リアルタイムカーネル勉強会 資料
コンフィギュレータ

藤田 静男

平成 13 年 9 月 17 日

1 概要

ここでは TOPPERS/JSP に付属するコンフィギュレータについて解説を行う．コンフィグレータは，プリプロセッサで処理されたコンフィギュレーションファイルをもとに，`kernel_cfg.c` と `kernel_id.h` を出力する [図1].

コンフィギュレータを構成するためのファイルは./cfg ディレクトリ内に格納されている．

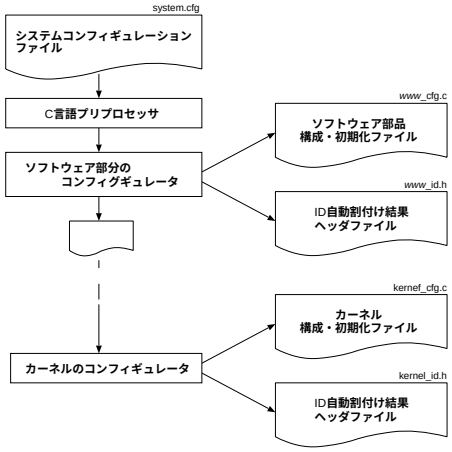


図 1: システムコンフィギュレーションファイルの処理手順

コンフィグレーションファイルはカーネルやソフトウェア部品の構成やオブジェクトの初期状態を定義するためのファイルである．システムコンフィギュレーションファイルには，カーネルやソフトウェア部品の静的 API と ITRON 仕様共通静的 API に加えて，C 言語処理系のプリプロセッサディレクティブを記述することができる．コンフィギュレーションファイル中の静的 API を解釈して，カーネルやソフトウェア部品を構成するためのツールがコンフィギュレータである．([仕様書], p32).

2　コンフィギュレーションファイル

2.1　sample1.cfg

TOPPERS/JSP で提供されているコンフィギュレーションファイルのサンプルを図2 に示す.

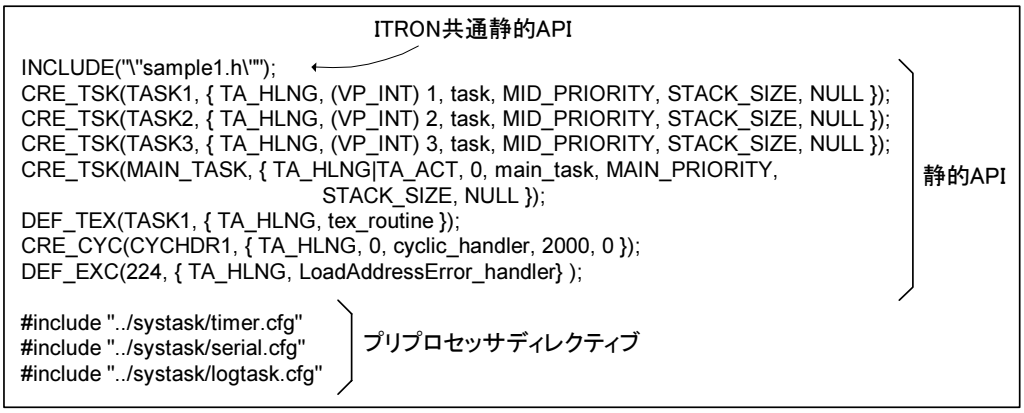


図 2: sample1.cfg

CRE_TSK などの静的 API で ID を記述すべきところで，数値が直接記述されていない場合，コンフィギュレータが kernel_id.h へ適切な値で定義する.

プリプロセッサディレクティブで宣言されているコンフィギュレーションファイルの内容は以下の通りである.

timer.cfg　システムクロックドライバ
serial.cfg　シリアルインタフェース
logtask.cfg　システムログタスク

3　コンフィグレータ

3.1　クラスセット

プログラム中では以下に示されるクラスを用いることで，コンフィギュレータファイルに記述されている静的 API やそのパラメータのチェック，必要であればタスク ID などを適切に割当て，それらの結果を適切な形式で出力する.

ID　ID の管理，割当等を行う
Varient　数字，文字列等を種別を問わずに一括管理する変数
Array　Valient 型配列
Exception　例外型
Manager　プログラム本体
MultiStream　複数の出力ファイルを一括管理するクラス
Parser　ファイルのパーシングを行うクラス
StaticAPI　静的 API のベースクラス
Serializer　ファイル出力を行うクラス

