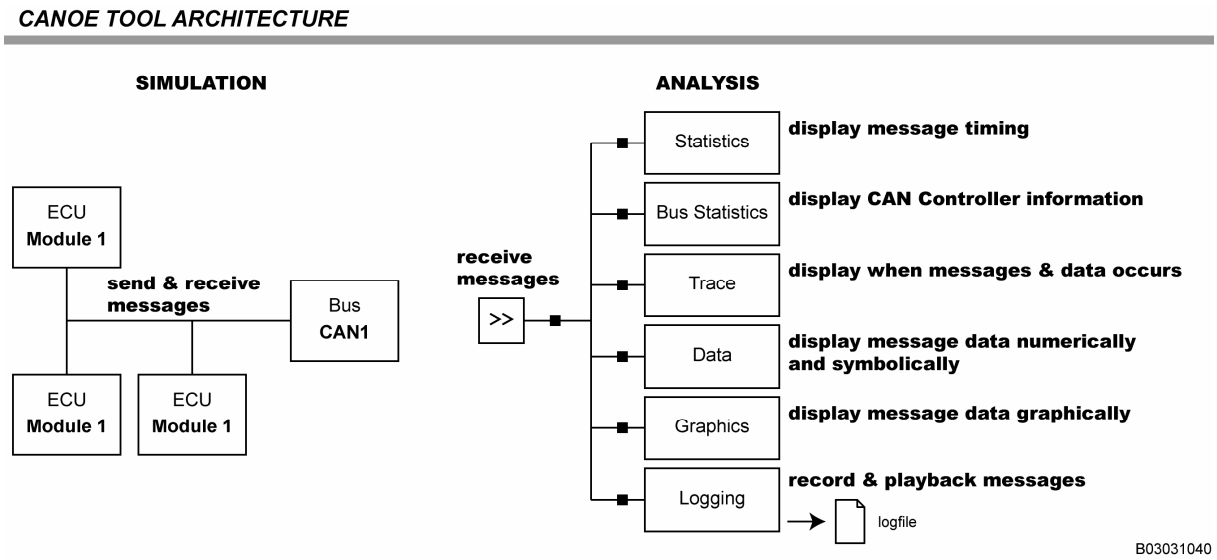

Figure 25 – Basic Tool Architecture of CANoe

The organization of CANoe includes a comprehensive network analysis interface, as shown in Figure 26.

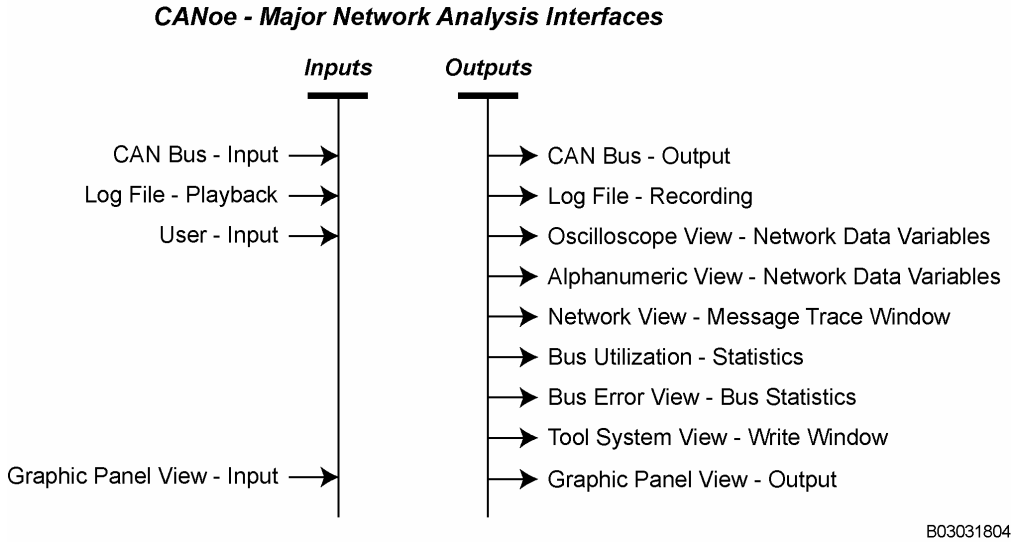

Figure 26 – CANoe’s Tool Interfaces

Just as CANalyzer, an unlimited amount of viewable network information is available in CANoe, including in the following forms:

- Oscillographic
- Numerical
- Alphanumeric or symbolic

4.5 Using CAPL Program Blocks in CANoe

For CANoe, CAPL program placement can target two specific application areas: one for simulation and the other for analysis.

4.5.1 CANoe Program Blocks for Simulation

Any number of nodes may be simulated in a CANoe environment. Each simulated node has its own corresponding CAPL program. Dividing a system into individual nodes may provide the additional advantage of reusability for subsequent system developments.

In addition, CANoe supports multiple CAN network system simulations. If we are using CANoe on a vehicle that requires four different CAN networks with multiple nodes in each network, for example, CANoe is capable of simulating all four CAN networks at the same time. If node simulations are not required, CANoe can be used as a gateway for sharing data from different networks and can perform cross-network calculations such as verifying data throughput and latency.

As shown in Figure 27, CANoe's **Simulation Setup** window shows each user-defined Transmit Branch to form the Simulation Branch.

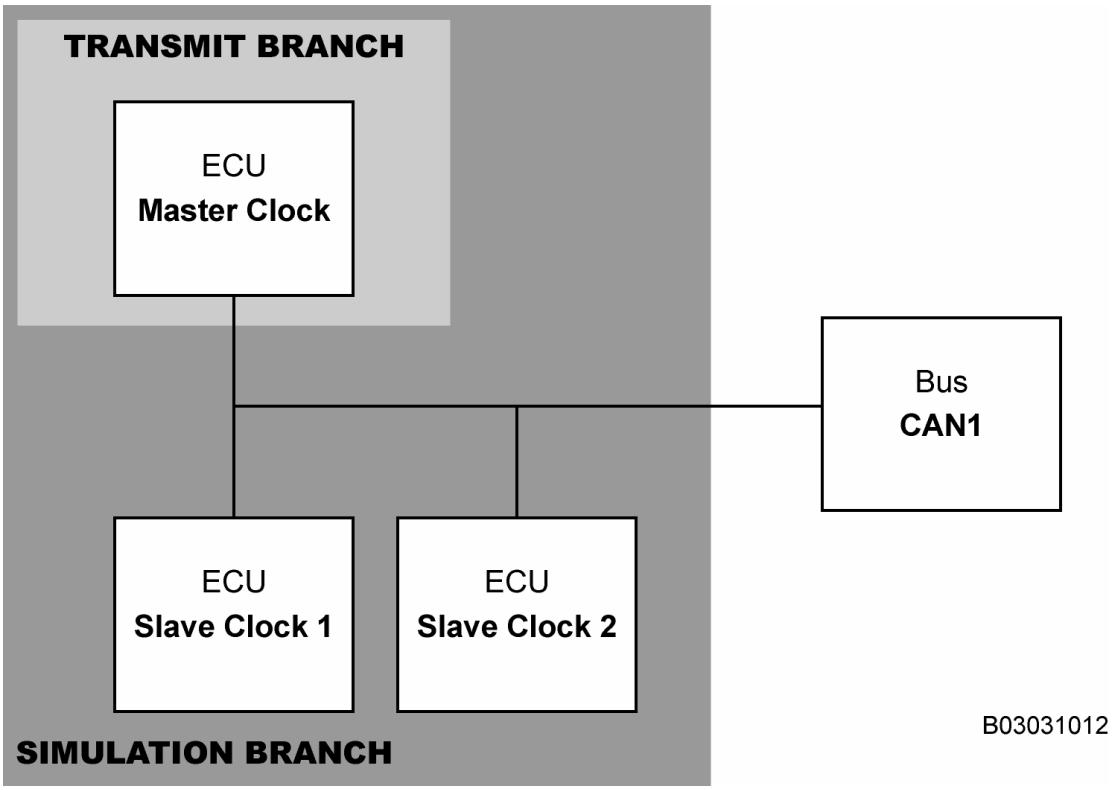

Figure 27 – Inserting a CAPL Program into the CANoe Simulation

4.5.2 CANoe Program Blocks for Analysis

Because CANoe's use of CAPL program blocks is essentially identical to the use of CAPL blocks in CANalyzer, they may be inserted into the Analysis Branch of CANoe. As with CANalyzer, they can be placed either in front of the entire Analysis Branch, as shown in Figure 28, or in front of any specific measurement activity if that activity is affected.

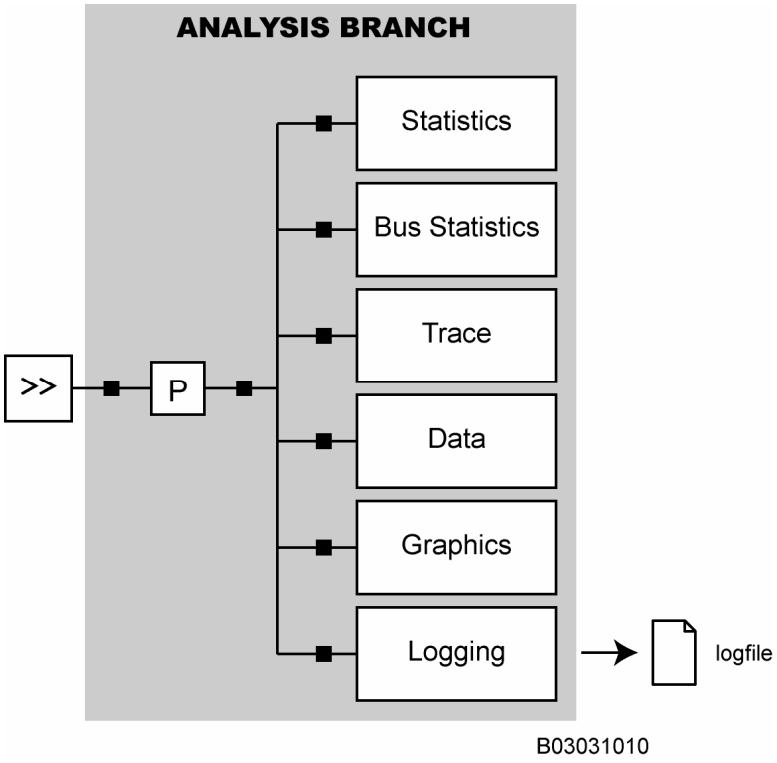

Figure 28 – Inserting a CAPL Program into CANoe's Analysis Branch

4.6 CANoe – CAPL Programming Environment

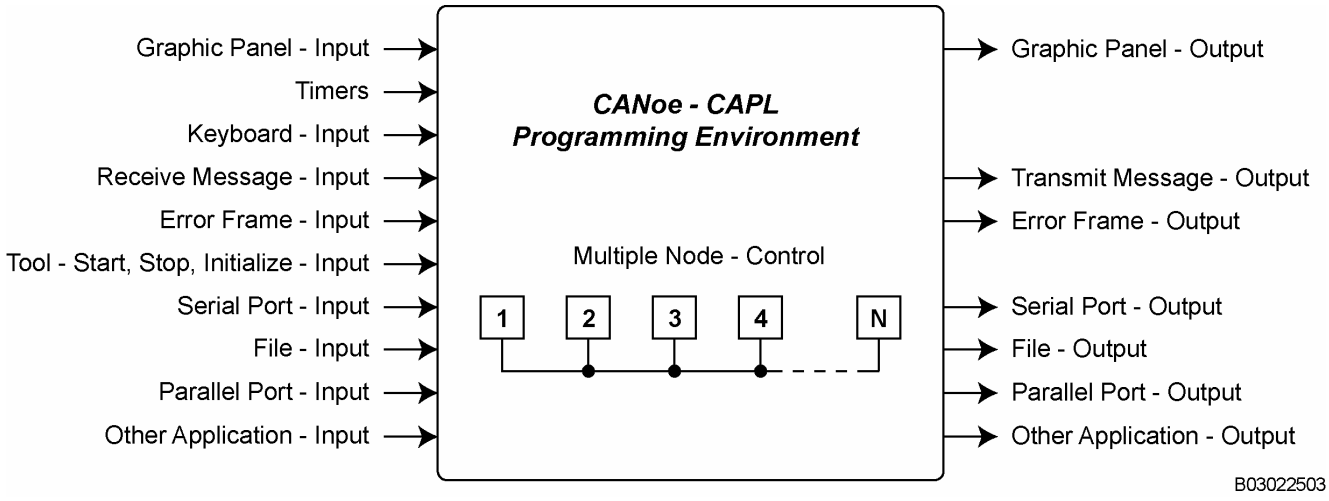

Figure 29 – CANoe – CAPL Programming Environment

The CAPL programming environment, as shown in Figure 29, provides a wide range of inputs and outputs. CANoe has no limit on the number of nodes that can be simulated or the number of interconnected networks. It is only limited by the PC hardware interface.

5 Using Databases with CAPL

What is a database? It is a look-up table that allows decoding of the bus-oriented raw data into symbolic interpretations. In other words, the database is much like a blueprint that categorizes bus data in symbolic terms. The **CANdb++ Editor** allows you to define and modify such a database so as to symbolically represent your CAN bus.

Within the CAPL programming environment, the developer has the ability to access user-defined information in a database. Databases may be used in both CANalyzer and CANoe for network measurements or custom applications.

Either database editor, CANdb or CANdb++, can be used to create a suitable network or application-specific database. To learn more about the database editor, see Using CANdb++ to Create a Database – Chapter 22.

5.1 Why Use a Database with CAPL?

The use of one or more databases to write CAPL programs is highly recommended. Having a database gives you the following advantages:

- Quicker CAPL program development
- Ability to share the database among all personnel involved in all project phases
- High level of data consistency – everyone uses the same names for variables and messages
- Elimination of case sensitivity, spelling, and memory issues
- Easier interpretation of bus data

Distribution and reusability is also a major advantage when a database is used, since many specifications are contained in one easily to manage database. Many companies that use the database platform made for CANalyzer and CANoe use system design groups to develop their database files. These database files are then made available corporate-wide, greatly increasing the compatibility and functionality of their CAN systems. Database files (*.dbc) can also be output as a document or sent by email, and are, therefore, easily distributed globally if desired.

Once associated with a database, the CAPL program can access the information defined within the database. This eliminates the burden of defining messages in the CAPL program itself.

Note: The advantage of using a database is so great that not to use this feature renders a major portion of using CAPL useless.

5.1.1 Additional Uses of the Database

The database also makes it possible to have input and output functionality in the graphical interfaces called panels. Panels are user-configurable interfaces that allow the user to make inputs and display outputs. See Using the Panel Editor to Create Panels- Chapter 23 for more information.

In addition to CANalyzer and CANoe, other Vector tools such as CANape, CANscope, and CANstress use a database as a reference to bus data interrelationships. These three tools may not come with a CANdb++ Editor, but they allow database association to simplify their functions.

5.2 Database Association with CAPL

To associate (assign) a database, use the following procedure:

1. Pull down the **File** menu and select **Associate Database** from the CAPL Browser.
2. Press the **Add** button and select the file containing the database.
3. Open the file and then press the **OK** button to return to the CAPL Browser. The database must be selected every time the CAPL Browser opens the CAPL program.

Note: These steps ONLY apply when the CAPL Browser is opened as a standalone program and not through CANalyzer or CANoe.

If the CAPL Browser is opened through CANalyzer or CANoe and a database(s) has already been assigned to the configuration before it is opened, the CAPL Browser is already associated with the database(s). Depending upon the CAPL Browser version, the CAPL Browser does not need to be reopened after a database(s) is modified and saved. The CAPL Browser will reflect the new changes made in the database(s).

5.3 Database Objects Accessible by CAPL

An object in a database is used to symbolically represent a type of data, or data values within a network. The behavior of some object types, such as attributes and environment variables, are not dependent upon other object types. Others object types have interrelationships with each other. Once a database is associated to the CAPL Browser, all objects accessible and their interrelationships can be exploited.

The database basically follows the architecture shown below in **– Database Organization** Figure 30.

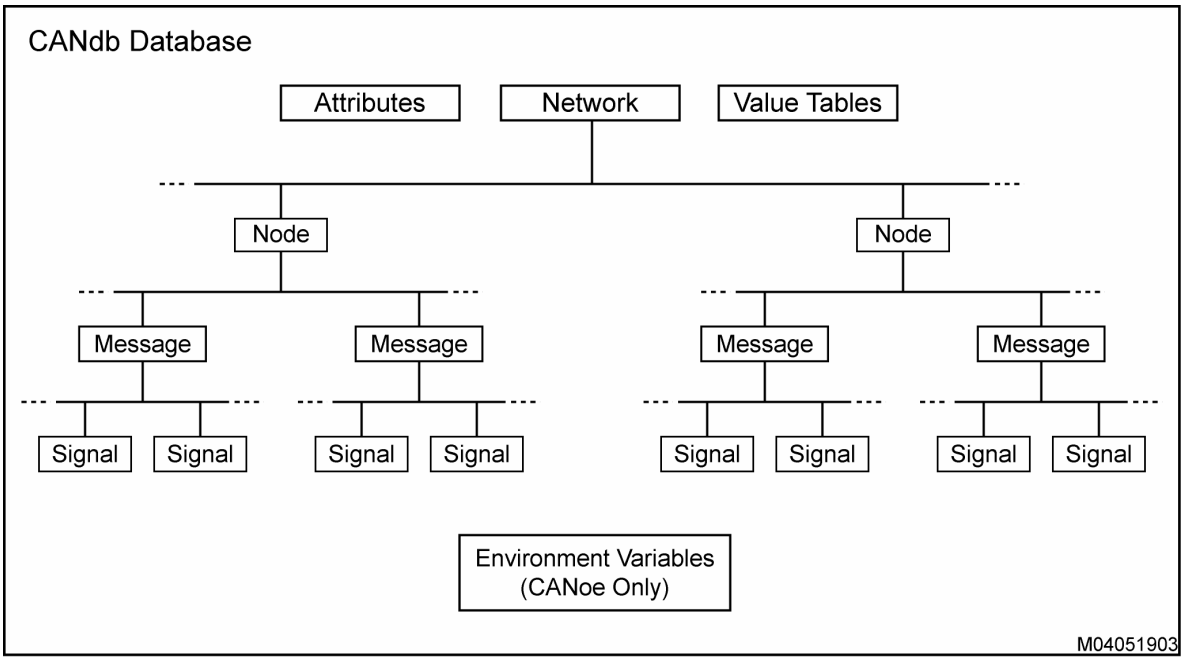

Figure 30 – Database Organization

Database Objects	
Object Type	**Description**
Networks	A CAN bus network that contains multiple ECUs to exchange data
ECUs	Distributed processing units within a network (for example, Traction Control, ABS, Adaptive Cruise Control)
Nodes	Network interfaces of an ECU used to transmit and receive data from the CAN bus
Messages	Containers of information that are transmitted on the CAN bus
Signals	Independent data regions in the data field of a message
Environment Variables	Input and output variables of network nodes, such as switch position, sensor signals and actuator signals

Table 1 – Objects in a Database

5.3.1 Attributes

An attribute is a distinct characteristic of an object (network, node, message, signal, environment variable, and so on) in the database. Most of the time, attributes are used to represent a relationship between each object of the same type. For example, you can define a message attribute to store the message periodic rate for transmission. By defining this attribute for messages, every message in that database can be configurable to have an initial cycle time value (Figure 58) by association with a particular attribute. Using a database with attributes makes CAPL tasks simpler because, from CAPL's perspective, an attribute is seen as a global variable.

Beside database objects, attributes may also be used as system parameters. The Vector CAN tools chain has programs that use the database and its attributes as a base to generate embedded software or CAPL code. Code generation is not possible without these attribute settings. Also, some CANoe features can only be enabled through the use of attributes. For example, attributes are used to support and enable the different types of protocols in CANoe such as J1939, NMEA2000, a specific OEM, and so on.

5.3.2 Value Tables

Value tables are used to define symbolic values. When data is received from the bus, it can be graphed or displayed in hexadecimal, decimal, or engineering units. Not all data makes sense if it is graphed or displayed in engineering units; some data actually makes more sense if it is displayed symbolically. For example, a data item CANalyzer received references the gear position in a vehicle with an automatic transmission. The data item can be viewed as 0, 1, 2, and 3, or can be displayed as Neutral, Drive, Reverse, and Park. If you wish to view the data item symbolically, you must define a value table for the numeric data.

5.3.3 Network Nodes

A network nodeconsists of one CAN Controller and a transceiver in an ECU. Occasionally, an ECU may contain more than one node. A node may have a number to identify itself and is responsible for a set of functions of that ECU. For example, a gateway module is an ECU used to exchange data between two different networks. To exchange data, two protocol controllers are used. In this case, there are two nodes in this ECU, each responsible for transmitting and receiving data on a network.

5.3.4 Messages

A messageis a container holding a block of data transmitted onto the bus. This block of data is shown numerically in CANoe in either hexadecimal or decimal. To display the message name and its data field, in engineering units or symbolically, the message has to be defined in the database.

5.3.5 Signals

Signals are the actual data objects to be exchanged between nodes, and are encoded in the data field of CAN messages. Signals typically occupy most of a CAN message's data bytes, if not all, and they must not overlap. In the database, signals can have parameters for conversion of raw data to physical units (also known as an engineering unit).

5.3.6 Environment Variables

Environment variables are data objects global to the CANoe environment, and are used to link the functions of a CANoe panel to CAPL programs.

6 Using Panels with CAPL (CANoe only)

As mentioned before, one key difference between CANalyzer and CANoe is that CANoe can simulate an unlimited number of nodes in one or more CAN networks. The other key difference is that CANoe supports user-defined graphical interfaces called panels. In most applications, a panel is made for each node for clarity. Each panel contains visible controls that will react to both external events (for example, activation of a switch) and internal events (for example, message reception from the CAN bus). Integrating the panels into CANoe is optional.

The **Panel Editor** is used to create graphic panels (see Figure 31 for an example of CANoe panels). Besides an I/O interface using buttons and switches, the **Panel Editor** also uses bitmaps for display and for controls to show different graphical values. After the panels are created, they must be associated to CANoe to be used.

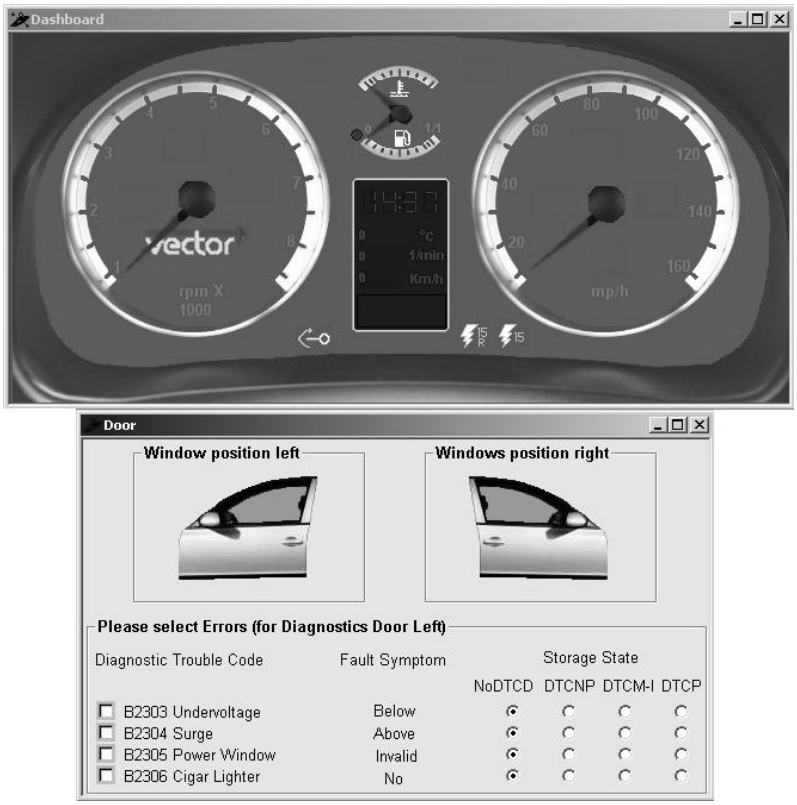

Figure 31 – CANoe Panels

Communication between panels and CAPL involves environment variables. All dynamic control elements within a panel should be associated with an environment variable while the panel is being created. When the user clicks or adjusts the value of a control, the associated environment variable's value will change as an environment variable event procedure is executed in CAPL. These variables should already be defined in the database, along with messages and signals. If they are defined in the database, CAPL will have access to them. On the other hand, the value of the environment variables can also be changed by CAPL. These changes will also affect the displayed controls on the panel. For more information on using panels, see Chapter 23 entitled Using the Panel Editor to Create Panels.

Developers should always keep in mind that panels are not used to exchange data between simulated network nodes. Environment variables are not capable of exchanging data between a simulated node and a real module. The only way to exchange data in a CAN network is by using a CAN message.

Because environment variables are global and may be used in every network node in a CANoe configuration, some people take advantage of this idea and use them to exchange data. The practice of using environment variables for data exchange is a mistake, the results lurking unknown until one or more real modules are connected to CANoe to communicate with the simulated network nodes.

7 The CAPL Browser

In contrast to traditional C program development, which uses a text editor and compiler, CAPL programs are developed using the CAPL Browser. The CAPL Browser is the fundamental programming environment used to develop all CAPL programs.

The CAPL Browser provides an easy-to-use “edit-thru-compilation” development process and is used to organize, create, edit, and modify event procedures. The CAPL Browser is more than a text editor because it not only supports source code development - it also organizes CAPL software into a tree-like structure of events, and it has an internal compiler.

In this programmer’s environment, you can develop new programs, import and export source files, associate a database, and compile CAPL software. The Browser is designed to quickly edit CAPL programs, and the integrated compiler also makes it easy to trace errors after compilation.

Figure 32 shows a general example of the CAPL Browser API.

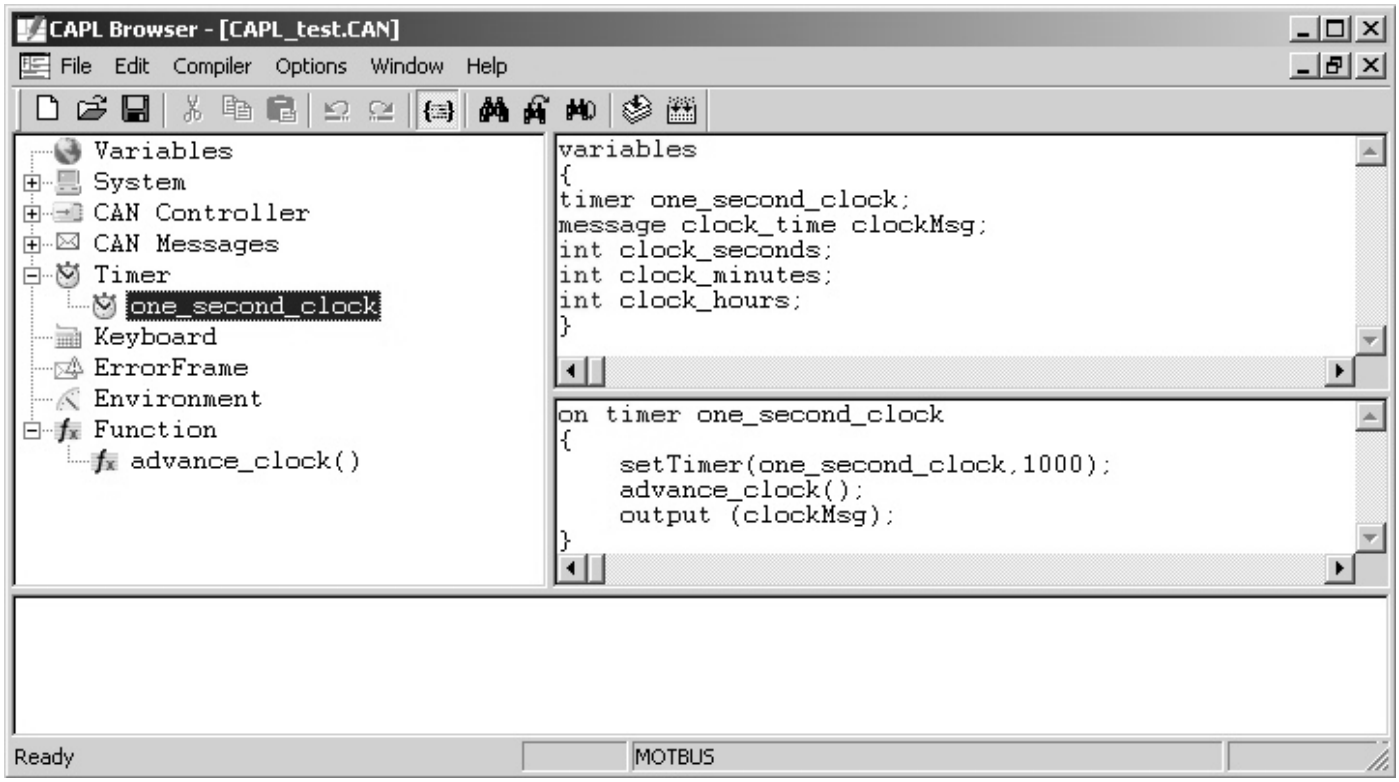

Figure 32 – Generalized Example of the CAPL Browser User Interface

7.1 Starting the CAPL Browser

While the CAPL Browser can be started in several ways, the developer normally first adds program blocks to the appropriate window in CANoe or CANalyzer and then double-clicks on the blocks to access the corresponding CAPL program. In CANoe, clicking on the pencil icon on a network node within the **Simulation Setup** window will also activate the CAPL Browser.

Although the CAPL Browser can be started as a stand-alone application, it is recommended that the program be used from inside the CANoe or CANalyzer application to ensure that it properly recognizes the associated database, hardware parameters, and CAPL-related DLLs. An example of incorrect compiling would be if the browser cannot find the variables in the program that are defined in the database.

7.2 Browser Organization

The CAPL Browser uses an event-driven control architecture with the corresponding CAPL program that is organized into event procedures. The CAPL Browser's main window is organized as shown in Figure 33. This arrangement shows and supports the creation and modification of event procedures.

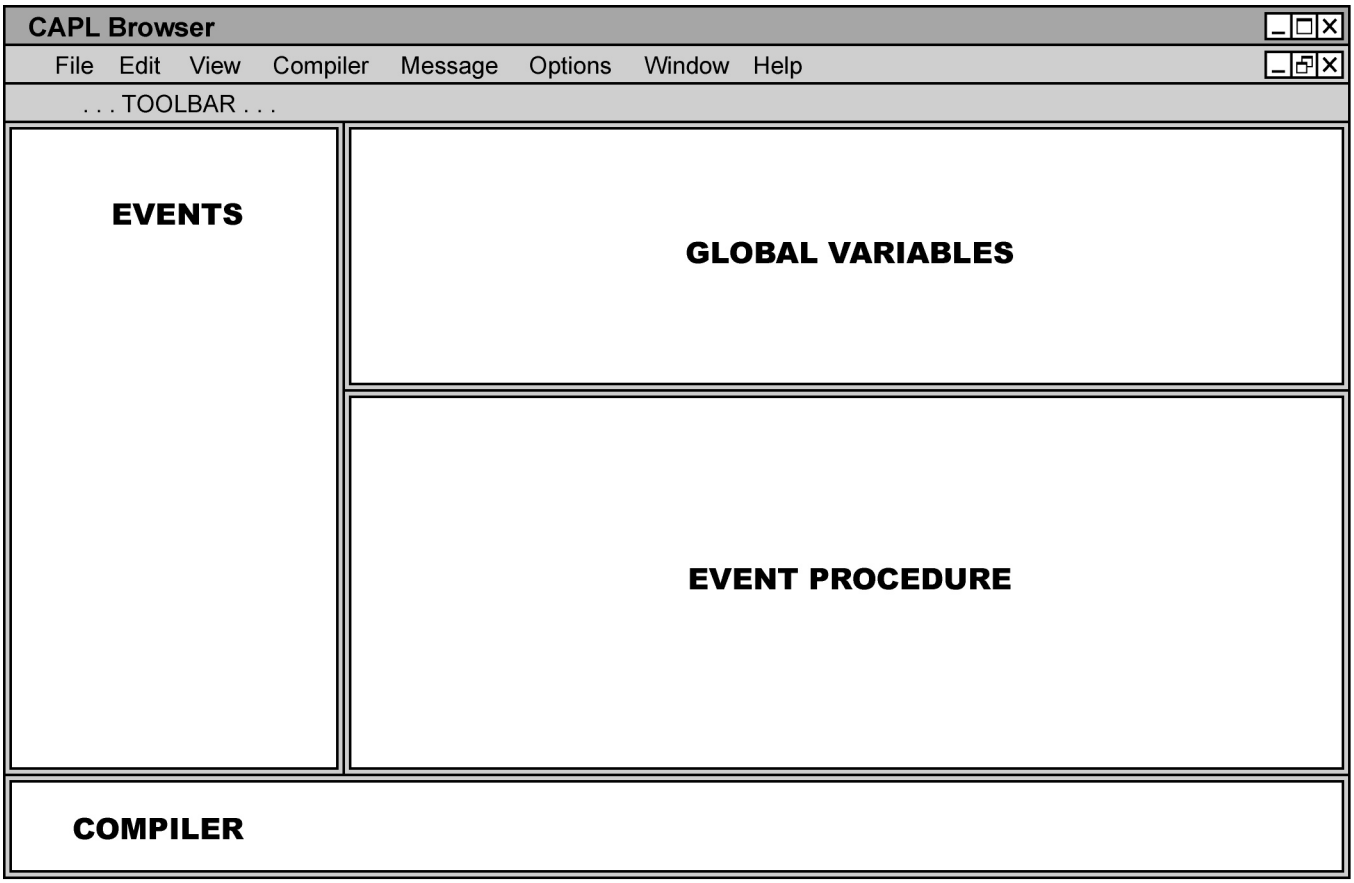

Figure 33 – General Organization of the CAPL Browser Windows

The upper left window, the **Events** window, lists both the different types of CAPL events and the names of the procedures associated with each event category in a tree view.

The upper right window, the **Global Variables** window, is used to declare all necessary global variables for the CAPL program.

The lower right window, called the **Event Procedure** window, displays the source code entered for the selected procedure highlighted in the **Events** window.

The bottom window, the **Compiler** window, shows compiler activities and results.

The windows have another layout that may be more suitable to some users. Global variables can be represented either in an individual window or in the window displaying the procedure text. If the **Event Procedure** window is used to display the global variables, then the global variables will not be shown if an event procedure is currently displayed. This feature is in the Editor Options dialog, reached with the **`Options → Editor`** command. All CAPL programs must be closed before it can be activated.

7.3 Using the Right Mouse Button

The right mouse button provides context-dependent functionality in most CAPL Browser windows. Learning how and when to use the popup (shortcut) menus, accessed with the right mouse button, will allow the user to use the browser most effectively. When in doubt, try clicking the right mouse button in a CAPL Browser window to see the context-sensitive options. Most commands are accessible in all versions by using the right mouse button in the appropriate editing window at the cursor location.

7.4 The Events Window

The **Events** window lists all event procedure categories in a tree-type structure, which may be expanded by clicking on it to view all currently defined event procedures.

The two types of events listed in the **Events** window are the following:

- System-specific events
- User-defined events

System-specific event categories include the following:

- System
- CAN Controller
- ErrorFrame

System-specific events only allow access to a set of pre-defined CAPL event procedures. Duplicates of these events are not allowed.

User-defined event categories include the following:

- CAN Message
- Timer
- Keyboard
- Environment
- Function

The developer can create an unlimited number of user-defined events. For example, you can define more than one timer to transmit a different message periodically.

7.4.1 Creating an Event Procedure

To create a new event procedure, highlight the desired event category in the **Events** window, then right-click and select **New** to create a new event.

For system-specific event categories, the **New** option extends to a submenu to select the available event procedures.

Selecting an event will show its source code in the **Event Procedure** window.

7.5 The Global Variables Window

Located in the upper right corner of the CAPL Browser, the **Global Variables** window, as shown in Figure 33, is where global variables are entered for the current CAPL program. You can only declare and initialize global variables in this window. Function invoking, event triggering, and processes execution are not allowed to implement in this window.

Note: Timer variables can only be declared in this window with either the **msTimer** or **Timer** data type.

The advantage of having global variables in an independent window is that they are always visible when editing event procedures. This makes it easier to edit, add or remove a variable without losing your place in the code.

7.6 The Event Procedure Window

Seen earlier in Figure 33, the **Event Procedure** window in the lower right corner of the CAPL Browser is where code is created for an event procedure.

Right clicking on a specific event in the **Events** window and selecting **New** from the popup menu creates a new event procedure. This sets up an empty procedure in the **Event Procedure** window. For example:

```
on errorFrame
{
}
```

This event procedure declaration is fully complete and does not need to be modified, but will need to enter the code between the braces. However, some event procedure headers require modification. Below is an example of the generic skeletal code for an **on message** event procedure:

```
on message <newMessage>
{
}
```

The user needs to fill in the identifier by replacing the text **<newMessage>** with the name of the actual message or the message's numeric identifier. If the database has already been associated with the message defined, right-click to a popup menu and select **CANdb Message** to open another window where the message list is displayed for you to choose the message(s) you want. This approach eliminates typing mistakes, and symbolic displays occasionally are more meaningful than numeric displays. All user-defined variables in a CAPL program or from a database are case-sensitive.

7.7 Colorized CAPL Syntax

CAPL programming syntax is shown in different colors in the **Event Procedure** and **Global Variables** window. Although the use of color is configurable, the default color scheme is listed in Table 2. To modify the color setting, go to the **Options** menu and then select **Editor**.

CAPL Code Type	Color
CAPL function	blue
Comment	green
Intrinsic function	dark blue
Keyword	blue
Number	black
Operator	black
String	purple
Text	black
Text Selection	white on black background

Table 2 – CAPL Syntax Color Defaults

7.8 Compiling Code

There are at least three ways to compile a CAPL program:

- Press the 'F9' key while in the CAPL Browser
- Choose **Compile** or **Compile All** from the **Compiler** menu or shortcuts
- Click on the compile shortcut icon on CANalyzer or CANoe's main menu

Choosing **Compile All** compiles all of the CAPL programs that are currently open in the CAPL Browser one at a time.

Compiling a CAPL program with the *.CAN extension (text-formatted file) creates a binary source file with the extension *.CBF (CAPL Binary Format) located in the same directory as the *.CAN file (if the *.CAN file exists).

Note: Even though the software does not need the *.CAN file to run, it is good practice to always have a *.CAN file version of the program because the CAPL Browser cannot read compiled *.CBF files. The *.CAN file will be needed if changes are required in the future.

If you try to compile the code of a new program you just implemented in the CAPL Browser, a prompt may appear to ask you to save the changes. This only saves the *.CBF file and not the *.CAN file. To save the program as a *.CAN file, go to the **File** menu and select **Save**.

7.9 Fixing Compilation Errors

If an error is found when compiling, the **Compiler** window will display the first compilation error it encountered. Double-clicking on the compilation error line will take the mouse cursor to either the **Event Procedure** or **Global Variables** window on the line or close to the line where the error was detected.

After correcting the error, compile the code again to see if there are any other errors. If there is another error, this process repeats until the CAPL program is error-free.

7.10 Debugging Run-Time Errors

A number of CAPL run-time errors are monitored. Some of these run-time errors are as follows:

- Division by zero
- Exceeding the upper or lower limits of the array
- Exceeding the upper or lower limits of the offset in the data range of messages
- Stack overflow when CAPL subroutines are called

If a run-time error is detected, the intrinsic (built-in) function **runError()** is called. This stops the measurement and outputs information to the **Write** window with the name of the CAPL program, the type of error and an error number.

The output in the **Write** window might look like the following:

Start of measurement 04:02:33 pm
Bus with 500000 BPS.
PRG (CAPL): < MYPROG > generated an error at CAPL pos: 1.
Locate with CAPL browser option: Find runError - err number -> 1.
Unknown error
measurement stopped
End of measurement 04:03:56 pm

The error number helps to find the place in the CAPL source code that generated the error.

The line that begins with **Locate…** reports the error number (in this example, the error number, is 1). Using the **Compiler** menu, select the **Find runtime error** command and enter the error number. The CAPL Browser will then show you where this run-time error occurred.

The user can also manually invoke the CAPL function **runError()** in the code to generate meaningful outputs. When using the **runError()** function in your CAPL programs or when an internal run-time error is generated, a message is displayed in the **Write** window with an error number. Different errors generate different messages.

7.11 Using Other Text Editors

While development of CAPL source programs is possible outside the CAPL Browser environment, the CAPL Browser allows errors to be more efficiently traced back to their source. The essential file format of CAPL source programs is in text (*.CAN), but CAPL programs may be modified using other text editing programs (such as Notepad and WordPad). This capability remains possible as long as you do not change the inherent CAPL-dependent structure of the file during editing.

Using the CAPL Browser has other advantages. If an associated database is required, the Browser can display the messages, signals, and environment variables' names. It is also possible to select the names and insert them directly into the CAPL program. These names are case sensitive in CAPL, therefore, it is a good idea to insert them by selecting rather than by manually entering them.

In addition, compiling for error checking can be done with the CAPL Browser itself. Outside of the CAPL Browser, you must rely on CANalyzer or CANoe to compile for you and use another text-editing program to correct any errors.

7.12 CAPL Source File Format

If there is no CAPL Browser available, a text editor can be used to write the CAPL program. CANalyzer or CANoe can subsequently compile it. Be advised, however, that this process is not recommended because the file formatting is quite unique, and it is easy to unintentionally render it unusable.

When a CAPL source file is displayed in a text editor outside the CAPL Browser environment, each section requires both a CAPL-specific beginning and an ending line that appears as a C source code comment. These special lines always begin with the same character sequence, include a CAPL-specific name, and end with the same character sequence as shown below:

Beginning character sequence: **/*@@xxx: */**
Ending character sequence: **/*@@end */**

The syntax above is typical for system-specific events. For user-defined events, the **xxx** is replaced at the beginning of a code section by the name of that section. The first beginning character sequence in a program should signify the start of the Variables section with **/*@@var: */**. Each successive time will indicate the beginning of a CAPL-specific event type or function (from the **Event Procedure**s window or the tree view). The name of the specific event procedure follows the colon, for example,, **/*@@timer:tempTimer */**, **/*@@caplFunc:StoreValues()*/**.

These lines appear if a text editor is used to open a *.CAN file; however, if a *.CAN file is created in another text editor other than the CAPL Browser, these lines must be included. Otherwise the CAPL Browser cannot open the file in browse mode to compile it. The same is true if trying to compile in CANalyzer or CANoe.

In the following CAPL source file, an example of three sections has been defined. The first group establishes a section for the global variables, the second code group is for a timer called **T1**, and the third is for the procedure related to the system start event.

Example: Global Variables

```
-----------------------------------------------
    /*@@var:    */
    variables {
        msTimer T1;
    }
    /*@@end     */
```

Example: Timer

```
/*@@timer:T1:     */
on timer T1 {
     message 100 M100 = {dlc = 4, word(0) = 0xaaaa, word(2)= 0x5555};

     setTimer(T1,50);
     M100.byte(0) = M100.byte(0) + 1;
     output(M100);
}
/*@@end      */
```

Example: System Start

```
/*@@startStart: Start    */
on start {
     setTimer(T1,50);
}
/*@@end      */
```

There is no special spacing requirement between the different CAPL-specific sections, and any number of blank lines may be used between procedures. Programmers who develop CAPL source programs outside the normal CAPL Browser may further use comment fields to enhance readability and potentially improve the documentation quality of CAPL programs.

8 CAPL Syntax and Semantics

This chapter presents an overview of the basic elements used to construct simple CAPL statements. Most of the material presented is similar to what is found in books that present C programming fundamentals.

Those individuals with experience in using the C programming language should be able to skim through this chapter and use the material only as a reference in the future.

8.1 Major CAPL Differences from C

There are several important differences between CAPL and the C programming language. As you read through these different variations, you'll notice that CAPL is somewhat more simplified - some of the burden of using the C programming language has been removed. Table 3 lists some of the major concepts found in the C programming language that are not supported by CAPL.

C concepts not supported in CAPL	Difference	Functional Alternative Available
main()	not supported in CAPL	not required
header files	not supported in CAPL	not required
preprocessor	not supported in CAPL	not required
macro definition	not supported in CAPL	not required
file inclusion	not supported in CAPL	not required
conditional compilation	not supported in CAPL	not required
computer platform dependencies	not supported in CAPL	not required
pointers	not supported in CAPL	not required
local variables are dynamic	local variables are static	yes[1]
void	not required in CAPL	yes[2]
structure	not supported in CAPL	not required
union	not supported in CAPL	not required
enumeration	not supported in CAPL	not required
define	not supported in CAPL	not required
typedef	not supported in CAPL	not required
sizeof	not supported in CAPL	not required
automatic	not supported in CAPL	not required
external	not supported in CAPL	not required
register	not supported in CAPL	not required
standard C function library	partial[3]	yes[4]
string constants	not supported in CAPL	yes[5]

Table 3 – Major CAPL Programming Differences from C

Notes -
1. All locally defined variables (in each event procedure) are static; in C, the word **static** is used before a variable declaration. The workaround in CAPL is to not initialize a variable and define it together (for example, **int value; value = 1;**).
2. **Void** is only used in CAPL for the function return type. The use of void is optional for functions that return no value.
3. Some standard C functions are being used in CAPL. These functions are typically mathematical functions or string functions (for example, **abs()**, **strncpy()**) that are already predefined internally to CAPL.
4. CAPL doesn't have the complete ANSI C standard library, but it does have CAPL-specific libraries called CAPL DLLs.
5. The **string** data type is not supported in CAPL; however, you may use a character array (for example, **char value[6] = "Hello";**).

Because CAPL is completely event-driven, the **main()** function is not supported in CAPL. As there are no header files to be linked, no preprocessor is required because.

C programmers will also be pleased to notice other simplifications:

- In CAPL you do not have to declare a function prototype before you can use it.
- CAPL does not use classes or structures – templates that define the characteristics and behaviors of an object.
- Functions such as **scanf()** are not supported because CAPL has no input stream. However, CAPL does have events defined for pressing a key or keys. Inputs can also be made using environment variables through user-configurable interfaces called panels. We will discuss these topics in other chapters.

8.2 CAPL Equivalents to C Functions

There are a few function equivalents that are helpful for a C programmer to know. Below is a concise list of the similar functions, but for more details on how the CAPL functions are used, see the CAPL Function Reference Manual.

C Function	CAPL Function	Notes
sizeof	elCount	sizeof has no exact equivalent
sprintf	snprintf	Similar string formatting parameters used
printf	write	Similar string formatting parameters used

Table 4 – Function Similarities between C and CAPL

8.3 Notation

To understand the examples in this manual, it is important to be familiar with the following notations:

Notation	Meaning
. . .	When used in source code, an ellipsis means that any code is valid in this place. It is used to indicate that code unrelated to the specific example is in this location, but has been removed for readability purposes.
message 101x msg;	An “x” following a message ID number indicates an extended identifier. This forces the use of a 29-bit identifier instead of the standard 11-bit identifier.
Msg.data = 0x35;	A “0x” preceding a number indicates that the number is hexadecimal (base 16). Hexadecimal numbers can be used in place of decimal numbers anywhere in CAPL.
write(); stop()	Names of functions not in the context of source code are followed by parentheses to indicate that they are functions; therefore, functions may require parameters.

Table 5 – CAPL Notation

8.4 Comments

Comments make it easier for people to understand the program. One nice feature of C comments is that they can be placed anywhere, even on the same line as the statement they explain. A longer comment can be placed on its own line or even spread over more than one line. Everything between the opening **/*** and the closing ***/** is ignored by the compiler. Below are four comment examples in CAPL:

1. **/* This is a comment.*/**

2. **/* This comment is spread**
   over two lines.*/

3. **/*
   You can do this, too.
   */**

   /* But not this.

4. **// CAPL also accepts the C++ comment.**

As you can see, the fourth example is used in the C++ programming language. The comment is on one line only, and it has no closing character string.

8.5 Naming Conventions

The names of variables, functions, and arrays may consist of letters and digits, in any order, except that the first character must not be a number.

Listed below are examples of acceptable object names:

```
sum
number_of_units
J5x7
_sysflag
```

The following are not acceptable:

```
int                 // because it is a reserved keyword
sum$value           // the $ is not a recognized character
3Times              // because a variable name cannot begin with a number
number of units     // because spaces are not allowed
```

8.6 CAPL is Case Sensitive

When writing in CAPL, it is important to remember that lower case and upper case letters are distinct for user-defined variables (all keywords and CAPL library functions are not). For example, a CAPL program will treat each of the following similar variable names as three unique and different variables:

```
input_1
Input_1
INPUT_1
```

Because variables are case sensitive, you should have a consistent way of naming variables. It is recommended that you adopt one of the three styles shown in the example above and then continue using that particular style throughout your CAPL programs.

If it is expected that CAPL programs will be shared among a group of software programmers, it is also highly recommended that an internal CAPL coding standard be established.

8.7 CAPL Keywords

Keywords cannot be used to name a variable or function. CAPL uses reserved keywords from the C programming language. Some common C keywords are, however, not supported in CAPL.

Below is a list of reserved keywords, both supported and non-supported by CAPL.

Supported:	Not supported:
break	auto
case	const
char	enum
continue	extern
default	goto
do	register
double	short
else	signed
float	sizeof
for	static
if	struct
int	typedef
long	union
return	unsigned
switch	volatile
while	

8.8 Data Types

The basic data types supported by CAPL are integer, character, and floating point, as listed in Table 6. **Message**, **timer**, and **msTimer** are considered data types because they define a variable that symbolizes the kind of data they can store and operate. Unsigned variables can only have non-negative values, whereas signed variables can either be positive or negative.

Data Type	Description	Size	Unsigned/Signed
char	character	8 bit	unsigned
byte	byte	8 bit	unsigned
int	integer	16 bit	signed
word	word	16 bit	unsigned
long	long integer	32 bit	signed
dword	double word	32 bit	unsigned
float	single precision floating point	64 bit[1]	signed
double	single precision floating point	64 bit	signed
message	a communication message	--	--
timer	a timer with second resolution	--	--
msTimer	a timer with millisecond resolution	--	--

Table 6 – CAPL Data Types

Note: Float signals defined in the database are 32 bits

8.9 Declarations

To allow variables to be read or changed in any part of a CAPL program, use global variable declarations in the **Global Variables** window of the CAPL Browser. The simple integer data types are **dword**, **long**, **word**, **int**, **byte**, and **char**. The data types **float** and **double** are synonyms and designate 64-bit floating-point numbers conforming to the IEEE standard. They are both compatible with each other and can be used in the same way as they are used in the C programming language. Arithmetic is performed with 32-bit resolution for integers and 80-bit resolution for floating point numbers. If local variables are used in event procedures, ensure they are defined first before any other code. Variables initialization is optional during their declarations.

Below is a list of some examples of declarations:

```
char letter_a = "a";
int number_days_in_year = 365;
message wake-up xxx;          // see Chapter 11 on using messages for more detail
timer one_second;             // see Chapter 14 on using timers for more detail
int j, k = 2;                 // j = 0 implicitly
double x = 33.7;
char p;
```

In addition to variable declarations, a CAPL program must also have declarations for each message and timer used by the program. Timers have to be globally declared. Messages can be declared globally or locally. See Chapter 14 (Using Timers) and Chapter 11(Using Messages) for more detail.

8.9.1 Local Variables are Static

One important difference in CAPL when compared to C is that local variables are always declared statically. This means that they are initialized only once – the first time the event procedure or user-defined function is executed. When variables enter the procedure, they assume the value they had at the end of the previous execution of the procedure.

Let's look at the following example:

```
void myFunc()
{
    byte value = 10;          // static; called once
    write("value = %d", value);
    . . .
    value = 35;
}
```

The first time **myFunc** is called, it will print **value = 10** as expected. However, the second time it is called, and every time after that, it will print **value = 35** *as long as the measurement is running*. (It may be helpful to consider that in a C program, **byte value = 10;** would be written **static byte value = 10;**). Because of this, the preferred way to initialize non-static variables is to use a separate assignment statement after the variables have been declared. In the following example, the variables are reset to their initial values at the beginning of every function iteration:

```
void myFunc()
{
    byte value;           // variable declarations
    int x, y;

    value = 10;           // variable initializations
    x = 0;
    y = 5;

    // Main code of the function
    . . .
}
```

To avoid confusion, it might sometimes be necessary to declare all variables as global in the **Global Variables** window of the CAPL Browser.

8.9.2 Initialization

In contrast to standard C, the compiler initializes all numeric variables to **0** and all string-type variables to **null**. All message-type variables in CAPL are usually initialized to the transmit request state with its data field defaulted to **0**. The state represents the direction of transmission before the message is transmitted. Timer-type variables, on the other hand, are not automatically initialized and do not need to be initialized.

8.10 Type Casting

CAPL supports type casting. There are times when you will want to convert the value of a variable of one data type to another data type for purposes of evaluating an expression. There are two ways to do this in CAPL, both of which are the same as in C. As an example, let us declare the following variable:

```
int sum;

sum = 1.6 + 1.7;
sum = (int) 1.6 + (int) 1.7;
```

The first assignment statement uses automatic conversion, that is, 1.6 and 1.7 are added to yield 3.3. This number is then truncated to the integer 3 to correspond to an **int** variable.

In the second statement, 1.6 is cast into an integer (1 in this example) before addition is applied to get value 1+1 or 2.

8.11 Arrays

Another type of variable used in CAPL is an **array**, a collection of data items that all have the same name. Individual data items within an array must all be of the same type. The array name is always followed by square brackets that contain either the array size for a declaration statement or an index value to specify the data location within the array.

Each of the individual elements in an array is accessed by an integer index. When numbering the elements of an array, the first element is always the number 0 (zero), not 1.

CAPL supports the following types of arrays:

- Integer arrays
- Character arrays
- One-dimensional arrays
- Multi-dimensional arrays

We could define a character array that contains the 26 letters of the alphabet. Because character strings must include one additional character, the null character (\0), to indicate the end of the string, the size of this array is 27 and the index sequence is 0 through 26.

This “alphabet” character array can be declared as follows:

```
char alphabet[27] = “ABCDEFGHIJKLMNOPQRSTUVWXYZ”;
```

We can now selectively distinguish between each individual data element by using an index value. For example, to request the first element of the alphabet (the letter ‘A’) we would use **alphabet[0]**.

To declare an integer array with four integer values in it, use the following:

```
int sample_data[4] = { 100, 300, 500, 600 };
```

The element **sample_data [0]** will then contain the integer value 100, **sample_data [1]** the value 300, **sample_data [2]** the value 500, and **sample_data [3]** will contain 600.

Not all elements have to initialize to a value. For example, the below declaration will do the same thing, but will set the remaining six elements to zero:

```
int sample_data [10] = { 100, 300, 500, 600 };
```

CAPL allows arrays of any dimension to be defined. For example, we could define the common matrix or two-dimensional array. It is analogous to what is used in math, such as the below 4 x 5 matrix:

```
10      5       -3      17      82
9       0       0       8       -7
32      20      1       0       14
0       0       8       7       6
```

If we label the above matrix **M**, **Mij** would refer to the element in the **ith** row and **jth** column. In CAPL, as in C, we would define and initialize a two-dimensional array M as shown in the example below:

```
int M[4][5] =       {
                    { 10, 5, -3, 17, 82 },
                    { 9, 0, 0, 8, -7 },
                    { 32, 20, 1, 0, 14 },
                    { 0, 0, 8, 7, 6 }
                    };
```

Note that except the last row, commas are required after each brace that closes off a row. The individual elements of this array can be easily accessed using the row and column number.

To determine the number of elements in an array, use the **elCount()** function as shown in the below example:

```
int i, j;
for ( j = 0; j < elCount(array); j++ )
    for ( i = 0; i <= elCount(array[j]); i++ )
            . . .
```

8.12 Constants

To initialize a variable means to assign an initial or beginning value to it. In CAPL, this can be done in the same line as the variable declaration. When values are assigned during a declaration, they are considered constants. CAPL supports the following four types of constants:

- Integer constants
- Floating point constants
- Character constants
- String constants

8.12.1 Integer Constants

CAPL supports integer constants in two different numbering systems -- decimal (base 10) and hexadecimal (base 16). References to integer constants appear in code as shown below:

```
int value = 20;
int value2 = 0x14;
```

Decimal integer constants consist of any combination of digits between 0 and 9, with the added rule that the first digit must be a nonzero number.

The following decimal integer constants are considered valid:

```
0
1
55
2047
```

The following are considered invalid:

```
2.1     // because the period is an illegal character
2 1     // because spaces cannot be used between numbers
2,1     // because a comma cannot be used between numbers
```

A hexadecimal integer constant must begin with either **0x** or **0X** - that is, the value must always start with the number zero followed by a lowercase or upper case letter ‘X’. After this comes any combination of digits between 0 and 9 including the upper or lower case letters A, B, C, D, E, or F.

The following hexadecimal integer constants are considered valid:

```
0x0
0x1
0xFF
0xaa55
```

The following are invalid hexadecimal integers:

```
0x2.1       // because the period is an illegal character
0xE A       // because spaces cannot be used between hexadecimal digits
0x2,B       // because a comma cannot be used between hexadecimal digits
A1          // because it does not begin with 0x
```

8.12.2 Floating Point Constants

In CAPL, a floating point constant is a base-10 decimal number. A floating point constant must contain a decimal point or an exponent or both, as seen in the example below:

```
float value = 0.23;
float value2 = 23E-2;
```

An exponent must be indicated with either a lower case or upper case letter ‘E’. The value of the exponent must be an integer.

Below are some examples of valid floating point constants:

```
2.1
2.1e0
3.1415
0.00034
22e+3
1E-6
```

The following are not valid:

```
4           // because no decimal point or exponent is indicated
1,024       // because commas are not allowed
2E3.4       // because the exponent is not an integer
1E 6        // because a space is not allowed in an exponent
```

8.12.3 Character Constants

A character constant is a single character surrounded by apostrophes, as seen in the examples below:

Below are some examples of character constants:

```
char value = ‘B’;
char value2 = ‘8’;
char value3 = ‘?’;
```

CAPL supports the use of the ASCII character set. The three character assignments above can be replaced by hexadecimal respectively:

```
char value = 0x42;
char value2 = 0x38;
char value3 = 0x3F;
```

It is often necessary to establish the value of characters that are not printable. These can be expressed as an escape sequence. An escape sequence begins with a back slash (\) and is followed by one or more special characters.

```
'\t'        // is the character constant for the tab
```

Of particular interest is the escape sequence **\0**, which represents the ASCII null character that is used to indicate the end of a string. The null character is never to be overlooked when you are using string functions like **strncpy()** and **strncat()**. The two functions are used to copy two strings into one and to concatenate two strings into one respectively. It is also important to note that the ASCII null character is not equal to the character constant **'0'** (zero).

8.12.4 String Constants

A string constant consists of a series of one or more consecutive characters enclosed by double quotation marks (“). Below are three examples of string constants:

```
char value[30] = “Here’s a string in C and CAPL”;
char value2[19] = “spaces are allowed”;
char value3[31] = “with a tab escape sequence \t”;
```

Strings are stored in an array of data elements of type **char**. The last element contains the null character **\0**, which is used to indicate the end of the string.

Note: Make sure you remember the string size is always length + 1.

In CAPL, **enum** data types are treated in a manner similar to strings because values are initialized with symbolic names to represent integer constants. Even though the CAPL language does not support the **enum** data type, the database editor allows the **enum** type to be defined for all attribute objects. When CAPL needs to reference an enum-type attribute, use the **strncpy()** function. Below is an example of how to get the value of an enum-type message attribute that is defined for the receiver node’s symbolic name:

```
char buffer[10];
strncpy(buffer, msg_name.node_name, elcount(buffer)-1);
```

8.12.5 Symbolic Constants

A symbolic constant can be used as a substitute name. In the C language, this is usually accomplished with a **define** statement as in the example below:

```
#define     TRUE        1
#define     FALSE       0
```

The above is **NOT** used in CAPL.

CAPL does have a few symbolic constants. For example, the **PI** constant might be useful in some situations. It contains the mathematical value of pi (π) to the maximum precision of 16 decimal places.

While some constants are only used in specific cases, such as message selectors, built-in constants will be explained in their specific sections. Constants used with specific CAPL functions may be found in the CAPL Browser’s Online Help or in the CAPL Function Reference Manual.

8.13 Operators

As with C, CAPL offers a wealth of operations to process information. You can do arithmetic, logic functions, bitwise operations, compare values, modify variables, combine relationships logically, and much more. The symbols used to do this are called operators. The data items that operators act upon are called operands.

While some operators act upon only one operand, other operators required two operands. Most operators allow the individual operands to be expressions. A few operators permit only single variables as operands.

CAPL operators include:

- Arithmetic Operators
- Assignment Operators
- Relational Operators
- Boolean Operators
- Bitwise Operators
- Miscellaneous Operators

8.13.1 Arithmetic Operators

Table 7 shows the various arithmetic operators supported in CAPL. Like C, CAPL does not support an exponential operator.

