Tips and tricks for the use of CAPL (part 2)

The first part of this series of articles addressed fundamental concepts of the CAPL programming language. This second part explains the time behavior of event procedures.

Authors

Marc Lobmeyer

Roman Marktl

Vector Informatik GmbH
Ingersheimer Str. 24
DE-70499 Stuttgart
Tel.: +49-711-80670-0
Fax: +49-711-80670-111

Link
www.vector.com

CAN Newsletter (print)
Tips and tricks for the use of CAPL (part 1)

The second part also offers tips for all types of users so that they can work more effectively with CAPL in the areas of "generic programming" and "conditional compiling."

Execution model

A key difference between CAPL and C or C++ relates to when and how program elements are called. In C, for example, all processing sequences begin with the central start function *main()*. In CAPL, on the other hand, a program contains an entire assortment of procedures of equal standing, each of which reacts to external events:

- Triggered by the system: These events include those that are useful for initializing and post-processing the measurement run: on *preStart, on start, on preStop* and *on stopMeasurement*, as well as the time control and keyboard events *on timer* and *on key*.
- Triggered by bus communication: There are many different types of event procedures that react to bus events such as those related to communication or error handling, and they are very dependent on the bus type. Examples of these are *on message* and *on busOff* in CAN and *on frFrame* and *on frStartCycle* in FlexRay.
- Triggered by access to a *Value Object*: Such objects include system and environment variables that are globally available in CANoe and CANalyzer as well as signal values that represent a data interpretation of the bus communication. Special databases perform the interpretation. Part 3 of this series will address this concept.

Event procedures are atomic: The simulation model of CANoe is event oriented. In event procedures, CANoe executes all actions simultane ously from the model perspective, namely at the point of in time of the triggering event. The actual computation time on a real PC is ignored.

Simulation time and time stamp: However, a real event generated by the PC, such as a bus output by *output()*, gets a time stamp of the real-time clock. The sequence and time points of these events can be influenced by bus protocols, driver, and hardware properties.

On a simulated bus, some of the mentioned influencing parameters are eliminated. In this case, bus events are initiated simultaneously; in the case of CAN, for example, this leads to a dependable arbitration of multiple messages that are output by *output()*.

Updating system variables: Users can also use CAPL to modify environment or system variables that are visible outside of the program. CAPL does not propagate value changes to a variable until after the current event processing is finished, but with the same time of the just handled event. A read access within the current procedure always returns the old value even if the variable appears to be set to a new value within the same procedure. The advantage is that only one value change occurs at a single point in time.

The execution model is situation dependent: There are many ways to use CAPL in CANoe and CANalyzer, and so the execution model varies somewhat, too: The simulation nodes of a CANoe simulation are in parallel on the bus. Hence, they are completely independent from each other. Triggered events are always dispatched to all programs. In contrast, nodes in the measurement setup and in CANalyzer are processed in sequential order: Each node passes its output to the next. Incoming events must be passed to the next node explicitly for further processing. The procedures *on ** and *on [*]* are provided for this purpose.

Another type of program is a test program whose test procedures can wait for external events. CAPL resumes execution with the simulation time of such events. In contrast, waiting in normal event procedures stalls the entire simulation system. This is a frequent source of errors when CAPL is used. It is therefore inadvisable to use a busy-wait or wait command in an external DLL.

Efficient programming in CAPL

The preprocessor is a powerful tool in the C language, ▷

but it can also lead to confusion and consequently to errors. Therefore, only a subset of the well-known preprocessor directives in C is offered in CAPL with comparable semantics.

#include: Include files contain arbitrary but complete sections of a CAPL program: *includes*, *variables* and *procedures*. In contrast to C, the text of an include file is not simply inserted into the CAPL file, rather the sections. All sections of the included file apply to the entire parent CAPL file "as if" they were contained in that file. The sequence of sections is irrelevant in CAPL anyways. This means the compiler reports any duplicate symbols as an error. Moreover, code and data from included and parent files may use each other mutually. One exception to the just stated prohibition of duplicate symbols is that *on start, on preStart, on preStop* and *on stopMeasurement* may coexist in both the included file and the parent file. In these functions, the code is executed sequentially: first the code from the included file and then the code from the parent file. This means that the Include files are used to perform three tasks: declare data types, define variables and provide an (inline) function library.

#pragma library: CAPL programs can use Windows DLLs created in other languages, as long as they implement a suitable CAPL DLL interface. These DLLs can be directly linked with the directive *#pragma library("capldll.dll")*.

Macros: In CAPL, there are a number of predefined macros that are available to users for use in the code or for conditional compiling. Macros for use in the code can be used anywhere in the code without restriction. In contrast to C, macros may be used freely within string constants, identifiers for variables, and function names. They always begin and end with a % character, and they are primarily used to write generic programs.

Available code macros include the node name, index of the current channel, name of the current network and the type of bus being used. The code can access the name of the containing file with *%FILE_NAME%*, or it can access the name of the program file currently being compiled with *%BASE_FILE_NAME%*. In the case of Include files, the latter is the parent file. Here are two simple examples:

```
write("The node name"
" is %NODE_NAME%");
@Ch%CHANNEL% = 1;
```

There is a separate set of predefined macros for the conditional compiling of code sections. They are *#if, #else, #elif* or *#endif.* Within a program, they allow distinguishing between the program types simulation node, measurement node and test program as well as the CANoe version that is used. Here is an example that uses a *#pragma message*:

```
#if (TOOL_MAJOR_VERSION
== 7 && TOOL_MINOR_VERSION
== 5 && TOOL_SERVICE_PACK
< 2) || CANALYZER
#pragma message("This
program needs at least
CANoe 7.5 SP 3")
#endif
```

#pragma message: The *#pragma* message directive lets users output their own message during the compiling process, e.g. the version number of the currently compiling CAPL program. It appears together with the other messages, warnings, errors, and general messages of the compiler. ◀