3.2 動作概念

コンフィギュレータの動作概念は以下の通りである．

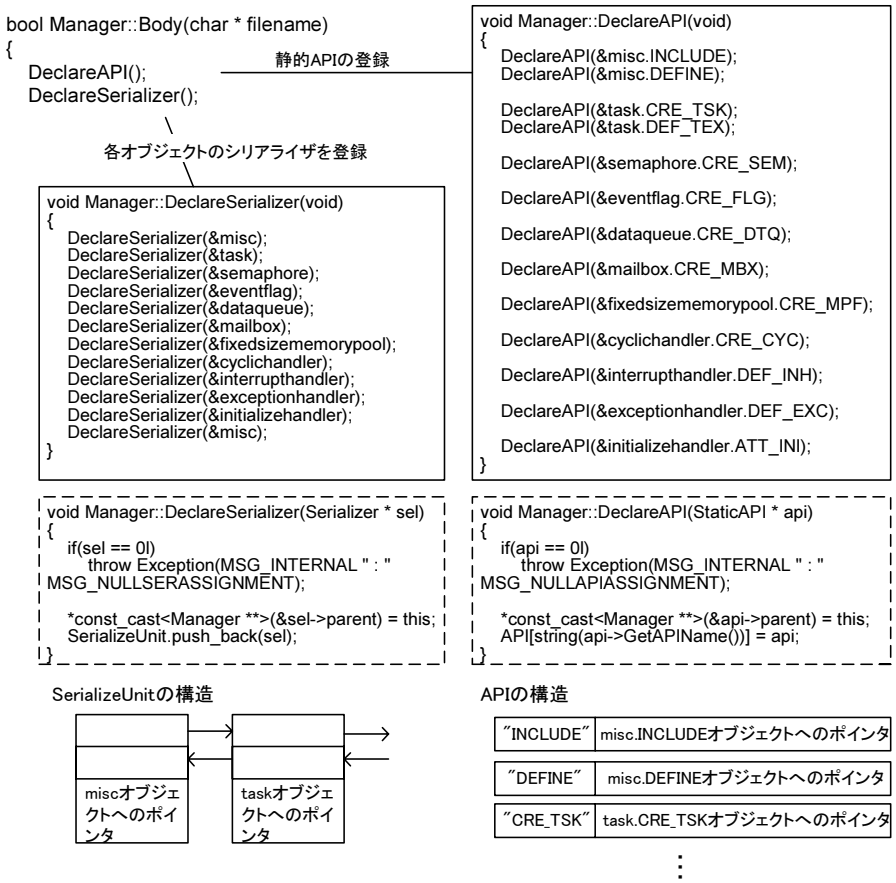
1. ファイル名，起動オプションをコマンドラインより取得する．
2. ファイル中で使用されている静的 API 名を登録し，各 API に対応した出力用のメソッド (シリアライザ) を登録する．
3. ファイルからコンフィギュレーション内容を取得し，その内容を API 名とパラメータに切り分け，正しい値であるかどうかを確認し，必要であれば値を割当てる．
4. ファイルに出力する情報の正当性を確認する．
5. 情報が正当であればそれらの情報を出力する．情報が出力される順序は，各 API のシリアライザの登録順序に従う．

2 項目目以降がプログラム本体での処理となり，次の節からはこの流れに沿って説明していく．

3.3 静的 API 名・シリアライザ登録

プログラムの本体部分は manager.cpp の `bool Manager::Body(char * filename)` である．メイン関数からコンフィギュレーションファイル名を引数として与えられることで，動作概念の 2 番目以降の処理を受け持つ．