Symbol	Operation	Usage	Example
+	Add	arithmetic_expression_1 + arithmetic_expression_2 Sum of arithmetic_expression_1 and arithmetic_expression_2	x = y + 2 Set x equal to the value of y plus 2
-	Subtract	arithmetic_expression_1 - arithmetic_expression_2 Difference of arithmetic_expression_1 and arithmetic_expression_2	x = y - 2 Set x equal to the value of y minus 2
-	Negation	- arithmetic_expression_1 Minus arithmetic_expression_1	x = - x Set x equal to the value of minus x
*	Multiply	arithmetic_expression_1 * arithmetic_expression_2 Product of arithmetic_expression_1 and arithmetic_expression_2	x = y * 2 Set x equal to the value of y times 2
/	Divide	arithmetic_expression_1 / arithmetic_expression_2 Quotient of arithmetic_expression_1 and arithmetic_expression_2	x = y / 2 Set x equal to the value of y divided by 2
%	Modulo	integer_expression_1 + integer_expression_2 Remainder of integer_expression_1 divided by integer_expression_2	seconds = minutes % 60 Set seconds equal to the remainder of the minutes divided by 60
++	Postfix Increment	integer_variable_1++ Increases integer_variable_1 by 1 Value is used before increment is applied	y = x++ Set x to y before x + 1
++	Prefix Increment	++integer_variable_1 Increases integer_variable_1 by 1 Value is used after increment is applied	y = ++x Set x to y after x + 1
--	Postfix Decrement	integer_variable_1-- Decreases integer_variable_1 by 1 Value is used before decrement is applied	y = x-- Set x to y before x - 1
--	Prefix Decrement	--integer_variable_1 Decreases integer_variable_1 by 1 Value is used after decrement is applied	y = --x Set x to y after x - 1

Table 7 – Arithmetic Operators

The following are examples of arithmetic operator usage:

```
int x,y,z;
y = 8;
z = 4;

x = y + z;          // Addition. Result = 12
x = y - z;          // Subtraction. Result = 4
x = y * z;          // Multiplication. Result = 32
x = y / z;          // Division. Result = 2
x = z % y;          // Modulo. Result = 4
x = y++;            // Increment. Result = 9
x = z--;            // Decrement. Result = 3
```

8.13.2 Assignment Operators

Table 8 lists the various assignment operators supported in CAPL. The equal sign (**=**) does not mean “equals”. This operator is used to assign a value to a variable, with or without an operation to be performed first.

Symbol	Operation	Usage	Example
=	Assignment	variable = expression Assign the value of the expression to the variable	x = 16 Sets x to 16
+=	Addition and assignment	arithmetic_variable += arithmetic_expression Increment of arithmetic_variable by arithmetic_expression	x += 4 Increments x by 4
-=	Subtraction and assignment	arithmetic_variable -= arithmetic_expression Decrement of arithmetic_variable by arithmetic_expression	x -= 2 Decrements x by 2
*=	Multiplication and assignment	arithmetic_variable *= arithmetic_expression Multiply arithmetic_variable by arithmetic_expression	x *= 8 Multiplies x by 8
/=	Division and assignment	arithmetic_variable /= arithmetic_expression Divide arithmetic_variable by arithmetic_expression	x /= 32 Divides x by 32
%=	Remainder (Modulo) and assignment	arithmetic_variable %= arithmetic_expression Set arithmetic_variable to arithmetic_variable modulo arithmetic_expression	x %= 10
<<=	Left shift assignment	arithmetic_variable <<= arithmetic_expression Shift arithmetic_variable left by arithmetic_expression bits	x <<= 1 Shifts x 1 bit position left
>>=	Right shift assignment	arithmetic_variable >>= arithmetic_expression Shift arithmetic_variable right by arithmetic_expression bits	x >>= 2 Shifts x 2 bit positions right
&=	AND assignment	arithmetic_variable &= arithmetic_expression AND arithmetic_variable with arithmetic_expression	x &= 0x0F ANDs x to isolate least significant 4 bits
|=	OR assignment	arithmetic_variable |= arithmetic_expression OR arithmetic_variable with arithmetic_expression	x |= 0x01 ORs x to set the lowest bit to 1
^=	XOR assignment	arithmetic_variable ^= arithmetic_expression XOR arithmetic_variable with arithmetic_expression	x ^= 0x01 XORs x to toggle the lowest bit

Table 8 – Assignment Operators

The following are examples of assignment operator usage:

```
int y, z;
y = 8;
z = 4;
                    // each statement below is independent from the others
y += z;             // Addition. Result: y = 12
y -= z;             // Subtraction. Result: y = 4
y *= z;             // Multiplication. Result: y = 32
y /= z;             // Division. Result: y = 2
y %= z;             // Modulo. Result: y = 0
y <<= 1;            // Left-shift. Result: y = 16
y &= z;             // AND. Result: y = 0 (binary arithmetic)
y |= z;             // OR. Result: y = 12 (binary arithmetic)
y ^= z;             // XOR. Result: y = 12 (binary arithmetic)
```

8.13.3 Boolean Operators

As shown in Table 9, CAPL supports the same Boolean operators found in the C programming language.

Symbol	Operation	Usage	Example
!	Logical NOT	!arithmetic_expression TRUE if integer_expression_1 is equal to integer_expression_2; otherwise FALSE	if(!x) Checks to see if x is equal to FALSE[1]
||	Logical OR	expression_1 || expression_2 TRUE if expression_1 is TRUE or expression_2 is TRUE	if(x <0 || x>59) TRUE[1] if x is outside of range 0 to 59
&&	Logical AND	expression_1 && expression_2 TRUE if expression_1 is TRUE and expression_2 is TRUE	if(x >0 && x<13) TRUE[1] if x is inside the range 1 to 12

Table 9 – Boolean Operators

Note: The Boolean value TRUE is represented by the integer 1 (one) and FALSE is represented by the integer 0 (zero).

The following are examples of Boolean operator usage:

```
byte y, z;
y = 0x00;
z = 0x01;

if (!y)         // test result is TRUE
if (!z)         // test result is FALSE

if (y == 0 && z == 1)          // test result is TRUE
if (y == 1 || z == 1)          // test result is TRUE
```

In the last example above, the code that follows the statement is executed as long as y = 1 or z = 1.

8.13.4 Relational Operators

Table 10 lists the various relational operators supported in CAPL.

Symbol	Operation	Usage	Example
==	Test for equal	integer_expression_1 == integer_expression_2 TRUE if integer_expression_1 is equal to integer_expression_2; otherwise FALSE	if(x == 0) Checks to see if x is equal to zero
!=	Test for not equal	integer_expression_1 != integer_expression_2 TRUE if integer_expression_1 is not equal to integer_expression_2; otherwise FALSE	if(x != 0) Checks to see if x is non-zero
>	Test for greater than	arithmetic_expression_1 > arithmetic_expression_2 TRUE if arithmetic_expression_1 is greater than arithmetic_expression_2; otherwise FALSE	if(x > 59) Checks to see if x is greater than 59
>=	Test for greater than or equal to	arithmetic_expression_1 >= arithmetic_expression_2 TRUE if arithmetic_expression_1 is greater than or equal to arithmetic_expression_2; otherwise FALSE	if(x >= 60) Checks to see if x is greater than or equal to 60
<	Test for less than	arithmetic_expression_1 < arithmetic_expression_2 TRUE if arithmetic_expression_1 is less than arithmetic_expression_2; otherwise FALSE	if(x < 256) Checks to see if x is less than 256
<=	Test for less than or equal to	arithmetic_expression_1 <= arithmetic_expression_2 TRUE if arithmetic_expression_1 is less than or equal to arithmetic_expression_2; otherwise FALSE	if(x <= 59) Checks to see if x is less than or equal to 59

Table 10 – Relational Operators

The following are examples of relational operator usage:

```
int y,z;
y = 8;
z = 4;

if (y == z)          // test result is FALSE
if (y != z)          // test result is TRUE
if (y <= z)          // test result is FALSE
if (y >= z)          // test result is TRUE
```

8.13.5 Bitwise Operators

As shown in Table 11, CAPL supports similar bitwise operators found in the C programming language.

Symbol	Operation	Usage	Example
~	One's complement	~integer_expression compute the one's complement of integer_expression	x = ~y Sets x equal to the one's complement of y
&	Bitwise AND	integer_expression_1 & integer_expression_2 Bitwise AND integer_expression_1 with integer_expression_2	x = y & z Sets x equal to the logical AND of y with z
|	Bitwise OR	integer_expression_1 | integer_expression_2 Bitwise OR integer_expression_1 with integer_expression_2	x = y | z Sets x equal to the logical OR of y with z
^	Bitwise XOR	integer_expression_1 ^ integer_expression_2 Bitwise XOR integer_expression_1 with integer_expression_2	x = y ^ z Sets x equal to the logical XOR of y with z
<<	Left shift	integer_expression_1 << integer_expression_2 Shift left integer_expression_1 by integer_expression_2 bit positions and fill with zeros on the right	x << 2 Shifts x to the left by 2 bit positions
>>	Right shift	integer_expression_1 >> integer_expression_2 Shift right integer_expression_1 by integer_expression_2 bit positions and fill with zeros on the left	x >> 1 Shifts x to the right by 1 bit position

Table 11 – Bitwise Operators

The following are examples of bitwise operator usage:

```
byte x, y, z;
y = 0x0AA;      // y = 1010 1010
z = 0x05A;      // z = 0101 1010
x = y & z;      // AND result = 0000 1010
x = y | z;      // OR result = 1111 1010
x = y ^ z;      // XOR result = 1111 0000
x = y << 1;     // shift left result = 0101 0100
x = y >> 1;     // shift right result = 0101 0101
x = ~y;         // 1's complement result = 0101 0101
```

8.13.6 Miscellaneous Operators

CAPL supports several miscellaneous operators as listed in Table 12.

Symbol	Operation	Usage	Example
[]	Array	named_array[integer_expression] The integer_expression element of the named_array	x = y[z] Sets x to the value of the z-th element of the y array
.	Member	structure_variable.structure_member Member structure_member of structure_variable	x = message.signal Sets x to a signal value of a message[1]
?:	Conditional evaluation	arithmetic_expression ? expression_1 : expression_2 If arithmetic_expression is TRUE, then evaluate expression_1; otherwise evaluate expression_2. The value is expression_1 or expression_2.	x = (y <= 0) ? -y : y Sets x to absolute value of y

Table 12 – Miscellaneous Operators

Note: CAPL supports the concept of the structure, but does not allow them to be declared. The C structure member reference (dot) operator is used to react to events.

The following are examples of miscellaneous operator usage:

```
byte x, y, z;
z = 3;
y = 5;

x = (y < z) ? z : y;          // conditional result x = 5
x = messagename.signalname    // x is assigned a signal value
                              // see chapter on Using Messages for more detail
```

8.13.7 Unsupported Operators

Symbol	Operation	NOT ALLOWED IN CAPL	Example
&	Address	&variable Address of variable	x = &y
#	Preprocessing	#keyword Preprocessing directives	#define …
*	Address	*pointer_expression Address of variable	x = *y
->	Member	structure_pointer_expression -> structure_member Member structure_member of structure pointed to by structure_pointer_expression	x = y -> z

Table 13 – Unsupported Operators in CAPL

8.14 Control Statements

As in C, CAPL is implemented using building blocks called **statements**. A CAPL program is essentially a composition of statements that form instructions, which are the executed by the computer. Typically, statements are ended by a semicolon (;). Compound statements are statements grouped together; therefore, they are enclosed by braces.

CAPL supports the same traditional control and conditional branching statements used in the C programming language. Statements may be on one or more lines. A statement is not finished until the semicolon is reached, or the closing brace is reached.

Control statements allow conditions to be tested during program execution. Depending on the outcome of the test, several possible program actions can be executed.

The relational operators can be used for logical testing.

Several types of control statements are available including:

- Selective branching
- Looping
- Unconditional branching

8.14.1 Branching Statements

Sometimes more than one action can be performed, depending on what the condition is. If it's cold in your house, you turn on the furnace. If it's not, you don't. The same logic applies to selective branching statements.

In the following example, assume that a car's automatic speed (cruise) control is set when the speed of a car exceeds 70 MPH. The following code is an example of **if** statement usage in CAPL:

```
float MPH = 70.0;
float speed;
float cruising_speed;
…
if (speed >= MPH)  cruising_speed = speed;
...
```

The **if** statement is called a branching statement because the program may process one of two paths with the statement. If the expression in the parentheses (in this case, speed >= MPH) evaluates to be true, the following statement or whatever is within the braces is executed. Otherwise, it is skipped. The above example is a simple **if** statement; therefore, braces are not required. For readability, it may be helpful to use braces as shown below:

```
if (speed >= MPH)
{
    cruising_speed = speed;
}
```

8.14.2 If Statement

The **if** statement is used to evaluate an expression and then take actions if and only if the outcome of the evaluation is true.

Generalized format :

```
if (expression) statement;
```

The expression must be placed in parentheses. In this form, the statement will be executed only if the expression is evaluated as being true or has a nonzero value. If the expression is evaluated as false or has a value of zero, then the statement will be ignored. See the example below:

```
// this is a function that advances clock minutes
increment_clock_minutes()
{
    int clock_minutes;

    clock_minutes = clock_minutes + 1;
    if (clock_minutes >= 60)                    // if true
    {
            clock_minutes = 0;
            // increment hour variable
    }
}
```

8.14.3 If-Else Statement

The **if-else** statement is used to evaluate an expression and then takes one of two possible actions, depending on whether the outcome of the evaluation is true or false.

Generalized format:

```
if (expression) statement_1
else statement_2
```

If the expression is evaluated to be true or has a nonzero value, then **statement_1** will be executed; otherwise, **statement_2** will be executed.

Because the **else** portion of the statement is optional, ambiguity exists when the else is omitted from a nested **if** sequence. See the example below:

```
// this is a function that advances clock minutes
increment_clock_minutes()
{
    int clock_minutes;

    if (clock_minutes >= 60)                // if true
    {
        clock_minutes = 0;
    }
    else                                    // if false
    {
        clock_minutes = clock_minutes + 1;
    }
}
```

8.14.4 Switch Statement

The **if-else** statement makes it easy to compare two alternatives. If several alternatives are possible, you can use a nested **if-else** statement. Sometimes, however, it is more convenient to use the **switch** statement. The **switch** statement provides multiple control branching with a selection process that tests a value of an expression against a list of integer or character constants. This is useful when more than two alternatives are possible, such as comparing a constant value against several different incoming temperatures. The value of the expression is tested in consecutive order against the values of the constants specified in the individual **case** statements. When a match is found, the associated statement sequence specific to that case is executed until a break statement is encountered or until the end of the switch statement is reached.

For each case choice, the first statement within the group must be preceded by one or more case labels, also known as case prefixes. Each case must use a unique label or constant - two identical labels are not allowed.

Each case must be labeled by either an integer or a character constant or a constant expression. A constant expression cannot involve variables or function calls.

Note: A case labeled as a character constant is internally converted to an integer.

Below is the general format of a **switch** statement:

```
switch (expression) {
    case value1: statement1; statement2; … break;
    case value2: statement1; statement2; … break;
    . . .
    default: statement1; statement2; … break;
}
```

The **default** statement is executed if none of the cases match the value of the expression. If none of the cases match the value of the expression and the default statement is not implemented, then no action will be taken by the switch statement. Below is an example:

```
float value1, value2, result;
char operator;

...
switch ( operator )
{
    case '+':       result = value1 + value2;
                    break;

    case '-':       result = value1 – value2;
                    break;
```

```
        case '*':       result = value1 * value2;
                        break;

        case '/':       if ( value2 == 0) write ("Division by zero!");
                                else result = value1 / value2;
                        break;

        default:        write ("Unknown operator.");
                        break;
    }
```

8.15 Loops

Another basic operation performed in C that is also allowed in CAPL is a loop. A loop is a way to repetitively execute a set of statements. There are three constructs used in C to perform this: the **for** statement, the **while** statement and the **do–while** statement.

8.15.1 While Statement

The **while** statement provides a looping mechanism during which a statement or group of statements is repetitively executed until some condition has been satisfied.

The general form of the **while** statement contains both an expression and a statement. If the expression is true or evaluates to any nonzero value, then the statement is executed until the expression becomes false. Once the expression is evaluated as being false, then the loop terminates and the program resumes after the loop.

The generalized format for the **while** statement is as follows:

while (expression) {statement};

The expression is always evaluated before the statement is executed. Although this may eliminate performing a separate conditional test before executing a loop, remember that if the expression is false the first time through, then the statement is never executed.

The following example converts a decimal number into a binary array:

```
byte binaryArray[16];       // global variable

binary ( int number )       // user-defined CAPL function
{
    int index;
    index = 0;

    while ( number != 0 )
    {
            binaryArray[index++] = number % 10;
            number = number / 10;
    }
}
```

If the input is: **1234**
the output will be: **11010010**

Note: The statement **int index = 0;** will NOT work here because local variables in CAPL are static, that is they are initialized only ONCE at the start and not each time that the function is called.

If you want to have the statement executed first before the expression is evaluated, then use the alternate **do-while** statement.

8.15.2 Do–While Statement

The **while** loop and **for** loop are entry-condition loops. The test condition is checked before each iteration of the loop, so it is possible that the statements in the loop will never execute. CAPL also recognizes a C language exit-condition loop in which the condition is checked *after* each iteration of the loop, guaranteeing that statements in the loop will be executed at least once. The **do-while** statement repetitively executes a statement or group of statements until a condition has been satisfied.

The general form of the **do-while** statement contains both a statement and an expression. If the expression is true or evaluates to any nonzero value, then the statement is executed until the expression becomes false. Once the expression is evaluated as being false, then execution resumes after the loop.

The generalized format for the **do-while** statement is as follows:

```
do {statement}
while(expression);
```

The following is an example of **do-while** usage:

```
byte binaryArray[16];       // global variable

binary ( int number )       // user-defined CAPL function
{
    int index;
    index = 0;

    do
    {
            binaryArray[index++] = number % 10;
            number = number / 10;
    }
    while ( number != 0 );
}
```

If the input is: **1234**
the output will be: **11010010**

Note: The statement **int index = 0;** will NOT work here, because local variables in CAPL are *static*, i.e., they are initialized only ONCE at the start and not each time that the function is called.

8.15.3 For Statement

The **for** statement also provides a looping mechanism. This statement includes an expression used for loop initialization, another expression that determines whether or not the loop is to continue, and a third expression that is executed after each iteration.

The generalized format for the **for** statement is as follows:

```
for (expression_1; expression_2; expression_3)
    {statement}
```

The parentheses enclose the three expressions separated by two semicolons. The first expression initializes any user parameters. It is done just once, when the loop first starts. The second expression is the test condition; it is evaluated before each potential execution of a loop. When the expression is false, the loop is terminated. The third expression, the change or update, is evaluated and executed at the end of each loop.

If **expression_2** is omitted, it assumes the default value of permanently true. This results in an infinite loop. It is a potential problem for CAPL to have infinite loops because CAPL programs are intended to intercommunicate with real-time ECUs on the CAN network.

Note: The **for** statement is equivalent to the following code:

```
expression 1;
while (expression 2)
{
    statement;
    expression 3;
}
```

The below example converts a decimal number into a binary array:

```
byte binaryArray[16];       // global variable

binary ( int number )       // user-defined CAPL function
{
    int index;
    index = 0;

    for ( ; number != 0; )
    {
            binaryArray[index++] = number % 10;
            number = number / 10;
    }
}
```

If the input is: **1234**
the output will be: **11010010**

Note: The statement **int index = 0;** will NOT work here, because local variables in CAPL are *static*, i.e., they are initialized only ONCE at the start and not each time that the function is called.

8.15.4 Unconditional Branching

8.15.4.1 Break Statement

At times we may want to use an *infinite loop*, that is, a loop that theoretically will be executed forever. Such a loop can be used as long as there is some other way to terminate it, such as throwing a switch in the testing lab or powering down the micro. It can be terminated in the code by using a **break** statement. The **break** statement is simply the word 'break' followed by the semicolon. Executing a **break** statement from within an enclosed **switch**, **do-while**, **while**, or **for** statement causes immediate termination to the loop.

The following is an example of **break** statement usage:

```
int x, y, z;

while ( x != 0 )
{
    doThis (x);
    if ( x == 0 )
            break;
    doThat (x);
}
doSomethingElse (y, z);
```

If **x** is equal to zero, program execution will skip the function **doThat(x)**, leave the **while** loop, and execute the function **doSomethingElse(y,z)**.

8.15.4.2 Continue Statement

A **continue** statement causes the program flow to go to the top of the enclosed **while**, **do-while**, or **for** loop, causing it to execute again. At the point that the **continue** statement is executed, any statements in the loop that appear *after* the **continue** statement are automatically skipped. This is the opposite of the **break** statement.

In the follwoing example, the code in the loop above the **continue** statement will only be repeated if the variable 'x' has the value '4':

```
int x, y, z;

while ( x != 0 )
{
    doThis (x);
    if ( x == 4 )
        continue;
    doThat (x);
}
doSomethingElse (y, z);
```

8.15.5 Return Statement

A loop can also be exited by using a **return** statement with no argument. Most of the time, the **return** statement is used in a user-defined function to return a value or value of an expression to an event procedure, as seen in the example below:

```
long Power(int x, int y)
{
    int i;
    long result;
    result = 1;

    for (i = 1; i <= y; i++)
        result *= x;

    return result;
}
```

The **Power()** function above returns a value **x** raised to the **y** power.

User-defined functions can return a value of any basic data type (for example, **int**, **float**, **long**, **double**, **char**, **byte**, or **word**).

Note: All event procedures do not return a value.

8.16 The "this" Keyword

Because CAPL provides event-driven program design, the **this** keyword is used to reference the object that triggered the event. The **this** keyword is used to reference data from the object or the object value itself, such as a message or an environment variable (CANoe only) within the corresponding event procedure.

The only event procedures that can use the **this** keyword are **on message**, **on envVar**, **on key**, **on errorframe** (only to get the CAN channel number), and the four CAN Controller events: **on busOff**, **on errorPassive**, **on errorActive**, and **on warningLimit**. The only time the **this** keyword is used in CAN Controller events is to access the transmitted and received error counts.

Although the **this** keyword may be conceptually difficult to understand, its use is straightforward. Think of it as a pointer to an object. For example, set up an event procedure to execute when a CAN message is received with ID 555 in decimal. The **this** pointer is then used to access the message's selectors or data bytes. (For more information, see Chapter 11 – Using Messages).

```
on message 555
{
    byte val;
    val = 0;

    /* checks to see if the message was received on CAN channel 1*/
    if (this.CAN == 1)
    {
    // sets "val" to the value of the first data byte in the CAN message
            val = this.byte(0);
    }
}
```

The **this** pointer can be similarly used for environment variables in CANoe. The event shown below is triggered when the value of the environment variable **LEDSwitch** changes. To read the new value, use the **this** keyword with the **getValue()** function, then store it into a variable, in this case **val**.

```
on envVar LEDSwitch      /* Triggered when LED Switch changes */
{
    int val;

    val = getValue(this);
}
```

Note: Changes made to whatever the **this** keyword replaced remain local within the current event procedure. The element value of the **this** pointer cannot be modified, but can be passed as a function parameter. The **this** keyword is reset after the event procedure terminates.

9 CAPL Functions

In addition to the traditional functions inherited from the C programming language, CAPL has additional functions that provide a variety of special-purpose operations useful in the CANalyzer or CANoe programming environment.

When the word "function" comes up in CAPL, it can be confusing because CAPL has broken functions into three categories. CAPL's internal library provides the first category of functions. This category of functions is built into the CAPL Browser, and each pre-defined function will be discussed later. The second category is user-defined functions. The third category is DLL functions, which require the user to implement a dynamic linked library. This topic will be further explained in the Chapter 24 – Introduction To CAPL DLLs.

9.1 User-defined Functions

The classical definition of a function is a section of program code that can accept 0 or more parameters (arguments) and which may return a value to the calling code upon completion. User-defined functions can return a value of any simple data type.

Do not confuse these user-defined functions with the pre-defined functions, even though their purpose is the same: a preprogrammed calculation that is carried out upon request.

In CAPL, a user-defined function is similar to a function defined in the C programming language. A function call is possible in any CAPL event procedure. These functions can contain any legal CAPL code, and are globally accessible. Putting frequently used code in a function makes programs more efficient and organized.

There is no need to use a declaration statement or function prototype for a user-defined function in CAPL. User-defined functions are created by highlighting **Function** in the **Events** window of the CAPL Browser and right-clicking to **New**. When a return type is not specified, the return type **void** is assumed. Once the function is implemented, you can call it in any event procedure. But be careful in naming the function. The function name must be unique compared to the pre-defined functions that come with CAPL (for example, you cannot define a function called **write()** regardless of the number of parameters).

9.2 Function Overloading is Allowed

CAPL has the ability to overload functions similar to the way C++ does. This means that multiple functions can have the same name but different parameter lists. The function that is called depends on the parameters that are passed; therefore, appropriate type coercions are performed automatically when compiled. Below is an example for printing:

```
void printme (double num)
{
    write("Floating point %f", num);
}

void printme (int num, char units[])
{
    write("%d %s", num, units);
}
```

The results of calling the functions are shown below:

printme(5.7); will print **"Floating point 5.700000"**

printme(3, "Feet"); will print **"3 Feet"**

printme(2.5, "angstroms"); will print **"2 angstroms"**

The last case is the most interesting. The first parameter to that function is expected to be an integer, but the compiler applies implicit type coercion to the floating point value 2.5, truncating the fractional part to an integer (2) before the argument is passed to the function.

9.3 Function Naming Conventions

CAPL uses a consistent and easy-to-read naming convention for its built-in functions. You may apply the following simple rules to your user-defined functions. These simple rules are:

- All standard C functions are in lower case (for example, **sin()**, **cos()**, **strlen()**, **strncat()**)
- Non-C one-word function names are in lower case (for example, **trigger()**, **outport()**, **inport()**)
- For non-C function names with more than one word, capitalize the first letter of all words except the first (for example, **swapInt()**, **timeDiff()**, **putValueToControl()**)

Although CAPL function names are not case sensitive, it is recommended that this convention be maintained in CAPL programs for ease in editing and readability.

9.4 CAPL Function Categories

Internally pre-defined CAPL functions are organized by categories. CAPL programs do not require function libraries or linking header files to use and compile these functions. Functions are added to every version of CANoe and CANalyzer, and some are obsolete because of changes in newer versions. Therefore, almost all the functions should be compatible with the latest version of CANoe, but older versions may not support all the functions. The CAPL function categories include the following:

- Mathematical functions
- User interface functions
- Time and timer functions
- Message handling functions
- Database functions
- Byte swapping functions
- File input and output functions
- Logging support functions
- String handling functions
- Measurement control functions
- Statistics functions
- CAN and port functions
- Replay block functions
- Environment variable and panel functions (for CANoe only)
- Language support and debugging functions

For detailed explanations and usage examples of each CAPL function, refer to the CAPL Function Reference Manual.

9.4.1 Mathematical Functions

Mathematical functions are used for complex calculations. This is a group of functions that also exist in the C language. One example of when this group of functions is used is when implementing a gateway, and the data being exchanged requires unit conversion.

Math Functions	
Function	**Purpose**
abs	Calculates an absolute value
cos	Calculates a cosine
exp	Calculates an exponent
random	Calculates a random value
sin	Calculates a sine
sqrt	Calculates a square root

Table 14 – Mathematical Functions

Below is an example of how mathematical functions are used:

```
double x;
x = cos(PI);          // returns –1, PI is a built-in constant

double tangent(double x)  // user-defined function tangent()
{
    return sin(x) / cos(x);
}
```

9.4.2 User Interface Functions

CAPL does not have many functions for the system interface. The most commonly used function is the **write()** function. Most of the time, this function is used for debugging by outputting text and values to the **Write** window of CANalyzer or CANoe. In order to categorize the information output from each node, the **Write** window can be adjusted to accommodate more panes, typically one for each node, by using the **writeClear()**, **writeCreate()**, **writeDestroy()**, **writeEx()**, and **writeLineEx()** functions. The color of text and background of the pane are set by the **writeTextColor()** and **writeTextBkgColor()** functions respectively.

User Interactions Functions	
Function	**Purpose**
beep	Makes the system beep (obsolete)
keypressed	Queries the currently pressed key
msgBeep	Makes a predefined Windows system sound
sysExit	Exits CANalyzer or CANoe
sysMinimize	Minimizes/restores CANalyzer or CANoe window
write	Outputs text to the **Write** window
writeClear	Clears a **Write** window
writeCreate	Creates a **Write** window
writeDestroy	Destroys a **Write** window
writeEx	Outputs text to a **Write** window without new line
writeLineEx	Outputs text to a **Write** window with new line
writeTextBkgColor	Sets the background color of a specific pane in the **Write** window
writeTextColor	Sets the text color of a specific pane in the **Write** window

Table 15 – User Interface Functions

Below is an example of sending a message when a key is pressed using the **write()** and **keypressed()** functions:

```
variables
{
    msTimer mytimer;          // timer
    message 100 msg;          // CAN message
}

on key F1
{
    setTimer(mytimer, 100);   // start 100 ms timer
    write("F1 pressed");      // output to Write window
```

```
on timer mytimer
{
    if (keypressed())          // true if any key is pressed
    {
            setTimer(mytimer, 100);          // restart timer
            output(msg);    // send message while key is pressed
    }
    else write("F1 released");
}
```

9.4.3 Time Functions

To learn more about timers, refer to Chapter14 – Using Timers.

When it comes to CAPL programming, you can define as many timers as you like. You only need to know how to declare two simple functions: **cancelTimer()** and **setTimer()**. One function cancels a specific timer if it is running, and the other set/reset a timer if it is not running or it has just expired. In most cases, timers are used to send periodic messages.

The timers you declare in CAPL can be in seconds (**Timer**) resolution or milliseconds (**msTimer**) resolution. After these timers have been set by the **setTimer()** function within an event procedure, the corresponding **on timer** event procedure is executed when the timer expires. If this is a periodic event, you can reset the timer with the **setTimer()** function at the end of the timer event procedure to make it recursive.

The **setTimer()** function takes two parameters: the name of the timer, and the length of time the timer will run before expiring. The parameter for the length of time will be expressed in different units, depending on what kind of timer is declared. The **cancelTimer()** function can be called to cancel a timer before it expires to prevent the timer event from triggering. Calling the **cancelTimer()** function has no affect if the timer is not set or has already expired.

The following example starts and resets a timer for 5 seconds:

```
variables
{
    Timer t1;
}

on start
{
    setTimer(t1, 5);          // Initializes a 5-second cyclic timer
}

on timer t1
{
    ...
    setTimer(t1, 5);          // Resets timer for another 5 seconds
}
```

In CAPL, programmers also need to know how the system clock operates. When a measurement starts, the system clock initializes. It works independently of the Window's timers, and it is incremented in 10 microsecond units (for example, time stamp = 12345 = 123.45 ms). To get the current system time, the **timeNow()** function is used. Every message, sent or received, will also have a system time stamp set by the CAN controller. This time stamp becomes important when you are performing network tests (node responsiveness, for example). To calculate the time difference between two messages, the **timeDiff()** function is used, or using the **TIME** selector (refer to Chapter 11 – Using Messages).

If you need to reference the Window's clock, the **getLocalTime()** and **getLocalTimeString()** functions are available.

For those projects where controlling processing time variation is important, CAPL provides two functions: **setDrift()** and **setJitter()**. The CAPL **setDrift()** function sets a constant deviation between –100% and +100% for all timers used in a network node. The **setJitter()** function, however, sets the fluctuation interval between –100% and +100%. Setting

either function will reset the other. To set both the drift and the jitter, use the **setJitter()** function only. For example, to set a jitter of ±5% and a drift of 12%, the minimum jitter is 7% and the maximum jitter is 17%. To recall the drift or jitter values, use the **getDrift()**, **getJitterMax()**, and **getJitterMin()** functions.

Note: If you are working on a network node in CANoe, you do not have to program these two parameters in CAPL. Right click on the network node in the **Simulation Setup** window and select `Configuration` to go to the `Extended` pane.

Time Functions	
Function	**Purpose**
cancelTimer	Stops an active timer
getLocalTime	Gets windows time information
getLocalTimeString	Gets windows time information in a string
setMsgTime	Assigns time to message (obsolete)
timeDiff	Calculates time difference between two messages
timeNow	Gets the current system time
setTimer	Sets a timer
CANoe Simulation Branch Only	
getDrift	Gets the deviation value set by setDrift()
getJitterMax	Gets the maximum deviation value set by setJitter()
getJitterMin	Gets the minimum deviation value set by setJitter()
setDrift	Sets the constant timer deviation
setJitter	Sets the timer deviation interval

Table 16 – Time and Timer Functions

9.4.4 Message Handling Functions

Message identifiers indicated are used by nodes to ascertain the origin, characteristics, and data contents of a message. When a message is received from the CAN bus, the user may use the **isExtId()** and **isStdId()** functions to check to see if the message has an 11-bit or 29-bit identifier. If a message has an 11-bit identifier but a 29-bit identifier is desired, use the **mkExtId()** function to convert it. Regardless of the type of message, the **valOfId()** function can be used to retrieve the message identifier, or by using the **ID** selector explained in Chapter 11 – Using Messages.

Whenever a message needs to be sent from a node, the **output()** function (as seen in Table 16) is used.

Message Handling Functions	
Function	**Purpose**
isExtId	Checks for extended identifier
isStdId	Checks for standard identifier
mkExtId	Creates extended identifier
output	Ouputs message, pg (specific to J1939), or error frame onto the bus
valOfId	Gets the value of a message identifier

Table 17 – CAN Message Handling Functions

Below is an example of how to get the message ID:

```
on message *                // the * symbol is a wildcard for all messages
{
    long id;
    id = valOfId(this);     // works with extended ID as well
}
```

9.4.5 Database Functions

Database Functions	
Function	**Purpose**
getFirstCANdbName	Gets the name of the first assigned database
getMessageAttrInt	Gets the value of a message attribute of type int only
getMessageName	Gets the corresponding message name from the database based on an identifier
getNextCANdbName	Gets the database name by specifying its assigned position

Table 18 – Database Functions

CANalyzer and CANoe have access to some database entities other than just messages, signals, and environment variables. The database names, as listed in Table 17, can be accessed by CAPL, and a message name specified in the database can be retrieved if its identifier is given. The **getMessageName()** function can also check for the inclusion of a received message in the specified database.

The **getMessageAttrInt()** performs the useful function of returning the value of an attribute of type integer from the database for a message. The function accesses the specified database and retrieves the attribute value, even if the value has been modified while a CANalyzer or CANoe measurement is running. If the value has been modified before a CANalyzer or CANoe measurement starts, then the recommended procedure to retrieve the value of such an attribute is by using the syntax <messagename>.<attributename>.

The example below checks whether an incoming CAN message is defined in the database.

```
on message *                // the * symbol is a wildcard for all messages
{
    char buffer[64];
    dword contextCAN = 0x00010000;

    if (getMessageName(this.ID, contextCAN | this.CAN, buffer, elcount(buffer)))
    {
            write("Message: %s", buffer);
    }
    else write("Message ID = %d is not defined.", this.id);
}
```

9.4.6 Byte Ordering Functions

Knowing the importance of using a database, these four functions are rarely used. Most of the time, these functions are used to convert data bytes from Intel format (Little-Endian) to Motorola format (Big-Endian) or vice versa to get the correct signal value. If a database is used, the format parameter has already been set up and CANalyzer or CANoe will automatically perform the format changes based on the database.

Byte Swapping Functions	
Function	**Purpose**
swapDword	Re-orders bytes of a double word(32-bit data type)
swapInt	Re-orders bytes of an integer (16-bit data type)
swapLong	Re-orders bytes of a long integer (32-bit data type)
swapWord	Re-orders bytes of a word (16-bit data type)

Table 19 – Byte Ordering Functions

9.4.7 Logging Functions

Logging Functions	
Function	**Purpose**
setLogFileName	Sets the name of the log file
setPostTrigger	Sets the posttrigger time for logging
setPreTrigger	Sets the pretrigger time for logging
startLogging	Triggers a Logging block to start logging
stopLogging	Triggers a Logging block to stop logging
trigger	Starts all Logging blocks to a log
writeToLog	Writes a formatted string to the log file with time stamp
writeToLogEx	Writes a formatted string to the log file without time stamp

Table 20 – Logging Functions

CAPL also supports log file triggering and writing to a log file. The functions used for these tasks are shown above in Table 19. All of the logging-related functions can be used in the Analysis Branch of both CANalyzer and CANoe, but only the last five functions can be used in the Simulation Branch.

Note: To use these functions, logging must be enabled in the Analysis Branch (double left-click on the black square to the left of the Logging block to remove the break). Also, ensure that the configuration of triggering in the logging block is set to CAPL (to verify, right-click on the Logging block and select `Configuration`).

The **startLogging()** and **stopLogging()** functions bypass all logging triggering conditions, and they are used to start and stop logging. You also have choices to specify the Logging blocks to trigger and to set up either the pre-trigger time or post-trigger time passed in each function to overwrite the settings.

Below are two examples of how these functions are used:

```
on message 100    // this example triggers all the Logging blocks
{
    writeToLog("Logging starts");

    // this example sets the pre-trigger time before logging to 25 milliseconds:
    setPreTrigger(25);
    trigger();                  // start logging
}

on key *
{
```

```
    // starts the Logging block "Logging 1" with pre-trigger time of 2000 milliseconds.
    startLogging( "Logging 1", 2000);
}
```

9.4.8 String Handling Functions

Strings are not often used in CAPL. These functions are mostly used to format a string to be output to the **Write** window and when CAPL is performing file I/O. Table 20 lists the string handling functions.

String Handling Functions	
Function	**Purpose**
atol	Converts a string to a long integer
ltoa	Converts a number to a string
snprintf	Creates a formatted string
strlen	Gets the length of a string
strncat	Concatenates two strings
strncmp	Compares two strings
strncpy	Copies a string

Table 21 – String Handling Functions

In the example below, **atol()** converts a string to a **long** number so that digits in the string can be used as characters for numeric calculations. The number base is decimal.

```
long z1;
long z2;

z1 = atol("200");
z2 = atol("0xFF");

// Result: z1 = 200, z2 = 255
```

9.4.9 Measurement Control Functions

Measurement control functions are used to control the flow of a measurement.

Measurement Control Functions	
Function	**Purpose**
canOffline	Disconnects node from bus (CANoe's **Simulation Setup** window only)
canOnline	Reconnects node to the bus (CANoe's **Simulation Setup** window only)
getBusContext	Returns current bus context (CANoe's **Simulation Setup** window only)
getBusNameContext	Returns a specific bus context (CANoe's **Simulation Setup** window only)
getStartdelay	Gets delay time value of current node (CANoe's **Simulation Setup** window only)
halt	Halts the simulation (only possible in the Simulated Mode of CANoe)
inspect	Updates the variables in the *Inspect* pane of the **Write** window (CANoe only)
setBusContext	Sets new context of current bus (CANoe's **Simulation Setup** window only)
setStartdelay	Sets delay time value of present node (CANoe's **Simulation Setup** window only)
stop	Stops the measurement

Table 22 – Measurement Control Functions

In this category, the function used most frequently is probably the **stop()** function. Calling the **stop()** function anywhere and at any time during CAPL execution will stop the CANalyzer or CANoe measurement.

If a network node in CANoe needs to temporarily stop executing, the **canOffline()** and **canOnline()** functions are used. **CanOffline()** stops the node from transmitting messages onto the CAN bus. In this passive state, all the events will still trigger if the condition is right. All changes made during this passive state become effective once the network node becomes online again by calling the function **canOnline()**.

A network node can also be set up to have a delay period right after a measurement starts. This can be set up in the configuration of that network node in the **Simulation Setup** window. This can also be done in CAPL by calling the **setStartDelay()** function (only works in the preStart event). The delay time can be returned by the **getStartDelay()** function.

Both the **halt()** and **inspect()** functions are used to debug a CAPL program in a simulated mode (CANoe only). The **halt()** function halts the simulation immediately when called and updates the *Call Stack* pane of the **Write** window automatically. To resume the simulation, the user can press the 'F9' key or click on the "lightning bolt" icon on the taskbar of CANoe. Whenever the variables need to be updated in the *Inspect* pane, the **inspect()** function can be called in any event procedure/function of a CAPL program.

The following example is a timer event procedure used to update the *Inspect* pane of the **Write** window every 100ms. This is one strategy to update the window that does not update automatically.

```
on timer updateTimer
{
    inspect();
    setTimer(updateTimer, 100);
}
```

The remaining measurement control functions are used in modeling gateway nodes. By definition, a gateway is a piece of software or hardware that enables communication between two network buses that are the same or different to some extend. The difference in the network buses may just be the bus rate, or to something more unique, a different communication protocol. For CANoe, these communication protocols are implemented using DLLs call node layer DLLs. To have one CAPL program (CANoe only) to model gateway behavior between two different communication protocols is easy because there are two different node layer DLLs. But, to model gateway behavior that uses the same communication protocol, we have to introduce what is called a bus context because the same node layer DLL is used.

A distinction must be made between the instances of the node layer, both for calls to CAPL functions that are implemented in the node layer DLL and for implementing callback functions. To facilitate this distinction, a bus context is placed in the CAPL program by the runtime environment while a callback is being executed by the node layer DLL. This context unambiguously identifies the node layer that is making the call. In a similar manner, the call of a CAPL function that is implemented in a node layer is forwarded on to the appropriate node layer, depending on the current bus context

```
dword oldValue, newContextValue;

// previous context value is stored in oldValue
oldValue = setBusContext(newContextValue);
```

9.4.10 Statistics Functions

Statistics Functions	
Function	**Purpose**
isStatisticAcquisitionRunning	Checks to see if acquisition range is running
startStatisticAcquisition	Sets new acquisition range
stopStatisticAcquisition	Stops acquisition range

Table 23 – Statistics Functions

To use the statistics functions in CAPL, the CAPL program must be placed right before the Statistics block in the **Measurement Setup** window. These functions, as shown in Table 22, are only in effect if the configuration of the Statistics block is set to active for statistics report and histogram evaluations. In the CAPL program, these functions are called to start and stop the data acquisition.

The following example begins generating a statistical report when an error frame is received.

```
on errorframe
{
    startStatisticAcquisition();
}
```

9.4.11 CAN Protocol Functions

Vector has many hardware interface cards and cables (known as CANcabs) to connect the software to the physical CAN bus. Often, a particular card and cable will be more suitable for the characteristics of a particular CAN bus. If the wrong hardware is used, there could be no CAN communication. Software like CANalyzer and CANoe recognize these hardware interface cards and cables, and allow the user to adjust the baud rate settings, sampling point percentage, and other important protocol control settings.

CAN Protocol Functions	
Function	**Purpose**
canSetChannelAcc	Sets channel acceptance
canSetChannelMode	Activates/deactivates TX and TXRQ display of a channel
canSetChannelOutput	Activates/deactivates acknowledgement bit
getCardType	Determines the type of CAN platform being used
getChipType	Determines the type of CAN controller being used
resetCAN	Resets all CAN Controllers
resetCANEx	Resets a specific CAN Controller
setBtr	Sets the baud rate for a particular channel in the Bit Timing Register (BTR)
setCanCabsMode	Sets a mode for a CANcab
setOcr	Sets the Output Control Register

Table 24 – CAN Protocol Functions

In some cases, the hardware settings may require a change while the software measurement is running. CAPL has built-in functions to determine the type of hardware being used and to configure the hardware settings. The type of hardware interface cards accessed by CANalyzer or CANoe (CANcardXL, CAN-AC2-PCI, and so on) is returned by the **getCardType()** function and the CAN controllers (Philips PCA82C200, INTEL 82526, and so on) identified by the **getChipType()** function. To change the acceptance filter on the card (to accept only certain messages), you would call the **canSetChannelAcc()** function. Blocking messages only makes sense if you do not want the application to process those messages. The blocked messages will still be acknowledged by the controller. To disable acknowledgement and make a CAN channel like a spy on the bus, use the **canSetChannelOutput()** function.

For applications implemented to test a node or multiple nodes on a CAN bus, numerous error frames may be encountered. Eventually, the CAN controller on the interface card could go into the Bus Off condition if an excess number of error frames are encountered (see Chapter 15 to learn more about this condition). To reset the CAN Controller due to error frames, you can use either the **resetCAN()** or **resetCANEx()** function. CAN interface cards, such as the AC2-PCI, may first require invoking the **setBtr()** and **setOcr()** functions before the controller is reset. The **setBtr()** function sets a new baud rate on a CAN channel.

Some Vector CANcabs enclosing a specific CAN transceiver support network sleep / wakeup capability and transitions from high-speed mode to low-speed mode, or vice versa. For example, if a module requires so-called 'high-voltage wakeup' to enable communication, the CANcab has to be set to a high-voltage mode before transmitting a message. The function that makes these mode transition possible is the **setCanCabsMode()** function.

9.4.12 Port Functions

CAPL has only four built-in parallel port functions for communication: **inport()**, **outport()**, **inportLPT()**, and **outportLPT()**. The latter two are new and have better connecting mechanisms. All four functions require special setup for NT users (NT 4.0 and Windows 2000). Refer to the ReadMe.txt file in the **…EXEC32\GploDrv** directory of CANalyzer or CANoe for help. Table 23 below lists the CAN and Port functions.

Port Functions	
Function	**Purpose**
inport	Reads a byte from a port
inportLPT	Reads a byte from a parallel port
outport	Sends a byte to a port
outportLPT	Writes a byte to a parallel port

Table 25 – Port Functions

Below is an example of a port function:

```
...
val = inport(0x3f8);          // reads a byte from Port 0x3f8
...
```

The following example sets the CAN Controller's Output Control Register (OCR). The values do not take effect until the next time the function **resetCan()** is called.

```
// set OCR Register, CANcardX or XL does not require this function call
setOcr(0, 0x02);
resetCan();          // reset CAN Controllers
```

9.4.13 Replay Block Functions

Replay Block Functions	
Function	**Purpose**
ReplayResume	Resumes replaying log file after it was suspended
ReplayStart	Starts replaying log file from the beginning
ReplayState	Returns the state the Replay block is in
ReplayStop	Stops replaying log file
ReplaySuspend	Suspends replaying log file

Table 26 – Replay Block Functions

CAPL has access to the Replay blocks in both the **Simulation Setup** and **Measurement Setup** window. By calling the functions above, the user can start, stop, resume, or suspend a log file from replaying and check a Replay block's state. This can only be done if the Replay block is configured to trigger by CAPL. The Replay Block functions are listed above in Table 24.

The following example illustrates how a Replay block named “ibus_data” is used to replay when the ‘s’ key is pressed.

```
on key 's'
{
    char replayName[32] = "ibus_data";
    replayStart(replayName);
}
```

9.4.14 Environment Variable and Panel Functions (CANoe only)

The user-defined panels associated with CANoe used primarily to communicate with the CAPL program using environment variables. These environment variables are accessed by three functions: **getValue()**, **getValueSize()**, and **putValue()**. The **putValueToControl()** function is used to assign a value to the multi display control element on a panel without using an environment variable. The type of values to be displayed in the multi-display control element include integer, float, text, and the message’s data bytes. The type of messages supported includes CAN, LIN, VAN, BEAN, and J1939 PGNs.

If a particular object or element in a panel needs to be enabled or disabled, use the **enableControl()** function. This function has access to panel help elements, panel recorder elements, panel control buttons, and other elements with association to either an environment variable or signal. The elements’ color can be set using the **setControlBackColor()**, **setControlForeColor()**, and **makeRGB()** functions.

If all environment variables need to be initialized before the start of a measurement in CANoe, use the **callAllOnEnvVar()** function. Calling this function in a CAPL program triggers all the environment variable events in that CAPL program to execute. The CANoe Environment Variables and Panels functions are listed below in table 25.

CANoe Environment Variables and Panels Functions	
Function	**Purpose**
callAllOnEnvVar	Calls all environment variable procedures
enableControl	Enables or disable an element on a panel
getValue	Reads the value of an environment variable
getValueSize	Gets the size of an environment variable in bytes
makeRGB	Sets primary color values
putValue	Assigns a value to an environment variable
putValueToControl	Assigns values to multi display control in panels
setControlBackColor	Sets the background color of a panel element
setControlForeColor	Sets the foreground color of a panel element
setControlProperty	Sets a property of an ActiveX control

Table 27 – Environment Variable Functions

Below are some examples of how to get environment variable values:

```
int val;
float fval;
char buff[25];

// Assign the value of the environment variable “Switch” to val
val = getValue(Switch);

// Assign the value of the environment variable “Temp” to fval
fval = getValue(Temp);
```

```
// Read the value of environment variable “NodeName”
// val in this case is the number of bytes returned
val = getValue(NodeName, buff);
```

9.4.15 Miscellaneous Functions

This group of functions is essential to give additional support for your CAPL program, but are probably the least—often used. The **fileName()** function is used to display which node is active during a simulation in the **Write** window. This is helpful if you are simulating many network nodes. The **runError()** function outputs an error string to the **Write** window and stops the measurement. These errors are intentionally induced by the user in a CAPL program, and they are configurable. The last two functions are used to set and output string messages to the **Write** window based on priorities ranging between 0 and 15. For example, if a node is set to a write priority of 5, only those **WriteDbgLevel()** functions with a priority of 5 or lower can output their string message to the **Write** window.

Table 26 lists the Miscellaneous functions.

Miscellaneous Support	
Function	**Purpose**
elCount	Counts the number of elements in an array
FileName	Outputs the program’s file name in the **Write** window
RunError	Triggers a run-time error
SetWriteDbgLevel	Sets write priority for present node
WriteDbgLevel	Outputs a message to the **Write** window with specific priority

Table 28 – Miscellaneous Functions

CAPL does not allow traditional C pointers or memory allocation functions such as **malloc()** and **free()**. Array dimensions must be specified in the array declaration. However, it is possible to pass arrays of arbitrary size as function parameters using the **elCount()** function. The **elCount()** function returns the number of elements in a dimension of an array. For multidimensional arrays, **elCount()** can be used separately on each dimension.

The example below shows how to implement a **sum()** function that returns the sum of every element in a two-dimensional array of arbitrary size:

```
variables
{
    int mat1[2][2] = {{1,2}, {3,4}};
    int mat2[3][3] = {{1,2,3}, {4,5,6}, {7,8,9}};
}

void sum(int matrix[][])
{
    int i, j;
    int answer;

    answer = 0;
    for (i = 0; i < elCount(matrix); i++)
    {
            for (j = 0; j < elCount(matrix[i]); j++)
            {
                    answer += matrix[i][j];
            }
    }

    write(“The sum is %d”, answer);
}
```

The example below sets the priority level for the **writeDbgLevel()** function and outputs a message to the **Write** window with the specified priority.

```
int i = 10;
int j = 12;

setWriteDbgLevel(7);                    // sets the write priority for this node

writeDbgLevel(4, "This is shown: h = %lxh", j);
// display in the Write window: This is shown: h = ch

writeDbgLevel(9, "This is not shown: d = %ld", i);      // no output
```

9.5 CAPL Function Compatibilities

Not all CAPL functions are available in all P Blocks or Network Nodes.

Based on the speed of currently available PCs, some CAPL functions are too slow to be used in CANalyzer's Transmit Branch or CANoe's Simulation Branch for some real-time activities. The **getLocalTime()** function used to get the Window's clock is an example of a 'slow' function. It is advisable to have the **getLocalTime()** function available only in the Analysis Branch for tracking data evaluations and logging.

Because the CAPL Browser cannot tell where the P Block is placed in the setup window of CANalyzer or CANoe, it will not always detect whether some of the functions in the program are allowed in that branch (for example, **seqFile…()** functions). If such file I/O functions are used, compile the CAPL program using the compile option in CANalyzer or CANoe's main menu. This action provides the compiler with the location of the P Block, allowing it to recognize restricted function calls.

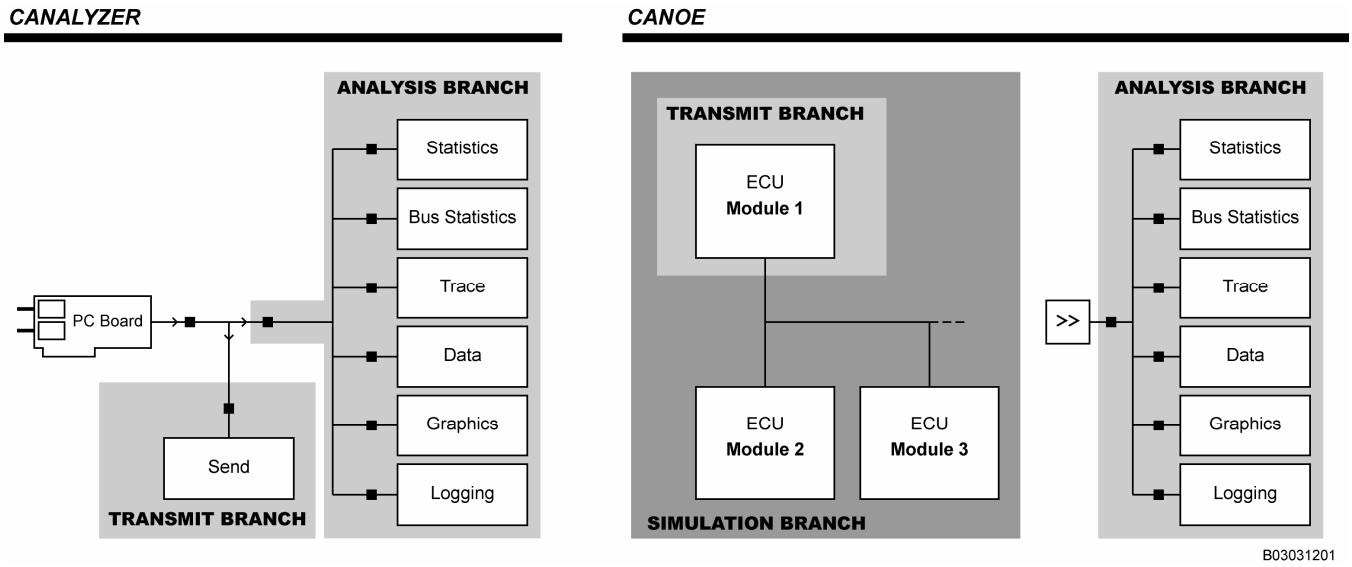

Figure 34 – CAPL Functions Depend on Placement

Finally, a CAPL function may not work in your CAPL program because the function was not introduced until a newer version of CANalyzer or CANoe. These functions are either available through version upgrade or service packs.

To see if a function has these two limitations, refer to the CAPL Function Reference Manual.

10 CAPL Events and Event Procedures

A CAPL program is organized around event procedures. Each event procedure is associated with a single event. Whenever a predefined CAPL event occurs, the corresponding event procedure will execute.

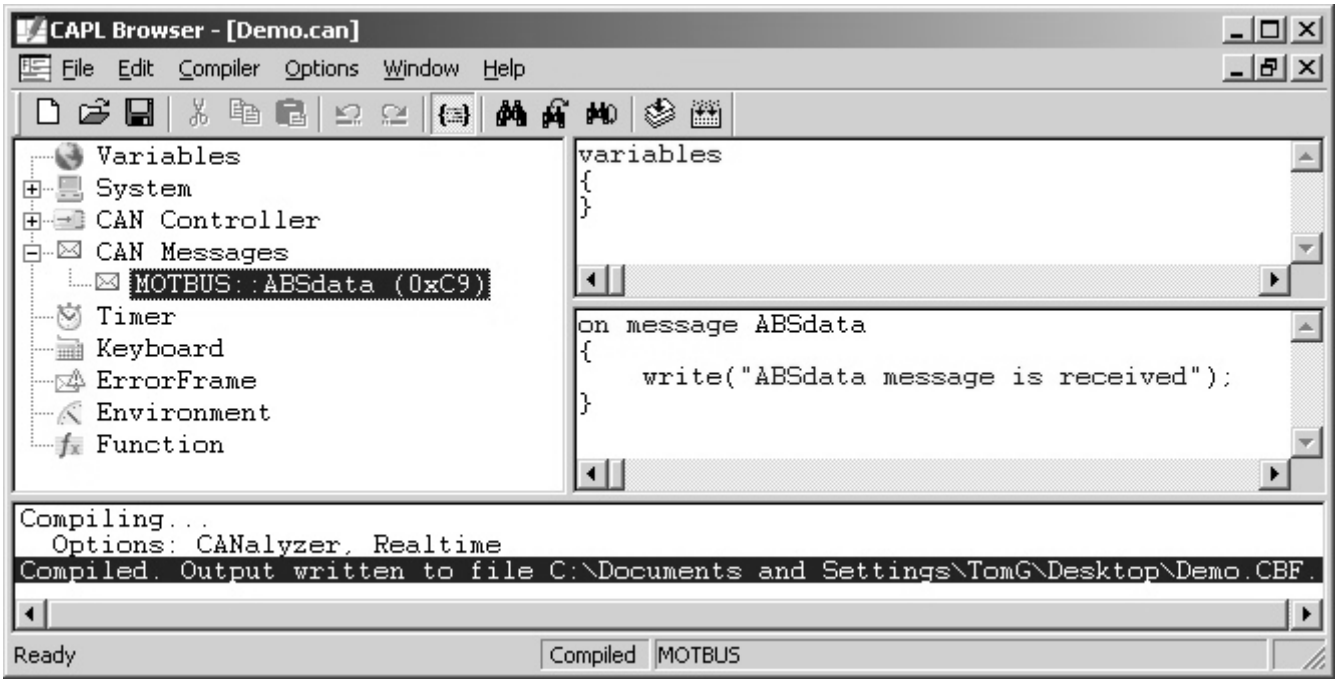

Figure 35 – A CAPL Event Procedure

Events and their corresponding event procedures are grouped into functional classifications to simplify program organization. These same functional classifications also establish the upper level structure of the CAPL Browser and its graphical user interface.

CAPL event procedures are functionally classified as follows:

- Message events
- Timer events
- Keyboard events
- Error frame events
- CAN controller events
- System (tool) events
- Environment variable events (CANoe only)

Each of these functional classifications can be selected from the tree view at the left in the CAPL Browser (see Figure 35).

Table 29 shows a list of recognizable CAPL events and the corresponding procedures that will be executed.

CAPL Event	Description	Procedure Executed
Timer	The named timer expires (timeout)	on timer *name*[1]
Keyboard	The named key is pressed	on key *name*[2]
CAN Message	The named message is received	on message *name*[3]
Error Frame	An Error Frame is detected	on errorFrame[4]
Bus Off detected	CAN Controller - Bus Off is detected	on busOff[5]

Error Active detected	CAN Controller - Error Active is detected	on errorActive[5]
Error Passive detected	CAN Controller - Error Passive is detected	on errorPassive[5]
Warning Limit detected	CAN Controller - Warning Limit is detected	on warningLimit[5]
Initialization	System - initialized	on preStart[6]
Start	System - start	on start[6]
Stop	System - stop	on stopMeasurement[6]
Function	Invocation of the user's procedure	user-defined function[7]
Environment Variable	Change in environment variable (CANoe only)	on envVar *name*[8]

Table 29 – CAPL Events and Corresponding Event Procedures

Notes:
1. Each timer has a unique name.
2. Each key has a unique name.
3. Each message has a unique name or identifier.
4. This is a detected invalid CAN transfer.
5. This is a CAN Controller internal state.
6. This is a system tool (CANalyzer or CANoe)-internal state.
7. Any number of user-defined functions is allowed.
8. Any number of user-defined CANoe-based environment variables is allowed.

10.1 Creating an Event Procedure

There is no need to use a declaration statement for any type of event. Adding code to the corresponding event procedure is all that is necessary. Simply right-click on the event and select **New** to initialize a new event procedure (see Figure 36).

