Dependable *Responsive Multithreaded Processor*

ユーザーズマニュアル

第 3版

平成29 年2 月10 日
慶應義塾大学 理工学部 山﨑研究室

http://www.ny.ics.keio.ac.jp/research/rmt/

目 次

1

DRMTP-I 評価キットの使い方

本章では，*DRMTP-I* (*Dependable Responsive Multithreaded Processor-I*) 評価キットの使い方を説明します．

1.1 準備するもの

- *DRMTP-I* 評価キット（図 1.1）
- 作業用 PC
- 電源ケーブル
  - AC アダプタ 5V（図 1.2）
  - USB ケーブル A-miniB タイプ（図 1.3）
- シリアルケーブル（図 1.4）
  - 図 1.2 では，*DRMTP-I* と接続する方から順に RS232C コネクタ，RS232C ケーブル（ストレート），USB-RS232C コンバータとなっています．

1.2 使用方法

1.2.1 電源の接続

電源は家庭用 AC 電源または USB 給電のどちらかを使うことができます．家庭用電源を用いる場合は AC アダプタ 5V を接続してください．USB 給電を用いる場合は USB ケーブル A-miniB タ

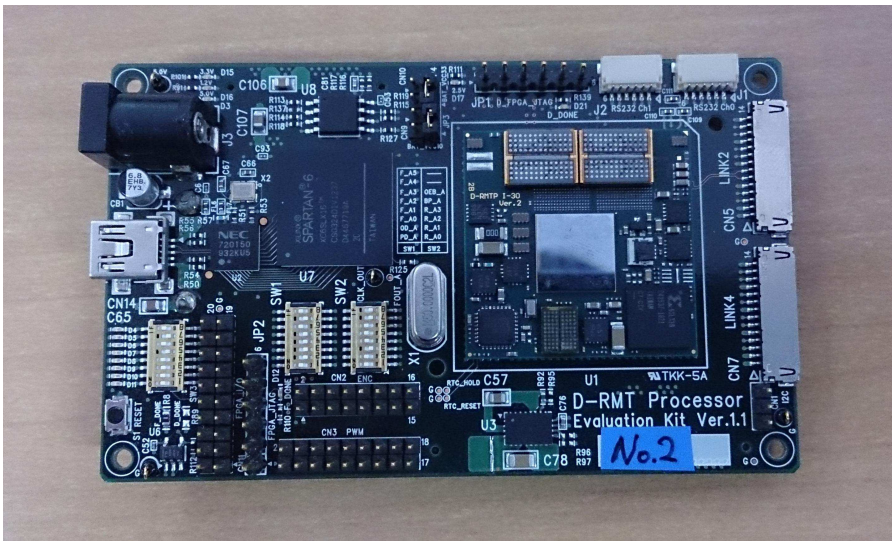


図 1.1: 評価キット

図 1.2: AC アダプタ 5V

図 1.3: USB ケーブル A-miniB タイプ

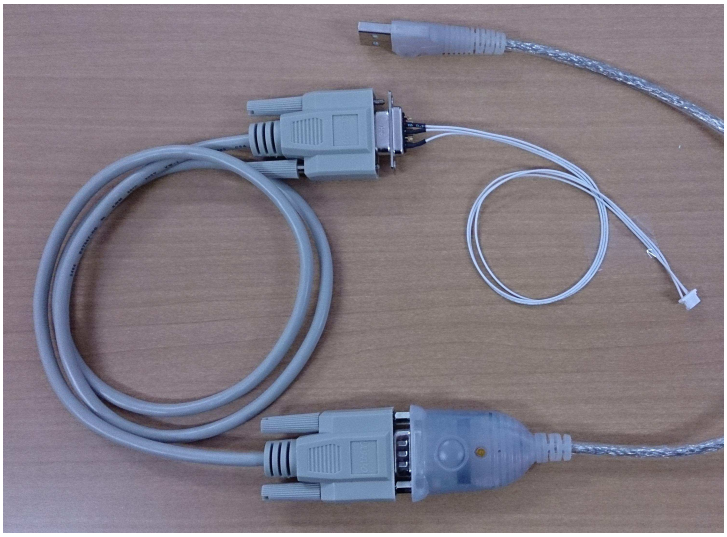


図 1.4: シリアルケーブル

イプを用意し，A 端子を PC に，miniB 端子を評価キットに接続してください（図 1.5 では，USB 給電を用いています）．電源を投入すると LED が点灯します．

1.2.2 シリアルケーブルの接続

シリアルケーブルで *DRMTP-I* と PC を接続します．シリアルポートがどのように認識されるかは環境によって異なりますが，例えば Linux では/dev/ttyS0，/dev/ttyUSB0（USB-シリアル変換ケーブルを通して接続した場合）のようになります．cu コマンドを使う場合の例を以下に示します．-l オプションでデバイス名を，-s オプションでボーレートを指定します．

```
% cu -l /dev/ttyUSB0 -s 115200
```

シリアルケーブル接続後，評価キットに電源を投入してリセットボタンを押してください．ローダからのメッセージがコンソールに出力されれば成功です．

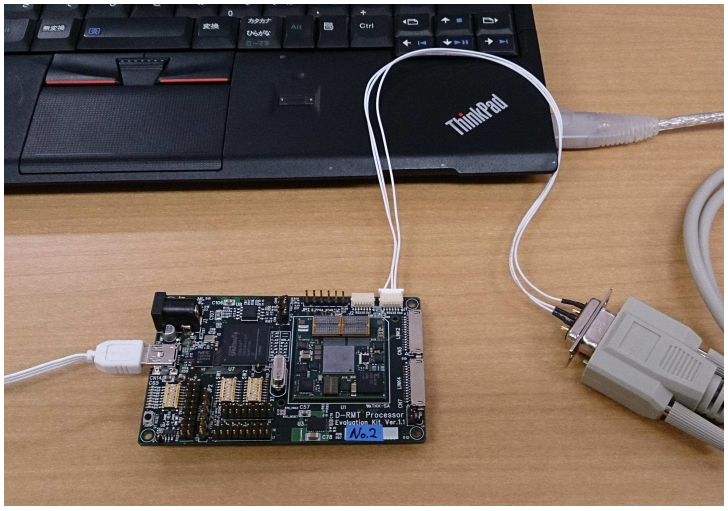


図 1.5: 全体の接続

1.2.3 プログラムの転送

評価キットには，シリアルローダと USB ローダの 2 つが用意されています．この 2 つはディップスイッチ（図 1.6）によって切り替えることができます．シリアルローダと USB ローダのディップスイッチの設定をそれぞれ表 1.1 と表 1.2 に示します．

転送には XMODEM を使用します．ここでは cu コマンドでの転送方法を例に説明します．

まず，プログラムを用意します．プログラムデータの作成方法は他の章で解説しているので参考にしてください．使用するファイルの拡張子は.bin です．

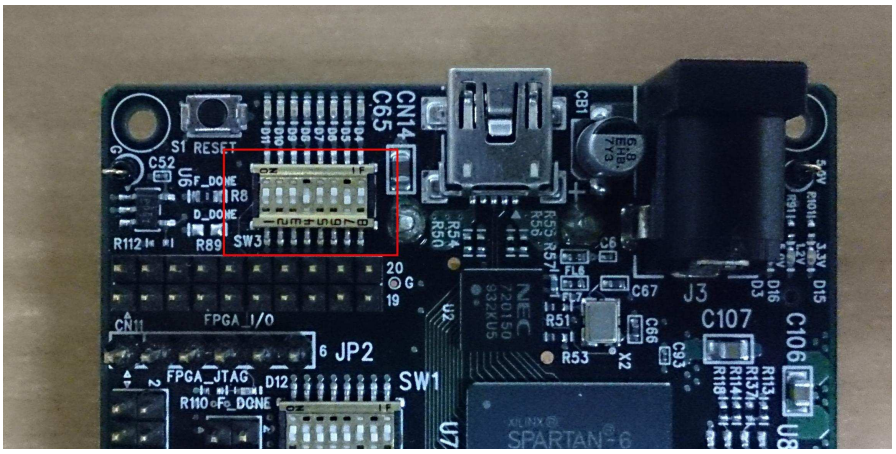


図 1.6: ディップスイッチ

表 1.1: シリアルローダの場合のディップスイッチ

1	2	3	4	5	6	7	8
ON	ON	ON	OFF	ON	ON	OFF	ON

表 1.2: USB ローダの場合のディップスイッチ

1	2	3	4	5	6	7	8
ON	ON	ON	ON	OFF	ON	OFF	ON

cu 上で ～（チルダ）を入力するとコマンドの入力待ち状態になります．ここで以下のように入力し (hoge.bin は転送したいファイル名)，エンターキーを押すと転送開始待ち状態になります．この状態で評価キットのリセットボタンを押すと転送が開始されます．

```
+sx hoge.bin
```

2

RmtSimの使い方

本章では，*RmtSim* の使い方を説明します．*RmtSim* は *RMT Processor* の命令レベルシミュレータであり，実機を使わずに PC 上で *RMT Processor* の命令をエミュレートして動作を確認することができます．また， *RmtSim* 固有のデバッグ機能を使用することもできます．

2.1 ビルド方法

RmtSim のトップディレクトリで以下のコマンドを入力します．gcj-devel 等のパッケージのインストールが必要になる場合があります．make install には root 権限が必要になります．デフォルトのインストール先は /opt/rmtp-sdk/rmtsim/ 以下です．/opt/rmtp-sdk/rmtsim/bin/ にパスを通してください．make install を行わない場合は，rmtsim.bin をパスの通った場所に置いてください．

```
% make
% make install
```

