# Steuergerätetests effizienter programmieren – Basics, Tipps und Tricks beim Einsatz von CAPL

## Teil 3: CAPL für Fortgeschrittene

**Der dritte und letzte Teil dieser Artikelreihe stellt einige Tipps und Tricks für fortgeschrittene Anwender vor. Insbesondere behandelt er die in den vorhergehenden Teilen angekündigten Themen Assoziative Arrays, Performance, Speicherbedarf sowie weitere Möglichkeiten des Datenbankzugriffs.**

### Assoziative Arrays

CAPL kennt keine Pointertypen und damit auch keine dynamische Speicherverwaltung, wie sie in anderen Sprachen, beispielsweise C, üblich ist. Dadurch ist CAPL sehr robust und auch für Ablaufumgebungen geeignet, die knapp an Speicher und umständlich zu debuggen sind. Davon profitiert insbesondere das CANoe Feature „CAPL-on-Board", das die Programme zur Verbesserung des Echtzeitverhaltens direkt auf der Hardware bestimmter Bus-Interfaces ausführt. In der zumeist verwendeten Windows-Ablaufumgebung hingegen ist Speicher selten knapp. Darum können hier Daten, deren Umfang vor Programmstart nicht feststeht, mit Assoziativen Arrays gespeichert werden. Assoziative Arrays sind Container, die den Maps oder Dynamischen Arrays in anderen Programmiersprachen entsprechen. Intern verwendet CAPL für diese Arrays eine effiziente Hashtable. Diese speziellen Arrays ermöglichen damit das Speichern von Busbotschaften oder Messwerten, auch wenn vorher nicht bekannt ist, welche Botschaften oder wie viele Messwerte überhaupt auftreten.
Das Deklarieren von Assoziativen Arrays erfolgt in CAPL als einfache Arrays, allerdings mit einem Schlüsseltyp anstatt der sonst üblichen Größenangabe. Zwei Beispiele für Assoziative Arrays:

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

Der Bezeichner *lastTime* steht für ein Array, das *long*-Schlüssel auf *long*-Werte abbildet. Während *translate* für eines steht, das Zeichenkettenschlüssel (ohne Längenbegrenzung!) auf *char[30]*-Zeichenkettenwerte abbildet. Eine mögliche Anwendung von *lastTime* besteht darin, für jede auf dem Bus auftretende Botschafts-ID einen Wert abzulegen:

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

Für den komfortablen Umgang mit Assoziativen Arrays enthält CAPL noch weitere Funktionen:

- *containsKey* fragt ab, ob ein spezifischer Schlüssel bereits enthalten ist.
- *size* liefert die Anzahl der enthaltenen Schlüssel.
- *clear* leert ein Assoziatives Array komplett, während *remove* einen einzelnen Schlüssel entfernt. Beide Funktionen geben Speicher frei.

Zu guter Letzt existiert eine Spezialform der *for*-Anweisung für Assoziative Arrays. Diese Form iteriert über alle tatsächlich enthaltenen Schlüssel in *lastTime:*

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

### Zugriff auf Datenbanken

Teil 1 dieser Artikelreihe hat bereits die Hauptanwendung von busspezifischen Datenbanken in CAPL aufgezeigt: es werden Namen für Botschaften und Signale eingeführt. Das Komplizierte an Signalen aus Programmierersicht ist, dass sie in den Nutzdaten von Botschaften aus Effizienzgründen meistens recht eng gepackt sind. Deshalb weisen sie im Allgemeinen beliebige Bit-Längen und Offsets innerhalb der Nutzdaten auf. Zudem lassen sie sich im Intel- oder im Motorola-Format ablegen.
Der symbolbasierte Zugriff über den Namen entlastet den CAPL-Anwender von all diesen Details. Bei einem lesenden oder schreibenden Zugriff wird der nötige Code für den bit-genauen Zugriff, das Maskieren, das Drehen und das Schieben vom Compiler eingefügt.
Datenbanken enthalten noch weitere Objekte, die dem Anwender in CAPL-Programmen zur Verfügung stehen. Insbesondere lassen sich Signale symbolischen Wertetabellen zuordnen. Damit sind als Werte kodierte Zustände mit ihren Klarnamen nutzbar. Außerdem steht es dem Ersteller einer Datenbank völlig frei, eigene weitere Attribute zu definieren, die er dann im Programmcode verwenden kann.
Anhand der symbolischen Namen ist CAPL in der Lage, bekannte Datenbankobjekte direkt zu verwenden. Manchmal sind die potentiell interessierenden Objekte aber beim Erstellen des Programmes noch gar nicht bekannt. Deshalb kann CAPL zusätzlich zur Laufzeit auf die Gesamtheit aller Botschaften eines Knotens zugreifen. Außerdem kann es zu

