リアルタイムカーネル勉強会

システム構築手順 2 (コンパイル手順、 Makefile)

名古屋市工業研究所
斉藤 直希
saito@nmiri.city.nagoya.jp

平成 13 年 9 月 17 日 (月)

1 はじめに

ここではまずシステムを構築する手順、および構築の際に使用される make というツールについて紹介し、そこで使用される Makefile というファイルの内容について解説する。 make および Makefile を使用することで面倒かつ複雑なコンパイル作業を自動化することが可能になる。

その後、 TOPPERS/JSP カーネル上で動作するサンプルプログラムについて解説する。

2 システム構築作業

2.1 クロス開発

クロス開発とは、開発を行っている環境と異なる環境上で動作するソフトウェアを開発することをいう。クロス開発を行うためにはそのための開発環境を新たに用意する必要がある。

2.2 システム構築に必要なもの (「TOPPERS/JSP カーネル ユーザズマニュアル」の「6.2 開発環境」より引用)

JSP カーネルを用いたシステム構築には，以下のツールが必要である．

ホスト環境用のツール
- 標準規格に準拠した C コンパイラ， C ライブラリ
- 標準規格に準拠した C++ コンパイラ， C++ ライブラリ， STL
- perl （動作確認は 5.005）
- GNU Make （動作確認は 3.77）

クロス環境用のツール
- GNU 開発環境
  - BINUTILS （アセンブラ，リンカなど）
  - GCC または GCC-CORE （C コンパイラ）
  - GDB （デバッガ）
- NEWLIB （標準 C ライブラリ）

ホスト環境用の C コンパイラと C ライブラリは，クロス環境用のツールのインストールに必要になる．また， C++ コンパイラ， C++ ライブラリと STL （Standard Template Library）は，カーネルコンフィギュレータのコンパイルに必要である．

クロス環境用の標準 C ライブラリは，アプリケーションが標準 C ライブラリを使用しない場合には，必須ではない． GNU 開発環境の動作確認バージョンとインストール方法については， GNU 開発環境構築マニュアルを参照すること．

2.3 システム構築作業の手順

1. 開発環境の構築 (以下、 (b) を除き、「GNU 開発環境構築マニュアル」の「1.2 開発環境の構築方法」より引用)

(a) ホスト環境の準備

ホスト上に必要なツールが足りない場合には，あらかじめインストールしておく．具体的には，perl と GNU Make が必要である．さらに，開発環境の構築に使うために，ホスト上にも最新の GCC をインストールしておくことが望ましい．

なお，JSP カーネルの配布キットに含まれる perl スクリプトは，perl のプログラムが /usr/bin/perl にあるものと仮定して記述している．perl のプログラムのパスがこれと異なる場合は，各 perl スクリプトの先頭の perl の絶対パスを修正する必要がある．

(以降は既にインストールされているものとして話を進める)

(b) ツールおよびライブラリのソースファイル取得

以下はツールおよびライブラリのソースコードをダウンロード可能なサイトの一覧である。

- BINUTILS，GCC-CORE，GDB，GNU Make:

  GNU プロジェクト http://www.gnu.org/

  Ring Server http://www.ring.gr.jp/

- NEWLIB:

  Red Hat http://sources.redhat.com/newlib/

  または ftp://sources.redhat.com/pub/newlib/

(c) ソースファイルの展開

BINUTILS，GCC-CORE，GDB，NEWLIB のソースファイルを展開する．以下では，展開により作成されたディレクトリ名をそれぞれ次のように表記する．

<BINUTILS-SRCDIR>	BINUTILS のソースを展開したディレクトリ
<GCC-SRCDIR>	GCC-CORE のソースを展開したディレクトリ
<GDB-SRCDIR>	GDB のソースを展開したディレクトリ
<NEWLIB-SRCDIR>	NEWLIB のソースを展開したディレクトリ

