# 組込みコンポーネントシステム TECS 初めの一歩

大山 博司

2017 年 8 月 15 日

# 目 次

# 組込みコンポーネントシステム TECS 初めの一歩

著者者 大山 博司

バージョン： 非常に早期のバージョン

## 0.1 はじめに

# 第1章　TECS 概観

## 1.1　組込みコンポーネントシステム TECS 仕様

TECS は、TOPPERS プロジェクトにより開発された、組込みソフトウェアのためのコンポーネントシステムです。TECS という名前は、 TOPPERS Embedded Component Systemm の頭文字から命名されました。

さて、コンポーネントシステムとは、何でしょうか？

実際のところ、コンポーネントシステムに、明確な定義はないはずですが、ソフトウェアをコンポーネント（部品）化し、部品を組合わせてアプリケーションソフトウェアを構築するための一連の仕様、言語、ツール類、ライブラリを総称するものになります。

TECS では、以下のような一連の仕様やツールからなるシステムです。

- TECS コンポーネント仕様

・TECS コンポーネントモデル
・TECS コンポーネント図
・コンポーネント記述言語 TECS CDL
・TECS コンポーネント実装モデル

- TECS ジェネレータ
- 各種プラグイン
- TECS コンポーネント図エディタ
- TOPPERS/ASP などのカーネルオブジェクトの TECS コンポーネント

TECS は、特に TOPPERS/ASP などの RTOS に最適なように設計されています。省メモリ、低オーバーヘッドは、組込みシステムにとって重要なキーワードです。このために TECS では、コンポーネントの静的な生成と結合を基本としています。このことは、RTOS のカーネルオブジェクトのように比較的粒度の小さなものでさへ、躊躇なくコンポーネントとして扱うことができますし、TOPPERS/ASP などの静的なコンフィグレーションに非常に適したものとなっています。

TECS では、コンポーネント記述言語 TECS CDL により、コンポーネントの構成や、コンポーネントを組合わせてアプリケーションを組上げる記述

を行いますが、コンポーネントの振る舞いには C 言語を用います。本書では、C 言語のプログラミングの仕方までは説明しませんので、C 言語をある程度理解している必要があります。初級レベルでも十分に TECS を理解し、使用できますが、プリプロセッサの働きを理解していると、TECS が内部で何をしているか、理解しやすいはずです。

それでは TECS が何であるか、何ができそうか、まずは TECS 版 Hello World! で見てみましょう。その前に、TECS のツール類をインストールします。

## 1.2　インストール

今回は、POSIX 環境用のアプリケーションを簡単に作成するところから始めてみます。POSIX 環境とは UNIX 系の API を備えた環境のことです。Windows であれば cygwin がよいでしょう。Linux は、もともと POSIX 環境ですし、MacOS もそうです。

POSIX 環境のアプリケーションをビルドするのには、TECS ジェネレータ個別パッケージが必要となります。TECS ジェネレータ個別パッケージは、以下の URL からダウンロードすることができます。

```
https://www.toppers.jp/tecs.html#d-package
```

最新バージョンをダウンロードしてください。今 (2017 年 7 月) ですと tecsgen-1.5.0.tgz をダウンロードできます。パッケージを展開したいディレクトリへ移してから、以下のコマンドで展開します。以下の内、' プロンプトは、環境ごとに相違する可能性があります。(私が使っている環境では '$' になっています)

```
% tar xvzf tecsgen-1.5.0.tgz
```

さて、以下のものも必要になりますので、手元になければ、併せて準備してください。

- Ruby V2.0 以降
- GNU make
- gcc C コンパイラ

tecsgen コマンドへのパスを通しておく必要があります。PATH 環境変数を設定します。B シェル系の場合、展開したディレクトリ直下にある set_env.sh をシェルに読み込むことで、一時的に設定することができます。

```
% cd tecsgen
% source set_env.sh
```

上記のコマンドは、必ず set_evn.sh のあるディレクトリへ移ってから実行する必要があります。set_env.sh への相対パス指定では、正しく設定されません。

## 1.3 TECS 版 Hello World!

TECS は、本来組込みシステムのためのものですが、まずは cygwin, Linux, MacOS などの POSIX 環境で動作する簡単なアプリケーションを作ってみましょう。

TECS の場合、始めに CDL ファイルを作成する必要があります。CDL ファイルは、コンポーネント記述を格納するもので、TECS CDL 言語を用いて記述します。拡張子は .cdl とします。

ここでは、全体を概観するために、CDL ファイルの具体例を示します。この内容が何を意味するかは、後から見ることにします。

```
--- HelloWorld.cdl の内容 ---
/* POSIX 用の簡単なテスト環境のコンポーネント記述を取り込む */
import( <cygwin_kernel.cdl> );

/* tHelloWorld セルタイプ（コンポーネントの型）*/
celltype tHelloWorld {
  entry sTaskBody eMain;
};

/* HelloWorld セル（コンポーネントインスタンス）*/
cell tHelloWorld HelloWorld {
};

/* Task1 セル（コンポーネントインスタンス）*/
cell tTask Task1 {
  cBody = HelloWorld.eMain;
  priority = 11;      /* この値は使われていない */
  stackSize = 4096;  /* この値は使われていない */
  taskAttribute = C_EXP( "TA_ACT" );
;
--- ここまで HelloWorld.cdl ----
```

上記の内容のファイルを、以下のディレクトリ位置に作成します。

```
% mkdir HelloWorld
```

```
% ls
doc/  glade/   HelloWorld/  Makefile  MANIFEST  README.txt
README-eng.txt samples/  set_env.bat  set_env.sh  tecs/
tecsgen/  test/
% cd HelloWorld
% EDIT HelloWorld.cdl
```

EDIT は、適当なテキストエディタを用います。必ずしもコマンドラインから起動する必要はありませんが、上記と同じ位置に　HelloWorld.cdl が作られるようにします。なお、文字コードは UTF-8 で保存するのが最適です（他の文字コードでもビルドできますが、文字コードが混じることになるかもしれません）。

さて、ここまで記述したら、TECS ジェネレータ tecsgen を使って、テンプレートファイルを作成します。

```
% tecsgen -I ../test/cygwin HelloWorld.cdl
```

オプション -I で cygwin ディレクトリを指定しています。もし ../test/cygwin が無いようでしたら、想定しているディレクトリとは異なるディレクトリに置かれています。HelloWorld ディレクトリまたは、-I の引数を調整してください。

```
% mkdir src
% tecsmerge gen/tHelloWorld_templ.c src
```

src ディレクトリの下に tHelloWorld.c というファイルが作られたはずです。ここで、tHelloWorld_templ.c をテンプレート (ファイル)、tHelloWorld.c をセルタイプコード (ファイル) と呼びます。

次に、セルタイプコードを編集しプログラムを完成させます。以下に変更箇所を中心に示します。

```
--- tHelloWorld.c の内容 ---
...

/* プロトタイプ宣言や変数の定義をここに書きます #_PAC_# */
#include  <stdio.h>                    <<<  追加


...

void
eMain_main(CELLIDX idx)
{
```

```
    CELLCB  *p_cellcb;
    if (VALID_IDX(idx)) {
      p_cellcb = GET_CELLCB(idx);
    }
    else {
      /* エラー処理コードをここに記述します */
    } /* end if VALID_IDX(idx) */

    /* ここに処理本体を記述します #_TEFB_# */
    printf( "Hello World!\n" );        <<<  追加
  }
  --- ここまで tHelloWorld.c ---
```

