

μ T-Kernel3.0

AP-RX63N-0A (RXv1コア) 向け
構築手順書

Version. 01. 00. 01

2022. 11. 28

目次

1. 概要	4
1.1 目的	4
1.2 対象 OS およびハードウェア	4
1.3 対象開発環境	4
2. システム構成	5
2.1 機器構成	5
2.2 デバッガ	5
2.3 デバッグコンソール	6
3. μ T-Kernel3.0 を実行する	7
3.1 μ T-Kernel3.0 のビルドと実行	7
3.2 ビルド・オプション	12
3.3 コンフィギュレーション変更時の注意点	16
3.4 コンフィギュレーションとプロファイル	17
3.5 RXv1 コア特有のコンフィギュレーション	18
4. CS+のパートナーOS 対応デバッグプラグイン	19
4.1 ダウンロードとインストール	19
4.2 プラグイン機能の有効化	19
4.3 OS 定義ファイル	20
4.4 パートナーOS 対応デバッグプラグインの機能	21
5. 各種スタックサイズの算出	24
5.1 スタック見積もりツール	24
5.2 リアルタイム OS オプションの設定	25
5.3 各種スタックサイズの計算方法	28
5.3.1 サイズ計算が必要なスタック領域	28
5.3.2 OS 管理外割込みのサイズ	28
5.3.3 タスクコンテキストのサイズ	29
5.3.4 タスクスタックのサイズ	29
5.3.5 例外スタックのサイズ	29
5.3.6 スタックサイズの算出例	31
5.3.7 デバイスドライバのシステムコールのスタックサイズ	33
6. C 言語規格準拠の仕様	34
6.1 C 言語規格に準拠していない部分	34
6.2 マクロ定義による API の変更	34
6.2.1 T-Monitor のライブラリ	35
6.2.2 高速ロック・マルチロックライブラリ	35
6.2.3 カーネルオブジェクト生成時のパラメータパケット	35

6.2.4	μ T-Kernel/OS システムコール	37
6.2.5	μ T-Kernel/SM システムコール	37
6.2.6	μ T-Kernel/DS システムコール	37
7.	アプリケーションプログラムの作成	38
8.	問い合わせ先	38

1. 概要

1.1 目的

本書は TRON フォーラムから公開されている μ T-Kernel3.0 (V3.00.00) のソースコードをルネサスエレクトロニクス社の RX 用に改変したソースコードの開発環境の構築手順を記載します。

以降、本ソフトとは前述の μ T-Kernel3.0 を改変したソースコードを示します。

1.2 対象 OS およびハードウェア

本書は以下を対象とする。

分類	名称	備考
OS	μ T-Kernel3.00.00	RX用に改変したソースコード
実機	AP-RX63N-0A	アルファプロジェクト製
搭載マイコン	RX63N R5F563NEDx FB	ルネサスエレクトロニクス製
デバugga	E1/E2等	JTAGデバugga

1.3 対象開発環境

本ソフトは C 言語コンパイラとして、ルネサスエレクトロニクス社の CS+ (CC-RX) の利用を前提とします。ただし、本ソフトはハードウェア依存部を除けば、標準の C 言語で記述されており、他の C 言語コンパイラへの移植も可能です。

2. システム構成

本章では、本ソフトを動作させるために必要となるシステム全体の構成などについて説明します。

2.1 機器構成

本ソフトを動作させるために必要となる機器構成は次の通りです。AP-RX63N-0A の CPU ボードは JTAG デバッガとシリアル/USB 変換アダプタでホスト PC と接続します。

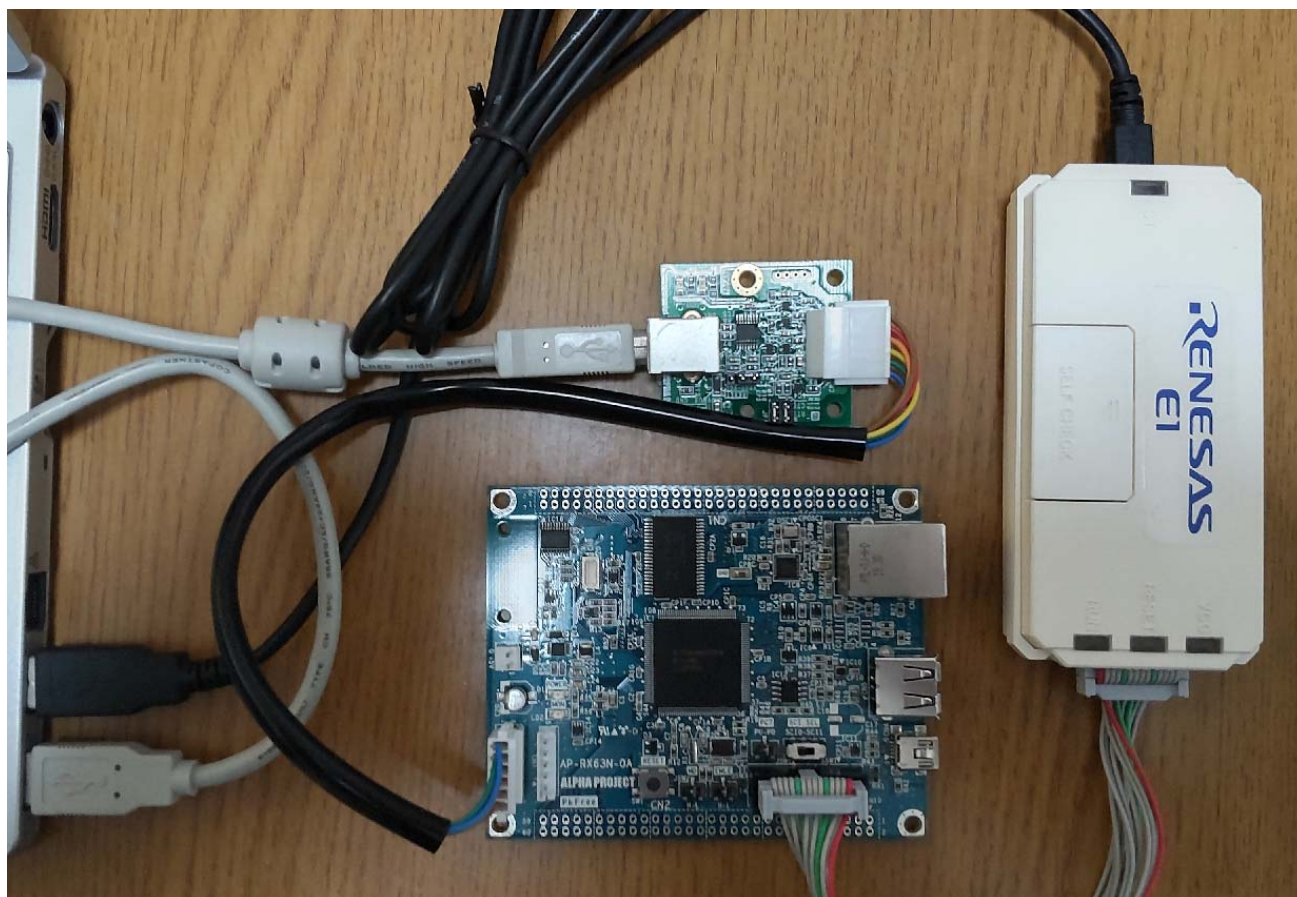


図 2.1 AP-RX63N-0A のハードウェア構成

2.2 デバッガ

ターゲットハードウェアとホスト PC をデバッガで接続して開発を行います。デバッガには E1 や E2 等の JTAG デバッガを利用し、CN10 コネクタに接続します。図 2.1 は E1 デバッガを接続した時のハードウェア構成です。

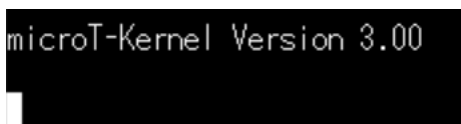
2.3 デバッグコンソール

CN6 コネクタにシリアル/USB 変換アダプタ（PC-USB-03 や PC-USB-04）を使用してシリアルポートをデバッグコンソールとして使用することができます。必須ではありませんが、デバッグコンソールを利用することで μ T-Kernel の起動メッセージなどのログを出力することができます。

ホスト PC では通信ソフトを起動してください。通信パラメータは以下の通りです。

- ・ 通信速度 115,200bps
- ・ ビット長 8bit
- ・ パリティ なし
- ・ ストップビット 1bit
- ・ フロー制御 なし

デバッグコンソールでは、 μ T-Kernel の RS-232C ドライバは利用しません。ポーリングによってシリアルを制御しています。



```
microT-Kernel Version 3.00
```

図 2.2 μ T-Kernel の起動メッセージ例

なお、デバッグコンソールではシリアルポートとして RX63N 内蔵の SCI0 を利用しています。デバッグコンソールを利用する場合は SSW1 のスライドスイッチを図 2.3 に示すように SCI0 側にスライドしてください。



図 2.3 SSW1 の設定

3. μ T-Kernel3.0 を実行する

CS+の動作が確認できたら、本ソフトをビルドして実行します。

本章では、本ソフトのビルドや実行方法、デバッグ方法について説明します。

3.1 μ T-Kernel3.0 のビルドと実行

(1) CS+の起動

本ソフトをダウンロードし、展開した μ T-Kernel3.0のフォルダに含まれる、以下のプロジェクト・ファイルをダブルクリックしてください。CS+が起動します。

mtkernel_3/ide/cs/AP_RX63N.mtpj



図 3.1 CS+のプロジェクト・ファイル

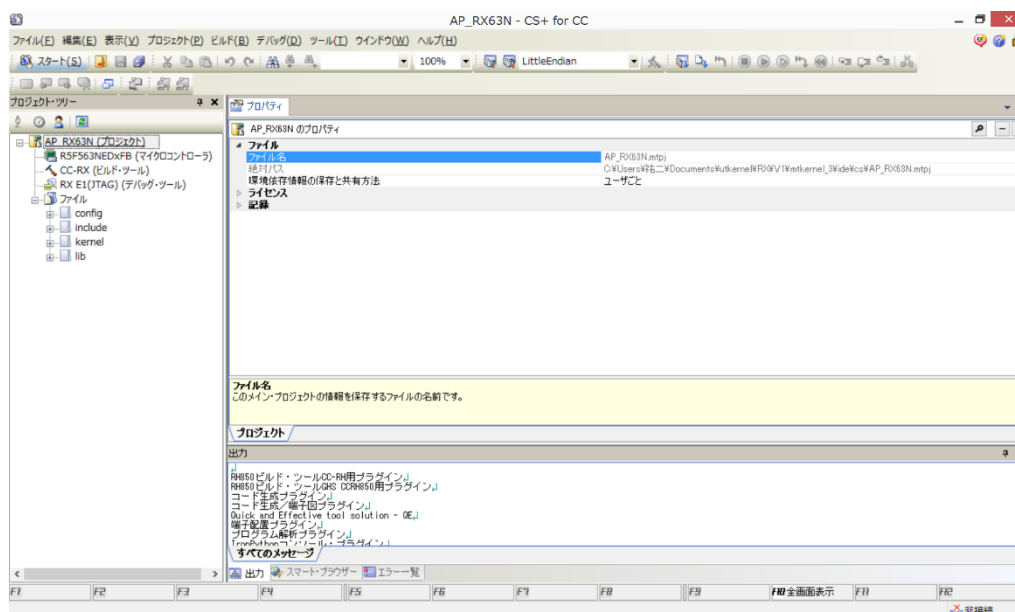


図 3.2 CS+起動直後の画面

この状態からビルドが可能です。エミュレータ等との接続設定などは通常の場合不要です。CS+を構成する各ウィンドウやパネルのサイズなどは適宜調整してください。

(2) デバッグ・ツールの設定

プロジェクト・ツリーで使用するハードウェア構成に合わせてデバッグ・ツールを選択してください。初期状態は RX E1 (JTAG) が選択されています。

その他のデバッガを使用するのであれば、プロジェクト・ツリーのデバッグ・ツールで右クリックし、適切なデバッグ・ツールに切り替えてください。

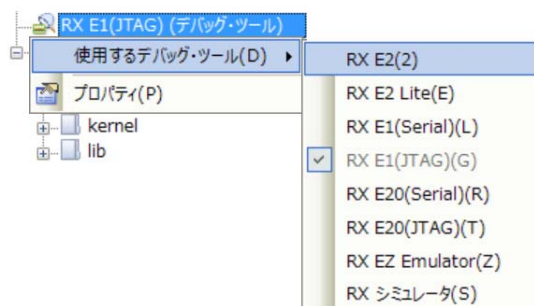


図 3.3 デバッグ・ツールの切り替え

また、JTAG デバッガより電源給電を行うのであれば、プロジェクト・ツリーのデバッグ・ツールをダブルクリックし、当該デバッガのプロパティより電源供給を許可してください。