(d) 開発環境構築のためのディレクトリの決定

開発環境を構築するために，以下のディレクトリを用意する．

<PREFIX>	開発環境をインストールするディレクトリ
<BINUTILS-OBJDIR>	BITUTILS のオブジェクトを生成するディレクトリ
<GCC-OBJDIR>	GCC-CORE のオブジェクトを生成するディレクトリ
<GDB-OBJDIR>	GDB のオブジェクトを生成するディレクトリ
<NEWLIB-OBJDIR>	NEWLIB のオブジェクトを生成するディレクトリ

<PREFIX>/bin が実行パスに含まれるようにシェルの設定を行っておく．また，make install は，<PREFIX> 以下に書き込み権限があるユーザで行う必要がある．

(e) ターゲット (ソフトウェアを動作させる対象環境) の選択

ターゲットプロセッサに応じて，ターゲット環境を選択する．具体的には，次の通り．

プロセッサ	ターゲット環境（<TARGET>）
M68040	m68k-unknown-elf
SH1, SH3	sh-hitachi-elf
V850	v850-nec-elf

以下，ターゲット環境を表す文字列を <TARGET> と表記する．

なお，ターゲットによっては，ツールまたはライブラリのソースコードの修正が必要な場合がある．修正内容については，ターゲット毎のマニュアルに記述する．

(f) BINUTILS のインストール（2.10）

BINUTILS は，GCC-CORE のインストールに必要なため，GCC-CORE に先だってインストールする．BINUTILS のインストール手順は次の通り．

```
% mkdir <BINUTILS-OBJDIR>
% cd <BINUTILS-OBJDIR>
% <BINUTILS-SRCDIR>/configure --target=<TARGET> --prefix=<PREFIX>
% make
% make install
```

(g) GCC-CORE のインストール（2.95.2）

NEWLIB のインストールには GCC-CORE が必要なため，GCC-CORE のインストールを先に行う．configure は，newlib のヘッダーファイルを <PREFIX> 以下にインストールするため，<PREFIX> 以下に書き込み権限のあるユーザーで行う必要がある．GCC-CORE のインストール手順は次の通り．

```
% mkdir <GCC-OBJDIR>
% cd <GCC-OBJDIR>
% <GCC-SRCDIR>/configure \
    --target=<TARGET> --prefix=<PREFIX> \
    --with-gnu-as --with-gnu-ld --with-newlib \
    --with-headers=<NEWLIB-SRCDIR>/newlib/libc/include
% make
% make install
```

(h) GDB のインストール（4.182）

次の手順に従って，GDB をインストールする．

```
% mkdir <GDB-OBJDIR>
% cd <GDB-OBJDIR>
% <GDB-SRCDIR>/configure --target=<TARGET> --prefix=<PREFIX>
% make
% make install
```

(i) NEWLIB のインストール（1.8.1）

次の手順に従って，NEWLIB をインストールする．

```
% mkdir <NEWLIB-OBJDIR>
% cd <NEWLIB-OBJDIR>
% <NEWLIB-SRCDIR>/configure --target=<TARGET> --prefix=<PREFIX>
% make
% make install
```

2. カーネルコンフィギュレータのコンパイル (「TOPPERS/JSP カーネル ユーザズマニュアル」の「6.3 カーネルコンフィギュレータのコンパイル」より引用)

最初に，カーネルコンフィギュレータをコンパイルする．コンフィギュレータのディレクトリで，make を実行すればよい．

```
% cd cfg
% make
```

これにより，cfg という名前のコマンドが作られる．

3. Makefile の依存関係定義部およびカーネルイメージ作成

まず (現在コンフィグレータのディレクトリにいると仮定して)、
サンプルプログラムのディレクトリに移動する．

```
% cd ../MS7709ASE01/
```

次に make depend で Makefile の依存関係定義部を作り、

make でサンプルプログラムを作る．

```
% make depend
% make
```

2.4 Makefile の修正 (「GNU 開発環境構築マニュアル」の「6.5 Makefile の修正」より引用)

JSP カーネルを他の環境で動作させる場合や，ユーザのアプリケーションプログラムを構築する場合には，Makefile の修正が必要になる．ここでは，Makefile の中で，修正が必要となる箇所について説明する．