これは C 言語ですから、すでにおなじみのコードのはずです。ただ、関数名が、少しわからないですよね。これも後から見ることにします。

次に Makefile を用意します。TECS ジェネレータ tecsgen は Makefile のテンプレートも生成します。Makefile の場合は、以下のように mv コマンドで移動するだけで Makefile とすることができます。

```
% mv gen/Makefile.templ Makefile
```

この Makefile は、GNU make と gcc を用いるのであれば、無変更でビルドできます。

```
% make
```

これにより HelloWorld.exe ができあがります。拡張子 .exe は、Linux 等では不要ですが、ついてきます。コマンドを起動する際には、注意してください。

それでは HelloWorld.exe を実行してみましょう。

```
% ./HelloWorld.exe
Hello World!
*** starting task 'tTask_Task1'  1004010E0
Hello World!
*** exiting task 'tTask_Task1'
```

Hello World! と表示されましたね！

*** で始まる行は、タスクの開始と終了に出力されるメッセージです。とりあえず、無視してください。

さて、それでは、CDL ファイルとセルタイプコードの記述内容について見ていきましょう。

## 1.4　CDL ファイル HelloWorld.cdl

ここでは CDL ファイル HelloWorld.cdl の内容を要素ごとに見ていきます。

### 1.4.1　インポート

初めに CDL ファイルをインポートします。

```
/* POSIX 用の簡単なテスト環境のコンポーネント記述を取り込む */
import( <cygwin_kernel.cdl> );
```

cygwin_kernel.cdl は、(cygwin 用に見えますが) POSIX 環境で簡単なテストコードを実行してみることのできる環境を提供するものです。TOPPERS/ASP 用の kernel.cdl を元に、簡略化して作られています。HelloWorld の例など、単純な構成のプログラムを実行するのには向きますが、RTOS をエミュレートできるほどの機能は、持っていません。

インポート文は以下のように書くこともできます。

```
import( "cygwin_kernel.cdl" );
```

この違いは C 言語の #include に似ていますが、TECS の場合は、¡¿ で囲んだ場合、再利用される記述であることを表します。このことはセルタイプコードは、すでに記述済みであることを意味します。従って TECS ジェネレータは、¡¿ で囲まれてインポートされた CDL ファイル内のセルタイプに対してテンプレートファイルを生成しません。その差だけです。

### 1.4.2　tHelloWorld セルタイプ

次は、 tHelloWorld セルタイプの定義です。

```
/* tHelloWorld セルタイプ（コンポーネントの型）*/
celltype tHelloWorld {
  entry sTaskBody eMain;
};
```

セルタイプとは、コンポーネントの型のことです。型を定義しているだけですから、これを記述しただけでは、コンポーネントのインスタンスは生成されません。コンポーネントのインスタンスは、次の項で説明します。

さて、celltype はキーワードで、これからセルタイプを定義することを表します。

次の語 tHelloWorld は、セルタイプ名です。HelloWolrd を実現するものですので、tHelloWorld と命名しました。先頭の 't' はセルタイプの接頭辞で

す。TECS CDL の文法としては、't' で始める必要はありませんが、慣習としてセルタイプ名は 't' で始めます。

' '、'' で囲んで、セルタイプ定義の本体を記述します。

entry は、キーワードで、これから受け口の定義をすることを表します。

次の語、sTaskBody はシグニチャ名です。sTaskBody は、cygwin_kernel.cdl の中で定義されています。これは、タスクのメインのためのシグニチャで、TOPPERS/ASP のものと同じです。シグニチャは、コンポーネント間のインタフェースの型のことで、関数ヘッダの集合から成ります。sTaskBody が、どのような関数から成るかは、後から見ることにします。

さらにその次の語　eMain は受け口名です。受け口とは、コンポーネントの機能を提供するための口（ポート）で、シグニチャで定義されている関数を実装します。これを受け口関数と呼びます。

セルタイプは受け口の他に、呼び口、属性、変数、ファクトリを持つことができますが、セルタイプ tHelloWorld は単純なコンポーネントですので、一つの受け口だけを持ちます。

本体の後ろに ';' を記述して、セルタイプの定義を終わります。

### 1.4.3 HelloWorld セル

次は HelloWorld セルの定義です。

```
/* HelloWorld セル（コンポーネントインスタンス）*/
cell tHelloWorld HelloWorld {
};
```

セルとは、コンポーネントのインスタンスのことです。セルはあるセルタイプを元に生成されたコンポーネントの実体になります。

さて、cell はキーワードで、これからセルを定義することを表します。

次の語 tHelloWorld はセルタイプ名で、2) で定義されたセルタイプを参照しています。

次の語 HelloWorld がセルの名前です。セルの名前にはセルタイプのように既定の接頭字はありませんが、慣習として大文字で始めます。名前は、アルファベットまたは '_' で始まり、任意個のアルファベット、数字、'_' を続けます。これは C 言語の識別子と同等です。

続いて '', '' で囲んでセルの本体を記述します。本体には属性の値や、呼び口の結合先を記述します。セル HelloWorld の属するセルタイプ tHelloWorld には、属性も呼び口もありませんから、HelloWorld の本体として記述する物はありません。

本体の後ろに ';' を記述して、セルの定義を終わります。

### 1.4.4 Task1 セル

最後は、Task1 セルの定義です。

```
/* Task1 セル（コンポーネントインスタンス）*/
cell tTask Task1 {
  cBody = HelloWorld.eMain;
  priority = 11;      /* この値は使われていない */
  stackSize = 4096;  /* この値は使われていない */
  taskAttribute = C_EXP( "TA_ACT" );
};
```

セル Task1 は、セルタイプ tTask1 に属します。cygwin_kernel.cdl の tTask の定義を見ると、受け口 (entry)、呼び口 (call)、属性 (attr)、変数 (var)、セルタイプファクトリ (FACTORY) が記述されています。このうちセルの本体には、呼び口と属性の値を記述します。以下のは cygwin_kernel.cdl から、tTask のコードを抜き出したものです。受け口、変数、セルタイプファクトリの記述は省略しています。

```
[active]
celltype tTask {
              call  sTaskBody  cBody;  /* タスク本体 */
  [optional] call  sTaskExceptionBody  cExceptionBody;
                /* タスク例外処理ルーチン本体 */
  attr{
    /*
     *  TA_NULL     0x00U   デフォルト値
     *  TA_ACT      0x01U   タスクの生成時にタスクを起動する
     */
    ATR    taskAttribute = C_EXP("TA_NULL");
    /*
     * タスク例外処理ルーチンに指定できる属性はないため
     * TA_NULL を指定する
     */
    ATR    exceptionAttribute = C_EXP("TA_NULL");
    PRI    priority;
    SIZE   stackSize;
    char_t *name = C_EXP( "\"$id$\"" );
  };
  // entry, var, FACTORY は省略
};
```