ファイルを指定せずに起動し，以下のようなエラーメッセージが出力されれば成功です．

```
% rmtsim.bin
MIPS/IDT R3052E Simulator - (c) 1996-1997 gfa
RMTSIM - RMTP Simulator (c) 2003-2009 AXE,Inc.
Version 1.52 2009/MAR/14
Configured as trace=false, verbose=false, verbose2=false, debugMode=00000000,
dump=false, doublefault=0, mipsLikeExceptionOnDelaySlot=false
ROM:00000000-001fffff(2MB)
RAM:80000000-83ffffff(64MB)
Not specified ROM image or RAM image
```

2.2 使用方法

以下のように，コマンドライン引数にモトローラ S レコード形式のファイルを指定して起動します．なお，ELF 形式にも対応しています．プログラムデータの作成方法は他の章で解説しているので参考にしてください．モトローラ S レコード形式の拡張子は .srec です．以下は命令レベルシミュレータ用にビルドした *Favor OS* を実行した例です．

```
% rmtsim.bin favor.srec

MIPS/IDT R3052E Simulator - (c) 1996-1997 gfa
RMTSIM - RMTP Simulator (c) 2003-2009 AXE,Inc.
Version 1.52 2009/MAR/14
Configured as trace=false, verbose=false, verbose2=false, debugMode=00000000,
dump=false, doublefault=0, mipsLikeExceptionOnDelaySlot=false
ROM:00000000-001fffff(2MB)
RAM:80000000-83ffffff(64MB)
Loading from '/usr/users/ito/favor/build/favor.srec'

Entry address: 0x80000000
Loading from 'boot.eprom'
ROM not initialized: boot.eprom (No such file or directory)
**************************
***** Favor OS 1.2.0 *****
**************************

!
A
B
C
!
!
A
```

2.3 主なコマンドラインオプション

RmtSim は，起動時にコマンドラインオプションを指定することによって様々な機能を使用することができます．

```
% rmtsim.bin [options...] hoge.srec
```

-t

トレースモード．1 命令ごとに実行した命令を出力する．

-rom size

ROM のサイズを指定する．size は MB 単位で指定する．デフォルトは 2MB.

-ram size

RAM のサイズを指定する．size は MB 単位で指定する．デフォルトは 64MB.

-norl

レスポンシブリンクの一部の機能を無効にする．無効にすることにより，少し実行速度が向上する．

-2

2CPU 構成で起動する．

-4

4CPU 構成で起動する．

2.4 複数プロセッサ実行

-2 または **-4** オプションを指定することで，複数のプロセッサでの同時実行をシミュレーションすることができます．全てののプロセッサのメモリに同じプログラムがロードされます．このとき，CPU ID をメモリアドレス 0xfffe0080 から取得 (32bit; *RmtSim* 固有機能) することができます．CPU ID は 0 から始まります．

```
unsigned long cpu_id = *((unsigned long *)0xfffe0080);
switch (cpu_id) {
case 0:
    ...
case 1:
    ...
...
}
```

2.5 レスポンシブリンクの接続

各 CPU 数でのレスポンシブリンクの接続を表 2.1，表 2.2，表 2.3 に示します．

2.6 RmtSim 拡張機能

RmtSim では，システムレジスタの 0xff 番地に値を書き込むことによって独自の拡張機能を呼び出すことができます．システムレジスタへの書き込みには *mtc0* 命令を使用します．詳細は *RmtSim* のドキュメントを参照してください．

表 2.1: 1CPU での接続

CPU ID	Port		CPU ID	Port
0	OUT0	→	0	DPM
0	OUT1	→	0	IN2
0	OUT2	→	0	IN3
0	OUT3	→	0	IN4
0	OUT4	→	0	IN3

表 2.2: 2CPU での接続

CPU ID	Port		CPU ID	Port
0	OUT0	→	0	DPM
0	OUT1	→	1	IN1
0	OUT2	→	1	IN2
0	OUT3	→	1	IN3
0	OUT4	→	1	IN4
1	OUT0	→	1	IN0
1	OUT1	→	0	IN1
1	OUT2	→	0	IN2
1	OUT3	→	0	IN3
1	OUT4	→	0	IN4

2.7 ドキュメントについて

より詳細なドキュメントは docs/ 以下にあります．make install した場合は /opt/rmtp-sdk/rmtsim/docs/ にインストールされています．

表 2.3: 4CPU での接続

CPU ID	Port		CPU ID	Port	CPU ID	Port		CPU ID	Port
0	OUT0	→	0	DPM	2	OUT0	→	2	DPM
0	OUT1	→	2	IN3	2	OUT1	→	0	IN3
0	OUT2	→	1	IN4	2	OUT2	→	3	IN4
0	OUT3	→	2	IN1	2	OUT3	→	0	IN1
0	OUT4	→	1	IN2	2	OUT4	→	3	IN2
1	OUT0	→	1	DPM	3	OUT0	→	3	DPM
1	OUT1	→	3	IN3	3	OUT1	→	1	IN3
1	OUT2	→	0	IN4	3	OUT2	→	2	IN4
1	OUT3	→	2	IN1	3	OUT3	→	1	IN1
1	OUT4	→	1	IN2	3	OUT4	→	2	IN2

3

クロス開発ツールのビルド方法

本章では，RMTP 用クロス開発環境のビルド及びインストール方法について説明します．本章で必要なファイルはすべて http://www.ny.ics.keio.ac.jp/research/rmt/からダウンロードすることができます．上記のサイトから以下のファイルをダウンロードします[1]．

- GCC-RMT-1.0.5
- BINUTILS-RMT-2.0.0
- NEWLIB-RMT-1.0.1
- OBJ2PRG

RMTP 用クロス開発ツールのプレフィクスは **mips-rmt-elf-** です（例：mips-rmt-elf-gcc）．

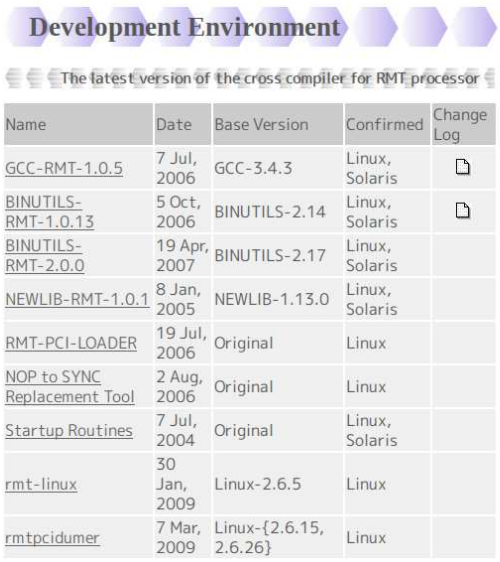
以下の説明では，クロス開発ツールのインストール先が環境変数 PREFIX に設定されているものとし，すべての作業はホームディレクトリ直下の install ディレクトリ内で行うものとします．

```
% set PREFIX = /opt/rmt
% mkdir ~/install
% cd ~/install
```

3.1 binutils のビルド

はじめに，RMTP 用の binutils をビルドします．binutils はバイナリを扱うためのツール群です．主なものを以下に挙げます．

[1] バージョン数字は平成 29 年 2 月 10 日現在のもの．



Development Environment

The latest version of the cross compiler for RMT processor

Name	Date	Base Version	Confirmed	Change Log
GCC-RMT-1.0.5	7 Jul, 2006	GCC-3.4.3	Linux, Solaris	
BINUTILS-RMT-1.0.13	5 Oct, 2006	BINUTILS-2.14	Linux, Solaris	
BINUTILS-RMT-2.0.0	19 Apr, 2007	BINUTILS-2.17	Linux, Solaris	
NEWLIB-RMT-1.0.1	8 Jan, 2005	NEWLIB-1.13.0	Linux, Solaris	
RMT-PCI-LOADER	19 Jul, 2006	Original	Linux	
NOP to SYNC Replacement Tool	2 Aug, 2006	Original	Linux	
Startup Routines	7 Jul, 2004	Original	Linux, Solaris	
rmt-linux	30 Jan, 2009	Linux-2.6.5	Linux	
rmtpcidumer	7 Mar, 2009	Linux-{2.6.15, 2.6.26}	Linux	

図 3.1: クロス開発環境のビルドに必要なファイルのダウンロードページ

mips-rmt-elf-as
　アセンブラ

mips-rmt-elf-ls
　リンカ

mips-rmt-elf-objcopy
　主にフォーマットの変換に使用

mips-rmt-elf-objdump
　逆アセンブル等に使用

ダウンロードした binutils-rmt-xxx.tar.gz（xxx はバージョン数字）を install ディレクトリに移動し，展開します．展開後に生成されたディレクトリに入り，その中で configure スクリプトを実行した後に make, make install を実行します．インストールには管理者権限が必要です．binutils はこれ以降の作業に必要ですので，インストールが完了したら `$PREFIX/bin` にパスを通してください．

```
% tar xzf binutils-rmt-xxx.tar.gz
% cd binutils-xxx
% ./configure \
    --prefix=${PREFIX} \
    --target=mips-rmt-elf \
    --disable-werror
% make
% sudo make install

（パスが通っているかどうかを確認）
% mips-rmt-elf-as --help
% mips-rmt-elf-ld --help
```

3.2 gcc のビルド

次に，C コンパイラ (gcc) をビルドします．gcc のビルドには binutils が必要です．

binutils と同様に，展開して生成されたディレクトリの中で configure スクリプトを実行した後に[2] make, make install を実行します．mips-rmt-elf-gcc は binutils と同じ場所にインストールされます