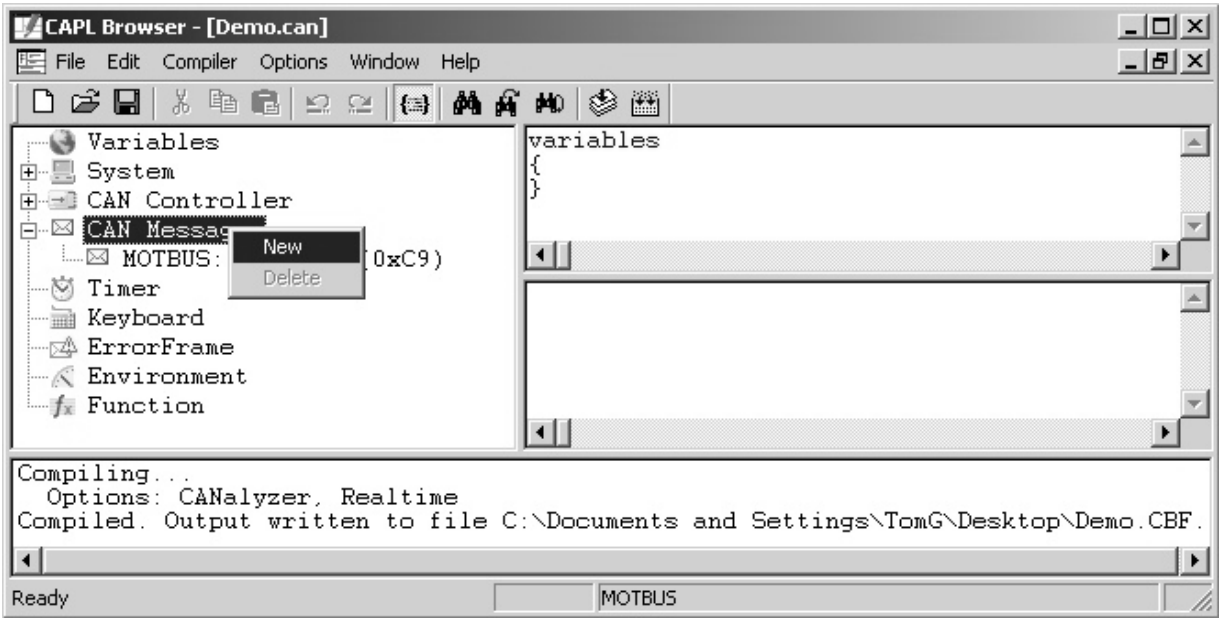

Figure 36 – Creating a New Event Procedure

10.2 Event Procedure Requirements

Whenever a new event procedure is defined, additional CAPL code is usually implemented to carry out a task. Having an event defined is of no use if the event procedure is not implemented, as the event will trigger with no process to execute.

Some event procedures require a specific parameter to define the event. For example, a keyboard input event triggers whenever a key is pressed – the desired key has to be specifically defined for the event procedure. If a keyboard input event has not been defined for that key, nothing will happen when that key is pressed.

10.3 Some Event Procedures Use the Keyword “this”

The keyword **this** is only used on a number of CAPL event procedures. The keyword **this** acts like a pointer, and essentially points at data related to the current event where the keyword is used. Table 30 shows the CAPL event procedures which use the keyword **this** and how they are used. Refer to their corresponding chapters for more details.

CAPL Event	Event Procedure	Value of "this"
Timer	on timer *name*	not used
Keyboard	on key *name*	the key pressed
CAN Message	on message *name*	the message received
Error Frame	on errorFrame	CAN channel
Bus Off detected	on busOff	error counts
Error Active detected	on errorActive	error counts
Error Passive detected	on errorPassive	error counts
Warning Limit detected	on warningLimit	error counts
Initialization	on preStart	not used
Start	on start	not used
Stop	on stopMeasurement	not used
Environment Variable	on envVar *name*	the value of the environment variable

Table 30 – Event Procedures That Use the Keyword “this”

Because the **this** keyword can be used in many events, what it references will be different in every event, including the same type of event (such as two **on key** events). This is simply because once an event is done executing, the **this** keyword is initialized to null. If another event has used the **this** keyword, it will have a new reference.

You do not have to reference and release every time you use the keyword. It automatically references the event that the keyword is used in, and release once the event is done.

Note: Any changes you make to what the **this** keyword references are only valid within that event procedure. In other words, you can only use what it has referenced, but you can't assign a new value to it.

10.4 The * symbol

Sometimes the same task applies to more than one message received from the bus or to different keystrokes on the keyboard. When this condition occurs, the first answer most people think of is to define a common subroutine so the message events or keyboard events can call that subroutine to perform the task. In CAPL, there is another solution. The ***** symbol is a wildcard that can be used to replace the key parameter for both the **on key** and **on message** event procedures. If the same action is performed for two or more keys, this symbol is used because it represents all key presses. All incoming messages with similar tasks may use this symbol as well.

Below is an example using the ***** symbol.

```
on key *     // for all keys
{
            // print the key that was pressed
    write(“The key you have pressed is %c”, this);
}
```

10.5 Event Priority

Events do not have any priority whatsoever in terms of execution. They execute whenever the condition is right and do not stop in the middle of execution. In other words, events cannot be nested. Only one type of event can be processed at a time. If a function needs to be called within an event procedure, that function will execute first before the event procedure continues. If an event is executing and it is urgent to stop in the middle of execution, the **stop()** function is used. Calling the **stop()** function will terminate the measurement in either CANalyzer or CANoe. If CANoe is running in a simulated mode (not interfacing with the physical bus), the **halt()** function may be used to halt all processes. To continue, press the 'F9' key on the keyboard or click on the "lightning bolt".

Note: The **halt()** function will only execute in simulation mode. It will not execute while you have live access to a physical CAN bus.

Because the * symbol can be used as a wildcard in both **Keyboard** and **Message** event procedures, there is a level of priority for which event to execute. Not all keys or incoming messages usually execute the same task. If all keys except one execute the same task, then two **on key** event procedures have to be defined. Below is an example:

```
on key *              // for all keys other than 'a'
{
    // print the key that was pressed
    write("The key you have pressed is %c", this);
}

on key 'a'            // for the lowercase 'a' key only
{
    // print the 'a' key
    write("The 'a' key is pressed");
}
```

Pressing the 'a' key will only execute the **on key 'a'** event procedure because that event procedure is more specific to the **Keyboard** event. Every other key will execute the first event procedure.

Priority is no exception for incoming messages. Below is an example:

```
on message *                // for all messages
{
    // print
    write("The message received is %c", this.id);
}

on message CAN1.*           // for all messages received on CAN channel 1
{
    // print
    write("The message received on CAN channel 1 is %c", this.id);
}

on message 0x100            // for messages with ID equal to 100 in hex only
{
    // print
    write("The message with ID 0x100 is received.");
}

on message CAN2.0x100   // for messages with ID equal to 0x100 from CAN channel 2
{
    // print
    write("The message with ID 0x100 is received from CAN channel 2.");
}
```

When it comes to priority, message event definitions are more involved than Keyboard events. With the addition of CAN channel-specific Message events, there are two rules CAPL follows to determine which event procedure to execute:

- A specified CAN channel has precedence over the absence of a CAN channel.
- A specified message ID number has precedence over a * symbol wildcard.

Following the two rules in our examples above, if message 0x100 is received on CAN channel 1 it would print, **“The message with ID 0x100 is received”**. If that message is received on CAN channel 2, it would print, **“The message with ID 0x100 is received from CAN channel 2”**.

11 Using Messages

Within the CAPL programming language, a message is a block of data transmitted over the CAN bus in one or more data frames. CAPL functions with three of the four different CAN frame types (listed below):

- Data
- Error
- Remote

Overload frames are not supported by the CAPL programming language.

Note: For more details about CAN frame types see Section 26.8.

Note: In the case where the block of data (sometimes referred to a packet) is larger than one CAN data frame, a transport protocol is usually implemented to handle this multi-packet message.This text concentrates on single-framed messages only.

11.1 Messages in CAPL

Because it is used in almost all node simulations, the message object is never to be overlooked when programming in CAPL. This object, which is essentially a structure (as in C and C++), allows users to work with a CAN message at a higher, more abstract level if it has already been defined in a database. Instead of assigning the data field of a message bit-by-bit, the data field can be split and set in terms of signals, the part of the message that is used to store data. Several combinations are possible to fill the 64 bits of data. For example, a CAN message could carry four 8-bit signals and a 32-bit signal, sixteen 1-bit signals and three sixteen-bit signals, and so on.

When handling messages in CAPL, be sure that you know which messages are used to send data and which messages are received to extract data. Any valid CAN transfer unit is a received message. Message events are ONLY used for message reception, since sending a message can take place within any event. Any messages that require transmission must be first declared with the **message** data type.

All messages have a unique numeric identifier. The message identifier is entered as a number (integer) in decimal or hexadecimal notation to identify the message. To make a message global to all the events and functions, declare it in the **Variables** section of the program.

11.1.1 Declaring Messages in CAPL with a Database

Messages in CAPL are objects; therefore, they must be declared (similar to variables) before they can be used. Declaring a message is simple, defining one takes a little bit of effort. Defining a message is commonly done in a database using a database editor like CANdb++. The database editor also allows message associations to be defined (for example, message to signals, message to attributes, and message to network nodes). If the database-defined message is to be used in a CAPL program, you do not have to define it again. Simply declare it with the following syntax:

Syntax: **message <message identifier or name> <variable name>**

It is easy to declare messages with the help of a database. As mentioned in the database sections of this book, messages are given a symbolic identifier when defined. This symbolic identifier can be used anywhere in a CAPL program where a numeric identifier is used, especially when declaring a message. When declaring a message already defined in the database, CAPL automatically knows structural information like the numeric identifier, data length code (DLC), and signal list.

Below are examples for declaring a message called **EngineTemp** already defined in the database:

```
message EngineTemp eTemp;
// or use its numeric identifier
message 0x100 eTemp;

// two instances of the same message
message EngineTemp eTemp1, eTemp2;
```

Note: The message identifier is case sensitive. Rather than typing it in, right-click on the location and select **CANdb message...** to get the message identifier.

11.1.2 Declaring Messages in CAPL Without a Database

If the messages are not defined in the database, you usually have to spend more time programming. For example, CANalyzer and CANoe, by default, receive a message in the Intel (little-endian) format. If the data bytes in a message use the Motorola (big-endian) format, you have to swap the bytes using CAPL. But that's not all - message properties and associations have to be defined in the CAPL program. Without the defined message in a database, you can only declare the message with a numeric identifier. Below is an example:

```
message 0x100 eTemp;
```

Unlike the example in the previous section, this message has no data bytes because it is not defined nor declared with a DLC, and there is no signal list because it is not defined in a database. To assign a DLC to the message, you can do it one of two ways: when declaring the message or by assignment using the DLC selector.

```
// when declaring the message:
message 0x100 eTemp = { DLC = 8 };

// after declaring the message:
eTemp.DLC = 8;
```

If you are trying to define an extended message with a 29-bit identifier, an "x" is appended to the suffix of the numeric identifier as seen below:

```
// declaring a 29-bit identifier message:
message 0x100x eTemp = { DLC = 8 };
```

If you are unsure about what the identifier is, replace the identifier with an "*". You will have to establish the identifier later with the ID selector before sending the message.

```
message * eTemp = { DLC = 8};

// in another part of the program, establish the identifier:
eTemp.ID = 0x100;
```

11.1.3 Message Selectors

The message object not only contains information about its signals, but also about some properties and components called selectors. These selectors are accessible in both transmitted and received messages. For most applications, they do not need to be explicitly set, but they can be helpful in some cases. For example, the **TIME** selector can be used in arbitration testings. To change the transmit channel from 1 (default) to 2, the **CAN** selector is used. To find out whether the message is a transmitted or received message, the **DIR** selector is used. See Table 29 below for CAPL Message Selectors

CAPL Message Selectors		
Selector	**Description**	**Valid Values**
ID	Message identifier	Any valid CAN message ID
CAN	Transmit Channel number	1 or 2 (depends on number of CAN controller)
DLC	Data Length Code	0 to 8 data bytes
DIR	Direction of transmission	RX (Receive) TX (Transmit) TXREQUEST (Transmit Request)
RTR	Remote Transmission Request	0 (not an RTR) 1 (RTR)
TYPE	Combination of DIR and RTR	See below
TIME	Time stamp of the message in units of 10ms (read-only)	Long integer
SIMULATED	Sent by a simulated node (read-only)	0 (real node) 1 (simulated node)

Table 31 – CAPL Message Selectors

Both the **DIR** and **TYPE** selectors provide predefined constants that can be used to make CAPL programs more readable. More importantly, these constants are not case-sensitive. The **TYPE** selector is provided as an efficient way to evaluate other selectors as one unit.

```
TYPE = (RTR << 8)  |  DIR
```

The constants accessed with the **TYPE** selector are listed below.

Valid Values for the TYPE Message Selector		
Constant	**Meaning**	**Logical Derivation**
RXREMOTE	Remote message was received	((DIR == RX) && RTR)
TXREMOTE	Remote message was transmitted	((DIR == TX) && RTR)
TXREQUESTREMOTE	Transmission request was set for remote message	((DIR == TXREQUEST) && RTR)
RXDATA	Data message was received	((DIR == RX) && !RTR)
TXDATA	Data message was transmitted	((DIR == TX) && !RTR)
TXREQUESTDATA	Transmission request was set for data message	((DIR == TXREQUEST) && !RTR)

Table 32 – Valid Message Selector Values

To set or retrieve the value of a selector, type the name of the message variable follow by a period and then the selector. Below are some examples.

```
messagename.CAN = 2;          // set the transmit channel for this message to 2
value = messagename.TIME;     // retrieve the time stamp of a message's last transmission or reception
messagename.ID = 100;         // set the message identifier to 100 in decimal
```

11.2 Accessing Data

CAPL provides two ways to access the data field in a message. Using signals defined in the database to access data in a message is as simple as assigning a value to a variable. However, not everyone has a database defined. If you do not have a database, the following method can be used for message access.

CAPL provides four keywords to access data: **BYTE** (8 bits), **WORD** (16 bits), **LONG** (32 bits), and **DWORD** (32 bits). **LONG** and **DWORD** can be used reciprocally. The keywords are used after a message variable followed by a period. In addition, a parameter is used for each keyword to specify the byte location, just like specifying an index for an array. For **WORD**, **LONG**, and **DWORD**, the parameter specifies the position of the first byte, therefore the parameter cannot exceed 7 for **BYTE**, 6 for **WORD**, and 4 for **LONG** and **DWORD** since a maximum of 8 bytes is allowed in a message.

Note: Both CANalyzer and CANoe see data in Intel format, so CAPL uses the Intel format for data access. In other words, the first byte is the least significant byte whereas the last byte is the most significant byte (little-endian). If the data is in Motorola format (big-endian), then the bytes have to be swapped.

Shown in the illustration below is a message received from the bus with the data in Intel format:

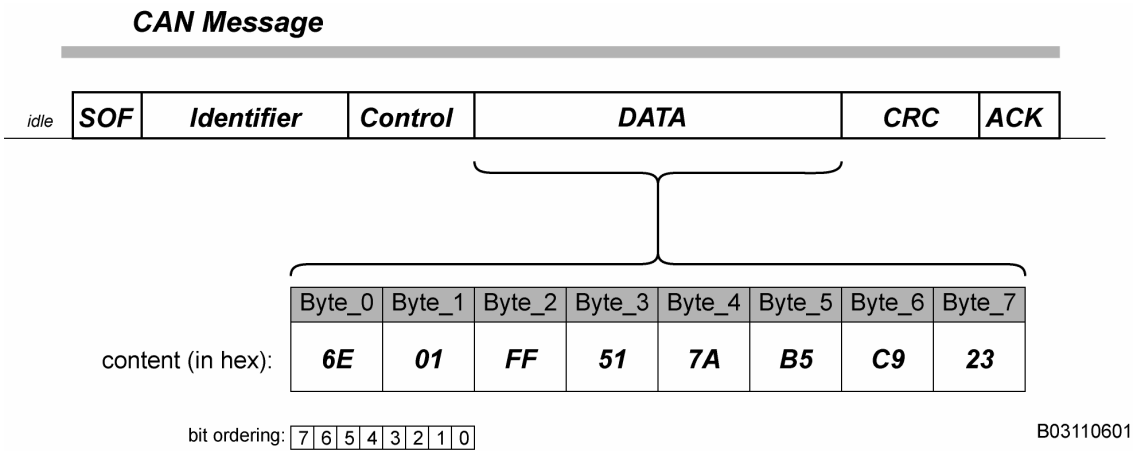

Figure 37 – Message with Data in Intel Format

Assignments	Results (in hex)
Messagename.BYTE(0)	0x6E
Messagename.BYTE(5)	0xB5
Messagename.WORD(0)	0x16E
Messagename.WORD(3)	0x7A51
Messagename.LONG(0)	0x51FF016E
Messagename.LONG(2)	0xB57A51FF
Messagename.LONG(4)	0x23C9B57A
Messagename.LONG(6)	Invalid

Table 33 – Assignment Results with Data in Intel Format

In reality, using the four keywords is not very productive, since data can only be accessed in terms of bytes. If the value you want is not perfectly stored within a byte or bytes, you would have to use operators. Using the message example in Figure 37, let's say the value you want is 0x1F at Bytes 2 and 3. Below is an example of how you would obtain it:

```
int myvalue;                    // declare an integer
```

```
myvalue = messagename.WORD(2);      // initialize the integer to 0x51FF
myvalue >>= 4;                      // shift 0x51FF to the right 4 bits to get 0x51F
myvalue &= 0xFF;                    // clears all unnecessary bits to get 0x1F
```

11.2.1 Using Signals (Only Available with a Database)

Each CAN message carries 0 to 8 data bytes, defined by the user and always a whole number of bytes. Signals partition these data bytes into sections from 1 to 32 bits each. A CAN message may carry several signals, each with its own significance. Several combinations are possible to fill all possible 64 bits (8 bytes) of data. For example, a CAN message could carry four 8-bit signals and a 32-bit signal, sixteen 1-bit signals and three sixteen-bit signals, and so on.

Because signals are already defined in the database, you do not have to worry about swapping the value from Intel to Motorola format or vice versa. Byte ordering is also predefined in the database as bit locations where the signal is located are set. In the previous section, the value 0x1F is obtained after three lines of code. If an 8-bit signal is defined in the database for the same message to start at bit location 20, you can obtain the 0x1F value with just one line of code. See the code below for an example:

```
int myvalue;                        // declare an integer
myvalue = messagename.signalname;   // 'myvalue' is assigned 0x1F
```

11.2.2 Physical Values and the “phys” Attribute

A signal is like a fixed-size variable that holds numeric data. Once a database is finalized according to specification, and a node or a system is built using that database, the size of a signal is rarely required to change. However, signal value representation may have to change to make it easier for the user to understand it. The signal data transferred across the bus consists of raw values. Raw values may not mean anything from your standpoint. What you are interested in is their physical values.

Physical (also referred to as ‘engineering’) values are derived from the raw values, generally by using a linear equation. For example, a node that outputs a message with an unsigned 16-bit signal denoting the engine speed (RPM) could actually hold a value between 0 and 65,535. To make it more meaningful, you probably have to convert it into a physical value in RPM. In the simplest case, divide the raw value by 10 to get a range between 0 and 6,553 – a meaningful range for RPM.

Let us take a look at an opposite scenario. You have another unsigned 16-bit signal (maximum value of 65,535). This time, the physical value requires a range between 0 and 100,000. One way to correct this problem is to use more than 16 bits to specify a larger range, but that would waste bits and make message size larger. Shown below is a better approach, which is more feasible if you can afford to lose precision.

Set up a conversion formula in the database by specifying an Offset and Factor for the signal – see Section 22.7. The physical value is then calculated as follows:

Physical Value = (Raw Value * Factor) + Offset

For the signal example above, entering a Factor of 2.0 and an Offset of 0 would give the desired range of numbers between 0 and 100,000.

To access the physical value (or to have CAPL automatically convert the number from the appropriate raw value), simply add **.phys** to the signal name in CAPL. Assume **EngSpeed** in the next example is an unsigned 16-bit signal using a Factor of 2.0 and an Offset of 0.

```
message EngData EDMsg;              // declares EDMsg as type EngData
EDMsg.EngSpeed = 50000;             // sets raw value, the same value on the bus
EDMsg.EngSpeed.phys = 50000;        // sets physical value
```

Note that if a conversion formula is defined for **EngSpeed**, the two assignment statements do not have the same affect. Using the **.phys** attribute above sets the raw value to 25000.

Note: Always use the **.phys** attribute whenever a formula is defined; not doing so can cause errors that are difficult to track down.

11.2.3 Round-Off Error in Symbolic Signal Access

It is important to remember that signal values are always saved as discrete numbers. A discrete number is a whole number, with no significant digits after the decimal point. If a non-discrete physical value is assigned to a signal, the closest discrete raw value is saved (after applying the conversion formula). Then when the signal's value is read, it may not match the previously set value.

To demonstrate the round-off error, consider the example in the previous section. The conversion formula had a Factor of 2.0 and an Offset of 0. If the code has the following assignment the conversion formula is used to calculate a raw value of 2000.5.:

```
EDMsg.EngSpeed.phys = 4001;
```

Because only discrete (or whole number) values are legal, and **EngSpeed** is an unsigned 16-bit signal, 2000 is saved instead of 2000.5. To get the value of **EDMsg.EngSpeed.phys** after the line executed above, the conversion formula now returns the value 4000 and not 4001!

11.3 Message Transmission

Messages are mostly declared in the **Global Variables** section of the CAPL Browser, so they are globally accessible throughout the program. The main reason for declaring a message in CAPL is to send it out onto the bus. CAPL supports sending messages triggered by an event, such as pressing a keyboard key, receiving another message, receiving an error frame, or the expiration of a timer. To send a message is very simple. Just call the **output()** function in any event procedure or user-defined function.

Before transmission, it is necessary to set all the message properties and signals to ensure a successful transmission. You can set the signals of a message in one event procedure and the send the message in another event procedure. When the message is ready to send, the signals will contain the latest values; therefore, setting one signal value will have no affect on the other signals within that message.

```
message 0x100 msg1 = {dlc = 8};        // declare message to send

msg1.byte(0) = 0x12;                   // set the data field
msg1.long(1) = 0x90785634;
msg1.CarSpeed = 100;                   // assume CarSpeed occupies the last three bytes

msg1.CAN = 2;
output(msg1);                          // send the message
```

11.4 Message Reception

In CAPL, a message reception process occurs only when a message is received from the CAN bus and an event procedure is defined for that message. If a particular message is received from the CAN bus and the CAPL program does not have an event for it, the CAPL program will ignore that message. It should be noted that this property causes CAPL programs to behave much like a CANalyzer/CANoe Stop filter.

Once a message is received, the corresponding message event procedure will execute. The **on message** event procedure must have a parameter to specify the message or messages. Simply replace the **<newMessage>** string with a numeric or symbolic identifier. If you want to use a symbolic identifier from the database, right-click on the spot and select **CANdb Message…** to select the symbolic name from the message list. The message name will automatically be inserted into the location. Be advised that double-clicking the message name from the message list more than once will insert it more than once.

One important concept to remember when defining a message event procedure is event priorities (especially when the ***** symbol is used to replace the identifier) – see Section 10.5, Event Priority.

Table 34 below shows examples of various receive message events that can be declared:

Message(s)	Message Event Procedure Name	Occurs When
100	on message 100	the message ID 100 (decimal) is received
0x100	on message 0x100	the message ID 100 (hexadecimal) is received
4 or 8	on message 4,8	either message ID 4 or 8 (decimal) is received
0x200-0x2FF	on message 0x200-0x2FF	any message ID 200 thru 2FF (hexadecimal) is received
Engine	on message Engine	message named Engine defined in database is received
any message	on message *	any message is received
any message on CAN1	on message CAN1.*	any message is received on CAN channel 1

Table 34 – Examples of Supported Receive Message Events

Message events are defined to pick up useful data from the bus. The desired data is usually stored in the data field of a message. As with populating a message for transmission, you must use the same methods to retrieve the data out of the message, either by using signals or by using a slightly more complicated approach that uses the **BYTE**, **WORD**, **LONG**, and **DWORD** keywords. Other selectors (**DLC**, **CAN**, **RTR**, and so on) may also be used to select messages by key attributes. Regardless of the information you seek, it is important to know about the **this** keyword.

The **this** keyword is used in every message event to access the received message's properties (through selectors) and signals. Think of it as a pointer to the message members of that event. The **this** keyword allows you to retrieve any valid information from the message, and it only allows read-only access. Assigning to what the **this** keyword points to is prohibited.

Below is a valid example using the **this** keyword:

```
on message ABSdata
{
    if (this.DIR == RX)
    {
            write("Message ID = %d is received from channel %d", this.ID, this.CAN);
            write("The signal value of car speed is %d", this.CarSpeed);
    }
}
```

Below is an INCORRECT way to use the **this** keyword:

```
on message CAN1.ABSdata
{
    this.CAN = 2;   // trying to set the transmit channel to 2
    output(this);
}
```

The invalid example is frequently found in CAPL programs for gateway simulation, network nodes that connects two or more buses together to share/exchange information. The intention was to send the **ABSdata** message received from CAN Channel 1 onto CAN Channel 2. The **ABSdata** message will indeed go back onto the bus, but onto CAN Channel 1 and not 2. It goes back to CAN Channel 1 because it is the default transmit channel.

Because local changes to the **this** keyword are not allowed within its event, there is only one way to send the same information onto CAN Channel 2: copy the **ABSdata** message onto another message and then transmit it. In this case, no value is assigned to whatever the **this** keyword points to.

Below is the workaround for the invalid example above:

```
on message CAN1.ABSdata
{
    message * gatewayMsg;
    gatewayMsg = this;
    gatewayMsg.CAN = 2;
    output(gatewayMsg);
}
```

or

```
on message CAN1.ABSdata
{
    message CAN2.* gatewayMsg;
    gatewayMsg = this;
    output(gatewayMsg);
}
```

Below is an example of using a message in the CAPL Browser:

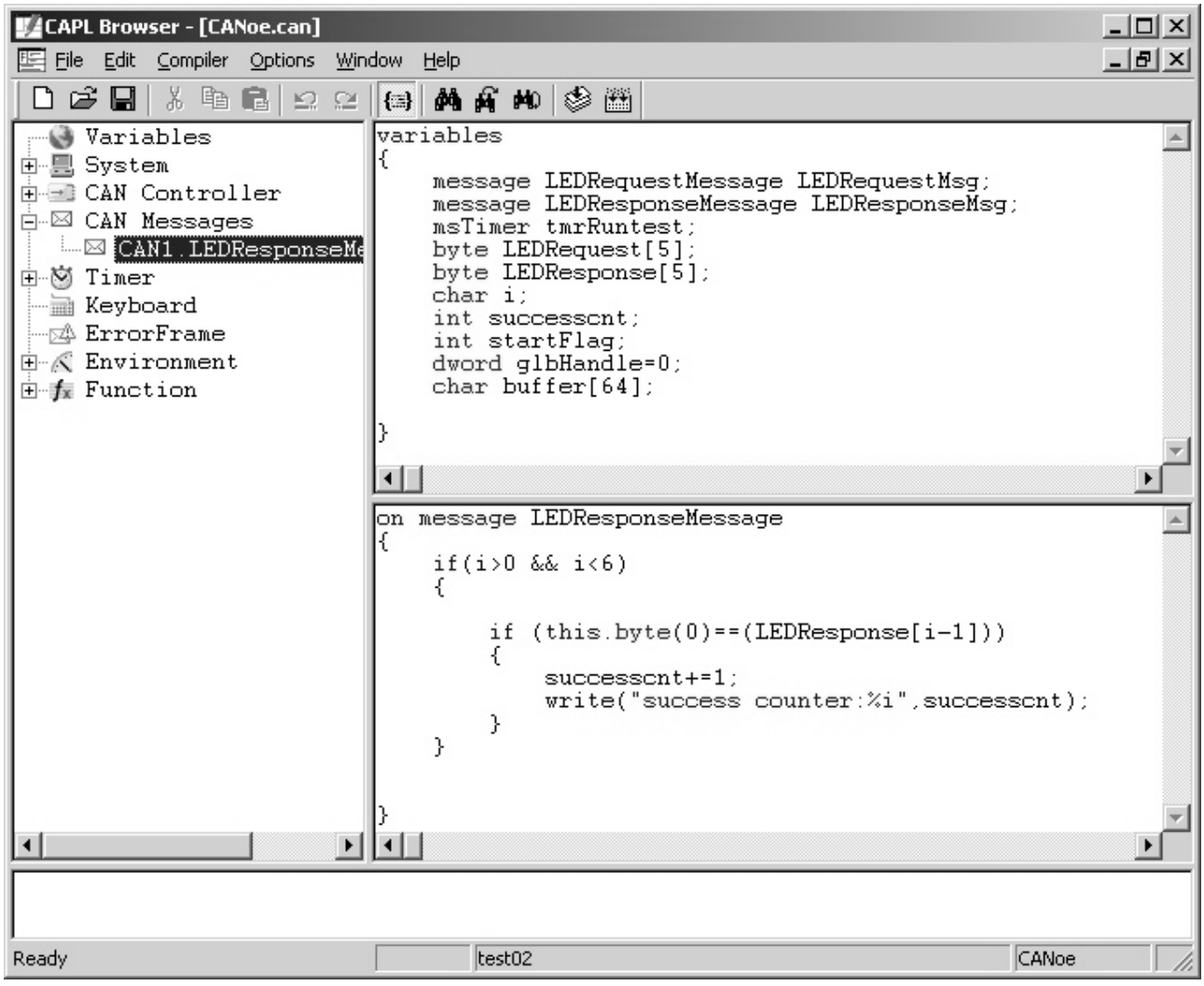

Figure 38 – Receiving a Message in the CAPL Browser

11.5 Error Frames

CAN protocol controllers are capable of detecting message errors and will alert the network to the presence of an error by transmission of an Error Flag within the message frame. CAN messages found to contain errors are referred to as Error Framesand are discarded by all bus receivers. While occurrences of Error Frames on a functioning CAN bus communication system should be at or near zero, encountering such problems is quite common during the early development stage of a CAN-based module or distributed embedded system.

Common causes of error frames include the following:

- Incorrect bus termination for high-speed CAN networks (hardware)
- Poor grounding (electrical)
- Improper interface circuitry (hardware)
- Incorrect CAN bit timing parameters (software)
- Generated on purpose by the program calling:
  output(errorFrame);

It is important to understand that there is no information about what caused the error or which node may have signaled the error contained within with the Error Frame. This is essentially a CAN protocol detail that is beyond the scope of this text.

11.5.1 Error Frame Event

Error Frames are detected by the tool's CAN hardware, and can be processed through the use of the **on errorFrame** event. The **on errorFrame** event is similar to the **on message** event, except that it occurs whenever an error frame is received on the bus. This event is normally used to keep statistics on the number and timing of error frames. For example, the following Error Frame event procedure could be used to analyze bus errors (frequency, activities during the time frame of the event, and so on.) in a CAPL node:

```
on errorFrame
{
    if (this.can == 1 && (timeNow() – pretime) < 100000)
        write(“Error frames received less than one second apart on channel 1.”);
}
```

Note: The system time returned by the **timeNow()** function and using the **TIME** selector are in units of 10 microseconds.

Figure 33 illustrates how this would look in the CAPL Browser:

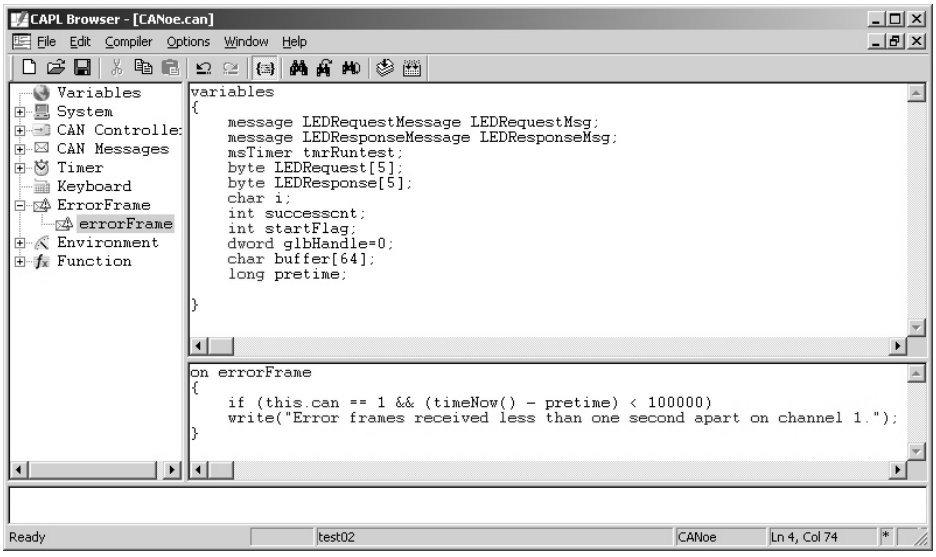

Figure 39 – On Error Frame Function in the CAPL Browser

12 Using Keyboard Events

Keyboard inputs are an excellent way to make things happen with CAPL. CAPL provides an extensive set of keyboard service through keyboard events.

A keyboard event occurs whenever a key is pressed. A corresponding keyboard event procedure can be assigned to the key pressed. All letters, both upper case and lowercase, and number keys are available. For function or extended keyboard keys (for example, ‘F1’, ‘PageDown’, ‘End’, and so on) use symbolic identifiers.

If uncertain about which key to use, the “*****” symbol can also be used to represent all keystrokes.

12.1 Keyboard Events and Procedures

There is no need to use a declaration statement for a keyboard event. Creating the corresponding keyboard event procedure is all that is necessary.

The **on key** event procedure requires a parameter to identify by name that the key has been pressed. Virtually all standard key presses common to other PC-based programs (Visual Basic, for example) are available in CAPL. Table 35 shows examples of supported key presses:

Keystroke	Event Procedure	Occurs When
a	on key 'a'	the lower case "a" key is pressed
A	on key 'A'	the uppercase (capital) "A" key is pressed
A	on key 0x41	the uppercase (capital) "A" key is pressed
2	on key '2'	the number "2" is pressed
$	on key '$'	the special character "$" is pressed
End	on key End	the End key is pressed
Shift + F1	on key shiftF1	simultaneous Shift + F1 keys are pressed
Control + PageDown	on key ctrlPageDown	simultaneous Control + Page Down keys are pressed
any key	on key *	any key is pressed

Table 35 – Examples of Supported Key Presses

Key presses are case sensitive, therefore CAPL treats the lower-case letter differently from the upper-case letter. If an upper-case key is used and Caps Lock is disabled, you have to press both the ‘Shift’ key and the upper-case key. The ‘Shift’ key does not have to be included as a parameter. However, if an extended key and ‘Shift’ is required, then it has to be included as a parameter, as shown in the table above (for example, ‘Shift’+’F1’). Multiple key events require no spaces when defining as a parameter.

ASCII hexadecimal code can also be used to replace the symbolic identifier (for example, 0x20 for the spacebar, 0x41 for ‘A’, and so on). Extended keys do not require single quotation marks, as shown in the table above.

12.2 Using the Keyword “this” with the Wildcard Symbol “*”

The wildcard symbol might be useful if you want to either place all key stroke-handling routines into a single event procedure, or to have several different keystrokes execute the same instructions. The **this** keyword is used to represent and determine the actual key that was pressed.

For example:

```
on key *                    // respond to all keys
{
    switch (this)           // “this” holds the value of the key pressed
    {
        case ‘a’:
        case ‘b’:
            write(“a or b was pressed.”);
            break;
        default:
            write(“You have pressed the %c key.”, this);
            break;
    }
}
```

Note: When more than one event statement matches the event, only the most specific event procedure is executed. For example, if an **on key *** event procedure is declared as well as an **on key ‘a’** event procedure, only the **on key ‘a’** event procedure is executed when the ‘a’ key is pressed.

13 Using System Events

A CAPL program also has events defined that may be used to control the flow of execution. Unlike the **main()** function used in the C/C++ language that outlines how the C/C++ program executes, system events in CAPL control the program “lifecycle” – the operational state.

13.1 Types of System Events

For CAPL, the detectable CANalyzer or CANoe system events used within the program environment include the following:

- Initialization
- Start
- Stop

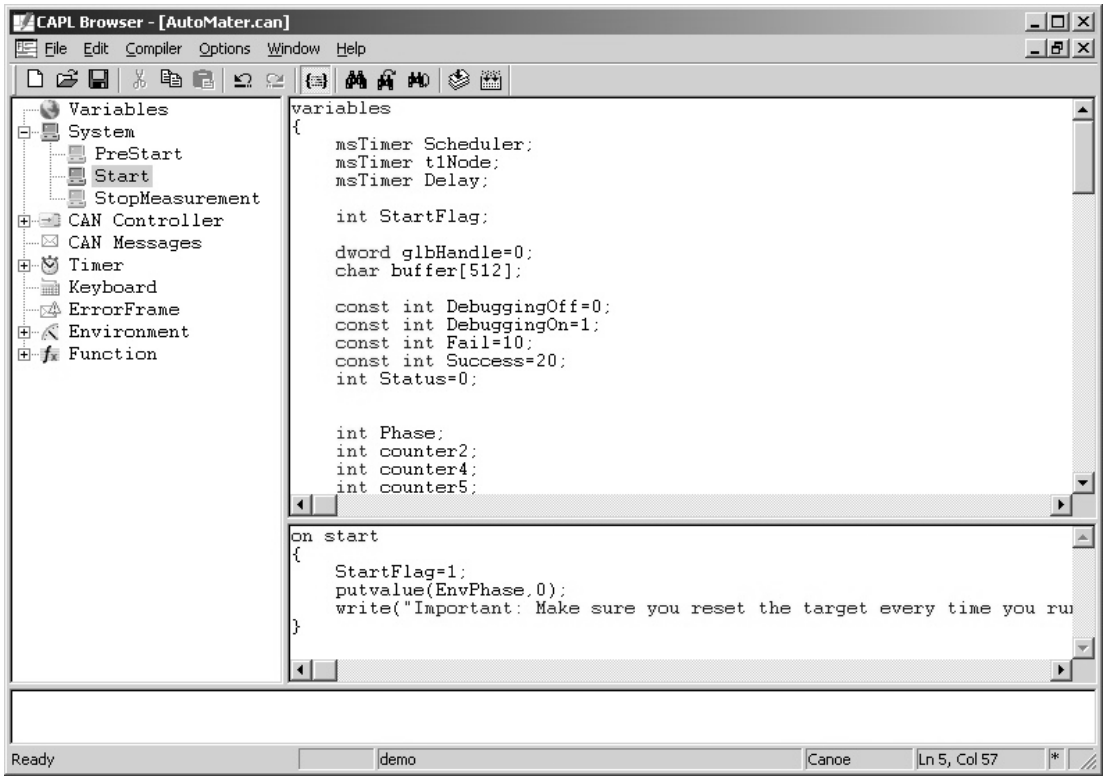

Figure 40 – System Events in CAPL

13.2 What Happens When a System Event Occurs?

CAPL provides event procedures to execute actions before, at the start of, and after CANalyzer or CANoe measurements. Their names correspond to when they are used, and they only execute once in each measurement. Table 36 shows these system event procedures.

Tool Event	Occurs When	Procedure Executed
Initialization	The tool’s Start Button is pressed	on preStart
Start	The tool’s Start Button is pressed	on start[1]
Stop	The tool’s Stop Button is pressed or if the CAPL function stop() is invoked.	on stopMeasurement

Table 36 – System Events

Note:
1. The **on start** event procedure executes after the **on preStart** event procedure.

When a measurement is started in CANalyzer or CANoe, the **on preStart** event procedure will be executed first (if one exists). This procedure is typically used to read data from files, initialize variables, or to print characters in the **Write** window. Other actions, such as outputting a message onto the bus, are not available in the **on preStart** event procedure. Most of the time, actions or processes that cannot be executed in the **on preStart** event procedure can be placed into the **on start** event procedure.

Once the **on preStart** event procedure has finished executing, the **on start** event procedure immediately follows (if one exists). The **on start** event procedure can be used to set timers, output messages onto the CAN network, or initialize environment variables in CANoe. When the **on start** event procedure finishes executing the measurement is then started, and other defined event procedures are enabled.

When the measurement stops in CANalyzer or CANoe, the **on stopMeasurement** event procedure(if one exists) is the last to execute. Applications of this procedure might include:

- printing statistics in the **Write** window
- sending final messages onto the CAN communication network
- writing to a log file

After the **on stopMeasurement** event procedure has finished executing, the measurement is terminated.

Note: It may be necessary to execute or call all environment variable event procedures at the beginning of the measurement. This action is sometimes necessary to initialize variables, to set timers related to environment variables, and to send out messages onto the bus containing values of environment variables. CAPL provides the **callAllOnEnvVar()** function to execute all your event procedures involving environment variables. This function calls every environment variable event procedure that you have defined. **CallAllOnEnvVar()** function is normally called within the **on start** event procedure.

14 Using Timers

Because CAPL is designed to provide an event-driven environment, timers provide an easy way to trigger periodic events. CAPL allows for an infinite number of user-defined timers to be set.

A timer is a programmable relative clock. After starting the clock with your pre-set amount of time, the timer runs until it expires. the corresponding timer event procedure will be executed when the alarm expires. As a result, using timers is a three-step process:

1. Declare a timer variable
2. Pre-set the timer in an event procedure (except the **preStart** event) or a user-defined function
3. Define an **on timer** event procedure for that timer

Based on typical timing requirements, two types of timers are available:

- Millisecond timer - pre-set to a number in milliseconds
- Seconds timer - pre-set to a number in seconds

14.1 Declaring a Timer

To use a timer, declare the timer by name in the **Global Variables** window of the CAPL Browser. A timer cannot be declared within an event procedure. If the unit of interest is in seconds, the timer is defined as **Timer**. If the unit of interest is in milliseconds, the timer is defined as **msTimer**.

As an example, let's say we need to use 2 timers - one with a duration of a tenth of a second and the other for one minute. We could define these timers by using the following CAPL code:

```
msTimer tenth_second_clock;
Timer one_minute_clock;
```

14.2 Starting a Timer

To start a timer, you must invoke the CAPL function **setTimer()**.

To continue using our example with our tenth of a second clock and our one-minute clock, we can start both of these timers by using the **setTimer()** function.

```
setTimer(tenth_second_clock, 100);      // set timer to 100 milliseconds
setTimer(one_minute_clock, 60);         // set timer to 1 minute
```

14.3 Expiration of a Timer

When a timer or clock expires, the corresponding event procedure will be executed.

To illustrate a delayed message transmission process, message 100 will be sent to the bus 20 milliseconds after the ‘a’ key is pressed as illustrated below:

```
variables
{
    msTimer myTimer;            // creates a millisecond timer
    message 100 msg;            // creates message 100
}

on key ‘a’                  // when the ‘a’ key is pressed
{
    setTimer(myTimer,20);       // set myTimer to 20 ms
}

on timer myTimer            // when myTimer expires (after 20 ms)
{
    output(msg);            // output the message
}
```

14.4 Resetting a Timer

All defined timers are resettable. Resetting can be done in any event procedure. Simply call the **setTimer()** function again, passing the timer variable and using the same or different time duration.

If you are trying to reset a timer, and that timer has not yet expired, you will get a run-time warning in the **Write** window of CANalyzer/CANoe and your attempt to reset that timer will be ignored. To reset a timer that is still running, call the **cancelTimer()** function first and then the **setTimer()** function.

14.5 Periodic Clock

When implementing a CAPL program to emulate node behaviors, you may have to implement a periodic timer. The most common way to create and use a periodic timer is shown in the following steps:

1. Declare the timer in the **Variables** window of the CAPL Browser
2. Set the timer in the **on start** event procedure
3. Define an event procedure for the timer to perform a given task
4. Reset the timer at the beginning or end of the task.

In the following example, a periodic message is sent using a timer.

```
variables
{
    message 0x555 msg1 = {dlc = 1};
    mstimer timer1;                     // define timer1
}

on start
{
    setTimer(timer1, 100);              // initialize timer to run for 100 msec
}

on timer timer1
{
    msg1.byte(0) = msg1.byte(0) + 1;    // increment the data
    output(msg1);                       // output message
    setTimer(timer1, 100);              // reset the timer
}
```

14.6 Stopping a Timer Before It Expires

As mentioned before, a timer can be stopped by using the **cancelTimer()** function.

For example, if we wanted to stop the one-minute clock at some precise instant, we would write:

```
cancelTimer(one_minute_clock);          // stops the 1 minute clock
```

This function call is ignored if the one-minute clock is either not set or has already been expired.

14.7 Common Timer Mistake

The **cancelTimer()** function can indeed cancel a timer and avoid the errors that occur when trying to reset a timer that has not expired; however this function may also introduce a common mistake. In the example below, in addition to setting up a timer to send the periodic message every 100 milliseconds, an **on key** event is implemented to also send the message whenever the ‘a’ key is pressed.

```
variables
{
    message 0x555 msg1 = {dlc = 1};
    mstimer timer1;                     // define timer1
}

on start
```

```
{
    setTimer(timer1,100);          // initialize timer to run for 100 msec
}

on timer timer1
{
    msg1.byte(0) = msg1.byte(0) + 1;    // increment the data
    output(msg1);                       // output message
    setTimer(timer1,100);               // reset timer
}
on key ‘a’
{
    cancelTimer(timer1);                // cancel timer
    setTimer(timer1,100);               // reset timer
}
```

In this example, the user added an **on key** event to set a timer so as to output the message. The user knows that when the **on key** event is called, a timer error will occur because the timer cannot be set again while it is running. The user called the **cancelTimer()** function first, and then set the timer. This action, however, introduced another mistake. If the user presses the ‘a’ key faster than 100 milliseconds, the message will never be sent because the timer is cancelled every time before it expires. To send a message every 100 milliseconds and on the ‘a’ key, outputting the message within the **on key** event will correct the problem, as seen in the example below:

```
on key ‘a’
{
    output(msg1);
}
```

15 Using CAN Protocol Controller Events

A CAN protocol controller is the hardware device used to transmit and receive CAN messages. The typical CAN bus interface used with CANalyzer or CANoe contains one or more CAN controllers, so as to permit interconnection to CAN networks. While it is beyond the scope of this text to describe how a CAN controller operates, the reader must be aware that CAPL supports controller-level events. These events generally cause error frames, and error frames affect the state of a CAN controller. Error frames and the state of the CAN controller have tremendous impact on CAN communication; therefore, CAPL has system events defined, each representing the state of the CAN controller.

The detectable CAN controller events used within the CAPL program environment include the three primary CAN protocol error states and the optional CAN controller interrupt called "warning limit ". Each CAN channel accessible in CAPL corresponds to a CAN controller in one of the possible four states.

15.1 CAN Controller States

The CAN protocol controllers error states detected (from all active CAN protocol controllers) in CAPL as events are

- Error Active
- Warning Limit
- Error Passive
- Bus Off

Each state is determined by the accumulated count values within the CAN controller's Transmit Error Counter and Receive Error Counter. These error states occur under the following conditions:

Error Active	When the CAN controller is initialized, and whenever the transmit and receive error counts are less than or equal to 95
Warning Limit	Whenever the transmit or receive error count equals or exceeds 96
Error Passive	Whenever the transmit or receive error count equals or exceeds 128
Bus Off	Whenever the transmit error count is greater than or equal to 256

Figure 41 shows these different error states and the counter values that cause a transition from one state to another.

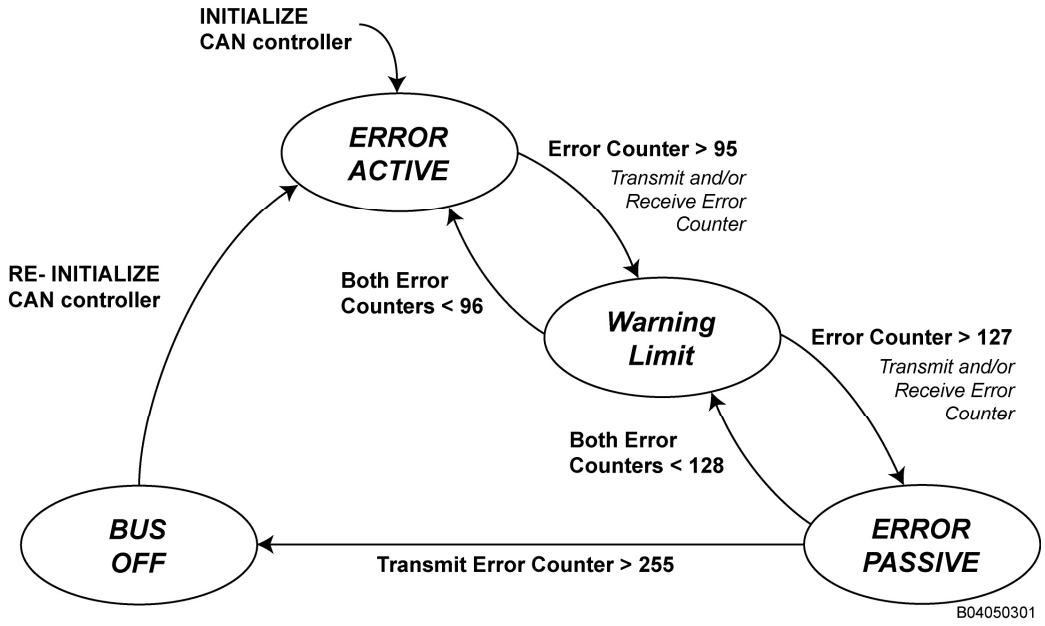

Figure 41 – CAN Controller Error States

To get a summary of how the counter increases or decreases, see Section 26.8.4. – CAN Error Counting

15.2 CAN Controller Events

As shown in Table 37 below, whenever the CAN Controller's error state changes from one state to a new state, this transition will cause the corresponding event procedure to be executed.

CAN Controller Event	Occurs When	Procedure Executed
Bus Off detected	CAN Controller error state changes to Bus Off state	on busOff
Error Active detected	CAN Controller error state changes to Error Active state	on errorActive
Error Passive detected	CAN Controller error state changes to Error Passive state	on errorPassive
Warning Limit detected	CAN Controller error state changes to Warning Limit	on warningLimit

Table 37 – CAN Controller Events and the Corresponding Event Procedures

When a CAN Controller is powered and initialized, the internal CAN Controller error state always begins as Error Active, with both the Transmit Error Counter and the Receive Error Counter set to zero. Only errors detected on the CAN network will cause the error counters to increase.

If enough errors are detected, the CAN Controller error state will transition to the Error Passive state. If an Error Passive event procedure has been established to do something when this condition occurs, that procedure will now be executed. If CAN communication transfer conditions improve, either by successful transmissions or receptions in the Error Passive state, causing the error counters to decrease, then the error state will transition back to the Error Active state.

The Bus Off condition occurs when the CAN Controller's internal Transmit Error Counter is greater than or equal to a count of 256. Because the error count that triggers Bus Off (256) only applies to a transmitter, a receiver cannot go Bus Off, because the receive error counter can only reach 128, or the Passive state. During a CAN-based module's normal operation, Bus Off is considered to be a serious condition that needs to be reported.

The event that corresponds to the Warning Limit condition is not specifically associated with a CAN Controller error state change, but is a detectable condition that is optionally included in some CAN Controllers. The Warning Limit condition occurs when either the receive error count or transmit error count exceeds 96.

Note: The CAN hardware used with CANalyzer and CANoe supports this optional and detectable warning limit condition.

The **this** keyword is used in CAN Controller event procedures to access the CAN Controller's error counters. Both the Transmit Error Counter and the Receive Error Counter are accessible. To access the value of the Transmit Error Counter, use the variable name **this.errorCountTX**. Use **this.errorCountRX** to access the value of the Receive Error Counter.

For example, a warning could be placed in the **Write** window when the warning limit condition is reached. The following CAPL program could accomplish this:

```
on warningLimit
{
    write ("CAN Controller at Warning state");
    write (" Tx errors = %d", this.errorCountTX);
    write (" Rx errors = %d", this.errorCountRX);
}
```

16 Using Environment Variables

CANoe's graphic panels are very useful, because they help the user visualize events by recreating the automotive environment, such as the dashboard console, switches, lights, and so on. The behavior of network nodes with regard to external input and output signals is described by special variables called environment variables, that can represent such things as switch positions or engine speed. CAPL programs are used to regulate these variables. Environment variables make connection possible between elements on the panel and the associated CAPL program.

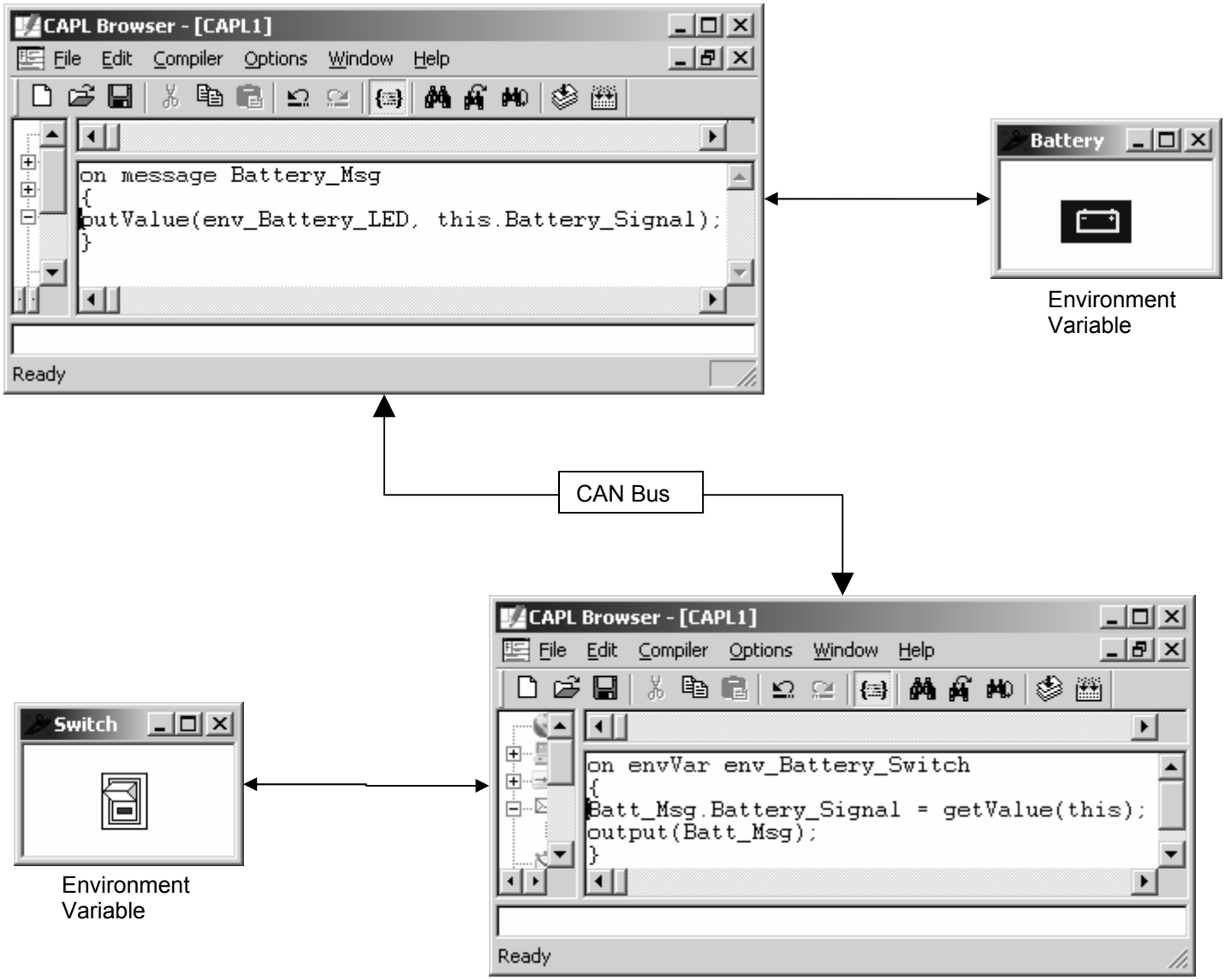

Figure 42 –Environment Variables In CANoe

The concept of environment variables is an important one to understand when using CAPL with CANoe. Environment variables are used in CANoe to represent external values on the network, such as the position of a switch. The environment variables are defined in an associated database with the CANdb++ or CANdb Database Editor, and can be read and changed by a CAPL program. Because they are accessible to all events and functions in a CAPL program, they are considered global variables. These global variables are also available to all CAPL programs in the same CANoe configuration and not just one single CAPL program. In other words, they are global to the CANoe environment.

Note: It is considered a bad programming technique to use environment variables to exchange information between CAPL programs, because CAPL programs normally represent nodes on a CAN network. In reality, environment variables do not exist on a physical CAN system. Therefore, use only CAN messages to exchange data between CAPL programs.

16.1 Environment Variable Types

Just as with variables you define in CAPL programs, environment variables have to be assigned a data type in the database. The choice of data type includes INTEGER, FLOAT, STRING, and DATA. The INTEGER and FLOAT data type are commonly used, but STRING and DATA are not, because the data field within a message contains numeric data. The STRING data type is used to store ASCII characters. DATA on the other hand, is used to store values in term of bytes. In addition, the DATA data type has a length attribute specifying the size of the data field.

16.2 Environment Variable Initialization

By default, CANoe initializes all environment variables to their default value assigned in the database whenever the measurement starts. This feature is not so useful if a measurement has to stop and then restart again to continue with a test exactly where it left off before. For example, in the first measurement you send the necessary messages to start a car and put it in a driving position by utilizing the states (values) of an environment variable. Then you have to restart your CANoe measurement for another test with the car already running; however, because the state of the environment variable has to reset to its default value, your second test will always fail. Fortunately, in CANoe there is an option to disable reinitializing all the environment variables every time the measurement starts by going to `Configure → Options`.

Sometimes it is necessary to initialize all environment variables at the beginning of the measurement before their values change during the measurement, even if these initial values are different than the default values in the database. CAPL provides the **callAllOnEnvVar()** function to execute all environment variable event procedures. This function calls every environment variable event procedure that you have defined in a CAPL program. You would normally call **callAllOnEnvVar()** in the program's **on start** event procedure to initialize your environment variables. Related actions may include the following:

- Initializing variables
- Setting timers to activate in response to changes in environment variables
- Sending out messages onto the CAN bus containing the starting values of your environment variables.

16.3 Declaring an Environment Variable Event

Environment variables can only be defined in a database, so they cannot be declared in the CAPL language. They are, therefore, declared using the CANdb++ Database Editor as illustrated in the steps below:

1. Select `Environment` from the **Events** window in the CAPL Browser (tree view at the left), right-click and select `New`. This creates a new environment variable event procedure name in the left window and an outlined procedure in the right window where the event procedure code is entered.
2. Replace the string **<newEnvVar>** in the right window with the name of the environment variable by right-clicking on the highlighted string, choose `Environment Variable from CANdb…` and select the environment variable from the list. If the selection is deactivated, it means the database is not associated to CANoe when opening the CAPL Browser.

Below is an environment variable event procedure to lock all the doors after the ignition is on by sending a message:

```
on envvar envIgnitionSwitch
{
    if (getValue(this) == 1)
            // set signals here to lock all doors
    else
            // set signals here to unlock all doors
    output(doorStateMessage);
}
```

16.4 Event Execution

When the value of an environment variable changes, the corresponding environment variable event procedure is executed. The value can only be changed by two methods. The most common method is through a panel. The input/output user-defined elements on a graphical panel are associated with environment variables (or with signals if for display only). Whenever the user changes the value or state of these elements, it will change the value of the associated environment variable automatically, thereby triggering the corresponding environment variable event. The second way to change the value of an environment variable is by calling the **putvalue()** function.

16.5 The putValue() and getValue() Functions

Unlike other variable assignments, assigning an environment variable requires calling a function. The **putValue()** function is used to set or initialize an environment variable. This function takes two or three parameters, depending on the data type of the environment variable.

Data Type	Syntax Format
Integer	void putValue(envVarName, int value);
Float	void putValue(envVarName, float value);
String	void putValue(envVarName, char value[ ]);
Data	void putValue(envVarName, byte value[ ]);
Data[1]	void putValue(envVarName, byte value[ ], long size);

Table 38 – putValue() Function Syntax

Notes:
1. Instead of assigning the entire array of bytes into the environment variable, this format specifies the number of bytes to assign.

Unless the environment variable's value is different, calling the **putValue()** function will not reset the environment variable. The corresponding environment variable event procedure will not execute.

Calling the **getValue()** function will return the environment variable value. It has five formats like the **putvalue()** function.

Data Type	Syntax Format
Integer	int getValue(envVarName);
Float	float getValue(envVarName);
String	long getValue(envVarName, char value[ ]);
Data	long getValue(envVarName, byte value[ ]);
Data[1]	long getValue(envVarName, byte value[ ], long size);

Table 39 – getValue() Function Syntax

Notes:
1. Instead of accessing the entire array of bytes from the environment variable, this format specifies the starting byte location to access.

The formats return the active value of the environment variable for integers and floats. If an array is required as in the STRING and DATA data type, the function returns the number of bytes copied.

16.6 The "this" Keyword

The **this** keyword is used in conjunction with the **getValue()** function to access the value of the environment variable.

Note: Be careful! Do **not** use the **this** keyword with the **putValue()** function in an environment variable event procedure, because that may cause the event to execute in an infinite loop.