それでは、Task1 の本体を見ていきましょう。

cBody は呼び口の名前です。呼び口とは、受け口へ結合するためのインタフェースです。'=' の右辺に結合先、すなわちセルの受け口を記述します。セルの受け口は、セル名とセルの受け口を '.' でつないで記述します。ここでは HelloWorld セルの受け口 eMain へ結合しています。これで、タスクが起動されると、HelloWorld　セルの受け口関数を呼びだすことができるようになります。受け口関数が何であるかは、セルタイプコードのところで説明することにして、先へ進みます。

次は属性を定義します（この順序は、自由です)。属性は、セルに初期値を与えるものです。

priority は属性で、タスクの優先度を指定します。TOPPERS/ASP 版の tTask では有効ですが、cygwin 版ではダミー記述となります。stackSize はスタックサイズを指定しますが、cygwin 版ではやはりダミーです。

taskAttribute は、タスクの属性を指定するものです。ここでは C_EXP( "TA_ACT" ) を指定しています。C_EXP は、C 言語のヘッダファイルの中で #define により定義されたマクロを参照する場合に用います。taskAttribute は C 言語のコンパイル時に TA_ACT に初期化されます。import_C でインポートしたヘッダから TECS ジェネレータは typedef と struct のみ取り込みます。C 言語のプリプロセッサは import_C が読み込まれるごとに、別に起動されます。このためマクロ定義 #define は、各 import_C で取り込むヘッダ内でのみ有効です。

さて、cygwin_kernel.cdl の tTask の定義を見てみると、属性には上記の他に exceptionAttribute と name があります。なぜダミーであるにも関わらず priority と stackSize が記述されていて、exceptionAttribute と name は記述されなったのでしょうか？

これはセルタイプ tTask の定義で属性に初期値が与えられているかどうかで、セルの定義で記述する必要性が決まります。tTask の定義で priority と stackSize には初期値が与えられていません。このような属性は、セルの定義時に何らかの初期値を与えなくてはなりません。一方、exceptionAttribute と name には初期値が与えられています。このような属性は、この値でよければ、セルの定義時に初期値を与える必要はありません。

属性 taskAttribute には tTask で初期値が与えられていますが、セル Task1 でも初期値が与えられています。この場合、セルで定義した値で上書きされます。TA_ACT で、起動時にタスクを起動することを指定します。C_EXP は先にも説明したように、#define で定義した値を参照するのに用います。

セルの本体の後ろに ';' を記述して Task1 の記述は終わりです。

以上で CDL ファイルの記述は終わりです。

## 1.5　HelloWorld のコンポーネント図

HelloWorld.cdl の内容をコンポーネント図に表すと、図 1.1 のようになります。


図 1.1: HelloWorld の TECS コンポーネント図

TECS のコンポーネント図は、セルのみを表現します。その意味で UML におけるインスタンス図に近いものになります。しかしながら、TECS のコンポーネント図は、より客観性の高いものになっています。

実際、このコンポーネント図は HelloWorld.cdl を TECS コンポーネント図エディタ (tecscde) に入力して、逆に作成しています。CDL ファイルを直接編集した場合であっても、コンポーネント図へのフィードバックが容易であることも TECS の強みの一つです。

## 1.6　セルタイプコード tHelloWorld.c

次にセルタイプコードを記述します。セルタイプコードは、セルタイプごとに分けて記述します。ファイル名は、セルタイプ名とすることが決められています。

テンプレートで、大部分が出力されているので、書き加える必要のある部分だけを記述します。

### 1.6.1　ヘッダのインクルード

必要であれば #include によりヘッダファイルをインクルードすることを指定します。

```
/* プロトタイプ宣言や変数の定義をここに書きます #_PAC_# */
#include  <stdio.h>                <<<<  追加
```

今回は printf を用いるため stdio.h をインクルードしています。

### 1.6.2 受け口関数

次に受け口関数を記述します。受け口関数とは、セルが機能を提供するものであり、セルタイプごとに振る舞いを記述するものです。受け口関数の名前は、受け口名とシグニチャの関数名を '_' で連結したものになります。ですので、以下は受け口 eMain、シグニチャ sTaskBody の唯一の関数 main の受け口関数です。

```
void
eMain_main(CELLIDX idx)
{
  CELLCB   *p_cellcb;
  if (VALID_IDX(idx)) {
    p_cellcb = GET_CELLCB(idx);
  }
  else {
    /* エラー処理コードをここに記述します */
  } /* end if VALID_IDX(idx) */

  /* ここに処理本体を記述します #_TEFB_# */
  printf( "Hello World!\n" );        <<<<  追加
}
```

テンプレートで大部分が出力されているので、追加の行だけ記述すればよくなっています。CDL ファイルを記述するのは、面倒ですが、ようやくここで少し楽ができましたね！

### 1.6.3 結合の内側

おさらいになりますが、結合とは、セルの呼び口を受け口につなぐことです。この項では、結合が、どのように行われるのかを見ていきます。

さて、前項までで、記述すべきソースを一通り見てきました。どうしてこれだけの記述で、eMain_main が呼び出されるのか、疑問を感じませんでしたか？隠されたコードがないのが C 言語のよいところ、なのに TECS では何かが隠されていて気持ち悪い、きっと C 言語に精通した貴兄、貴女はそう思われたはずですね。でも、そんなことはありません。TECS はすべてを、さらけ出す… いやいや、そうなんだけれど、「システムの下に何も隠さない」が一つの開発目標となっています。ですから、何かが隠されているというようなことはありません。

さて、少しだけ種明かしをしましょう。でも、C 言語にそこまで精通していないという方は、とりあえず読み飛ばしていただいて構いません。しかし

ながら、デバッグする時には、多少の知識が必要になるでしょう。その時に、振り返ってもらってもよいでしょう。

HelloWorld.cdl の Task1 セルのところで、cBody = HelloWorld.eMain と記述しましたね。TECS ジェネレータは、この記述をもとに、グルーコードを生成します。グルーとは接着剤のこと。つまり、グルーコードとは、呼び口と受け口をつなぎ合わせるコードのことです。このコードは、呼び口側で関数ごとに生成されます（結合関係が複雑なときは、受け口側にも生成されます)。

では、具体的に見てみましょう。呼び口側ですから Task1 の側を見る必要があります。実際には、セルタイプごとに生成されていますから、tTask に対してみる必要があります。

セルタイプコード tTask.c は cygwin_kernel.cdl と同じディレクトリに置かれていますが、ここを見てもグルーコードはありません。それでは、どこにあるかと言えば、TECS ジェネレータが生成した gen ディレクトリ下の tTask_tecsgen.h にあります。120 行目あたりにある、以下のような記述です。

```
/* 呼び口関数マクロ #_CPM_# */
#define tTask_cBody_main( p_that ) \
    tHelloWorld_eMain_main( \
     (tHelloWorld_IDX)0 )
```

これがグルーコードになります。でも、名前が少し変わっていますね。

受け口関数は eMain_main と記述しましたが、ここでは tHelloWorld_eMain_main になっています。頭にセルタイプ名 tHelloWorld がくっついています。この名前 (tHelloWorld_eMain_main) のことをグローバル名と呼びます。このように名前を付けることで、C 言語レベルで名前衝突が起きないようにしています。

もうお気づきかと思いますが、eMain_main を tHelloWorld_eMain_main に置き換える #define が別に定義されています。これは、tHelloWorld_tecsgen.h の方に含まれています。eMain_main を短縮名と呼びます。短縮名は、往々にして衝突しえます。このため、セルタイプコードはセルタイプごとに分けて記述します。