以下のコードによって静的 API 名と，それに対応するシリアライザを登録する．



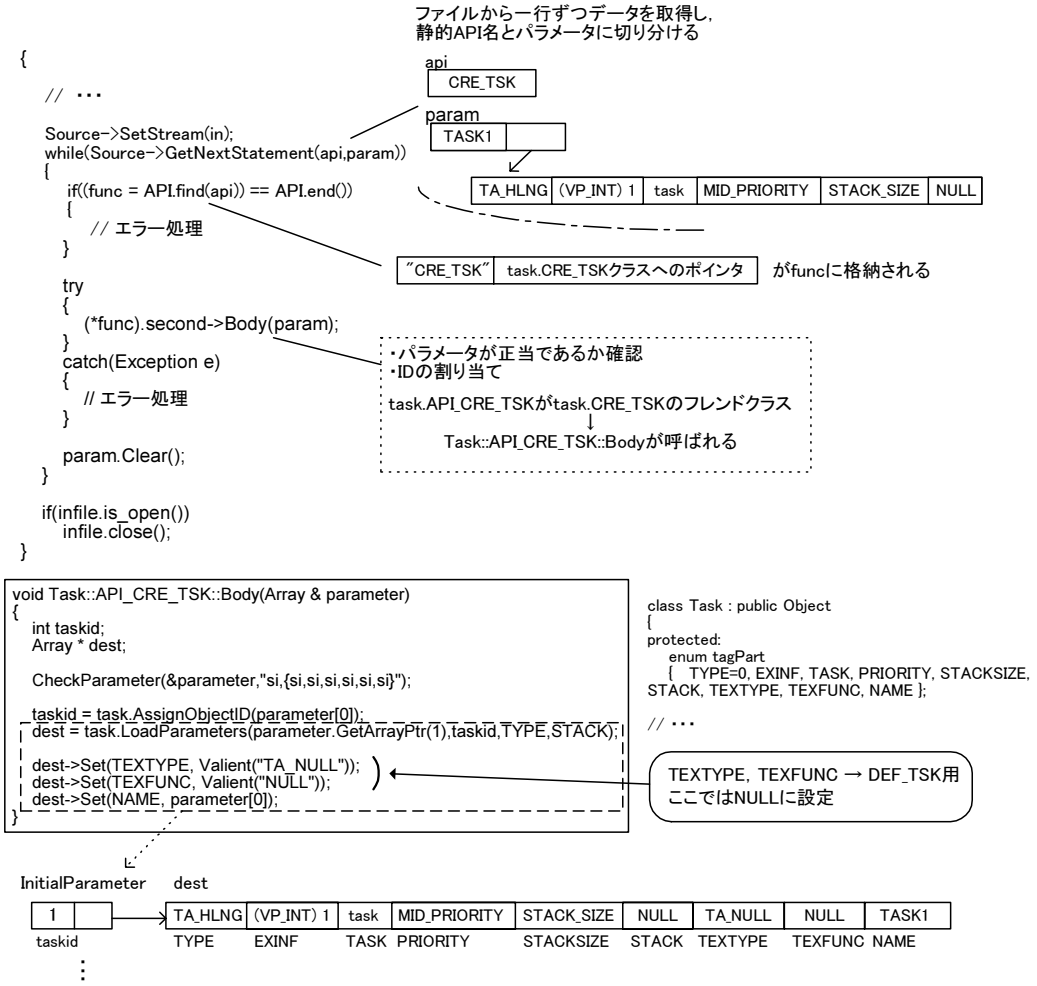
DeclareAPI，DeclareSerializer を呼び出すことで，上の図のように API と SerializeUnit が生成され，これ以降のプログラム中で利用される．

3.4 パラメータチェック

bool Manager::Body(char * filename) の次のブロックでの処理で，ここでは，コンフィギュレーションファイルからデータを取得して，静的 API 名とパラメータに切り分ける処理を行う．下の図では

```
CRE_TSK(TASK1, { TA_HLNG, (VP_INT) 1, task, MID_PRIORITY, STACK_SIZE, NULL });
```

という行を読み込んだ場合の処理を示している．



CheckParameter(&parameter,"si,{si,si,si,si,si,si}"); で取得したパラメータの形式を確認する．si はパラメータが文字列か符号付き整数であるかを確認することを意味する．他にも，以下の型を確認することができる．

- Ｓ ｓ ― 文字列
- Ｉ ｉ ― 符号付き整数

- `U u` ― 符号なし整数
- `D d` ― 浮動小数点
- `P p` ― ポインタ
- `-` ― 型の指定なし

`taskid = task.AssignObjectID(parameter[0]);` では ID 番号の割り当てが行われる．`parameter[0]` が単なる整数値であれば，すでに割り当てられていないかどうか確認し ID を割り当てる．文字列であれば `kernel_id.h` に`#define` 文で割り当てた ID 番号を宣言する必要があるため，オブジェクト名と ID 番号を組としたデータ構造を生成する [図3].

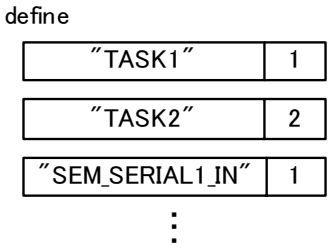


図 3: define 文出力用のデータ構造

3.5 情報の正当性確認

`bool Manager::Body(char * filename)` の以下の箇所で ID 番号が正当に割り当てられているかどうかを確認する．次の条件を満たしている場合が正当であり，下図の `Identifier.IsAvailable()` がその確認を行う．

1. ID 値が連続でなければならない．
2. ID 値は有効範囲でなければならない．
3. ID 値は最小値から始まらなければならない．

- オブジェクトの ID 番号には 1 から連続した正の値を用いる．([仕様書]，p27)
- スタンダードプロファイルでは，少なくとも 1 〜 255 の範囲の正の値の ID 番号をサポートしなければならない．([仕様書]，p27)