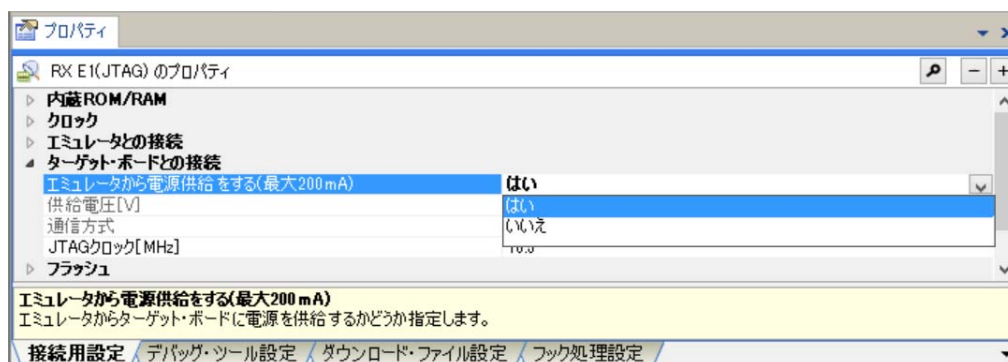


図 3.4 JTAG デバッガからの電源供給

(3) ビルドの開始

デバッグ・メニューの[ビルド&デバッグ・ツールヘダウンロード]、または F6 で μ T-Kernel3.0 をビルドしてください。

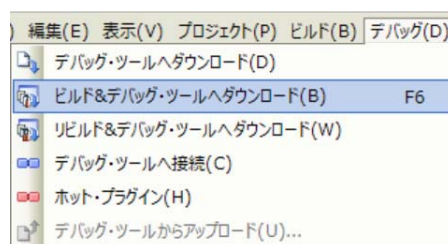


図 3.5 ビルドを開始

(4) ダウンロード

ビルドが完了すると、自動的にプログラムが AP-RX63N-0A にダウンロードされます。

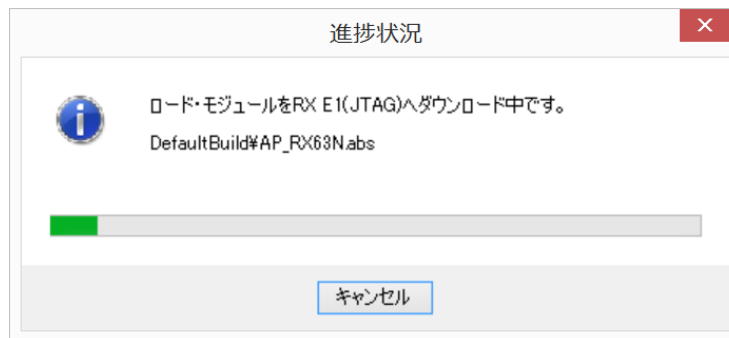


図 3.6 プログラムをロード中

(5) ダウンロード直後の表示

ダウンロードが完了すると現在のプログラムカウンタ (PC) の位置に対応するソースコード (usermain 関数の先頭) が表示された状態で停止します。

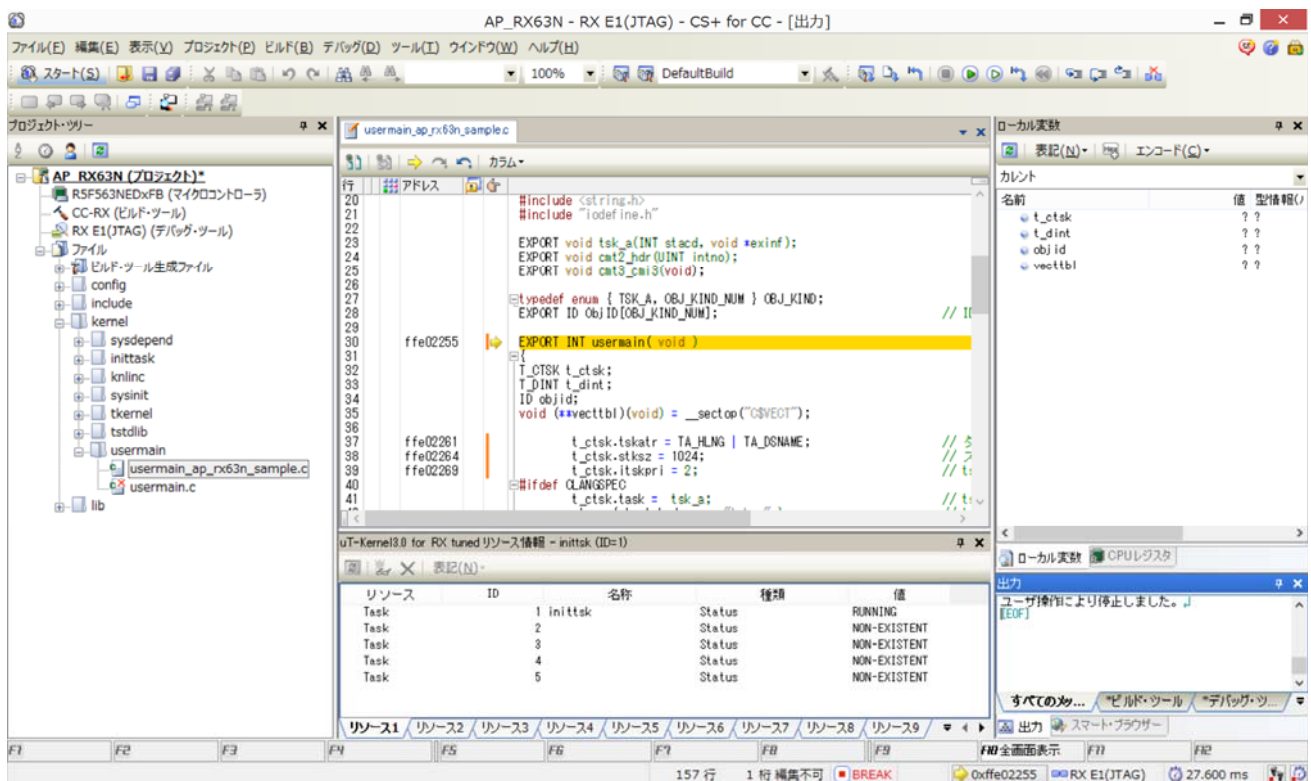


図 3.7 ロード直後の表示

(6) プログラムの実行

デバッグ・メニューの [実行]、または F5 で μ T-Kernel3.0 環境下でのプログラムを実行できます。サンプルでは、AP-RX63N-0A 搭載の LD2 が 100ms の間隔で点灯・消灯を繰り返します。また、OS 管理外割込みのガイダンスに従って設定を追加すれば、5 秒間隔で LD2 の点滅間隔が変わります。

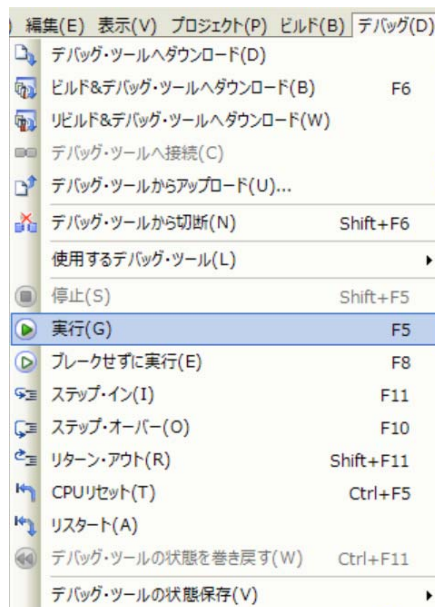


図 3.8 プログラムの実行

(7) プログラム停止

デバッグ・メニューの [停止] または、Shift+F5 でプログラムは停止します。



図 3.9 プログラムの停止

(8) その他のデバッグ機能

プログラムの実行・停止以外にも JTAG デバッガに準拠したデバッグ機能（例えば、ブレークポイントやシングルスステップ、各種デバッグ用パネルの表示）が利用可能です。詳細に関しては CS+ のマニュアルを参照ください。

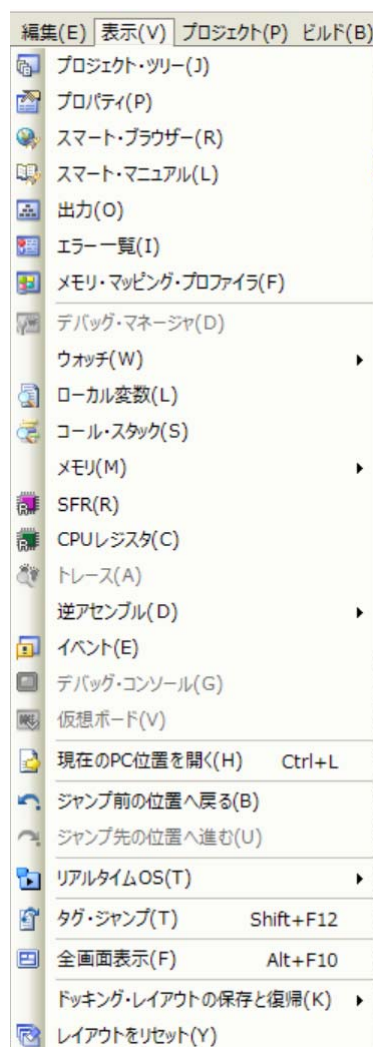


図 3.10 表示・メニュー

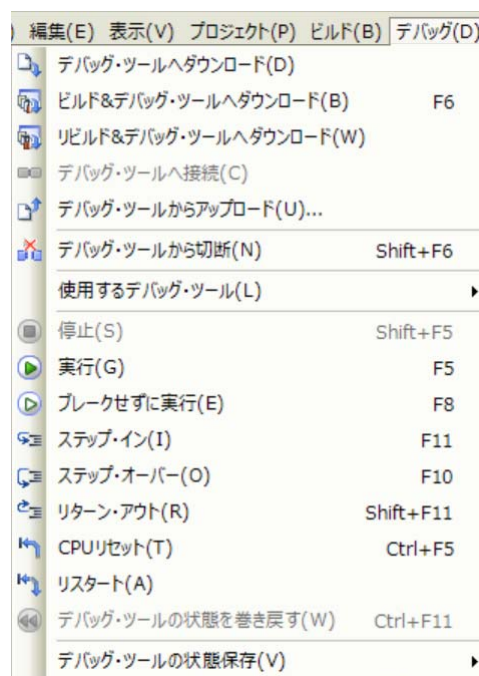


図 3.11 デバッグ・メニュー

3.2 ビルド・オプション

CS+のビルド・オプションを変更することで本ソフトの各種設定を変更することができます。この節では、本ソフトにとって重要な CS+のビルド・オプションを紹介します。なお、CS+のビルド・オプションは、プロジェクト・ツリーの CC-RX（ビルド・ツール）をダブルクリックし、エディタ・パネルに表示される CC-RX のプロパティから変更することができます。