ところで TECS ジェネレータは、呼び口と受け口ずいぶんと単純なことをしていると思われましたか？tTask_cBody_main を tHelloWorld_eMain_main に置き換えているだけのように見受けられます。しかし、実際には、このような単純な場合ばかりとは、限りません。TECS ジェネレータは、結合状況に応じて、生成コードを変化させます。今回は、tTask と tHelloWorld のセルが一対一でしたから、このように単純になりましたが、多対多の関係になると相違が生じます。このことは、次の機会に見ることとします。

## 1.7 出力先を別のコンポーネントに分離

前節で使用した HelloWorld をフォルダごとコピーして HelloWorld2 としてます。まずは、HelloWorld2 フォルダにある HelloWorld.cdl を編集するところから始めます。

```
% cp -pr HelloWorld HelloWorld2
% cd HelloWorld2
% make clean
```

最後の make clean は、中間生成物を含めビルドされたファイルを消すためのものです。これからの作業で、手順誤りに気付きにくくなりますから、きれいにしておきましょう。

### 1.7.1 出力先を別のコンポーネントに分離の方針

HelloWorld の例では、printf を使用して文字列を出力しました。しかし、これが使えるのは、組込みシステムとしては少々リッチな環境かもしれません。TOPPERS/ASP では syslog であれば、機能削除していない限りは、使えるようになっています。

まずは tHelloWorld の出力部分を別のコンポーネントに分けてみます。以下の手順で行います。

1. シグニチャ (コンポーネント間のインタフェース) を決める
2. セルタイプ (コンポーネントの型) を決める
3. 組上げ (セル) 記述を書く

通常の開発では、1.、 2.、3. が同時並行で考えることになると思います。これらは、コンポーネントで役割をどのように分担するか、コンポーネント間のインタフェースをどうするするか、で決まってきます。しかしながら、書きおろす段階では、上記の順番で記述することになります。ここでは、上記の 3 つのステップを sPutString シグニチャ、tPutStringStdio セルタイプ、PutStringStdio セルの追加として説明します。

### 1.7.2 sPutString シグニチャ

まずは、シグニチャ、すなわちコンポーネント間のインタフェースを決めましょう。シグニチャの名前は sPutString とします。tHelloWorld.c では C 言語の入門書にならって printf を用いましたが、今回は単に文字列を出力するだけの機能としますので、このような名前としました。

```
/* sPutString シグニチャ（インタフェースの型）*/
signature sPutString {
  void  putString( [in,string]const char_t *str );
};
```

signature は、これからシグニチャを定義するというキーワードです。sPutString がシグニチャの名前です。

’’, ’’ で囲んで、関数頭部を記述します。関数頭部は、C 言語のプロトタイプ宣言に似ていますが、以下の点が異なります。

a) ’[’, ’]’ で囲んで引数の特性を指定する

b) 仮引数を省略できない

c) 引数が無い場合は、void 型を指定する (仮引数は不要です)

d) 引数が in かつポインタ型の場合、参照する型は、必ず const 修飾する

’[’, ’]’ の部分を取り除くと C 言語のプロトタイプ宣言になります。b), c), d) は C 言語では、必須ではないものが、TECS では必須となるものです。

第一引数 str には d) の理由により const で修飾されています。in 指定子から、この引数が入力引数であることを示します。入力引数とは、呼び元から呼び先へ値が渡されることを意味します。string 指定子は、str が文字列で、NULL 終端されていることを表しています。

最後に ’;’ を記述して、シグニチャの定義を終わります。

### 1.7.3　tPutStringStdio セルタイプ

tPutStringStdio は標準入出力に文字列を出力するセルタイプです。

```
/* tPutStringStdio セルタイプ */
celltype tPutStringStdio {
  entry  sPutString ePutString;
};
```

受け口 ePutString を持ち、そのシグニチャは sPutString です。

### 1.7.4　PutStringStdio セルの追加

PutStringStdio は標準入出力に文字列を出力するセルであり、tPutStringStdio セルタイプに属します。

```
/* PutStringStdio セル */
cell tPutStringStdio PutStringStdio {
};
```

### 1.7.5 tHelloWorld セルタイプ、HelloWorld セルの変更

tHelloWorld セルタイプに呼び口 cPutString を設けます。シグニチャは、sPutString です。出力先をこの呼び口に結合されたセルとします。

```
/* tHelloWorld セルタイプ */
celltype tHelloWorld {
  entry sTaskBody  eMain;
  call  sPutString cPutString;
};
```

HelloWorld セルの、呼び口 cPutString をセル PutStringStdio の受け口 ePutString に結合します。

```
/* HelloWorld セル */
cell tHelloWorld HelloWorld {
  cPutString = PutStringStdio.ePutString;
};
```

ここまでで、セルを準備できました。PutStringStdio を追加した後のコンポーネント図を図 1.2 に示します。

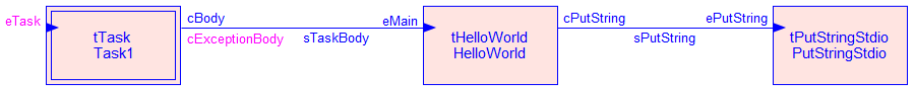


図 1.2: HelloWorld の TECS コンポーネント図

### 1.7.6 セルタイプコードの更新

まずは tHelloWorld.c をテンプレートをマージして更新しましょう。そのためには TECS ジェネレータを起動します。2 通りあります。
まずは、以前と同じように tecsgen を起動する方法です。

```
% tecsgen -I ../test/cygwin HelloWorld.cdl
```

もう一つは Makefile を使用する方法です。

```
% make tecs
```

どちらを用いても構いません。
次にマージします。

```
% tecsmerge gen src
```

今回はディレクトリごとマージしました。これにより src ディレクトリの下の tHelloWorld.c が更新されるとともに、tPutStringStdio.c が新たに作られます。

tHelloWorld.c は更新されたといっても、呼び口関数に関するコメントが追加されているだけです。以下の #_TCPF_# の箇所です。ですので、必ずしもマージする必要はありませんでした。

```
/* #[<PREAMBLE>]#
 * #[<...>]# から #[</...>]# で囲まれたコメントは編集しないでください
 * tecsmerge によるマージに使用されます
 *
 * 呼び口関数 #_TCPF_#
 * call port: cPutString signature: sPutString context:task
 *   void            cPutString_putString( const char_t* string );
 *
 * #[</PREAMBLE>]# */
```

tHelloWorld.c の受け口関数 eMain_main を以下のようにします。printf を cPutString_putString に置き換えます。これにより ヘッダファイル stdio.h を取り込む必要はなくなっていますので、#include ¡stdio.h¿ の行は、削除します。

```
--- tHelloWorld.c の内容 ---
...

/* プロトタイプ宣言や変数の定義をここに書きます #_PAC_# */
// #include  <stdio.h>                    <<<  削除


...

void
eMain_main(CELLIDX idx)
{
  CELLCB  *p_cellcb;
  if (VALID_IDX(idx)) {
    p_cellcb = GET_CELLCB(idx);
  }
  else {
```

```
    /* エラー処理コードをここに記述します */
  } /* end if VALID_IDX(idx) */

  /* ここに処理本体を記述します #_TEFB_# */
  // printf( "Hello World!\n" );            <<< 変更前
  cPutString_putString( "Hello World!\n" ); <<< 変更後
}
```