beliebigen Botschaften und Signalen die Datenbankdefinitionen liefern. Ein kurzes Beispiel:

```
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);
}
```

Diese Zugriffstechniken erlauben es dem Anwender, zusammen mit den zuvor eingeführten Assoziativen Arrays, bei Bedarf generische Programme zu schreiben.

## Performance

Die meisten CAPL-Programme müssen nicht-triviale Echtzeitbedingungen einhalten. Das Ausführungsmodell eines mit CAPL simulierten Knotens verwendet sogar die Modellvorstellung, dass CAPL-Programme beliebig schnell ablaufen (siehe Teil 2 dieser Artikelreihe). Um diesem Idealbild ausreichend nahe zu kommen, werden CAPL-Programme kompiliert, also in die Maschinensprache des jeweils ausführenden Prozessors übersetzt. Außerdem werden für den oft komplexen Zugriff auf Signale optimierte Codesequenzen verwendet.
Nachfolgend gibt es ein paar Tipps, wie der Anwender die Performance beeinflussen kann.

### WriteEx()

Die Funktion *write* dient zur Ausgabe einzelner Meldungen in das Write-Fenster von CANoe und CANalyzer. Für das Ausgeben größerer Datenmengen steht als Alternative die Funktion *writeEx* zur Verfügung. Mit ihr kann unter anderem direkt in das Trace-Fenster oder in eine Logging-Datei geschrieben werden. Das Zeitverhalten von Meldungen zu diesen Zielen ist identisch zu dem Zeitverhalten von Busereignissen.

### Ereignisprozeduren

Ein CAPL-Programm besteht aus einer beliebigen Anzahl von Prozeduren, die auf Ereignisse reagieren. Einige dieser Ereignisse treten ausgesprochen häufig auf. Die Performance eines Programmes ist daher signifikant besser, wenn nur die jeweils interessierenden Ereignisse verarbeitet werden. Sollten zum Beispiel nur solche Slots interessieren, die ein bestimmtes Signal enthalten, ist es besser `on frSlot signalname` zu schreiben, als `on frSlot *`.

### Signalflanken

Für Signale und Systemvariablen existieren jeweils zwei Varianten für die Ereignisprozeduren. *on signal_update* und *on sysvar_update* werden bei jedem schreibendem Zugriff auf die jeweiligen Datenobjekte aufgerufen, auch dann, wenn sich der Wert dabei gar nicht ändert. Im Gegensatz dazu bringen *on signal_change* und *on sysvar_change* einen Performance-Vorteil, sofern nur Wertänderungen interessieren. Diese beiden Prozedurarten werden nur dann aufgerufen, wenn sich der jeweilige Wert auch tatsächlich geändert hat. Die Kurzformen *on signal* und *on sysvar* entsprechen dabei schon den optimierten Formen, die nur auf echte Flanken reagieren.