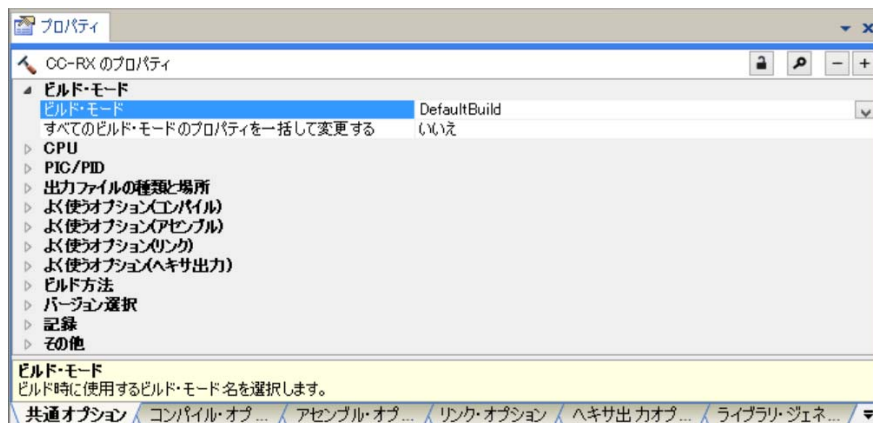


図 3.12 CS+のビルド・オプション

(1) C 言語規格

本ソフトでは、コンパイル・オプションのタブ、ソースのカテゴリ、**C ソース・ファイルの言語のオプションは C99 に固定**とし、C 言語の規格を C99 としています。これは OS 内部の処理において、long long 型をサポートした方が効率の良いオブジェクトを生成できるためです。

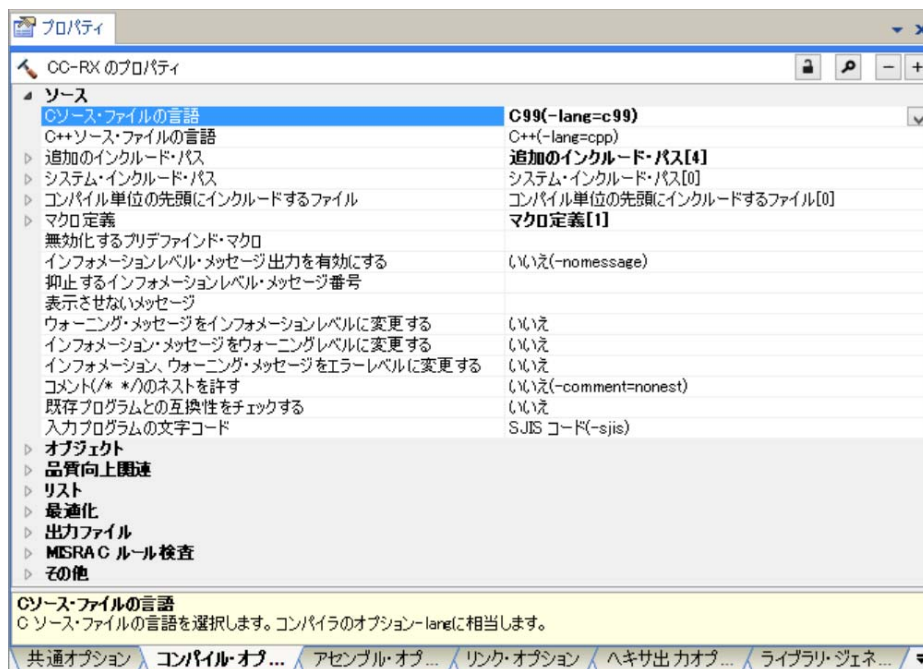


図 3.13 C 言語の規格

(2) マクロ定義

本ソフトでは、システム構成を定義するための2つの定義マクロ（C言語は2個、アセンブリ言語は1個）を持っています。C言語の定義マクロは、コンパイル・オプションのタブ、ソースのカテゴリ、マクロ定義で指定します。アセンブリ言語のマクロ定義は、アセンブル・オプションのタブ、ソースのカテゴリ、マクロ定義で指定します。

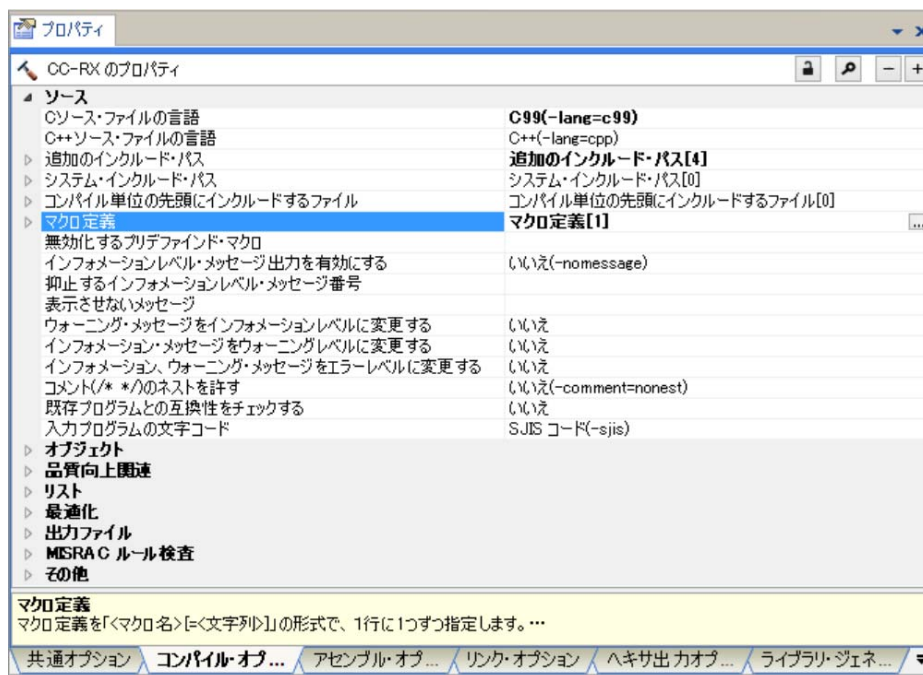


図 3.14 C言語のマクロ定義

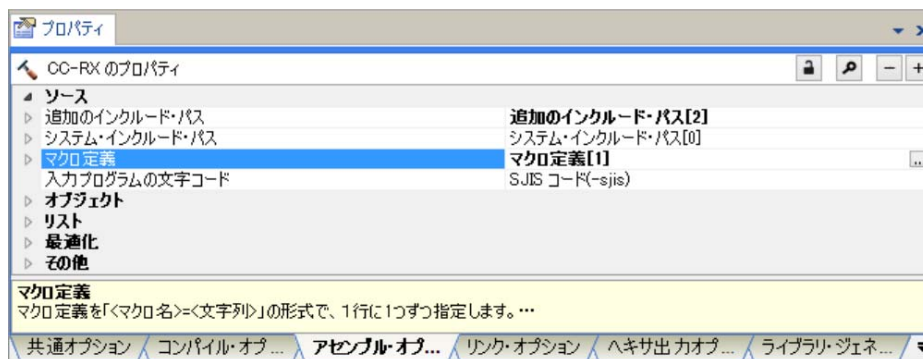


図 3.15 アセンブリ言語のマクロ定義

- ・「_AP_RX63N_」マクロ定義（デフォルトで設定済み）

「_AP_RX63N_」は、ビルドする本ソフトがAP-RX63N-0A用の μ T-Kernel3.0であることを意味するマクロ定義です。このマクロ定義を削除した場合、正しくビルドできません。必ず「_AP_RX63N_」を定義した状態でビルドを行ってください。なお、「_AP_RX63N_」マクロ定義は、C言語、アセンブリ言語の両方に必要となります。

- ・「CLANGSPEC」マクロ定義（デフォルトは未設定）

「CLANGSPEC」は、第6章で説明する μ T-Kernel3.0のシステムコールAPIをC言語規格準拠の仕様に変更するためのマクロ定義です。デフォルトは未設定です。詳細は第6章で説明します。

(3) データのエンディアン

本ソフトでは、データのエンディアンを変更することが可能です。データのエンディアンは、共通オプションのタブ、CPUのカテゴリ、データのエンディアンで指定します。デフォルトの設定はLittle Endianですが、システムの都合に合わせてBig Endianに変更することが可能です。また、本オプションの設定に合わせて、プロファイル TK_BIGENDIAN の設定値は変化します。

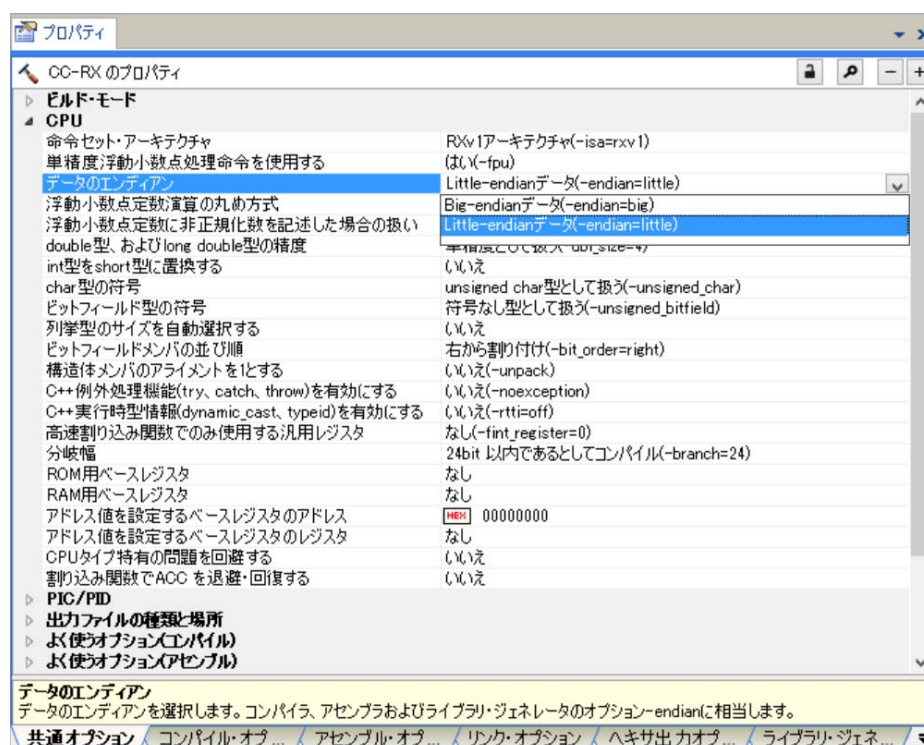


図 3.16 データのエンディアン

なお、本オプションの変更はデバッグとの接続を解除した状態で行ってください。デバッグとの接続が解除されていれば、以下のダイアログの表示と共にデバッグ・ツールのエンディアン設定も自動的に変更することが可能です。