次に tPutStringStdio.c の受け口関数に追加します。

```
void
ePutString_putString(CELLIDX idx, const char_t* string, int32_t len)
{
  CELLCB  *p_cellcb;
  if (VALID_IDX(idx)) {
    p_cellcb = GET_CELLCB(idx);
  }
  else {
    /* エラー処理コードをここに記述します */
  } /* end if VALID_IDX(idx) */

  /* ここに処理本体を記述します #_TEFB_# */
  while( *string != 0 ){                  <<< 追加
    putchar( *string );                   <<< 追加
    string++;                             <<< 追加
  }                                       <<< 追加
}
```

これですべてのコードの修正が終わりました。それでは make して実行してみましょう。Makefile を変更する必要はありません。tPutStringStdio.c を追加しましたが、変更しなくても TECS ジェネレータがうまく調整してくれます。このことは、後で説明します。

```
% make
% ./HelloWorld.exe
*** starting task 'tTask_Task1'  1004010E0
Hello World!
*** exiting task 'tTask_Task1'
```

## 1.8 tecsmerge 使用上の注意

tecsmerge は、以下の場合に使用することができます。

- TECS ジェネレータの生成するテンプレートから、セルタイプコードを作成します (初期作成)
- 受け口のシグニチャの関数に増減があった場合に、受け口関数を追加、削除します
- 受け口に増減があった場合に、受け口関数を追加、削除します
- 受け口のシグニチャの関数の引数、戻り値に変更があった場合、関数ヘッダを変更します
- 受け口名、関数名に変更があった場合、それらを変更します (tecsmerge にオプション指定が必要です)

TECS での開発にあたって、tecsmerge によりテンプレートファイルの変更をセルタイプコードに反映する場合、セルタイプコードの初期生成を tecsmerge で行う必要があります。#[¡PREAMBLE¿]# や #[¡/PREAMBLE¿]# などは、キーワードです。tecsmerge は、この並び順をチェックしますから、不用意に順序を変えないようにしてください。

関数頭部では、終了キーワードを特別に扱っています。以下の例のように、行頭の ’’ がキーワードになっています。このため tecsmerge を使う場合には、流儀が相違したとしても、関数本体を開始する ’’ は、行頭から始めるようにしてください。

```
/* #[<ENTRY_FUNC>]# eMain_main   <<< 関数頭部開始
 * name:          eMain_main
 * global_name:   tHelloWorld_eMain_main
 * oneway:        false
 * #[</ENTRY_FUNC>]# */
void
eMain_main(CELLIDX idx)
{                                   <<< 関数頭部終了
                                    <<< （行頭に ’{’ があること）
    …
}
```

tecsmerge は便利なツールですが、少し落とし穴がありますので、注意点を記しておきます。

以前にインポート文に二種類あることを記しました。

```
import( "cygwin_kernel.cdl" );   … (1)
import( <cygwin_kernel.cdl> );   … (2)
```

(1) の書き方をしている場合には、cygwin_kernel.cdl 内で定義されているセルタイプのテンプレートが生成されます。この書き方をした場合、tTask のテンプレートファイルが生成されます。それを以下のコマンドでマージすると src ディレクトリに tTask.c が作成されます。

```
% tecsgen gen src
```

この状態で make した場合、src/tTask.c により tTask.o が生成されます。この結果、本当の tTask.c とは異なるソースが用いられますが、ビルドは成功してしまいます。動かすと中身のない tTask.c により動作しますから期待した動作はしません。

このような事態を避けるには tecsmerge にオプション -e を付加すると安全です。

```
% tecsgen -e gen src
```

オプション -e は、src ディレクトリに存在 (exist) するファイルだけをマージすることを指示します。このようにすれば、tTask.c が src ディレクトリ下に作られることを防ぐことができます。

もちろん、以前にも説明したように個別にファイルを指定する方法も使えます。多少面倒になりますが、テンプレートをマージしたいファイルだけを更新できます。

```
% tecsgen gen/tHelloWorld_templ.c src
```

なお、必ずしも tecsmerge を使用しなくてはならないということはありません。例えば TOPPERS/ASP3 のターゲット依存部では、tecsmerge 用のキーワードが削除されています。

あともう一点書き添えます。Makefile のテンプレート gen/Makefile.templ をマージすることはできません。

## 1.9 メッセージを変更可能にしてみよう

作成した HelloWorld　では、メッセージが Hello World! 固定になっていますが、変更可能にしてみましょう。もちろん、tHelloWorld.c を編集すれば変更可能なのですが、振る舞いのコード (セルタイプコード) から切り離して、セルの定義時にメッセージを指定可能にしてみましょう。

HelloWorld2 をベースに開発することにします。コピーして HelloWorld3 とします。

```
% cp -pr HelloWolrd2 HelloWorld3
% cd HelloWorld3
% make clean
```

それでは、CDL ファイル HelloWorld.cdl を編集します。変更箇所は 2 か所です。

まずは、セルタイプ tHelloWorld を変更します。

```
/* tHelloWorld セルタイプ */
celltype tHelloWorld {
  entry sTaskBody eMain;
  call  sPutString cPutString;
  attr {
    char_t  *message = "Hello World!\n";  <<<  メッセージ追加
  };
};
```

セルタイプ tHelloWorld に、char_t * 型の属性 (attr) として message を設けました。属性の値は、実行時に変更することはできません。ROM 化システムでは ROM に置かれることが想定されています。初期値として ”Hello World!
n” を指定しています。属性には、必ずしも初期値を与える必要はありません。

次にセル HelloWorld を変更します。

```
/* HelloWorld セル */
cell tHelloWorld HelloWorld {
  cPutString = PutStringStdio.ePutString;
  message = "Good luck with TECS!\n";    <<< メッセージ指定
};
```

セル HelloWorld では message に ”Good luck with TECS!
n” を指定しています。ここで message を初期化しない場合は、セルタイプで指定した ”Hello World!
n” が message の値となります。なお、属性は、必ずセルタイプかセルのいずれかで初期化されなくてはなりません。両方で初期値が指定されている場合には、セルの初期値が採用されます。

次にセルタイプコードをの変更です。再び tecsmerge しましょう。

```
% make tecs
% tecsmerge gen src
```

tecsmerge の結果 tHelloWorld.c には、以下のコメントが追加されています。属性 message はセルタイプコードの中では、以下のように ATTR_ を前置きして ATTR_message として参照します。なお、この場合、コメントしか変更されませんので、必ずしも tecsmerge を実行する必要はありません。

```
* 属性アクセスマクロ #_CAAM_#
* message          char_t*          ATTR_message
```

次に tHelloWorld.c の eMain_main 関数を一部変更します。

```
void
eMain_main(CELLIDX idx)
{
  CELLCB  *p_cellcb;
  if (VALID_IDX(idx)) {
    p_cellcb = GET_CELLCB(idx);
  }
  else {
    /* エラー処理コードをここに記述します */
  } /* end if VALID_IDX(idx) */

  /* ここに処理本体を記述します #_TEFB_# */
  // cPutString_putString( "Hello World!\n" );   <<< 変更前
  cPutString_putString( ATTR_message );          <<< 変更後
}
```