## Speicherbedarf

In CAPL sind alle Variablen statisch. Das heißt, sie werden alle bei Programmstart angelegt und der Speicherplatz erst bei Programmende wieder freigegeben. Das ist ein Unterschied zu den meisten blockorientierten Sprachen, wie beispielsweise C, bei denen lokale Variablen nur existieren, bis der sie enthaltene Block verlassen wird. Dadurch kann CAPL überraschend viel Speicher benötigen, wenn viele Prozeduren gleichartige große Variablen anlegen, die sie sich eigentlich teilen könnten. Ein Beispiel:

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

Es gibt CAPL-Programme mit tausenden solcher Testprozeduren, von denen stets nur eine auf einmal laufen kann. Ein Puffer, etwa um Meldungen zu formatieren, sollte in diesem Fall nur einmal global im variables-Abschnitt angelegt werden.
Eine andere nicht sinnvolle Verwendung ist das Anlegen von übergroßen Arrays, etwa um Daten von Ereignissen unter ihrer jeweiligen ID abzulegen. Eine extended ID bei CAN umfasst 29 Bit, kann also über 500 Millionen Werte annehmen. Hierfür ein fast leeres Array zu verwenden, ist Speicherverschwendung. Besser ist es, für solche Zwecke Assoziative Arrays wie weiter oben beschrieben zu verwenden. Sie benötigen zwar für jedes tatsächlich verwendete Element etwas mehr Speicher, brauchen dafür aber keinen Speicher für nicht verwendete Elemente.

## Nützliche, aber weniger bekannte Features

Den Abschluss dieser Artikelreihe über CAPL bildet diese kurze Aufzählung einiger weniger bekannter Features, die zumeist neueren Datums sind:
Mit *Structs* lassen sich ähnlich wie in C Strukturen definieren. Zusammen mit Kopieroperationen, die auch zwischen Intel- und Motorola-Format innerhalb einer *struct* umwandeln können, bilden sie eine flexible Möglichkeit zur Datenkonvertierung.
Beim Aufruf von CAPL-Funktionen hat der Anwender die Möglichkeit außer Werte-Parameter auch Referenz-Para-

meter zu übergeben. Wichtigste Anwendung ist die Rückgabe von mehr als einem Ergebniswert aus einer Funktion. Auch von CAPL-DLLs aus lassen sich Referenz-Parameter nutzen.
CAPL-Programme sollen auch bei fehlerhafter Anwendung nicht abstürzen. Diese Robustheit wird einerseits durch die Sprachstruktur erreicht, etwa indem es keine allgemeinen Pointer gibt. Andererseits verbessern automatische Laufzeittests von Array-Grenzen, Stack-Grenzen und benötigter Rechenzeit die Stabilität.
Es steht eine eigene Kommandozeilenversion des Compilers zur Verfügung. Diese Version ist sehr hilfreich zum Automatisieren von Abläufen in Skriptsprachen.

## Schlussbetrachtung

Die vorliegende 3-teilige Artikelreihe hat CAPL als ein Beispiel für eine anwendungsbezogene Programmiersprache vorgestellt. Die vertraute Syntax der Sprache C erleichtert den Einstieg in CAPL. Spezifische Symboldatenbanken und Konzepte für das Anwenden zur Simulation, Emulation und dem Test von Feldbusknoten unterstützen die Anwendungsdomäne. Das umsichtige, kontinuierliche Erweitern der Sprache durch Vector kombiniert Kompatibilität zu Vorgängerversionen mit der Öffnung für neue Anwendungsgebiete.

**Übersetzung der englischen Veröffentlichung im CAN Newsletter, Ausgabe 4/2014.**

**CAPL – „Communication Access Programming Language"**

CAPL ist eine von Vector Informatik entwickelte prozedurale, C-ähnliche Programmiersprache. Die Ausführung wird von Progammblöcken durch Ereignisse gesteuert. CAPL-Programme werden mit einem eigenen Browser entwickelt und kompiliert. Dabei kann auf alle in der Datenbasis enthaltenen Objekte (Botschaften, Signale, Umgebungsvariablen) und Systemvariablen zugegriffen werden. Darüber hinaus bietet CAPL eine Vielzahl von vordefinierten Funktionen, die das Arbeiten mit dem Entwicklungs-, Test- und Simulations-Werkzeug CANoe und CANalyzer unterstützen.

**Marc Lobmeyer (Dipl.-Inf.)**
arbeitet seit 1994 bei Vector Informatik als Entwickler an CANoe und CANalyzer.

**Roman Marktl (Dipl.-Ing)**
arbeitet seit 2012 bei Vector Informatik als Produktmanager im Bereich für CANoe und CANalyzer.