(A) ターゲット名の定義

CPU はターゲットプロセッサの名称，SYS はターゲットシステムの名称に定義する．

(B) Cygwin 上でコンパイルするかどうかの設定

Cygwin 環境でコンパイルする時には，CYGWIN を true に定義する．これは，Cygwin 環境では，オブジェクトプログラムに拡張子 "exe" が付加されるのに対応するためのものである．

(C) C プリプロセッサのコマンド名の定義

C プリプロセッサのコマンド名を CPP に定義する．C プリプロセッサが標準のパスにない場合には，フルパスで定義する必要がある．

(D) 共通コンパイルオプションの定義

全体に共通するコンパイルオプションの追加が必要な場合には，CFLAGS の定義を変更する．そのコンパイルオプションが，特定のターゲットで常に必要な場合には，ターゲット依存の定義を入れた Makefile.config を修正すべきである．追加の可能性のあるコンパイルオプションについては，「6.6 コンパイルオプション」を参照されたい．

(E) アプリケーションプログラムに関する定義

アプリケーションプログラムが一つの C ソースファイル（*.c）のみで構成されている場合には，UTASK にそのファイル名を定義すればよい．アプリケーションプログラムが複数のソースファイルで構成される場合には，UTASK にそのアプリケーション名を定義し，オブジェクトファイル名を UTASK_ASMOBJS および UTASK_COBJS に列挙する．いずれの場合にも，コンフィギュレーションファイルは，UTASK に定義した名前に拡張子 "cfg" を付加した名前とする．

ソースファイルをコンパイルするのとは別のディレクトリに置く場合には，UTASK_DIRS にそのディレクトリを追加する．また，アプリケーションのコンパイルに必要なコンパイルオプションや，アプリケーションがライブラリを必要とする場合には，UTASK_CFLAGS および UTASK_LIBS に定義する．

(F) ターゲットファイルの定義

最終的に作られるオブジェクトファイルの形式を指定する．具体的には，ELF 形式の時は jsp または jsp.exe（Cygwin 環境の時），バイナリ形式の時は jsp.bin，モトローラ S 形式の時は jsp.S を指定する．

(G) カーネルのコンフィギュレーションファイルの生成

ソフトウェア部品のコンフィギュレータを追加する場合には，この規則を修正することが必要である．

3 make

3.1 make の概要

make はコンパイルなどの作業を自動化するツール、またはそのような機能を果たす実行プログラムの名称である。 TOPPERS/JSP カーネルではカーネルの実行イメージ作成作業に GNU make を使用している。

make は不要な再コンパイルを自動的にスキップしたり、複雑なコンパイル手順を自動的に行うことができるので作業時間を短縮したり、作業ミスを減らすことが可能になる。

3.2 Makefile

Makefile とは make に行わせる作業内容を記述したファイルのことである。基本的にそのようなファイルの名称はなんでも良いが (-f オプションで指定できるため)、特にファイルの名前を Makefile または makefile とすると、 make 実行時に (-f オプションによる明示的ファイル指定がなかった場合) 自動的に読みとられる。

makefile と Makefile の両方がカレントディレクトリに存在する場合、 makefile という名前のファイルが最初に検索され、次に Makefile が検索される。

3.3 Makefile の例

3 つのソースファイル prog1.c, prog2.c, prog3.c から実行ファイル prog を作成するとき、以下のようにコンパイルを行うとする。

```
gcc -c prog1.c
gcc -c prog2.c
gcc -c prog3.c
gcc -o prog prog1.o prog2.o prog3.o
```