[2] Windows 7 (64bit) 上の Cygwin (64bit) にインストールする場合，
% ./configure x84_64-unknown-cygwin –prefix=${PREFIX}（以下同様）
でインストールできることを確認しました．

```
% tar xzf gcc-rmt-xxx.tar.gz
% tar xzf newlib-rmt-xxx.tar.gz
% cd gcc-xxx
% ln -s ../newlib-xxx/libgloss .
% ln -s ../newlib-xxx/newlib .
% ./configure \
    --prefix=${PREFIX} \
    --target=mips-rmt-elf \
    --with-newlib \
    --with-gnu-as \
    --with-gnu-ld \
    --enable-languages=c
% make
% sudo make install

(正常にインストールされたかどうかを確認)
% mips-rmt-elf-gcc
mips-rmt-elf-gcc: no input files
```

3.3　obj2prg のビルド

```
% tar xzf obj2prg.tar.gz
% gcc -o mips-rmt-elf-obj2prg obj2prg.c
% sudo mv mips-rmt-elf-obj2prg ${PREFIX}/bin
```

3.4　トラブルシューティング

ビルドに使用する各種ツールのバージョン等によっては，gcc の make でエラーが発生することがあります．以下はその対処例です．

3.4.1　gcc/c-parse.y でエラー

bison ツールで c-parse.y から c-parse.c を生成する際にエラーが発生した場合の対処例です．c-parse.y は c-parse.in から生成されているため，これを修正します．c-parse.in 内を以下のように修正してください．マイナスは修正前の行を，プラスは修正後の行を表します．それぞれ 2 箇所ずつ，計 4 箇所あります．

```
- $$ = start_struct
+ $<ttype>$ = start_struct
```

```
- $$ = start_enum
+ $<ttype>$ = start_enum
```

3.4.2 gcc/collect2.c でエラー

collect2.c でコンパイルエラーが発生した場合，以下のように open 関数の引数を追加してください．マイナスは修正前の行を，プラスは修正後の行を表します．

```
- redir_handle = open (redir, O_WRONLY | O_TRUNC | O_CREAT);
+ redir_handle = open (redir, O_WRONLY | O_TRUNC | O_CREAT, 0755);
```

4

Favor OS の使い方

本章では，*Favor OS* の使い方を説明します．*Favor OS* は山崎研究室オリジナルのリアルタイム OS です．

4.1 ビルド方法

まず，トップディレクトリ直下（ここでは favor ディレクトリとします）にある app ディレクトリ内で main.c を作成します．次に，ビルドの設定を configure を用いて行った後，make コマンドを実行します．*Favor OS* では 2 つの configure を実行する必要があります．それぞれアーキテクチャ依存（*RMT Processor* 特有）部分と OS 依存部分であり，この順序で設定します．

```
（共通）
% cd favor
% ls
Common.make    Makefile  app/   build/      doc/      include/  lib/      tool/
CompileOption  README    arch/  configure*  drivers/  kernel/   scripts/
% vim app/main.c

（DRMTP-I 評価キットの場合）
% ./arch/rmt/configure --cpu=drmtp \
                       --mode=pciboot \
                       --memory=sdram \
                       --serial=fpga_kit

（命令レベルシミュレータ RmtSim の場合）
% ./arch/rmt/configure --cpu=drmtp \
                       --mode=ils \
                       --memory=sdram

（共通）
% ./configure --tick=10000 --sched=fp
% make
```

アーキテクチャ依存のオプション

mode オプションには評価キット用の場合は **pciboot** [1]，命令レベルシミュレータ用の場合は **ils** を指定してください．

RmtSim ではキャッシュ機能をサポートしていないので，cache オプションは **off** を指定してください．*RmtSim* がサポートしていない機能についてはドキュメントを参照してください[2]．

その他のオプションについては favor/arch/rmt/README を参照してください．

OS 依存のオプション

tick オプションにはリアルタイムスケジューラが呼び出される間隔 (μs 単位) を指定します．値を小さくするほどより高精度のスケジューリングを行えますが，カーネルのオーバーヘッドも大きくなります．デフォルトは **10000** ($= 10ms$) です．

sched オプションにはスケジューリングアルゴリズムを指定します．現在 *Favor OS* がサポートしているアルゴリズムは固定優先度 (**fp**)，Earliest Deadline First (**edf**)，Rate Monotonic (**rm**)

[1] シリアルブートおよび USB ブートのどちらも **pciboot** を使います．この名前は歴史的な理由によります．
[2] favor/tool/RMTSIM/rmtsim.1.html にあります．

です．デフォルトは **fp** です．

その他のオプションについては favor/README を参照してください．

4.2 生成されるファイル

make が成功すると，結果は build ディレクトリに出力されます．主なものを表 4.1 に示します．トップディレクトリで make clean を行うと，中間生成ファイルを含むこれらのファイルを削除することができます．

表 4.1: *Favor OS* の出力ファイル

ファイル	説明
favor.bin	評価キットへロードするファイル
favor.srec	命令レベルシミュレータ用のファイル
favor.dmp	逆アセンブルリスト
favor.map	マップファイル（リンカの動作ログ）

4.3 アプリケーションコードの記述

アプリケーションコードは app ディレクトリ内に置きます．*Favor OS* カーネルとアプリケーションは同時にビルドされます．

以下に app/Makefile の内容を示します．アプリケーションコードに合わせて main.o の部分を追加，修正してください（任意の xxx.o は xxx.c に依存し，mips-rmt-elf-gcc で生成されるよう Common.make に記述されています）．

```
TOPDIR  = ..

.PHONY: all
# ここを追加，修正
all: main.o
include $(TOPDIR)/Common.make

clean:
    @rm -fr *.d *~
```

4.4 Favor API

4.4.1 システム制御

create_task_rt

```
long create_task_rt(int priority,
                    unsigned long period,
                    unsigned long wcet,
                    void (*entry)(void));

/* entry に渡す関数の形式*/
void task_function(void);
```

スケジューラ（ソフトウェア）によって管理される周期リアルタイムタスクを生成します．

priority
: タスクの優先度．0 から 63 の 64 段階の設定が可能で，0 が最高優先度．この優先度が反映されるのは固定優先度スケジューリング (**fp**) の場合に限ります．

period
: タスクの周期．単位は μs．tick の倍数でないと正しく反映されない場合があります．

wcet
: タスクの最悪実行時間．単位は μs．

entry
: タスクとして実行する関数へのポインタ．

返り値
: タスク ID．タスク生成に失敗すると 0 が返されます．

create_task_hw

```
long create_task_hw(int priority,
                    unsigned long period,
                    unsigned long wcet,
                    void (*entry)(void));

/* entry に渡す関数の形式*/
void task_function(void);
```

スケジューラとは独立して自身のタイマ割り込み（ハードウェア）で起床を繰り返す周期リアルタイムタスクを生成します．favor/configure で指定した論理プロセッサ数 (**cores**) とは別にスレッドが生成されますので，論理プロセッサ数とこの API で生成したタスク数の合計が 7 以下でないと誤動作を起こす可能性が有ります．また，スケジューラに管理されないため，オーバランのハンドリングはできません．

priority
: タスクの優先度．0 から 63 の 64 段階の設定が可能で，0 が最高優先度．

period
: タスクの周期．単位は μs．tick の倍数でなくても構いません．

wcet
: タスクの最悪実行時間．単位は μs．

entry
: タスクとして実行する関数へのポインタ．

返り値
: タスク ID．タスク生成に失敗すると 0 が返されます．

create_task

```
long create_task(void (*entry)(void));

/* entry に渡す関数の形式 */
void task_function(void);
```

通常のタスクを生成します．

entry
: タスクとして実行する関数へのポインタ．

返り値
: タスク ID．タスク生成に失敗すると 0 が返されます．

system_run

```
void system_run(void);
```

タスクスケジューリングを開始します．リアルタイムスケジューラによって選ばれたタスクから実行が開始されます．

wait_period

```
void wait_period(void);
```

タスクが 1 周期分の実行を完了したことをシステムに通知します．create_task_rt で生成した周期タスク内から呼び出してください．wait_period を呼び出した関数は待ち状態に遷移し，次の周期が来たときにスケジューラによって実行可能状態に遷移させられます．

hw_scheduler_wait_period

```
void hw_scheduler_wait_period(void);
```

タスクが 1 周期分の実行を完了したことをシステムに通知します．周期タスク内から呼び出してください．create_task_hw で生成した周期タスク内から呼び出してください．

delete_task

```
void delete_task(tid_t id);
```

タスクを削除します．

id
: タスク ID.

set_priority

```
void set_priority(tid_t id,
                  int priority);
```

タスクの優先度を変更します．

id
: タスク ID.

priority
: タスクの優先度．0 から 63 の 64 段階の設定が可能で，0 が最高優先度．この優先度が反映されるのは固定優先度スケジューリング (**fp**) の場合に限ります．

set_period

```
void set_period(tid_t id,
                unsigned long period);
```

タスクの周期を変更します．

id
: タスク ID．

period
: タスクの周期．単位は μs．tick の倍数でないと正しく反映されない場合があります．

current_count

```
unsigned long current_count();
```

システムタイマのクロックカウントの下位 32bit を取得します．

返り値
: システムタイマの下位 32bit．

current_count_high

```
unsigned long current_count_high();
```

システムタイマのクロックカウントの上位 32bit を取得します．

返り値
: システムタイマの上位 32bit．

current_time

```
unsigned long long current_time(void);
```

システムタイマのクロックカウンタを取得します．

返り値
: 64bit のシステムタイマのカウント値．

mfc0

```
unsigned long mfc0(unsigned long addr);
```

スレッド制御レジスタの値を取得します.

addr
: スレッド制御レジスタ番号.