ユーザがコンフィギュレーションファイルに ID 番号を直接指定する場合，この規則に適応していないかも知れないため，このような処理が必要になる．

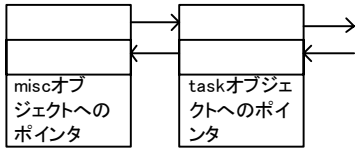
```
{
   bool cont;
   list<Serializer *> Checkers;
   list<Serializer *>::iterator scope;
   unsigned int LastCheckersCount;

   scope = SerializeUnit.begin();
   w hile(scope != SerializeUnit.end())
   {
      Checkers.push_back(*scope);
      scope ++;
   }

   do
   {
      LastCheckersCount = Checkers.size();
      scope = Checkers.begin();
      w hile(scope != Checkers.end())
      {
         try
         {
            cont = false;
            (*scope)->Check(cont);
            if(!cont)
               scope = Checkers.erase(scope);
            else
               scope ++;
         }
         catch(Exception e)
         {
            char buffer[1024];
            sprintf(buffer,MSG_VALIDATIONFAIL " (%s)",e.GetReason().c_str());
            throw  Exception(buffer);
         }
      }
   }w hile(Checkers.begin() != Checkers.end() && LastCheckersCount != Checkers.size());

   if(Checkers.begin() != Checkers.end())
      throw  Exception(MSG_VALIDATIONDEADLOCK);
}
```

Checkers : SerializeUnitのコピー

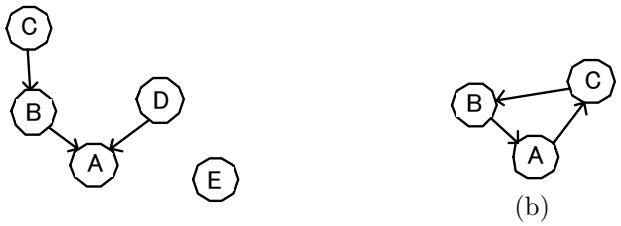


リストの長さを取得
Manager::DeclareSerializerで登録したオブジェクト数 = 12

```
void Object::Check(bool & cont)
{
   if(!Identifier.IsAvailable())
      throw Exception(MSG_ILLEGALIDENTIFIER
MSG_ILLEGALIDENTIFIER2);
}
```

リストを削除していくので，ループ文を抜けるときにはbeginとendは一致しなければならない

プログラム中では `cont` の値で処理を分けて書かれているが，実際は `cont` が `true` になることがない．ID 値の正当性を確認する際に，オブジェクト間に依存関係があった場合に使用する．



(a) (b)

図 4: 依存関係

例えば，図4のようにあるオブジェクトの ID 値の正当性が確認されてから，次の ID 値を確認しなければならないときがそれである．

(a) の場合，A が B と D に依存するため確認が保留される．同じく B も C に依存するため保留される．C，D，E は他に依存することがないので，ID 値が確認されたらリストから削除される．2 度目の do - while 文の処理に入り，B の ID 値が確認されリストから削除される．3 度目の do - while 文の処理で A がリストから削除される．(b) のような場合，1 つもリストから削除されることがなく，LastCheckersCount と Checkers.size() が等しくなるためループから抜け，次の if 文で例外処理される．