Below is an example of using the **this** keyword with the **getValue()** function:

```
on envVar FlowRate
{
    int val;
    val = getValue(this);                    // read value of FlowRate into val
    write("The flow rate is %d", val);       // display it
}
```

Below is an illustration of an on envVar function created in the CAPL Browser:

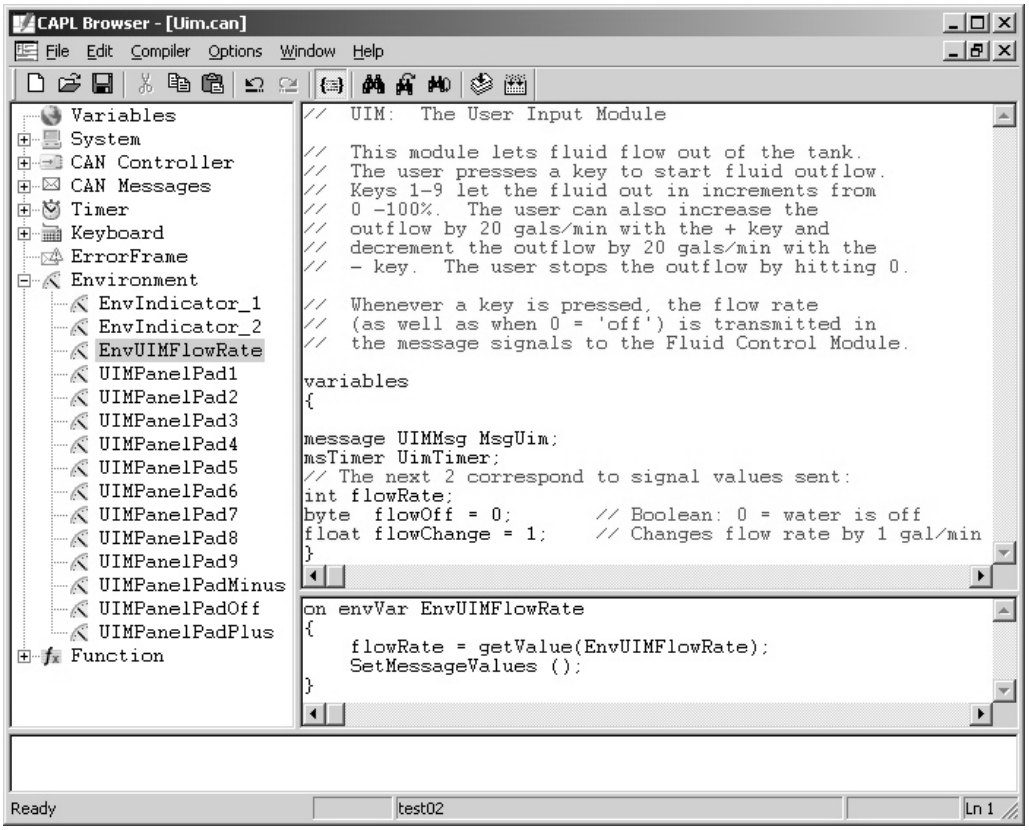

Figure 43 – An Environment Variable Function in the CAPL Browser

17 Using File Input/Output Functions

The file input and output functions allow read and write access to data files, making it easy to record data results. For performance reasons, these functions do not support complex searching processes. Alternative methods that make complex searching possible include implementing a COM interface or a CAPL DLL.

In one application area, it might be valuable to record a set of events in your distributed system, time stamp the events, and then use these time values to determine the performance of your system. Having such performance parameters available over time may provide some engineering benefits.

In another application area, it might be beneficial to use an input file that contains a script of system input events to play back into your distributed system. Predefined CAPL File I/O functions may help to accomplish this.

17.1 File Input/Output Functions

CAPL supports file I/O with several functions split into categories. The new releases of CANalyzer and CANoe have made many file I/O functions obsolete, and a new set of functions has been added to CAPL to replace those functions. The new functions substantially improve searching mechanisms.

Functions starting with **fileWrite…** and **fileRead…** have changed to **filePut…** and **fileGet…** respectively. Functions starting with **seqFile…** have changed to **writeProFile…** and **getProFile…**. Only two file I/O functions for logged files have not changed: **writeToLog()** and **writeToLogEX()**.

Categories	Example Functions	Read/Write	Limitations	Status
File I/O functions	fileWriteInt(); fileReadInt()	Read and write	Special file set-up is required.	Obsolete
	fileGetString(); filePutString()	Read and write	None	Supported
ProFile functions	getProFileArray(); writeProFileFloat()	Read and write	Special file set-up is required.	Supported
Sequential file functions	seqFileRewind(); seqFileGetLine()	Read only	CAN.INI set-up is required. Program node must be placed in the Simulation branch.	Obsolete
Write to log file	writeToLog()	**Write** only	Logging has to be enabled to take effect.	Supported

Table 40 – Categories of File Input/Output Functions

Note: Obsolete functions can still be used in CAPL programming; however, they are not recommended for long-term use in the future, because no support will be given as new software releases are issued.

17.1.1 Set-Up for File Input/Output Functions

File I/O functions such as **fileWriteInt()** or **fileReadString()** are obsolete because of their search mechanism. A file path is used as one of the parameters every time a function is called. This results in overheads. A new group of functions starting with **fileGet…** and **filePut…** (including **fileWriteBinaryBlock()**) has been implemented to replace this group of I/O functions. This new group of functions has two advantages. First, there is no need to set up the CAN.INI file, so CANalyzer or CANoe does not have to reload. Second, this group has two new functions, one called **openFileRead()** and the other **openFileWrite()**, to return a file handle to eliminate the overheads mentioned earlier. Once the handle is opened for data access, it can be closed any time during measurement using the **fileClose()** function.

File I/O Functions	
Function	**Purpose**
fileReadArray	Read an array of data from a file (obsolete)
fileReadFloat	Read a floating point number from a file (obsolete)
fileReadInt	Read an integer from a file (obsolete)
fileReadString	Read a string from a file (obsolete)
fileWriteFloat	**Write** a floating point number to a file (obsolete)
fileWriteInt	**Write** an integer to a file (obsolete)
fileWriteString	**Write** a string to a file (obsolete)
fileClose	Close a specific file
fileGetBinaryBlock	Read a character from a file in binary format
fileGetString	Read a string from a file
fileGetStringSZ	Read a string from a file w/o new line character
filePutString	**Write** a string to a file
fileRewind	Reset pointer to beginning of file
fileWriteBinaryBlock	**Write** a character to a file in binary format
openFileRead	Open a file for read access
openFileWrite	Open a file for write access
setFilePath	Set the directory path for file to read and/or write access
setWritePath	Set write path for function openFileWrite and ProFile functions

Table 41 – File I/O Functions

17.1.2 Set-Up for Sequential File Access Functions

CAPL offers functions for sequential read-only file access such as **seqFileGetLine()** and **seqFileGetBlock()** as listed in Table 40.These functions require data in the data file to be structured like the one shown below in Figure 44 (the same format used by initialization files or *.INI files). These functions are only available for use in nodes located in CANalyzer's Transmit Branch and CANoe's **Simulation Setup** window.

Sequential File Access Functions	
Function	**Purpose**
seqFileClose	Closes a file opened by seqFileLoad()(obsolete)
seqFileGetBlock	Reads a block of characters from a file (obsolete)
seqFileGetLine	Reads a line from a file (obsolete)
seqFileGetLineSZ	Reads a line from a file (null-terminated) (obsolete)
seqFileLoad	Opens a file for reading (obsolete)
seqFileRewind	Resets a file to read from the beginning (obsolete)

Table 42 – Sequential File Access Functions

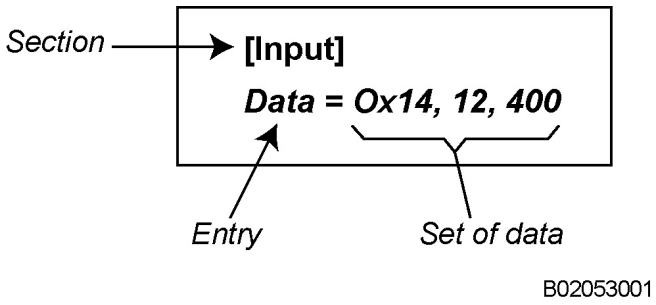

Figure 44 – File Format Example

Before calling these functions in the CAPL code, the CAN.INI file must be edited to specify the path where the data file is located. The CAN.INI file is located in the CANalyzer or CANoe system directory (e. g. c:\program files\CANoe\EXEC32). Open the file with a text editor, and find a section that looks like the following:

```
[CAPL]

SeqFilePath = "C:\usr\max\capldata"
; working folder for files that are to be read from
; CAPL with seqFileLoad
```

When the **seqFileLoad()** function is called to open a file, the path specified in the quotation marks above is searched for the file. If the file cannot be found in that directory, the system directory (for example,, EXEC or EXEC32) is searched for the file. If CANalyzer or CANoe is running while the changes are made to the CAN.INI file, the program has to be reloaded for the changes to take effect.

ProFile I/O Functions	
Function	**Purpose**
getProFileArray	Reads an array section from a file
getProFileInt	Reads an integer section from a file
getProFileFloat	Reads a float section from a file
getProFileString	Reads a string section from a file
writeProFileInt	Writes an integer value to a section of a file
writeProFileFloat	Writes a float value to a section of a file
writeProFileString	Writes a string to a section of a file

Table 43 – ProFile I/O Functions

With the new group of ProFile functions,functions, as listed in Table 41, the file path is set with either the **setWritePath()** or the **setFilePath()** function in CAPL. This group of functions still requires data in a structural data formatted file. A file name is required as one of the parameters to use the **getProFile…** functions to read data. This file has to be located in the same directory as the associated database(s) or saved configuration, or as pointed to by the **setFilePath()** function. A file name is also used with **writeProFile…** functions to write data into a file; however, the destination path can be set by the **setWritePath()** function. If the **setWritePath()** function is never called, the saved configuration directory is used. More importantly, this group can do both read and write access to a file, whereas the **seqFile…** functions can only read.

18 Using the Serial and Parellel Port

Two functions are defined in CAPL for the parallel port: one function to transmit a byte (**outportLPT()**) and the other to receive a byte (**inportLPT()**). Although the parallel port expansion is useful, it is only supported on certain PCs.Because of difficulties with using the parallel port across NT-based operating systems (for example Windows 2000), it is recommended that the parallel port not be used. To use these functions requires a generic parallel port driver found at the path **…EXEC32/GploDrv**. Follow the instructions in the **readme.txt** file, and remember to change the BIOS setting in the PC for the parallel port to use ECP.

18.1 Installation of RS232VC DLL

There are no pre-defined functions for the serial port in CAPL. You must associate a DLL to CANalyzer or CANoe to give CAPL serial port functionalities. Before CAPL can gain access to the serial port, the DLL file rs232vc.dll must be saved in the CANoe/CANalyzer EXEC32 directory (the file is downloadable at www.vector-CANtech.com). EXEC32 is the default directory for all CAPL DLLs.

There are two ways to associate the DLL. To inform the CAPL compiler of the existence of new CAPL functions for the serial port and to allow it to compile the CAPL programs correctly, the following must be entered in the CAN.ini file located in the EXEC32 directory. Scan the file for the SIMULATION entry, and then under it, make the DLL association as follows:

```
[SIMULATION]
CAPLDLL=rs232vc.dll
```

This method is really tricky. Be careful not make a mistake in the initialization file. The file is used to initialize and load the CANalyzer or CANoe tool with predefined settings. To avoid any mistakes, below is another method that is safer:

1. Go to the **Configure** menu of CANoe/CANalyzer (Version 5.0 or newer)
2. Select **Options**.
3. Add the rs232vc.dll to the list under CAPL-DLL. Once it has been added, you can activate or deactivate the DLL whenever necessary as long as the tool's measurement is not running. If the CAPL Browser is opened, close it and reopen it again to utilize the functions in the associated DLL.

Note: The mentioned above DLL is not recommended for NT and Windows 95 users.

18.2 RS232 CAPL Functions

18.2.1 RS232SetCommState

This function initializes the serial port. Currently, only one COM port can be opened at a time. The registers of the interface chip are accessed directly: therefore, the function must not be executed while data is still being transmitted.

All RS232 functions access the port specified by this function. The current implementation only provides for the operation of one serial port (Table 42).

RS232SetCommState	
Rs232SetCommState (long COMport, long baudrate, long length, long stopbits, long parity);	
COMport	1 or 2 (COM1 or COM2 where device is located)
baudrate	Baud rate of the device (for example, 4800, 9600, 115200)
length	7 or 8 (size of device control block in bytes)
stopbits	0 = 1; 1 = 1.5; 2 = 2
parity	0 = no; 1 = even; 2 = odd

Table 44 – Parameters of the RS232SetComm State Function

The function referenced above calls the Windows API function **SetCommState()** to set parameters. The specified parameters are passed to this API function.

18.2.2 RS232ByteCallback

If an event occurs at the serial port, CANoe/CANalyzer calls the CAPL function **RS232ByteCallback()**. Although this function does not return a time stamp of when the event actually occurred, the time the event occurred can be queried with the CAPL function **timeNow()**. The function does return the COM port in which the data is received, but it is always the COM port set by the **RS232SetCommState()** function, because only one port can be opened at a time. The **RS232ByteCallback** function parameters are listed below in Table 43.

Rs232ByteCallback	
Rs232ByteCallback (long COMport, long data, long dir);	
COMport	Port where data is received or transmitted
data	Data received
dir	0 = event due to transmit operation 1 = event due to receive operation 2 = event due to error at the serial port

Table 45 – RS232ByteCallback Function Parameters

18.2.3 RS232WriteByte

The function **RS232WriteByte()**, as listed in Table 44, transmits data from CAPL programs via the serial port.

Up to and including the current version, the COMport entry is ignored and all transmit requests are rerouted to the COM port defined in **RS232SetCommState()**.

Rs232WriteByte	
Rs232WriteByte (long COMport, long data);	
COMport	Port in which data is to be transmitted
data	A byte of data to transmit

Table 46 – RS232WriteByte Function Parameters

18.2.4 RS232WriteBlock

The function **RS232WriteBlock()**, as listed in Table 45, transmits an entire data block from CAPL programs via the serial port. Up to and including the current version the COMport entry is ignored and all transmit requests are rerouted to the COM port defined in **RS232SetCommState()**.

Rs232WriteBlock	
Rs232WriteBlock (long COMport, unsigned char dataBlock[], long size);	
COMport	Port in which data is to be transmitted
dataBlock	An array of data to transmit
size	Number of data bytes to transmit from data array

Table 47 – RS232WriteBlock Function Parameters

18.2.5 RS232EscapeCommFunc

The function **RS232EscapeCommFunc()**, as listed in Table 46, is used to set the DTR (Data Terminal Ready) and RTS (Request To Send) status lines from CAPL. This function calls the Windows API function *EscapeCommFunction()* to set the status lines.

Rs232EscapeCommFunc	
Rs232EscapeCommFunc (long modemControl);	
modemControl	0 = clears both DTR and RTS 1 = sets DTR 2 = sets RTS 3 = sets both DTR and RTS

Table 48 – The RS232EscapeCommFunc Function

18.2.6 RS232CloseHandle

The **RS232CloseHandle()**, as listed in Table 47, function explicitly closes the serial port from within CAPL programs.

Rs232CloseHandle	
Rs232CloseHandle (long COMport);	
COMport	Serial port to disconnect from communication

Table 49 – The RS232CloseHandle Function

19 Constructing CAPL Programs

CAPL has been designed to make functions and events efficient and easy to use. CAPL programs generally consist of many small events and reusable functions to emulate node behaviors and to analyze bus traffic. A CAPL program is only found in one source file that requires no previously compiled functions from library files for compilation. After the ASCII CAPL program is first compiled, a binary file with extension CBF (CAPL Binary File) is created. Both CANalyzer and CANoe only need this binary file to execute the program's behavior. Because this approach varies from system to system, we will not go into where the program should be assigned in CANoe and CANalyzer.

19.1 CAPL Program Organization

In contrast to a traditional C program, CAPL programs have three distinct parts:

Global Variable Declarations
In this section of the CAPL Browser (upper right), global variables are declared and initialized so they can be used throughout the CAPL program. It is also essential to declare messages and timers in this section.

Event Procedures
Event procedures are blocks of code executed when an event occurs. CAPL has many types of event procedures. Most of the program code will be in event procedures, because most actions are performed after an event occurs, such as when a message is received on the CAN bus. Event procedures cannot return a value; therefore, you must declare a global variable instead.

User-Defined Functions
In addition to CAPL's built-in functions, users can implement their own functions/procedures without function prototypes. These functions are globally accessible and can contain any legal permissible CAPL code. Putting frequently used code in a procedure makes programs more efficient. User-defined functions can return a value of any simple data type.

19.1.1 Creating Network Nodes

As mentioned before, a CAPL program can be implemented using any text editor as long as you know the syntax and layout. By using the CAPL Browser, you are provided a user-friendly interface to program. The CAPL Browser is also a compiler that allows you to trace the source of errors. Other text editors, such as Notepad, lack this ability; therefore, after the program is implemented, you must use CANoe or CANalyzer to compile it. Compiling it with CANoe or CANalyzer is harder to trace where the error is located, and it is time-consuming because you have to go back and forth between the tool and the text editor.

As a result, the best approach to implement a CAPL program is to use the CAPL Browser. The CAPL Browser is a standalone program that came with CANoe and CANalyzer. You need to use the easiest method to create a new network node because this network node has to be referenced by CANoe and the database (assuming it is already associated to CANoe). Use the following procedure to create a new network node:

1. Go to the **Simulation Setup** window of CANoe and right-click on the red and black parallel lines to select `insert network node`.
2. Associate this node a CAPL program in order to simulate node behavior. In CANalyzer, you must go to the **Measurement Setup** window to insert a network node called a P block.
3. After the network node is inserted, either double-click on it or click on the pencil icon to associate or create a new CAPL program. The Open Dialog will open, allowing you to either select an existing CAPL program or you can type in a name and create a new CAPL program. If you select an existing CAPL program, that program will get associated to the network node you defined and the CAPL Browser will open that CAPL program allowing you to modify it. If you create a new CAPL program, then the CAPL Browser will open the empty program, allowing you to implement and automatically associate it to the network node once you save it.

Opening the CAPL Browser through a network node has two advantages: First, the program you implement using the CAPL Browser is automatically associated to the network node. If you open the CAPL Browser as a standalone program, you have to associate what you have implemented into the corresponding network node by right-clicking on the network node and selecting `Configuration`. Second, the CAPL Browser is already referenced to the database you assign to that network (the network where you insert the network node). The database already has the messages and signals defined so you can reference and use them.

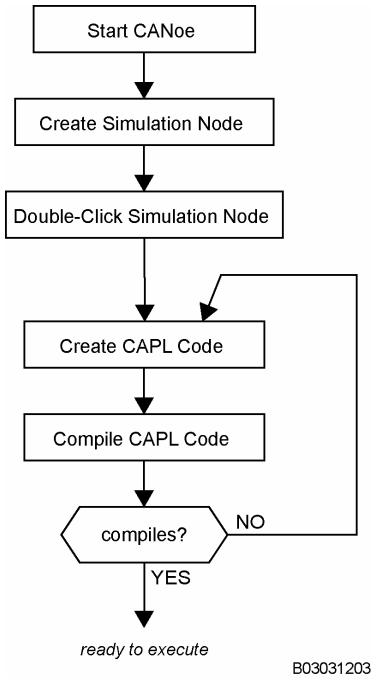

Figure 45 – Creating New Network Nodes

19.1.2 P Block Placement and Message Pass-Through

CAPL program blocks (**P** blocks or Network Nodes) are not inherently transparent to CAN network events. This means that a message or error frame in the Analysis Branch of CANalyzer or CANoe will not pass through a P Block unless the **P** Block is specifically programmed to pass the message or error frame on to the next activity.

To make a P Block that will allow messages and error frames to pass though, implement the following CAPL code:

```
on message *
{
    . . .              // insert additional code here if desired
    output(this);
}


on errorFrame
{
    . . .              // insert additional code here if desired
    output(errorFrame);
}
```

This is the only method used to allow messages to bypass the CAPL program, but there are two programming segment to keep in mind. If there is another message event procedure in addition to the **on message *** event procedure that is more specific, then the **on message *** event procedure will not execute (See Chapter 10 on CAPL event procedures for the * symbol and event priorities).

The second warning has to do with where this CAPL program is being associated in CANalyzer and CANoe. The event procedures defined above may cause an infinite loop for incoming messages or error frames if the CAPL program is located on the Simulation or Transmission Branch. An example: if you defined a network node on the **Simulation Setup** window of CANoe, and you have the above message event procedure defined, every message this network node receives will be transmitted onto the bus again. In this situation, other nodes on the same network will get the same message twice, first sent by the original message owner and then by the network node with this message event defined.

20 CAPL Program Examples

This chapter gives several examples of functionality commonly implemented by CAPL users. The examples are very straightforward if you keep in mind that everything in CAPL is triggered by an event. Each defined event has an event procedure and only one event procedure can be executed at a time.

Each example gives you good practice in CAPL programming. The tasks may not be node related, but they will certainly help you later to implement CAPL programs.

Note: If you are not familiar with how to use the CAPL Browser, please read Section 7 – The CAPL Browser.

If you need to look up a function, use either the CAPL Browser's online help system or the CAPL Function Reference Manual.

20.1 The Write() Function

When it comes to CAPL programming, mistakes tend to be hard to locate. You may not have a compilation error, but you see the wrong result returned by a function or see the wrong signal value sent to the CAN bus. These mistakes are almost unavoidable because no program is perfect the first time it is written. Because CAPL does not support debugging features like Visual C++ (due to performance reasons), the best way to debug in CANoe is to simulate the CAPL program and invoke the **write()** function whenever necessary.

Note: Do not test the CAPL program with a real node unless you are very confident of the results. Unimaginable things may occur if the wrong information is sent to a real node. You can simulate a CAPL program using the simulated mode or have your CAN cables connected in a loopback, as shown in Figure 8.

To better understand how this function is used, see the following example that illustrates that the **write()** function supports many value types. It displays the key that was pressed in hexadecimal, decimal, and character, and can be used in any event procedure.

```
on key *
{
    write("Key pressed: hex = %x, dec = %d, char = %c", this, this, this);
}
```

20.2 Sending a Periodic Message

The most important task a network node should be capable of is to send messages. In addition, it needs to know when to send, how often to send, and what data to send.

When information needs to be transferred on a repetitive basis, a periodic message is used. The message is resent every time a timer expires, as shown in Figure 46.

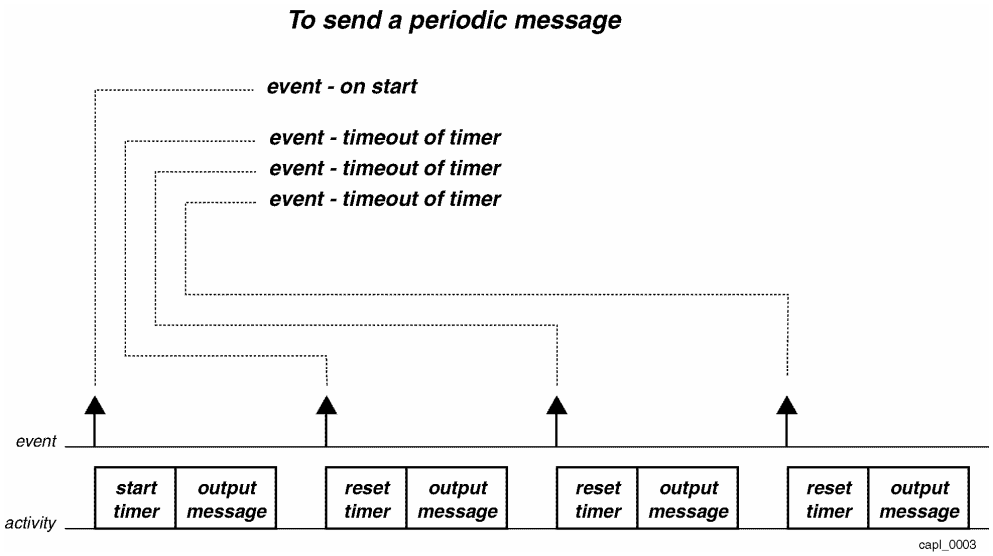

Figure 46 – Periodic Message Transmission

We can see that sending a periodic message requires the use of a timer. The timer is started when the measurement starts, and the associated message is output to the bus. The timer eventually expires, which triggers the event for which we have been waiting. The timer event procedure immediately resets the timer and outputs the periodic message. The timer will run cyclically, and this repetitive process will continue until the measurement is stopped or the **cancelTimer()** function is called for that timer.

The timer must be defined by name in the **Global Variables** area (the upper right window of the CAPL Browser), as shown below, to get this timer running, the timer will be set in the **on start** event procedure, which executes whenever a measurement starts.

```
variables
{
    message 0x555 msg1 = {dlc = 1};
    msTimer timer1;                    // define timer1 in milliseconds
}


on start
{
    setTimer(timer1, 100);          // initialize timer to 100 ms
}
```

In the Variables section, the message **msg1** is defined with a hex ID of 555 and has a data payload size of one byte. The millisecond timer is then defined as **timer1**. The **on start** event procedure is used to set the timer; therefore, the **setTimer()** function is called passing the timer variable and its duration, in this case 100 milliseconds.

When this timer expires (or times out), the corresponding **on timer** event procedure for **timer1** will be executed. The timer will first be reset to 100 milliseconds to make it recursive. Then the first data byte of the CAN message will then be incremented before it is sent onto the bus with the **output()** function (see the example below).

```
on timer timer1
{
    setTimer(timer1,100);              // reset the timer

    msg1.byte(0) = msg1.byte(0) + 1;   // increment the data
    output(msg1);                      // output the message
}
```

20.3 Sending a Conditionally Periodic Message

It is very common for a simulated node to send messages periodically. But periodic messages do not usually get sent when the measurement starts in CANalyzer or CANoe. In reality, a node usually listens for a common network key to begin sending periodic messages. In a vehicle, for example, the CAN bus is not fully awake until the key is in the ignition.

A conditionally periodic message is only sent in response to the occurrence or end of an event, as it is not certain when the event will take place. For example, the conditionally periodic message may be sent when a car door opens and stop when it is closed. Another case might be to send a conditionally periodic message when the cruise control is turned on and stop when it is deactivated.

Sending a conditionally periodic message also requires the use of a timer, but this timer does not need to run cyclically.

In our next example, the CAPL program will generate a CAN message that is transmitted periodically when active at a frequency of 10 Hz. This activates and deactivates the timer when the ‘a’ key on the PC keyboard is pressed (Figure 47). The first press of the ‘a’ key will start the periodic message and the second press of the ‘a’ key will stop the periodic message. Each subsequent press of the ‘a’ key will toggle back and forth between these two states.

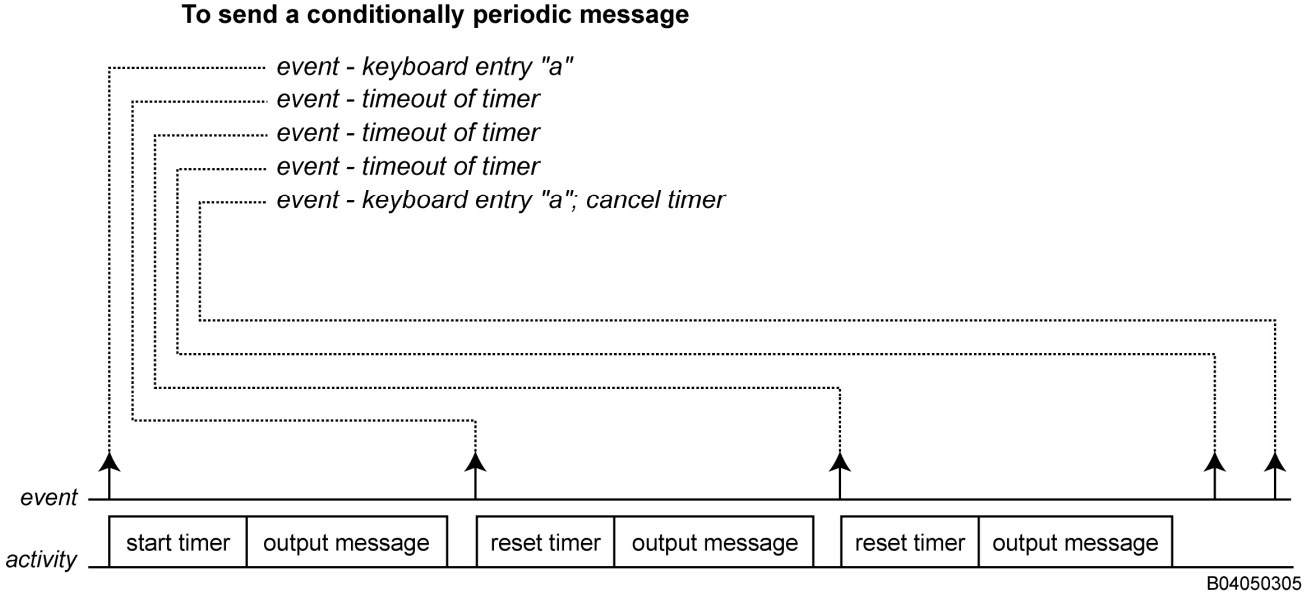

Figure 47 – Conditionally Periodic Message Transmission

Now, we will go a step further and use a database. In the database, a message is defined (see the example below), associated with a car speed signal, and sent by the ABS (anti-lock braking system). When the condition is right, the ABS module will continuously send out the message for the ICM (instrument cluster module) to display on the speedometer.

```
variables
{
    message ABSmessage msgA;
    mstimer timerA;
    int condition = 0;                    // initialize condition = false
}


on key ‘a’
{
    condition =! condition;          // toggle condition
    if (condition == 1)                    // if condition is true
```

```
    {
        cancelTimer(timerA);        // cancel the timer
        setTimer(timerA,100);       // then start timer
    }
}


on timer timerA
{
    if (condition == 1)             // if condition is still true
    {
        setTimer(timerA,100);       // then continue timer
    }
    msgA.carspeed = msgA.carspeed + 1;      // increment the car speed
    output(msgA);                   // output a message
}
```

To properly begin the process, we have defined an integer variable called **condition** in the **Global Variables** area and initialized its value to 0, which represents the false condition. Once the user presses the ‘a’ key, the **on key ‘a’** event procedure will execute. This procedure toggles **condition** to 1 and checks to make sure the condition is active or true. If the condition is true, then set the timer to start.

When **timerA** expires, the corresponding event procedure, **on timer timerA**, will execute. If the condition remains true, the timer will reset before the data is modified and the CAN message is sent.

Notice that when the condition has been toggled back to false by pressing the ‘a’ key, the car speed signal is incremented one last time and sent.

20.4 Reading Data in a Received Message

The message chapter in this text explains two ways to access data. Data can either be accessed by using symbolic signal names defined in an associated database, or by referencing in terms of **BYTE**, **WORD** or **LONG**. Shown below is an example of how to read data from a message using symbolic names:

```
on message EngineData
{
    write(“engine speed = %d”, this.EngSpeed.phys);
    write(“engine temperature = %d.”, this.EngTemp.phys);
}
```

Note the **phys** modifier is used in the above code segment to ensure that the conversion formulas defined in the database are used to get the physical (engineering) values of speed and temperature.

If no signals are defined for a message in a database, the second way is to access the raw data bytes in the received message. This is the quickest way to access message data without a database defined or to copy blocks of data from one message to another. If the **EngineTemp** message was defined without any signals assigned in the database, then the only way to access the message data is illustrated by the following:

```
on message EngineTemp
{
    float temp;

    // assume temp is within the first four bytes
    temp = this.long(0);

    write(“The temperature is %g degrees.”, temp);
}
```

Because there is no signal defined for the first four bytes of data, there is no conversion formula, so the same value is returned with or without the **phys** modifier.

Note: If the number needs to be scaled or otherwise adjusted, the only solution is to implement the codes for the conversion formula.

To prove that a database greatly reduces time and effort, let's look at another example where a signal layout in the message is not as perfect. If the signal value is desired from the data bytes of a message that only occupies 12 bits, neither 8 nor 16 bits, it would be difficult to get the value. If 6 bits come from the first byte of the message and the remaining 6 bits come from the second byte, the following is what the code would look like to retrieve the 12-bit value:

```
on message ExampleMessage
{
    int value;

    value = this.word(0);          // loads in the first two bytes
    value >>= 2;                   // right shifts out two bits
    value & 0x0FFF;                // clears all unnecessary bits

    write("Value = %d", value);
}
```

Figure 48 illustrates what the actual bits would look like in this case:

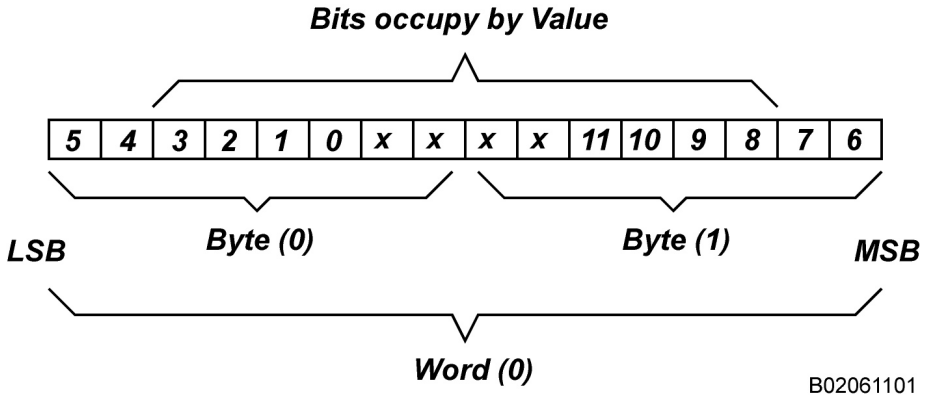

Figure 48 – Actual Bus Representation of Message Read in by CAPL

Note: CAPL reads data from received messages in Intel format unless the signal in the database is configured to use Motorola.

20.5 Responding to a Message After a Delay

Messages are not only sent just to provide data, but they can request data. If a message is sent to a receiver requesting data, the receiver may respond immediately or with a delay. If it is an immediate response, you just invoke the **output()** function in the **on message** event. To set up a delay, you need a timer. For example, a message might be sent after a driver door closes saying that the door is now closed. A few seconds later, the dome light would react to this message by turning off. To respond after a delay, use both a timer event procedure and a message event procedure. Below are the steps to program a timer event procedure and a message event procedure:

1. Declare a timer and delay variable in the **Global Variables** section:

```
variables
{
    timer delayTimer;                  // seconds-based timer
    long delayTimerPeriod = 5;         // initialize delay of 5 seconds
}
```

2. Create the **on message** event procedure that sets the timer if the doors are closed:

```
on message doorState
{
    if (this.Closed == 1)                                   // door is closed
        setTimer(delayTimer, delayTimerPeriod);
    else                                                    // door is opened
    {
        . . .           // do this if door is opened
    }
}
```

3. Create an **on timer** event procedure that will actually turn off the light five seconds later:

```
on timer delayTimer
{
    message DomeLight dlMsg;
    dlMsg.Status = 0;                   // turns off dome light
    output(dlMsg);
}
```

20.6 Sending an Error Frame

While occurrences of error frames on a functioning CAN bus communication system should be at or near zero, encountering such problems is quite common during the early development of a CAN-based module or distributed embedded system.

A CAPL program can also generate error frames of its own. Typically, the user needs to send an error frame to the CAN bus to test its integrity and to test the nodes' responsiveness. The **output()** function is used to send an error frame in the same way as sending a message. Simply place the following line of code in the event procedure wherever it is needed.

```
output(errorFrame);
```

The error frame event procedure is called when an error frame occurs on the bus. Use the following procedure to analyze bus errors (frequency, activities during the time frame of the event, and so on) in a CAPL node in the Evaluation Branch.

```
on errorFrame
{
    write("Error frame detected on channel %d.", this.CAN);
}
```

20.7 Using Panels

In the panel example illustrated in Figure 49, assume there are two simulated nodes defined and two panels already associated to CANoe. The transmitter node has a switch on the panel assigned to an environment variable, **evLEDSwitch**. This switch is the triggering device used to turn on and off a LED assigned to an environment variable, **evLED**, on the receiver node. Typically, users create one panel per node and one set of environment variables per node to eliminate confusion.

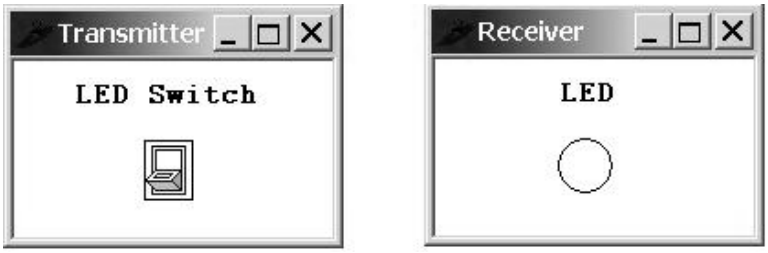

Figure 49 – Example Panels

For the transmitter node, you would have one CAPL program to handle the user input by clicking on the switch on the Transmitter panel. The switch is assigned to **evLEDSwitch**, therefore, the program should have an event procedure defined to handle the value change. When the value of **evLEDSwitch** changes, it will set a signal in a message and then transmit that message to the CAN bus.

Below is the code for the Transmitter node:

```
variables
{
    message LEDmsg msgA;        // LEDmsg is defined in the database
}


on envVar evLEDSwitch
{
    msgA.LED = getvalue(this);    // LED is a signal in the LEDmsg message
    output(msgA);
}
```

When the Receiver node on the CAN bus receives the **LEDmsg** message, it will turn the LED on its panel either on or off based on the signal in that message.

```
on message LEDmsg
{
    putValue(evLED, this.LED);
}
```

20.8 Bus Off Condition

There are many reasons why an error frame could be present on the bus. CAN controllers always detect an error frame, but often cannot specify what caused it. The best approach to determine the cause of error frames is to use an oscilloscope (or CANscope from Vector). At the application level, what causes error frames is really unimportant. What is important is to know how many error frames have been received and what state the CAN controller is in. The CAN controller can go from the Error Active state to Error Warning to Error Passive, and finally Bus Off. In the Bus Off state, the CAN controller does not communicate. This condition is true to all Vector interface peripherals because they all use the Phillips SJA1000 CAN controller. To make the CAN controller active again, it must be reset.

Typically, the CAN controller must be reset by the local microcontroller. However, some CAN controllers handle the bus off condition a little bit differently. A CAN controller may be reset externally or permitted to reset after certain conditions are met.

For CANoe, the user has full control. In most situations, the user restarts the CANoe measurement to reset the CAN Controllers. If you do not want the measurement to be restarted, implement the bus off event procedure to have the CAN controller(s) reset itself, as seen in the code below:

```
on busOff
{
    setBtr (1, 0x01, 0x14);          // set the baud rate for channel 1
    resetCanEx (1);                  // reset channel 1
}
```

The **setBtr()** function allows you to set or reset the baud rate for a channel. This function is optional before you reset the CAN controller. If the baud rate has to be changed, this function must be called before the **resetCanEx()** function. If you want to reset all the CAN controllers at once instead of one at a time, use the **resetCan()** function.

Note: Resetting a CAN channel requires the CAN controller to be disconnected. Because of this disconnection, data in both the transmit and receive queues are lost.

20.9 Using a CAPL Program to Control Logging

CAPL provides built-in functions to fully control logging in CANalyzer and CANoe. When **CAPL** is selected as the Trigger Condition in the **Trigger Configuration** window of a Logging block in the **Measurement Setup** window, CAPL programs can be used to trigger and stop logging data to a log file (Figure 50).

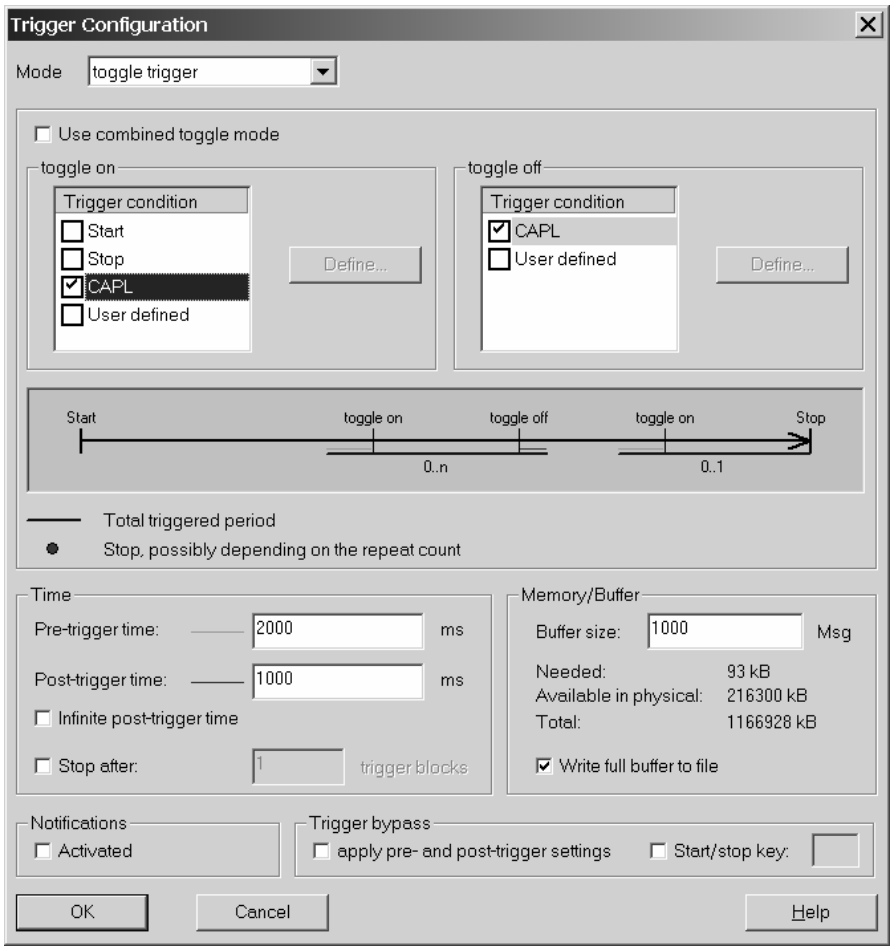

Figure 50 – The CANalyzer/CANoe Trigger Configuration Dialog

It is beyond the scope of this text to cover how to use logging in CANalyzer and CANoe, but there are several concepts that need to be understood to use CAPL to control logging.

Because logging an entire simulation can produce large log files full of mostly irrelevant data, CAPL allows you log portions of a simulation. When the conditions to begin logging are met, call the **startLogging()** function in the CAPL code. If the function call shows no result, make sure the Logging block is not deactivated in the **Measurement Setup** window (the line leading to that Logging block is not broken).

By using CAPL to control logging, all the time settings in the configuration of the Logging block can be offset by the settings that were set with a CAPL program. The pretrigger time is the period of time during which bus activity is recorded before logging is triggered. The posttrigger time is the period of time during which bus activity is recorded after logging is stopped.

In the following example, pressing key ‘1’ starts logging with a 1000 ms pre-trigger time, and key ‘2’ is used to stop logging with a 2000 ms post-trigger time.

```
on key ‘1’
{
    //To enable all Logging blocks and bypass all
    //triggering conditions:
    startLogging();

    //To enable a specific Logging block and bypass all
    //triggering conditions:
    startLogging(“loggingblockname”);

    //To enable a specific Logging block with a specific
    //pre-trigger time and bypass all triggering conditions:
    startLogging(“loggingblockname”, 1000);
}

on key ‘2’
{
    //To disable all Logging blocks and bypass all
    //triggering conditions:
    stopLogging();

    //To disable a specific Logging block and bypass all
    //triggering conditions:
    stopLogging(“loggingblockname”);

    //To disable a specific Logging block with a specific
    //post-trigger time and bypass all triggering
    //conditions:
    stopLogging(“loggingblockname”, 2000);
}
```

20.10 Data Files and CAPL

Occasionally there is a need to save measurement values over a period of time to a file or to retrieve data from a file and transmit it to the CAN bus. File I/O functions are provided in CAPL for this purpose; however, file I/O processes are not very fast because file accesses are conceived for ease of use and reliability.

Note: The data files explained in this section are not log files generated by the Logging block.

Basically, there are two sets of file I/O functions currently supported. The first set can read and write using an array of binary characters or strings. Because CAPL does not support pointers, it is a bit difficult to parse through the data file. The other set can read and write as well, and can use arrays, integers, floats, or strings. This set of functions navigates through the data file very well and to the right position quickly, but the data file must follow a structural layout. The file format used is the same as the INI files format used by Microsoft Windows applications.

Note: Sometimes users prefer to write a CAPL Dynamic Linked Library (DLL) for better navigation in the data file. See Chapter 24 – Introduction To CAPL DLLs for more information.

20.10.1 Set 1: File I/O Functions

This set of functions requires no format limitation on the data file. The data file can be accessed in either binary or ASCII mode. The mode and functions used are determined by the **openFileRead()** and **openFileWrite()** functions. The directory these two functions use to scan for the data file in either in the same directory as the database or in the configuration file. If the data file is in another location, the **setFilePath()** function is called first to set the directory path.

Most commonly used is the ASCII mode, rather than binary files because ASCII (or text) files are bigger in size, and the mode of operation to retrieve data into CAPL is easier. Users typically prefer alphanumeric values to binary because they are easier to read and debug. By using **fileGetString()** or **fileGetStringSZ()**, the function returns the maximum number of characters possible until the end-of-line character is reached in the data file. The data returned by the functions is stored in an unsigned array buffer, while the maximum number of characters to be read is determined by the size parameter. To write to a data file, use the **filePutString()** function. This function has the same syntax as the other two functions, but the buffer is written into a file associated by a file handle. Because CAPL does not support pointers, each call to the function will move to the next line in the data file. To jump back to the beginning of the data file, call the **fileRewind()** function.

Opening a file in binary mode to read or write is rarely done. To use this mode, the user must know the structure of the binary file to retrieve the desired data. The two functions associated with the binary mode are **fileGetBinaryBlock()** and **fileWriteBinaryBlock()**. Both functions use an array buffer to read or write a binary block limited to the size parameter.

The following example opens a data file in ASCII mode to read the car speed signal and uses the time stamp in the file to transmit a message onto the CAN bus (Figure 51). When EOF (end-of-file) is reached, the program creates a new file to inform the user that all the signal values have been sent (Figure 52).

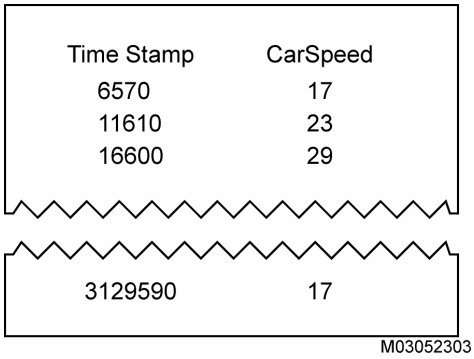

Figure 51 - Data file (random.asc)

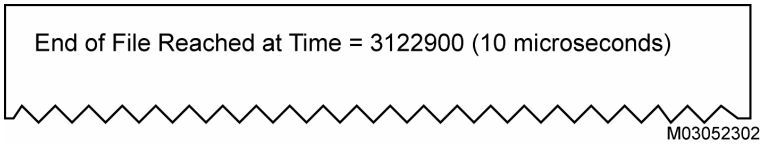

Figure 52 - Confirmation file (EOFFile.txt)

```
variables
{
    long cyclicPeriod=0;
    long prePeriod=0;
    dword readHandle = 0;
    byte endOfFileFlag=0;
    char signalBuffer[24];
```

```
    msTimer cyclicTimer;
    message ABSdata msg1;
}

on start
{
    // initialize handle to data file opened in ASCII mode
    readHandle = openFileRead ("random.asc", 0);

    // the next line of code is also used to skip the first row of data in the data file and to make sure the handle is
valid
    if (readHandle != 0 && fileGetString(timeBuffer, elcount(timeBuffer), readHandle) != 0)
        // set transmission to trigger after 100 ms
        settimer(cyclicTimer, 100);
    else
        write("Data file cannot be opened for read access.");
}

on timer cyclicTimer
{
    getData();
    if (endOfFileFlag != 1)
    {
        output(msg1);
        settimer(cyclicTimer, cyclicPeriod);
    }
    else
        endoffileconfirm();
}

on stopMeasurement
{
    // close handle to data file opened in ASCII mode
    fileClose(readHandle);
}

getData ()
{
    int i;
    i = 0;
    //store data into timeBuffer
    if (readHandle != 0 && fileGetString(timeBuffer, elcount(timeBuffer), readHandle) != 0 )
    {
    // convert period from 10 microseconds unit to milliseconds unit
        cyclicPeriod = (atol(timeBuffer) - prePeriod) / 100;
        prePeriod = atol(timeBuffer);
        while (timeBuffer[i] != 0x9) {i = i + 1;};          // skip the time stamp
        i = i + 1;
        signalBuffer[0] = timeBuffer[i];
        signalBuffer[1] = timeBuffer[i+1];
        signalBuffer[2] = timeBuffer[i+2];
        msg1.CarSpeed = atol(signalBuffer);
    }
    else
    {
        // set end of file flag if end of file is reached
        write("End of data file reached, timer cancelled.");
        endOfFileFlag = 1;
    }
```

```
}

endoffileconfirm ()
{
    dword handle;
    char buffer[64];
    setFilePath("C:\\Projects\\Data Files\\Result ", 1);

    // file will be created automatically if one does not exist. If the file exists, everything will be overwritten
    handle = openFileWrite("EOFFile.txt", 0);
    snprintf(buffer, elcount(buffer), "End of File Reached at Time = %d (10 microseconds)\n", timenow());

    if (handle!=0)
        filePutString(buffer, elcount(buffer), handle);
    fileClose(handle);
}
```

20.10.2 Set 2: ProFile Functions

The ProFile functions are another set of functions a user can use to access a data file; however, the data file must be formatted like the INI files used by Microsoft Windows applications. Below is a data file example:

```
[Message1]
ID = 0x123
DLC = 6
Signal1 = 0x23
Signal2 = 35
Signal3 = 0.125

[Control]
Type = System
Nodes = 5
NodeIDs = 0x32,56,0x12,53,0x11,33
```

The section is denoted within the square brackets. Variables under each section are independent entries that can be assigned with integers (in hex or decimal), floats, strings, and arrays. Below is an example of how CAPL retrieves the above information:

```
on key 'i'
{
    message dummyMsg msg1;
    char charArray[8];
    int rxValue;
    int flag;

    setFilePath("C:\\data file directory", 1);

    msg1.ID = getProfileInt("Message1","ID", -1, "datafile2.txt");
    msg1.DLC = getProfileInt("Message1", "DLC", -1, "datafile2.txt");

    // assume database exists for the following signals
    msg1.intSignal1 = getProfileInt("Message1", "Signal1", -1, "datafile2.txt");
    msg1.intSignal2 = getProfileInt("Message1", "Signal2", -1, "datafile2.txt");
    msg1.floatSignal = getProfilefloat("Message1", "Signal3", -1, "datefile2.txt");

    rxValue = getProfileString("Control", "Type", "Error", charArray, elcount(charArray), "datafile2.txt");
    write("charArray = %s", charArray);
    // output should be charArray = System
```

```
    rxValue = getProfileInt("Control", "Nodes", -1, "datafile2.txt");
    write("rxValue = %d", rxValue);
    // result should be rxValue = 5

    flag = getProFileArray("Control", "NodeIDs", charArray, 6, "datafile2.txt");
    write("Array: %d, %c, %x, %s", charArray[0], charArray[0], charArray[0], charArray);
    // result should be Array: 50, 2, 32, 28R5A

    //The output file, sections, and entries will be created if they
    //don't exist. Be careful - entries will be overwritten.
    writeProfileInt("Output", "Message ID", msg1.ID, "data2.txt");
    writeProfileString("Output", "charArray", charArray, "data2.txt");

    output(msg1);
}
```

21 Basic Steps in Creating Your First CANoe Application

In this step-by-step tutorial, we will develop a simple application to teach the basics of the CANoe development process.

21.1 CANoe Development - Five Step Process

There is no correct approach in constructing a CANoe application. The process depends on what needs to be done and how to get it done with the least amount of time and effort. For example, auto suppliers usually do not have to create a database to start a CANoe application because their customers usually have a database that has been adapted for CANoe applications. In this case, the foundation already exists. CANoe can start its measurement with that database and see data on the CAN bus in symbolic terms instead of numeric values.

Note: CANoe must be associated with at least one database to start its measurement (it can be an empty database).

Some projects can cause more headaches then others. Assume that the supplier is not given a database, but does have the ECU (electronic control unit) module and its specification documents. The ECU is already in its testing phase, and the supplier does not have a database to help build the tests in subsequent phases. To simplify the effort put into implementing the test processes, the supplier must first consult the specification document to construct a database.

CANoe application development projects can typically be divided into two categories: projects were the entire control system was modeled with CANoe at the onset and those that were not. CANoe users involved in projects in the first category realize that CANoe can be used to model the entire system development process. They can design and distribute the overall functionality of a system among different network nodes, and refine the design to the level of specification in the CANoe tool. These CANoe users have the ability to simulate network behavior, observe the bus load, and determine the necessary performance of the hardware being developed. Once the physical system is built and ready to test, the existing CANoe application may be used or readjusted to perform the necessary tests, leading to significant time and cost savings.

CANoe users involved in projects in the second category do not use CANoe to create their system model first. They do not have a CANoe model to use to perform the necessary tests; therefore, they must model the system before they can test it with CANoe. This process is time-consuming and generally leads to defects.

Regardless of which category your project falls into, you must build a CANoe application from scratch, so you will have to follow the five main steps shown below. These steps focus on the software end of the CANoe setup, assuming all hardware requirements are met:

1. Create a new CANoe configuration
2. Create a database
3. Create panels
4. Set up analysis functions
5. Create network nodes (CAPL)

Steps 2 and 3 will be discussed in detail in the following two chapters. Step 5 is a programming step that is discussed in Chapter 19 – Constructing CAPL Programs.

The process remains the same regardless of which set of CAN hardware is used. In this case, the hardware we refer to are CAN Controllers and CAN transceivers provided by Vector.

21.1.1 Create a New Directory

Before you begin to create a new CANoe configuration, create a new directory for the application files (Figure 53). This is not a main step in the process because it is a matter of an individual's organizational habit. A normal-sized CANoe application may contain 20-30 files; however, large applications may exceed 100 files by the time everything is finished. Having such organization also makes file sharing easy.

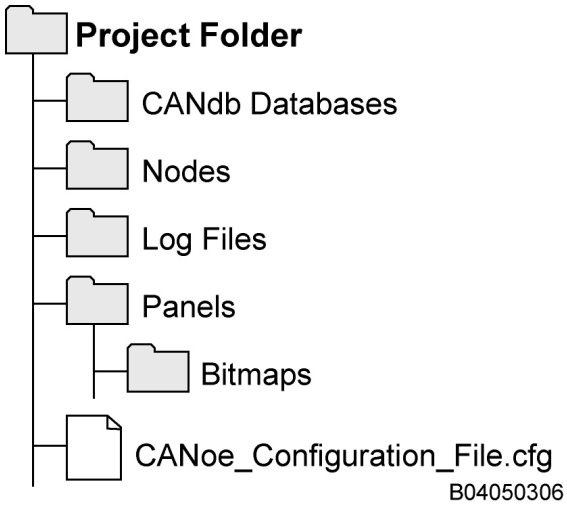

Figure 53 – Directory for CANoe Configuration

21.1.2 Creating a New CANoe Configuration

Creating your CANoe application is like opening up a business. First, you need a foundation to build your empty building complex. Before it is operational, employees have to be hired, you need an infrastructure to communicate, a business handbook to operate the business, equipment to get the job done, and some quality controls to keep it from going bankrupt. A new CANoe configuration is just like this empty building complex. It cannot be operational unless all of the following components are defined:

- Network nodes
- CAN bus parameters
- A database
- Panels
- Analysis functions for the CAN bus traffic.

To start CANoe, run the Windows-executable file CANoe32.exe from the 'Start' menu's CANoe program group. Because some background applications may limit the performance level of CANoe, consistency checks, such as testing for an enabled screen saver, are made. These checks can be ignored by checking the "hide this message in the future" option.

The most recently saved configuration is usually opened by CANoe every time it loads. Once loaded, you can make a new configuration in one of two ways. Both methods use about the same amount of time, but one method is organized using a wizard. The result of both is to have all the network nodes defined on the simulated network (CAN bus), the database associated to that network, and the CAN channel associated to that network. The wizard allows you to create the simulated networks with the **Simulation Setup** Assistant.

To start the wizard, go to the menu **File** and select **New Configuration**. The **Template** window that opens next will have a checkbox at the bottom to enable the wizard. If you prefer not to use the wizard, you can manually define the network(s), its nodes, its database(s), and CAN channel in the **Simulation Setup** window of CANoe (see Figure 54). Regardless of which method you choose, you still must define the bus parameters in this step.

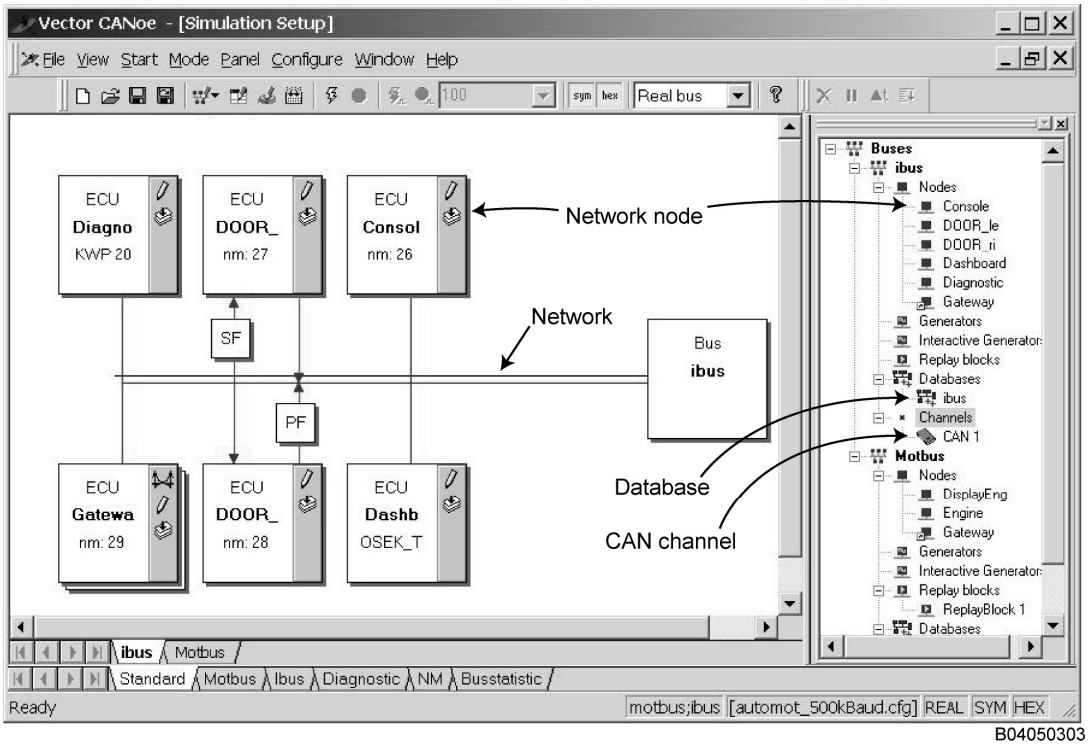

Figure 54 – Simulation Setup Window

21.1.3 Bus Parameters

Vector CANtech provides a variety of PC-based CAN interfaces: USB, PCI, and the most common PCMCIA. All CAN interface hardware has two CAN Controllers, and so two CAN channels are capable of connecting to two different networks. Every time a new CANoe configuration is made, each CAN channel has to be configured for CAN communication.

Use the following steps to configure the CAN channel parameters in CANoe:

1. Go to **Configure** from the main menu and select **Hardware Configuration**. Expand the **+** sign for either or both **CAN 1** and **CAN 2**; they correspond to CAN channels 1 and 2 respectively. Under each channel, there are three sections; **Setup**, **Filter**, and **Options**. The **Setup** section allows you to set the baud rate for that CAN channel, values for its Bit Timing Registers (BTR) or sampling point percentage, and most importantly, the **Acknowledge On** checkbox. This checkbox is used to configure CANoe to actively (checked) or passively (unchecked) communicate with that CAN channel. To actively monitor the CAN bus means that the CAN Controller participates like a real node on the network. The channel is capable of both transmitting and receiving CAN messages. All the CAN messages that flow through this channel will be acknowledged before any type of filtering.

2. The **Filter** section allows you to set up a pair of mask and code for both standard (11-bit) and extended (29-bit) CAN messages. By default, the acceptance filter on CAN Channel 1 is always set up to pass all messages, while CAN Channel 2 always has its messages blocked except identifier 0x7FF. Because only one pair of mask and code is available per channel, not all messages can be filtered. In this case, those unexpected messages are filtered later in the **Simulation Setup** window. The mask and code settings take place in the CAN Controller. Some people find this an advantage because the CANoe software has fewer messages to analyze, which improves its performance.

3. The **Options** section has two checkboxes: **Activate TxRq** and **Activate Bus Statistics**. **Activate TxRq** (transmit request) setting is for CAN messages. By default, this box is unchecked because most users do not require this feature. This feature displays the TxRq messages with a time stamp in the **Trace** window of CANoe. These are requests to send messages by CANoe that have not yet been transmitted onto the CAN bus. If they have been transmitted on the CAN bus, they would be Tx messages. The **Activate Bus Statistics** checkbox is to enable or disable the **Bus Statistics** window of CANoe. The window updates cyclically in milliseconds with the time you specify in the box.