返り値
: スレッド制御レジスタの値.

mtc0

```
void mtc0(unsigned long data,
          unsigned long addr);
```

スレッド制御レジスタの値を変更します.

data
: スレッド制御レジスタに書き込むデータ.

addr
: スレッド制御レジスタ番号.

getcid

```
unsigned long getcid(tid);
```

コンテキスト ID を取得します.

tid
: タスク ID.

返り値
: コンテキスト ID.

this_cid

```
unsigned long this_cid();
```

呼び出したタスクのコンテキスト ID を取得します.

返り値
: コンテキスト ID.

getotid

```
unsigned long getotid(void);
```

呼び出したタスクのスレッド ID を取得します.

返り値
スレッド ID.

thread_state

```
int thread_state(int tid);
```

Thread Status レジスタの値を取得します.

tid
タスク ID.

返り値
Thread Status レジスタの値.

set_status_reg

```
void set_status_reg(int value,
                    int th);
```

Thread Status レジスタへ書き込みます.

value
Thread Status レジスタへ書き込む値.

th
スレッド ID.

read_rmt_prio

```
int read_rmt_prio(int tid);
```

RMTP 固有機能であるハードウェア上の優先度を取得します.

tid
タスク ID.

返り値
ハードウェアの優先度.

setup_hw_timer

```
void setup_hw_timer(void);
```

CPU タイマを初期化します.

start_hw_timer

```
uint_t start_hw_timer(int cid,
                      unsigned long count,
                      uint_t type,
                      void (*func)(void));
```

CPU タイマを開始します.

cid
: コンテキスト ID.

count
: タイマの設定値.

type
: タイマの設定. 0 を指定すると 1 回だけタイマが起動します. 1 を指定すると繰り返しタイマが起動します.

func
: タイマ割込み発生時の例外ハンドラ.

stop_hw_timer

```
uint_t stop_hw_timer(int cid);
```

CPU タイマを停止します.

cid
: コンテキスト ID.

timer_int_clear

```
void timer_int_clear(int cid);
```

CPU タイマ割込みをクリアします.

cid
: コンテキスト ID.

set_timeout

```
void set_timeout(int cid,
                 unsigned long time,
                 void (*func)(void));
```

タイムアウト時間を設定します.

cid
: コンテキスト ID.

time
: タイムアウト時間.

func
: タイマ割込み発生時の例外ハンドラ.

set_intr_handler

```
int set_intr_handler(unsigned long irq,
                     void (*handler)(void));
```

割込みハンドラを設定します.

irq
: ハンドリングする割込みチャネル.

handler
: 割込みハンドラのポインタ.

返り値
: 割込みハンドラの設定に成功したら 1 を返します.

set_intr_sub_handler

```
int set_intr_sub_handler(unsigned long irq,
                         void (*handler)(void));
```

Sub IRC の割込みハンドラを設定します.

irq
: ハンドリングする Sub IRC の割込みチャネル.

handler
: 割込みハンドラのポインタ.

返り値
: 割込みハンドラの設定に成功したら 1 を返します.

cache_off_all

```
void cache_off_all(void);
```

全てのスレッドでキャッシュを無効にします.

cache_on_all

```
void cache_on_all(void);
```

全てのスレッドでキャッシュを有効にします.

cache_off

```
void cache_off(unsigned int cid);
```

キャッシュを無効にします.

cid
: コンテキスト ID.

cache_on

```
void cache_on(unsigned int cid);
```

キャッシュを有効にします.

cid
: コンテキスト ID.

flush_i_cache

```
void sys_flush_i_cache(void);
```

命令キャッシュの内容をフラッシュします.

flush_d_cache

```
void sys_flush_d_cache(void);
```

データキャッシュの内容をフラッシュします.

4.4.2 NAND フラッシュ

flash_chip_erase

```
unsigned long flash_chip_erase( void )
```

NAND フラッシュに chip erase コマンドを発行します.

返り値
: 常に 0.

flash_block_erase

```
unsigned long flash_block_erase( unsigned long blk )
```

NAND フラッシュに block erase コマンドを発行します.

blk
: 消去するブロックアドレス.

返り値
: 常に 0.

flash_write_to_buffer

```
unsigned long flash_write_to_buffer(unsigned long addr,
                                    unsigned long word);
```

NAND フラッシュに書き込みます.

addr
　書き込み先のアドレス.

word
　書き込むデータ.

返り値
　常に 0.

flash_auto_write_enable

```
void flash_auto_write_enable( void );
```

NAND フラッシュの自動書き込みモードを ON にします.

4.4.3 Responsive Link

DPLL_ALL

```
unsigned long DPLL_ALL(unsigned int e1,
                       unsigned int e2,
                       unsigned int e3,
                       unsigned int e4,
                       unsigned int d1,
                       unsigned int d2,
                       unsigned int d3,
                       unsigned int d4);
```

DPLL の設定値を作成します.

e1
　イベントリンク 1 番ポートの DPLL 設定値

e2
　イベントリンク 2 番ポートの DPLL 設定値

e3
　イベントリンク 3 番ポートの DPLL 設定値

e4
　イベントリンク 4 番ポートの DPLL 設定値

d1
　データリンク 1 番ポートの DPLL 設定値

d2
　データリンク 2 番ポートの DPLL 設定値

d3
　データリンク 3 番ポートの DPLL 設定値

d4
　データリンク 4 番ポートの DPLL 設定値

返り値
　DPLL の設定値.

MAKE_FER

```
unsigned long MAKE_FER(unsigned int blk_ecc,
                       unsigned int ecc,
                       unsigned int linecode);
```

Responsive Link のコーデック設定値を作成します.

blk_ecc
　ブロックエラー訂正の設定.

ecc
　bit エラー訂正の設定.

linecode
　ラインコードの設定.

返り値
　コーデックの設定値.

MAKE_FER_4PORT

```
void MAKE_FER_4PORT(unsigned int p1,
                    unsigned int p2,
                    unsigned int p3,
                    unsigned int p4);
```

4 ポート分の Responsive Link のコーデック設定値を作成します.

p1
: 1 番ポートのコーデック設定値.

p2
: 2 番ポートのコーデック設定値.

p3
: 3 番ポートのコーデック設定値.

p4
: 4 番ポートのコーデック設定値.

返り値
: 4 ポート分のコーデック設定値.

link_init

```
void link_init(unsigned long para_flag,
               unsigned long dpll,
               unsigned long event_fer,
               unsigned long data_fer,
               unsigned long switch_mode);

/* 使用例 */
link_init(
        RL_SERIAL_MODE,
        DPLL_ALL(MODE32, MODE16, MODE8, MODE4,
                 MODE2, MODE4, MODE8, MODE16),
  MAKE_FER_4PORT(MAKE_FER(REED_SOLOMON, NO_ECC, LC_8B10B),
         MAKE_FER(NO_BLOCK_ECC, NO_ECC, LC_4B10B),
         MAKE_FER(NO_BLOCK_ECC, HAM, BS_NRZI),
         MAKE_FER(NO_BLOCK_ECC, BCH, LC_8B10B)),
  MAKE_FER_4PORT(MAKE_FER(NO_BLOCK_ECC, BCH, BS_NRZI),
                       MAKE_FER(REED_SOLOMON, NO_ECC, LC_4B10B),
         MAKE_FER(NO_BLOCK_ECC, HAM, BS_NRZI),
         MAKE_FER(NO_BLOCK_ECC, NO_ECC, LC_8B10B)),
  STORE_FORWARD);
```

Responsive Link の初期設定を一括して行います.

para_flag
: パラレルモード/シリアルモードの設定.

dpll
: 4 ポート分の DPLL の設定.

event_fer
: 4 ポート分のイベントリンクのコーデック設定.

data_fer
: 4 ポート分のデータリンクのコーデック設定.

switch_mode
: パケットスイッチ方式.

link_set_dpll

```
void link_set_dpll(unsigned long dpll);
```

DPLL の設定を行います.

dpll
4 ポート分の DPLL 設定値.

link_set_codec_data/link_set_codec_event

```
void link_set_codec_data(unsigned long fer);
void link_set_codec_event(unsigned long fer);
```

コーデックを設定します.

fer
4 ポート分のコーデック設定値.

link_sdram_init

```
void link_sdram_init(unsigned long memsize);
```

追い越しバッファから溢れた Responsive Link のパケットを SDRAM に退避する際に，退避用バッファとして使用するメモリサイズを設定します.

memsize
使用する SDRAM の容量.

link_stop

```
void link_stop(void);
```

Responsive Link の動作を停止します.

link_start

```
void link_start(void);
```

Responsive Link の動作を開始します.

link_switch_initialize

```
void link_switch_initialize(void);
```

Responsive Link のスイッチを初期化します.

link_encoder_reset/link_encoder_reset_port

```
void link_encoder_reset(void);
void link_encoder_reset_port(unsigned long ep,
                             unsigned long dp);
```

Responsive Link のエンコーダを初期化します.

ep
: 初期化するイベントリンクのポート

dp
: 初期化するデータリンクのポート

link_input_dpm_reset

```
void link_input_dpm_reset(void);
```

Responsive Link のデュアルポートメモリを初期化します.

link_decoder_reset/link_decoder_reset_port

```
void link_decoder_reset(void);
void link_decoder_reset_port(unsigned long ep,
                             unsigned long dp);
```

Responsive Link のデコーダを初期化します.

ep
: 初期化するイベントリンクのポート

dp
: 初期化するデータリンクのポート

link_interruption_clear_offline

```
void link_interruption_clear_offline(void);
```