文字列を直接指定していたのを、属性に置き換えました。では make して実行してみましょう。

```
% make
% ./HelloWorld.exe
*** starting task 'tTask_Task1'  1004010E0
Good luck with TECS!
*** exiting task 'tTask_Task1'
```

メッセージが変更されました。

このように振る舞いのコードとデータを切り離せば、コンポーネントの再利用性を高めることができます。

## 1.10 複合セルタイプ (composite) 化してみよう

次に複合セルタイプ (composite) 化してみましょう。複合セルタイプとは、セルを組合わせてひとまとめにすることで、一つのセルタイプのように扱う

ことを可能にするものです。HelloWorld3 をベースに開発することにします。

```
% cp -pr HelloWolrd3 HelloWorld4
% cd HelloWorld4
% make clean
```

再び HelloWorld.cdl を編集します。まずは、tTask と tHelloWorld をまとめたコンポーネントを作成します。

```
[active]
composite tHelloWorldTask {
    /*--- インタフェース表明 ---*/
    call  sPutString  cPutString;
    entry sTask       eTask;
    attr {
        char_t *message;
    };

    /*--- 内部セル ---*/
    /* HelloWorld セル */
    cell tHelloWorld HelloWorld {
        cPutString => composite.cPutString;
        message = composite.message;
    };

    /* Task1 セル */
    cell tTask Task1 {
        cBody = HelloWorld.eMain;
        priority = 11; /* この値は使われていない */
        stackSize = 4096; /* この値は使われていない */
        taskAttribute = C_EXP( "TA_ACT");
    };

    /*--- 外部結合（受け口のみ）---*/
    composite.eTask => Task1.eTask;
};
```

まず、初めに [active] との記述があるのは、内部にアクティブなセル Task1 を含んでいるためです。'composite' は、複合セルタイプの定義を開始することを表すキーワードです。それに続けて複合セルタイプ名を書きます。複合セルタイプは、セルの定義においては、セルタイプと同様に扱えます。

続いて '', '' で囲んで、インタフェース表明（呼び口、受け口、属性)、内部セル、外部結合を記述します。

インタフェース表明、すなわち 呼び口、受け口、属性の記述は、この複合セルタイプのセルを定義する際に必要なインタフェースです。インタフェース表明部分の書き方は、セルタイプ (celltype) と同様です。変数 (var) は含まれません。var は、外部から直接操作できない内部実装であということもありますが、そもそも複合セルタイプ自体の var を持つことはありません。

実装部分として HelloWorld と Task1 セルを composite の中に置いています。

HelloWorld セルの呼び口 cPutString は、外部結合します。外部結合とは、このセルの呼び口を複合セルタイプの呼び口とすることです。外部結合には '=¿' を用います。'=¿' の左辺にセルの呼び口、右辺には 'composite.' に続けて複合セルタイプの呼び口名を書きます。これらのシグニチャは一致する必要があります。

```
cPutString => composite.cPutString;
```

属性 message も外部結合となります。属性の場合、'=' を用います。'=' の左辺はセルの属性名、右辺は 'composite.' に続けて複合セルタイプの属性名を書きます。これらの型は一致している必要があります。

```
message = composite.message;
```

Task1 セルは、右にインデントしただけで、特に変更を加えていません。属性 priority, stackSize, taskAttribute の値は、決定され、複合セルタイプのセルを定義する時には変更できません。

外部結合部分には、受け口の外部結合を書きます。'=¿' の左辺は、インタフェース表明の受け口です。'composite.' に続けてインタフェース部の受け口名を書きます。'=¿' の右辺は、内部セルの受け口です。セル名、'.'、受け口名をつなげたものを指定します。

それでは組上げてみましょう。PutStringStdio セルはそのままにして、HelloWorld と Task1 を HelloWorldTask に置き換えます。

まず　Task1 を削除します。

```
// /* Task1 セル */
// cell tTask Task1 {
//  cBody = HelloWorld.eMain;
//  priority = 11; /* この値は使われていない */
//  stackSize = 4096; /* この値は使われていない */
//  taskAttribute = C_EXP( "TA_ACT");
// };
```

HelloWorld セルのセルタイプとセル名を変更します。

```
cell tHelloWorldTask HelloWorldTask{        <<< セルタイプを変更
  message = "Good luck with TECS!\n";       <<< この行は必須
  cPutString = PutStringStdio.ePutString;
};
```

ここで message の初期化は必須です。理由は複合セルタイプ tHelloWorld-Stdio の属性 message に初期値を与えていないからです。

複合セルタイプ化では、セルタイプコードへの影響はありません。それでは、ビルドしてみましょう。

```
% make
% ./HelloWorld.exe
*** starting task 'tTask_Task1'  1004010E0
Good luck with TECS!
*** exiting task 'tTask_Task1'
```

## 1.11　もう一組作ってみよう

さて、TECS 化した恩恵の一つが、再利用する場合に現れます。HelloWorld4 をベースに開発することにします。

```
% cp -pr HelloWolrd4 HelloWorld5
% cd HelloWorld5
% make clean
```

再び HelloWorld.cdl を編集します。

```
/* HelloWorldTask2 セル */
cell tHelloWorldStdio HelloWorld2 {
  message = "Have a nice dream!\n";
  cPutString = PutStringStdio.ePutString;
};
```

では、ビルドして実行してみましょう。

```
% make
% ./HelloWorld.exe
*** starting task 'tTask_Task1'  1004010E0
Good luck with TECS!
```

```
*** exiting task 'tTask_Task1'
*** starting task 'tTask_Task2'  1004010E0
Have a nice dream!
*** exiting task 'tTask_Task2'
```

なお、今回の実装では排他制御を取っていません。運が悪いと、以下のように 2 つのタスクのメッセージが混じって出力される可能性もありますが、ある意味では期待した動作結果と言えます。

```
% ./HelloWorld.exe
*** starting task 'tTask_Task2'  1004010E0
*** starting task 'tTask_Task1'  1004010E0
HGaovoed  al uncikc ew idtrhe aTmE!C
*** exiting task 'S!
tTask_Task2'
*** exiting task 'tTask_Task1'
```

TECS コンポーネント図は、以下のようになります。

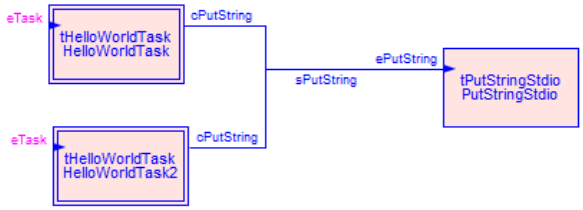


図 1.3: もう一つ作った後の TECS コンポーネント図

PutStringStdio は、HelloWorldTask と HelloWorldTask2 の呼び口から結合されています。これは PutStringStdio は、2 つのタスクで共用されていることを表します。このように呼び口は、合流して受け口に結合することができます。反対に分流することはできません。

## 1.12 排他制御を追加しよう

先の例では、複数のタスクのメッセージが入り混じる可能性がありました。それであれば、出力先で排他制御を行い、異なるタスクのメッセージが交わらないようにしましょう。HelloWorld5 をベースに開発することにします。

```
% cp -pr HelloWolrd5 HelloWorld6
% cd HelloWorld4
% make clean
```

