Tips and tricks for the use of CAPL (part 3)

The third and final part of this series presents tips and tricks for advanced users. Topics include associative arrays, performance, memory needs, and other database access options.

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)

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

Unlike languages such as C, CAPL does not support any pointer objects as a reference data type and therefore has no dynamic memory management. This makes CAPL very robust, and therefore well-suited for runtime environments that are short on memory and difficult to debug. In particular, CANoe's "CAPL-on-Board" feature benefits from this; in order to improve real-time behavior, it executes programs directly on certain hardware bus interfaces. Having said that, memory is seldom in short supply in the Windows' runtime environment. Therefore in this runtime environment CAPL offers associative arrays that can be used to store data even if the amount of data to be stored is unknown at the program start. Associative arrays are containers which are equivalent to maps or dynamic arrays of other programming languages. Internally, CAPL uses an efficient hash table for these arrays. Consequently, these special arrays enable saving bus messages or measurement values, even if it is unknown in advance which messages or how many measurement values will occur.

In CAPL, associative arrays are declared as simple arrays, but with a key type instead of the otherwise usual size entry. Two examples of associative arrays:

```
long lastTime [long];
char[30] translate[ char[] ];
```

The variable *lastTime* is an array that maps *long* keys to *long* values, while *translate* maps *string* keys (without length limitation!) to *string* values up to 30 characters. The following example uses *lastTime* to store a time value for each message ID occurring on the CAN network:

```
on message CAN1.*{
    lastTime [this.id]
      = this.time;
  }
```

To enhance the user's experience, CAPL provides the following list of methods for associative array variables using dot notation:

- *ContainsKey* queries whether a specific key is already contained;
- *Size* returns the number of contained keys;
- *Remove* removes one key from the associative array;
- *Clear* fully empties an associative array.

In fact, *Remove* and *Clear* free up memory.

Finally, there is a special form of the *for* instruction for associative arrays. This form iterates over all keys actually contained in *lastTime:*

```
for (long akey: lastTime)
  {…} …
```

Access to databases

Part 1 of this article series already illustrated the primary use of bus-specific databases in CAPL: they make it possible to introduce names for messages and signals. From a programming perspective, the complicated aspect of signals is that they are usually tightly packed in the data payload of messages for efficiency reasons. Therefore, signals generally exhibit arbitrary bit lengths and positions within the data payload of a message. They can also be stored in either Intel or Motorola format.

Symbol-based access via a signal name relieves the CAPL user of all of these details. In the case of reading or setting a signal value, the CAPL compiler automatically accounts for the signal's precise bit pattern that may include masking, swapping and shifting the bits.

To enhance user friendliness, other definable objects in the database may improve the linguistics of CAPL programming. For example, symbolic value tables may be associated with signals to use plain text names for signal value states. Furthermore, authors of a database have the freedom to define other attribute objects and to use them in the program code.

CAPL is able to use database objects directly based on their symbolic names. However, sometimes the potential objects of interest are not known at the time of program implementation. Therefore, the CAPL user may dynamically access the symbolic names and properties such as message names and identifiers transmitted by a network node. A brief example:

```
message * m;
int i, mx;
mx=elcount(aNet::aNode.Tx);
for (i = 0; i < mx; ++i)
{
  m.id=aNet::aNode.TX[i];
  write(DBLookup(m).Name);
}
```

These symbolic access methods allow the user to implement generic programs – together with the previously introduced associative arrays.

Performance

Most CAPL programs must meet non-trivial real-time conditions. The execution model of a node simulated with CAPL even follows the model concept that CAPL programs can be executed at any speed (see part 2 of this series of articles). To adequately approach this ideal, CAPL programs are compiled, i.e. they are compiled into the machine language of the specific executing microprocessor. Moreover, optimized code sequences are used for the often complex access to signals. Below are a few tips on how the user can affect performance.

writeEx(): the *write* function is used to output specific texts to the Write window in CANoe and CANalyzer. As an alternative, the *writeEx* function is available for outputting larger quantities of data. For one, it can be used to write directly to the Trace window or to a log file. The text output generated by *writeEx* is in all respects treated like a bus event, including the high priority processing and synchronizing the time stamps with real bus events.

Event procedures: a CAPL program consists of a combination of procedures that react to events. Some of these events may occur very frequently. Therefore, a program's performance is significantly better if only those events get processed, which are concerned. For example, if the user is only interested in those Flexray *slots* that contain a specific signal, it is more efficient to define `on frSlot signalname` than `on frSlot *`.

Signal edges: there are two event procedure versions for signals and system variables. *on signal_update* and *on sysvar_update* are called with each write access to the specific data objects, even if the object's value has not changed at all. By contrast, *on signal_change* (*on signal* in short) and *on sysvar_change* (*on sysvar* in short) offer a performance advantage if only signal edges are to be handled. Those event procedures are optimized to trigger on value changes only.

Memory needs

Unlike most block-oriented languages, such as C, all locally defined variables in CAPL are static by default. This means that they are all created at the program start, and memory used to store these variables is not freed until the end of the program. Consequently, CAPL may require a surprisingly large amount of memory if many event procedures define the same type of large variables, which they could actually share. An example:

```
testcase test789()
{
  char outBuffer[1024];
  [..]
```

There are CAPL programs with thousands of such test procedures, of which only one may be executed at any given time. Rather than defining a large local variable of the same type in each event procedure, defining the large variable once globally in the Variables section utilizes a lot less memory.

Another inadvisable practice is to create very large arrays, e.g. to store event data under the respective message IDs. An extended ID in CAN comprises 29 bits, so it can assume over 500 million values. To define an array for this purpose would be a waste of memory. In such cases, it is better to use associative arrays as described above. Although associative arrays need somewhat more memory for each key that is actually used, they do not need any memory for keys that are not used.

Useful, relatively unknown features

CAPL offers a number of less familiar and mainly newer features:

Structs can be used to define structures, similar to the approach in C. Together with copying operations, which can also convert Intel and Motorola formats within a *struct*, they represent a flexible method for data conversion.

When CAPL functions are called, the user has the option of passing reference parameters in addition to value parameters. Reference parameters make it possible to return more than one result value from a function. Reference parameters can also be used within CAPL–DLLs.

CAPL programs should also not crash in case of faulty usage. On one hand, this robustness is attained by the language structure, since there are no general pointers. On the other hand, stability is improved by automatic runtime checks of array limits, stack limits and the necessary computing time.

A separate command-line version of the compiler is available. This version is very helpful in automating sequences in script languages.

Concluding Remarks

This series of articles has introduced CAPL as an example of a problem-oriented programming language. The familiar C language syntax of CAPL simplifies the user's learning curve. Specific symbolic databases and concepts for using CAPL in simulation, emulation, and testing of fieldbus nodes support the application domains. Vector is carefully and continually extending the language in a way that maintains compatibility with previous versions while cultivating new application areas. ◀

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.