Responsive Link のオンライン割込み/オフライン割込みをクリアする.

add_rtable_entry

```
void add_rtable_entry(unsigned long src,
                      unsigned long dst,
                      unsigned long cur_prio,
                      unsigned long chpr,
                      unsigned long new_prio,
                      unsigned long outport,
                      unsigned long type,
                      unsigned long index);
```

Responsive Link のルーティングテーブルを追加します.

src
: 送信元アドレス.

dst
: 送信先アドレス.

cur_prio
: 本エントリが有効なパケットの優先度. Responsive Link のパケットは送信元アドレスと送信先アドレス, パケットの優先度が一致したルーティングテーブルのエントリを参照します. ただし, 一致するエントリが存在しない場合は優先度 0 のエントリを参照します.

chpr
: ホップ時にパケットの優先度を変更するかどうかの指定.

new_prio
: ホップ時にパケットの優先度を変更する場合に新たに設定する優先度.

outport
: パケットを出力するポート.

type
: イベントリンク/データリンクどちらで有効なエントリかの指定. 両方の通信リンクで有効にすることも可能です.

index
: ルーティングテーブルのインデックス.

remove_rtable_entry

```
void remove_rtable_entry(unsigned long index);
```

Responsive Link のルーティングテーブルを削除します.

index
削除するルーティングテーブルのインデックス.

rtable_init

```
void rtable_init(void);
```

Responsive Link のルーティングテーブルを初期化します.

getDPMCtrlVal

```
unsigned long getDPMCtrlVal(unsigned long from_addr,
                            unsigned long to_addr,
                            unsigned long dpm_mode,
                            unsigned long dreq,
                            unsigned long intr);
```

DPM の制御レジスタの設定値を作成します.

from_addr
DMA 転送を開始するアドレス.

to_addr
DMA 転送を開始するアドレス.

dpm_mode
DPM の使用モード.

dreq
DMA リクエストの on/off.

intr
割込みの on/off.

setEventOutCtrl/setDataOutCtrl/setEventInCtrl/setDataInCtrl

```
void setEventOutCtrl(unsigned long val);
void setDataOutCtrl(unsigned long val);
void setEventInCtrl(unsigned long val);
void setDataInCtrl(unsigned long val);

/* 使用例 */
setEventOutCtrl(
        getDPMCtrlVal(PKTNUM_TO_MODE0EVENT_ADDR(0),
                      PKTNUM_TO_MODE0EVENT_ADDR(1),
                      MODE0, DREQ_ON, DPM_INTR_OFF));
```

DPM の制御レジスタを設定します.

val
: DPM 制御レジスタに書き込む値.

getEventInPacketnum/getDataInPacketnum

```
unsigned long getEventInPacketnum(void);
unsigned long getDataInPacketnum(void);
```

Responsive Link の受信パケット数.

返り値
: 受信パケット数.

getEventOutPacketnum/getDataOutPacketnum

```
unsigned long getEventOutPacketnum(void);
unsigned long getDataOutPacketnum(void);
```

Responsive Link の送信パケット数.

返り値
: 送信パケット数.

link_initialize/link_initialize_channel

```
void link_initialize(void);
void link_initialize_channel(unsigned long value);
```

Responsive Link の各種機能を初期化する．初期化する機能はデュアルポートメモリ，デコーダ，エンコーダ，スイッチです．

value
　初期化するポート．

setEventInDMA/setDataInDMA/setEventOutDMA/setDataOutDMA

```
void setEventInDMA(unsigned long src,
                   unsigned long dst,
                   unsigned long length,
                   unsigned long flag);
void setDataInDMA(unsigned long src,
                  unsigned long dst,
                  unsigned long length,
                  unsigned long flag);
void setEventOutDMA(unsigned long src,
                    unsigned long dst,
                    unsigned long length,
                    unsigned long flag);
void setDataOutDMA(unsigned long src,
                   unsigned long dst,
                   unsigned long length,
                   unsigned long flag);
```

Responsive Link 用の DMA の設定を行います．本 API は設定のみ行うため，本 API を実行しても DMA 転送は行われません．

src
　転送元アドレス．

dst
　転送先アドレス．

length
　転送バイト数．

flag
: DMA 転送のモード．何も指定しない場合，Responsive Link モードで DMA 転送する設定が行われます．

read_offline_reg/read_offline_reg_hr

```
void read_offline_reg(void);
void read_offline_reg_hr(void);
```

Responsive Link のオンラインレジスタの内容を表示します．

sendDataPacket/sendEventPacket

```
void sendDataPacket(unsigned long intr_flag,
                    unsigned long src,
                    unsigned long dst,
                    unsigned long prio);
void sendEventPacket(unsigned long intr_flag,
                     unsigned long src,
                     unsigned long dst,
                     unsigned long prio);
```

Responsive Link のテストパケットを送信します．

intr_flag
: パケット受信時に割込みを発生させるかどうかの指定．

src
: 送信元アドレス．

dst
: 送信先アドレス．

prio
: パケットの優先度．

get_rl_baudrate_port

```
unsigned long get_rl_baudrate_port(unsigned long link_type,
                                   unsigned long port);
```

Responsive Link の通信ボーレートを取得する

link_type
: 通信リンク．

port
: 通信ポート．

返り値
: 通信ボーレート．単位は *bps*

get_rl_speed_port

```
unsigned long get_rl_speed_port(unsigned long link_type,
                                unsigned long port);
```

Responsive Link の通信速度を取得する

link_type
: 通信リンク．

port
: 通信ポート．

返り値
: 通信速度．単位は *bps*

get_rl_throughput_port

```
unsigned long get_rl_throughput_port(unsigned long link_type,
                                     unsigned long port);
```

Responsive Link の実効スループットを取得する

link_type
: 通信リンク．

port
通信ポート．

返り値
実効スループット．単位は *bps*

rl_console

```
void rl_console(void);
```

Responsive Link のソフトウェアデバッグ用コンソール．

init_link_intr

```
void init_link_intr(void);
```

Responsive Link の割込みハンドラを初期化します．Responsive Link の割込みをハンドリングする場合は必ずユーザ領域で呼び出してください．

set_link_intr_handler

```
int set_link_intr_handler(unsigned long irq,
                          void (*handler)(void));
```

Responsive Link の割込みハンドラを設定します．Responsive Link は内部に複数の割込み源を持っており，それぞれにハンドリング方法を設定できます．

irq
Responsive Link 内の割込みチャネル番号．

handler
ハンドラとして実行する関数へのポインタ．

返り値
ハンドラの追加に成功すると 1，失敗すると 0 が返されます．

clear_link_intr_handler

```
int clear_link_intr_handler(unsigned long irq);
```

Responsive Link の割込みハンドラを削除します.

irq
: Responsive Link 内の割込みチャネル番号.

返り値
: ハンドラの削除に成功すると 1，失敗すると 0 が返されます.

4.4.4 SPI

spi_set_slave_control

```
void spi_set_slave_control(unsigned long ch,
                           unsigned long slave_num);
```

SPI のスレーブを選択する.

ch
: SPI のチャネル番号．D-RMTPI は SPI を 2 チャネル内蔵しています.

slave_num
: SPI のスレーブ番号．D-RMTPI の SPI は各チャネルに 4 個のスレーブが存在します.

spi_set_fifo_control_enable

```
void spi_set_fifo_control_enable(unsigned long ch,
                                 unsigned long bit);
```

SPI の FIFO Control レジスタに 1 を設定します.

ch
: SPI のチャネル番号.

bit
: FIFO Control レジスタに 1 を書き込む bit 番号.

spi_set_fifo_control_disable

```
void spi_set_fifo_control_disable(unsigned long ch,
                                  unsigned long bit);
```

SPI の FIFO Control レジスタに 0 を設定します.

ch
: SPI のチャネル番号.

bit
: FIFO Control レジスタに 0 を書き込む bit 番号.

spi_read_fifo_status

```
unsigned long spi_read_fifo_status(unsigned long ch);
```

SPI の FIFO Status レジスタを取得します.

ch
: SPI のチャネル番号.

返り値
: FIFO Status レジスタの値.

spi_write_fifo/spi_put

```
void spi_write_fifo(unsigned long ch,
                    unsigned long data);
void spi_put(unsigned long ch,
             unsigned long data);
```

SPI の FIFO に書き込みます.

ch
: SPI のチャネル番号.

data
: FIFO に書き込むデータ

spi_read_fifo/spi_get

```
unsigned long spi_read_fifo(unsigned long ch);
unsigned long spi_get(unsigned long ch);
```

SPI の FIFO からデータを取得します.

ch
: SPI のチャネル番号.

返り値
: FIFO から読み出した値

spi_interrupt_bit_clear

```
void spi_interrupt_bit_clear(unsigned long ch,
                             unsigned long bit);
```

SPI の割込みをクリアします.

ch
: SPI のチャネル番号.

bit
: SPI の割込みをクリアする bit 位置.

spi_set_interval

```
void spi_set_interval(unsigned long ch,
                      unsigned long interval);
```

SPI のスレーブに対する一連のアクセスが終了した後の待ち時間を設定します.

ch
: SPI のチャネル番号.

interval
: SPI の待ち時間.

spi_read_interval

```
unsigned long spi_read_interval(unsigned long ch);
```

SPI のスレーブに対する一連のアクセスが終了した後の待ち時間を読み出します.

ch
: SPI のチャネル番号.