実際には，JSP がサポートする静的 API の各オブジェクト間には依存関係が存在しないので，このような処理は行われることがない．

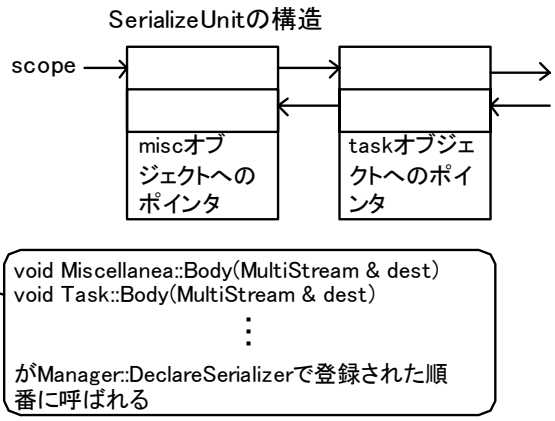
3.6 情報出力

コンフィグレーションファイルを処理した情報を出力する部分は `Manager::Body` の以下の部分で行われる．`kernel_cfg.c` に出力される順序は，`Manager::DeclareSerializer` でシリアライザを登録した順序に従う．

```
{
   list<Serializer *>::iterator scope;
   MultiStream outfile;

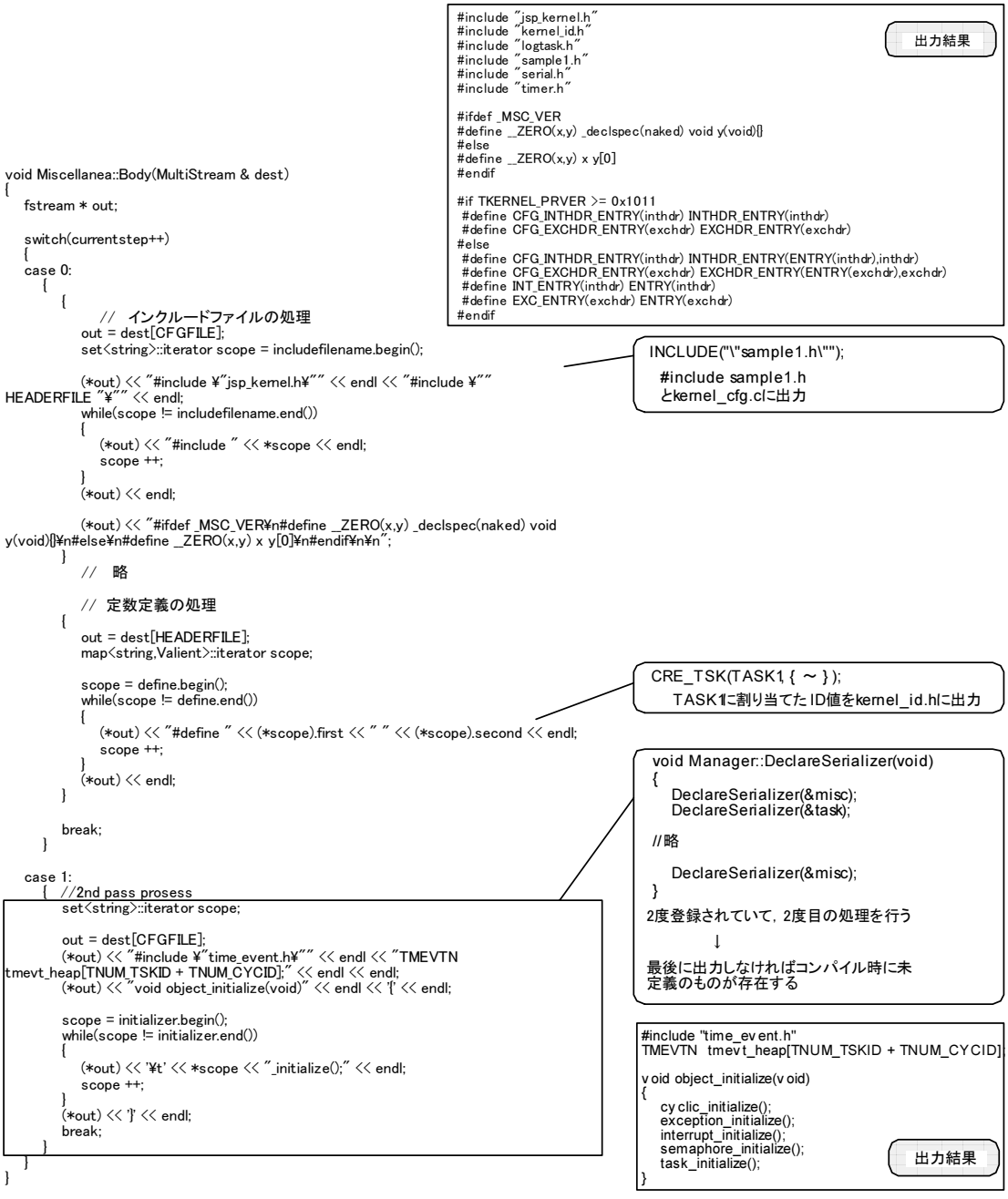
   scope = SerializeUnit.begin();
   w hile(scope != SerializeUnit.end())
   {
      try
      {
         (*scope)->Body(outfile);
      }
      catch(Exception e)
      {
         //  エラー処理
      }
      scope ++;
   }

   outfile.CloseAllStream();
}
```



