ECUテストのプログラミングをもっと効率的に ～ CAPLの基礎と使用上のヒント&コツ ～

パート3: 上級ユーザーのためのCAPL

本シリーズの最後となるパート3では、上級ユーザー向けのヒントとコツを紹介します。特にパート2で触れた、連想配列、パフォーマンス、必要なメモリ、データベースアクセスに関するオプションなどのトピックについて詳しくみていきます。

連想配列

Cなどの言語と違い、CAPLは参照データ型のポインタオブジェクトを一切サポートしないため、動的メモリ管理はありません。そのおかげでCAPLは非常に堅牢で、メモリが限られていてデバッグも難しいような実行環境には最適です。特に、CANoeの「CAPL-on-Board」機能にはこの特徴が有利に働きます。「CAPL-on-Board」はリアルタイム挙動を向上させるため、特定のハードウェアバスインターフェイス上で直接プログラムを実行する機能ですが、その際も、Windowsの実行環境でメモリが不足することはほとんどありません。ゆえにCAPLは、プログラム開始時にどれだけのデータ量が保存されるか分からなくても、データ保存のために連想配列を使うことができます。連想配列は、他のプログラミング言語のマップや動的配列に相当するコンテナです。内部ではCAPLがこれらの配列用に効率的なハッシュテーブルを用いているため、事前にどのメッセージあるいはどのくらいの数の測定値が生じるのかが不明であっても、これらの特別な配列によってバスメッセージや測定値の保存が可能になります。

CAPLでは連想配列を単純配列として宣言しますが、通常のようにサイズを指定するのではなく、キーの型を指定します。以下に連想配列の例を2つ示します。

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

変数lastTimeは、longキーをlong値にマッピングする配列です。一方、translateは、stringキー（長さ制限なし）を30文字までの文字列値にマッピングします。以下の例では、lastTimeを使用して、CANバス上で発生する個々のメッセージIDの時間値を保存しています。

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

ユーザーの操作性を高めるため、CAPLでは連想配列変数用にドット表記を用いた以下のメソッドが用意されています。

> ContainsKey：特定のキーがすでに含まれているかを照会する
> Size：含まれているキーの数を返す
> Remove：その連想配列から1つのキーを削除する
> Clear：連想配列を完全に空にする

なお、RemoveおよびClearによってメモリが解放されます。

最後に、連想配列に対する特殊な形式for命令を紹介します。この形式では、lastTimeに実際に含まれているすべてのキーで反復処理を行います。

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

データベースへのアクセス

本シリーズのパート1で、CAPLにおけるバス固有のデータベースの基本的な使用について解説しましたが、これらデータベースを使うことで、メッセージとシグナルを名前で処理することができます。たいてい、シグナルは効率性を考慮してメッセージのデータペイロードの中に隙間なく定義されているため、プログラミングの観点から扱いが難しいという面があります。一般的にシグナルは、メッセージのデータペイロード内の恣意的なビット長やビット位置で表され、シグナルはIntel形式あるいはMotorola形式のいずれかで定義されます。

シグナル名を用いたシンボルベースのアクセスにより、CAPLユーザーはこうした詳細を一切考慮せずに済みます。シグナル値の読出しや設定の際は、CAPLコンパイラーがビットのマスク、入れ替え、シフトなどのシグナルの正確なビットパターンを自動的に読み取ります。

他のオブジェクトを定義してデータベースに登録しておけば、CAPLプログラミングの作業が効率化し、ユーザーの使い勝手もよくなります。たとえば、シンボリックな値のテーブルをシグナル値と関連付けることにより、シグナル値のステータスを簡単なテキスト名を使って表現することが可能です（例：シグナル値’0’ = off, シグナル値’1’ = on）。さらに、データベースの作成者は、他の属性オブジェクトを自由に定義し、それらをプログラムのコード内で使用することができます。

CAPLでは、シンボル名に基づいてデータベースオブジェクトを直接使用することが可能です。ただし、プログラムを実装する時には、まだ使用したいオブジェクトが確定されていないこともあります。そのためCAPLでは、ネットワークノードから伝送されるメッセージ名や識別子などのシンボル名とプロパティーへ動的にアクセスできるようになっています。以下に簡単な例を示します。

```
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);
}
```

このシンボリックアクセスの手法と先に紹介した連想配列により、ユーザーは汎用的なプログラムを実装することができます。

パフォーマンス

大部分のCAPLプログラムは、複雑なリアルタイム条件を満たさなければなりません。CAPLでシミュレーションされるノードの実行モデルには、CAPLプログラムを任意の速度で実行できるというモデルコンセプトが採用されています（本シリーズのパート2を参照）。このような理想的な状態に近づくために、CAPLプログラムはそれを実行する特定のマイクロプロセッサーのマシン言語にコンパイルされます。さらに、複雑になりがちなシグナルへのアクセスには、最適化されたコードシーケンスが使用されます。

以下に、ユーザーがパフォーマンス向上のために利用できるヒントをいくつか紹介します。

writeEx()

write関数は、CANoeやCANalyzerの書込みWindowに特定のテキストを表示するために使われますが、通常のバスイベント処理と比べて表示処理の優先度が低いため、表示が遅れる場合があります。よって、大量のデータを遅延なく表示するためのwriteEx関数も用意されています。これはトレースWindowやログファイルに直接書き込む場合などに使用します。writeExで生成されるテキスト出力はバスイベントとまったく同様に扱われるため、高い優先度で処理され、実バスイベントのタイムスタンプと同期しています。