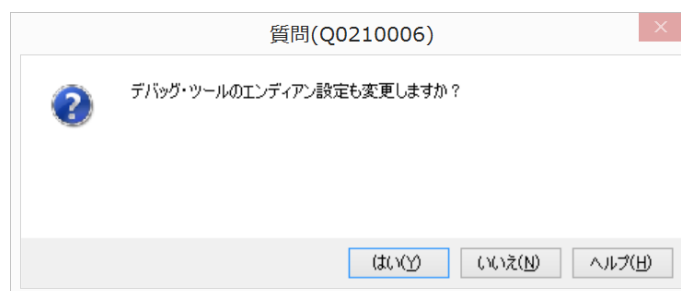


図 3.17 デバッグ・ツールのエンディアン

(4) モジュール間最適化

本ソフトでは、最適化リンクが持つ最適化方法のオプションを有効にしています。最適化リンクの最適化方法は、リンク・オプションのタブ、最適化のカテゴリ、最適化方法で指定します。これはシステムで使用しない μ T-Kernel3.0の関数を削除し、ROM領域やRAM領域の使用メモリ量を削減するためです。

テストスイート等により、本オプションを設定しても μ T-Kernel3.0の動作に支障がないことは動作検証していますが、システムの都合により本オプションの設定値は変更しても構いません。

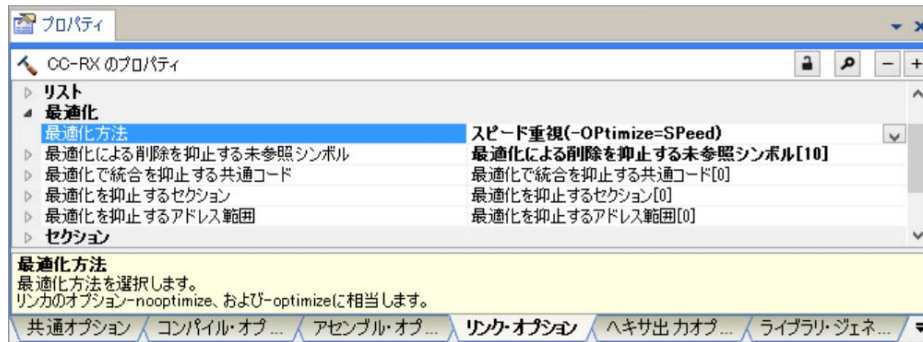


図 3.18 最適化リンクの最適化方法

(5) 外部変数アクセスの最適化

本ソフトでは、コンパイラが持つ外部変数アクセス最適化のオプションを有効にしています。外部変数アクセスの最適化は、コンパイル・オプションのタブ、最適化のカテゴリ、外部変数アクセス最適化を行うで指定します。これは静的変数の構造化を行い、命令コードの使用メモリ量を削減するためです。

テストスイート等により、本オプションを設定しても μ T-Kernel3.0の動作に支障がないことは動作検証していますが、システムの都合により本オプションの設定値は変更しても構いません。

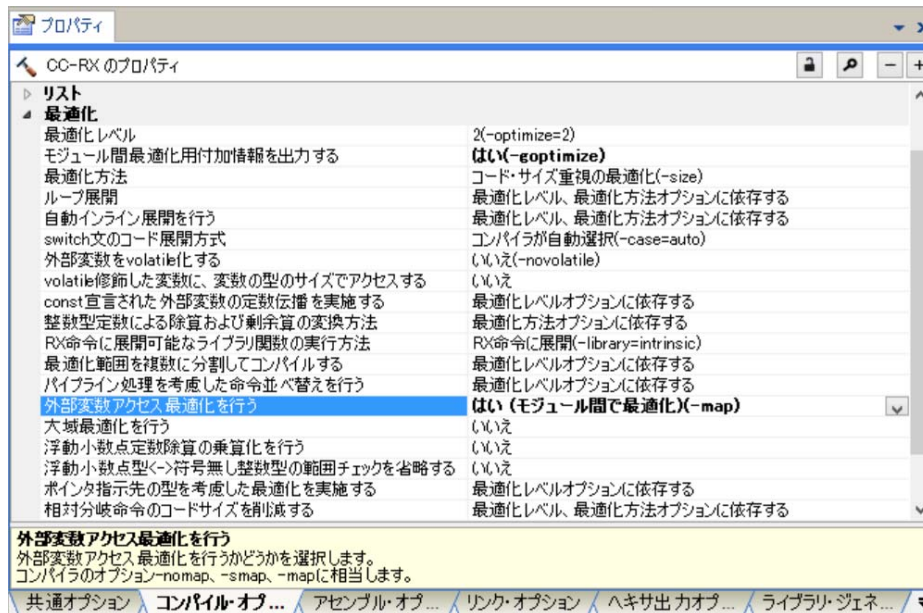


図 3.19 外部変数アクセスの最適化

3.3 コンフィギュレーション変更時の注意点

CS+ではC言語とアセンブリ言語でヘッダファイルを共有することができません。

そこで本ソフトでは、CS+のアセンブリ言語用に専用のヘッダファイル(拡張子: inc)を追加しています。追加したヘッダファイルは、C言語用のヘッダファイルの拡張子をhからincに変更した名称とし、元となったC言語用のヘッダファイルと同一のディレクトリに入れてあります。

C言語用のヘッダファイルとアセンブリ言語用のヘッダファイルが対になっている場合、必ず両方のヘッダファイルでの設定値が同じになるように調整してからビルドしてください。**両者の設定が異なったままビルドした場合、 μ T-Kernel3.0は正常に動作しません。**

例えば、カーネル内部の動的メモリ割当て機能の有無を指定するUSE_IMALLOCを変更する場合、以下の2つのファイルにある設定値を両方とも変更してからビルドする必要があります。

```
/*----- */
/* System function selection
 * 1: Use function. 0: No use function.
 */
#define USE_IMALLOC (1) /* Use dynamic memory allocation */
```

リスト 3.1 mtkernel_3/config/config.h (C言語用のヘッダファイル)

```
;/*----- */
;/* System function selection
; * 1: Use function. 0: No use function.
; */
USE_IMALLOC .EQU (1) /* Use dynamic memory allocation */
```

リスト 3.2 mtkernel_3/config/config.inc (アセンブリ言語用のヘッダファイル)

他の設定についても、C言語用とアセンブリ言語用のヘッダファイルが存在し、**同名のマクロ定義が存在する場合、同じ設定値で修正**する必要があります。

3.4 コンフィギュレーションとプロファイル

コンフィギュレーションとプロファイルで変更可能な設定値のあるファイルは以下のフォルダに格納されています。なお、コンフィギュレーションとプロファイルに定義されている内容は uTK3.0 共通実装仕様書、割込み優先度に関しては uTK3.0_AP-RX63N-0A 実装仕様書を参照ください。

(1) 基本コンフィギュレーション

mtkernel_3/config/config.h
mtkernel_3/config/config.inc

(2) 機能コンフィギュレーション

mtkernel_3/config/config_func.h

(3) 割込み優先度

【MAX_INT_PRI、TIM_INT_PRI】

mtkernel_3/include/sys/sysdepend/cpu/r5f563n/sysdef.h
mtkernel_3/include/sys/sysdepend/cpu/r5f563n/sysdef.inc

(4) プロファイル

【TK_SUPPORT_LOWPPOWER】

mtkernel_3/include/sys/sysdepend/ap_rx63n/profile.h
mtkernel_3/include/sys/sysdepend/ap_rx63n/profile.inc

【TK_SUPPORT_SUBSYSTEM】

mtkernel_3/include/sys/profile.h

【TK_SUPPORT_INTCTRL、TK_HAS_ENAINTLEVEL、TK_SUPPORT_CPUINTLEVEL】

mtkernel_3/include/sys/sysdepend/cpu/core/rxv1/profile.h

【TK_SUPPORT_UTC、TK_SUPPORT_TRONTIME】

mtkernel_3/include/sys/profile.h

【TK_SUPPORT_DSNAME、TK_SUPPORT_DBGSP、TK_MAX_TSKPRI】

TK_SUPPORT_DSNAME	⇒	config.h(inc)のUSE_OBJECT_NAMEで指定
TK_SUPPORT_DBGSP	⇒	config.h(inc)のUSE_DBGSPで指定
TK_MAX_TSKPRI	⇒	config.h(inc)のCFN_MAX_TSKPRIで指定

3.5 RXv1 コア特有のコンフィギュレーション

RXv1 コアには2つの特有なコンフィギュレーションが存在します。2つ共、基本コンフィギュレーションとして mtkernel_3/config/config.h(inc)に定義されています。

(1) USE_FPU

タスクコンテキストに FPSW（浮動小数点ステータスワード）を含めるかどうか意味します。

【config.h】

```
#define USE_FPU (1) /* use float, double, long double */
```

【config.inc】

```
USE_FPU .EQU (1) /* use float, double, long double */
```

リスト 3.3 mtkernel_3/config/config.h(inc)

FPSW は浮動小数点型の演算を行うと使用されます。従って、タスク内で浮動小数点型の演算を行うのであれば、USE_FPU のマクロ定義を (1) にする必要があります。ただし、**全タスクが浮動小数点型の演算を行わない、または、ある1つのタスクしか浮動小数点型の演算を行わないのであれば、FPSW をタスクコンテキストに含める必要はありません**。その場合は USE_FPU のマクロ定義を (0) に変更してください。

(2) USE_DSP

タスクコンテキストに ACC（アキュムレータ）を含めるかどうか意味します。

【config.h】

```
#define USE_DSP (0) /* use rmpab, rmpaw, rmpal */
```

【config.inc】

```
USE_DSP .EQU (0) /* use rmpab, rmpaw, rmpal */
```

リスト 3.3 mtkernel_3/config/config.h(inc)

ACC は積和演算を行うと使用されます。DSP の演算は、コンパイラが持つ組み込み関数の積和演算の関数（rmpab/__rmpab 関数、rmpaw/__rmpaw 関数、rmpal/__rmpal 関数）を利用すると使用されます。従って、タスク内で積和演算を行うのであれば、USE_DSP のマクロ定義を (1) に変更する必要があります。ただし、**全タスクが積和演算を行わない、または、ある1つのタスクしか積和演算を行わないのであれば、ACC をタスクコンテキストに含める必要はありません**。その場合は USE_DSP のマクロ定義を (0) のままとしてください。

4. CS+のパートナーOS 対応デバッグプラグイン

CS+のプラグイン機能「CS+ パートナーOS 対応デバッグプラグイン」（以降、「パートナーOS 対応デバッグプラグイン」と記述）を利用することで、プログラム停止時に μ T-Kernel のカーネルオブジェクトの状態を CS+のパネルに表示することが可能です。この章ではパートナーOS 対応デバッグプラグインの利用方法を説明します。

4.1 ダウンロードとインストール

パートナーOS 対応デバッグプラグインはルネサスエレクトロニクス社の Web サイトよりダウンロードし、インストールを行ってください。



図 4.1 パートナーOS 対応デバッグプラグインのインストーラ

4.2 プラグイン機能の有効化

インストール後は CS+を起動し、ツール・メニューの「プラグインの管理」でプラグイン機能が有効になっているかどうかを確認してください。



図 4.2 プラグインの管理

表示された「プラグインの機能」ダイアログにおける追加機能のタブで、「パートナーOS 対応デバッグプラグイン」にチェックが入っていれば機能が有効になっています。もし、チェックが入っていなければ、チェックしてプラグイン機能を有効にしてください。