3.6.1 ヘッダ部の出力

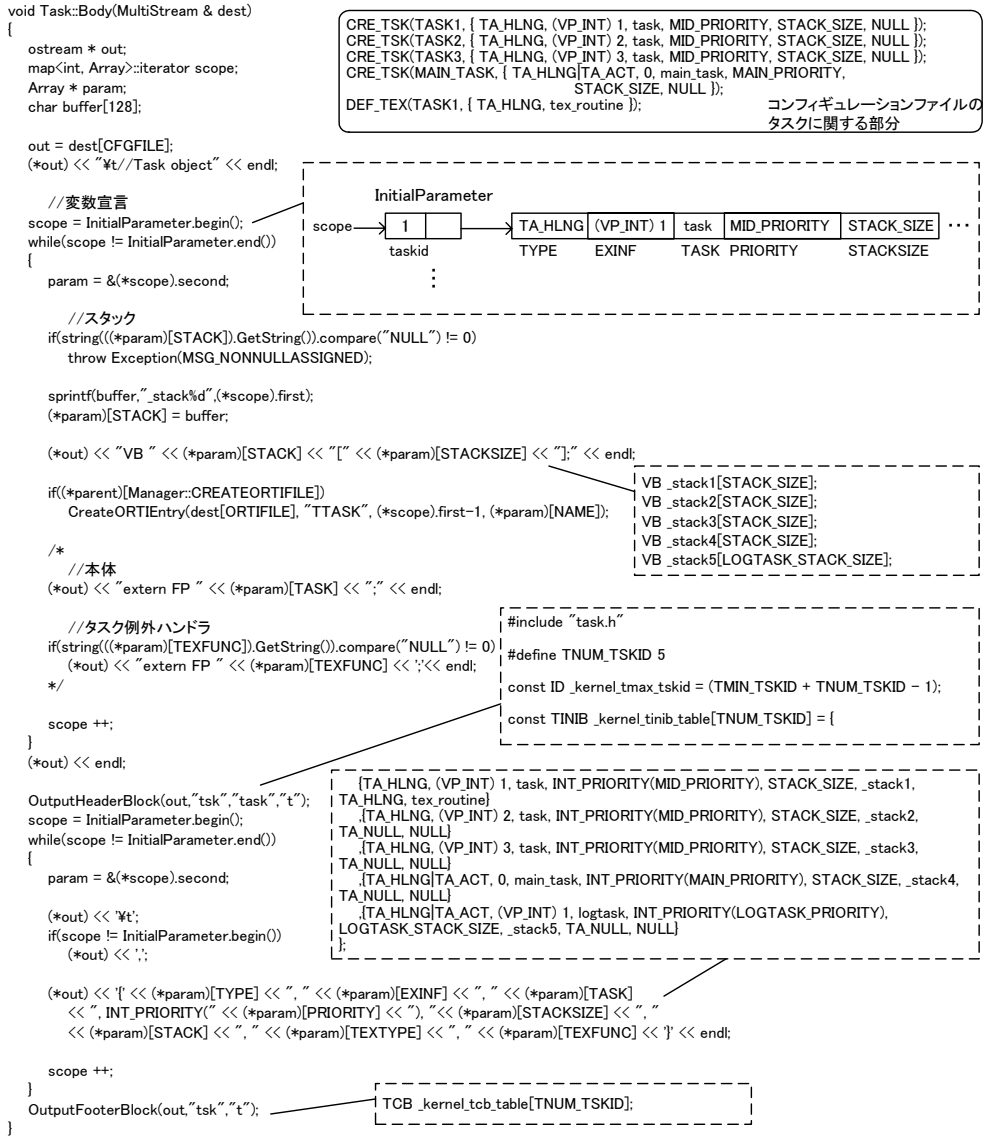
void Miscellanea::Body(MultiStream & dest) では，オブジェクトに割り当てた ID 値を kernel_id.h に出力し，kernel_cfg.c へはファイルの先頭に記述される#include 文や各オブジェクトの初期化関数を呼び出す部分を出力する．



tmevt_heap のサイズは各オブジェクトのシリアライザが呼び出されると決定するので，初期化関数を呼び出す部分と共に最後に出力される．

3.6.2 各オブジェクトの出力

ここでは，タスクに関する情報を出力する関数を示す．セマフォやイベントフラグ等も若干の違いはあるものの，処理の流れは同じである．



まずはじめに，各タスクのスタックサイズを出力する．次に生成されるタスクの数を定義している．TNUM_TSKID は kernel_cfg.c 内で使用され，_kernel_tmax_tskid はカーネル内で使用される．

```
kernel/jsp_rename.h:110:#define tmax_tskid _kernel_tmax_tskid
kernel/jsp_rename.h:226:#define _tmax_tskid __kernel_tmax_tskid
kernel/check.h:88: (TMIN_TSKID <= (tskid) && (tskid) <= tmax_tskid)
kernel/task.c:101: for (tcb = tcb_table, i = 0; i < tmax_tskid; tcb++, i++) {
kernel/task.h:228:extern const ID tmax_tskid;
```

そして，タスク初期化ブロックとタスクの数だけ TCB を生成する．

3.7 静的 API の追加

以下は./cfg 内の internal.txt の引用である．
新しく静的 API を追加するには，次の作業が必要になる．

1. 静的 API クラスの作成
2. シリアライザの作成
3. Manager::DeclareAPI への追加
4. Manager::DeclareSerializer への追加

静的 API クラスは StaticAPI クラスを派生させることで生成することができる．以下に，例として INCLUDE 文を処理するための StaticAPI クラスを追加する場合のものを示す．

```
class INCLUDE : public StaticAPI
{
public:
    char * GetAPIName(void) { return "INCLUDE"; };
    void Body(Array &);
};
```

StaticAPI::GetAPIName はコンフィギュレーションファイル中で利用される静的 API の名称を定める．StaticAPI::Body はコンフィギュレーション情報を処理するための関数である．Manager はこれをもとに処理のディスパッチを行う．

参考文献

[仕様書] 坂村 健監修 / 高田 広章編：μITRON 4.0 仕様 (ver 4.01.00).