まずは CDL ファイル HelloWorld.cdl を編集します。3 か所編集します。排他制御には cygwin_kernel.cdl で提供されているセマフォを用いることにします。

1 か所目は、tPutStringStdio セルタイプです。セマフォを結合するための呼び口を設けます。

```
/* tPutStringStdio セルタイプ */
celltype tPutStringStdio {
  entry  sPutString ePutString;
  call   sSemaphore cSemaphore;       <<< 追加
};
```

2 か所目は、PutStringStdio セルです。セマフォ PutStringStdioSemaphore を結合します。

```
/* PutStringStdio セル */
cell tPutStringStdio PutStringStdio {
    cSemaphore = PutStringStdioSemaphore.eSemaphore; <<< 追加

};
```

3 か所目は、セマフォ PutStringStdioSemaphore セルを作成します。cygwin_kernel.cdl にある tSemaphore の定義からすると count と attribute は初期化されていませんから、セルで初期化する必要があります。これらの初期値として TOPPERS/ASP と同じ値を与えることができますが、有効ではありません。ここでは適当な値として count を 0 に, attribute を TA_NULL (ヘッダにあるマクロを参照) に初期化します。

```
/* PutStringStdioSemaphore セル */          <<<  追加
cell tSemaphore PutStringStdioSemaphore { <<<  追加
    count = 0;                            <<<  追加
    attribute = C_EXP( "TA_NULL" );       <<<  追加
};                                        <<<  追加
```

以上で CDL ファイル HelloWorld.cdl の編集は終わりです。

引き続きセルタイプコード tPutStringStdio.c を編集します。セマフォを獲得 (wait)、解放 (signal) するコードを追加します。

```
void
ePutString_putString(CELLIDX idx, const char_t* string)
{
 CELLCB *p_cellcb;
```

```
  if (VALID_IDX(idx)) {
  p_cellcb = GET_CELLCB(idx);
  }
  else {
  /* エラー処理コードをここに記述します */
  } /* end if VALID_IDX(idx) */

  /* ここに処理本体を記述します #_TEFB_# */
   cSemaphore_wait();                  <<< 追加
   while( *string != 0 ){
       putchar( *string );
       string++;
   }
   cSemaphore_signal();                <<< 追加
}
```

それでは、ビルドして実行してみましょう。

```
% make
% ./HelloWorld.exe
*** starting task 'tTask_Task1'  1004010E0
Good luck with TECS!
*** exiting task 'tTask_Task1'
*** starting task 'tTask_Task2'  1004010E0
Have a nice dream!
*** exiting task 'tTask_Task2'
```

今度は、何度やってもメッセージが入り混じったりはしません。もちろん、メッセージの順序が入れ替わる可能性はあります。

## 1.13 自前のバッファを使用しよう

高水準の I/O を用いていましたが、自前でバッファを用意して低水準の I/O を用いるようにしてみましょう。TECS には、組込みステム向きのバッファを準備するのによい手段を持っています。HelloWorld6 をベースに開発することにします。

```
% cp -pr HelloWolrd6 HelloWorld7
% cd HelloWorld4
% make clean
```

それでは、例によって CDL ファイル HelloWorld.cdl から編集しましょう。属性としてバッファサイズ bufSize を、(内部) 変数にバッファ buf を設けます。バッファ buf には bufSize で指定された大きさのメモリ領域が確保されます。

```
/* tPutStringStdio セルタイプ */
celltype tPutStringStdio {
  entry  sPutString ePutString;
  call   sSemaphore cSemaphore;
  attr {                              <<< 追加
      int  bufSize = 256;             <<< 追加
  };                                  <<< 追加
  var {                               <<< 追加
      [size_is(bufSize)]              <<< 追加
          char_t *buf;                <<< 追加
  };                                  <<< 追加
};
```

引き続きセルタイプコード tPutStringStdio.c を編集します。まず、カウンタ変数 i を追加します。(内部) 変数は VAR_buf のように 'VAR_' を前置きすることで参照可能になります。バッファ buf のサイズは size_is で指定された bufSize になります。セルタイプコードでは ATTR_bufSize として参照できます。低水準 I/O write が二回呼びされますが、最初のものは、バッファ buf のサイズ以上に、文字列 string で渡された文字列の長さが長い場合、バッファがフルになった時点で吐き出すためのものです。2 つ目は、バッファにあるものを吐き出します。文字列の長さが bufSize 以下の場合、こちらの write のみ呼び出されます。

セマフォは、引き続き必要です。複数のタスクから呼び出されますから、バッファの操作は排他的に行う必要があります。

```
void
ePutString_putString(CELLIDX idx, const char_t* string)
{
 CELLCB *p_cellcb;
  int i;                                    <<< 追加
 if (VALID_IDX(idx)) {
 p_cellcb = GET_CELLCB(idx);
 }
 else {
 /* エラー処理コードをここに記述します */
 } /* end if VALID_IDX(idx) */
```

```
  /* ここに処理本体を記述します #_TEFB_# */
   cSemaphore_wait( );
   i = 0;                                      <<< 追加
   while( *string != 0 ){
       // putchar( *string );                  <<< 削除
       VAR_buf[i] = *string;                   <<< 変更
       string++;
       i++;                                    <<< 追加
       if( i == ATTR_bufSize ){                <<< 追加
           write( 1, VAR_buf, ATTR_bufSize );  <<< 追加
           i = 0;                              <<< 追加
       }                                       <<< 追加
   }
   if( i > 0 )                                 <<< 追加
       write( 1, VAR_buf, i );                 <<< 追加
   cSemaphore_signal( );
  }
```

以上で変更は終わりです。それではビルドして実行してみましょう。排他制御を取っていますから、メッセージが混じることはありませんし、バッファ操作を正しく行えます。

```
% make
% ./HelloWorld.exe
*** starting task 'tTask_HelloWorldTask_Task1'  1004010E0
Good luck with TECS!
*** starting task 'tTask_HelloWorldTask2_Task1'  1004010E0
Have a nice dream!
*** exiting task 'tTask_HelloWorldTask_Task1'
*** exiting task 'tTask_HelloWorldTask2_Task1'
```