返り値
: 設定されている SPI の待ち時間.

spi_set_clock_ratio

```
void spi_set_clock_ratio(unsigned long ch,
                         unsigned long slave_num,
                         unsigned long ratio);
```

同期クロックで出力するクロックの分周率を設定します.

ch
: SPI のチャネル番号.

slave_num
: SPI のスレーブ番号.

ratio
: クロックの分周率．実際には指定した値の 2 倍で分周されます．また，0 を指定した場合は $2^{22} \times 2$ 分周されます.

spi_set_haol

```
void spi_set_haol(unsigned long ch,
                  unsigned long slave_num,
                  unsigned long haol);
```

SPI の動作モードを指定します.

ch
: SPI のチャネル番号.

slave_num
SPI のスレーブ番号.

haol
SPI の動作モード

spi_set_size

```
void spi_set_size(unsigned long ch,
                  unsigned long slave_num,
                  unsigned long size);
```

SPI のデータ転送サイズを設定します.

ch
SPI のチャネル番号.

slave_num
SPI のスレーブ番号.

size
SPI のデータ転送サイズ．実際には指定した値+1bit が転送されます.

spi_set_lsb

```
void spi_set_lsb(unsigned long ch,
                 unsigned long slave_num);
```

SPI の転送を LSB から行うように設定します.

ch
SPI のチャネル番号.

slave_num
SPI のスレーブ番号.

spi_set_msb

```
void spi_set_msb(unsigned long ch,
                 unsigned long slave_num);
```

SPI の転送を MSB から行うように設定します.

ch
SPI のチャネル番号.

slave_num
SPI のスレーブ番号.

spi_set_read_disable

```
void spi_set_read_disable(unsigned long ch,
                          unsigned long slave_num);
```

SPI を外部からデータを読み込まない設定にします.

ch
SPI のチャネル番号.

slave_num
SPI のスレーブ番号.

spi_set_read_enable

```
void spi_set_read_enable(unsigned long ch,
                         unsigned long slave_num);
```

SPI を外部からデータを読み込む設定にします.

ch
SPI のチャネル番号.

slave_num
SPI のスレーブ番号.

spi_set_write_disable

```
void spi_set_write_disable(unsigned long ch,
                           unsigned long slave_num);
```

SPI を外部へデータを書き込まない設定にします.

ch
SPI のチャネル番号.

slave_num
SPI のスレーブ番号.

spi_set_write_enable

```
void spi_set_write_enable(unsigned long ch,
                          unsigned long slave_num);
```

SPI を外部へデータを書き込む設定にします.

ch
: SPI のチャネル番号.

slave_num
: SPI のスレーブ番号.

spi_read_config_reg

```
unsigned long spi_read_config_reg(unsigned long ch);
```

SPI の Config レジスタの値を読み出します.

ch
: SPI のチャネル番号.

返り値
: SPI の Config レジスタの値.

spi_init

```
void spi_init(unsigned long ch,
              unsigned long slave_num);
```

SPI を初期化します.

ch
: SPI のチャネル番号.

slave_num
: SPI のスレーブ番号.

4.4.5 PWM 発生器

pwmgen_init

```
void pwmgen_init(int ch);
```

PWM 発生器を初期化します.

ch
: PWM 発生器のチャネル番号.

pwmgen_start

```
void pwmgen_start(int ch,
                  unsigned long cycle,
                  unsigned long reversetime,
                  int deadtime,
                  int way);
```

PWM 発生器を開始します.

ch
: PWM 発生器のチャネル番号.

cycle
: PWM 発生器の周期.

reversetime
: PWM 発生器の反転制御時間.

deadtime
: PWM 発生器のデッドタイム.

way
: PWM 波の論理．0 を指定すると負論理，1 を指定すると正論理になります．それ以外の値を指定すると，Reverse Mode になります.

pwmgen_stop

```
void pwmgen_stop(int ch);
```

PWM 発生器の停止します.

ch
: PWM 発生器のチャネル番号.

pwmgen_clear

```
void pwmgen_clear(int ch);
```

PWM 発生器のカウンタをクリアします.

ch
: PWM 発生器のチャネル番号.

pwmgen_clear_and_stop

```
void pwmgen_clear_and_stop(int ch);
```

PWM 発生器を停止し，カウンタをクリアします.

ch
: PWM 発生器のチャネル番号.

pwmgen_control_write

```
void pwmgen_control_write(int ch,
                          unsigned long data);
```

PWM 発生器の Control レジスタに書き込みます.

ch
: PWM 発生器のチャネル番号.

data
: PWM 発生器の Control レジスタに書き込む値.

pwmgen_control_read

```
unsigned long pwmgen_control_read(int ch);
```

PWM 発生器の Control レジスタを読み出します.

ch
: PWM 発生器のチャネル番号.

返り値
: PWM 発生器の Control レジスタの値.

pwmgen_forward_counter_write

```
void pwmgen_forward_counter_write(int ch,
                                  unsigned long data);
```

PWM 発生器の周期を設定します.

ch
: PWM 発生器のチャネル番号.

data
: PWM 発生器の周期.

pwmgen_forward_counter_read

```
unsigned long pwmgen_forward_counter_read(int ch);
```

PWM 発生器の周期を読み出します.

ch
: PWM 発生器のチャネル番号.

返り値
: PWM 発生器の周期の値.

pwmgen_reverse_counter_write

```
void pwmgen_reverse_counter_write(int ch,
                                  unsigned long data);
```

PWM 発生器の反転制御時間を設定します.

ch
: PWM 発生器のチャネル番号.

data
: PWM 発生器の反転制御時間.

pwmgen_reverse_counter_read

```
unsigned long pwmgen_reverse_counter_read(int ch);
```

PWM 発生器の反転制御時間を読み出します.

ch
　PWM 発生器のチャネル番号.

返り値
　PWM 発生器の反転制御時間の値.

pwmgen_deadtime_write

```
void pwmgen_deadtime_write(int ch,
                           unsigned long data);
```

PWM 発生器のデッドタイムを設定します.

ch
　PWM 発生器のチャネル番号.

data
　PWM 発生器のデッドタイム.

pwmgen_deadtime_read

```
unsigned long pwmgen_deadtime_read(int ch);
```

PWM 発生器のデッドタイムの値を読み出します.

ch
　PWM 発生器のチャネル番号.

返り値
　PWM 発生器のデッドタイムの値.

pwmgen_start_forward

```
void pwmgen_start_forward(int ch);
```

PWM 発生器を正論理で出力開始します.

ch
: PWM 発生器のチャネル番号.

pwmgen_start_reverse

```
void pwmgen_start_reverse(int ch);
```

PWM 発生器を負論理で出力開始します.

ch
: PWM 発生器のチャネル番号.

pwmgen_set_first

```
void pwmgen_set_first(int ch,
                      unsigned int cycle,
                      unsigned int reversetime,
                      unsigned short deadtime);
```

PWM 発生器の周期，反転制御時間，デッドタイムを設定します.

ch
: PWM 発生器のチャネル番号.

cycle
: PWM 発生器の周期.

reversetime
: PWM 発生器の反転制御時間.

deadtime
: PWM 発生器のデッドタイム.

pwmgen_output

```
void pwmgen_output(int ch,
                   int way);
```

PWM 発生器を出力開始します.

ch
: PWM 発生器のチャネル番号.

way
: PWM 波の論理．0 を指定すると負論理，1 を指定すると正論理になります．それ以外の値を指定すると，Reverse Mode になります.

4.4.6 PWM 入力器

pwmin_init

```
void pwmin_init(int ch);
```

PWM 入力器を初期化します.

ch
: PWM 入力器のチャネル番号.

pwmin_intr_set

```
void pwmin_intr_set(int ch,
                    int flag);
```

PWM 入力器の割込みを発生させます.

ch
: PWM 入力器のチャネル番号.

flag
: 割込みフラグ．1 を指定すると割込みを発生させます.

pwmin_intr_clear

```
void pwmin_intr_clear(int ch);
```

PWM 入力器の割込みをクリアします.

ch
: PWM 入力器のチャネル番号.

pwmin_lpo_write

```
void long pwmin_lpo_write(int ch,
                          unsigned int period);
```

デューティー比を平均化する周期を設定します.

ch
: PWM 入力器のチャネル番号.

period
: デューティー比を平均化する周期.

pwmin_lpo_read

```
unsigned long pwmin_lpo_read(int ch);
```

デューティー比を平均化する周期を読み出します.

ch
: PWM 入力器のチャネル番号.

返り値
: デューティー比を平均化する周期.

pwmin_lp_read

```
unsigned long pwmin_lp_read(int ch);
```

現在の周期を読み出します.

ch
PWM 入力器のチャネル番号.

返り値
現在の周期.

pwmin_h_read

```
unsigned long pwmin_h_read(int ch);
```

指定した周期で High の期間の合計を読み出します.

ch
PWM 入力器のチャネル番号.

返り値
指定した周期で High の期間の合計.

pwmin_l_read

```
unsigned long pwmin_l_read(int ch);
```

指定した周期で Low の期間の合計を読み出します.

ch
PWM 入力器のチャネル番号.

返り値
指定した周期で Low の期間の合計.

pwmin_ctr_read

```
unsigned long pwmin_ctr_read(int ch);
```

PWM 入力器のコントロールレジスタを読み出します.

ch
: PWM 入力器のチャネル番号.

返り値
: PWM 入力器のコントロールレジスタの値.

pwmin_ctr_write

```
void pwmin_ctr_write(int ch,
                     unsigned long data);
```

PWM 入力器のコントロールレジスタに書き込みます.

ch
: PWM 入力器のチャネル番号.

data
: PWM 入力器のコントロールレジスタへ書き込む値.