このような処理を make に行わせるためには Makefile を次のように記述する。ただし、”gcc ...” の行は Tab で始まっている必要があることに注意する。

```
prog: prog1.o prog2.o prog3.o
    gcc -o prog prog1.o prog2.o prog3.o

prog1.o: prog1.c
    gcc -c prog1.c

prog2.o: prog2.c
    gcc -c prog2.c

prog3.o: prog3.c
    gcc -c prog3.c
```

実際に make コマンドを使用して prog を作成するためには”make prog” と実行する。この例では単に”make” としても同様の結果が得られる。

このように一旦構築が済んだ後、 prog1.c だけに変更が加えられた場合、再度”make prog” とすると

```
gcc -c prog1.c
gcc -o prog prog1.o prog2.o prog3.o
```

のように変更が行われたファイルに関連するコンパイル処理のみが実行され、変更されていない prog2.c, prog3.c のコンパイルは実行されない。

3.4 Makefile の基本構成

Makefile は基本的にいくつかのエントリから成り、それぞれのエントリは依存関係行 (ルール行) とコマンド行とから成る。

依存関係行とコマンド行を記述することで何を生成するか、そのためには何が必要か、どのように処理を行えばいいかを指定する。

3.4.1 依存関係行

依存関係行とは依存関係 (後述) を記述する行であり、以下のようにターゲット (後述) とそれが依存するターゲットの間をコロンで区切って記述する。必ずしも以下の形式に従っていないものも存在するが、そのようなものについてはコンパイルルールの項で説明する。

```
target ... : dependencies ...
```

3.4.2 コマンド行

コマンド行とは処理内容を記述する行であり、依存関係行の後に記述して使用する。コマンド行は行頭が必ずタブで始まっている必要がある。

3.4.3 エントリの持つ意味

先程の例でいうと

```
prog: prog1.o prog2.o prog3.o
    gcc -o prog prog1.o prog2.o prog3.o
```

が一組の依存関係行とそれに対するコマンド行のエントリである。

この記述は、次のような意味として解釈される。

「ターゲットは prog であり、 prog は prog1.o と prog2.o と prog3.o とに依存している。 prog を作成するには prog1.o と prog2.o と prog3.o とが必要で prog1.o と prog2.o と prog3.o のいずれかよりも prog が古い場合、コマンド行に書かれている処理を実行する。」

3.4.4 ターゲット

「ターゲット」とは多くの場合、ファイル名のことを指すが、必ずしもファイル名であるとは限らない。ファイル名でない場合、ターゲットはある一連の処理に付けられた処理名のことを指す。

3.4.5 依存関係

「依存関係」とは 2 つのターゲット間の関係のことをいう。ターゲット B で施された変更によりターゲット A を変更する必要がある場合、ターゲット A はターゲット B に依存しているという。

3.5 make の実行

make を実行する時は生成したいターゲットを make の引数として指定し実行する。

（例）`make prog`

引数としてターゲットを指定しない場合、 Makefile の最初のターゲットが指定されたものとして処理される。

上の例のように “make prog” として make が起動された場合 make は以下の手順で実行される。

1. ターゲット prog はターゲットが依存している 3 つのファイル (*.o) が最新のものであるかどうかを調べる。例えば prog1.o が、依存するファイル prog1.c より古い場合、 prog1.o は最新ではない。
2. 依存ファイルが古いファイルであれば全て新しいバージョンを作成する。
3. ターゲットが最新かどうかを調べる。例えば上の 1.,2. の手順で作られた新しい依存ファイルがターゲットよりも新しい場合、ターゲットは最新ではない。
4. 古い場合には、新しいバージョンのターゲットを作成する。

make はターゲットを作成する際に、依存するファイルが最新であることを確認するために再帰的に処理を実行する。ターゲットの作成に必要なファイルが全て最新であることを保証できるまで依存関係を調べ続ける。

(注) ターゲットがファイルとして存在していなかった場合、 make 内部ではターゲットが存在し、かつそれを古いものとみなしているので、ターゲットがファイル名でなくタスク名の場合は常に古いという扱いになる。 (必ずコマンド行は実行される)