付録 A

起動オプション

現在の TOPPERS/JSP (version 1.1.1) では，コンフィグレータに起動オプションとして-O または--odl を指定することができる．

```
#
#  カーネルのコンフィギュレーションファイルの生成
#
kernel_cfg.c: $(UTASK_CFG)
    $(CPP) $(INCLUDES) $(CDEFS) $(UTASK_CFG) | ../cfg/cfg --odl
```

どちらも動作内容は同じで，ORTI(OSEK/VDX Runtime Interafce) 風[1]の記述ファイル toppers_info.odl を出力します．

またこの記述ファイルは，タスク ID などシステムごとに変化する部分のみしか出力されていないため，記述ファイルの本体[2]とマージする必要がある．図5にコンフィグレーション結果として生成される記述ファイルを示す．メイン部分は別資料として配布する．

```
TEMPLATE TTASK          TASK1("0");
TEMPLATE TTASK          TASK2("1");
TEMPLATE TTASK          TASK3("2");
TEMPLATE TTASK          MAIN_TASK("3");
TEMPLATE TTASK          LOGTASK("4");
TEMPLATE TSEMAPHORE     SEM_SERIAL1_IN("0");
TEMPLATE TSEMAPHORE     SEM_SERIAL1_OUT("1");
TEMPLATE TSEMAPHORE     SEM_SERIAL2_IN("2");
TEMPLATE TSEMAPHORE     SEM_SERIAL2_OUT("3");
TEMPLATE TCYCLIC        CYCHDR1("0");
TEMPLATE TINTERRUPT     INHNO_TIMER("0");
TEMPLATE TINTERRUPT     INHNO_SERIAL("1");
TEMPLATE TEXCEPTION     TEXCEPTION_224("0");
```

図 5: 記述ファイル

[1]ITRON 用に拡張が施されている．
[2]http://www.ertl.ics.tut.ac.jp/ takayuki/ prog/gdb/からダウンロードすることができる．

付録 B

ID 型

コンストラクタ

- `ID(const long Size = 255);`
  有効な ID 値の個数を与えて ID 型を生成する．最小値は 1 である．
- `explicit ID(long Lower, unsigned long Size = 255);`
  最小値と個数を与えて ID 型を生成する．
- `explicit ID(long Lower, long Upper = 255);`
  最小値と最大値を与えて ID 型を生成する．

  ID 型はコピーコンストラクタを持たない．これは ID をコピー可能とすることで該当オブジェクトの ID 管理に問題が起るのを避けるためである．

値の割当

- `bool Assign(long id);`
  値 id を使用 (予約) する．以後値 id は解放されるまで利用出来なくなる．値の割当に成功した場合，関数は true を返す．値が利用できない範囲である場合，および値がすでに利用されている場合，関数は false を返す．

値の解放

- `bool Resign(long id);`
  値 id を解放する．値の解放に成功した場合，関数は true を返す．値が範囲外または割当済みで無い場合，関数は false を返す．

値の割当確認

- `bool IsAssigned(long id);`
  `bool operator [](long);`
  値 id がすでに割り当てられているかどうかを取得する．値がすでに割当済みである場合，関数は true を返す．値が範囲外または割当られていない場合，関数は false を返す．

値の範囲取得

- `long GetMaxID(void);`
  `long GetMinID(void);`
  GetMaxID は割当済み ID の最大値を返す．GetMinID は割当済み ID の最小値を返す．割当済み ID が無い場合 または 値が範囲外である場合，GetMaxID は最小値-1 を，GetMinID は最大値+1 を返す．

割当済み ID の個数の取得

- `unsigned long GetCount(void);`
  `operator const unsigned long(void);`
  すでに割り当てた ID の数を返す．

新規 ID 番号の取得

- `long GetNewID(void);`
  利用可能でもっとも小さい ID 番号を返す．この関数では値の割当は行わない．

ID 値の正当性判定

- `bool IsAvailable(void);`
  `operator const bool(void);`
  次の規約に従い，ID 値が正しく割り当てられているかを判定する．正しい場合は true を，そうでない場合は false を返す．

  割当規約

  1. ID 値は連続でなければならない
  2. ID 値は有効範囲内でなければならない
  3. ID 値は最小値から始まらなければならない

Valient 型

型識別子

- `enum Valient::tagType`
  INTEGER : 整数値
  DOUBLE : 浮動小数点
  STRING : 文字列
  POINTER : ポインタ

コンストラクタ

- `Valient(void);`
  空の Valient 型を生成する.
- `Valient(const Valient &);`
  コピーコンストラクタ.
- `Valient(int);`
  `explicit Valient(unsigned int);`
  `Valient(double);`
  `explicit Valient(char *);`
  `Valient(void *);`
  それぞれ引数を初期値とするコンストラクタである.