## 1.14 一つのタスクで複数のメッセージを出力

これまでと変わって、1 つのタスクで複数のメッセージを読み出しながら出力する方法を考えます。PutStringStdio だけ再利用することにします。

今回は、先にコンポーネント図をお見せします。

少し複雑な形になっています。構成要素を概観すると、まずタスク Task1 があります。タスクの本体が GetAndPutMessage セルです。

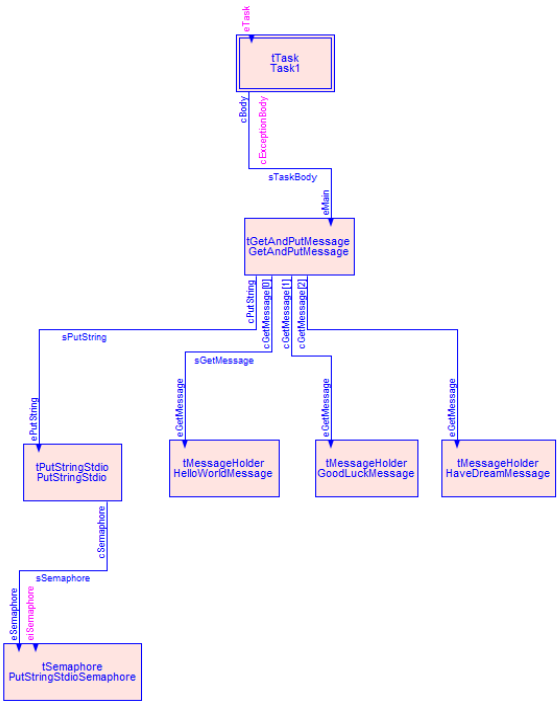


図 1.4: 一つのタスクで複数のメッセージを出力

GetAndPutMessage セルは、出力先である tPutStringStdio と、3 つのメッセージを保持するセル HelloWorldMessage, GoodLuckMessage, HaveDreamMessage が結合されています。3 つのメッセージを保持するセルの結合には、呼び口配列が用いられています。今回は、この点が主題になります。

CDL ファイル HelloWorld.cdl を編集します。シグニチャ、セルタイプ、セルに分けて、説明します。以下の要素、すなわち出力先 PutStringStdio に関する記述を省略しています。前項のものをそのまま用います。

```
sPutString
tPutStringStdio
PutStringStdio
PutStringStdioSemaphore
```

まずは、シグニチャです。シグニチャ sGetMessage には一つだけ関数があります。文字列を返す関数 getMessage です。出力引数 buf に文字列が返されます。TECS では、引数の指定子が out の場合、関数の呼び元が記憶域を用意しなくてはなりません。また、ポインタ型の引数に size_is が指定されている場合、ポインタは配列を指していることを表します。getMessage の第一引数 buf は、長さ len の大きさを持つ char_t 型の配列へのポインタです。な

お size_is は配列の長さであって、バイト数ではありません。sizeof と混同しないよう、注意してください。size_is の引数 len は、in または inout の引数でなくてはなりません（ただし inout の場合はポインタ型になる)。buf の記憶域は、関数の呼び元が準備しますから、その大きさは呼び元でないとわかりません。従って、size_is の引数は in 方向の値が用いられます。

```
/* sGetMessage シグニチャ  */
signature sGetMessage {
    void getMessage( [out, size_is(len)]char_t *buf, [in]int32_t len );
};
```

次は、セルタイプです。2つのセルタイプ tMessageHolder, tGetAndPutMessage を追加します。tMessageHolder は、メッセージを記憶するだけのコンポーネントです。tGetAndPutMessage は、呼び口配列 cGetMessage からメッセージを取り出し、呼び口 cPutString を通してメッセージ文字列を出力します。

```
/* tMessageHolder セルタイプ */
celltype tMessageHolder {
  entry sGetMessage eGetMessage;
  attr {
      char_t *message = "Hello World!";
  };
};

/* tGetAndPutMessage セルタイプ */
celltype tGetAndPutMessage {
    call sGetMessage cGetMessage[];
    call sPutString  cPutString;
    entry sTaskBody  eMain;
};
```

最後にセルです。タスクのメインとなるセル GetAndPutMessage と 3 つのメッセージホルダーセル HelloWorldMessage, GoodLuckMessage, HaveDreamMessage とタスク Task1 を追加します。

```
/* GetAndPutMessage セル */
cell tGetAndPutMessage GetAndPutMessage {
    cGetMessage[0] = HelloWorldMessage.eGetMessage;
    cGetMessage[1] = GoodLuckMessage.eGetMessage;
    cGetMessage[2] = HaveDreamMessage.eGetMessage;
    cPutString = PutStringStdio.ePutString;
```

```
};

/* HelloWorldMessage セル */
cell tMessageHolder HelloWorldMessage {
    message = "Hello World!\n";
};

/* GoodLuckMessage セル */
cell tMessageHolder GoodLuckMessage {
    message = "Good luck with TECS!\n";
};

/* HaveDreamMessage セル */
cell tMessageHolder HaveDreamMessage {
    message = "Have a nice dream!\n";
};

/* Task1 セル */
cell tTask Task1 {
    cBody = GetAndPutMessage.eMain;
    priority = 11; /* この値は使われていない */
    stackSize = 4096; /* この値は使われていない */
    taskAttribute = C_EXP( "TA_ACT");
};
```

セル GetAndPutMessage はタスクのメインとなるものです。このセルには呼び口配列 cGetMessage があります。cGetMessage には、3 つのメッセージホルダセルが結合されており、添数 0 〜 2 が指定されています。セル記述において、呼び口配列の添数は省略することができます。その場合は、出現順に 0 〜 2 が割り付けられます。添数は、省略するか、記載するか、どちらかに統一しなくてはなりません。なお、セルタイプにおいて呼び口配列の要素数が指定されていませんので、何個でも結合できます。

tGetAndPutMessage セルタイプの 3 つのセル HelloWorldMessage, GoodLuckMessage, HaveDreamMessage は、メッセージを保持するセルです。

最後にタスクセル Task1 です。これは、すでに定義したものと同様です。ただ、結合先を合わせる必要があります。

## 1.15 おさらい

1 章では、TECS を概観しました。いくらかの機能を見てきましたが、まだまだ重要な機能がたくさんあります。それらを挙げてみます。

- 呼び口配列
- 受け口配列
- ファクトリ (factory, FACTORY)
- リクワイア (require)
- 固定結合 (逆結合)
- プラグイン
- 指定プロトタイプ宣言

まだまだ重要な機能はたくさんあります。

- 動的結合
- アロケータ

# 第2章　シグニチャ

## 2.1　シグニチャの指定子

### 2.1.1　context

### 2.1.2　deviate

### 2.1.3　generate

### 2.1.4　関数がゼロ個のシグニチャ

## 2.2　関数

## 2.3　関数の指定子

### 2.3.1　oneway

## 2.4　引数の指定子

### 2.4.1　in

### 2.4.2　out

### 2.4.3　inout

### 2.4.4　size_is

### 2.4.5　count_is

### 2.4.6　string

# 第3章　セルタイプ

## 3.1　属性と変数

## 3.2　ファクトリ

## 3.3　リクワイア

## 3.4　逆リクワイア

## 3.5　セルタイプの指定子

# 第4章　セルと組上げ

## 4.1　プロトタイプ宣言

## 4.2　指定プロトタイプ宣言

## 4.3　合流と分流

# 第5章　複合セルタイプ

# 第6章　アロケータ

## 6.1　アロケータ概観

## 6.2　(通常の) アロケータ

## 6.3　セルフアロケータ

## 6.4　リレーアロケータ

# 第7章　前置部

## 7.1　generate 文

## 7.2　import 文

## 7.3　import_C 文

# 第8章　Makefile

TOPPERS_MACRO_ONLY TOPPERS_CB_TYPE_ONLY

# 第9章　ネームスペース

# 第10章　リージョン

# 第11章　TECS とオブジェクト指向

# 第12章　RPC

## 12.1　トランスペアレント RPC

## 12.2　オペイク RPC

# 第13章　プラグイン

# 第14章　TOPPERS/ASP3

# 第15章　TOPPERS/HRP2