3.6 変数の定義と参照

Makefile 中では変数を使用することができる。

3.6.1 変数の定義

変数はどこで定義しても良いが、その変数が依存関係行、コマンド行などで使用される場合、使用される位置よりは前で定義する必要がある。

GNU make の場合、変数には 2 通り存在する。

1. Recursively expanded variables

   [書式] `Variable-NAME = Value`

   - 変数が別の変数を参照する場合、その参照される変数の定義位置が参照する変数の位置より後ろにあっても変数の値が正しく展開される。そのため変数同士の間では変数定義する順番を気にする必要がない。
   - 複数の位置で同じ変数が定義されているような場合、最後に定義された値が割り当てられる。
   - 変数に定義されている値に別の値を追加するような使い方ができない。

2. Simply expanded variables

   [書式]　`Variable-NAME := Value`

   - 変数に定義されている値に別の値を追加するような使い方ができる。
   - 参照される変数が参照する変数よりも後ろにある場合は、未定義の変数として扱われるので、参照する時点では空文字列として扱われる (エラーは出ない)。

3.6.2　変数の参照

- 変数を参照する場合、ドル記号 ($) をマクロ名の前に置き、どこまでが変数名かを明らかにするため中括弧 {} または小括弧 () で囲む。
- シェル変数や環境変数なども変数として参照することができる。
- CC, AS, CPP などあらかじめ定義されている変数もある。 -p オプションを付けて make を起動すれば調べることができる。
- 変数参照時に変数中の特定の部分を別のパターンに置き換えた値として参照する事もできる (次節の例参照)。

3.6.3　例

1. 次のような Makefile を用意した場合、実行すると”echo ABC” が実行される。

```
a = ${b}
b = ABC
all :
    echo ${a}
```

   ここで 1 行目を a := ${b} に変更すると単に ”echo”が実行される。

2. 変数 CFLAGS の末尾にオプションを追加する。ここで := が = ならばエラーとなる。

```
CFLAGS := $(CFLAGS) -m3 -mhitachi -O2 -Wall
```

3. 変数 foo の値の.o の部分を.c に置き換えた値を bar の値として定義する場合。こうすると変数 bar には a.c b.c c.c が変数の値として格納される。

```
foo := a.o b.o c.o
bar := $(foo:.o=.c)
```

3.6.4　条件付き変数定義

1.
```
ifdef arg1
    text_if_true
else
    text_if_false
endif
```

   arg1 が定義されていれば text_if_true の部分が Makefile の一部とみなされる。

   そうでなければ text_if_false が Makefile の一部となる。

2. `ifndef arg1`

   ```
       text_if_true
   else
       text_if_false
   endif
   ```

   arg1 が定義されていなければ text_if_true の部分が Makefile の一部とみなされる。

   そうでなければ text_if_false が Makefile の一部となる。

3. `ifneq (arg1,arg2)`

   ```
       text_if_true
   else
       text_if_false
   endif
   ```

   arg1 と arg2 の内容が等しくなければ text_if_true の部分が Makefile の一部とみなされる。

   そうでなければ text_if_false が Makefile の一部となる。

3.7 include 命令

[書式] `include` ファイル名

指定したファイルを取り込み、取り込んだファイルを Makefile の一部であるかのように扱うための命令である。変数定義やコンパイルルール (後述) 等を 1 ヶ所に集めておく場合などに使われる。

3.8 コンパイルルール

あるターゲットがターゲットとして依存関係行に記述されていないような場合、そのターゲットのパターンによっては特定のルールに従って処理が行われる。そのようなルールのことをコンパイルルールという。

コンパイルルールにはあらかじめ用意されているものがある。既存のルールを変更したり新たにルールを作成したりすることもできる。

最初から用意されているコンパイルルールは make コマンドを -p オプション付きで実行することで知ることができる。

3.8.1 あらかじめ用意されている (既定の) コンパイルルールの例

- C のソース (ファイル名が.c で終るファイル) からオブジェクトファイル (ファイル名が.o で終るファイル) を作成する場合。このときは以下のようにコマンド行が与えられたかのように実行される。

  ```
  $(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -c -o $@ $<
  ```