値の設定

- `bool Set(int);`
  `bool Set(unsigned int);`
  `bool Set(double);`
  `bool Set(char *);`
  `bool Set(void *);`
  現在保持していいる値を解放し，指定された値を保持する．関数は正しく設定できた場合は true を，そうでない場合は false を返す.

値の取得

- `int GetInteger(int =0);`
  `char * GetString(char * =0l);`
  `double GetDouble(double =0.0);`
  `void * GetPointer(void * =0l);`
  現在保持している値を取得する．関数は保持している型が合わない場合，引数として与えられた値を返す.

値の強制解放

- `bool Clear(void);`
  現在保持している値を強制的に解放する.

値の型の取得

- `enum tagType GetType(void);`
  `operator const enum tagType(void);`
  現在保持している値の型を取得する.

値の型の比較

- `bool operator ==(enum tagType);`
  現在保持している値の型の比較を行う．一致した場合は true，そうでない場合は false を返す.

Array 型

型

- `enum Array::tagType`
  EMPTY : 要素は空である
  ARRAY : 要素は配列 (Array 型) である
  VALUE : 要素は値 (Valient 型) である Q

コンストラクタ

- `Array(void);`
  空の配列を生成する.
- `Array(const Array &);`
  コピーコンストラクタ.

配列要素への値の設定

- ```
  bool Set(unsigned int pos, Valient & val);
  bool Set(unsigned int pos, Valient * val);
  ```

  pos で示される配列要素へ値 val を設定する．pos が既存の配列である場合は値を変更し，無い場合は新たに pos までの配列を自動で生成する．pos 以外の自動生成された要素の中身は空 (Array::EMPTY) である．1 番目はコピーを生成して配列要素に設定する．2 番目はそのものを配列要素として設定し，破棄は Array クラスが行う．注) ポインタ渡しした Valient 型を手動で破棄すると実行時エラーとなる

  ```
  bool Set(unsigned int pos,Array & ary);
  bool Set(unsigned int pos,Array * ary);
  ```

  pos で示される配列要素へ配列 ary を設定する．pos が既存の配列である場合は値を変更し，無い場合は新たに pos までの配列を自動で生成する．pos 以外の自動生成された要素の中身は空 (Array::EMPTY) である．1 番目はコピーを生成して配列要素に設定する．2 番目はそのものを配列要素として設定し，破棄は Array クラスが行う．注) ポインタ渡しした Array 型を手動で破棄すると実行時エラーとなる

Valient 型の配列要素のポインタ取得

- `Valient * GetValuePtr(unsigned int pos);`
  pos で示される配列要素の値へのポインタを取得する．pos が範囲外または pos で示される配列要素が値で無かった場合，関数は (Valient *)0l を返す.

Array 型の配列要素のポインタ取得

- `fArray * GetArrayPtr(unsigned int pos);`
  pos で示される配列要素の配列へのポインタを取得する．pos が範囲外または pos で示される配列要素が配列で無かった場合，関数は (Array *)0l を返す.

配列要素の型の取得

- `tagType GetType(unsigned int pos);`
  pos で示される配列要素の型を返す．pos が範囲外であった場合は Array::EMPTY を返す.

配列要素の値への参照

- `Valient & operator [](unsigned int pos);`
  pos で示される配列要素の値への参照を取得する．注) 本関数は pos が範囲外であるかどうかを確認しない.

配列要素が配列であることの確認
  ```

- `bool IsArray(unsigned int pos);`
  配列要素 pos が配列であった場合は true を返す．そうでない場合や pos が有効範囲外であった場合は false を返す．

配列要素の数の取得

- `unsigned int Size(void);`
  配列要素の数を取得する．

配列要素の全削除

- `bool Clear(void);`
  配列要素を全て破棄する．

Exception

コンストラクタ

- `Exception(void)`
  ”原因不明のエラー”という詳細情報をもつ例外クラスを生成する．
- `Exception(const char * src)`
  `Exception(const std::string & src)`
  引数を詳細情報とする例外クラスを生成する．
- `Exception(const Exception & src)`
  コピーコンストラクタ．

詳細情報の取得

- `virtual std::string & GetReason(void)`
  `operator const char * (void)`
  `operator const std::string & (void)`
  詳細情報を取得する．注) 下 2 つのオペレータは GetReason 関数を実行する．

Manager

API の定義・登録

- `void DeclareAPI(void);`
  `void DeclareAPI(class StaticAPI *);`
  静的 API を定義・登録する．詳細は後述する．

シリアライザの定義・登録

- `void DeclareSerializer(void);`
  `void DeclareSerializer(class Serializer *);`
  シリアライザを定義・登録する．詳細は後述する．

コンストラクタ

- `Manager(class Parser * p);`
  静的 API ファイルパーサ p を利用してクラスを初期化する．パーサ p は Manager が自動で削除する．

プログラム本体

- `bool Body(char * filename);`
  コンフィギュレーションファイル filename に対して処理を行う．