Note: More information about each setting is available by clicking on the **Help** button.

21.1.4 Set Up Analysis Functions

Setting up analysis functions is the fourth step in building a CANoe configuration. The configuration processes for the database and the panels require separate programs to edit and will be discussed in the following two chapters. Analysis functions are set up in the **Measurement Setup** window of CANoe (Figure 55).

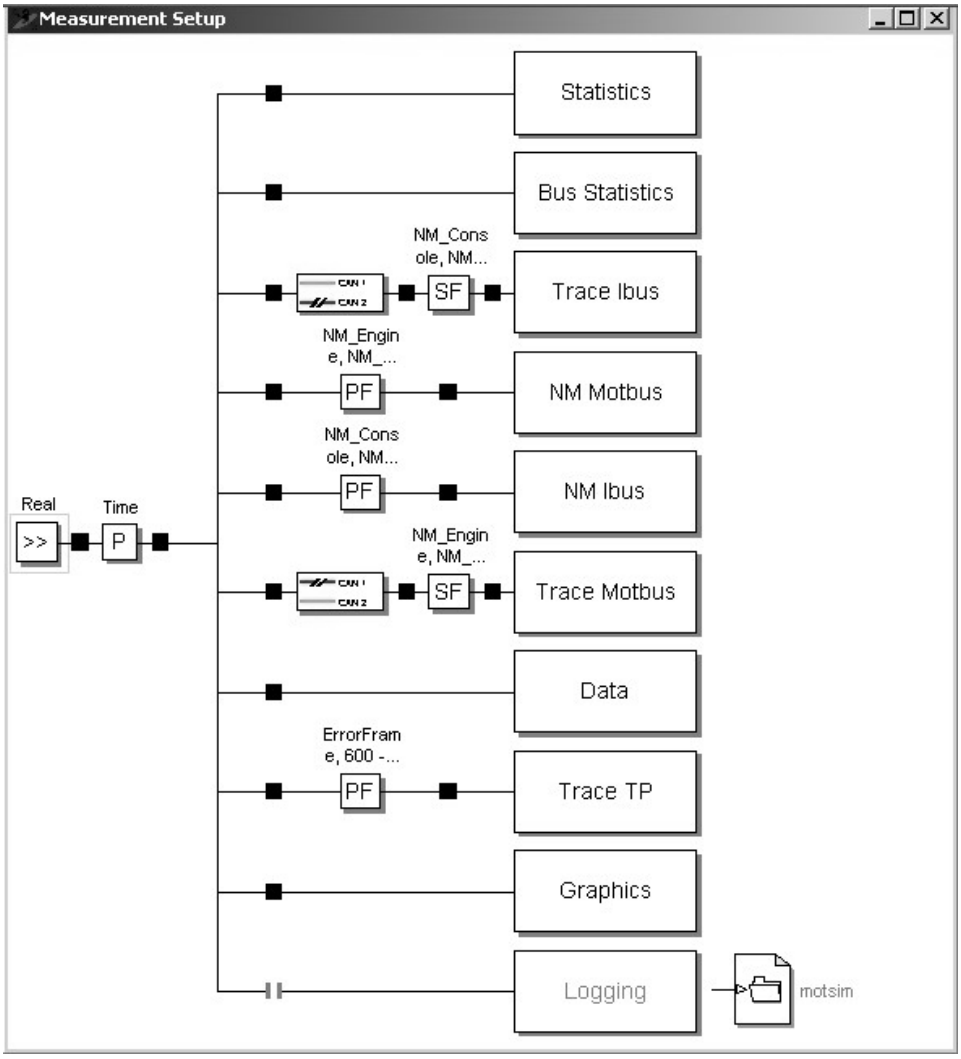

Figure 55 – CANoe Measurement Setup Window

The **Measurement Setup** window consists of a data flow diagram. Connection lines and branches are drawn between the individual blocks to clarify the data flow. CAN messages flow from the Real block on the left to each individual analysis block on the right. The data path consists of black “hotspots”, where you can add function blocks and data sinks such as filters and user-defined analysis programs (CAPL). As a result, the data flow can be configured in many ways to suit many different types of analysis tasks.

Note: Every analysis block, except the Logging block, has a configurable window in which received data is displayed. More information about each analysis window is available on Chapter 3, A Brief Introduction to CANalyzer.

The method by which you configure this **Measurement Setup** window depends on what you intend to do. New function blocks, such as filters and generator blocks, can be inserted at each hotspot to carry an analysis function. If you wish, you can activate or deactivate any function block or path (using the hotspots) in this window by first selecting it and then pressing the space bar. Copying and pasting are also possible by using your right mouse button.

21.1.5 Working with the Analysis Windows

Every configuration has analysis blocks represented by a corresponding window in the **Measurement Setup** window. Available windows include **Statistics**, **Bus Statistics**, **Graphics**, **Data**, and **Trace**. You can add more than one window for each analysis block to suit your needs, except for **Bus Statistics**. For example, users usually add a second **Trace** window for messages received from a second CAN bus, a second **Data** window for a different node on the CAN bus, or a second **Graphics** window for graphing bus statistics data, instead of signals received from the CAN bus. To insert a new analysis block in the **Measurement Setup** window, right-click on any of the analysis blocks and select the one to insert. This popup menu is also used for deletion; however, at least one of each type of analysis block must exist. A new window is created for every new analysis block; the opposite applies for deletion.

Note: You may add or delete the Logging block the same way; however, these analysis blocks do not come with an online evaluation window.

Every analysis block has to be configured to perform bus analysis through either its window or its configuration menu, such as the Logging block. For example, the Graphics block does not automatically graph the signals received from the bus unless the user-defined signals are associated.

See Figure 56 below for and example of a configured CANoe **Graphics** Window.

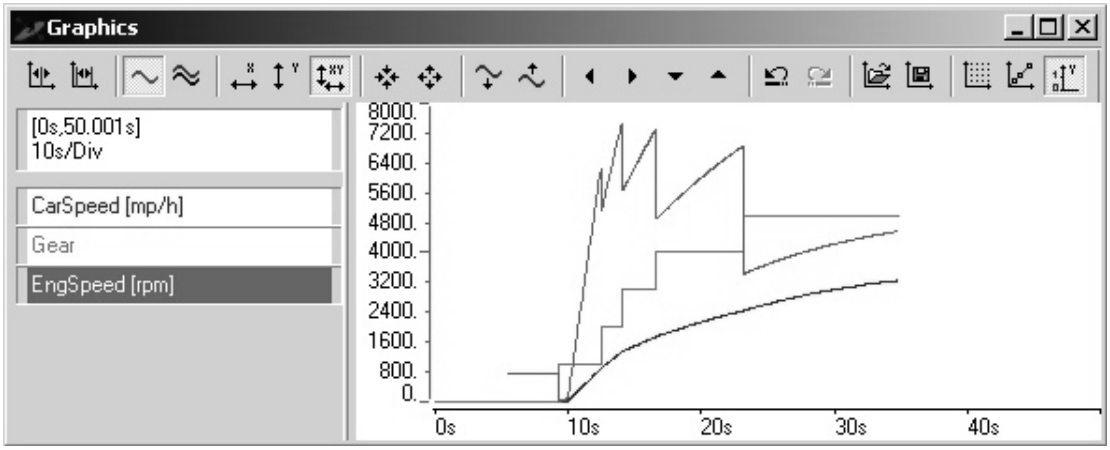

Figure 56 – Configured CANoe Graphics Window

22 Using CANdb++ to Create a Database

Messages on a network can easily be organized and modified by a database. Distribution and reusability are two major advantages, because specifications are contained in one easily manageable database. The system design group of many companies uses the Vector database platform known as CANdb to develop database files. These database files are then made available corporate-wide, which greatly increases the compatibility and functionality of their CAN systems.

22.1 CANdb++ Database Development

The basic steps in learning how to create a database and associate it to CANoe are as follows:

1. Start the CANdb++ program from CANoe
2. Define attributes (optional)
3. Define value tables (optional)
4. Define nodes
5. Define messages
6. Define signals
7. Establish associations
8. Define environment variables
9. Save the database with a specific file name
10. Associate the database with CANoe

While you may see a considerable amount of detail inside the database program, stay focused on the basic steps. The CANdb++ online help is a great source to get more information should you come to a standstill. See Using Databases with CAPL – Chapter 5 for more information on each object types.

- The **CANdb++ Editor** shows all of the database objects grouped in the **Overview** window, where almost all database operations are performed (as shown in Figure 57).

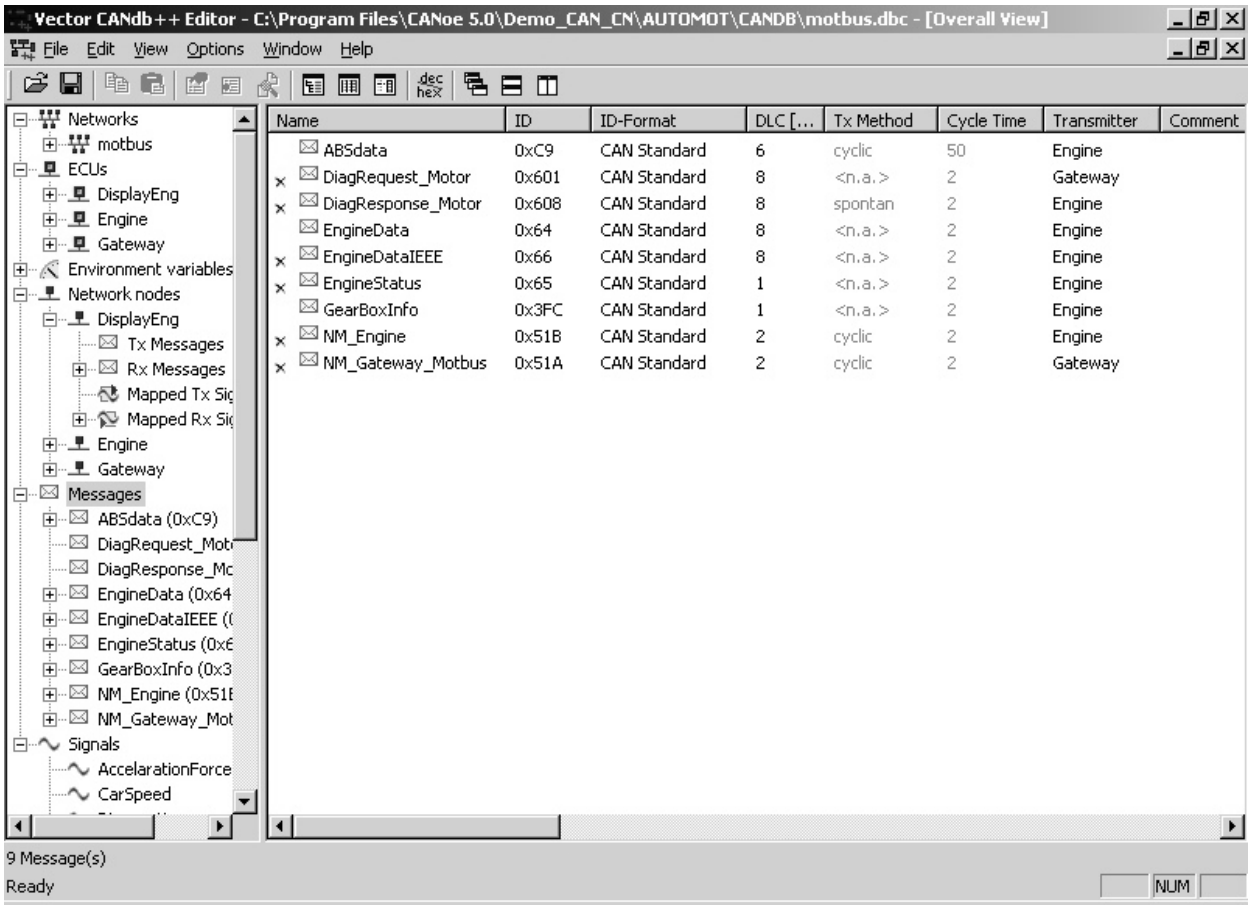

Figure 57 – CANdb++ Editor - Overall View

The following database objects are all categorized there:

- Network name
- ECUs
- Nodes
- Messages
- Signals
- Environment variables (not directly related to the CAN bus network, but are global variables that enable users to make inputs and outputs to the CAN bus through a CAPL program).

Note: Using the CANdb++ program is very simple if you keep one rule in mind: always right-click on the mouse to get the pop-up menu if you get stuck. For example, you have to right-click to add a new object.

22.2 Starting the CANdb++ Program

While it is possible to start CANdb++ as an independent application program, it is sometimes easier to load it from inside the CANoe or CANalyzer environment. Starting it from CANoe creates a link to that database, and saving the database automatically updates the CANoe configuration. On the CANoe toolbar, click on the shortcut icon for CANdb++ that looks like a small network with interconnected red boxes. If a database has already been associated with the current CANoe configuration, that database will be opened. Otherwise, you have to create a new database.

Note: Using the CANdb++ program is very simple if you keep one rule in mind: always right-click on the mouse to get the pop-up menu if you get stuck. For example, you have to right-click to add a new object.

22.3 Defining Attributes

When creating a database, it is most efficient to first define the attributes before creating any database objects. Follow the procedure below to create an attribute:

1. Open the attribute definition window by going to **View** and select **Attribute Definitions**.
2. Right-click to add a new attribute.

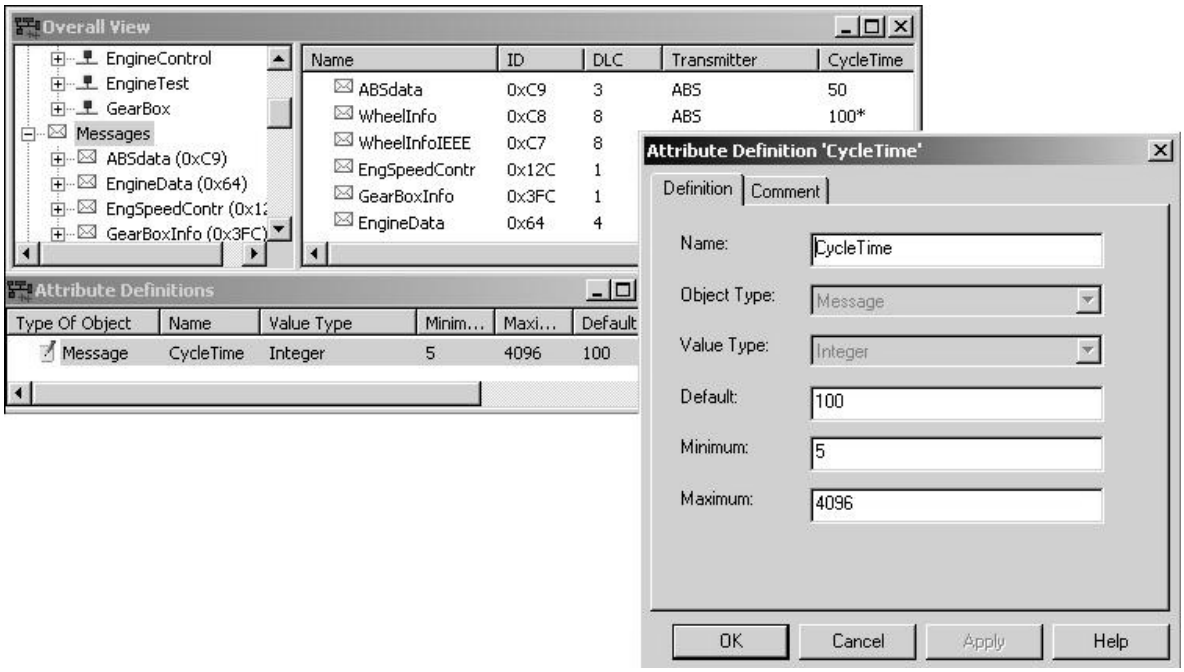

Figure 58 – "Cycle Time" Attribute Definition

22.4 Defining Value Tables

Although value tables are defined in databases, they are rarely referenced in CAPL. Value tables, however, make numeric data easy to read in CANalyzer and CANoe's analysis windows.

To define a value table in CANdb++, go to the **Value Table** window via the menu command **View → Value Tables**.

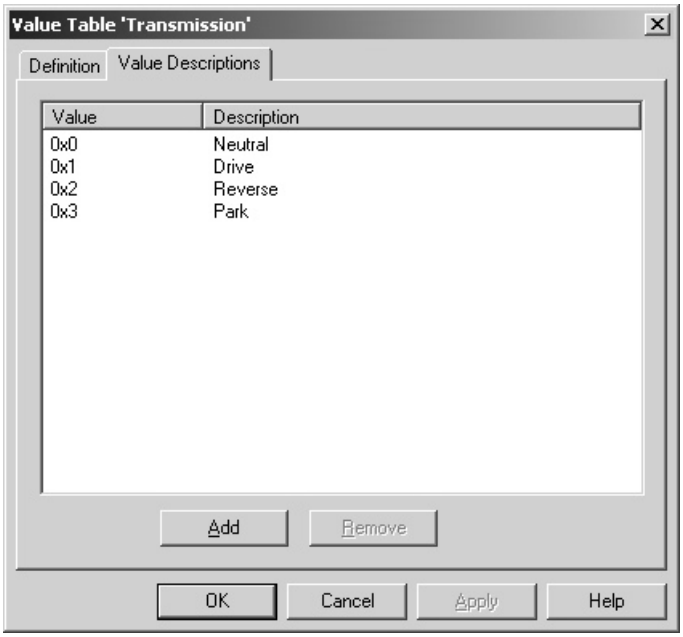

Figure 59 – Automatic Transmission Values in a Value Table

22.5 Defining Nodes

To define a node, short acronyms are usually used to replace its name. For example, TCM is the acronym commonly used for transmission control module. The acronym used depends on the user. In the above example, we could find that another organization uses the acronym TCM to represent the temperature control module instead. To solve the dilemma, a comment is usually added in the database to explain the acronym.

To define a node in CANdb++:

1. Right-click on the **Overview** window where it says Network Nodes
2. Select New.

22.6 Defining Messages

When defining a message, three parameters have to be known: its numeric identifier, identifier size (11-bit or 29-bit), and its data length code. Other message properties are sometimes easily ignored but crucial to a database: message transmission and reception node, message attributes, and signals. Signal associations are usually not done at this stage because they have not been created yet.

To define a message in CANdb++:

1. Right-click on the **Overview** window where it says **Messages**
2. Select **Add**.

For a more complete understanding of message structure, see Chapter 26 – Appendix A: Introduction to CAN Communications.

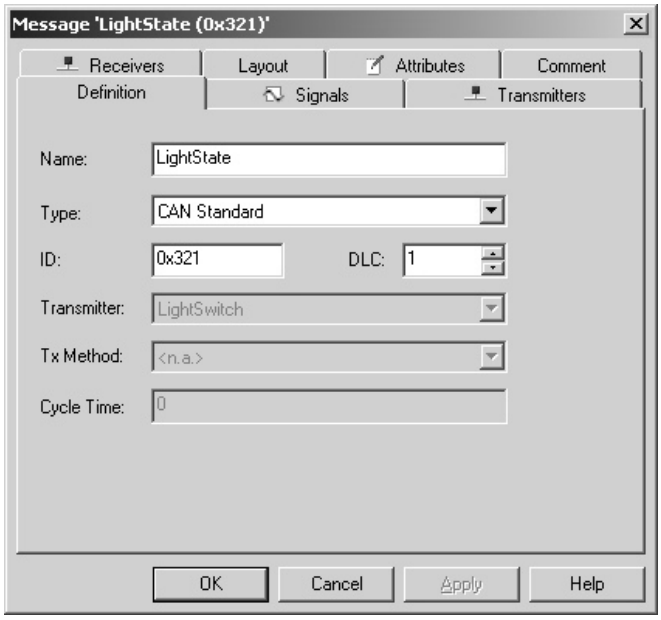

Figure 60 – A Message in the Database

22.7 Defining Signals

A signal is from 1 bit up to 64 bits in size; therefore, a message may data ranging from containing no signals (DLC = 0) or up to 64 signals (1 bit each for DLC = 8). Unfortunately, most applications only allow a maximum of 32 bits for a signal.

When defining a signal, you must give it a symbolic name, define its size in number of bits, choose either Intel or Motorola format, and its value type (data type). Optional properties include a factor and offset to convert the signal raw value into an engineering value, its maximum and minimum value for display only, and a value table to display the numeric signal value in symbolic form (Figure 61).

You may choose to click on the **Messages** tab to assign the signal to a message(s), or you may wait until all the signals have been defined. To define a signal in CANdb++:

1. Right click on the **Overview** window where it says **Signals**
2. Select **New**.

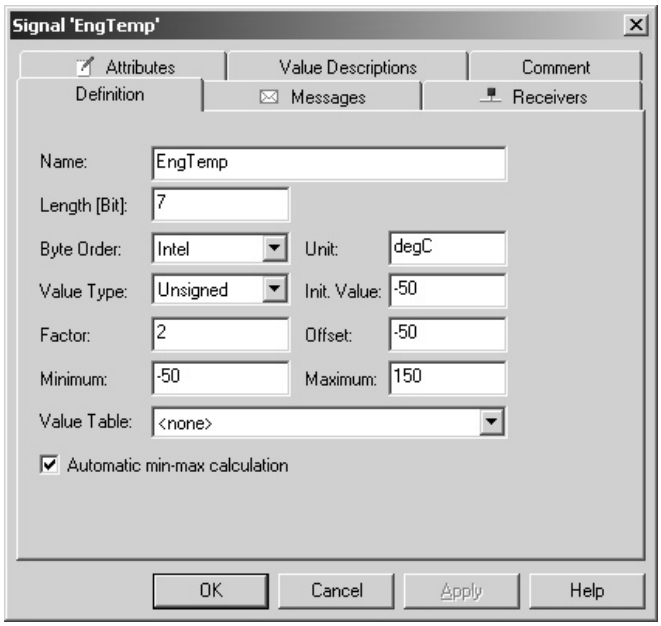

Figure 61 – Signal in the CANdb++ Database Editor

After you have defined all the signals, there are many ways in which you can assign the signals to their message(s). You can drag and drop in the **Overview** window, or edit either the message or the signal to assign. Any way you choose to make this assignment, you must make sure the starting bit for each signal in a message is correct. To confirm this, edit the message and go to its Layout tab to view its signal associations (Figure 62).

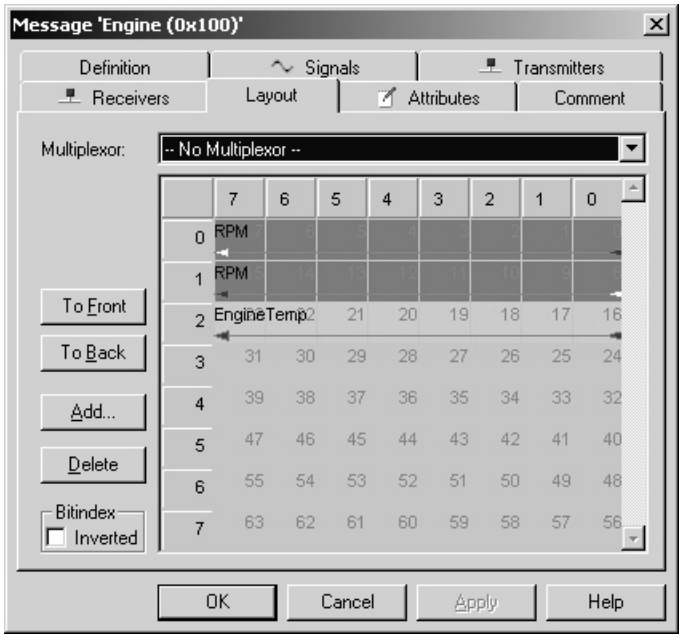

Figure 62 – Layout of a Message in CANdb++

22.8 Establishing Object Associations

Defining every object in the database is only half the task. To realize the advantages of having a database for data interpretation in CANoe and for CAPL programming, you must make all the object associations by associating the value tables to the corresponding signals, signals to the corresponding messages, and messages to the corresponding nodes, and so on. Making these associations requires absolute understanding of the system. Without these associations, CAPL code will not generate correctly, and object relations will not be shown in CANalyzer/CANoe.

Note: Generally, a member of the development team makes these associations.

22.9 Defining Environment Variables

When defining an environment variable, you must give it a name, identify its value type (data type), and specify the kind of access it has. You may also assign a value table to it, and this value table may be the same one you assign to a signal. Other optional properties include its default, minimum, and maximum value (see Figure 63). No object associations are needed for environment variables.

To define an environment variable in CANdb++:

1. Right click on the **Overview** window where it says `Environment Variables`
2. Select `New`.

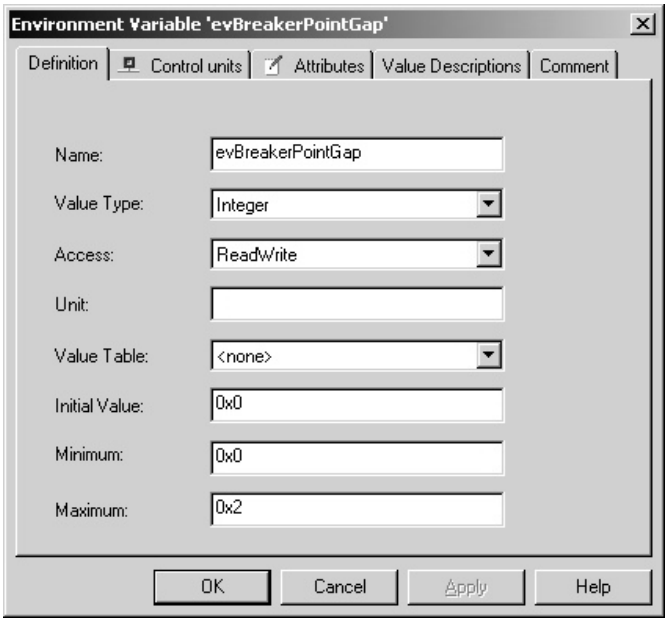

Figure 63 – Properties of an Environment Variable in CANdb++

22.10 Saving the Database

The database cannot be used in CANoe and the CAPL Browser until it is saved. If the CANoe measurement is running and then the database from that configuration is saved again, the saved information will not take effect until the measurement is restarted. You do not have to restart CANoe or reopen the CAPL Browser for the changes in the database to become effective.

22.11 Associating a Database

Associating a database with CANoe is a fairly simple, yet crucial task. It is crucial because you must know to which CAN channel the database belongs. A CANoe configuration may consist of more than one network, each connecting to a CAN channel. Also, if a database already is associated on a particular network and you then add a second one, you must ensure the message identifiers in both databases are unique. CANoe can be configured to perform a consistency check when you associate a second database. If you ignored a warning from the consistency check and a message having an ID described in more than one database is received from the CAN bus, the message defined in the first database associated for that CAN channel is used.

In CANoe, a database can only be associated in the **Simulation Setup** window by right-clicking on where it says **`Databases`** and select **`Add`** (see Figure 64).

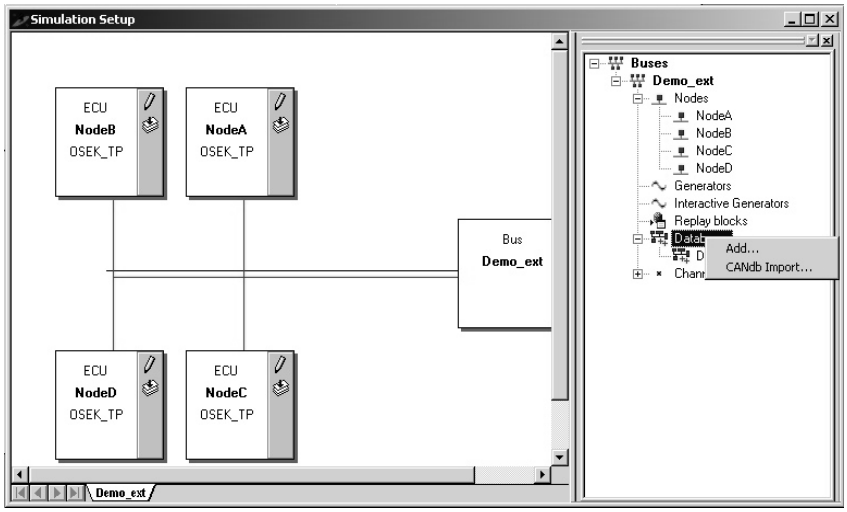

Figure 64 – Associating a Database in CANoe

23 Using the Panel Editor to Create Panels

In automotive engineering, it is always helpful to be able to visualize your work. CANoe provides the unique opportunity to work with simulated network nodes. The network node models in the **Simulation Setup** window you created in CAPL are able to react to external events, such as activating a switch. CANoe also provides you with the option of creating your own user-defined interfaces called panels and integrating them into the program.

Graphic panels are an important and useful feature of CANoe because they function as control panels where the user can change the values of variables during a simulation or a measurement.

23.1 What are Panels?

A panel is a user-configurable graphical interface. Every control that can be placed on a panel on a panel is called an element. These elements – buttons, switches, sliders, and so on – can be placed anywhere on the panel to create the look and feel of automotive control panels (Figure 65).

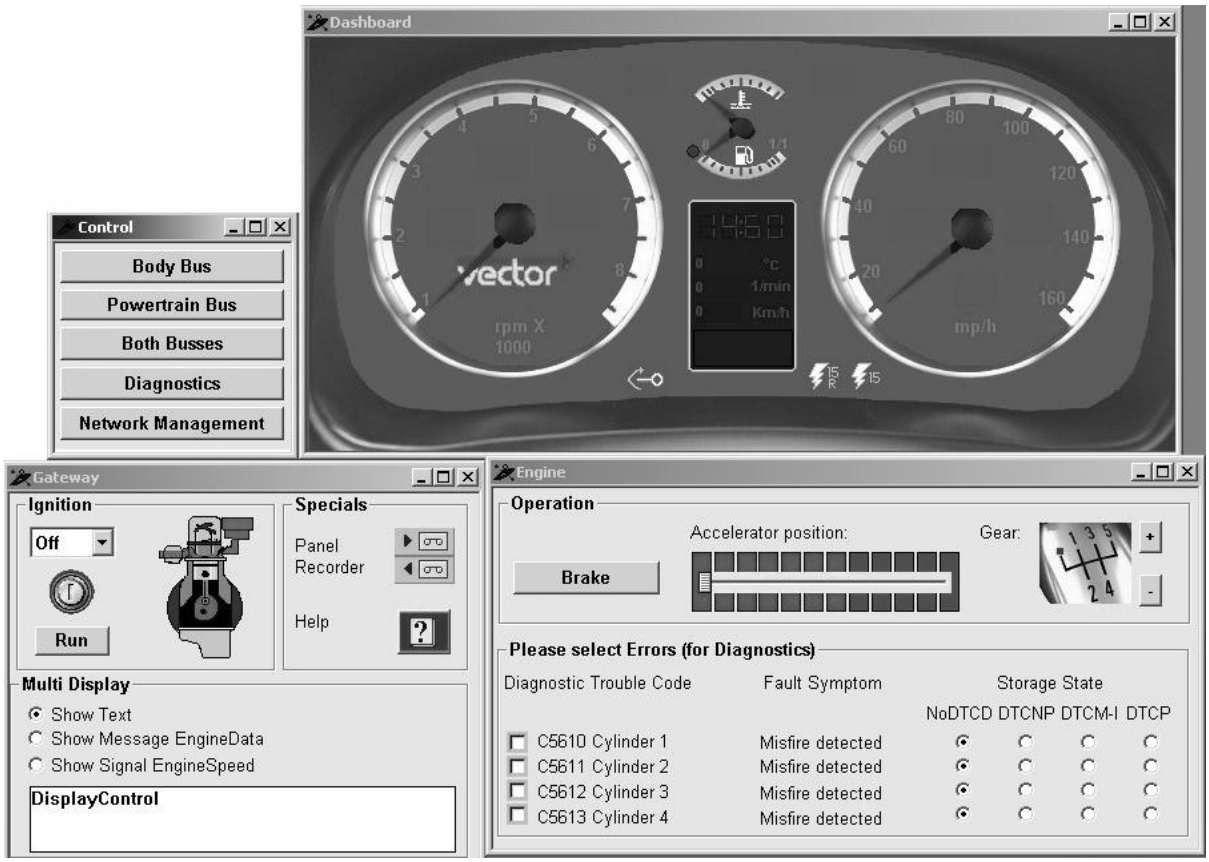

Figure 65 – Panels Created Using the Panel Editor

The **Panel Editor** is the environment used for creation and arrangement of elements for displaying and controlling variables with values to be changed, and for assigning variables (either environment variables or signals) to all of these control and display elements. You can also put text and static bitmaps on the panel.

You can interact and change the values of these variables during a measurement by manipulating the elements on the panels. The network node models react to changes in the variable values and then execute the appropriate actions (for example, sending out a message).

Not only can the user change the values of the variables; but CAPL programs can also be used to change the values of the associated variables when certain events occur. This value change can then be visualized on the panel using display elements.

23.2 Advantages of Using Panels

Vector panels support ActiveX controls, which are component programs that can be reused by several application programs in a computer. An implemented ActiveX control is a dynamic link library (DLL) module. ActiveX controls run in so-called containers, which are application programs that use Microsoft's Component Object Model program interfaces. This reusable component approach reduces application development time and improves the quality of the program.

Another clear advantage of using panels is that they increase your ability to interpret input and output bus data, such as on a dashboard. When a node receives a message, the node can display the data bounded by the message onto a panel. If you need to set a control on the panel and want to send the data onto the bus using a message, you just need to configure the corresponding network node (CAPL program).

You can also select which of the panels you create to open permanently, and which ones to open automatically at startup. These settings can be made when you associate the panels to the CANoe configuration. CANoe also supports multiple configurable desktop environments. This is a great advantage if you have several panels in the configuration that overcrowd the Windows desktop.

You can define multiple desktops in CANoe to display the panels in groups instead of accessing each panel from the Windows taskbar.

23.3 Requirements

To use the **Panel Editor**, you must have CANoe. Because the **Panel Editor** is a separate application, it can be run in stand-alone mode; however, it is much easier to create panels in a **Panel Editor** opened through CANoe because the variables to be associated with the elements are in the database(s) associated to CANoe.

23.3.1 Database Association

To use panels in CANoe, you must use a database. The database has all the variables defined for the elements on the panel. Because the database has already been associated to CANoe, you will save time and effort when creating panels because the **Panel Editor** recalls the variables defined in the database.

A database is only optional before panels are created. You can open the **Panel Editor** in stand-alone mode and create a panel, but if there is no database, you will have to save the panel to be reconfigured at a later time. This is extra work because it is a two-step process. Eventually you have to reload the unfinished panel and link the variables to the controls.

23.3.2 Environment Variables

The variables that CANoe uses to describe external events are called environment variables. These are the same variables that can be used to associate with panel elements. Environment variables can be used both as input and output variables of network nodes, such as sensor signals and actuator signals. Environment variables can represent switch positions, temperature, or engine speed. Character string variables can also be used to store text.

Environment variables are defined in the database by the following system parameters:

- Symbolic name of the environment variable

- Value type (data type) of the environment variable
  Integer – Signed 32 Bit Integer
  String – ASCII String
  Float – 64 Bit Float
  Data – Byte field of a specified length (the length is set in the "Length" attribute)

- Access rights that Electronic Control Units (ECUs) have when they access environment variables.

  - **Unrestricted**: All ECUs have read and write access.
  - **Read**: The assigned ECUs can read the environment variable. (The variable represents a sensor.)

  - **Write**: The assigned ECUs can write to the environment variable. (The variable represents an actuator.)
  - **Read/Write**: The assigned ECUs can read from and write to environment variables.

- Unit (the physical quantity of the environment variable)

- Initial Value (only available for environment variables of value type "Integer" or "Float")
  Initial value of the environment variables at the start of a simulation or measurement.

- Minimum and Maximum (only available for environment variables of value type "Integer" or "Float")
  Minimum and Maximum of a valid physical value.

- Length of the data field in bytes (only available for environment variables of value type "Data")

You can interact and change the values of these environment variables during a measurement by manipulating the controls on the panels. The network nodes will then react to the changes you made to the environment variable and then execute the appropriate actions, such as sending out a message. The new value is retrieved by the **getValue()** function in CAPL.

CAPL programs can also change the values of associated environment variables when certain events occur using the **putValue()** function. The change will visually affect the corresponding element in the panel.

23.4 Creating a New Panel

There are three methods available to start the **Panel Editor**:

- The shortcut button on the CANoe toolbar,
- The menu command **Panel → Configure panels…**,
- Select a panel in the **Configured Panels** section of the Panel configuration dialog and then click the **Edit** button. This will ensure that the databases of your CANoe configuration will be automatically associated to the **Panel Editor**.

If you are trying to open an existing panel for modification, right-click on the title bar of the panel and select **Edit**. If the database does not contain the environment variable names associated with the panel's indicator and control elements, or if there is an element not associated to a variable yet, you will receive a warning message (Figure 66). In this case you should make sure you associated the correct database, start the **Panel Editor** directly from CANoe, or find and associate the element(s) to a variable.

Figure 66 – Panel Editor Warning Message

23.4.1 Properties of the Panel

To set the properties of your panel in the **Panel Editor**, click on the **Window Setting** button on the toolbar or select **Options → Window Setting…**.

The first thing your panel needs is a title. This is not just to distinguish each panel from another, but helps the CANoe operator by displaying the name of the open panels on the Windows taskbar at the bottom of the screen. CAPL functions like **putValueToControl()** also need a panel title passed as a parameter to process.

This dialog is also where you specify the size of the panel, the fonts for each element, and the background color.

23.4.2 Using Elements

Elements allow a variety of value formats to be assigned. In addition to hexadecimal, decimal, string, and bitmaps, symbolic values from value tables of the CAN database (CANdb) can be selected and/or displayed on the panel. The **Panel Editor** already has a set of pre-defined elements that are ready for you to use. Each element has a corresponding button on the toolbar that you can click and place onto the panel.

23.4.2.1 Control Elements

Control elements are used to represent values from a sensor. They let you change the values assigned to environment variables. Elements that are used exclusively for this purpose are:

- Push button
- Radio button
- Switch

23.4.2.2 Display Elements

Display elements are used to represent values from an actuator and allow let you display changes in the values assigned to environment variables. Elements that are exclusively for display are:

- Meter
- Multi-control

23.4.2.3 Static Elements

Static controls are elements to which no environment variables can be assigned. They include:

- Bitmap
- Frame
- Text (label)

23.4.2.4 Input/Output Controls

Many elements are not listed above because they can be used either to regulate input values resulting from your programmed actions or because they display the resulting output values. These elements are:

- Bitmap push button
- Bitmap switch/indicator
- Hex Editor
- Input/Output Control
- Slider
- Value table box

Note: You can change the width and height of these controls when you position them by dragging while you hold down the left mouse button.

23.5 Configuring Elements

Every new element must be configured for inputs or outputs – or both.

To configure a new element after dragging it onto the panel, right-click on it and select **`Configure this element`** or just double left-click on it. This opens a dialog where you specify the value type or data type (integer, float, string), as well as the font and background color (if possible).

23.5.1 Associating Elements

Before an element can be used for input / output, it must be associated with either an environment variable or a signal defined in the database. To do this, select either **`EnvVar`** (environment variable) or **`Signal`** from the drop-down list (see Figure 67). A list of available environment variables or signals will immediately appear in a list directly above your

selection, from which you select the environment variable / signal you want. If the environment variable / signal you are looking for is not there, click the **Select** button to find it. The choice of environment variable or signal also depends on the value type (integer, float, string), selected by the radio buttons in the **Type** area of the dialog (see Figure 67).

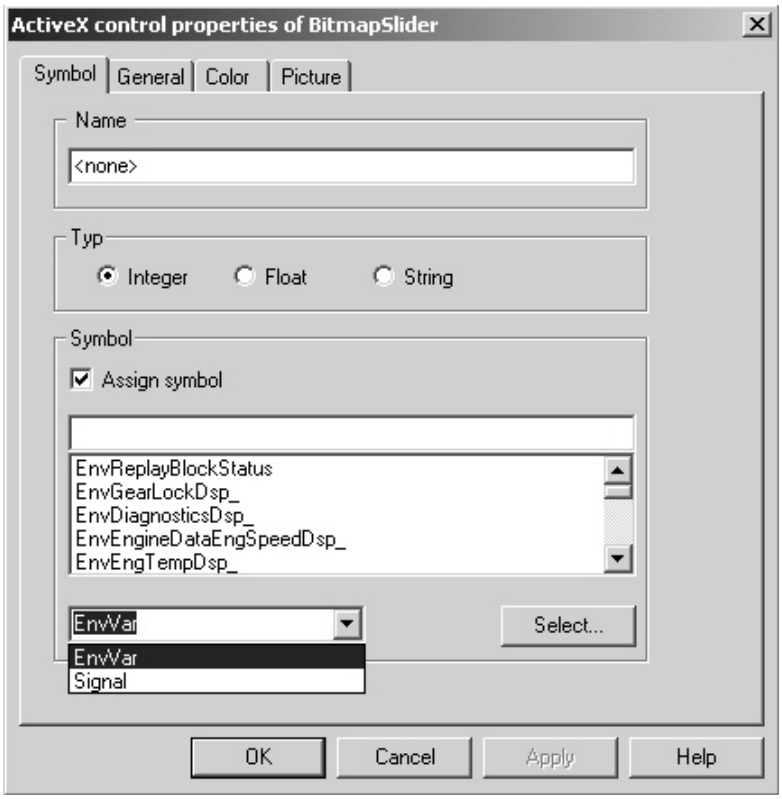

Figure 67 – Associating a Variable to an Element

Environment variables can be used as both inputs and outputs, but signals can only display output values.

23.5.2 Alarms

An alarm state is a condition reached when a value of a panel element goes beyond its pre-defined maximum or minimum range. Under such conditions, a visual representation of that panel element is displayed.

The Configuration Dialog for some elements, such as meters, asks you to configure Alarm States (Figure 68).

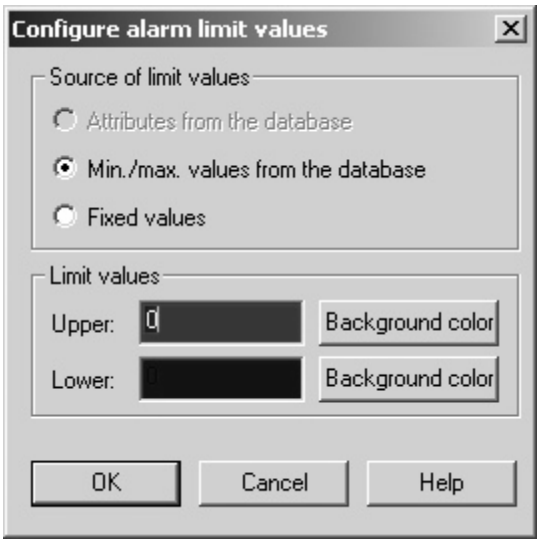

Figure 68 – Setting an Alarm State Range

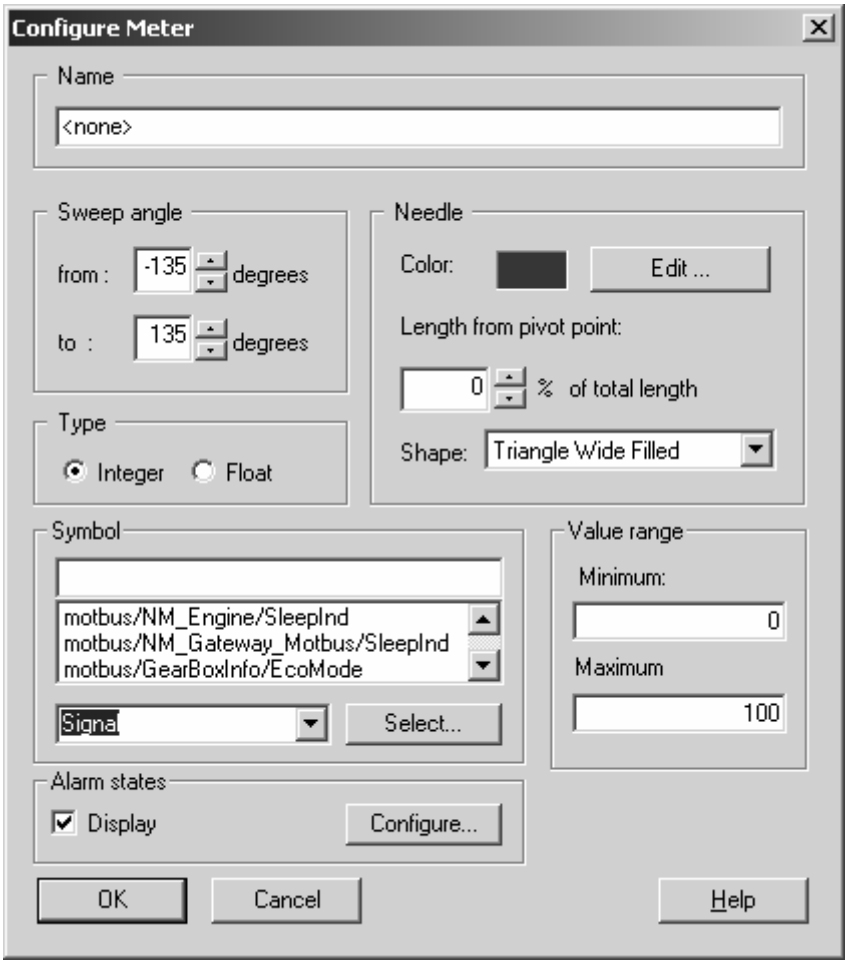

Figure 69 – Enable Alarm Via the "Display" Checkbox

The Alarm States feature makes it possible for different panel elements to respond to a situation in which a value is above or below a previously defined value range (Figure 69).

This in turn makes it possible to change the visual representation of the panel element that displays the value when a specific limit is reached. The extent to which the panel element display changes can be adjusted to reflect the fact that an upper or lower value range limit has been exceeded. For example, it is possible to respond to a situation in which a value is above or below a previously defined value range for a meter element by changing the color of the meter.

Table 49 lists the panel elements that can display alarm states:

Element	Properties that can change color depending on its value limit
Meter	Meter color
Slider	Background color
Input/Output box	Font Font color Background color
ActiveX Controls	Depend on the possible properties

Table 50 – Elements with Alarm States

23.6 Using Bitmaps

Only bitmap (*.bmp) images may be used on panels. (Do NOT use JPEG, PCX, or any other graphic format.) Bitmaps, reached by right-clicking on the new element and selecting **`Configure this element...`**, are assigned to elements in the **Configure** dialog.

Static elements usually use a bitmap to display as a background. Most of the time, bitmaps are used to represent the state or condition of a variable. These bitmaps are normally used for display purposes, but can be used as inputs by clicking on the bitmap to change its state. Changing the display state, therefore, changes the value of the associated environment variable.

The elements that support I/O and bitmaps fall into two categories:

- Two-state elements
  For example, Bitmap pushbutton and bitmap switch/indicator with two states

- Multi-state elements
  For example, Bitmap switch/indicator with more than two states

23.6.1 Creating a Dynamic Bitmap

You can create and edit bitmap files with any bitmap editor (for example, Paintbrush, Photoshop, or Visio).

A bitmap file for a two-state element always consists of 3 separate bitmap pictures, each with the same height and width, arranged to be horizontally adjacent to one another (see Figure 70). The rectangle on the left shows the inactive state. This state only appears when you edit the panel or when no environment variable has been assigned to the element. Just to the right of this is the logical OFF state (state 0), which corresponds to the element value 0. The third rectangle is the logical ON state (state 1), which corresponds to the selected or predefined switch value of the environment variable (normally with a value of 1).

Figure 70 - A Two-State Switch Bitmap

A bitmap file with n-states consists of n + 1 rectangular partial bitmaps of the same height and width, arranged to be horizontally adjacent. The inactive state is shown on the upper left bitmap. The n switch values are located in the n partial bitmaps to its right.

The surrounding color of the bitmaps can be anything, and you can fill the areas around the bitmap to appear transparent to the background, as shown in Figure 71, through an element property.

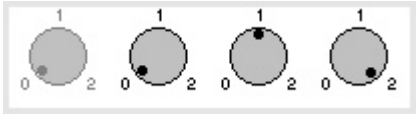

Figure 71 - A Three-Stage Switch Bitmap

Note: When creating your own bitmaps, be sure to save them as uncompressed Windows bitmaps. Some graphics drivers cannot process specially compressed bitmaps, which may result in undesirable optical effects or even system crashes.

Only one file is allowed per element, so the bitmap states have to be saved under one file. It is advisable to save all bitmap files belonging to a panel in a separate subdirectory. If you wish to forward a panel to another user, you must also forward the bitmap files.

23.7 Associating Panels to CANoe

After the elements on your panel have been configured and saved, they must be associated with the desired CANoe configuration. To associate you panels:

1. Select the menu item **Panels → Configure Panels**.
2. Click **Add**
3. Select your panel (it has the extension *.cnp.).

If the panel has already been added and you made some adjustment using the **Panel Editor**, you do not have to add the updated panel to CANoe again. The associated panel will automatically update once you've saved it in the **Panel Editor**. Shown below in Figure 72 is an example of panel association.

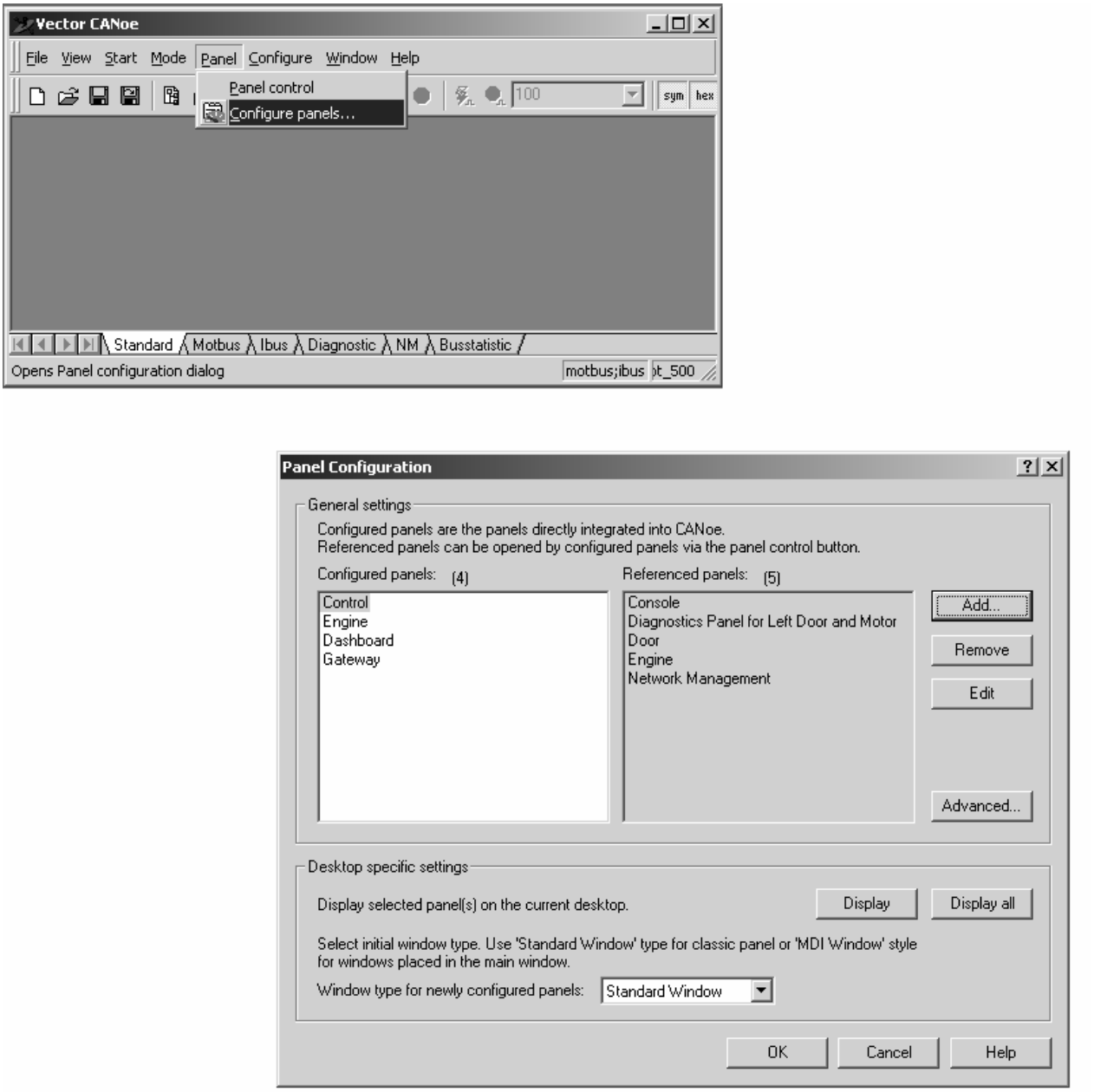

Figure 72 - Associating Panels to CANoe

24 Introduction To CAPL DLLs

CAPL is an event-based language, meaning its code is written to react to the occurrence of events, such as the press of a switch. It allows you to simulate network and node behavior and to perform bus testing and analysis. Its syntax is similar to C, but CAPL does not require any file associations to compile or run like C/C++. It does, however, have a set of internally defined functions to do all the simple calculations and specific tasks. When these functions cannot help you perform a complex calculation or task, the best solution is to implement the calculation or task in a CAPL DLL.

24.1 DLL – Dynamic Link Library

According to the Microsoft Corporation, a DLL is a library of routines loaded and linked into applications at run time. This file is stored and run separately from the applications that use it. An application will map the DLL into its address space when a process is starting. This file is often given a ".dll" extension, and is usually implemented in C or C++.

24.2 CAPL DLLs

CAPL programs can be created, modified, and maintained so they can interface with a wide variety of inputs, outputs, and other functions and files. There are two ways to enhance the CAPL programming environment. The first way is to use COM, a standard defined by the Microsoft Corporation for communication between different software components.

The second way is with DLLs. DLLs are primarily made to have inputs and outputs with other applications or files, for example, an outside program that has generated a data file and cannot be brought into CANalyzer or CANoe to distribute the data onto the bus. CAPL has two sets of file I/O functions to handle ASCII files with INI file format, but it lacks file searching abilities. For file structures that require the use of pointers and complex algorithms, DLLs are generally implemented to handle file I/O and data exchange using C/C++.

Note: When implementing a DLL for CAPL, all methods are available to access system resources (memory management, hardware, and so on). However, because of the magnitude of any problems that might arise, only **experienced programmers** should use this option. Vector does not provide any support for creating DLLs.

Note: CAPL also supports functions from a node layer DLL. These DLLs are different than the CAPL DLL mentioned in this chapter. Node layer DLLs are capable of simulating node behavior such as a CAPL program. It can handle both message transmission and reception. Node layer DLL creation is beyond the scope of this book. (Find more information on node layer DLL in Section 9.4.9)

24.3 Performance

Unless absolutely necessary, a DLL should not be considered to replace CAPL functions (this involves user-defined DLLs only). A DLL may slow down CANalyzer/CANoe software performance. If the functions inside the DLL are called in the **Simulation** or **Transmission Branch** (for example, the **Simulation Setup** window of CANoe and above the **Send Block** of the **Measurement Setup** window in CANalyzer), then the code will run in a higher priority thread. This could have a negative impact on the measurement.

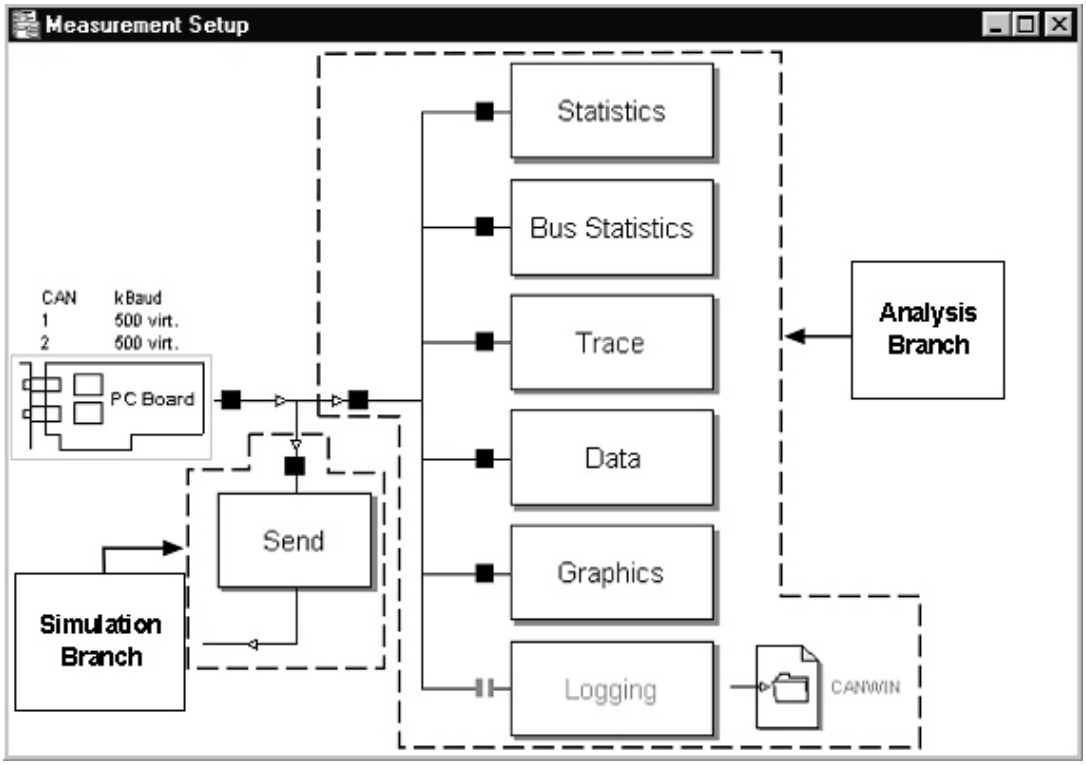

Figure 73 – The Simulation Branch of the CANalyzer Measurement Setup Window

below are some requirements for DLL function calls in the Simulation or Transmission Branch:

- File accesses and calls to the Windows API are forbidden.
- “Hidden” Windows API calls are also forbidden. The cores of many linked libraries are based on Windows APIs.
- Using dynamic memory management based directly or indirectly on GlobalAlloc[], (for example, new, new[], delete, delete[], malloc, free, and so on), is not recommended. Instead, it is better to reserve memory statically before the measurement starts.
- Because parameters are passed by 32-bit registers, Compilers must be operated on at least a “386” processor.

To speed up function calls, the return value is usually stored first in the EAX register (if your processor has one) before it is returned. The EAX register should be used as much as possible to receive and to load into memory, because it works faster than other registers. Windows’ APIs also use the EAX register to return values to the caller. Below is an example:

```
long CAPLEXPORT far CAPLPASCAL appMultiply(long x, long y)
{
    long z = x * y;
    MOVE_EAX(z);          // function defined in the header file
    return z;
}
```

24.3.1 Using Microsoft Visual C++ to Implement a CAPL DLL

This chapter does not intend to teach the Visual C++ language or its components. It is recommended that the user have some C++ background, and, of course, knowledge of CAPL. All the examples in this document use the tools listed below:

- Microsoft Visual C++ 6.0 Professional Edition (SP5)
- CANalyzer or CANoe Professional Edition with CAPL programming (Latest version is preferred)
- CAPL Browser (Latest version is preferred)

24.4 CAPL Export Table

Functions in a DLL are exported to CAPL using a function table called the CAPL_DLL_INFO_LIST, which can be found in the file “capldll.cpp”. This table has three columns.

Column	Description
1	Contains the name of the function to be exported. This function name is used in CAPL to call the corresponding function defined in the DLL.
2	Contains the address of the function to be called in the DLL. These functions must be defined in the source code and exported using the PASCAL calling convention.
3	Contains the descriptions of the called function parameters and return values.

Table 51 – CAPL_DLL_INFO_LIST Export Table

Table example:

```
CAPL_DLL_INFO CAPL_DLL_INFO_LIST[] =
{
    #ifdef _MSC_VER {CDLL_VERSION_NAME,(CAPL_FARCALL)CDLL_VERSION, CAPL_DLL_CDECL,
    0xabcd, CDLL_EXPORT},
    #else {CDLL_VERSION_NAME,(CAPL_FARCALL)CDLL_VERSION, CAPL_DLL_PASCAL, 0xabcd,
    CDLL_EXPORT},
    #endif

    {"Dec2Hex",   (CAPL_FARCALL)appDecToHex,    'V', 2, "LC", "\x0\x1"},
    {"Hex2Dec",   (CAPL_FARCALL)appHex2Dec,     'D', 1, "C", "\001"},
    {"Bin2Dec",   (CAPL_FARCALL)appBin2Dec,     'D', 1, "C", "\1"},
    {"Dec2Bin",   (CAPL_FARCALL)appDec2Bin,     'V', 2, "LC", "\000\001"},
    {"Str2Long",  (CAPL_FARCALL)appStr2Long,    'L', 1, "C", "\001"},
    {"Str2Int",   (CAPL_FARCALL)appStr2Int,     'L', 1, "C", "\001"},
    {"Str2Flt",   (CAPL_FARCALL)appStr2Flt,     'F', 1, "C", "\001"},
    {0,0}
};
```

The last parameter is used to end the table. It must be **{0,0}**.