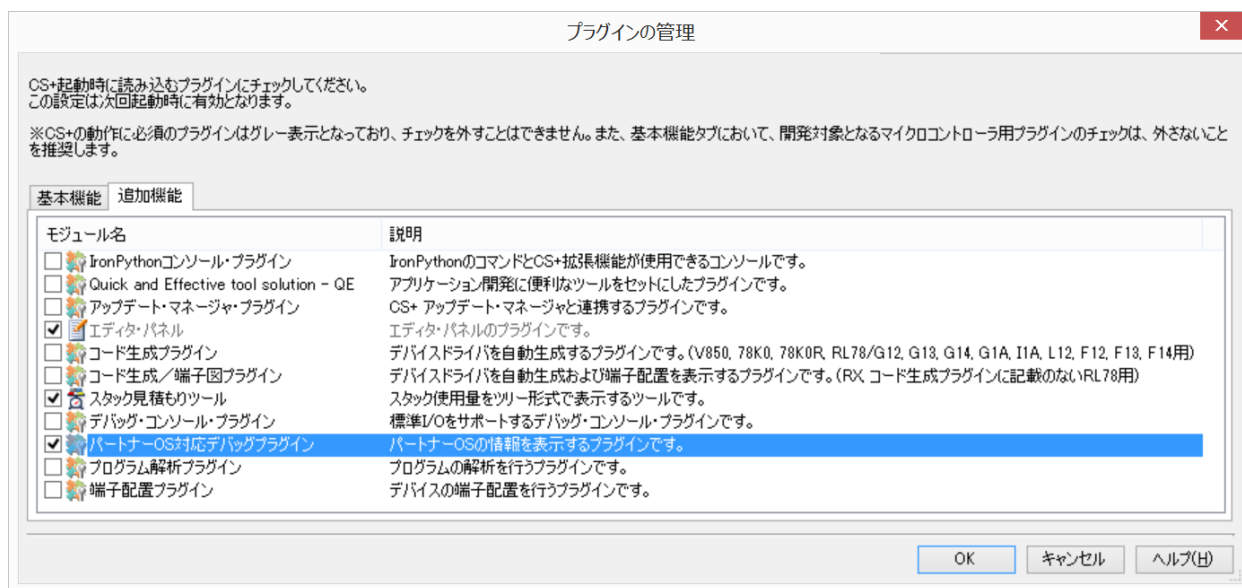


図 4.3 「プラグインの管理」ダイアログ

4.3 OS 定義ファイル

μ T-Kernel のパートナーOS 対応デバッグプラグインは、本来 μ T-Kernel 2.0 仕様用に開発されており、一部の状態表示が機能しません。本機能を μ T-Kernel 3.0 に対応させるためには、 μ T-Kernel 3.0 用の OS 定義ファイルをパートナーOS 対応デバッグプラグインのインストール先フォルダにコピーまたは移動する必要があります。

RX の μ T-Kernel 3.0 用の OS 定義ファイルは、 μ T-Kernel 3.0 の展開フォルダの mtkernel_3/others フォルダにファイル名「UT-RX.mtod」として格納されています。



図 4.4 OS 定義ファイル

この OS 定義ファイルをパートナーOS 対応デバッグプラグインのインストール先フォルダにコピーまたは移動してください。インストール先フォルダは以下のようになります。

[CS+のインストール先フォルダ]/CS+/CC/Plugins/PartnerOS/OSDefine

また、[CS+のインストール先フォルダ]は通常のインストール状態であれば、

C:/Program Files (x86)/Renesas Electronics

または

C:/Program Files/Renesas Electronics

となります。以下のように他の OS 定義ファイルが存在していることでもインストール先は確認できると思います。

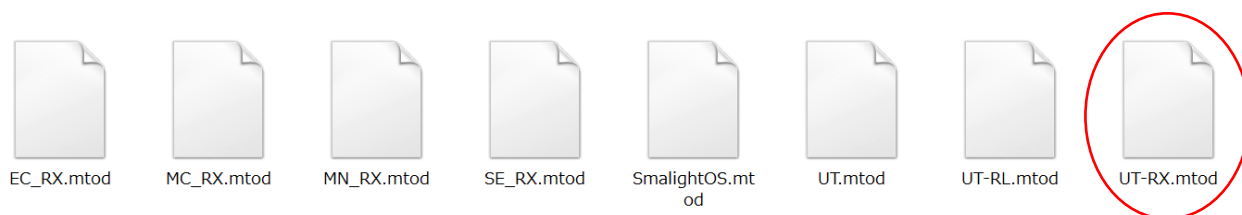


図 4.5 OS 定義ファイルの格納先フォルダ

4.4 パートナーOS 対応デバッグプラグインの機能

パートナーOS 対応デバッグプラグインは、ターゲットハードウェアにプログラムをダウンロード時に以下の「OS 選択」のダイアログにより OS の選択が問い合わせされます。

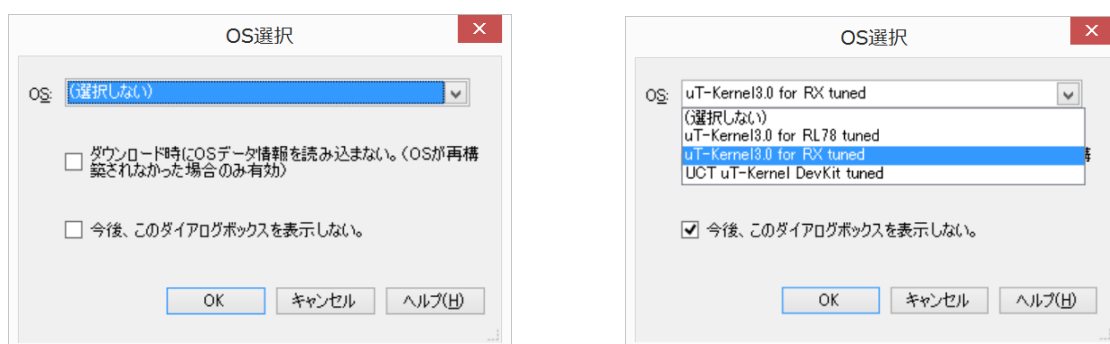


図 4.6 「OS 選択」のダイアログ

ドロップダウンで「**uT-Kernel3.0 for RX tuned**」を選択し、OK してください。そうするとリアルタイム OS のリソース情報パネルが表示されます。



図 4.7 リアルタイム OS のリソース情報パネル

あとは必要に応じて、リソースの追加・削除を行ってください。リソースの追加・削除はアイコンまたは目的のパネル内で右クリックし、表示されたポップアップ・メニューから行うことができます。

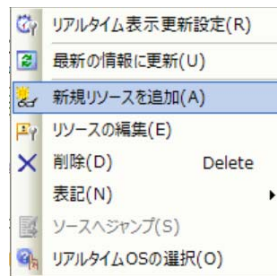
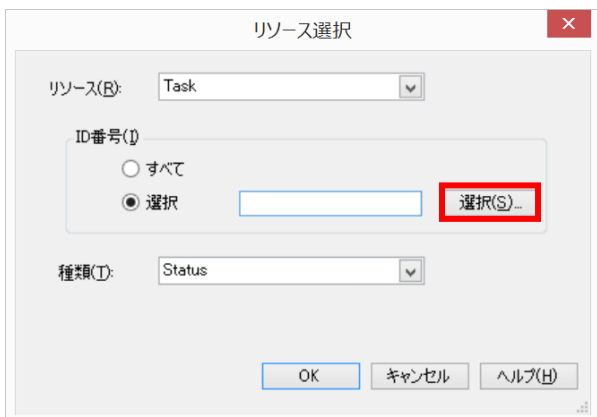
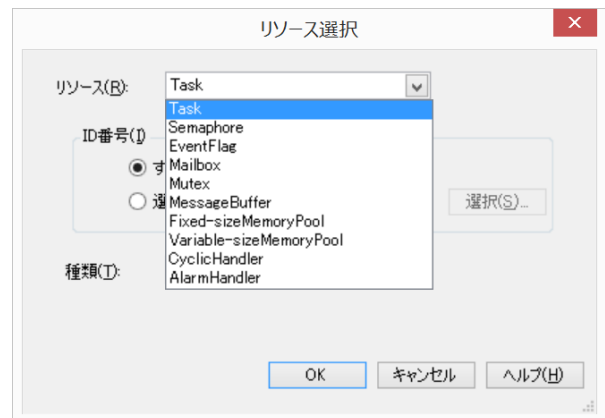
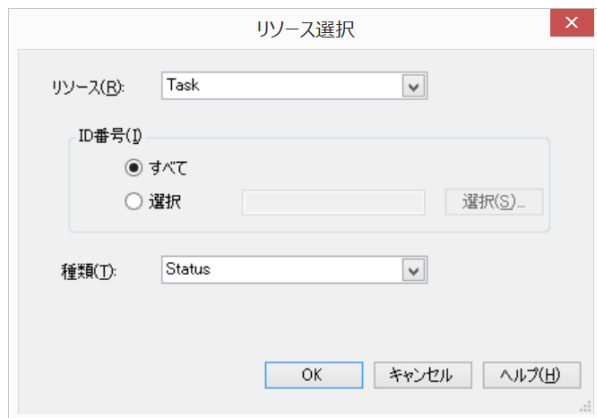


図 4.8 リソース情報の追加・削除

「新規リソースを追加」を実行すると、「リソース選択」のダイアログが表示されますから、表示したい「リソース」（カーネルオブジェクト）、「ID 番号」、「種類」を選択し、OK してください。なお、「種類」の内容は表示する「リソース」の種類によって変化します。



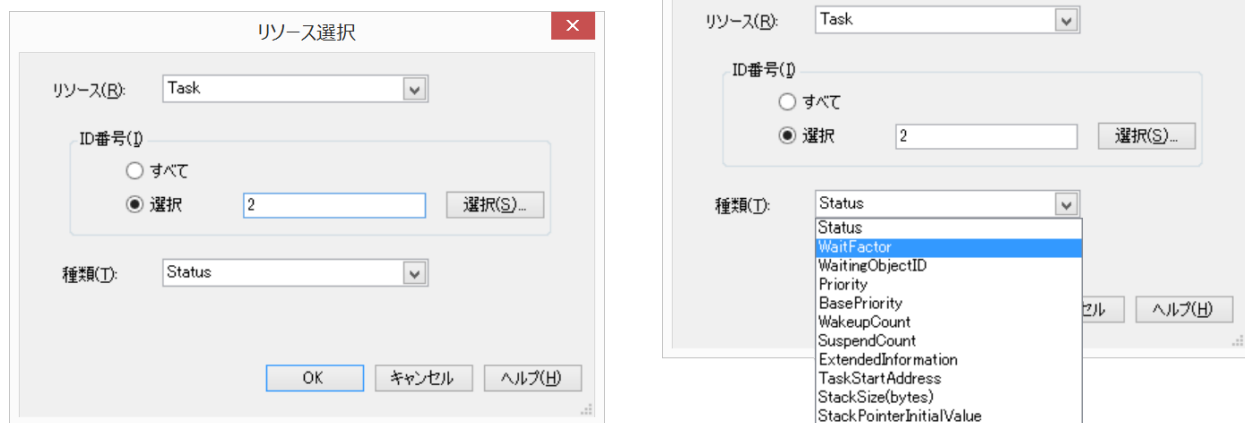


図 4.9 「リソース選択」、「ID 選択」のダイアログ

リソースを選択することにより、リソース情報パネルに必要なカーネルオブジェクトを表示させることが可能となります。

uT-Kernel3.0 for RX tuned リソース情報 - OS or Idle (ID=0)

リソース	ID	名称	種類	値
Task	1	inittsk	Status	WAITING
Task	2	tsk_a	Status	WAITING
Task	2	tsk_a	WaitFactor	SLP
Task	3		Status	NON-EXISTENT
Task	4		Status	NON-EXISTENT

リソース1 リソース2 リソース3 リソース4 リソース5 リソース6 リソース7 リソース8 リソース9

図 4.10 リソースを修正したリソース情報パネル

5. 各種スタックサイズの算出

CS+では、スタック見積もりツールにより、システムで使用するスタックサイズの計算が可能です。また、リアルタイム OS オプションを利用することで「uTK3.0_AP-RX63N-0A 実装仕様書」に記載した各種スタックに対応したスタックサイズの計算が可能です。本章では、スタック見積もりツールの使い方、および各種スタックサイズの計算方法を説明します。

