Tips and tricks for the use of CAPL

CAPL is a programming language available in the software tools CANoe and CANalyzer. In three consecutive articles, CAPL fundamentals will be discussed as well as tips for all levels of user knowledge.

Authors

Marc Lobmeyer

Roman Marktl

Vector Informatik GmbH
Ingersheimer Str. 24
DE-70499 Stuttgart
Tel.: +49-711-80670-400
Fax: +49-711-80670-555

Links
www.vector.com

This first part focuses on the basics of CAPL. It is primarily intended for those who are new to this language; however, it also offers a few insights for well-informed users into the motivation behind individual CAPL constructs. The second part will discuss advanced functionalities of CAPL. Finally, the third part will address performance and memory needs and offers tips and tricks on using databases and associative arrays.

For over 20 years – it was initially used in CANalyzer for DOS – CAPL has been used to implement a broad bandwidth of tasks: from simple stimuli to the simulation of complex bus nodes. In the following, CANoe is illustratively named for the two products CANoe and CANalyzer. The goal of CAPL has always been to solve specific tasks as simply as possible. Typical tasks are reacting to received messages, checking and setting signal values and sending messages. A program should restrict itself to precisely these things and not require any additional overhead.

Many programming tasks that CANoe users typically perform might actually be as brief and trivial as the example presented below – of course many other tasks are not so trivial. That is why CAPL has been continually extended over the years to be a tool that can also solve complex tasks according to the principle “as simple as possible”.

“CAPL” is an acronym for Communication Access Programming Language. The original focus on CAN has long been extended to all automotive bus systems such as LIN, Flexray, Most, J1587, as well as a few others like Arinc and CANopen.

As in many other languages, the syntax of CAPL is closely based on the syntax of the C language. Those who are familiar with C, C#, or various modern script languages will quickly be quite comfortable using CAPL. However, a few unique aspects distinguish a CAPL program from a C program:

CAPL programs are event-driven. This means that they consist of individual functions, each of which reacts to an event within the system under analysis: receipt of a message, change of a signal, expiration of a timer, or even a change in the environment. To react to the message “EngineState”, for example, you would use: “On message EngineState” (Figure 1).

CAPL programs use specific databases for the concepts of the system under analysis. Messages and signals get their names there, and these names can be used directly in the program code. In Figure 1, they are the names “EngineState” for a message and “EngineSpeed” for a signal in this message.

CAPL programs do not give the user any pointer types to use. Right from the outset this excludes numerous potential programming errors and causes of program crashes, such as those that frequently happen in C programming. Nonetheless, since pointers – aside from their susceptibility to errors – also represent a very powerful concept, CAPL provides a substitute for some things, e.g. associative arrays as a substitute for dynamic memory.

One important property that CAPL shares with C should be mentioned: CAPL is always compiled, i.e. it is converted to efficiently executable and flexible machine code.

Example: a simple CAPL program

In Figure 1 a simple CAPL program is presented, which ▷

CAPL

CAPL is a procedural programming language similar to C, which was developed by Vector Informatik. The execution of program blocks is controlled by events. CAPL programs are developed and compiled in a dedicated browser. This makes it possible to access all of the objects contained in the database (messages, signals, environment variables) as well as system variables. In addition, CAPL provides many predefined functions that support working with the CANoe and CANalyzer development, testing and simulation tools.

```
1.    variables
2.    {
3.      const long kOFF = 0;
4.      const long kON = 1;
5.    }
6.
7.    on message EngineState {
8.      @sysvar::Engine::EngineSpeedDspMeter = this.EngineSpeed / 1000.0;
9.    }
10.
11.   on message LightState {
12.     if (this.dir == RX) {
13.       SetLightDsp(this.HeadLight,this.FlashLight);
14.     } else {
15.       write("Error: LightState TX received by node %NODE_NAME%");
16.     }
17.   }
18.
19.   SetLightDsp (long headLight, long hazardFlasher) {
20.     long tmpLightDsp;
21.
22.     tmpLightDsp = 0;
23.     if(headLight == kON)
24.       tmpLightDsp = 4;
25.     if(hazardFlasher == kON)
26.       tmpLightDsp += 3;
27.     @sysvar::Lights::LightDisplay = tmpLightDsp;
28.   }
```

Figure 1: A simple example of a CAPL program

performs one of the basic tasks of a bus monitoring tool: it listens to traffic on the bus and prepares a couple of events on the bus for observation/monitoring by the user. This is a shortened, sample CANoe program: “Display.can” from the sample “Easy.cfg”. In the following, first the overall functionality is briefly summarized, and then the individual sections are described in more detail.

Task description

The task is to observe a CAN network whose elements – e.g. bus nodes, messages and transported signals – are described in a database. When the “EngineState” message is received, then the Engine-Speed signal it contains is conditioned for display on a display panel, and it is routed to the panel. When the “LightState” message is received, the “HeadLight” and “FlashLight” signals it contains are conditioned for display on a panel, and they are routed to the panel for graphic display.

Description of the program

The line numbers are not part of the CAPL program and are only inserted here to make it easier to reference individual lines or sections. To achieve the most compact representation possible, opening brackets were not placed on a separate line. In a CAPL program, it is possible to define global variables and constants. This is done in the “variables” section (lines 1 to 5). These constants and variables are globally defined for this program: they can be used anywhere in the program, but not in other programs within the same CANoe application. The other sections define reactions to events (lines 7 to 17) and an auxiliary function (lines 19 to 28).

Lines 7 to 9 show a minimal form of a message event procedure. This function is called whenever this message has been transmitted on the bus. In reference to CAN, this means that the precise time point is the TX or RX interrupt of the CAN controller, i.e. immediately after correct transmission of the message. The message that triggers the call of the particular function is referenced by “this” syntax.

In line 8, the value of the “EngineSpeed” signal is read out from the received message (“this”) and is assigned to a system variable with a conversion (/1000.0). Lines 11 to 17 show a message event procedure for the “LightState” message, which transmits the information for a turn signal flasher. Its processing is similar to that of the “EngineState” message with the following unique aspects: In line 12, the direction flag (.dir) is now checked in the message (“this”) that has been just transmitted. Only received messages should be considered in this program (value RX), because a message sent by the node itself would also trigger an event procedure (value TX). In this case, an error message would be output in line 15.

Since conditioning of the signal for display on the user interface (a panel on which different states are shown by different bitmaps) is somewhat more complex, its implementation is outsourced to a separate function: In line 13, “SetLightDsp” is called with the two message signals that are needed as parameters. Finally, lines 19 to 28 define a separate function, which writes different values to the system variable “LightDisplay” in the “Lights” name space according to the value of the transmitted signal. In this demo configuration, this variable then selects the appropriate bitmap on a display panel. ◀