You probably notice that the third column of the export table is subdivided into four more columns to define the function parameters and return values. These subcolumns have the following meanings:

Sub-Column	Description
1	Data type of return value
2	Number of function parameters
3	Function parameters’ data type
4	Depth of an array (only for functions with a parameter declared as an array)

Table 52 – CAPL Export Table Sub-Columns

The first sub-column denotes the data type of the function’s return value, and it is explained in Table 52. It is important to make sure that the same data type is defined in CAPL.

Symbol	Data Type
V	void
L	long
D	unsigned long
I	int (only in connection with array size !=0)
W	unsigned int (only in connection with array size !=0)
B	unsigned char (only in connection with array size !=0)
C	char (only in connection with array size !=0)
M	message
T	msTimer
F	float (means double in 8-byte 80387 format)

Table 53 – Function Parameter and Return Value Data Types

The third sub-column also uses the table above to denote the data type of the function parameters. If one or more parameters is defined as an array, the fourth sub-column is required to specify the depth of the array. For example, if a function has a 1-dimensional linear array parameter, the fourth sub-column is used to specify all the parameters. This sub-column uses “escape sequences” – character combinations consisting of a backslash (\) followed by a letter or by a combination of digits. If a backslash precedes an unknown character, such as numbers, the compiler will handle the unknown character as the character itself. Below are some examples:

"\x0\x1" = “\0\1” = “\000\001”

The first one is in hexadecimal notation, the second is undefined (the zero and one should be considered as the character itself), and the third is in octal notation. These examples show that a couple of functions have two parameters. The first one is not an array, but the second is a linear array.

24.5 Project Configuration

Before implementing a DLL, you can use the DLL demo project that came with CANalyzer or CANoe as a base to create new applications, or you can create one from scratch with the header file CDLL.H. The demo project and header file are located in the **…EXEC32/CapIdll** directory. To create one from scratch using Microsoft Visual C++, create a new Win32 Dynamic Link Library project and include the header file.

24.5.1 Compiling the DLL

If the demo project is used as a base for making a DLL, do not forget to change the output directory under “Project Settings”. Only the C DLL.h header file is needed, and any additional files depend on the project. The name of the DLL can be anything, but it must be 8 characters or less. CANoe and CANalyzer will not accept a compiled DLL if the name has more than 8 characters.

24.6 Linking the CAPL DLL

There are two ways to link a CAPL DLL, as mentioned in Chapter 18. Adjusting the initialization file is not recommended, so use the association method in CANalyzer or CANoe by making the menu selection **Configure → Options**. If the DLL is associated correctly, there will be a notification in the **Message** window of the CAPL Browser after compiling a CAPL program as shown in Figure 74.

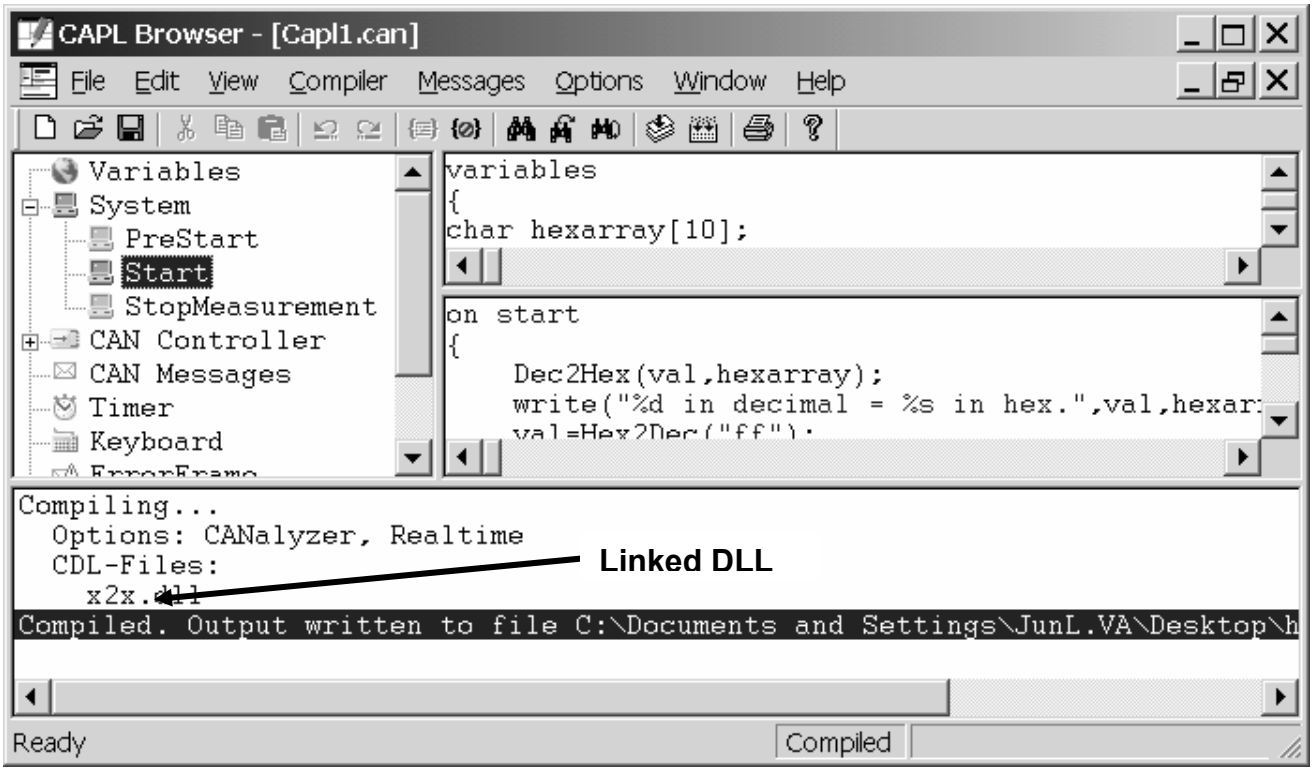

Figure 74 – A CAPL Program Utilizing the x2x DLL

24.6.1 CAN.ini File

If you do not have Version 5.0 or later of CANoe or CANalyzer (on the tool, go to **Help → About**), you must associate the DLL through the CAN.ini file. This initialization file is a Microsoft Corporation standard, and it contains a majority of the CANalyzer or CANoe settings. The new DLL has to be associated to one of the CAPLDLL entries in the `[SIMULATION]` section. Up to 25 DLLs are allowed. Below is an example of how to associate a CAPL DLL called x2x.dll.

```
[SIMULATION]
CAPLDLL = x2x.dll
;CAPLDLL2 =
;CAPLDLL3 =
```

The semicolon at the beginning must be removed for CANalyzer or CANoe to enable the new setting.

24.6.2 DLL Search Sequence

The following sequence is used to search for the CAPL DLL. The file is searched for in the following directories and paths:

- The EXEC32 directory
- The active working directory
- The Windows system directory
- The Windows directory
- The computer's active search path

Note: It is strongly recommended that the DLL be stored in the EXEC32 directory for performance reasons.

24.6.3 Demo DLL and Source Code

The following section explains how a demo DLL is used to convert numbers based on the following data types:

- Decimal
- Hexadecimal
- Float
- Binary
- String(array)

24.6.3.1 CAPL Code

To test the DLL, use the following CAPL code:

```
char hexarray[10];
long val = 255;
int intval;
float fltval;

Dec2Hex(val, hexarray);
write("%d in decimal = %s in hex.", val, hexarray);

val = Hex2Dec("ff");
write("FF in hex = %d in decimal", val);

val = Bin2Dec("11111111");
write("11111111 in binary = %d in decimal", val);

Dec2Bin(val, hexarray);
write("%d in decimal = %s in binary.", val, hexarray);

val = Str2Long("98 miles per hour");
write("98 miles per hour = %d in decimal(long)", val);
intval = Str2Int("98 miles per hour");
write("98 miles per hour = %d in decimal(int)", intval);

fltval = Str2Flt("98.34 miles per hour");
write("98.34 miles per hour = %g in IEEE(float)", fltval);
```

When the above code is executed, the **Write** Window in CANalyzer or CANoe will display the returned values as shown in Figure 75.

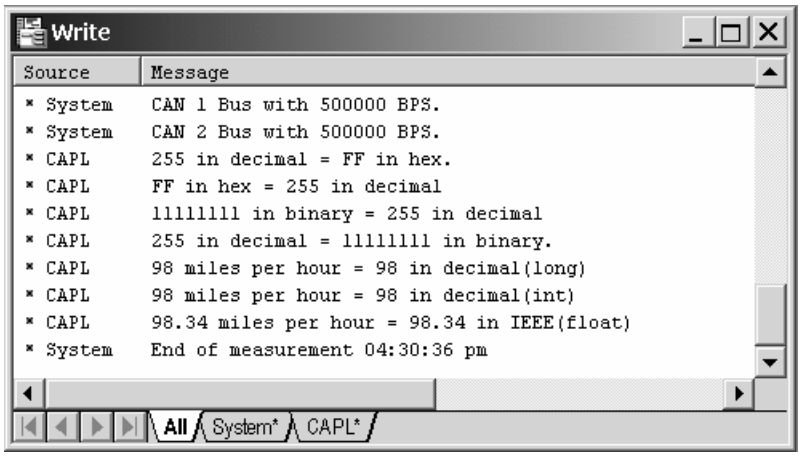

Figure 75 – Return Values in Write Window

24.6.4 C++ Code

The following code is use to build the DLL that defines seven functions for data type conversion.

```
/*----------------------------------------------------------------------------
|
| File Name: capldll.cpp
|
|       Example of a CAPL DLL implementation module.
|-----------------------------------------------------------------------------
|           A U T H O R   I D E N T I T Y
|-----------------------------------------------------------------------------
|   Author            Initials
|   ------            --------
|   Andreas Klegraf    Kf           Vector Informatik GmbH
|   Marc    Lobmeyer   Lm             Vector Informatik GmbH
|   Thomas  Riegraf    Ri           Vector Informatik GmbH
|   Hans    Quecke     Qu           Vector Informatik GmbH
|-----------------------------------------------------------------------------
|           R E V I S I O N   H I S T O R Y
|-----------------------------------------------------------------------------
| Date         Ver  Author            Description
| ---------    ---  ------            -------------------------------------------
| 6.10.2003    1.0  Jun Lin                  Creation
| 6.10.2003    1.0  Jun Lin                  Microsoft Visual C++ 6.0
|-----------------------------------------------------------------------------
|           C O P Y R I G H T
|-----------------------------------------------------------------------------
| Copyright (c) 1994 - 1998 by Vector Informatik GmbH. All rights reserved.
 ----------------------------------------------------------------------------*/

    #define USECDLL_FEATURE
    #include "cdl\cdll.h"
    #include <stdio.h>
    #include <math.h>                          //added for POW()

    void CAPLEXPORT far CAPLPASCAL appDecToHex(unsigned long decValue, char hexr[])
    {
        long hex[10];
        long temp = 0, i;

        for (i = 1; i < 10; i++)
        {
                hex[i] = decValue % 16;
                decValue = decValue / 16;
                if (decValue == 0)
                        break;
        }

        for (int j = 0; j <= 10; j++)
        {
                switch (hex[i])
                {
                        case 0: hexr[j] = '0';
                                        break;
                        case 1: hexr[j] = '1';
                                        break;
                        case 2: hexr[j] = '2';
                                        break;
                        case 3: hexr[j] = '3';
```

```
                                        break;
                        case 4: hexr[j] = '4';
                                        break;
                        case 5: hexr[j] = '5';
                                        break;
                        case 6: hexr[j] = '6';
                                        break;
                        case 7: hexr[j] = '7';
                                        break;
                        case 8: hexr[j] = '8';
                                        break;
                        case 9: hexr[j] = '9';
                                        break;
                        case 10: hexr[j] = 'A';
                                         break;
                        case 11: hexr[j] = 'B';
                                         break;
                        case 12: hexr[j] = 'C';
                                         break;
                        case 13: hexr[j] = 'D';
                                         break;
                        case 14: hexr[j] = 'E';
                                         break;
                        case 15: hexr[j] = 'F';
                }
                i = i - 1;
                if (i <= 0)
                        break;
        }
}

DWORD CAPLEXPORT far CAPLPASCAL appHex2Dec(char hexr[])
{
        DWORD value = 0;
        int flag = 0, s = 0, i, j;

        for (i = 0; i <= 9; i++)
        {
                if (hexr[i] == NULL)
                        break;
        }

        for (j = i - 1; j >= 0; j--)
        {
                switch (hexr[j])
                {
                        case '0':          // do nothing
                                         break;
                        case '1': value = value + DWORD(1.0 * pow(16, s));
                                         break;
                        case '2': value = value + DWORD(2.0 * pow(16, s));
                                         break;
                        case '3': value = value + DWORD(3.0 * pow(16, s));
                                         break;
                        case '4': value = value + DWORD(4.0 * pow(16, s));
                                         break;
                        case '5': value = value + DWORD(5.0 * pow(16, s));
                                         break;
                        case '6': value = value + DWORD(6.0 * pow(16, s));
                                         break;
```

```
                case '7': value = value + DWORD(7.0 * pow(16, s));
                          break;
                case '8': value = value + DWORD(8.0 * pow(16, s));
                          break;
                case '9': value = value + DWORD(9.0 * pow(16, s));
                          break;
                case 'A':
                case 'a': value = value + DWORD(10.0 * pow(16, s));
                          break;
                case 'B':
                case 'b': value = value + DWORD(11.0 * pow(16, s));
                          break;
                case 'C':
                case 'c': value = value + DWORD(12.0 * pow(16, s));
                          break;
                case 'D':
                case 'd': value = value + DWORD(13.0 * pow(16, s));
                          break;
                case 'E':
                case 'e': value = value + DWORD(14.0 * pow(16, s));
                          break;
                case 'F':
                case 'f': value = value + DWORD(15.0 * pow(16, s));
                          break;
                default: flag = 1;
            }
            s = s + 1;
    }
    if (flag == 1)
            return 0;                                   // return 0 for invalid character
    else
            return value;
}

DWORD CAPLEXPORT far CAPLPASCAL appBin2Dec(char hexr[])
{
    DWORD value = 0;
    int flag = 0, i, j, s = 0;

    for (i = 0; i < 32; i++)
    {
            if (hexr[i] == NULL)
                    break;
    }

    for (j = i - 1; j >= 0; j--)
    {
            if (hexr[j] == '1')
                    value = value + DWORD(pow(2,s));
            else if (hexr[j]!='0')
            {
                    value = 0;
                    break;
            }
            s = s + 1;
    }
    return value;
}

void CAPLEXPORT far CAPLPASCAL appDec2Bin(long intValue, char hexr[])
```

```
{
    for (int i = 0; i < 32; i++)
    {
        hexr[i] = (intValue % 2) + 48;
        intValue = intValue / 2;
        if (intValue == 0)
            break;
    }
}

long CAPLEXPORT far CAPLPASCAL appStr2Long(char hexr[])
{
    long value;
    value = atol(hexr);
    return value;
}

long CAPLEXPORT far CAPLPASCAL appStr2Int(char hexr[])
{
    long value;
    value = atoi(hexr);
    return value;
}

double CAPLEXPORT far CAPLPASCAL appStr2Flt(char hexr[])
{
    double value;
    value = atof(hexr);
    return value;
}

#ifdef __BCPLUSPLUS__
    #pragma warn -pin
#endif

CAPL_DLL_INFO CAPL_DLL_INFO_LIST[] =
{
    #ifdef _MSC_VER {CDLL_VERSION_NAME, (CAPL_FARCALL)CDLL_VERSION, CAPL_DLL_CDECL,
    0xabcd, CDLL_EXPORT },
    #else {CDLL_VERSION_NAME, (CAPL_FARCALL)CDLL_VERSION, CAPL_DLL_PASCAL, 0xabcd,
    CDLL_EXPORT },
    #endif
    {"Dec2Hex",         (CAPL_FARCALL)appDecToHex,          'V', 2, "LC", "\x0\x1"},
    {"Hex2Dec",         (CAPL_FARCALL)appHex2Dec,           'D', 1, "C", "\001"},
    {"Bin2Dec",         (CAPL_FARCALL)appBin2Dec,           'D', 1, "C", "\1"},
    {"Dec2Bin",         (CAPL_FARCALL)appDec2Bin,           'V', 2, "LC", "\000\001"},
    {"Str2Long",        (CAPL_FARCALL)appStr2Long,          'L', 1, "C", "\001" },
    {"Str2Int",         (CAPL_FARCALL)appStr2Int,           'L', 1, "C", "\001" },
    {"Str2Flt",         (CAPL_FARCALL)appStr2Flt,           'F', 1, "C", "\001" },
    {0,0}
};

#ifdef __BCPLUSPLUS__
    #pragma warn +pin
#endif

// Return the table so the compiler knows which functions are available

#ifdef _MSC_VER
unsigned long CAPLEXPORT __cdecl caplDllGetTable(void)
```

```
{
    return (unsigned long)CAPL_DLL_INFO_LIST;
}
#else
CAPL_DLL_INFO far * CAPLEXPORT pascal caplDllGetTable(void)
{
    return CAPL_DLL_INFO_LIST;
}
#endif

#ifdef __WIN32__
int WINAPI DllEntryPoint(HINSTANCE, unsigned long reason, void*)
{
    switch (reason)
    {
        case DLL_PROCESS_ATTACH:
            return 1;                   // Indicates that the DLL was initialized successfully
        case DLL_PROCESS_DETACH:
            return 1;                   // Indicates that the DLL was detached successfully
    }
    return 1;
}
#else
int FAR PASCAL LibMain(HINSTANCE, WORD, WORD, LPSTR)
{
    return 1;       // Indicates that the DLL was initialized successfully
}

int FAR PASCAL WEP (int)
{
    return 1;
}
#endif
```

24.6.5 Troubleshooting CAPL DLL Errors in the CAPL Browser

Error: No NODE LAYER (for example, NWM) available; DUMMY DLL LOADED!!
Some CANoe configurations require additional expansions for the CAPL nodes (for example, Network Management). These so-called node layers exist as run-time libraries (DLLs). This error occurs if CANoe does not find the necessary DLL. Ensure that the DLL is located in the CANoe system directory **…CANoe\EXEC32**.

Error: Invalid type in DLL <dllname.dll> <functionname>
This error is caught while compiling a CAPL program. The function defined in the CAPL Export Table is incorrect. Most of the time the error is the result of the parameter settings in the CAPL Export Table.

Warning: Could not open <dllname.dll>
In most cases, the user has incorrectly typed in the name of the DLL in the CAN.ini file. The name is case-sensitive, the extension must be included, and it must be 8 characters or less. Other than that, ensure that the DLL is in the EXEC32 directory of CANoe or CANalyzer.

24.6.6 CAPL DLL Questions and Answers

Below are two commonly asked questions and answers regarding the troubleshooting of DLL-related issues:
Q: After building the DLL, I don’t see it in the working directory or the project folder.
A: If you use the demo project, the output DLL is in the EXEC32 directory. The default name is capldll.dll. To specify a different path, change the output directory under Project Properties.

Q: After compiling my CAPL program, why is there an error message that says “Error: ambiguity between dllname.dll”?
A: A node layer DLL can be associated on a node-by-node basis in the database (*.dbc) file. This problem occurs if you integrated the same DLL into CAPL through the CAN.ini file.

25 Introduction to COM

The purpose of this chapter is to give a general introduction to the use of CANoe/CANalyzer as a COM server. The basic technical aspects and possibilities of the COM server functionality will be presented. The COM interface is often used to remotely command CANoe or CANalyzer to begin testing or share data through CAPL. Any tools that support COM will do. For the purpose of training, Microsoft Visual Basic will be used for the examples in this chapter.

CANoe and CANalyzer are Vector's simulation and analysis tools for the CAN bus. Both applications can also be used for other bus systems such as LIN, MOST, and FlexRay. CANalyzer is the right tool to analyze, observe, and simulate data traffic, while CANoe can be used to create a whole functional model of the network system. This model can be used in the entire development process of the network from planning to final testing.

In some applications, CANalyzer or CANoe is not essential to emulate all the tasks. A second tool in this case may be necessary. Tools such as LabView, Simulink, Matlab, and so on are often used together with CANalyzer or CANoe. To permit data sharing among the tools, the Component Object Model is introduced in CANalyzer and CANoe.

Different programming and scripting languages can be used to access the COM server functionality of CANoe and CANalyzer. All scripts used must be based on Microsoft's Windows Script software component. Scripting languages such as VBScript or JScript, that are parts of Microsoft Windows 98, Windows NT, and Windows 2000 packages, can be used as scripting languages. Such scripts can be used to create test reports in Microsoft Excel or Microsoft Word. Programming languages such as Microsoft Visual Basic, Microsoft Visual C++, and Borland Delphi can be used to create user-specific applications.

25.1 COM – Component Object Model

COM is a standard defined by Microsoft for communication between different software components. Different programming languages can be used to create such components. These components can be built by different software developers, independently from each other.

25.2 CANoe/CANalyzer as a COM server

CANoe and CANalyzer have a build-in COM interface beginning with version 3.0. The following goals can be achieved by using this COM server functionality:

- Exchange of data between CANoe or CANalyzer and other external programs or applications suites. The external programs must support the COM technology as well.
- Automation of test sequences with customer-specific control panels.
- Remote control of measurements using CANoe or CANalyzer. The remote control of CANoe or CANalyzer over a network is also possible.

25.3 COM Server Registration

The COM Server is already registered when CANoe/CANalyzer is installed. If the installation directory has been moved, a new registration is necessary. To do this, open the MS-DOS prompt and change to the installation directory (EXEC32). Type the following command:

```
canw32 –regserver      // for CANalyzer
canoe32 –regserver     // for CANoe
```

To delete the registration of the COM Servers type in the following:

```
canw32 –unregserver  // for CANalyzer
canoe32 –unregserver // for CANoe
```

25.4 Using Microsoft Visual Basic to Access the COM server

All the examples and illustrations in this chapter reference the following tools:

- Microsoft Visual Basic 6.0 Professional Edition (SP5)

- CANalyzer or CANoe Professional Edition with CAPL programming (Latest version is preferred)
- CAPL Browser (Latest version is preferred)
- Panel Editor (Latest version is preferred)

This chapter does not teach the Visual Basic language or its interface components. It is, however, recommended that you have some Visual Basic background.

25.4.1 Project Configuration

Microsoft Visual Basic is a Rapid Application Development (RAD) tool that allows the application engineer to create impressive applications for the Microsoft Windows platform with relatively little effort. In the description of the COM server functionality, examples written for Microsoft Visual Basic will be used to demonstrate in detail how to program for the CANoe/CANalyzer COM server.

The CANoe/CANalyzer COM server consists of several objects that are related to each other, using a certain object hierarchy. Each COM server object is defined using three elements that give the application engineer access to the COM server's functionality. These elements include:

- Object Properties (for example, **CANController.Baudrate**)
- Object Methods (for example, **Measurement.Start**)
- Object **Events** (for example, **Application.OnQuit**)

Basic knowledge of the COM server object hierarchy will help the application engineer obtain a good overview of the COM server functionality. A description of the COM server object hierarchy can be found in CANoe and CANalyzer's online help system.

To have access to the COM server from Microsoft Visual Basic, the Microsoft Visual Basic project has to be configured to use the CANalyzer (or CANoe) type library. In Microsoft Visual Basic, this is accomplished using the project's references, as can be seen in Figure 76.

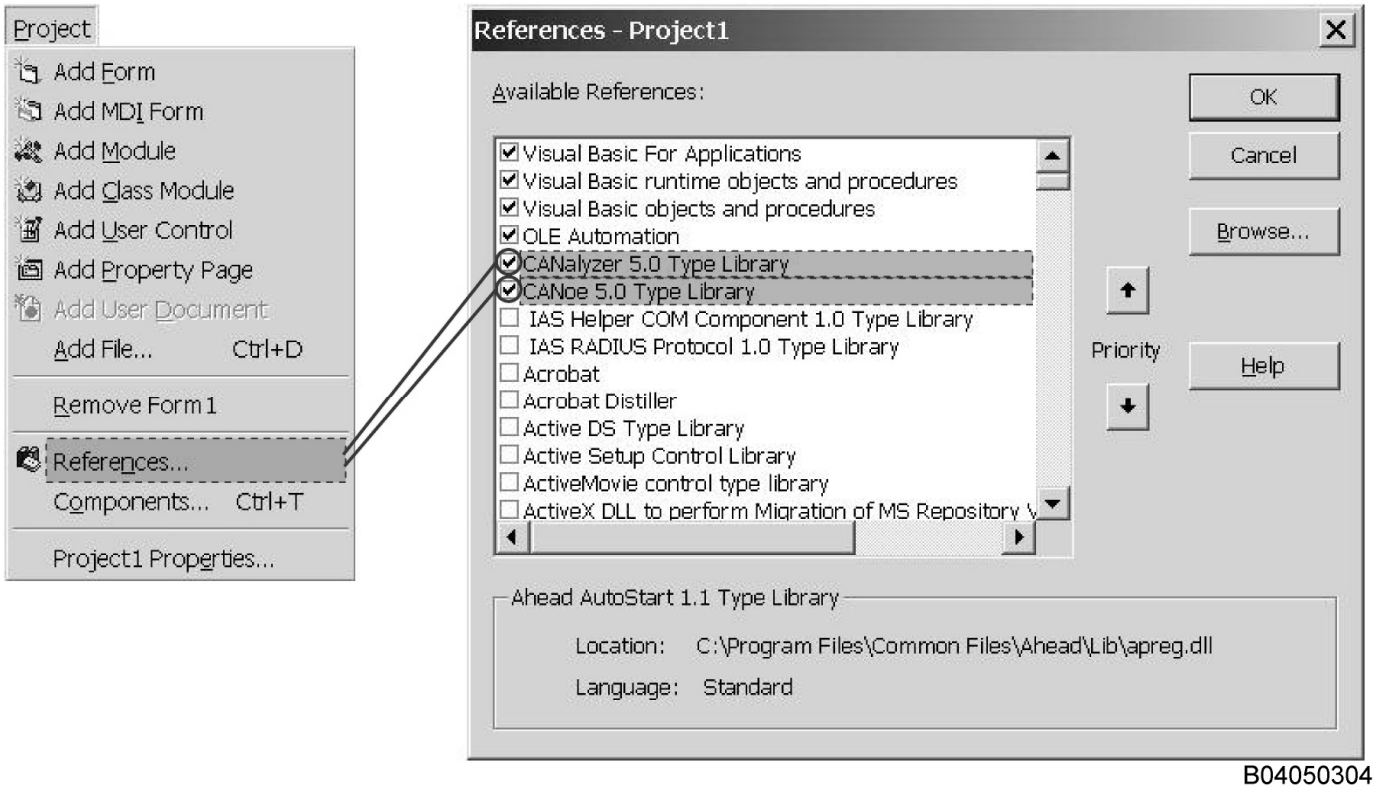

Figure 76 – Referencing the CANalyzer Type Library

25.5 Controlling a Measurement

Once a Microsoft Visual Basic project has been configured to use the CANalyzer (or CANoe) type library, the application has access to the COM server. In almost every situation, the COM server is used to perform a measurement using the application engineer's CANalyzer or CANoe configuration. A global variable is needed in Visual Basic through which the COM server's **Application** object can be accessed. This global variable should be declared as:

```
Dim gCanApp As CANalyzer.Application
```

The next step is to initialize this global variable to let Microsoft Visual Basic know that this global variable will be used to access the COM server's **Application** object. This can be accomplished using the following statement:

```
Set gCanApp = New Application
```

If CANalyzer has not yet been started, the previous line of code will load CANalyzer. As mentioned earlier, the COM server is used primarily to perform a measurement with the application engineer's CANalyzer or CANoe configuration. This requires the COM server load the application engineer's configuration. The **Open** method of the COM server's **Application** object can be used for this with the following statement:

```
gCanApp.Open ("C:\Program Files\CANalyzer\Demo_CL\motbus.cfg")
```

If the configuration contains CAPL code, it is a good practice to re-compile the CAPL code before a new measurement is started. This ensures that the most recent CAPL code will be used in the measurement. Invoking the compilation process is also possible using the COM server. In this case, the **Compile** method of the **CAPL** object can be used. The **CAPL** object is also part of the **Application** object (refer to the COM server object hierarchy). This means that the already initialized **gCanApp** variable can be used to access the **CAPL** object. This can be accomplished using the following statement:

```
gCanApp.CAPL.Compile
```

At this point everything is set up and ready for a new measurement. A new measurement can be started (and stopped) using the **Measurement** object. Similar to the CAPL object, the **Measurement** object is also part of the **Application** object. This means that the **Measurement** object can be accessed through the **gCanApp** variable. The methods **Start** and **Stop** of the Measurement object can be used to start and stop a new measurement. For example:

```
gCanApp.Measurement.Start
```

The COM server also contains functionality to quit the CANalyzer application. This is done using the **Quit** method of the **Application** object. The following code checks the **Running** property of the **Measurement** object to find out if a measurement is running and if so, stops the measurement before quitting the CANalyzer:

```
If gCanApp.Measurement.Running Then
    'stop the CANalyzer measurement
    gCanApp.Measurement.Stop
End If

'quit the CANalyzer application
gCanApp.Quit
```

25.6 Accessing Communication Data

The communication data used by a CANalyzer configuration is stored in a CANdb database. A CANdb database file has a .DBC file extension and is used as a common database for the entire Vector tool chain.

In Chapter 5 – Using Databases with CAPL, we learned that a CANdb database contains the communication matrix of a network using messages and signals. Each CAN message can contain up to 8 data bytes and consists of one or more signals.

Figure 77 displays a screenshot of an example CANdb++ database, that has been opened using the CANdb++ Database Editor. The CAN message **EngineData** is selected in the left window. The right window shows the signals in the selected message. The CAN message **EngineData** is defined in the CANdb database as a message with identifier 64 (hex) and the length of the message is 4 data bytes. Signals are located inside these 4 data bytes as can be seen in the right window. Signal **EngSpeed** is defined as a signal with a length of 16 bits (2 bytes) and is located in byte 0 and byte 1 of the CAN message **EngineData**.

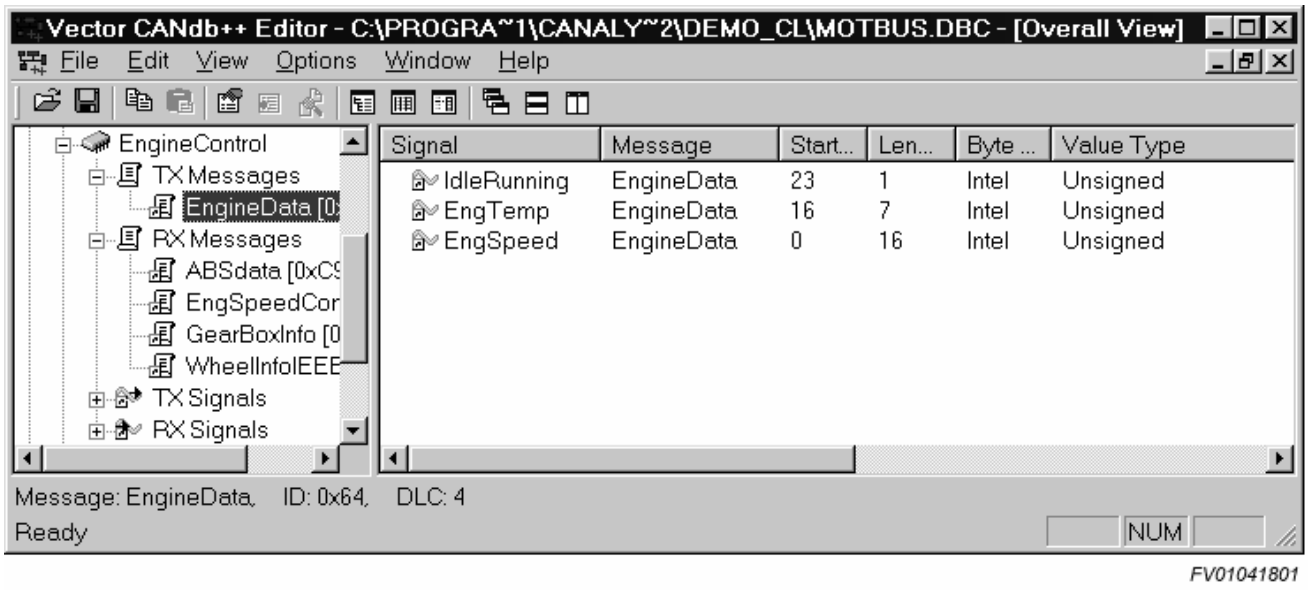

Figure 77 – CAN Messages and Signals in CANdb++

The main advantage of using a database is that the communication data can be accessed using its descriptive name. This means that if the **EngSpeed** signal is to be accessed, the name **EngSpeed** can be used to read or write the signal value. The application engineer does not have to worry about the location of this signal in the CAN message because this is already defined in the CANdb database. If the same thing must be done without a CANdb database, the application engineer must always memorize where signals are located in a CAN message, and the data bytes in the CAN message should be read or written directly.

Because every CANoe and CANalyzer configuration will have an assigned CANdb database, the COM server provides functionality to access communication data through the signals defined in the CANdb database. To access a **Signal** object in Visual Basic, it must first be assigned to a variable. This variable can be declared as the following:

```
Dim gEngSpeedSignal As CANalyzer.Signal
```

The method **GetSignal** of object **Bus** can be used to assign the signal of a CAN message to the previously declared variable **gEngSpeedSignal**. This signal should be defined in the CANdb database of the CANalyzer/CANoe configuration being used. The **Bus** object is part of the **Application** object (refer to the COM server object hierarchy in the online help). This means that the already initialized **gCanApp** variable can be used to access the Bus object as shown below:

```
Set gEngSpeedSignal = gCanApp.Bus.GetSignal(1, "EngineData", "EngSpeed")
```

The method **GetSignal** accepts the following three parameters:

- The channel on which the signal is sent.
- The name of the message to which the signal belongs.
- The actual name of the signal.

After the signal has been assigned to the variable **gEngSpeedSignal**, the following variable can be used to access the signal using the **Value** property of the **Signal** object:

```
lblEngSpeedDisplayControl.Caption = gEngSpeedSignal.Value
```

While using the COM server, it is not possible to directly change the value of a signal. We will have an example later to explain how to assign a new value to the signal of a CAN message.

25.7 Accessing Environment Variables (CANoe only)

Environment variables, which are only available for CANoe, describe the behavior of network nodes with regard to external inputs and outputs. These variables are similar to the signals of a CAN message. Both environment variables and CAN message signals are used for the input and output of a node. The difference between the two is that CAN message signals are used for inputs and outputs for the CAN network communication, whereas environment variables are used for the inputs and outputs of a node's external sources, such as switch positions, sensor signals, and actuator signals. This section explains how to access environment variables using CANoe's COM server functionality.

Environment variables are defined and stored in a CANdb database. Figure 78 displays a screenshot of an example CANdb database, that has been opened using the CANdb++ Database Editor.

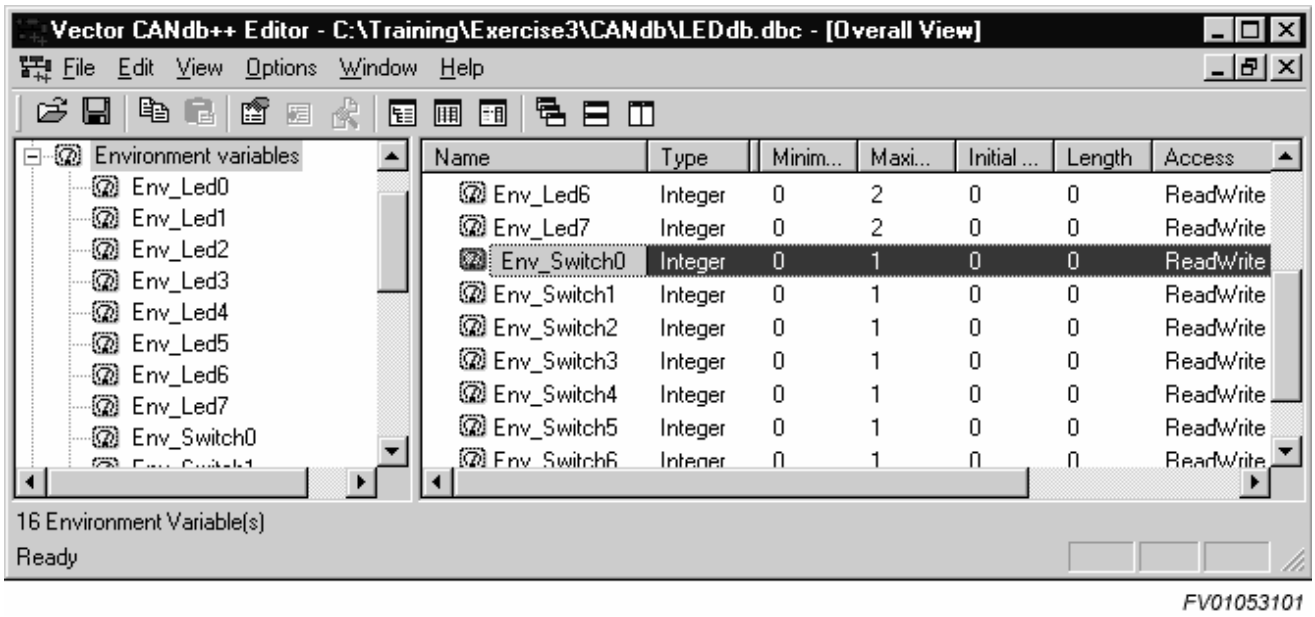

Figure 78 – Environment Variables in CANdb++

Because environment variables represent the external inputs and outputs of a node, they can be used for interaction with a panel or graphical user interface (GUI). In CANoe, it is possible to create your own GUI using the **Panel Editor**. Each object on the panel (button, textbox, and so on) in the **Panel Editor** can be linked to an environment variable that is defined in the CANdb database. This results in the user being able to change the value of environment variables using the panel. Every time an environment variable is changed, an event is triggered. CAPL makes it possible to react to these events to describe the behavior of a network node as to the external input/output that the environment variable represents.

Figure 79 shows an example of how environment variables are used in combination with a panel. In this example, switch number 0 on the control panel is linked to environment variable **Env_Switch0**. If the user activates the switch by clicking on it, the value of environment variable **Env_Switch0** changes from 0 to 1. This change triggers an event for which a handler is written in CAPL code.

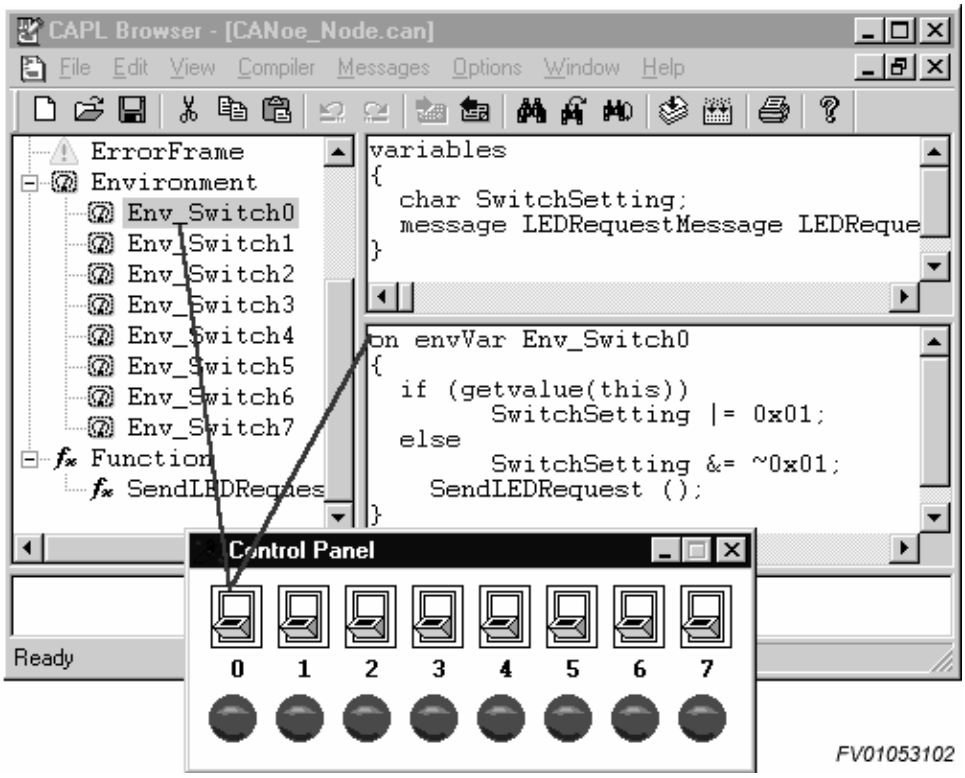

Figure 79 – Using Environment Variables with a GUI

The COM server makes it possible to read and write the values of environment variables. This means that Microsoft Visual Basic can be used to create a GUI and to implement the behavior of a node's external inputs and outputs to replace a CANoe panel. Using the COM server to access environment variables is the same as accessing CAN message signals, except that the value of environment variables can also be both read and written, whereas the value of CAN message signals can only be read.

To access an **EnvironmentVariable** object in Visual Basic, it must first be assigned to a variable. This variable can be declared as the following:

Dim gSwitch0EnvVar As CANoe.EnvironmentVariable

The function **GetVariable** of object **Environment** can be used to assign the environment variable to the previously declared variable **gSwitch0EnvVar**. This environment variable should be defined in the CANdb database of the CANoe configuration being used. The **Environment** object is part of the **Application** object. This means that the already initialized **gCanApp** variable can be used to access the **Environment** object as seen in the example below:

Set gSwitch0EnvVar = gCanApp.Environment.GetVariable("Env_Switch0")

The function **GetVariable** accepts one parameter – the name of the environment variable. This should be the exact same name of the environment variable as it is defined in the CANdb database. After the environment variable has been assigned to variable **gSwitch0EnvVar**, this variable can be used to access it using the **Value** property of the **EnvironmentVariable** object as seen below:

gSwitch0EnvVar.Value = 1

25.8 Reacting to CANoe/CANalyzer events

The COM server makes it possible to react to events triggered in CANalyzer/CANoe. An example of an event is the **OnStart** event of the **Measurement** object. This event is triggered every time a new measurement is started in CANalyzer/CANoe. Refer to the COM server object hierarchy in the CANalyzer/CANoe online help system to learn what other events are implemented.

This feature must be enabled for the COM server object to react to COM server object events in Visual Basic. In the beginning of this chapter, the global variable for the **Application** object was declared as the following:

```
Dim gCanApp As CANalyzer.Application
```

To enable the events for this specific object, the Visual Basic keyword **WithEvents** must be used in the declaration of the variable as shown below:

```
Dim WithEvents gCanApp As CANalyzer.Application
```

To react to one of the COM server objects events, a function in Visual Basic must be created and the name of this function is predefined. This is necessary because the event occurs externally from Visual Basic. The name of the function must be: **<object variable name>_<event name>**. For example, to react to the **OnQuit** event of the **Application** object (using the declaration in this section), the function would be as follows:

```
Private Sub gCanApp_OnQuit()
    'create application-specific OnQuit event handler here
End Sub
```

This Visual Basic function will be called every time CANalyzer shuts down. The procedure for reacting to an event is the same for all COM server objects.

25.9 Calling CAPL Functions

CAPL allows the application engineer to create applications for CANalyzer/CANoe. These programs help speed up the process of developing network nodes. CAPL can be used to simulate the behavior of CAN nodes, to analyze data traffic, and to create a gateway so that data can be exchanged between different CAN buses.

CAPL can also be used to implement application-specific functions to perform a certain task. The COM server functionality makes it possible to call these functions, which gives the application engineer total control over a measurement through the COM server. This section explains how the COM server can be used to call CAPL functions from a Microsoft Visual Basic application. CAPL programs are written using the CAPL browser, which is part of CANalyzer/CANoe. Figure 80 is a screenshot of the CAPL browser that shows the location of the CAPL functions.

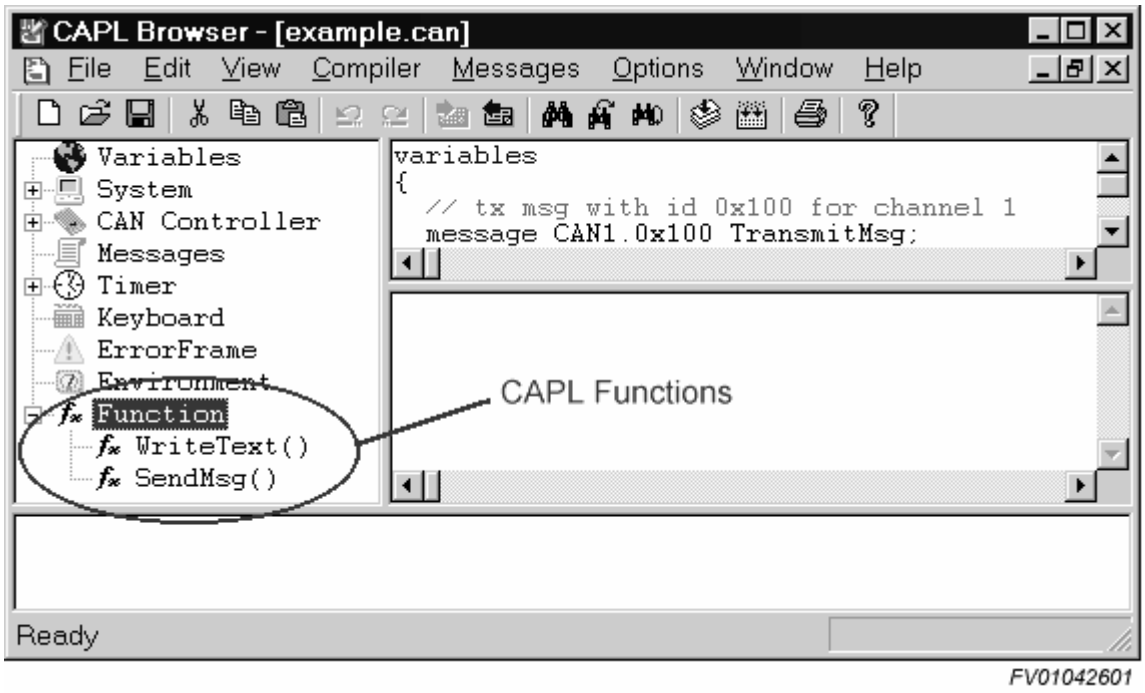

Figure 80 – Location of Functions in the CAPL Browser

A global variable is used to utilize the COM server's functionality to call CAPL functions. This variable should be declared to access the **CAPLFunction** object. Before the CAPL function can be called, it must be assigned to the variable. A CAPL function is assigned to a variable using the **GetFunction** method of the **CAPL** object. The parameter for the **GetFunction** method is the exact same name of the CAPL function.

Note: The assignment of a CAPL function to a variable can only be made in the **OnInit** event handler of the **Measurement** object. This means that the variable used to access the **Measurement** object must be declared using the **WithEvents** keyword in Visual Basic.

Once the variable has been declared and initialized, it can be used to call the CAPL function to which it refers. To call the CAPL function, the **Call** method of the **CAPLFunction** object is used as shown in the example below:

```
'CANalyzer objects
Dim WithEvents gCanApp As CANalyzer.Application
Dim WithEvents gCanMeasurement As CANalyzer.Measurement
Dim gWriteTextFunction As CANalyzer.CAPLFunction

'Initialization of CANalyzer objects
Set gCanApp = New Application
Set gCanMeasurement = gCanApp.Measurement

'Measurement OnInit event handler
Private Sub gCanMeasurement_OnInit()
    'assign CAPL function
    Set gWriteTextFunction = gCanApp.CAPL.GetFunction("WriteText")
End Sub

'Call WriteText() CAPL function
gWriteTextFunction.Call
```

It is also possible to pass input parameters to a CAPL function and read the return value of the CAPL function. The number of input parameters is limited to 10, and it is not possible to pass an array as a CAPL function parameter. The number of input parameters passed from Microsoft Visual Basic must always match the number of input parameters of the CAPL function. It is practical to use input parameters of type **Long** in CAPL. In this case, it is possible to pass on parameters from Microsoft Visual Basic that are from types **Byte** (1 byte), **Integer** (2 bytes), and **Long** (4 bytes), without having to worry about the types matching up between Microsoft Visual Basic and CAPL, as seen in the example below and illustrated in Figure 81:

```
'CANalyzer objects
Dim WithEvents gCanApp As CANalyzer.Application
Dim WithEvents gCanMeasurement As CANalyzer.Measurement
Dim gMultiplyFunction As CANalyzer.CAPLFunction
Dim gMultiplyResult As Integer

'Initialization of CANalyzer objects
Set gCanApp = New Application
Set gCanMeasurement = gCanApp.Measurement

'Measurement OnInit event handler
Private Sub gCanMeasurement_OnInit()
    'assign CAPL function
    Set gMultiplyFunction = gCanApp.CAPL.GetFunction("Multiply")
End Sub

'multiply two values and get the result in the CAPL function's return value
gMultiplyResult = gMultiplyFunction.Call (4, 5)
```

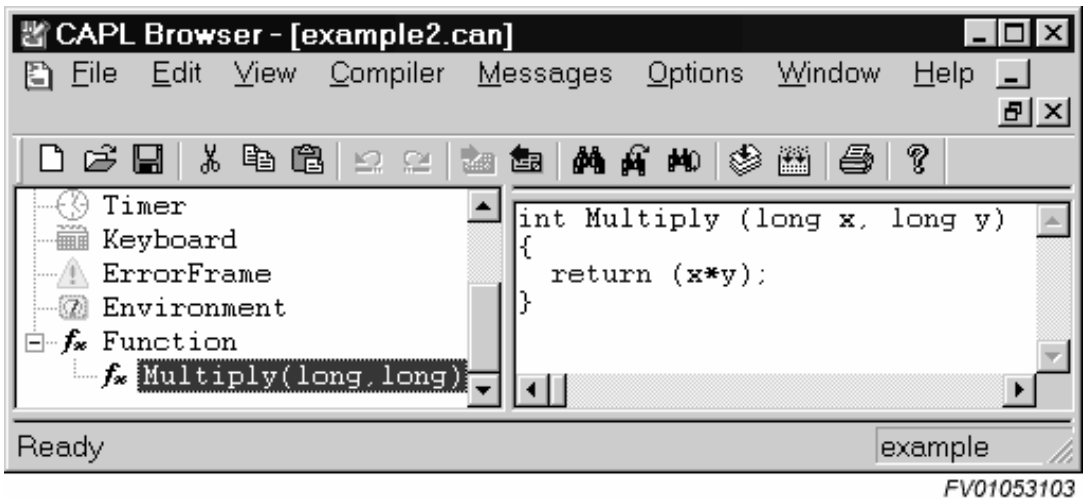

Figure 81 – CAPL User-Defined Function Multiply()

Note: It is important to remember that the return value of a CAPL function must always be of type integer. When using the COM server in combination with CANalyzer, it is only possible to use return values of CAPL functions if the P Block representing your CAPL program is configured in CANalyzer's Evaluation Branch. See Figure 82 below for an explanation of the different branches in CANalyzer's measurement setup (he Real-Time Branch and the Evaluation Branch are sometimes called Transmit Branch and Analysis Branch, respectively).

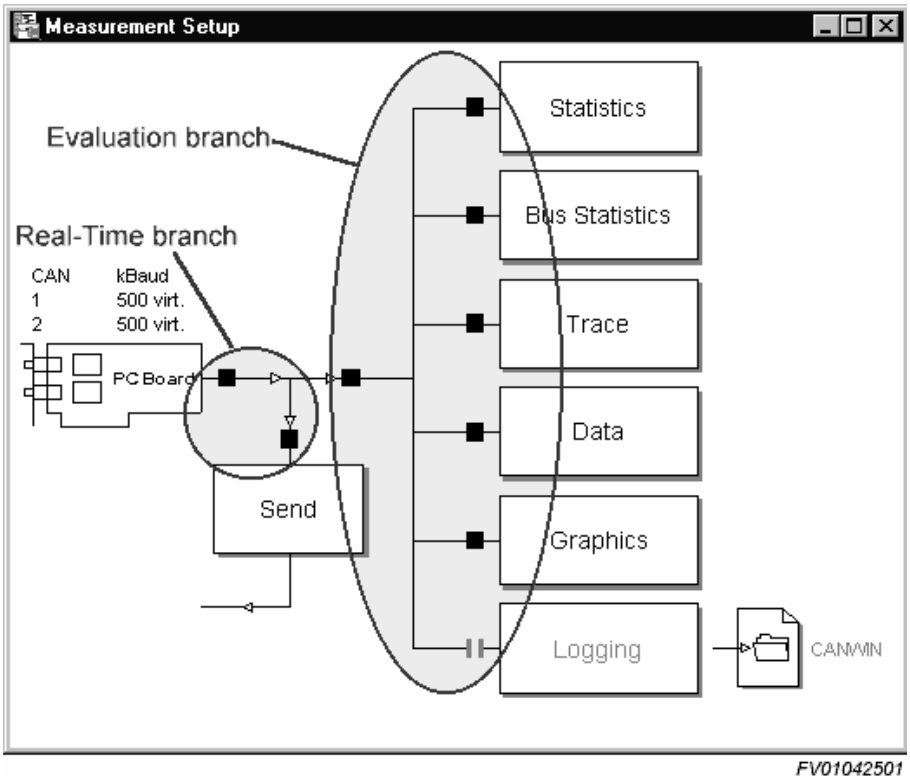

Figure 82 – CANalyzer's Measurement Branches

25.10 Transmitting CAN Messages

The COM server does not provide functionality to initiate the transmission of CAN messages. It is, however, still possible to initiate the transmission of CAN messages using the COM server. The application engineer can implement a function in his/her CAPL program that contains a call to the CAPL **output()** function.

If CANalyzer or CANoe Version 3.2 or older is used, a few restrictions apply:

- If a function implemented in CAPL contains a call to the **output()** function, it is not possible to call this CAPL function using the COM server if the CAPL program is located in CANalyzer's Transmit Branch. The obvious solution would be to locate the CAPL program in CANalyzer's Analysis Branch, but this doesn't solve the problem.
- When a CAN message is transmitted by a CAPL program that is located in CANalyzer's Analysis Branch, the CAN message will show up in CANalyzer's **Trace** window, but the CAN message does not actually go onto the CAN bus. This means that a CAPL program used for transmitting CAN messages must be located in CANalyzer's Transmit Branch.
- If the CAPL function called by the COM server can't contain a call to the **output()** function, then the **output()** function should be called at another point in the CAPL program. The CAPL function that is called by the COM server should then set a request to transmit a CAN message instead of transmitting it directly using a call to the **output()** function (for example, setting an environment variable or a timer). The CAPL program should also contain a routine that checks to see if this request is set or not. Whenever the request is set, this routine can make a call to the **output()** function to actually transmit the CAN message and finally reset the request.

The previously described way to transmit CAN messages using the COM server in CANalyzer is an indirect method and contains more overhead for the CAPL program. For this reason, an example is added that explains one possible way of transmitting CAN messages using the COM server and CANalyzer. The goal of this example is that the user can click a button on the Microsoft Visual Basic application to transmit a CAN message in CANalyzer. To realize this, the COM server is used to call the user-defined CAPL function **SendMsg()** whenever the button is clicked. All that this CAPL function does is start the **TxTimer_1ms** 1-millisecond timer. The timer expiration event handler in CAPL takes care of actually transmitting the CAN message. An extra feature of this example is that a value is passed on to the CAPL function that is called by the COM server. In CAPL, this value is read and assigned to a signal of the message that will be transmitted by the timer's expiration event handler (see Figure 83).

25.11 COM Server Example in Microsoft Visual Basic

Figure 84 is a screenshot of an example application using the COM server in Microsoft Visual Basic. It uses the "Motbus.cfg" CANalyzer demo configuration. You can also use the "Automot.cfg" CANoe demo configuration . The changes you need to make are the global variable declarations and have the configuration load remotely. These demo configurations are part of the CANalyzer/CANoe installation.