5.1 スタック見積もりツール

スタック見積もりツールは、ツール・メニューの「スタック見積もりツールの起動」で起動します。

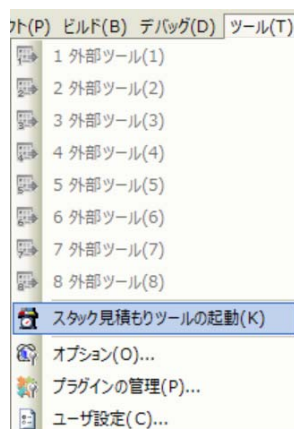


図 5.1 スタック見積もりツールの起動

起動すると現在ビルドの対象となっているプロジェクトのスタック解析結果がスタック見積もりツール（以降、Call Walker と記載）に表示されます。

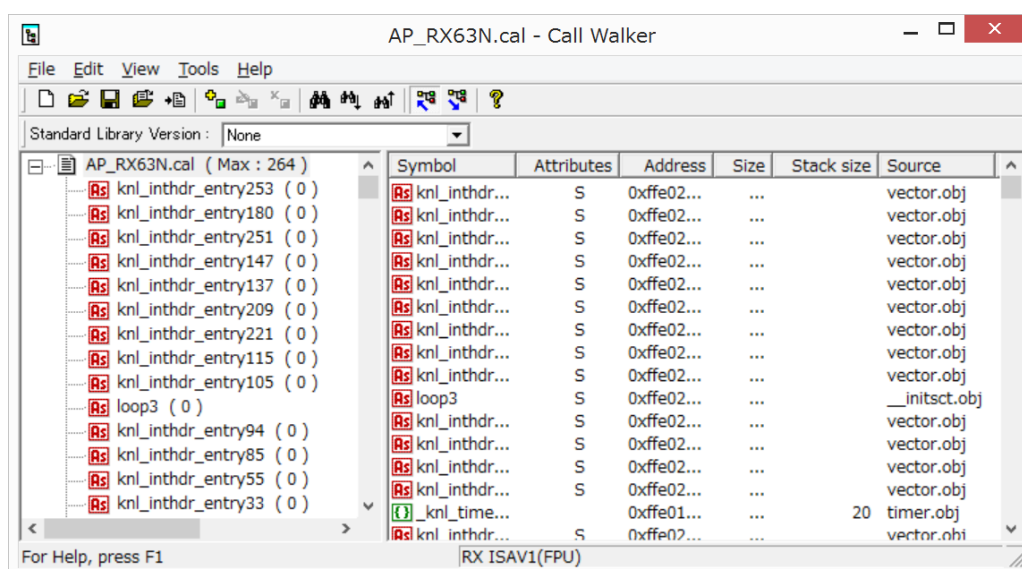


図 5.2 スタック見積もりツールの表示

もし、スタック見積もりツールが起動できない場合は、以下の2点を確認ください。

- ・リンク・オプションが外れていないか

スタック見積もりツールの起動には、リンケージの際にオプション設定が必要となります。サンプルのプロジェクトでは設定済みですが、必要であればリンク・オプションのタブ、その他のカテゴリ、スタック使用量情報ファイルを出力するオプションが設定されていることをご確認ください。

- ・プラグイン機能としてスタック見積もりツールにチェックが入っているか

スタック見積もりツールは CS+ のプラグイン機能として動作します。5.2 節に記載した内容と同じ手順で「プラグインの管理」ダイアログを開き、「スタック見積もりツール」にチェックが入っているかをご確認ください。

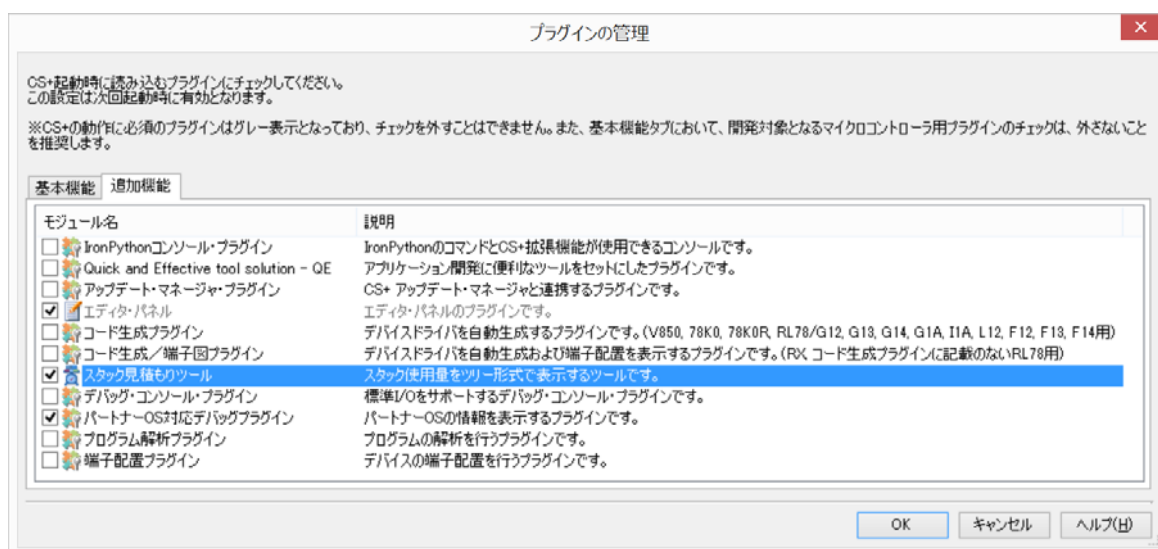


図 5.3 「プラグインの管理」ダイアログ

5.2 リアルタイム OS オプションの設定

デフォルトの状態に表示されたスタック解析結果から、 μ T-Kernel の各種スタックサイズの計算を行うのは容易ではありません。理由は、不要なシンボル（関数やサブルーチン）が表示されてしまうことと、アセンブリ言語で記述されたタスクコンテキストの切り替えに必要なスタックサイズが加味されていないことです。

そこでスタック解析結果が μ T-Kernel 3.0 に対応した表示となるように Call Walker が持つリアルタイム OS オプションを設定します。リアルタイム OS オプションは、Call Walker の Tools メニューの [Realtime OS Option...] から設定します。

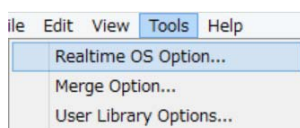


図 5.4 リアルタイム OS オプション

実行すると「Realtime OS Option」のダイアログが開きますから、[Display Symbols for the realtime

OS.] にチェックを入れ、[Browse...] で RTOS ライブラリデータファイルの指定を行ってください。

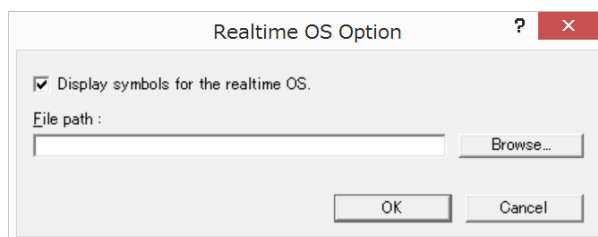


図 6.5 「Realtime OS Option」ダイアログ

RTOS ライブラリデータファイルは、 μ T-Kernel3.0 の展開フォルダ直下の mtkernel_3/others フォルダに CC-RX コンパイラのバージョン対応毎の RXv1 フォルダにファイル名「utkernel.csv」として格納されています。V3.04.00 以降も同様です。

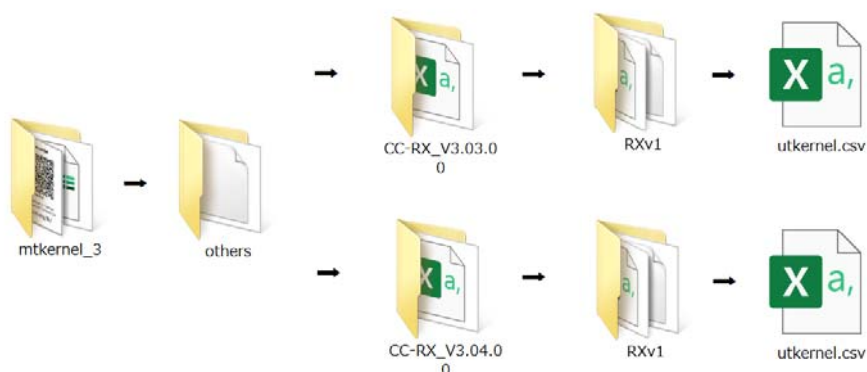


図 5.6 RTOS ライブラリデータファイル

使用している CC-RX のバージョンは、共通オプションのタブ、バージョン選択のカテゴリで確認できます。現時点では、CC-RX の V3.03.00 から V3.04.00 に対応しています。

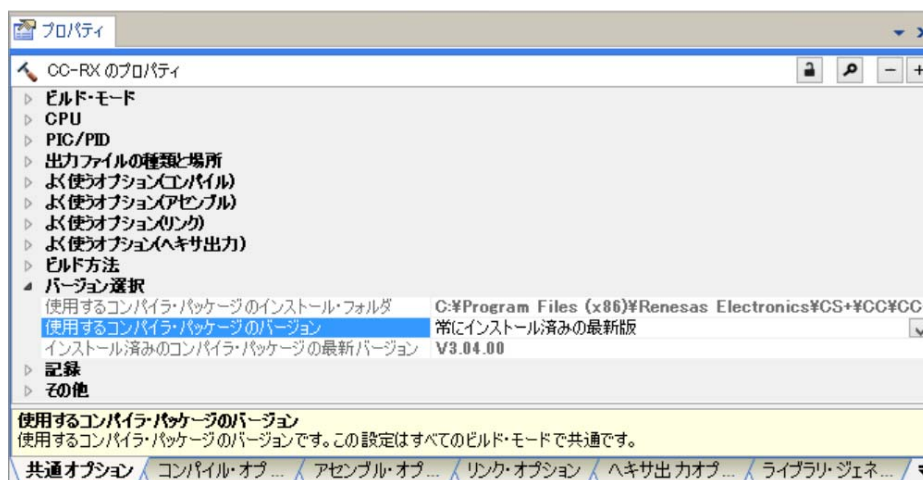


図 5.7 CC-RX コンパイラのバージョン情報

指定後は OK し、Call Walker の表示が次頁のようになることを確認してください。特長としては、ウィンドウ左側の呼び出し情報ビューが以下のように変化しているはずです。

- ・作成したタスク、割込みハンドラの関数呼び出し経路が表示されている
- ・初期タスク（init_task_main）の関数呼び出し経路が表示されている
- ・システムコールの呼び出し経路が RTOS（黄 **RTOS**）のシステムコール名に変化している
- ・システムで使用しているランタイムライブラリ（赤 **As** 表示）のシンボル名が表示されている