4.4.7 パルスカウンタ

plscntr_init

```
void plscntr_init(int ch);
```

パルスカウンタを初期化します.

ch
: パルスカウンタのチャネル番号.

plscntr_start

```
void plscntr_start(int ch);
```

パルスカウンタを開始します.

ch
　パルスカウンタのチャネル番号.

plscntr_stop

```
void plscntr_stop(int ch);
```

パルスカウンタを停止します.

ch
　パルスカウンタのチャネル番号.

plscntr_control_read

```
unsigned long plscntr_control_read(int ch);
```

パルスカウンタのコントロールレジスタを読み出します.

ch
　パルスカウンタのチャネル番号.

返り値
　パルスカウンタのコントロールレジスタの値.

plscntr_write

```
void plscntr_write(int ch,
                   unsigned long data);
```

パルスカウンタのコントロールレジスタへ書き込みます.

ch
パルスカウンタのチャネル番号.

data
パルスカウンタのコントロールレジスタへ書き込む値.

plscntr_count_read

```
unsigned long plscntr_count_read(int ch);
```

パルスカウンタのカウントレジスタを読み出します.

ch
パルスカウンタのチャネル番号.

返り値
パルスカウンタのカウントレジスタの値.

4.4.8 PIO

pio_setup

```
void pio_setup(void);
```

PIO を初期化します．全てのチャネルを出力に設定します.

pio_direction

```
void pio_direction(unsigned long direction);
```

PIO の入出力の方向を設定します.

direction
PIO の入出力の設定．1 に設定したチャネルは出力，0 に設定したチャネルは入力になります.

pio_put

```
void pio_put(unsigned long data);
```

PIO に出力します.

data
PIO に出力するデータ

pio_get

```
unsigned long pio_get(void);
```

PIO からデータを読み出します.

返り値
PIO から読み出した値

pio_intr_enable

```
void pio_intr_enable(unsigned long intr);
```

PIO の割込みを設定します.

intr
割込みの設定値. 1 を指定したチャネルの値が 0 から 1 に変化すると割込みがかかります.

4.4.9 ロック機構

rgpsh

```
unsigned long rgpsh(unsigned long shreg);
```

ロックをかけずに共有レジスタから整数値を読み出します.

shreg
共有レジスタ番号.

返り値
共有レジスタの値.

wgpsh

```
void wgpsh(unsigned long gpreg,
           unsigned long shreg);
```

ロックをかけずに共有レジスタへ整数値を書き込みます.

gpreg
書き込みデータ.

shreg
書き込み先の共有レジスタ番号.

rgpex

```
unsigned long rgpex(unsigned long shreg);
```

ハードウェアセマフォを獲得して共有レジスタから整数値を読み出します.

shreg
　共有レジスタ番号.

返り値
　共有レジスタの値.

wgpex

```
void wgpex(unsigned long gpreg,
           unsigned long shreg);
```

ハードウェアセマフォを解放して共有レジスタへ整数値を書き込みます.

gpreg
　書き込みデータ.

shreg
　書き込み先の共有レジスタ番号.

down_sem

```
void down_sem(semaphore_t *sem);
```

ソフトウェアセマフォを獲得します.

sem
　セマフォの構造体のポインタ.

up_sem

```
void up_sem(semaphore_t *sem);
```

ソフトウェアセマフォを解放します.

sem
　セマフォの構造体のポインタ.

4.4.10 DMAC

dma_set_priority

```
void dma_set_priority(unsigned long n,
                      unsigned long p);
```

DMAC の優先度を変更します.

n
: DMAC のチャネル番号.

p
: DMAC の優先度．0 を指定するとラウンドロビン，1 を指定すると $ch0 > ch1 > ch2 > ch3$ の優先度になります.

start_dma

```
void start_dma(int n,
               int ch);
```

DMAC の転送を開始します.

n
: DMAC の番号．D-RMTPI は 4 個の DMAC を内蔵しています.

ch
: DMAC のチャネル番号．DMAC は 1 個あたり 4 チャネルの入力チャネルを持っています.

set_dma_tmr

```
void set_dma_tmr(int n,
                 int ch,
                 unsigned long flag);
```

DMAC の転送モード制御レジスタを設定します.

n
: DMAC の番号.

ch
: DMAC のチャネル番号.

flag
: DMAC の転送モード制御レジスタに書き込む値.

clear_dma_tmr

```
void clear_dma_tmr(int n,
                   int ch);
```

DMAC の転送モード制御レジスタを初期化します.

n
: DMAC の番号.

ch
: DMAC のチャネル番号.

dma_setting

```
void dma_setting(int n,
                 int ch,
                 unsigned long src,
                 unsigned long dst,
                 unsigned long length,
                 unsigned long flag);
```

n
: DMAC の番号.

ch
: DMAC のチャネル番号.

src
: 転送元アドレス.

dst
: 転送先アドレス.

length
: DMAC 転送のバイト長.

flag
: DMAC の転送モード.

wait_dma_finish

```
void wait_dma_finish(int n,
                     int ch);
```

DMAC の転送終了まで待機します.

n
: DMAC の番号.

ch
: DMAC のチャネル番号.

4.4.11 LED

LED_ON

```
void LED_ON(unsigned int x);
```

LED の特定の 1 個を点灯します. それ以外の LED は消灯します.

x
: 点灯させる LED の bit 位置.

LED_SET

```
void LED_SET(unsigned int x);
```

LED を点灯させます.

x
: LED の点灯パターン.

4.4.12 ベクトル演算器

reserve_vreg_int/reserve_vreg_fp

```
int reserve_vreg_int(unsigned int mode);
int reserve_vreg_fp(unsigned int mode);
```

ベクトルレジスタを確保する.

mode
　ベクトルレジスタの構成.

返り値
　ベクトルレジスタの確保に成功すると 1 を返す.

set_mask_low_vreg_int/set_mask_high_vreg_int

```
void set_mask_low_vreg_int(unsigned long mask);
void set_mask_high_vreg_int(unsigned long mask);
```

整数ベクトル演算器のマスクレジスタを設定する.

mask
　マスク値.

set_mask_vreg_fp

```
void set_mask_vreg_fp(unsigned long mask);
```

浮動小数点ベクトル演算器のマスクレジスタを設定する.

mask
　マスク値.

set_round_vreg_fp

```
void set_round_vreg_fp(unsigned int mode);
```

浮動小数点ベクトル演算器の丸めモードを指定する.

mode
　浮動小数点の丸めモード.

set_length_vreg_int/set_length_vreg_fp

```
void set_length_vreg_int(unsigned int length);
void set_length_vreg_fp(unsigned int length);
```

演算を行うベクトル長を設定する.

length
　演算を行うベクトル長.

set_stride_vreg_int/set_stride_vreg_fp

```
void set_stride_vreg_int(unsigned int stride);
void set_stride_vreg_fp(unsigned int stride);
```

Load/Store 時のアドレスのストライドを設定する.

stride
: アドレスのストライド.

release_vreg_int/release_vreg_fp

```
int release_vreg_int(void);
int release_vreg_fp(void);
```

確保していたベクトルレジスタを解放する.

4.4.13 MMU

immu_on

```
void immu_on(unsigned long thid);
```

命令用 MMU を有効にします.

thid
: コンテキスト ID.

dmmu_on

```
void dmmu_on(unsigned long thid);
```

データ用 MMU を有効にします.

thid
: コンテキスト ID.

immu_off

```
void immu_off(unsigned long thid);
```

命令用 MMU を無効にします.

thid
: コンテキスト ID.

dmmu_off

```
void dmmu_off(unsigned long thid);
```

データ用 MMU を無効にします.

thid
: コンテキスト ID.

immu_on_all

```
void immu_on_all(void);
```

全てのコンテキストで命令用 MMU を有効にします.

dmmu_on_all

```
void dmmu_on_all(void);
```

全てのコンテキストでデータ用 MMU を有効にします.

4.4.14 ユーティリティ

printk

```
int printk(const char *fmt, ...);
```

シリアルの 0 番ポートに文字列を出力します.

fmt
: 出力文字列のフォーマット．使用可能なフォーマット一覧を下に示します.

- d: 10 進数整数
- x: 16 進数整数
- s: 文字列
- c: 文字

prints

```
void prints(char *s);
```

シリアルの 0 番ポートに文字列を出力します.

s
: 文字列のポインタ.

serial_putc

```
extern void serial_putc(int ch,
                        byte_t data);
```

シリアルに 1 バイト送信します.

ch
: 出力ポート. 0 か 1 を指定してください.

data
: シリアルに送信するデータ.

serial_getc

```
extern byte_t serial_getc(int);
```

シリアルから 1 バイト受信します.

ch
: 出力ポート. 0 か 1 を指定してください.

返り値
: シリアルから受信したデータ.

putxval

```
int putxval(unsigned long value,
            int column);
```

16 進数文字列をシリアルに出力します.

value
: シリアルに出力する整数.

column
: 表示桁数. value を文字列に変換した際の桁数よりも小さい値を指定した場合は, 変換した文字列そのものを出力します. value を文字列に変換した際の桁数よりも大きい値を指定した場合は, 変換した文字列の上位に'0' を付加して出力します.

putdval

```
int putdval(unsigned int value,
            int column);
```

10 進数文字列をシリアルに出力します.

value
: シリアルに出力する整数.

column
: 表示桁数. value を文字列に変換した際の桁数よりも小さい値を指定した場合は, 変換した文字列そのものを出力します. value を文字列に変換した際の桁数よりも大きい値を指定した場合は, 変換した文字列の上位に'0' を付加して出力します.

putbval

```
int putbval(const unsigned long value,
            const unsigned int bit_width);
```

2 進数文字列をシリアルに出力します.

value
: シリアルに出力する整数.

bit_width
: 表示桁数. value を文字列に変換した際の桁数は考慮されません.

print_double

```
int print_double(const double value);
```

倍精度浮動小数点をシリアルに出力します.

value
　シリアルに出力する値.

debug_printk/debug_prints

```
void debug_printk(...)
```

デバッグ用の printk 関数です．Favor OS の configure オプションで”–debugprint=on”を指定した場合のみ，シリアルに文字列を出力します.

read_byte/read_hword/read_word/read_long

```
unsigned char read_byte(unsigned char addr);
unsigned short read_hword(unsigned short addr);
unsigned int read_word(unsigned int addr);
unsigned long read_long(unsigned long addr);
```

指定したアドレスから値を読み出します.

addr
　読み出すアドレス.