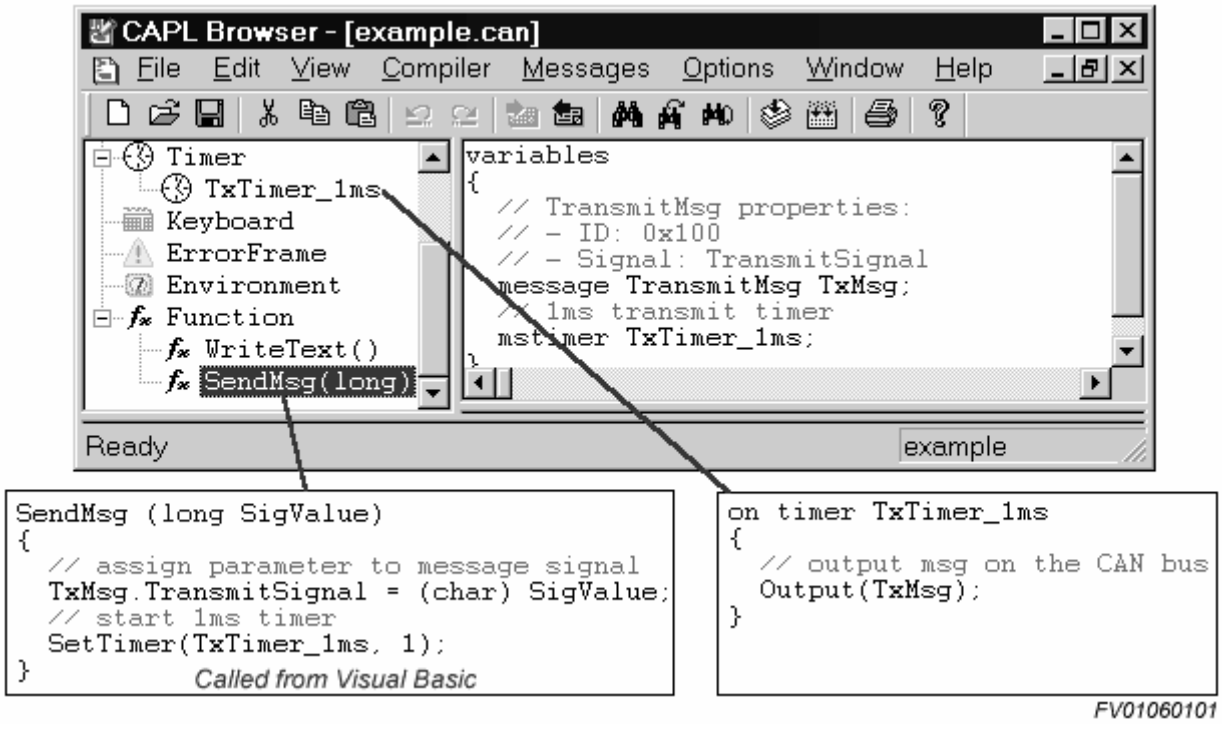

Figure 83 – Transmitting CAN Messages with CANalyzer

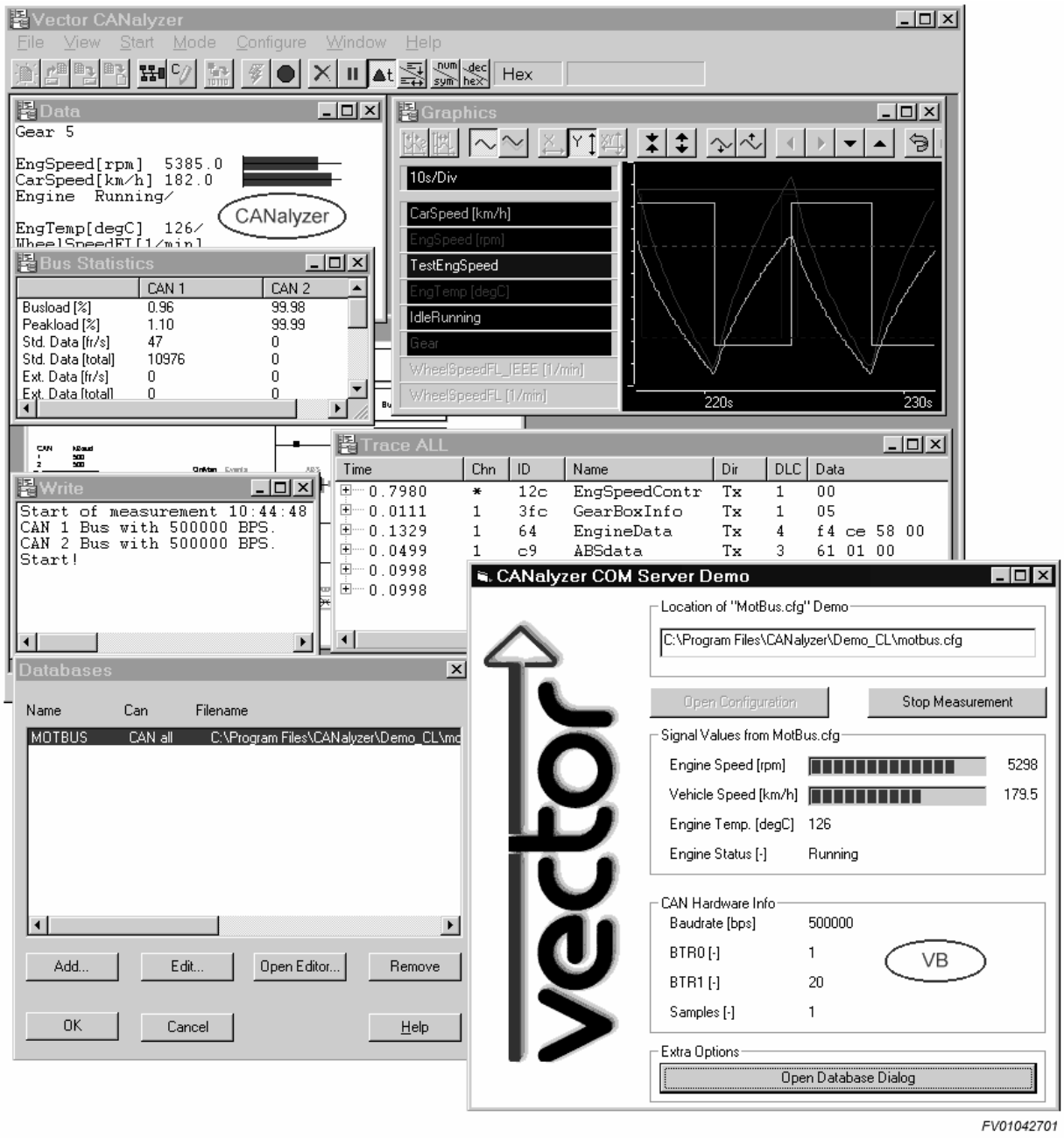

Figure 84 – Screenshot of Example Application

25.11.1 VB Source Code

The VB form interface in Figure 84 is left for you to build. All the example code used in this chapter is included in the source code below.

```
'*********************************************************************
' GLOBALS:
'*********************************************************************
' objects:
Dim WithEvents gCanApp As CANalyzer.Application
Dim WithEvents gCanMeasurement As CANalyzer.Measurement
' signals:
Dim gEngSpeedSignal As CANalyzer.Signal
Dim gCarSpeedSignal As CANalyzer.Signal
```

```
Dim gEngTempSignal As CANalyzer.Signal
Dim gEngStatusSignal As CANalyzer.Signal
' CAPL functions
Dim gSendGearFunction As CANalyzer.CAPLFunction
' CAN Channel 1
Dim gCANChannel1 As CANalyzer.CANController
' indicator flags
Dim gConnected As Boolean
Dim gMotBus As Boolean

'*********************************************************************
' NAME: CanConnect
' DESCRIPTION: Initializes CANalyzer object variables and sets
' some indication flags used in the application
'*********************************************************************
Private Sub CanConnect()
    ' init new CANalyzer application
    Set gCanApp = New Application
    ' init measurement object
    Set gCanMeasurement = gCanApp.Measurement
    ' init CAN channel variable of bustype CAN (=1) and CAN channel 1
    Set gCANChannel1 = gCanApp.configuration.GeneralSetup.ControllerSetup(1, 1)
    ' indicate that the connection has been made
    gConnected = True
    ' check to see if MotBus.cfg configuration is loaded
    If (gCanApp.configuration.Name = "motbus") Then
        gMotBus = True
    Else
        gMotBus = False
    End If
End Sub

'*********************************************************************
' NAME: UpdateMotBusSignals
' DESCRIPTION: Reads the signal values from the CAN messages
' and displays these values on the form
'*********************************************************************
Private Sub UpdateMotBusSignals()
    If (gMotBus = True) Then
        prgEngineSpeed.Value = gEngSpeedSignal.Value
        lblEngineSpeedVal.Caption = gEngSpeedSignal.Value
        prgVehicleSpeed.Value = gCarSpeedSignal.Value
        lblVehicleSpeedVal.Caption = gCarSpeedSignal.Value
        lblEngineTemperatureVal.Caption = gEngTempSignal.Value
        If (gEngStatusSignal.Value) Then
            lblEngineStatusVal.Caption = "Idle"
        Else
            lblEngineStatusVal.Caption = "Running"
        End If
    End If
End Sub

'*********************************************************************
' NAME: GetCANChannelParameters
' DESCRIPTION: Reads the CAN channel parameters using the COM
' server and displays them using labels.
'*********************************************************************
Private Sub GetCANChannelParameters()
    lblBaudrateValue = gCANChannel1.Baudrate
    lblBtr0Value = gCANChannel1.BTR0
```

```
    lblBtr1Value = gCANChannel1.BTR1
    lblSamplesValue = gCANChannel1.Samples
End Sub

'*********************************************************************
' NAME: ResetDisplayControls
' DESCRIPTION: Resets all the controls related to the MotBus
' signals
'*********************************************************************
Private Sub ResetDisplayControls()
    prgEngineSpeed.Value = 0
    lblEngineSpeedVal.Caption = ""
    prgVehicleSpeed.Value = 0
    lblVehicleSpeedVal.Caption = ""
    lblEngineTemperatureVal.Caption = ""
    lblEngineStatusVal.Caption = ""
    lblBaudrateValue = ""
    lblBtr0Value = ""
    lblBtr1Value = ""
    lblSamplesValue = ""
End Sub

'*********************************************************************
' NAME: btnDbDialog_Click
' DESCRIPTION: Opens CANalyzer database dialog
'*********************************************************************
Private Sub btnDbDialog_Click()
    gCanApp.UI.OpenDbDialog
End Sub

'*********************************************************************
' NAME: btnOpenConfig_Click
' DESCRIPTION: Opens the configuration as specified in the
' text box and compiles the CAPL code. It also stops all running
‘ measurements
'*********************************************************************
Private Sub btnOpenConfig_Click()
    If (gConnected = False) Then
        ' make the connection
        CanConnect
    End If
    If (gCanApp.Measurement.Running) Then
        ' stop the CANalyzer measurement
        gCanApp.Measurement.Stop
    End If
    ' load an existing CANalyzer configuration
    gCanApp.Open (txtConfigLocation.Text)
    ' compile any CAPL code of the CANalyzer configuration
    gCanApp.CAPL.Compile
    ' check to see if MotBus.cfg configuration is loaded
    If (gCanApp.configuration.Name = "motbus") Then
        gMotBus = True
    Else
        gMotBus = False
    End If
    ' enable database dialog button
    btnDbDialog.Enabled = True
End Sub

'*********************************************************************
```

```
' NAME: btnStartMeasurement_Click
' DESCRIPTION: Starts the measurement if no measurements are
' running; otherwise it stops the measurement
'*********************************************************************
Private Sub btnStartMeasurement_Click()
    If (btnStartMeasurement.Caption = "Start Measurement") Then
        If (gConnected = False) Then
            ' make the connection
            CanConnect
        End If
        ' start a CANalyzer measurement
        gCanApp.Measurement.Start
    Else
        ' stop a CANalyzer measurement
        gCanApp.Measurement.Stop
    End If
End Sub

'*********************************************************************
' NAME: gCanMeasurement_OnStart (event handler)
' DESCRIPTION: Disables the open configuration button and
' changes the caption of the measurement button
'*********************************************************************
Private Sub gCanMeasurement_OnStart()
    ' disable open configuration functionality
    btnOpenConfig.Enabled = False
    ' change the caption of the button
    btnStartMeasurement.Caption = "Stop Measurement"
    ' display CAN channel parameters
    GetCANChannelParameters
    ' enable the timer
    tmrSignals.Enabled = True
End Sub

'*********************************************************************
' NAME: gCanMeasurement_OnStop (event handler)
' DESCRIPTION: Enables the open configuration button and
' changes the caption of the measurement button
'*********************************************************************
Private Sub gCanMeasurement_OnStop()
    ' disable the timer
    tmrSignals.Enabled = False
    ' enable open configuration functionality
    btnOpenConfig.Enabled = True
    ' change the caption of the button
    btnStartMeasurement.Caption = "Start Measurement"
    ' update controls on the form
    ResetDisplayControls
End Sub

'*********************************************************************
' NAME: gCanApp_OnQuit (event handler)
' DESCRIPTION: Sets a flag to indicate that the connection
' with the COM server is no longer there
'*********************************************************************
Private Sub gCanApp_OnQuit()
    ' indicate that MotBus.cfg is not loaded
    gMotBus = False
    ' disable the timer
    tmrSignals.Enabled = False
```

```
    ' indicate that the connection is no longer there
    gConnected = False
    ' reset signals
    Set gEngSpeedSignal = Nothing
    Set gCarSpeedSignal = Nothing
    Set gEngTempSignal = Nothing
    Set gEngStatusSignal = Nothing
    ' reset CANalyzer object globals
    Set gCanApp = Nothing
    Set gCanMeasurement = Nothing
    Set gCANChannel1 = Nothing
End Sub

'*********************************************************************
' NAME: gCanMeasurement_OnInit (event handler)
' DESCRIPTION: Initializes the measurement signals of the
' MotBus.cfg configuration if this is loaded
'*********************************************************************
Private Sub gCanMeasurement_OnInit()
    If (gMotBus = True) Then
        ' assign signals
        Set gEngSpeedSignal = gCanApp.Bus.GetSignal(1, "EngineData", "EngSpeed")
        Set gCarSpeedSignal = gCanApp.Bus.GetSignal(1, "ABSdata", "CarSpeed")
        Set gEngTempSignal = gCanApp.Bus.GetSignal(1, "EngineData", "EngTemp")
        Set gEngStatusSignal = gCanApp.Bus.GetSignal(1, "EngineData", "IdleRunning")
    End If
    ' assign CAPL function
    Set gSendGearFunction = gCanApp.CAPL.GetFunction("sendGear")
    ' call CAPL function to set gear position to one
    gSendGearFunction.Call (1)
End Sub

'*********************************************************************
' NAME: tmrSignals_Timer (event handler)
' DESCRIPTION: On timer event that occurs every 100ms if a
' measurement is running. If the MotBus.cfg configuration is active,
' the signals on the form are updated
'*********************************************************************
Private Sub tmrSignals_Timer()
    If (gMotBus = True) Then
        UpdateMotBusSignals
    End If
```

26 Appendix A: Introduction to CAN Communications

This guide is divided into three major related topics. The first topic discusses many of the key aspects and elements of a distributed application. This “beginning at the systems level” overview helps to explain the need for a communication system like CAN. The second major topic moves to an overview of the CAN protocol with a sufficient amount of detail to help you master the “upper-level rules” of the protocol. The last topic discusses the common electrical wiring interface used for CAN communications - the high-speed, 2-wire differential, and CAN Physical Layer.

Understanding these three topics in combination will quickly help you learn how to get your first CAN application up and running quickly.

26.1 The Distributed Application

What are we trying to accomplish? The goal is not only to learn how to implement a communication system but to construct an application that operates in a distributed environment. Before we learn details about the CAN protocol, let us first begin at the system level. Learning about the system aspects of a distributed application will make the CAN protocol much easier to understand.

In general, there are two common types of electronic system or product architectures - centralized and distributed. Figure 85 shows a simple model of both architectures.

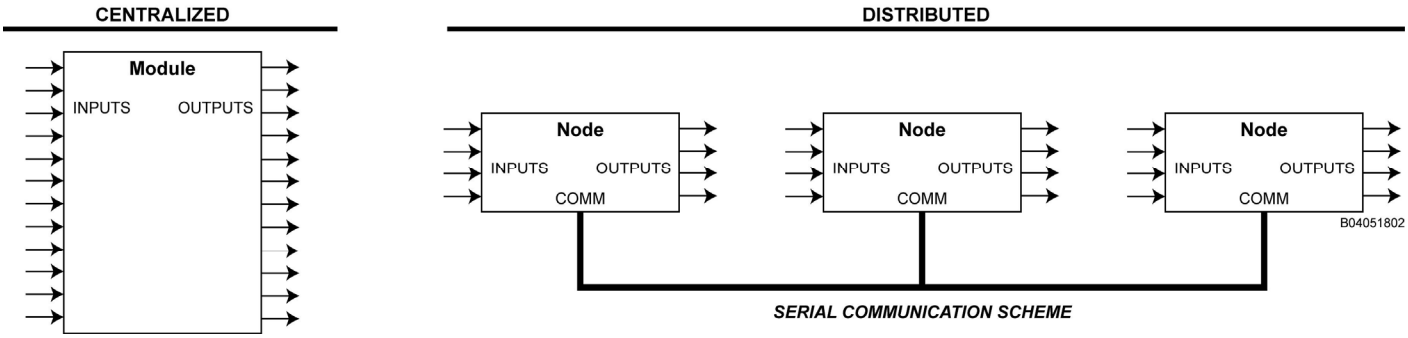

Figure 85 - Centralized and Distributed Product Architectures

While many systems have originated using a centralized architecture, many new electronic systems are distributed. The contemporary automobile is a good example of distributed product architectures. Initially consisting of one large electronic control unit (ECU) to control all car functions, vehicle feature content has been physically partitioned across a multitude of smaller electronic modules to reduce wiring cost and to help simplify the addition of options. Many other industries have made the transition to a distributed architecture for similar business reasons.

The distributed product or distributed embedded system is the container for the distributed application. The typical distributed application contains a large set of functions, and many of these functions are distributed. In the next section, we will examine the elements of the distributed function.

26.2 Distributed Functions

At the system level, the application is essentially a collection of functions. If the architectural decision is to distribute the system, many of the original functions may become distributed.

To understand the distributed function, let us first begin with the concept of a basic function, which can be easily modeled as three elements - an input process, a control process, and an output process - as shown in Figure 86. A basic function might be pressing a switch that activates a light.

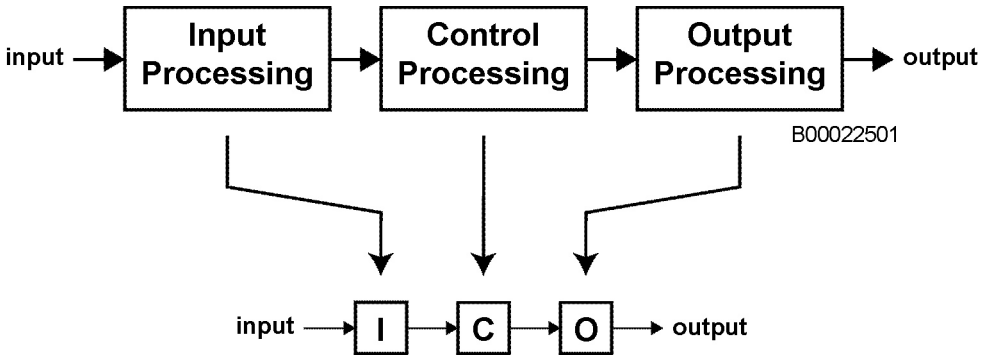

Figure 86 - The Basic Function

To distribute a function, one simply partitions the elements of a function across the membership of the distributed embedded system. This partitioning process requires the creation of a partition boundary as shown in Figure 87.

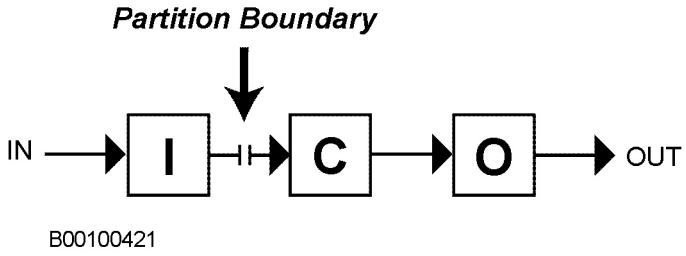

Figure 87 – The Partition Boundary

If we choose to partition the function using a partition boundary between the input and the control processes, a distributed function results (Figure 88). The act of distributing a function results in two new processes plus the need to transfer information across some communication channel. In comparison, the original function included only three elements. The distributed function increases the number of processes to five.

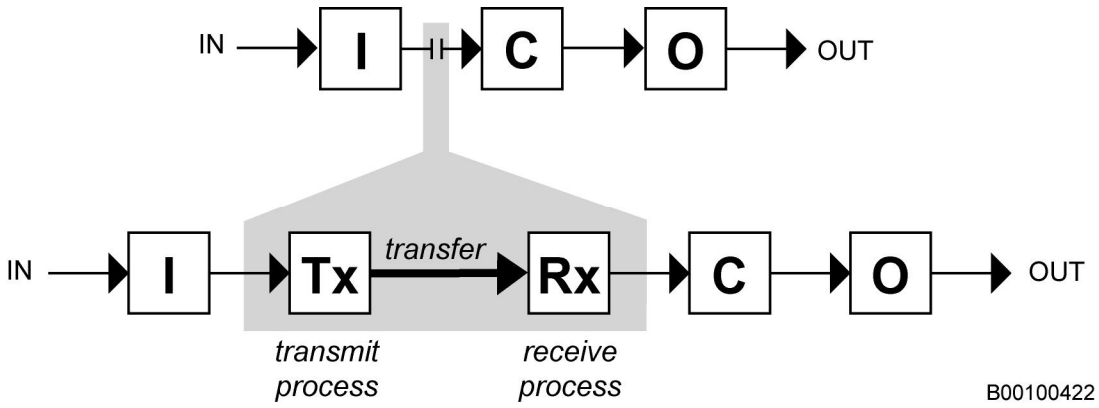

Figure 88 - The Distributed Function

Distributed functions result from the splitting of a function across the distributed system. From our previous example, we could let one module or node read the switch and have a different module or node turn on the light.

To assure proper operation, the distributed function requires a data or "signal" transfer between the modules. The format and size of this data depends on each function. While the movement of information, such as a switch press, requires only a single bit, an analog value like the position of a pot may require eight bits of data.

26.2.1 A Common Example of a Distributed Function

One common automotive distributed function with which almost everyone has interacted is the "door unlocking" function as shown in Figure 89.

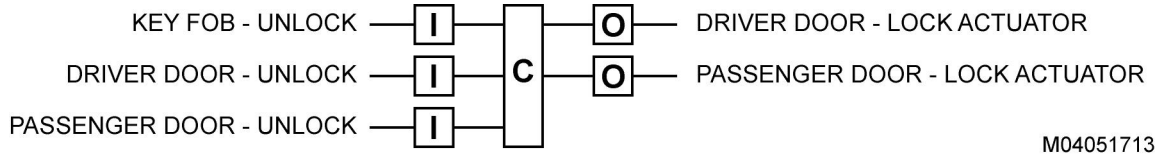

Figure 89 – Door Unlocking as a Distributed Function

In a typical 2-door vehicle, the unlocking function is accomplished using three inputs - one unlock switch on the driver's door, a similar second switch on the passenger door, and a single unlock button on the key FOB. Two outputs are used to activate the separate door unlock mechanisms. While the exact location of the control block is of little importance, many automobile system designers place the control process close to the output process.

As a result of this system level partitioning, the actual distributed function is commonly placed into three modules, similar to that shown in Figure 90. A communication bus is used to effectively glue the function together.

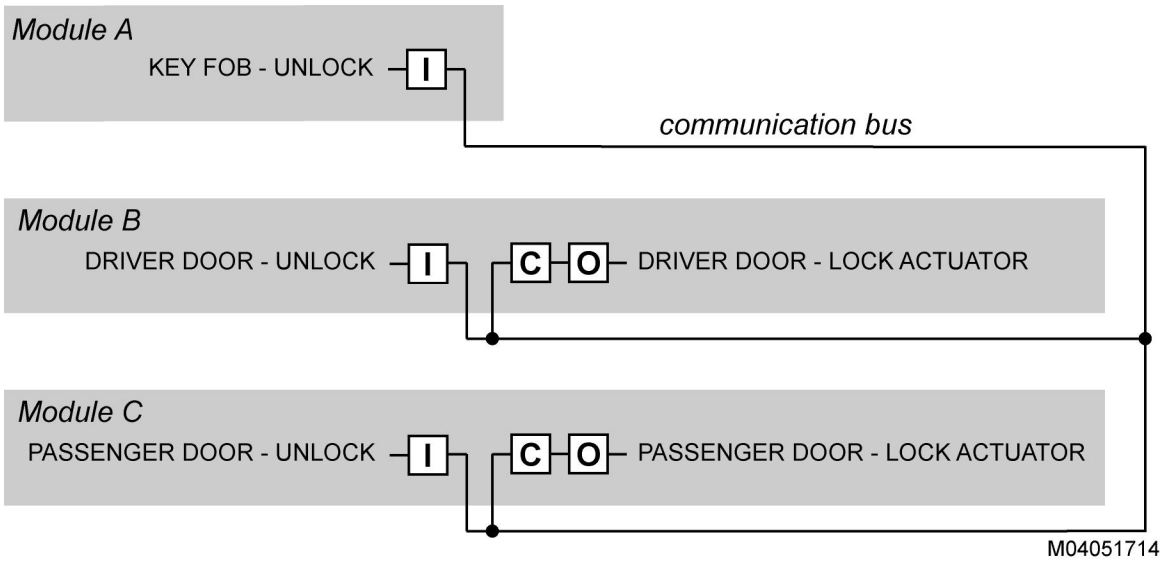

Figure 90 – Door Unlocking Function Distributed over Three Modules

26.3 Partitioning Requires Addressing

As you partition the distributed application into constituent parts, a way to point at the various distributed function elements becomes necessary.

For example, let us assume we were constructing a simple gaming system that could determine which of 32 individual switches was pressed first. Instead of using 32 wires, we have decided to partition the function immediately after the input process and to use a communication bus to reduce the wiring. The control process only needs to determine which contestant or switch is the winner and move this information to the output process to display the winning number.

To implement this system as a distributed application, we would need some way of indicating which of the 32 switches was pressed. One way of implementing this using the CAN protocol is to assign 32 separate message identifiers. Each input switch would be assigned a unique CAN identifier that specifically indicates which switch is being pressed.

If all input (switch reading) processes are identical, then the first input switch press to transfer its message wins. The control process only needs to look for the first incoming message and to determine the corresponding switch number.

Note - There are actually several ways of implementing this using CAN. You could use a set of contiguous message identifiers or use a set of random identifiers with a message vs. switch number lookup table or any set of identifiers with the switch number placed into the Data Field. However, based on the CAN protocol rules, it is important to understand that you cannot use the same message identifier for each individual (input switch) transmitter. Why? Two transmitters that simultaneously send the same message identifier will cause either one or both transmitters to go to the "Bus Off" condition.

Addressing schemes used for distributed applications may be physical, functional, or combinations of both. While a physical address is used to point directly at a specific physical module, the functional address points at a network object, which is essentially a designated system function or action. The functional address is independent of physical location.

For example, you could define “reset entire system” as a functional object and then create a single address to point at this function. From a standpoint of the number of total messages, this is much more efficient than defining a collection of “reset individual module” messages for each physical node. In the earlier example of the automotive door unlocking function, it is simpler to design a single functional address that points at “unlock” rather than to use 2 addresses; one to point at the driver door's unlocking mechanism and a second to point at the passenger door unlocking mechanism.

The development of a distributed embedded system’s address structure is a key design step that requires attention and documentation. While some developers use a spreadsheet or database to construct the network addresses, many use Vector’s CANdb tool.

26.4 Distributed Functions Require a Transfer Dialog

For each distributed function, a design decision must be made as to how and when the data is transferred across the partitioning boundary. The selected method and timing characteristics of the information transfer is called the transfer dialog. The selection of a particular transfer dialog is heavily dependent upon the requirements of the intended function.

While many variants are possible, the event, periodic, and conditionally periodic messages are three of the most common types of transfer dialog, as shown in Figure 91.

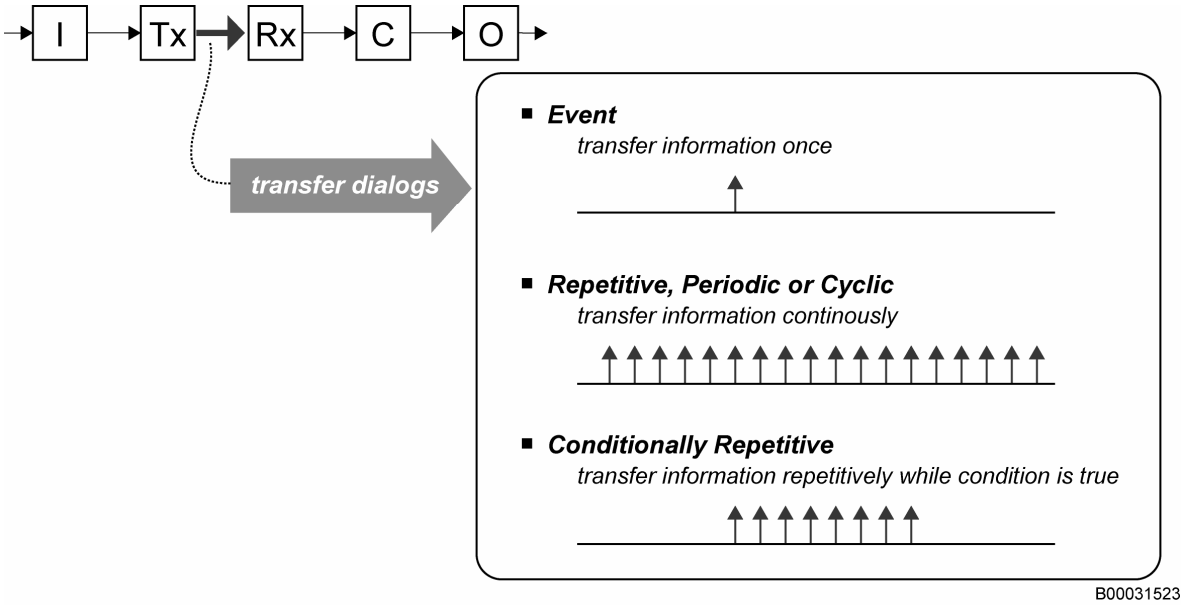

Figure 91 - Common Transfer Dialog Methods

The event message is sent once when an event occurs. For example, a switch press may result in a single message indicating this event. From a system perspective, the event message makes efficient use of the communications system and consumes a very low amount of network bandwidth.

The periodic message is continuously sent at a constant rate. The reoccurrence of the information can be used to provide a continuous connection across the distributed function. When compared to the event message, the periodic message requires a higher amount of network bandwidth.

The conditionally periodic message only transfers information continuously if a specific condition is true. For example, a system designer may choose that when a switch is pressed, then and only then is a periodic message transferred, then once the switch is released, the message transfer stops.

Other transfer dialogs are also possible. Imagine the press of a switch that activates an alarm system. Perhaps the designer may choose to send multiple event messages rather than a single event message to increase the reliability of the overall system.

Some designers choose to use another form of transfer dialog, "coupled messages". For example, if you are asked to activate an alarm and you send back a message that indicates the alarm is activated, this information confirms that the intended function has actually been achieved. While using coupled messages is another useful way of raising the reliability of the system, in general, it doubles the amount of messaging.

Using a dialog chart, Figure 92 shows this basic method of using “coupled” conversation between nodes A and B.

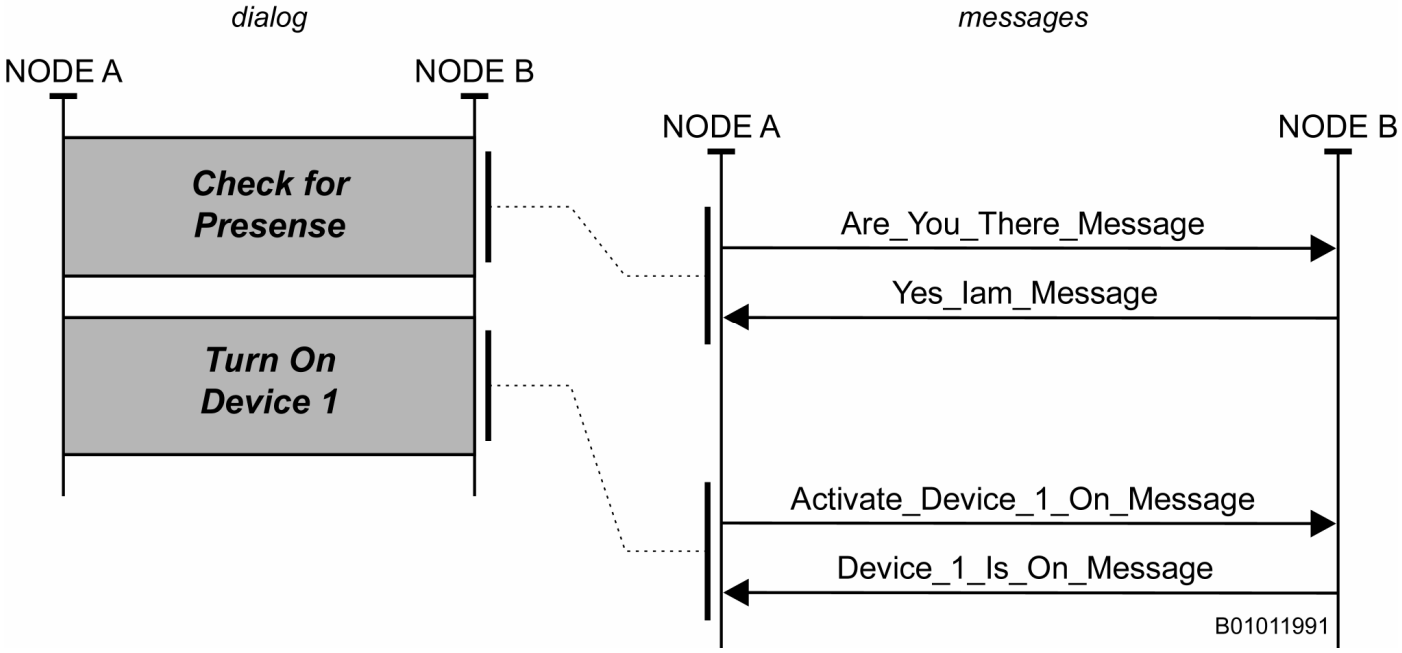

Figure 92 – Example of Message Coupling

26.5 A Distributed Function May Require File Transfer Capability

In some systems, the ability to transfer a large collection of information between network members may be necessary. It is common to use a transport protocol to accomplish such transfers.

Figure 93 shows a small file transfer with a simple 32-byte transfer that uses 4 data frames, each containing 8 data bytes. The complete transfer is accomplished by using 4 different segments.

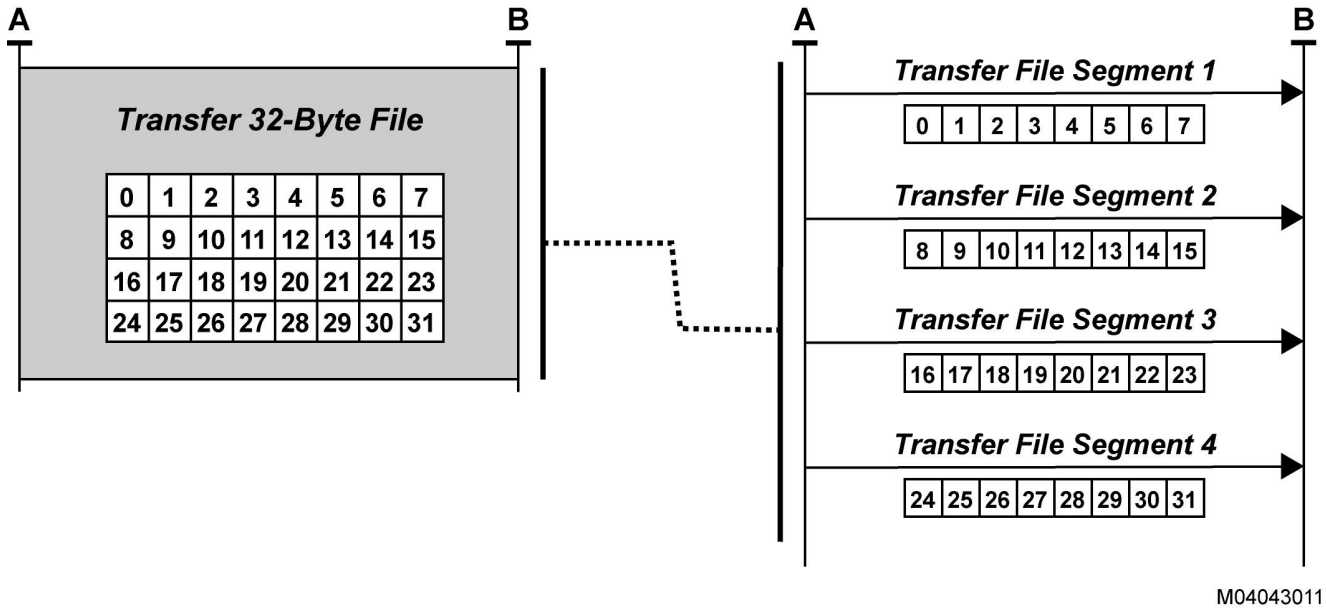

Figure 93 – Basic File Transfer

The process of transferring a file can also use a concept called indexing. Indexing requires that an additional informational component be added to each message to indicate the sequence position of the transfer. Figure 94 shows a 28-byte file transfer in which an index number has been attached to each transmitted segment.

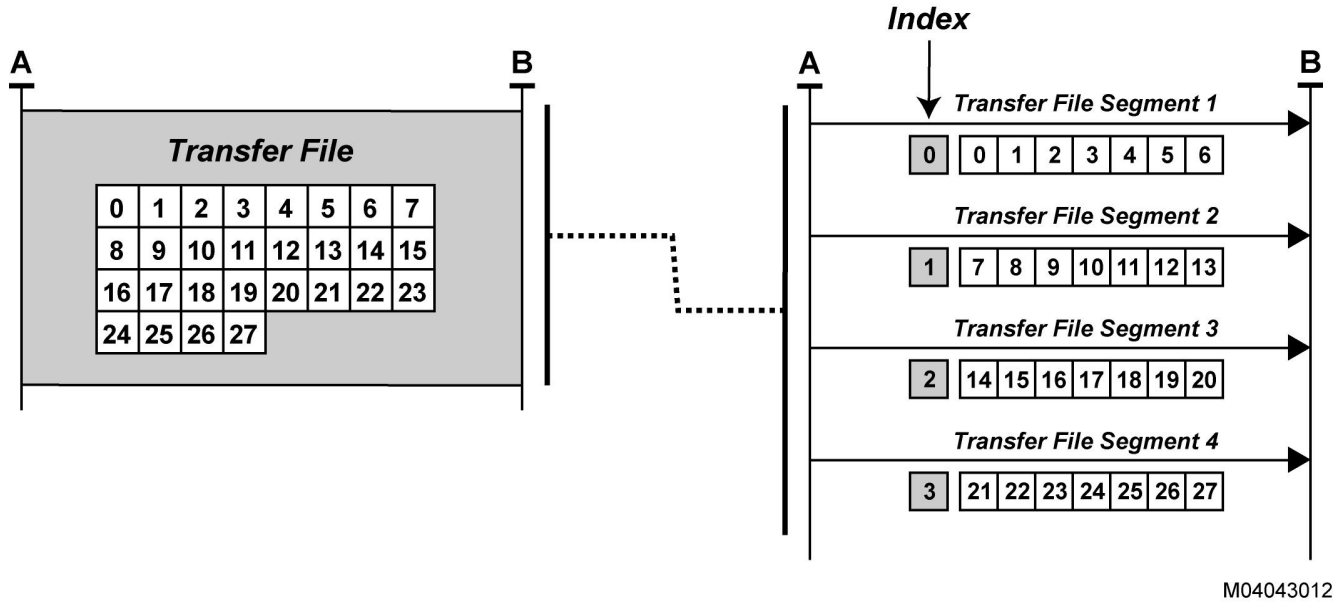

Figure 94 – Basic File Transfer with Indexing

26.6 The Distributed Application and the OSI Seven-Layer Model

As we have learned in the discussion surrounding the distributed application, several of the following key components are required:

- A communication method or set of methods to transfer data
- A means to distinguish between different distributed function elements
- A method or set of methods to accomplish the transfer dialog or conversation
- A set of transferable network data variables
- A potential need for file transfer

These same key components are the foundation of the OSI seven-layer model.

Independent of the type of network, the Open Systems Interconnect or OSI is a useful standard network architecture organized into a seven-layer reference model. Originated by the International Standards Organization or ISO, the governing document, ISO 7498, provides a comprehensive set of requirements for the exchange of information. The OSI 7-layer Reference Model has become a well-known standard, and as the standard has gained more recognition and popularity. The term “stack” has become a common one-word substitute for the OSI seven-layer reference model.

While typically diagrammed as a collection of seven interconnected blocks, as shown in Figure 95, it is important to understand that this structural set of requirements is implemented using software, hardware, and information technology.

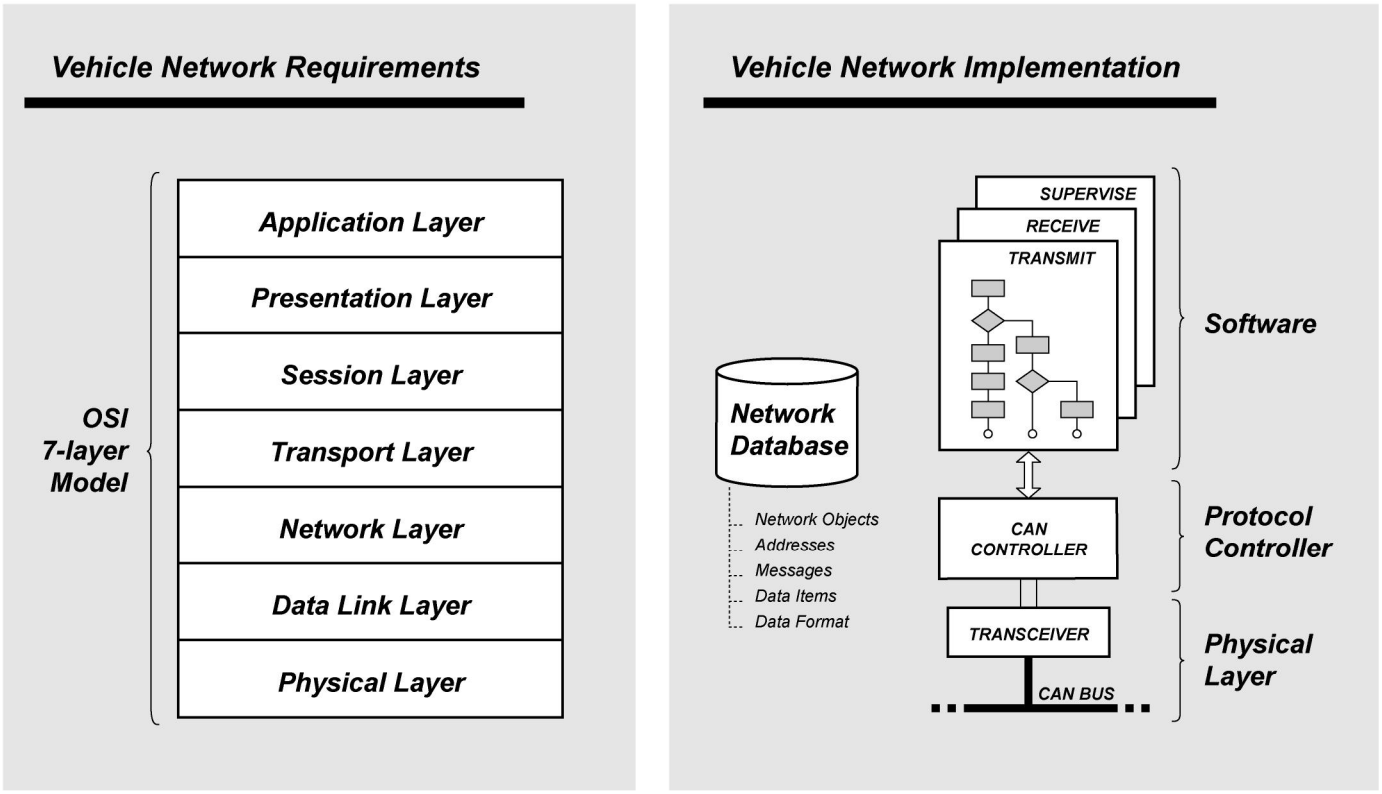

Figure 95 – The OSI Seven Layer Reference Model

26.6.1 Physical Layer

The Physical Layer consists of the network interconnection, the selected media (typically wire), connectors, the transceiver chip, and any other media interface-associated components (some designs use additional protection devices or common mode chokes). The Physical Layer provides a direct logical interface between the shared media or bus and the protocol controller (Data Link Layer) to handle the conversion of data logic levels into media states during transmission and the conversion of media states into data logic levels during reception. Additionally, the Physical Layer provides a media access technique that is used to solve a fundamental network sharing problem, that is, how does the system handles the event that more than one network member wants to use the media at exactly the same time.

26.6.2 Data Link Layer - Data Transfer Structure

The Data Link Layer establishes the requirements of the network’s data transfer structure. Typically implemented in silicon in a protocol controller, the Data Link Layer establishes the serial bit encoding method and handles all data transfer rules for proper transmission and reception. The Data Link Layer establishes rules that define the following:

- How to access the media (or bus) for transmitting
- How to recognize the media states during reception
- How to send information (from the bit level up to the message level)
- How to receive information (from the bit level up to the message level)

26.6.3 Network Layer - Address Structure

The Network Layer establishes the requirements of the network's address structure. Within any distributed system, we need the ability to point at network objects that come in two general types - physical objects and functional objects. It is the Network Layer that provides the ability to point at network objects.

26.6.4 Transport Layer - Message Transfer Structure

The Transport Layer establishes the message transfer structure and provides the ability to move larger amounts of information across the network. Because of several similarities, it is common to confuse the Transport Layer with the Data Link Layer. What is the real difference between the two? One way to understand is to use the following simple analogy. If the Data Link Layer handles the transfer of "words", and the Transport Layer handles the transfer of "sentences or paragraphs". When a distributed product architecture requires larger transfers beyond the Data Link Layer capability, the Transport Layer is used to provide this capability. This means that the Transport Layer may be an optional element of the architecture.

26.6.5 Session Layer - Conversation Structure

The Session Layer defines the network's conversation structure. When to talk or initiate a dialog, how often one should talk, and how one should respond are all conversational portions of a distributed embedded system. Typically, the designer must develop a wide range of conversation that may include the following:

- Network initialization
- Network synchronization activities (such as system reset, or perhaps sleep/wake-up)
- Dialog during normal operation
- Dialog during diagnostic operation
- Dialog during manufacturing activities (for example, flash programming)

26.6.6 Presentation Layer - Data Structure

The Presentation Layer establishes the entire data structure for the entire distributed embedded system. Each network data item, whether a bit, nibble, byte, multiple byte transfer, or file transfer, is a part of the data structure.

Commonly organized using a database, the Presentation Layer contains the following:

- The name of each network data variable (sometimes called a signal)
- To what message or messages the network data variable is assigned
- The size of the data
- The data content encoding or format
- The message name
- The home of the data (who the transmitter is)

26.6.7 Application Layer - Application-to-Network Interface Structure

The Application Layer defines the application-to-network interface and, when implemented, results in an API between the network software and the application software. The general application-to-network interface supports the following:

- Transmission activities
- Reception activities
- Network supervision activities

While this outside edge of the OSI model represents the major software interconnection between the application and the small area network, what actual services that may be provided by the Application Layer and how these services are implemented is a decision made by the systems architect.

26.7 Distributed Applications Require Data Structures

Distributed functions require data structures. The data must be named, sized accordingly, and organized into a message database. This system level data structure is equivalent to the OSI Presentation Layer.

The Presentation Layer establishes the entire data structure for the entire distributed embedded system. Each network data item, whether a bit, nibble, byte, multiple byte transfer, or file transfer, is a portion of the data structure.

Assigned by systems engineering activity and commonly organized using a database, the Presentation Layer contains the following:

- The name of each network data variable (sometimes called a signal)
- The message or messages to which the network data variable is assigned
- The size of the data
- The data content encoding or format
- The message name
- The home of the data (the transmitter)

It is common for many system architects to group the address structure (Network Layer), the data structure (Presentation Layer), and portions of the conversation structure (Session Layer) into the same database. Several automotive companies use Vector's CANdb++ tool to provide this organization.

26.8 Understanding the CAN Protocol

Now that we understand that the purpose of a small area network solution like CAN is to interconnect a collection of distributed functions into a working distributed application, let us move on to the CAN protocol.

CAN, or Controller Area Network, is a significant leap forward when compared to the use of serial communication methods based on the use of the UART or SPI. The CAN protocol is no ordinary substitute for the UART or SPI. This robust silicon-based implementation provides sufficient power to manage a major portion of a complete communication network solution. To build equivalent network functionality using the UART or SPI requires a significant amount of communication software to provide the same features.

26.8.1 Key Attributes

As a protocol, CAN has several features that make it an attractive choice for the user.

26.8.1.1 CAN Adoption – a Worldwide Standard

Developed at Bosch during the mid '80s for automotive applications, CAN supports distributed product and distributed embedded system architectures by providing a low-cost communication method to interconnect a network of electronic nodes, stations, or modules. The CAN protocol has emerged to be a worldwide standard both in the automotive industry and in several other industries including agriculture, heavy truck, bus, construction equipment, marine, industrial automation, and medical.

The Bosch CAN 2.0B specification has become the de facto standard that all new CAN Controller chip designs follow.

26.8.1.2 CAN Cost – the Lowest Business Cost Solution

For the automotive industry, the CAN protocol has essentially replaced all earlier proprietary protocols (for example, SAE J1850) which were individually developed at each car company. CAN has been adopted for many other industries and individual companies that use distributed product architectures. The basic reason for this widespread standardization is quite simple -- all things considered, CAN is currently a lower-cost solution than any other serial communication method.

The key business reasons to use CAN include the following:

- Low cost CAN hardware components - widely available
- Low cost CAN tools - in comparison to making your own tools
- Low cost CAN software components - in comparison to writing your own software
- Low cost CAN training - in comparison to making and maintaining your own training classes

26.8.1.3 CAN Transfer Rates - 1 Mbps and Beyond

Essentially, the transfer rate of the CAN protocol is not fixed to any one particular speed. Most of the common worldwide high-speed applications that are CAN-based are using 500K BPS. With few exceptions, 500K is the industry choice for automotive motion control area networks. At the other end of the automotive spectrum, low speed CAN bus applications also exist. General Motor's Single Wire CAN solution provides 33.3K BPS operation and handles most of the driver area functions including doors, windows, seats, mirrors, lighting, and other human-interface functions.

Because of the expectation at the beginning that most implementations would interconnect modules using a wiring harness, the Bosch CAN 2.0 specification inadvertently limits the transfer rate to 1 Mbps. In reality, the protocol speed is limited only by the network propagation delay. For example, if confined to the area of a single circuit board, a collection of interconnected high-speed microcomputers should be capable of CAN communication speeds well beyond 1 Mbps.

26.8.1.4 CAN Transfer Size - Unlimited

With a UART or SPI based protocol, you are only capable of single byte transfers. If you wish to transfer a message with multiple bytes, you must construct a software-based protocol that handles the process of message transmission, message reception, and message error handling. If you use the CAN Protocol instead, the CAN Controller will handle message transmission, message reception, and message error handling automatically. These message transfer processes are completely implemented in silicon.

A single CAN message allows a data transfer up to eight bytes in length. This means that for each message you have a choice of transferring 0, 1, 2, 3, 4, 5, 6, 7, or 8 bytes.

What if you must transfer more than eight bytes? What if you want to transfer files? Perhaps you might be interested in flash programming your module using the CAN bus. This is no problem. Just use some additional software to implement a special higher level protocol called a Transport Protocol, which essentially handles a file transfer.

26.8.1.5 CAN Data Consistency – Guaranteed for All Network Membership

Each and every transmitted CAN message includes a unique acknowledgment activity near the end of the message transfer. The CAN Protocol rules require that the entire network membership must participate in this acknowledgment process. Each and every CAN message receiver must signal either a positive or negative acknowledgment to the message transmitter. A simplified model of this important acknowledgment process at the system level is shown in Figure 96.

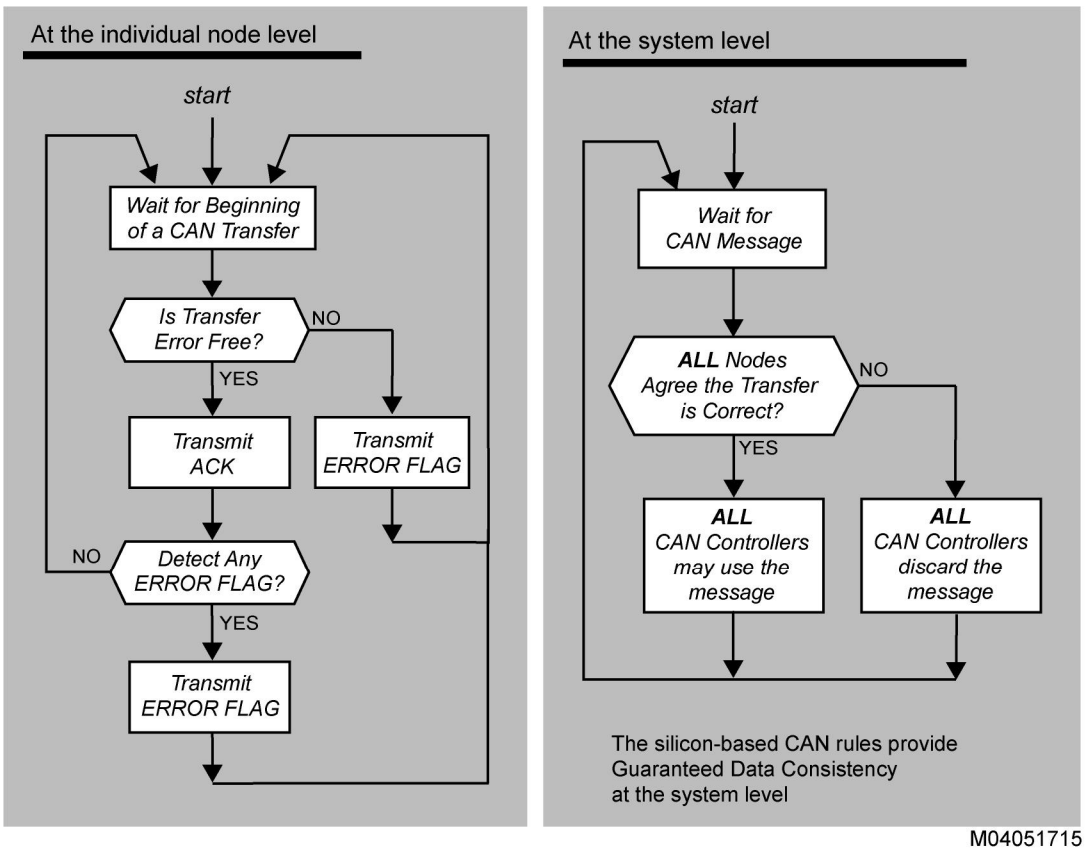

Figure 96 – All CAN Nodes Participate in the Acknowledgment Process

No software is required to handle this process because the acknowledgment process is entirely handled by the CAN Controller.

While this acknowledgment process is somewhat difficult to understand, it is one of the most important foundations of the CAN Protocol. The rule - that each and every CAN node (receiver) must agree (and confirm with an indication) that each and every message transfer is correct - is a virtual guarantee that the entire network membership is using exactly the same data. This rule is extremely valuable for real-time distributed control systems. This "guaranteed data consistency" feature is perhaps one of the most significant technical reasons for the success of the CAN protocol.

The acknowledgment process solves the problem of what happens if there is noise on the communication wires that corrupts the message transfer. Each node that detects a message transfer error will not agree that the message is correct and, subsequently, will signal a negative acknowledgment. This negative acknowledgment is transmitted as an "error flag" and is detectable by all network nodes - especially by those receiving nodes, that earlier may have indicated a positive acknowledgment. The negative acknowledgment (or "no agreement") is detectable by every CAN network member, and as a result all receiving nodes may not use the data. Additionally, the CAN Controller of the original message transmitter that detects the negative acknowledgment will automatically reattempt sending the message.

The CAN Protocol acknowledgment rule means:
For each data transfer, either all network members agree that the data is correct OR all network members agree that the data is not correct. It will not happen that some receivers consider the data to be correct while others consider the data to be incorrect.

Again, it is important to remember that these protocol rules are implemented in the CAN Controller silicon and no software interaction is required.

26.8.1.6 CAN Error Handling - Guaranteed Message Delivery

The CAN Protocol rules require that each CAN Controller include detailed transmission and reception error detection and handling processes. This automatic error handling includes the detection of bit errors and error frame detection or indication and further increases the CAN message transfer reliability.

Implementation of a similar error handling mechanism for a UART based or SPI based protocol must be accomplished in software.

In combination, the acknowledgment process, the CRC, and the error handling process - all accomplished in the CAN Controller silicon - virtually guarantee each and every message transfer.

26.8.1.7 CAN Transfer Reliability – Close to Perfect

With each CAN message transfer using a 15-bit cyclic redundancy check (CRC), CAN is considered a highly reliable protocol. Based on one academic study of transferring a periodic message at a continuous 100 millisecond rate, the CAN protocol undetected error rate was reported at a level of one single bad message every 1000 years!

In some cases, an application or applications may require even higher levels of reliability. In these cases, using the lessons learned from the military's early MIL STD 1553 protocol, the system designer can easily employ one of several techniques to increase the transfer reliability.

26.8.1.8 CAN Message Collisions – Never Occur

Two fundamental rules inside the CAN protocol guarantee that competing transmitters will not collide into each other during any message transfer. These rules are:

- The Physical Layer must support Dominant/Recessive Logic
- The interface must allow simultaneous comparison of the transmitted bit with the receive bit

Unlike Ethernet and other peer-to-peer UART-based protocols that must detect collisions and devote resources to resolve collisions, the CAN Protocol guarantees no message collisions. This means that 100 percent of the network bandwidth is available for message transfer.

We will learn more about dominant and recessive logic when we examine the CAN Physical Layer.

26.8.2 Understanding the Basic Elements of the CAN Message

For the student, learning about all details of the CAN protocol is not necessary. The following discussion targets only the key elements of the CAN protocol.

26.8.2.1 The Basic CAN Message Structure

The basic CAN message (also called the Data Frame) is shown in Figure 97.

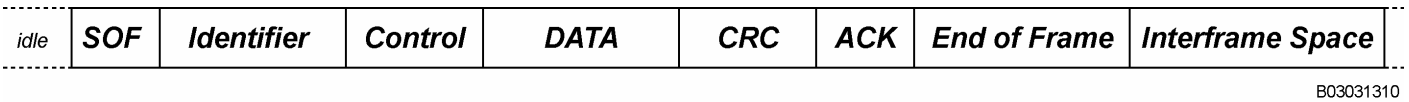

Figure 97 – The Basic CAN Message Structure

Note: - Because most distributed application developers do not use the CAN Remote Frame, the discussion of the basic CAN message structure will focus only on the Data Frame.

26.8.2.2 Idle

Each CAN message is preceded by a period of “idle” during which the bus is in the recessive state. The idle time interval must be at least 11 bit times. The detection (or time measurement) of the time duration of idle is an important component of CAN and essentially signifies a “ready for message” condition.

26.8.2.3 Start of Frame - SOF

Each CAN message begins with a Start of Frame indication. This is a single bit period during which the bus is in the dominant state.

26.8.2.4 Identifier

Each CAN message includes an identifier. Depending on your point of view - system, software, or hardware - the identifier essentially "names" the message or points at the data or acts as an address. The following two sizes of CAN identifiers are available:

- the 11 bit identifier - allowing the ability to point at about 2,000 network objects
- the 29 bit identifier - allowing the ability to point at millions of network objects

While most distributed application designers typically use only one type of identifier, some CAN Controllers do allow the ability to use both at the same time.

Figure 98 illustrates the basic organization of both the 11-bit and the 29-bit identifier.

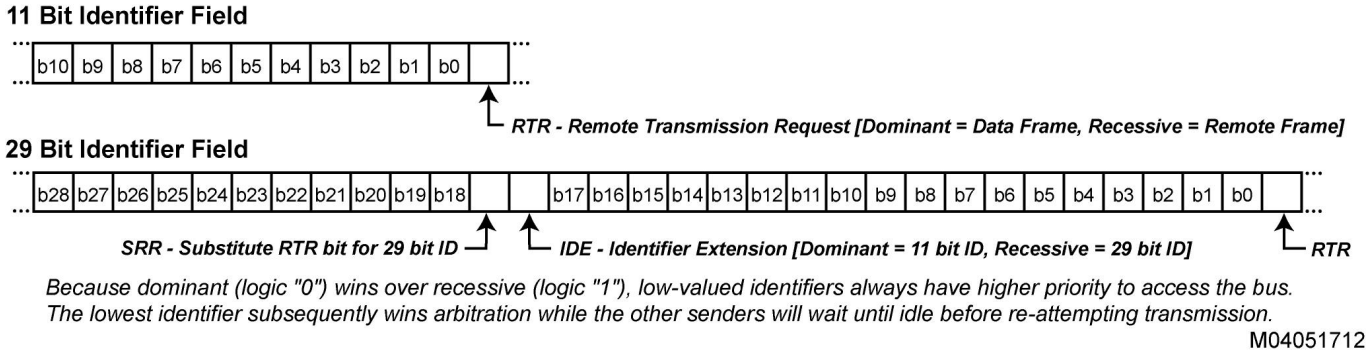

Figure 98 – The Organization of the CAN Identifier

26.8.2.5 Control

The Control Field contains the Data Length Code, usually called the DLC, which essentially tells the size of the data that is being transferred. Although 4 bits are used to indicate the Data Length Code, only data sizes between zero and eight are used. Figure 99 illustrates the organization of the Control Field. At the student level, the first 2 bits are relatively unimportant.

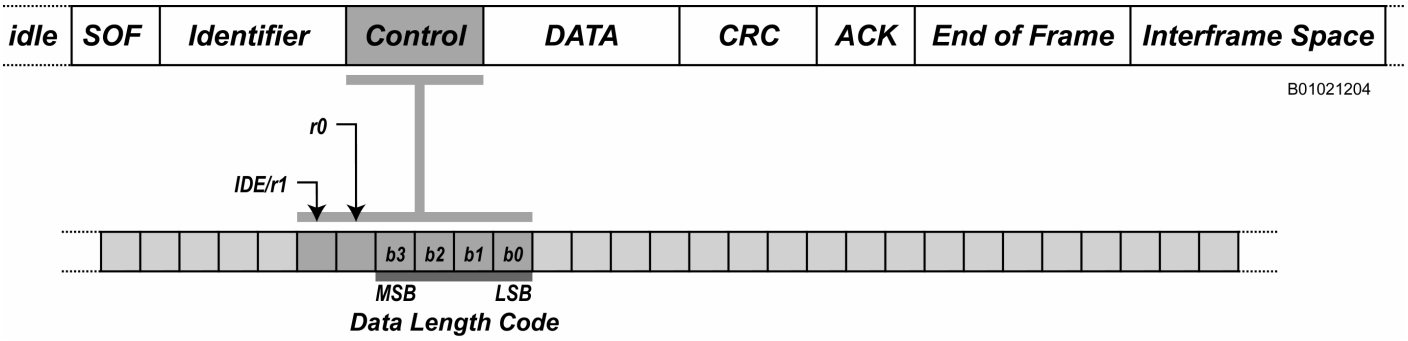

Figure 99 – The Control Field

26.8.2.6 Data

The Data Field contains up to 8 data bytes. Figure 100 illustrates the basic structure of the Data Field and includes the bit ordering.