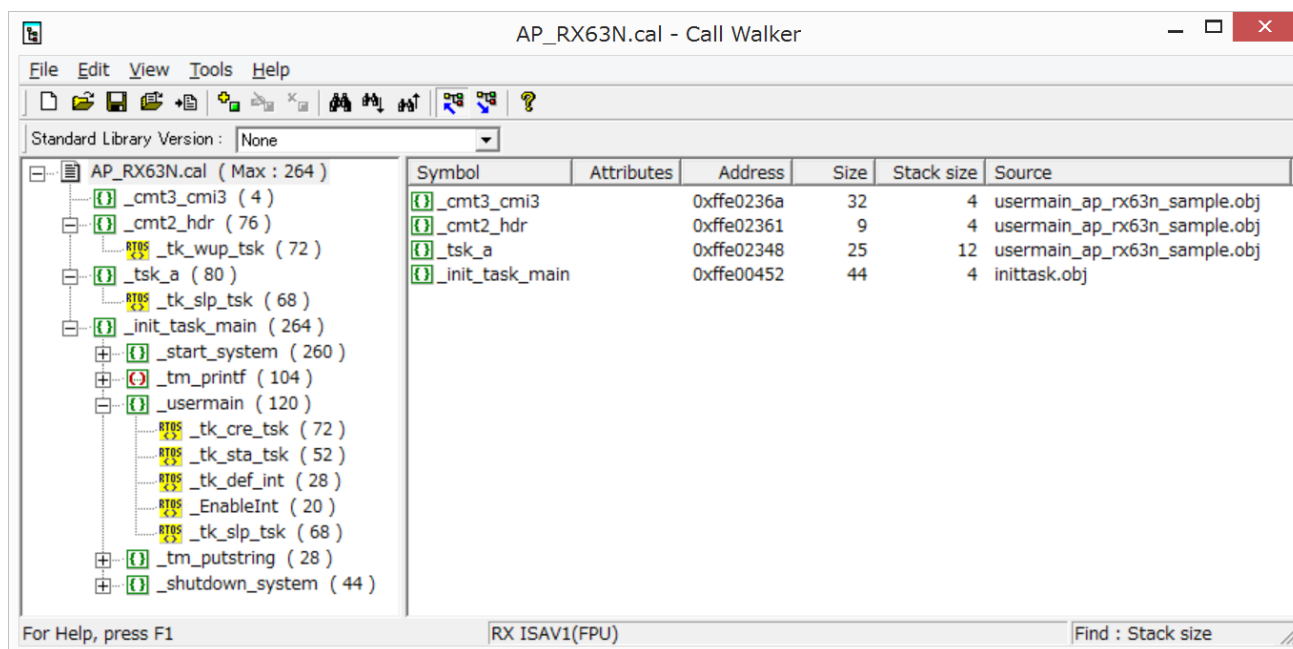


図 5.8 オプション指定後のスタック解析結果

上記の例であれば、

- ・初期タスク（init_task_main）が独自に使用するスタックサイズは 264 バイト
- ・tsk_a が独自に使用するスタックサイズは 80 バイト
- ・cmt2_hdr 割込みハンドラが独自に使用するスタックサイズは 76 バイト

となります。

5.3 各種スタックサイズの計算方法

Call Walker を利用した各種スタックサイズの計算方法を紹介します。なお、実際に確保するスタックサイズは、算出値よりも多少多めに確保することを推奨します。

5.3.1 サイズ計算が必要なスタック領域

サイズ計算が必要なスタック領域は以下の3つです。

(1) タスクスタック

割り込みハンドラ以外で使用するスタックであり、タスク毎に1本ずつ存在し、`tk_cre_tsk` 発行時のスタックサイズ(`T_GSTK.stksz`)で指定します。

(2) 例外スタック

割り込みハンドラで使用するスタックであり、`mtkernel_3/config/config.h(inc)`の `CFN_EXC_STACK_SIZE` で指定します。

5.3.2 OS 管理外割り込みのサイズ

OS 管理外割り込みはクリティカルセクションにおいても受け付けられます。このため、システムで OS 管理外割り込みを使用する場合、すべてのスタック領域に OS 管理外割り込みが使用するスタックサイズを加算する必要があります。

デフォルトの設定では、OS 管理外割り込みは割り込み優先度 13 ~ 15 ですが、コンフィギュレーションによっては割り込み優先度 2 ~ 15 を OS 管理外割り込みとすることが可能です。

結果、OS 管理外割り込みが使用するスタックサイズは以下の計算式で算出します。

割り込み優先度 15 の中で最も多くスタックを消費する OS 管理外割り込み関数が独自に使用するサイズ
+ 割り込み優先度 14 の中で最も多くスタックを消費する OS 管理外割り込み関数が独自に使用するサイズ
+ :
+ :
+ 割り込み優先度 2 の中で最も多くスタックを消費する OS 管理外割り込み関数が独自に使用するサイズ
+ 割り込み優先度 1 の中で最も多くスタックを消費する OS 管理外割り込み関数が独自に使用するサイズ

割り込み優先度が同一の OS 管理外割り込み関数は、ネストして実行されることがありません。従って、割り込み優先度毎に最も多くスタックを消費する OS 管理外割り込み関数のみがスタックサイズ計算の対象となります。

【注意】

OS 管理外割り込みが使用するスタックサイズは、すべてのスタック領域に加算する必要があります。従って、OS 管理外割り込みをサポートするとシステムのスタック消費量が増大します。また、OS 管理外割り込みが使用するスタックサイズの計算式は上記のようになるため、OS 管理外割り込みのレベル数を増やすと

更にスタック消費量が増大します。可能であれば OS 管理外割込みのレベル数は少なく、OS 管理外割込みそのものを使用しないことを推奨します。

5.3.3 タスクコンテキストのサイズ

タスクコンテキストのサイズは、タスクコンテキストに FPSW や ACC を含めるかどうかで変化します。従って、コンフィグレーションにおける USE_FPU と USE_DSP の設定値で変化します。タスクコンテキストのサイズは以下の算出表の値を用います。

表 5.1 タスクコンテキストのサイズ

USE_DSP \ USE_FPU	USE_FPU	(1) : 使用する	(0) : 使用しない
	USE_DSP	(1) : 使用する	(0) : 使用しない
(1) : 使用する		84	80
(0) : 使用しない		76 (デフォルト)	72

5.3.4 タスクスタックのサイズ

タスクスタックのサイズは以下の計算式で算出します。

各タスクが独自に使用するサイズ + タスクコンテキストのサイズ + OS 管理外割込みのサイズ

図 5.8 の例であれば、tsk_a のスタックサイズは以下のようになります (OS 管理外割込みは使用していない、タスクコンテキストのサイズはデフォルトと仮定しています)。

・ tsk_a : 80 バイト + 76 バイト + 0 バイト = 156 バイト

また、初期タスク (init_task_main) も同じ計算方法となりますが、初期タスクのスタックサイズは mtkernel_3/include/sys/inittask.h の INITTASK_STKSZ で指定します。図 5.8 の例であれば、初期タスクのスタックサイズは以下のようになります。

・ init_task_main : 264 バイト + 76 バイト + 0 バイト = 340 バイト

5.3.5 例外スタックのサイズ

デフォルトの設定では、OS 管理内割込みは割込み優先度 1 ~ 12 ですが、コンフィグレーションによって割込み優先度 1 ~ 15 や割込み優先度 1 のみに変更することが可能です。

結果、対象外となる割込み優先度もありますが、例外スタックは以下の計算式で算出します。

割込み優先度 15 の中で最も多くスタックを消費する OS 管理内割込みハンドラが独自に使用するサイズ + 割込み優先度 14 の中で最も多くスタックを消費する OS 管理内割込みハンドラが独自に使用するサイズ

+ :
 + :
 + 割込み優先度 2 の中で最も多くスタックを消費する OS 管理内割込みハンドラが独自に使用するサイズ
 + 割込み優先度 1 の中で最も多くスタックを消費する OS 管理内割込みハンドラが独自に使用するサイズ
 + 36 バイト * 割込みのネスト数
 + OS 管理外割込みのサイズ

割込み優先度が同一の OS 管理内割込みハンドラは、ネストして実行されることがありません。従って、割込み優先度毎に最も多くスタックを消費する OS 管理内割込みハンドラのみがスタックサイズ計算の対象となります。

また、割込みが発生すると CPU 内部レジスタの退避に 36 バイトのスタックサイズを消費します。従って、36 バイトに割込みのネスト数を乗算したサイズが必要となります。なお、割込みネスト数とは、使用した OS 管理内割込み優先度レベルの総数です。

なお、デフォルトの設定では、割込み優先度 10 にシステムタイマの割込みが存在します。デフォルト設定のままであれば、割込み優先度 10 の使用するサイズには、システムタイマの割込みを考慮する必要があります。システムタイマの割込みは 148 バイトのスタックサイズを消費します。ただし、この数値は周期ハンドラやアラームハンドラが実行されない時のサイズです。もし、作成した周期ハンドラやアラームハンドラの中に独自に使用するサイズが 120 バイトを超えるものがある場合、周期ハンドラ、またはアラームハンドラが使用するサイズに 28 バイトを加算した値でシステムタイマ割込みのスタックサイズを計算する必要があります。

《システムタイマ割込みのスタックサイズ》

- ・スタックサイズの消費が 120 バイトを超える周期ハンドラ/アラームハンドラが存在しない場合
148 バイト
- ・スタックサイズの消費が 120 バイトを超える周期ハンドラ/アラームハンドラが存在する場合
周期ハンドラ/アラームハンドラが独自に使用するサイズ + 28 バイト

なお、周期ハンドラやアラームハンドラは、同じシステムタイマ割込みでタイムアウトが発生してもシーケンシャルに実行されるため、作成した全ての周期ハンドラやアラームハンドラを対象とする必要はありません。同一割込み優先度の割込みハンドラと同様に最も多くスタックを消費する周期ハンドラ、またはアラームハンドラのみが対象となります。

図 5.8 の例であれば、周期ハンドラやアラームハンドラは存在せず、割込み優先度 9 に cmt2_hdr 割込みハンドラが存在しており、スタックサイズは以下ようになります (OS 管理外割込みは使用していないと仮定しています)。

割込み優先度 12 : 0 バイト
 + 割込み優先度 11 : 0 バイト

+ 割込み優先度 10	: 148 バイト (システムタイマ割込み)
+ 割込み優先度 9	: 76 バイト
+ 割込み優先度 8	: 0 バイト
+ :	
+ 割込み優先度 1	: 0 バイト
+ 割込みのネスト数	: 72 バイト (36*2 ⇒割込みネスト数は 2)
+ OS 管理外割込み	: 0 バイト
合計	: 296 バイト

【備考】

システムタイマ割込みはスタック領域を比較的多く消費します。これはタイムアウト機能を持つシステムコールのタイムアウト処理を行うためです。もし、システムでタイムアウト機能を使用しない場合 (TMO_FEVR と TMO_POL は除く)、システムタイマ割込みのスタックサイズを少なくすることができます (サンプルプログラムのように tsk_a で tk_slp_tsk(500)のようにタイムアウト機能を利用している場合、以下の計算式は利用できません)。その場合は以下の計算式でシステムタイマ割込みのスタックサイズを計算してください。

《タイムアウト機能を使用しない場合のシステムタイマ割込みのスタックサイズ》

- ・スタックサイズの消費が 20 バイトを超える周期ハンドラ/アラームハンドラが存在しない場合
48 バイト
- ・スタックサイズの消費が 20 バイトを超える周期ハンドラ/アラームハンドラが存在する場合
周期ハンドラ/アラームハンドラが独自に使用するサイズ + 28 バイト

【その他の注意事項】

uTK3.0 共通実装仕様書の 7.2.2 項に記載された周期ハンドラないしアラームハンドラ存在する場合、tk_cre_cyc には生成する周期ハンドラが使用するスタックサイズを、tk_sta_alm には起動するアラームハンドラが使用するスタックサイズを加算する必要があります。

5.3.6 スタックサイズの算出例

スタックサイズの算出で難しいのは OS 管理外割込みのサイズと例外スタックのサイズです。各割込みレベル毎に最も多くスタックを消費するハンドラを選択して加算するのですが、その算出例を紹介します。

(1) OS 管理外割込みのサイズ

デフォルトの設定のまま、割込み優先度 13~15 を OS 管理外割込みとし、下表のように OS 管理外割込みを使用していたものとします。割込み優先度は割込みコントローラ (INTC) の割り込み要因プライオリティレジスタ (IPRn) に設定した値、スタックサイズは Call Walker の表示結果より求めます。