- TEX ファイルから DVI ファイルを作成する場合

  ```
  $(TEX) $<
  ```

3.8.2 コンパイルルールを定義する方法

コンパイルルールを定義したり、既存のルールを変更したりする方法として、サフィックスルールを用いる方法とパターンルールを用いる方法の 2 つの方法がある。

JSP カーネルソースの Makefile ではパターンルールを用いた方法で行われているのでこちらを説明する。ただし、パターンルールは GNU make に用意されている機能であり、どの make でも用意されているとは限らない。

パターンルールは一見通常の依存関係行と同じだが、 % という記号が使用されている点が異なっている。 % は空文字でない文字に適合し、それ以外の文字はその文字そのものに適合する。簡単なパターンルールの記述例を以下に示す。

```
(例)
%.o : %.c
    $(CC) -c $(CFLAGS) $<
```

この例の場合、接尾子が.o のあらゆるファイル名のファイルがターゲットとして適合し、そのターゲットがターゲットファイル名の.o を.c に変えたファイル名のファイルに依存していることを示す。

ここで、

```
$<
```

は適用される関係依存行の依存ファイルを意味する変数で、これは make であらかじめ用意されている。(上の場合では接尾子が.c のファイル名のファイル)。この変数はコンパイルルールの中だけで使用することができる。

このような変数が必要とされるのは、ファイル名がルール適用の段階で決まるもので、 make 実行前にはファイル名として決定できないためである。

この他にも次のような変数が用意されている。

```
$@
```

ターゲット側のフルネームを表す。普通のコマンド行でも使用可能。

```
$*
```

ターゲット名から接尾子を削除した名前を表す。

コンパイルルールの中だけで使用することができる。

その他については GNU make のマニュアルを参照のこと。

3.8.3 静的なパターンルール (static pattern rules)

例えば、エントリの依存関係行部分が次の様な書式で表現されている場合を考える。

```
$(OBJS): %.o: %.S
```

上の場合、変数 OBJS に格納された値の中で、ファイル名が.o で終る名前のファイルはその接尾子の部分を.S に変更したファイル名のファイルに依存することを表している。

例えばOBJSの値がa.o b.o c.oの場合、OBJSで表されているオブジェクトファイルはそれぞれa.S b.S c.Sというファイルに依存していることを示す。

また、(変数OBJSの値は上と同じとして、)次の様な例ではa.s b.s c.sがそれぞれa.c b.c c.cに依存していることを表している。

```
$(OBJS:.o=.s): %.s: %.c
```

3.9 複数行に渡る場合の記述

物理的に複数行に渡るような場合はバックスラッシュを行末に置くことで一行として解釈される。バックスラッシュの後ろには改行以外のいかなる文字をも追加してはならない。

3.10 エラーによる処理停止の抑制

コマンドの前にハイフン(-)が置かれている場合、そのコマンドにエラーが発生してもmakeが実行を継続することを示す。

(例) -include Makefile.depend

ここでハイフンをとった場合、最初にMakefile.dependが存在しないとエラーが発生し、makeの処理が終了する。

3.11 ファイルのサーチリストの指定

通常makeはカレントディレクトリにあるファイルを探す。

vpath命令はカレントディレクトリにコンポーネントが存在しない場合にmakeが探しにゆくディレクトリのリストを指定するために使用される。

同様の目的を達成するためにVPATHという変数にディレクトリのリストを記述するという方法もあるが、vpath命令はある特定のファイルのパターンに対して探しにゆくディレクトリを指定することができるようになっている。

```
(例) vpath %.c ${SRCDIR}
```