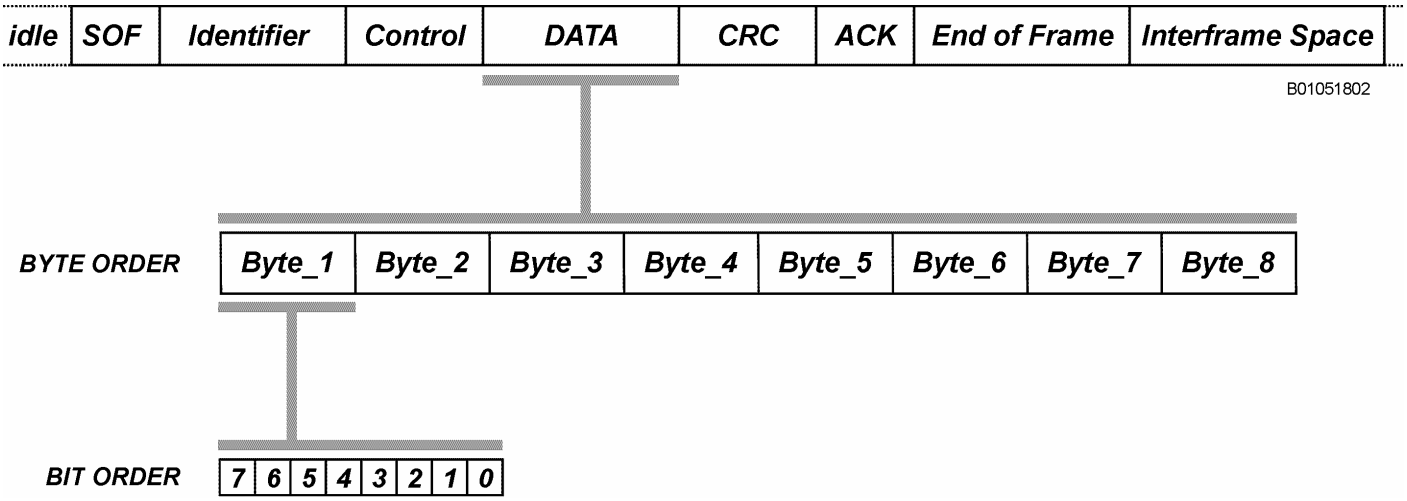

Figure 100 – The CAN Message Data Field Structure

26.8.2.7 Cyclic Redundancy Check

Each CAN message includes a Cyclic Redundancy Check (CRC). to increase the reliability of message transfers. A CRC is a type of checksum, a computed value near the end of a message frame which is based on the contents of a block of data starting from the start of frame to the last byte of the data field. The CRC computed by the transmitter is compared within each receiver to a locally calculated CRC. If the receiver's CRC matches the transmitted CRC, then the likelihood of a transfer error is very low.

While CAN uses a 15-bit CRC that is considered to be more than adequate for most applications, some safety-critical application developers increase the transfer reliability to an even higher level by embedding an additional CRC in the message data field.

26.8.2.8 Acknowledgment

The acknowledgment portion of the CAN message is used by all network members to indicate whether the CAN transfer is correct or incorrect. Entirely implemented in silicon, the acknowledgment process guarantees that all network members will agree as a group that the transfer is correct. If any network member detects the transfer to be incorrect, that network member will indicate this incorrect condition to the rest of the network.

The acknowledgment process is a sequential two-step process, organized in time as follows:

- First, all network members that agree the transfer is correct will indicate this condition during the acknowledgment bit by transmitting a dominant bit.
- Second, after this indication, any network members that disagree the transfer is correct will indicate this condition by transmitting an Error Flag.

Without using CAN terminology, a simpler way to explain this sequence is to understand that “yes” voters will vote first followed by the “no” voters. This means that anyone can veto the message. While this makes it sound as if a renegade node could ruin the network, additional protocol rules using transmit and receive error counters easily solves this problem.

26.8.2.9 End of Frame

The End of Frame is seven recessive bits, and serves as the following:

- A marker that indicates the end of the frame
- A time period of idle time between messages
- A time period to vote “no” for nodes that do not agree with the transfer

.

26.8.2.10 Intermission Space

The intermission space is 3 recessive bits, after which a subsequent CAN message may immediately begin, because the 11-bit idle time is already satisfied.

26.8.3 The CAN Error Frame

The CAN Error Frame is used to indicate an error condition to the entire network membership. The CAN protocol rules establish when to signal an error frame, and this entire process is completely handled in silicon. Again, no software intervention is required or necessary. Use the upcoming CAN receive and transmit state machines to understand more about how and when the Error Frame is used.

26.8.4 CAN Error Counting

The CAN protocol requires the use of error counters – one for reception and another for transmission – to determine the error state of the CAN Controller. Implemented in silicon, these counters are used to establish the key states of Error Active, Error Passive, and Bus Off.

Figure 101 shows the upper-level details and the major error counting rules. Use this diagram with the CAN receive and transmit state machines to learn more details.

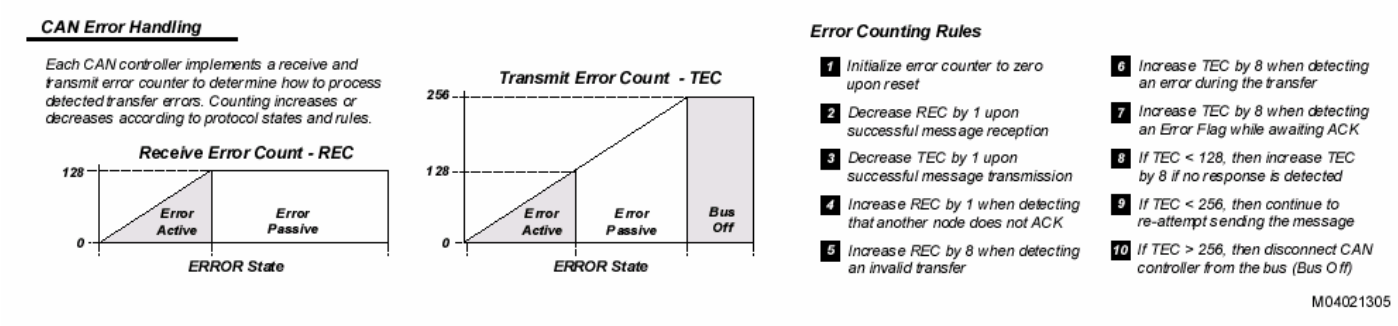

Figure 101 – CAN Error Counting

26.8.5 The CAN Receive State Machine

Figure 102 shows the silicon-based CAN receive state machine.

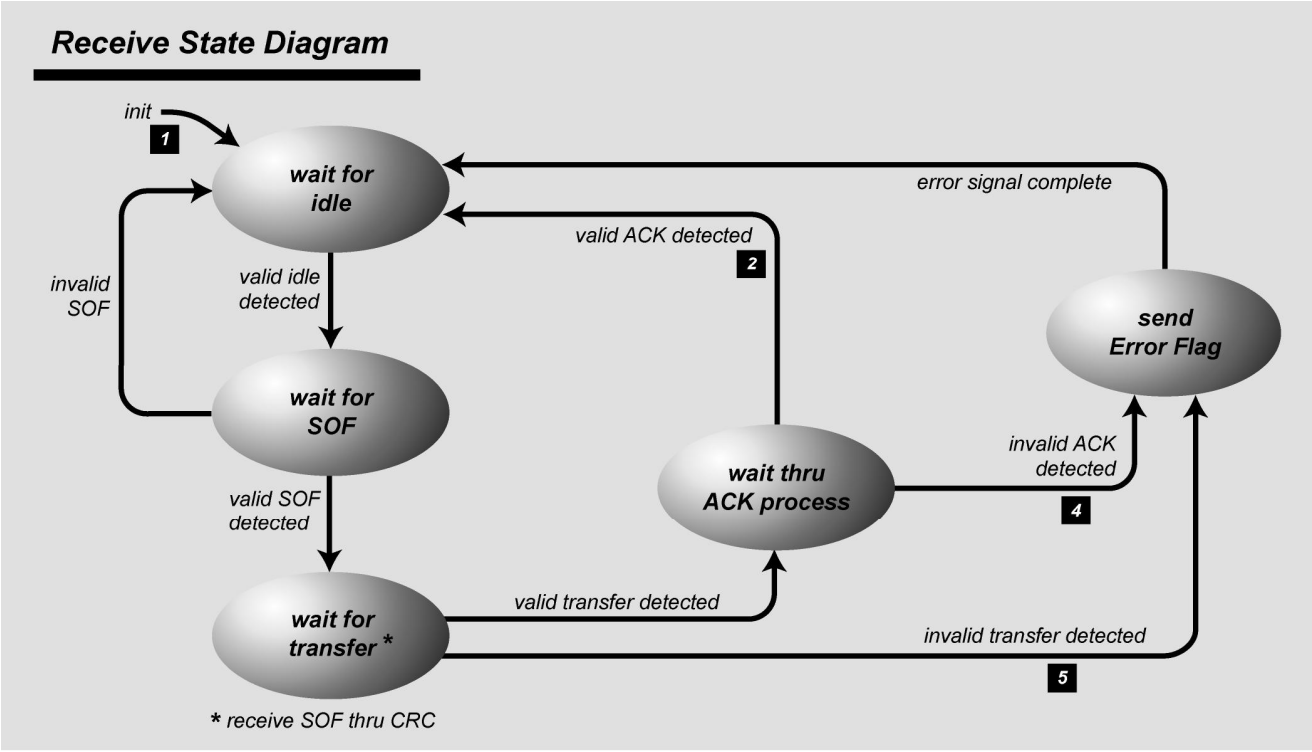

Figure 102 – The Basic CAN Receive State Machine

26.8.6 The CAN Transmit State Machine

Figure 103 shows the basic structure of the CAN transmit state machine. With the exception that an application must use software to initiate the transmission of a message, this transmit process is essentially all implemented in silicon.

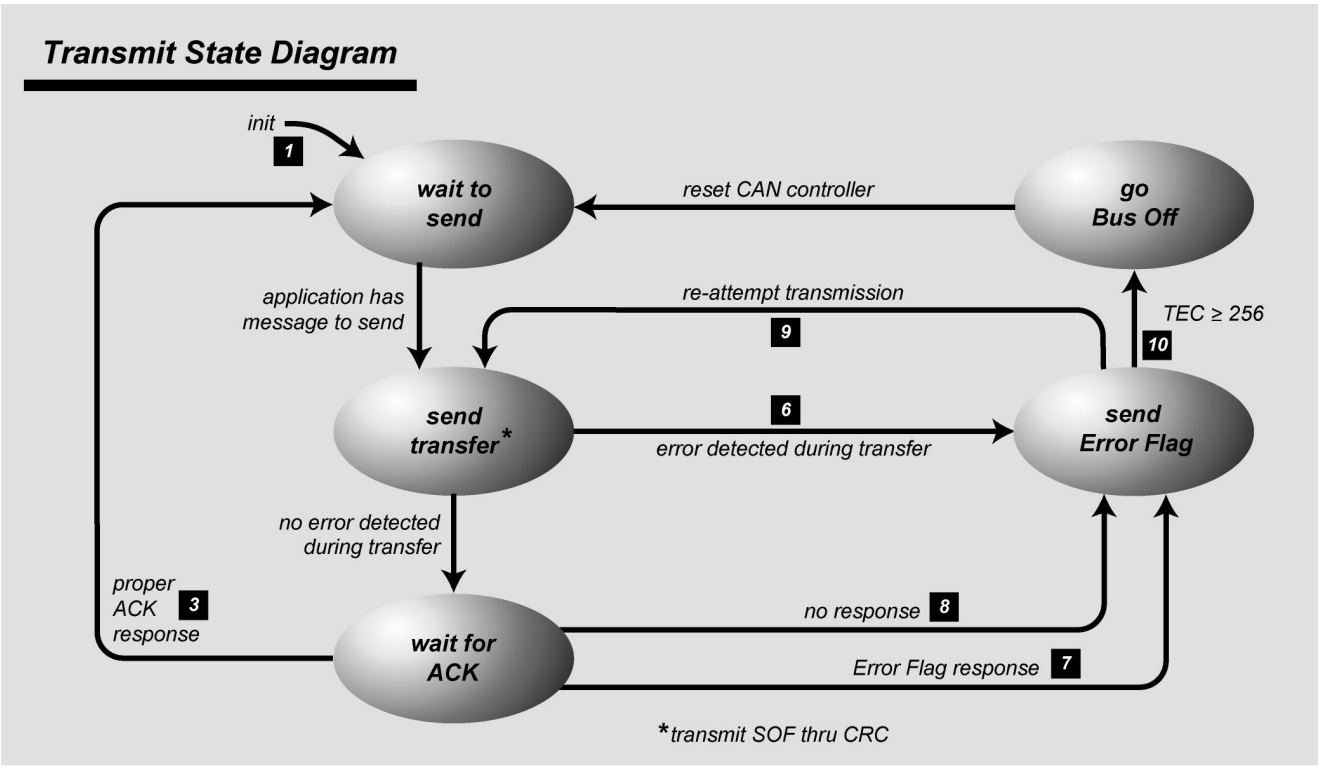

Figure 103 – The Basic CAN Transmit State Machine

26.8.7 CAN Implementation Ingredients – CAN Controller, Software, Physical Layer

As shown in Figure 104, to functionally implement a CAN node in a module essentially requires three key items: a CAN Controller, CAN communications software, and a suitable Physical Layer.

Note: A CAN Controller with no connection to a Physical Layer will not function. The CAN protocol rules require that some method be used to allow comparison of the transmitted bit with the receive bit. Without being connected to a suitable CAN Physical Layer, the CAN Controller cannot compare its transmitted bit with the receive bit.

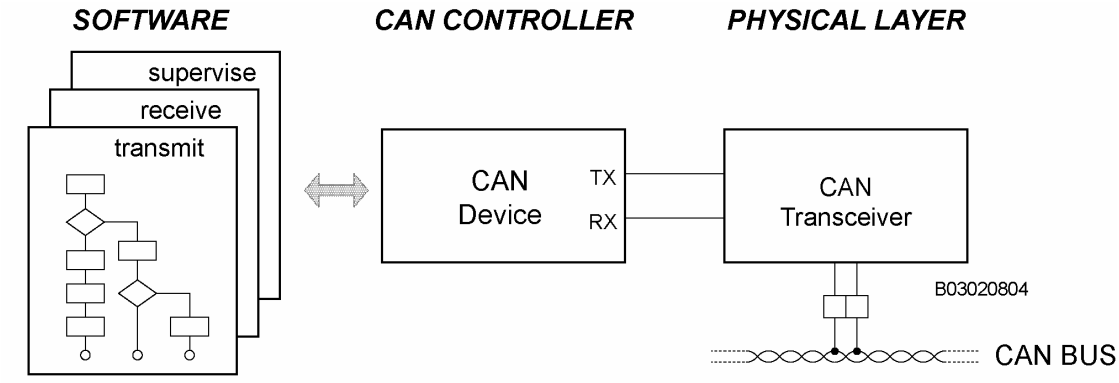

Figure 104 – Key Implementation Components

26.8.7.1 CAN Controller

CAN Controllers are available as integrated silicon peripherals co-resident with the microcomputer or as separate, external integrated circuits. A wide variety of CAN Controllers are available across the entire semiconductor industry. Some devices include multiple CAN Controllers. The CAN Controller has two major interfaces - one, an interface that interconnects to the selected Physical Layer and the other interface that interconnects to the network communication software.

Two prominent types of CAN Controller architectures are common-

- A simple transfer buffer architecture - also sometimes referred to as "Basic CAN"
- A moderately complex Dual-Port RAM-type architecture - also sometimes referred to as "Full CAN"

All new CAN Controllers implement the CAN Protocol rules as defined by the CAN 2.0B specification. When considering the available devices, notice that several semiconductor manufacturers also provided special features to allow special CAN operations. For example, a "listen only" mode is available on some CAN Controllers.

26.8.7.2 CAN Communication Software

CAN communication software is needed to interconnect the application to the network-based communication system. Receiving a message in the CAN Controller does not automatically result in any action. In the software domain, you must extract the information from the message, examine this information, and then potentially take the appropriate action. In addition, when an application needs to move information onto the network, the communication software is used to load the appropriate message into the CAN Controller for transmission.

You must develop CAN communication software to handle these fundamental transmit and receive activities. Typical CAN communication software components might also include the following:

- Network initialization
- Transmit message construction
- Receive message parsing
- Network supervision
- Network relationship management
- Network error management - like "bus off" handling

26.8.8 CAN Physical Layer

The CAN Physical Layer includes the network interconnection, the physical wiring (or other media), the transceiver, and perhaps a few other interface-associated components. The Physical Layer provides a direct logical interface to the CAN Controller and is one of the concentration areas directly related to electronic hardware design.

Several CAN Physical Layers are available, including many industry-specific standards. The semiconductor industry has created a selection of CAN transceivers to accommodate these different CAN Physical Layer standards. Keep in mind that several CAN interfaces are tailored to the application-specific environment.

The following lists three common CAN transceivers used by the auto industry:

- The high-speed, 2-wire, differential transceiver
- The low-speed, 2-wire, fault-tolerant transceiver
- The low-speed, 1-wire transceiver

26.8.9 Physical Layer Choices

While the overwhelming choice of CAN Physical Layer across most industries is the high-speed, 2-wire, differential transceiver, other possible CAN Physical Layer choices (of which some are already commercially available) include the following:

- Fiber-optic
- Infrared
- Ultrasonic
- Audio
- RF or wireless
- Magnetic
- Power line carrier

26.8.10 Basic CAN Physical Layer Principles

26.8.10.1 Dominant/Recessive Logic

Commonly used to establish the state of a shared medium, dominant/recessive logic is a simple two-state form of logic in which the logic state of the shared medium is established as either dominant or recessive, as shown in Figure 105.

Figure 105 – The Two States of Dominant and Recessive

For example, we could use an optical shared medium in which the dominant state could be indicated by light and the recessive state could be indicated by dark.

Once we begin to combine the dominant and recessive states on the shared medium, the fundamental operation rules of dominant/recessive logic are as follows:

- If any device places the shared medium into a dominant state, then the resulting shared medium state is dominant
- If no device or network member places the shared medium into a dominant state, then the resulting shared medium state will be recessive

As shown in Figure 106, this combinational logic demonstrates that the dominant state always wins over the recessive state.

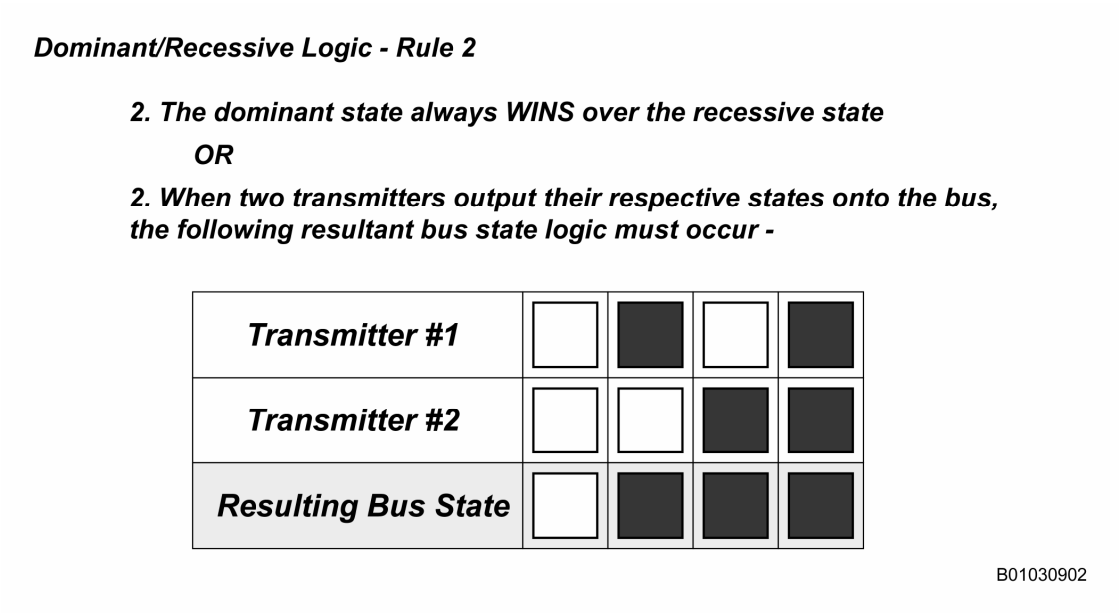

Figure 106 – Dominant/Recessive Logic

When dominant/recessive logic is used by the CAN protocol, the recessive state is also used to indicate the CAN protocol's idle condition, that is, when no CAN message is being transferred, as shown in Figure 107.

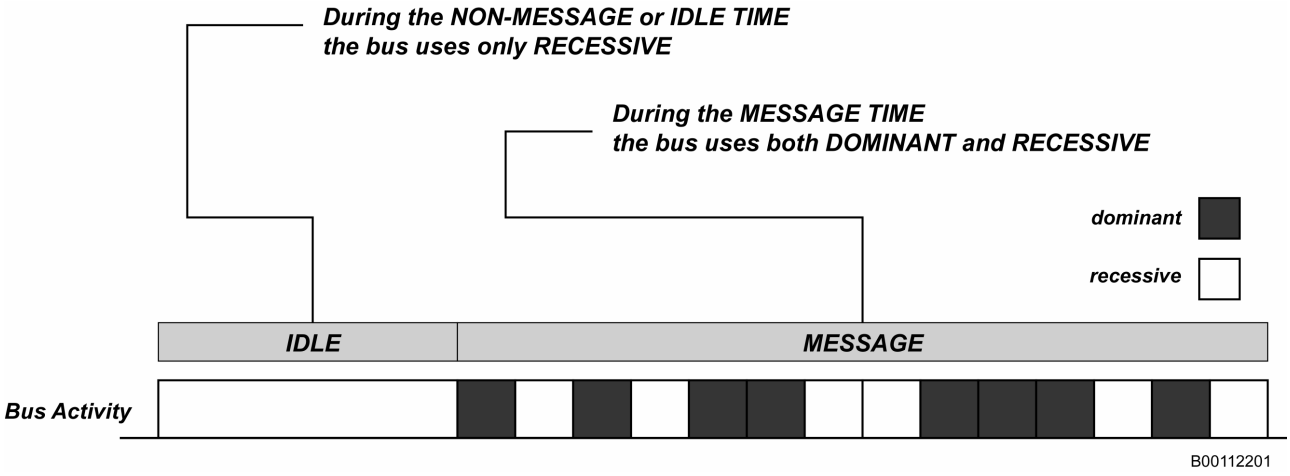

Figure 107 – The CAN Protocol Idle State Is Recessive

26.8.11 CAN Controller – Dominant/Recessive Logic Levels

Based on the CAN protocol rules and shown in Figure 108, all CAN Controllers use the same logic reference at the Transmit (TX) and Receive (RX) pins. Logic level "zero" is the dominant state, and logic level "one" is the recessive state.

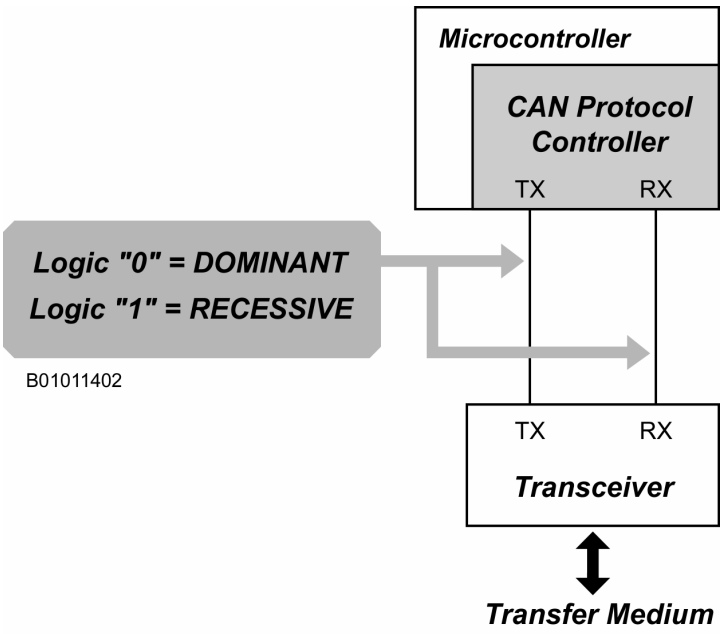

Figure 108 – Dominant/recessive Logic Levels Referenced to the CAN Controller

26.8.12 Common Dominant/Recessive Logic Implementations

When implementing dominant/recessive logic, the dominant state is commonly established by an activated electronic device, while the recessive state is automatically established by the absence of any activated device.

Dominant/recessive logic is easy to implement. Figure 109 shows two simplified implementations; one, based on open collector logic, and the other based on open emitter logic.

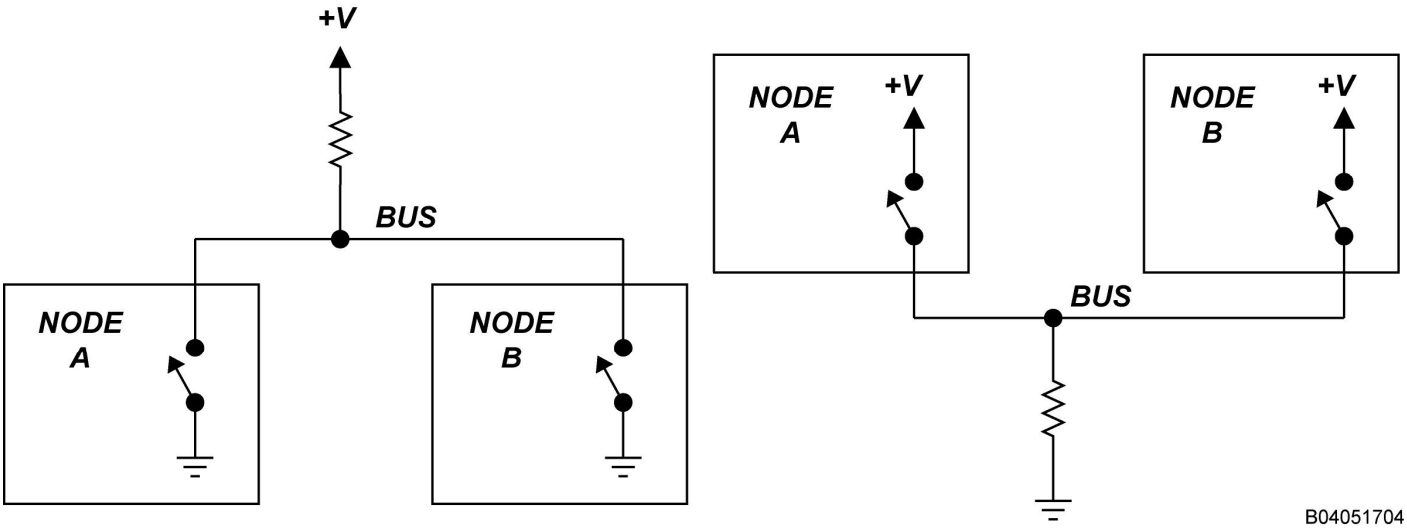

Figure 109 – Implementing Dominant/Recessive Logic with Open Collector and Open Emitter Logic

Independent of the circuit realization, both open collector and open emitter logic provide the same logic requirements for dominant/recessive logic as shown in Figure 110.

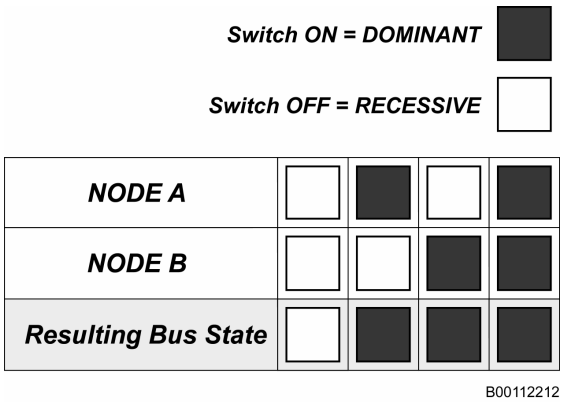

Figure 110 – Logic Table for Either Open Collector-type or Open DEmitter-type Implementations

26.8.13 The Common High-Speed, Two-Wire Differential Transceiver

The basic model of the common high-speed, two-wire differential transceiver is shown in Figure 111.

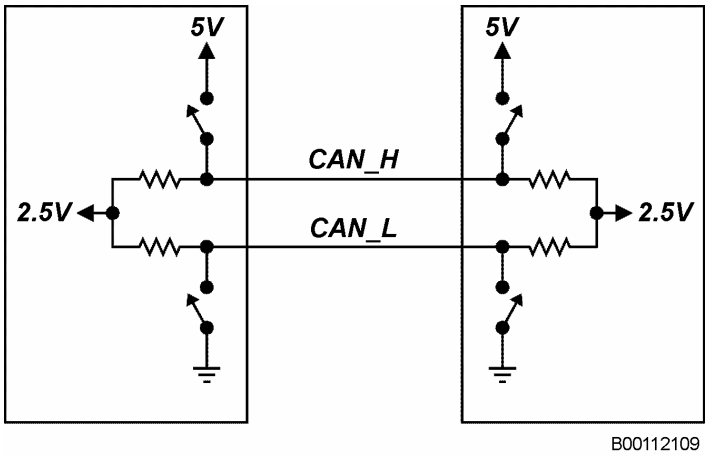

Figure 111 – Basic Model of the Industry-Standard Dual Wire Differential High-Speed Transceiver

When differential transmission and reception is used, typically a pair of twisted wires, either shielded or unshielded (dependent on EMC requirements) is employed. The individual wires are designated as CAN_L and CAN_H, and these CAN signal wire names are matched with the dominant state. When the bus is in the dominant state, CAN_H is HIGH and CAN_L is LOW.

Also notice that the recessive state is electrically placed at a mid-supply voltage level, as this minimizes the generation of RF energy.

26.8.14 The Common CAN Transfer Medium - Wire

For most high-speed CAN communication systems, copper wire is used as the transfer medium because of its low cost when compared to other alternatives.

26.8.14.1 The Common CAN Wiring Topology – the Linear Bus

Defined as the physical structure of the network that establishes how the shared media is interconnected, the wiring topology selected for almost all high-speed CAN bus systems is the Linear Bus Topology (shown in Figure 112). While other topologies including Tree, Star, and Loop are sometimes considered, their respective disadvantages, especially their signal reflection characteristics, typically render them useless.

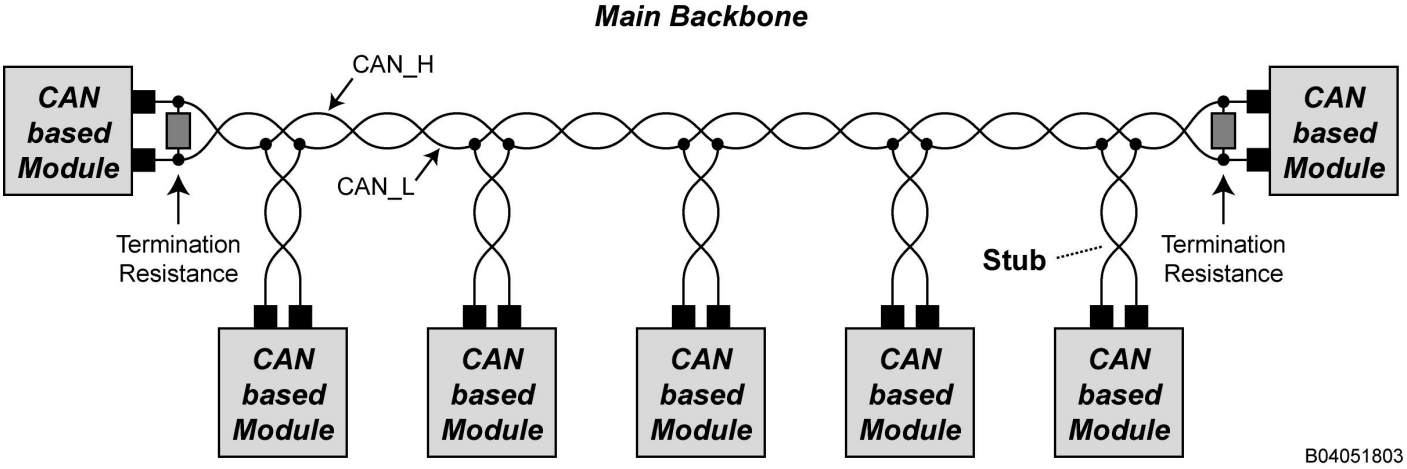

Figure 112 – Linear Bus Topology

With termination resistance at the physical ends of the bus, the **Backbone** is the main cable (or trunk) of the CAN bus network. Additional CAN nodes are attached to the backbone cable using a **Stub** or drop cable.
The advantages of this topology include the capacity to handle long lengths, the ease of adding stubs, and the ability to control signal reflection. Its primary disadvantage, however, is the need to insert bus termination to deal with transmission line effects.

For high-speed applications, the wire length and stub length becomes critical at high bit rates. As one continues to increase the wiring length, normal communications will eventually cease. At the CAN communication level, this appears as a continuous source of CAN error frames.

The maximum bus length is a system-designated parameter that depends upon many factors including bit rate, the stub length, the capacitance of the wire, and the internal capacitance of the nodes (transceiver and any utilized protection components).

26.8.14.2 Maximum Bus Length

Because excessive wiring length will typically become a source of CAN indication errors, what length should be used? For learning projects it is recommended to limit the backbone length to 100 feet and the stub length to 3 feet.

26.8.14.3 Bus Signal Reflections

A signal reflection occurs when an electrical signal, electromagnetic wave, or pulse propagates on a transmission line and encounters a discontinuity on the transmission line, which results in a reflection. As the reflection splits into the forward and backward directions, the signal reflection creates additional waves that mix with the original transmission signal. Signal reflections are common to high-speed CAN bus networks. Even the tee-connection used to add a stub will contribute a small amount of signal reflection.

26.8.14.4 Bus Termination – 120 Ohms at the Far Ends of the Bus

Used by most high-speed, differential communication networks, bus termination is the addition of electrical components to reduce signal reflections. So-called end-point bus termination is an integral component of high-speed CAN networks.

Bus termination on a high-speed network is commonly applied by using two simple resistors that must be placed at the far ends (or *end points*) of the bus. The span of wire between the two termination nodes is typically referred to as the ‘backbone’.

Note: - While some bench-top experiments may seem to indicate that termination is not excessively critical, such is not the case as the wire length increases.

The resistor values are usually 120 ohms, because this is approximately the characteristic impedance of most twisted pair cables used for communications. A more precise value for termination resistance can be used, the common 120 ohm value is more than adequate.

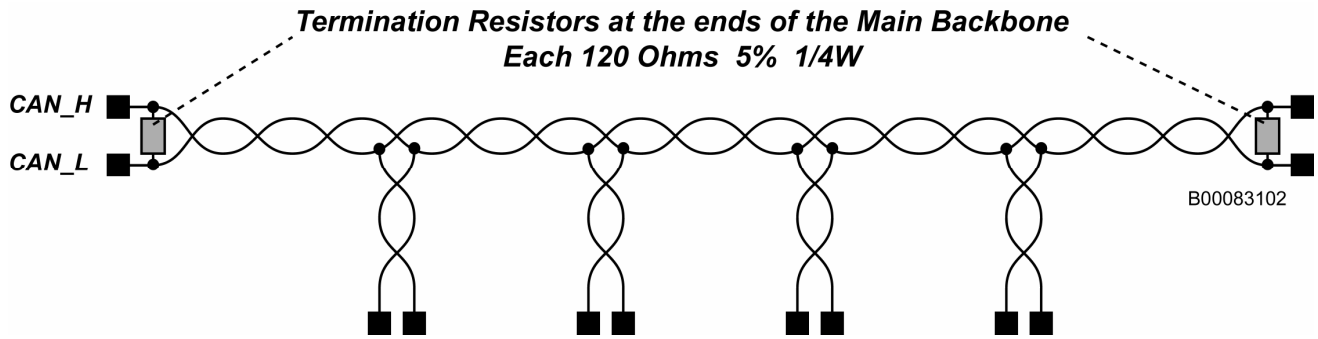

Figure 113 – Location of Termination Resistance

26.8.14.5 Incorrect Termination Location

Be careful when creating a wiring setup that results in a new connection that results in re-establishing the “far ends of the bus”. It is possible to take a working CAN network, that is properly terminated, add a new node or CAN tool and effectively kill the CAN communication network. This happens because the new wiring path causes additional bus signal reflections. As shown in Figure 114, if you connect the tool at one of the far ends of the bus, the added wire length has just become a part of the main backbone. This means the termination resistance should move to the tool’s interface.

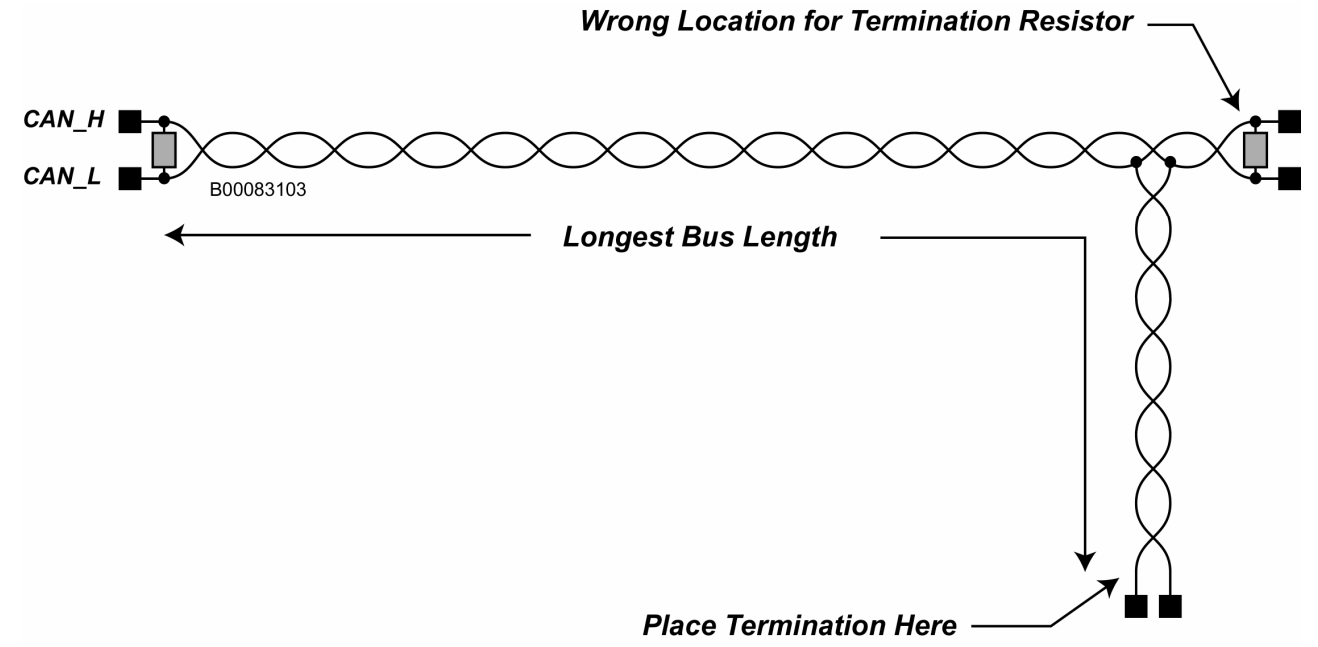

Figure 114 – Upsetting the Termination Resistance Location

As shown in Figure 115, the solution is simple. Tool connections should be added into the middle section of the backbone.

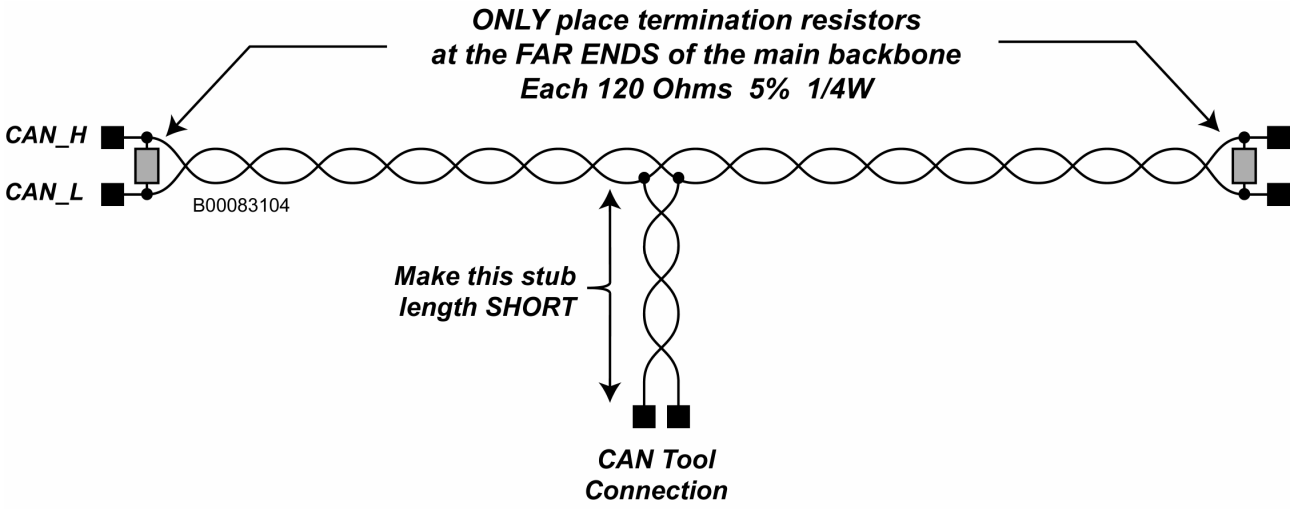

Figure 115 – Proper High-Speed CAN Tool Connections

26.9 First-Time CAN Steps

As you will learn in sections **Error! Reference source not found.** to 26.9.1.4, the first-time steps of developing a CAN bus application are relatively simple.

26.9.1 Step One – Wiring First

Build the CAN wiring system first.

26.9.1.1 Type of Wire

You may use almost any type of wire as long as you can twist the CAN_H and CAN_L wires.

26.9.1.2 Twisting the Wire

Twist the high-speed CAN communication wires (CAN_H & CAN_L) to minimize radio frequency energy from entering or exiting the communication bus. Only a small amount of wire twisting is necessary. 10 to 12 twists per foot is adequate for most applications. Excessive twisting only increases capacitance, and this will tend to slow up the communication signals and potentially result in error frames. Because twisting need not be precise, twisting by hand or using a drill will work.

26.9.1.3 Don't Forget the Ground

High-speed CAN signal transmission actually requires three wires: 2 communication differential signal wires (CAN High (CAN_H) and CAN Low (CAN_L)) plus a suitable ground connection between the different CAN modules.

While the ground connection is often pre-established, this may not be the case when you move to your actual application. For many laboratory bench-top experiments that interconnect electronic modules using the same power supply, all of the modules share the same power supply ground. As another example, in most automotive vehicle applications, the ground connection is also pre-established because all modules use the same chassis ground. However, when your CAN bus application uses a distributed power source in which each module has its own local power source, then you must interconnect the grounds between these various modules using an additional ground wire.

26.9.1.4 Include Termination Resistance

Remember to include two termination resistors, each 120 ohms, for the entire CAN bus. The placement of these two termination resistors is quite critical, as previously noted. Place the resistors at a far ends of the bus, not in the middle.

26.9.2 Step Two – Add Modules Incrementally

After you design your distributed application using CANoe to model the functionality of all the nodes, then incrementally add individual modules one at a time to the CANoe model to verify proper operation. It is much easier to debug distributed application problems in this manner.

26.9.3 Notes on Building Your Own CAN-Based Module

Try to use an off-the-shelf microcomputer development module that includes a high-speed CAN interface.

For the Physical Layer, use the common high-speed CAN transceiver. While this electronic component is widely available as a surface mount device, a few "through hole" parts, such as the Philips 250 or 251, can be obtained. If you have difficulty in obtaining these CAN transceivers, consider using a CAN-compatible interface using the ubiquitous RS 485 transceiver. This component can be effectively used and is explained in Vector's application note entitled “Using the RS 485 Transceiver as an Alternative CAN Interface”. The connection to the CAN Controller transmit pin (usually labeled as TX) is another usable point to examine the CAN transmission. The TX pin operates with the traditional 5-volt logic and makes the signal triggering somewhat easier when compared to the differential signals on the CAN bus.

27 Appendix B: Abbreviations/Acronyms

ABS	Anti-lock Braking System
ACK	acknowledgement bit in a CAN message
ANSI	American National Standards
API	Application Programmable Interface
ASCII	American Standard Code for Information Interchange
BEAN	Body Electronics Area Network
BTI	Bit Timing Interval
BTR	Bit Timing Register
CAN	Controller Area Network
CAN_H	CAN High signal wire
CAN_L	CAN Low signal wire
CANdb	CAN database
CAPL	CAN Access Programming Language
COM	Component Object Model – a Microsoft Corporation standard
CRC	Cyclic Redundancy Check
DIR	Direction of transmission
DLC	Data Length Code
DLL	Dynamic Linked Library
ECU	Electronic Control Unit
EOF	1. End Of Frame 2. End Of File
GND	ground wire
I/O	Input/Output
ID	Message identifier
IDE	Identifier Extension bit in a CAN message
IFS	Inter-Frame Space
J1939	a SAE (Society of Automotive Engineers) network standard for applications in construction, material handling, and forestry machines
LED	Light-Emitting Diode
LIN	Local Interconnect Network
LSB	1. Least Significant Byte 2. Least Significant Bit
MDI	Multiple Document Interface
MPH	Miles Per Hour
MSB	1. Most Significant Byte 2. Most Significant Bit
MSG	Message
NMEA	National Marine Electronics Association
OCR	Output Control Register
OEM	Original Equipment Manufacturer
PG	Parameter Group – messages in J1939 networks
PGN	Parameter Group Number – message identifiers in J1939 networks
PHYS	Physical unit
RGB	System used in computers for displaying color (red, green, blue)
RS232	A standard interface for serial data communication
RTR	Remote Transmission Request bit in a CAN message
RX	Receive
SOF	1. Start Of Frame 2. Start Of File
SPI	Serial Peripheral Interface
SRR	Substitute Remote Request bit in a CAN message
TX	Transmit
TXREQUEST (TxRq)	transmit Request
UART	Universal Asynchronous Receiver/Transmitter
VAN	Vehicle Area Network

28 Appendix C: Glossary

* symbol	A wildcard symbol used to replace the parameter(s) in an **on key** or **on message** event procedure to represent all keys or all message respectively. For example, all keystrokes on the keyboard will execute this one and only **on key** event procedure: **on key *** **{** **...** **}** P-blocks (program blocks in CANoe and CANalyzer), by default, stop the message traffic unless an **on message** event procedure is defined to pass all the messages like this: **on message *** **{** **…** **output(this);** **}**
Acceptance filter	Acceptance filtering implements a pair of mask and code which allow message objects (hardware buffers) to receive messages with a range of IDs instead of all or no IDs. This gives the application the flexibility to receive a wide assortment of specific messages from the bus. Because of just one pair of mask and code, unwanted message IDs may still pass the hardware filtering; therefore, these unwanted messages have to be filtered in the software application (CANalyzer or CANoe). The controller observes all messages on the CAN bus and stores any message that matches the pair of mask and code. It is possible to accept just one ID, no IDs, or all IDs.
Acknowledge (ACK)	CAN provides a global acknowledge field which tells the transmitter that at least one - although not saying how many or which ones - of the network nodes received the message properly, or, alternatively, that none were received. This field is simply one bit in the CAN message. Because CAN is based on broadcast communication, all nodes must respond with an acknowledged signal if they have not detected an error.
Active State	1. The state of a CAN controller after it is initialized, and whenever the transmit and receive error counters are less than or equal to 95. The error counters are incremented accordingly due to the type of error frames. If either error counters exceed 95, the CAN controller is in the Warning Limit State, follow by the Passive State, and finally Bus Off State. 2. The electrical state of a bus wire that results when one or more nodes transmit the "dominant" bus condition by "turning on" their physical layer transmit circuit. Alternate names: Active, Dominant. Refer to the designated physical layer specification for signal levels.
Analysis block	The Analysis Branch of the **Measurement Setup** window in both CANalyzer and CANoe contains six analysis blocks by default, each responsible for its analysis functions. For example, the Graphics block is an analysis block that is used to graph received signals. The analysis blocks come with its own window except the Logging block that is only used for logging bus traffic.
API	Application Programmable Interface. The calling conventions (Interface) by which an application program gains access to system resources and other services, often for the purpose of sending and receiving data. An API is defined at the source code level and provides a level of abstraction between the application and the kernel (or other privileged utilities) to ensure the portability of the code.
Array	1. A program data structure that contains a fixed maximum number of elements of the same data type which are referenced directly by means of an index, or subscript. The elements in a multidimensional array are ordered on N dimensions (N >= 1). Each element is accessed by N indices, each of which represents the element's position within that dimension. Arrays are used in Vector's CAPL as if they are used in C. 2. Another name for a matrix, which is an array of quantities or elements in a prescribed form. A matrix is usually capable of being subject to a mathematical operation using an operator or another matrix. A matrix of circuit elements can be capable of performing functions such as code conversion. The matrix elements may be diodes, transistors, magnetic cores, or other binary devices.

ASCII	American Standard Code for Information Interchange. A seven-bit code, intended as a U.S. standard (ANSI X3.4) for interchanging information among communications devices. The code represents characters as binary numbers, used on most microcomputers, computer terminals, and printers. In addition to printable characters, the ASCII code includes control characters to indicate carriage return, backspace, and so on. The first 32 values are reserved for control codes, with the remaining 96 available for graphics (alphabetic, numeric, and special characters). Several 8-bit variants exist; see ISO 646.
Attribute	A common characteristic of a database object (network, node, message, signal, environment variable, and so on.) that is used to represent the relationship of that object type. An attribute is defined by its name, data type, value range, and default value.
Baud rate	The total number of bits transmitted per unit of time.
Bit	A single digit in the binary (base-2) number system (binary digit). A single character in a binary number. These digits are represented electronically in the computer by pulses of electricity. These pulses are generated by electric circuits, which are either on or off. Usually a circuit that is on represents the binary number 1, while a circuit that is off represents the number 0. One bit is thus a single pulse in a coded group of pulses. Combinations of enough of these 1s and 0s can represent any kind of data.
Bus	A set of conductor logic lines (wires, PCB tracks, parallel circuit or connectors in an integrated circuit) that transfers data by electric impulses as logic signals from one connected component or functional unit to any other. Another word for data transfer medium or physical layer.
Bus Off State	A CAN controller state that occurs when the Transmit Error Counter (TEC) reaches a value of 256. This transition is usually caused by controller failure or extreme error bursts resulted in error frames. The CAN controller within this state is disconnected from the bus by setting it in a state of high resistance. During this condition, the CAN controller can be reset by the application (usually after certain amount of idle time) or by a hardware reset. The reset will set the transmit and receive error counter to zero and the CAN controller state to be Active (see Active State). The hardware interface cards used by CANalyzer and CANoe can only be reset when the measurement is restarted in the software. Another method is by calling either the **resetCAN()** or **resetCANEx()** functions in CAPL.
Byte	A group of 8 bits, representing any of 256 values. A byte may represent a single binary number, 8 bits of a longer binary number, two decimal digits, one decimal digit with plus or minus sign, or a graphic character (for example, a letter, a number or any symbol). A standard CAN message contains anywhere from 0 byte to a maximum of 8 bytes.
CAN	The Controller Area Network (CAN) is a serial communications protocol that efficiently supports distributed real-time control with a very high level of data integrity.
CAN controller	The chip or integrated silicon (with the microcontroller) that processes the CAN protocol. It interconnects the microcontroller to the Physical Layer.
CAN_H	The CAN_H bus wire of a differential connection is fixed to a mean voltage level during the recessive state and is driven in a positive voltage direction during the dominant bit state.
CAN_L	The CAN_L bus wire of a differential connection is fixed to a mean voltage level during the recessive state and is driven in a negative voltage direction during the dominant bit state.
CANalyzer	A powerful tool use to observe, analyze, and supplement data traffic on a distributed CAN network.
CANdb	1. First generation database editor before CANdb++ use to build *.dbc files. 2. The formats of a *.dbc file. Sometime Vector's database (*.dbc) files are called CANdb files.
CANdb++	A database editor used to build or modify database (*.dbc) files.
CANoe	A powerful tool that supports the entire development process of distributed CAN networks, from planning to testing. It features model creation, simulation, functional testing and analysis.
CAPL	CAN Access Programming Language. A programming language based on C used in CANoe and CANalyzer for node emulation and analysis.
CAPL Browser	An editor/compiler for CAPL programming

CRC	Cyclic Redundancy Check. A specialized 15-bit checksum (every message has this field) that provides the ability to detect serial communication transfer errors and increase the reliability of data transfers. The CRC computation uses a digital serial data single bit error detection algorithm. It is derived from a cyclic redundancy code.
Data frame	A standard (11-bit ID) or extended (29-bit ID) message frame that carries 0 to 8 bytes of data. An extended message may consist of more than one data frame because it has more than 8 data bytes. A data frame cannot exceed 8 data bytes.
Data Length Code (DLC)	A 4-bit field within the Control Field that defines the number of data bytes in a CAN data frame. The DLC bits can code data lengths from 0 to 8 bytes; other values are not permitted.
Database	A look-up table that symbolizes the data traffic on a CAN network. In a database, you can define the nodes, messages, signals, and their associations for a given CAN network.
ECU	The acronym for Electronic Control Unit. It consists of the microcontroller and all of the components comprising the functionality of a node.
Environment variable	Variables defined in a database that is capable of exchanging information between CAPL programs, CAPL programs and user-defined panels, and CAPL programs and a third party application utilizing CANoe's COM interface.
Error frame	A frame responded to an error on the CAN bus with either six dominant bits or six recessive bits.
Event	An action denoted by an object to code the response. Events are caused by a key pressed, a change in the CAN controller state, a message, and so forth.
Event procedure	A block of CAPL code that is executed when an event occurs. CAPL permits the definition of event procedures for several different kinds of events, such as on message, on key, on errorframe, on timer, etc.
Gateway	A node used to interconnect the transfer of data between two physically separate networks that use two different protocols. As differentiated from a bridge, a gateway acts as a protocol-to-protocol converter.
Identifiers (ID)	A field in a message that contains network address information. The 11 or 29-bit identifier is usually unique in a closed CAN network and is used to define a message and the data it carries.
Message	A block of data (formatted into a collection of signals) transmitted over the CAN bus in one or more data frames. It is associated with a unique CAN identifier.
Network	1. A distributed system capable of supporting communications between two or more nodes. 2. A method of interconnecting a collection of two or more distributed processing subsystems, devices, computers, sensors, and actuators with the intended purpose of sharing information.
Node	1. A key physical and functional element of a distributed embedded system, usually located within a module that supports the physical interconnection to a shared media and allows data communication according to a designated protocol. 2. An electronic subsystem or circuit used to interconnect a module to a network.
Panel	A graphical interface built within the Panel Editor, used to provide user inputs to a CAPL program and/or to display data received from the CAN bus.
Panel Editor	An editor used to build panels.
Passive State	A CAN controller state that arises when the TEC (transmit error counter) or REC (receive error counter) reaches or exceeds 128.
Protocol	A set of rules supporting the transfer of information between two entities, nodes, or OSI layers concerning the format and content of messages to be exchanged. In general, a protocol will govern the format of messages, the precedence among messages, generating checking information and flow control, as well as actions to be taken in the event of errors.
Remote frame	A message frame transmitted on the CAN bus to request the transmission of a Data Frame with the same identifier.
Signal	Independent data spaces that occupy the data field of a data frame. The size of a signal range from 1 bit to 32 bits.
This keyword	A local variable used to reference the message within an on message event, the key within an on key event, and the environment variable within an 'on envvar' event.

Time stamp	A time given by the CAN controller when a message changes state. The time stamp can reference when the message is transmitted, received, or requested to transmit.
Transport protocol	A set of rules to support information transfer by breaking long messages into a series of shorter messages to accommodate network protocol requirements. If the receiver is not capable of managing the entire transfer because of microcontroller resource limitations, some form of “flow control” mechanism may be appropriate to control the overall transfer process.
Value table	Used to assign symbolic names to signal or environment variable values.

29 Index

***** symbol, 75
ACK, 178
Analysis Branch, 19, 25
analysis functions
  setup, 124
Application Layer, 173
array, 42
  elcount(), 43
attribute
  define, 28
attributes
  define, 127
break statement, 56
bus length, 186
Bus Off, 95
bus parameters, 123
Bus Statistics window, 13
bus termination, 186
BYTE, 81
CAN, 79, 166
  development steps, 188
  features, 174
CAN communication software, 181
CAN communications, 166
CAN controller events, 94
CAN Controllers, 181
CAN protocol, 174
CANalyzer, 6
  architecture, 10
  download, 9
  hardware connection, 9
  introduction, 9
CANdb++, 26, 126
  loading, 127
CANoe, 21
  architecture, 23
  bus parameters, 123
  configuration, 122
  development, 121
  download, 21
  hardware connection, 21
CAPL
  C/C++ difereneces, 37
  case sensitive, 39
  comments, 38
  data types, 40
  keywords, 39
  naming conventions, 39
CAPL Browser, 30
  organization, 31
CAPL DLL, 140
  compile, 143
  demo, 145
  export table, 142
  limitations, 141
  linking, 143
CAPL program
  compile, 34
  text file format, 35
CAPL programs
  constructing, 106
  organization, 106
  text editors, 35
Channel Filter, 14
COM, 151
  events, 156
  message transmission, 159
  registration, 151
  VB demo, 160
  VB project configuration, 152
COM interface, 151
COM object
  environment variables, 155
  function, 157
  measurement, 153
  signal, 153
compilation error, 34
Compiler window, 31
constants, 43
  character, 44
  floating point, 44
  integer, 43
  symbolic, 45
continue statement, 57
Control Field, 178
CRC, 178
Cyclic Redundancy Check, 178
Data Field, 178
Data Link Layer, 172
data types, 40
Data window, 13
database, 26, 126
  associating, 131
  association, 26
  define attributes, 127
  define environment variables. *See*
  define messages, 128
  define nodes, 128
  define signals, 129
  define value tables, 128
  object associations, 130
  saving, 131
DIR, 79
distributed application, 166
  partition, 168
distributed functions, 166
  data structures, 173
  example, 168
  file transfer, 170
  transfer dialog, 169
DLL, 140
dominant, 182

dominant/recessive logic, 182
do-while statement, 55
DWORD, 81
elements, 135
  alarms, 136
  associating, 135
  configuring, 135
  control, 135
  display, 135
  I/O, 135
  static, 135
engineering values, 82
environment variable
  define, 28
environment variables, 96
  declare, 97
  define, 133
  initialize, 97
  type, 97
Error Active, 95
error counters, 179
Error Frame, 179
Error Frames, 86
Error Passive, 95
event procedure
  creation, 74
  requirements, 74
Event Procedure window, 31, 33
event procedures, 32, 73
events, 73
events priority, 76
Events window, 31, 32
example
  CAPL control logging, 115
examples
  bus off condition, 114
  conditionally periodic message, 110
  error frame, 113
  file I/O, 116
  message response, 112
  periodic messages, 108
  received messages, 111
  using panels, 113
  write(), 108
file I/O functions, 100
Filter block, 15
filtering blocks, 14
for statement, 55
function
  compatibilities, 72
function blocks, 13
functions, 59
  byte swapping, 64
  CAN and port, 68
  categories, 60
  database, 64
  environemnt variable and panel, 70
  language support, 71
  logging, 65
  mathematical, 60
  measurement control, 66
  message handling, 63
  naming conventions, 60
  overload, 59
  Replay block, 69
  statistics, 68
  string handling, 66
  time, 62
  user interface, 61
  user-defined, 59
Generator block, 15
getValue(), 98
Global Variables window, 31, 32
Graphics window, 13
hotspots, 13
identifier, 177
if statement, 52
if-else statement, 52
Interactive **G**enerator block, 16
keyboard event, 87
LONG, 81
Measurement Setup window, 12
message
  ACK Field, 178
  control field, 178
  CRC Field, 178
  data access, 81
  data field, 178
  declaring, 78, 79
  define, 28
  definition, 78
  identifier, 177
  object, 78
  receive, 83
  selectors, 79
  structure, 177
  transmit, 83
Network Layer, 173
network node
  create, 106
node
  define, 28
on errorFrame event, 86
on key event procedure, 87
on message event, 83
on preStart event procedure, 90
on start event procedure, 90
on stopMeasurement event procedure, 90
on timer event procedure, 91
operators, 46
  arithmetic, 46
  assignment, 47
  bitwise, 50
  Boolean, 48
  miscellaneous, 50
  relational, 49
  unsupported, 51
OSI model, 172
P Block
  placement, 107

panel
  definition, 132
Panel Editor, 132
panels, 29
  advantages, 133
  associating, 139
  bitmaps, 138
  requirements, 133
PC Board, 12
phys, 82
Physical Layer, 172
  types, 182
physical values, 82
Prerequisites, 2
Presentation Layer, 173
ProFile functions, 102
putValue(), 98
raw values, 82
recessive, 182
Replay block, 17
return statement, 57
run-time errors, 34
selectors, 79
serial port, 103
Session Layer, 173
signal
  define, 28, 82
signal reflection, 186
Simulation Branch, 24
SOF, 177
state machine
  receiver, 179
  transimitter, 180
Statistics window, 12
switch statement, 53
syntax
  color, 33
system events, 89
this keyword, 57, 75, 84, 87, 98
TIME, 79
timer
  declare, 91
  reset, 92
  stopping, 92
timers, 91
topology, 185
Trace window, 12
transceiver, 185
Transmit Branch, 18
Transport Layer, 173
Trigger block, 15
TYPE, 80
type casting, 42
value table
  define, 28
variables
  declarations, 41
  static, 41
while statement, 54
WORD, 81
Write window, 13

Visit our Website for more information on
>News
>Products
>Demo Software
>Support
>Training Classes
>Global Locations

www.vector-cantech.com

vector
the art of engineering