返り値
　読み出した値.

write_byte/write_hword/write_word/write_long

```
void write_byte(unsigned char *addr,
                unsigned char data);
void write_hword(unsigned short *addr,
                 unsigned short data);
void write_word(unsigned int *addr,
                unsigned int data);
void write_long(unsigned long *addr,
                unsigned long data);
```

指定したアドレスへ値を書き込みます.

addr
: 書き込み先のアドレス.

data
: 書き込む値.

4.4.15 標準ライブラリ関数

***memset**

```
void *memset(void *b,
             int c,
             long len);
```

メモリ領域に文字データを書き込みます.

b
: 書き込みを開始する先頭アドレス.

c
: メモリ領域に書き込む文字.

len
: 書き込むバイト長.

***memcpy**

```
void *memcpy(void *dst,
             void *src,
             long len);
```

メモリの内容をコピーします.

dst
: コピー先の先頭アドレス.

src
: コピー元の先頭アドレス.

len
: コピーするバイト長.

memcmp

```
int memcmp(void *b1,
           void *b2,
           long len);
```

メモリの内容を比較します.

b1
: 比較する領域の先頭アドレス.

b2
: 比較する領域の先頭アドレス.

len
: 比較するバイト長.

返り値
: メモリの内容が一致した場合は 0 が返ります.

strlen

```
int strlen(const char *s);
```

文字列の長さを調べます.

s
: 文字列の先頭ポインタ.

返り値
: 文字列の長さ.

strcpy

```
char *strcpy(char *dst,
             const char *src);
```

文字列をコピーします.

dst
: コピー先の文字列の先頭ポインタ.

src
　コピー元の文字列の先頭ポインタ.

返り値
　コピー先の文字列の先頭ポインタ.

strcmp/strncmp

```
int strcmp(char *s1,
           char *s2);
int strncmp(char *s1,
            char *s2,
            int len);
```

s1
　比較する文字列の先頭アドレス.

s2
　比較する文字列の先頭アドレス.

len
　比較する文字列の長さ.

返り値
　文字列が一致した場合は 0 が返ります.

atoi

```
int atoi(char *str);
```

10 進数文字列を整数に変換します.

str
　整数に変換する 10 進数文字列.

返り値
　変換した整数値.

atohex

```
unsigned int atohex(char *str);
```

16 進数文字列を整数に変換します.

str
　整数に変換する 16 進数文字列.

返り値
　変換した整数値.

dump

```
void dump(unsigned long addr,
          int dump_length);
```

メモリの内容を出力します.

addr
　出力を開始する領域の先頭アドレス.

dump_length
　出力するバイト数.

wait_msec

```
void wait_msec(unsigned long msec);
```

指定した時間だけ処理を停止します.

msec
　処理を停止する時間. 単位は ms.

5

ITRON仕様OSの使い方

本章では，RMTP 用 ITRON 仕様 OS の使い方を説明します．RMTP 用 ITRON 仕様 OS は μITRON4.0 仕様[1] のリアルタイム OS である Hyper Operating System (HOS) [2]を拡張したものになっています．

5.1 ビルド方法

hos の sample ディレクトリ以下でビルドの設定を config.mak ファイルに書き込んだ後，make コマンドを実行します．config.mak ファイルの設定サンプルは config_example/以下にあります．

```
% ls
README    config/   document/ include/  sample/   src/
% cd sample/
% ls
Makefile   cache.c         mk.sh      printk.c      script/
Rules.mak  config.mak      mmu.c      sample.c      system.cfg
backup/    config_example/ ostimer.c  sample.c.bak  uart.c
boot.S     conv            ostimer.h  sample.h
% cp config_example/config.mak.serial_load config.mak
(rmtsim の場合は cp config_example/config.mak.inst config.mak)
% make clean
% make
```

[1] http://www.ertl.jp/ITRON/SPEC/mitron4-j.html
[2] http://sourceforge.jp/projects/hos/

初回のみカーネルのビルドが必要なため，src ディレクトリ以下で make するか，sample ディレクトリ以下の mk.sh を実行して下さい

```
% ls
README    config/   document/ include/  sample/   src/
% cd sample/
% ./mk.sh
```

5.2 生成されるファイル

make が成功すると，結果は sample ディレクトリに出力されます．主なものを表 5.1 に示します．

表 5.1: ITRON 仕様 OS の出力ファイル

ファイル	説明
sample.bin	評価キットへロードするファイル
sample.srec	命令レベルシミュレータ用のファイル
sample.dmp	逆アセンブルリスト
sample.map	マップファイル (リンカの動作ログ)

sample ディレクトリで make clean を行うと，中間生成ファイルを含むこれらのファイルを削除することができます．

5.3 アプリケーションコードの記述

アプリケーションコードや各種設定ファイルは sample ディレクトリ内に置きます．ユーザが編集するファイルを表 5.2 に示します．

表 5.2: ITRON 仕様 OS の各種ファイル

ファイル	説明
sample.c	アプリケーションファイル
sample.h	ヘッダファイル (sample.c 内の関数を記述)
system.cfg	コンフィグレーションファイル (静的 API を記述)
ostimer.h	スケジューラが呼ばれる間隔 (TICK) を記述

5.4 ITRON API

5.4.1 cre_tsk

```
ER cre_tsk(ID tskid, {ATR tskatr,
                      VP_INT exinf,
                      FP task,
                      PRI itskpri,
                      SIZE stksz,
                      VP stk,
                      UW period,
                      UW wcet});
```

リアルタイムタスクを生成します.

tskid
生成対象のタスクの ID 番号.

tskatr
タスク属性．TA_HLNG（C 言語で記述）か TA_ASM（アセンブリ言語で記述）のどちらかを指定することができます.

exinf
タスクの拡張情報．開発者が任意に使うことができます.

task
タスクの起動番地．関数名を指定します.

itskpri
タスクの起動時優先度．1 以上の値を指定します．値が低いほど優先度は高くなります.

stksz
タスクのスタック領域のサイズ（バイト数）.

stk
タスクのスタック領域の先頭番地．NULL を指定するとカーネルが適当に確保します.

period
タスクの周期.

wcet
タスクの最悪実行時間.

5.4.2 act_tsk

```
ER act_tsk(ID tskid);
```

タスクを起動します.

tskid
起動対象のタスクの ID 番号.

5.4.3 ext_tsk

```
void ext_tsk(void);
```

自タスクを終了します.

5.4.4 slp_tsk

```
ER slp_tsk(void);
```

自タスクを起床待ち状態に移行します.

5.4.5 wup_tsk

```
ER wup_tsk(ID tskid);
```

タスクを起床させます.

tskid
起床対象のタスクの ID 番号.

5.4.6 dly_tsk

```
ER dly_tsk(RELTIM dlytim);
```

自タスクを遅延させます.

dlytim
自タスクの遅延時間（相対時間）.

5.4.7 cre_sem

```
ER cre_sem(ID semid, {ATR sematr,
                      UINT isemcnt,
                      UINT maxsem});
```

セマフォを生成します.

semid
生成対象のセマフォの ID 番号.

sematr
セマフォ属性. 待ち行列の順を指定します. TA_FIFO（FIFO 順）か TA_TPRI（タスクの優先度順）のどちらかを指定します.

isemcnt
セマフォの資源数の初期値.

maxsem
セマフォの最大資源数.

5.4.8 wai_sem

```
ER wai_sem(ID semid);
```

セマフォ資源を獲得します.

semid
資源獲得対象のセマフォの ID 番号.

5.4.9 sig_sem

```
ER sig_sem(ID semid);
```

セマフォ資源を獲得します.

semid
資源獲得対象のセマフォの ID 番号.

5.4.10 sig_sem

```
ER wai_sem(ID semid);
```

セマフォ資源を返却します.

semid
資源返却対象のセマフォの ID 番号.

5.4.11 cre_cyc

```
ER cre_cyc(ID cycid, {ATR cycatr,
                      VP_INT exinf,
                      FP cychdr,
                      RELTIM cyctim,
                      RELTIM cycphs});
```

周期ハンドラを生成します.

cycid
生成対象の周期ハンドラの ID 番号.

cycatr
周期ハンドラ属性．TA_HLNG か TA_ASM を指定します．同時に TA_STA（生成と同時に起動）を指定することができます.

exinf
周期ハンドラの拡張情報.

cychdr
周期ハンドラの起動番地．周期的に起動させたい関数名を指定します.

cyctim
周期ハンドラの起動周期.

cycphs
周期ハンドラの起動位相.

6

更新履歴

Revision	Date	Description
第 1 版	2013 年 6 月 17 日	初版公開.
第 2 版	2014 年 5 月 30 日	ITRON 仕様 OS の API を追加.
第 3 版	2014 年 12 月 8 日	*Favor OS* の API を追加.