ファイル名が.cで終るファイル名のファイルは変数SRCDIRで定義されているディレクトリの中を探す

4 サンプルアプリケーション

4.1 プログラムの概要

- サンプルアプリケーションはタスク構成として一つのメインタスクと，三つの並列実行タスクとからなる。(スタブを使用する場合はシステムログを出力するタスクも含まれる。)
- メインタスクはシリアルI/Oポートからの文字入力を行い（文字入力を待っている間は，並列実行されるタスクが実行されている）、入力された文字に対応した処理を実行する。Control-Cまたは'Q'が入力されると，プログラムを終了する。

- 並列実行される三つのタスクは，TASK_LOOP 回空ループを実行する度に，タスクが実行中であることを表すメッセージを表示する．
- システムログタスクは，システムログバッファからメッセージ文字を取り出し，シリアルインタフェースドライバを用いて，シリアルポートへ出力する。また、システムログタスクはメッセージを出力後、dly_tsk により 10ms だけ待ち状態に移行する。その間にメインタスクが実行される。メインタスクの中で 3 つの並列実行されるタスクが act_tsk により起動される。
- 上記タスクの他、周期ハンドラを使用している。周期ハンドラ（周期ハンドラ ID: CYCHDR1）は，一定時間 (2 秒) 毎に三つの優先度（HIGH_PRIORITY，MID_PRIORITY，LOW_PRIORITY）のレディキューを回転させる。プログラムの起動直後では周期ハンドラは停止状態になっている．
- プログラム起動直後は、メインタスクと、(WITH_STUB で構築した場合は) システムログタスクが TA_ACT 属性付きで静的 API により生成されるので、タスクが実行できる状態になっている。システムログタスクの方がメインタスクよりも高い優先度に設定されているので (ログタスクがあれば) 先に実行される。

4.2 アプリケーション用標準インクルードファイル

アプリケーションが用いることのできるインクルードファイルは，jsp/include ディクトリの下に置かれている。この中で jsp_services.h は，アプリケーション用の標準インクルードファイルである。このファイル中で jsp_stddef.h、kernel.h、itron.h、serial.h、syslog.h がインクルードされている。それ以外にもアプリケーションで使われる定義が含まれている．

4.3 システムコンフィギュレーションファイル (sample1.cfg)

コンフィギュレーションファイルではプリプロセッサディレクティブおよび静的 API を使用して、システムの構成や初期状態を定義している。

- INCLUDE によるヘッダファイル (sample1.h, 間接的には timer.h, serial.h, logtask.h も) の読み込み
- CRE_TSK によるタスク (メインタスク、3 つの並列実行タスク、ログタスク) の生成
- CRE_SEM によるセマフォの生成 (シリアルポートの入力、出力各 1 つずつ)
- CRE_CYC による周期ハンドラの生成
- DEF_TEX によるタスク例外処理ルーチンの定義
- DEF_EXC による CPU 例外ハンドラの定義 (CPU ロードアドレスエラーハンドラ)
- DEF_INH による割り込みハンドラ (タイマ割り込み、シリアル入出力割り込み) の定義
- ATT_INI による初期化ルーチン (タイマ、シリアルポート) の追加

コンフィギュレーションファイルがプリプロセッサ、コンフィギュレータで処理されて kernel_cfg.c, kernel_id.h が生成される。

5　参考資料

1. μITRON4.0 仕様書
2. GNU 開発環境構築マニュアル gnu_install.txt(JSP カーネル付属)
3. TOPPERS/JSP カーネル ユーザズマニュアル user.txt(JSP カーネル付属)
4. GNUPro ツールキットスタートガイド

   4. は http://www.jp.redhat.com/download/cygnus/doc/ よりダウンロード
5. Getting Started with GNUPro Toolkit
6. GNUPro Compiler Tools
7. GNUPro Development Tools

   5. から 7. は http://www.redhat.com/support/manuals/gnupro.html よりダウンロード
8. 「make」 (O'REILLY)
9. 「GNU ソフトウェアプログラミング」 (O'REILLY)