イベントプロシージャー

CAPLプログラムはイベントに反応するプロシージャーの組み合わせで構成されています。これらのイベントの中には、発生頻度が非常に高いものがあります。そのため、解析の対象となるイベントを適切に定義すれば、プログラムのパフォーマンスは大幅に向上します。たとえば、特定のシグナルが含まれるFlexRayのスロットのみが対象なのであれば、すべてのFlexRayスロットに

反応する “on frSlot *” と定義するよりも、特定のスロットとシグナルに反応する “on frSlot signalname” と定義した方がより効率的です。

シグナルの変化

イベントプロシージャーには、シグナル用とシステム変数用それぞれに対して、updateとchangeの2つのバージョンがあります。on signal_update と on sysvar_updateは、特定のデータオブジェクトへの書込みアクセスが発生するたび、たとえオブジェクトの値がまったく変化しない場合であっても、必ず呼び出されます。一方、on signal_change (on signalと略記) と on sysvar_change (on sysvarと略記) は、シグナルの変化のみに反応するため、シグナルの変化を扱う場合に高いパフォーマンス性を発揮します。つまりこのイベントプロシージャーは、値変更のトリガーとして最適です。

必要なメモリ

Cのようなたいていのブロック指向言語とは異なり、CAPLのローカル定義変数はすべて、デフォルトで静的変数となります。つまり、これらはすべてプログラムの開始時に作成され、その変数の保存に使われるメモリはプログラムが終了するまで解放されません。そのため、仮に多数のイベントプロシージャーが共有できるにもかかわらず、個別で同じタイプの大容量の変数を定義した場合、そのCAPLに大量のメモリが必要になることもあります。以下に例を示します。

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

こうしたテストプロシージャーを何千も備えたCAPLプログラムもありますが、実行されるのは一度に1つのみです。イベントプロシージャーごとに同じタイプの大容量のローカル変数を定義するよりも、その変数をグローバル変数としてVariablesセクションで1回だけ定義した方が、使用するメモリを大幅に削減することができます。

メッセージIDごとにイベントデータを保存するなど、非常にサイズの大きい配列を作ることは避けた方がいいでしょう。CANの拡張IDは29ビットであるため、その配列が保存する値の数は5億を超えることもあり、そのような配列を定義するのはメモリの無駄遣いです。このような場合は、先に述べた連想配列を使うようにしてください。連想配列の場合、実際に使用される各キーにはメモリが必要となりますが、使用されないキーにはメモリは不要です。

知っておくと便利な機能

本シリーズの締めくくりに、知っておくと便利な機能についていくつか紹介します。

> structを使用して、C言語に似たアプローチでストラクチャーを定義できます。コピー操作を行うと、struct内でIntel形式とMotorola形式の変換も行われます。この方法を使えば、柔軟なデータ変換が可能です。

> CAPL関数を呼び出す際、ユーザーはオプションで値パラメーターのほかに参照パラメーターを渡すことができます。参照パラメーターを使用すれば、1つの関数から複数の結果を返すことが可能になります。参照パラメーターはCAPL DLL内でも使用できます。

> CAPLプログラムは、万一使用を誤ってもクラッシュしません。汎用的なポインタを持たないという言語構造により、このような堅牢性が確保されています。また、配列の上限、スタックの上限、必要な処理時間などが実行時に自動チェックされることによって、より高い安定性が確保されています。

> 独立したコマンドライン版のコンパイラーが用意されています。このコマンドライン版は、スクリプト言語でシーケンスを自動化する際に非常に便利です。

最後に

本シリーズでは、問題指向のプログラミング言語の例として、CAPLを紹介してきました。CAPLのCライクな構文は、なじみやすく習得しやすい言語です。CAPL特有のシンボリックなデータベースやコンセプトによって、フィールドバスノードのシミュレーション、エミュレーション、テストを効率的に実行し、アプリケーションの開発をサポートします。ベクターは、以前のバージョンとの互換性を保ちつつ、新しいアプリケーション分野が開拓できるよう、今後もCAPLを慎重かつ継続的に拡張していきます。

CAPLとは？

CAPLはVector Informatik（ベクター本社）によって開発された、Cライクな手続き型プログラミング言語です。プログラムブロックの実行はイベントによって制御されます。CAPLプログラムは専用のブラウザーで開発およびコンパイルされます。これにより、システム変数をはじめ、データベースに含まれているあらゆるオブジェクト (メッセージ、シグナル、環境変数) へのアクセスが可能になります。さらに、CAPLは事前定義された関数を数多く用意しており、開発/テスト/シミュレーションツール「CANoe」や「CANalyzer」に活用できます。

本稿は、2014年4月CiA発行の『CAN Newsletter』に掲載されたベクター執筆による記事内容を和訳したものです。

執筆者：

Marc Lobmeyer (Dipl.-Inf.)
1994年よりCANoeおよびCANalyzerの開発者としてVector Informatikに勤務。

Roman Marktl (Dipl.-Ing)
2012年よりCANoeおよびCANalyzerのCAPL機能関連のプロダクトマネージャーとしてVector Informatikに勤務。

■ 本件に関するお問い合わせ先
ベクター・ジャパン株式会社
営業部
（東京）　TEL：03-5769-6980　FAX：03-5769-6975
（名古屋）TEL：052-238-5020　FAX：052-238-5077
E-Mail：sales@jp.vector.com