表 5.2 OS 管理外割込みのサポート状況

OS 管理外割込み	割込み優先度	スタックサイズ
int_irq0 割込み関数	13	24 バイト
int_irq1 割込み関数	13	32 バイト
int_irq2 割込み関数	14	20 バイト
int_sci1 割込み関数	13	40 バイト
int_sci2 割込み関数	15	36 バイト
int_tmr0 割込み関数	14	32 バイト
int_tmr2 割込み関数	15	24 バイト

上記の場合、各割込み優先度で最も多くスタックを使用する割込み関数は以下のようになります。

- ・ 割込み優先度 13 ⇒ int_sci1 割込み関数の 40 バイトが最大
- ・ 割込み優先度 14 ⇒ int_tmr0 割込み関数の 32 バイトが最大
- ・ 割込み優先度 15 ⇒ int_sci2 割込み関数の 36 バイトが最大

結果、これらの数値を全て加算した $40 + 32 + 36 = 108$ バイト が OS 管理外割込みのサイズとなります。この数値をタスクスタックのサイズ、例外スタックのサイズに加算することになります。

(2) 例外スタックのサイズ

デフォルトの設定のまま、割込み優先度 1~12 を OS 管理内割込みとし、下表のように OS 管理内割込みを使用していたものとします。また、システムタイマの割込み優先度はデフォルト設定の 10 でスタックサイズ 148 バイト、OS 管理外割込みのサイズは 108 バイトとします。

表 5.3 OS 管理内割込みのサポート状況

OS 管理内割込み	割込み優先度	スタックサイズ
irq3_hdr 割込みハンドラ	12	60 バイト
irq4_hdr 割込みハンドラ	10	84 バイト
irq5_hdr 割込みハンドラ	9	56 バイト
mtu0_hdr 割込みハンドラ	7	64 バイト
mtu1_hdr 割込みハンドラ	9	72 バイト
mtu2_hdr 割込みハンドラ	7	16 バイト
tmr1_hdr 割込みハンドラ	7	48 バイト
システムタイマ割込み	10	148 バイト

上記の場合、各割込み優先度で最も多くスタックを使用する割込みハンドラは以下のようになります。

- ・ 割込み優先度 12 ⇒ irq3_hdr 割込みハンドラの 60 バイトが最大
- ・ 割込み優先度 10 ⇒ システムタイマ割込みの 148 バイトが最大
- ・ 割込み優先度 9 ⇒ mtu1_hdr 割込みハンドラの 72 バイトが最大
- ・ 割込み優先度 7 ⇒ mtu0_hdr 割込みハンドラの 64 バイトが最大

結果、 $60 + 148 + 72 + 64 + 36 * 4 + 108 = 596$ バイト が例外スタックのサイズとなります（割込みネスト数は、割込み優先度 12、10、9、7 の 4 個）。

5.3.7 デバイスドライバのシステムコールのスタックサイズ

以下の表 5.2 に示すデバイスドライバのシステムコールでは、ユーザ作成の処理関数が内部で呼ばれます。このため、各システムコールのスタックサイズにはユーザ作成の処理関数が使用するスタックサイズを考慮しなければなりません。ただし、処理関数が使用するサイズは無条件で加算する必要はなく、一定のサイズを超えた場合に加算することになります（システムタイマ割込みのタイムイベントハンドラと同じ考え方）。表 5.4 に示す条件に従って計算を行ってください。

表 5.4 処理関数のサイズを考慮するデバイスドライバのシステムコール

システムコール	処理関数	加算条件	計算式
tk_opn_dev	openfn	140 バイトを超える	超えない : 200 バイト 超える : 60 バイト + openfn のサイズ
tk_cls_dev	closefn	168 バイトを超える	超えない : 244 バイト 超える : 76 バイト + closefn のサイズ
	abortfn	128 バイトを超える	超えない : 244 バイト 超える : 116 バイト + abortfn のサイズ
	waitfn	128 バイトを超える	超えない : 244 バイト 超える : 116 バイト + waitfn のサイズ
tk_rea_dev	execfn	128 バイトを超える	超えない : 204 バイト 超える : 76 バイト + execfn のサイズ
tk_srea_dev	execfn	128 バイトを超える	超えない : 220 バイト 超える : 92 バイト + execfn のサイズ
	waitfn	136 バイトを超える	超えない : 220 バイト 超える : 84 バイト + waitfn のサイズ
tk_wri_dev	execfn	128 バイトを超える	超えない : 204 バイト 超える : 76 バイト + execfn のサイズ
tk_swri_dev	execfn	128 バイトを超える	超えない : 220 バイト 超える : 92 バイト + execfn のサイズ
	waitfn	136 バイトを超える	超えない : 220 バイト 超える : 84 バイト + waitfn のサイズ
tk_wai_dev	waitfn	128 バイトを超える	超えない : 196 バイト 超える : 68 バイト + waitfn のサイズ
tk_sus_dev	eventfn	96 バイトを超える	超えない : 180 バイト 超える : 84 バイト + eventfn のサイズ
tk_evt_dev	eventfn	128 バイトを超える	超えない : 168 バイト 超える : 40 バイト + eventfn のサイズ

6. C 言語規格準拠の仕様

6.1 C 言語規格に準拠していない部分

μ T-Kernel3.0 仕様の API には、一部 C 言語規格の仕様に準拠していない部分があります。その主な内容は以下の 3 つです。

- ・ 文字列の型

μ T-Kernel3.0 仕様の API では文字列を unsigned char 型を指すポインタ型 (UB *型) で扱っているが、C 言語の規格で文字列は char 型を指すポインタ型である。このため、文字列が引数となっている API に対して、"" (ダブルクォーテーション) の文字列を指定すると警告が表示されてしまう。

もっとも、char 型を unsigned char 型で扱う処理系では、これらの問題が発生せず、char 型を signed char 型で扱う処理系や、char 型を signed char 型や unsigned char 型とは異なる型として扱う処理系では問題となる。

そこで処理系依存の char 型を VB 型として宣言し、API では文字列を VB 型を指すポインタ型とすることで C 言語規格に準拠できる。

- ・ 必ずキャストが必要な API

tk_snd_mbx と tk_rcv_mbx は必ずキャスト演算子が必要である。理由は、ユーザシステム毎に異なるメッセージフォーマットの型を T_MSG 型に固定しているからです。

また、tk_get_mpl と tk_get_mpf も必ずキャスト演算子が必要である。理由は、void *型は汎用ポインタ型ですが、void **型は汎用ポインタ型ではなく、汎用ポインタ型しか指せないポインタ型だからです。引数の意味としては正しいのですが、汎用ポインタ型の利用と言う点では使い方が間違っており、色々なポインタ型が引数となる部分に対しては、一律に void *型の汎用ポインタ型を用いることで無駄なキャストを回避できる。

- ・ 関数を指すポインタ型

μ T-Kernel3.0 仕様では関数を指すポインタ型として FP 型や FUNCP 型を宣言しているが、引数の型を明確にしていない。引数の型を明確にしない場合、引数は何でも良いという意味にはならず、言語仕様では引数の個数や型のチェックを行わないという意味になってしまう。

このため、関数のアドレス値を指定する API では警告が表示されてしまう。これを回避するためには引数の型を明確にした形式で関数ポインタを宣言する必要がある。

6.2 マクロ定義による API の変更

「CLANGSPEC」定義マクロをオプションで与えることにより、 μ T-Kernel3.0 の API を C 言語規格準拠の API に変更できます。以下に「CLANGSPEC」定義マクロにより API が変化するシステムコールの一覧を示します。なお、変更されるのは API の引数の型だけであり、引数の意味は μ T-Kernel3.0 の API と同じです。

6.2.1 T-Monitor のライブラリ

【 μ T-Kernel3.0仕様】

```
IMPORT INT  tm_getline( UB *buff );
IMPORT INT  tm_putstring( const UB *buff );
IMPORT INT  tm_printf( const UB *format, ... );
IMPORT INT  tm_sprintf( UB *str, const UB *format, ... );
```

【C言語規格準拠 (「CLANGSPEC」定義マクロ指定時)】

```
IMPORT INT  tm_getline( VB *buff );
IMPORT INT  tm_putstring( const VB *buff );
IMPORT INT  tm_printf( const VB *format, ... );
IMPORT INT  tm_sprintf( VB *str, const VB *format, ... );
```

6.2.2 高速ロック・マルチロックライブラリ

【 μ T-Kernel3.0仕様】

```
EXPORT ER CreateMLock( FastMLock *lock, CONST UB *name );
EXPORT ER CreateLock( FastLock *lock, CONST UB *name );
```

【C言語規格準拠 (「CLANGSPEC」定義マクロ指定時)】

```
EXPORT ER CreateMLock( FastMLock *lock, CONST VB *name );
EXPORT ER CreateLock( FastLock *lock, CONST VB *name );
```

6.2.3 カーネルオブジェクト生成時のパラメータパケット

【 μ T-Kernel3.0仕様】

```
typedef void    (*FP)();                                /* Function address general */
① タスク生成情報 (T_CTSK の内容)
    FP      task;                                        /* Task startup address */
    UB      dsname[OBJECT_NAME_LENGTH];                /* Object name */
② 周期ハンドラ生成情報 (T_CCYC の内容)
    FP      cychdr;                                      /* Cycle handler address */
    UB      dsname[OBJECT_NAME_LENGTH];                /* Object name */
③ アラームハンドラ生成情報 (T_CALM の内容)
    FP      almhdr;                                      /* Alarm handler address */
    UB      dsname[OBJECT_NAME_LENGTH];                /* Object name */
④ 割り込みハンドラ定義情報 (T_DINT)
    FP      inthdr;                                      /* Interrupt handler address */
⑤ その他の生成情報 (T_GSEM、T_CFLG、T_CMBX、T_CMTX、T_CMBF、T_CMPF、T_CMPL の内容)
    UB      dsname[OBJECT_NAME_LENGTH];                /* Object name */
⑥ サブシステム定義情報 (T_DSSY の内容)
    FP      svchdr;                                      /* Extended SVC handler address */
```

⑦ デバイスドライバ定義情報

```

FP      openfn;                /* Open function */
FP      closefn;               /* Close function */
FP      execfn;                /* Execute function */
FP      waitfn;                /* Completion wait function */
FP      abortfn;               /* Abort function */
FP      eventfn;               /* Event function */

```

【C 言語規格準拠 (「CLANGSPEC」 定義マクロ指定時)】

```

typedef void (*TSKFP) (INT, void *); /* Task Function address general */
typedef void (*TIMFP) (void *);      /* Timevent Function address general */
typedef void (*INTFP) (UINT);        /* Interrupt Function address general */
typedef INT  (*SVCFP) (void *, FN);  /* SVC Handler address general */
typedef ER   (*OPNFP) (ID, UINT, void *); /* Open function address general */
typedef ER   (*CLSFP) (ID, UINT, void *); /* Close function address general */
typedef ER   (*EXCFP) (T_DEVREQ *, TMO, void *); /* Execute function address general */
typedef INT  (*WAIFP) (T_DEVREQ *, INT, TMO, void *); /* Completion wait function address general */
typedef ER   (*ABTFP) (ID, T_DEVREQ *, INT, void *); /* Abort function address general */
typedef INT  (*EVTFP) (INT, void *, void *); /* Event function address general */

```

① タスク生成情報 (T_CTSK の内容)

```

TSKFP  task;                  /* Task startup address */
VB      dsname[OBJECT_NAME_LENGTH]; /* Object name */

```

② 周期ハンドラ生成情報 (T_CCYC の内容)

```

TIMFP  cychdr;                /* Cycle handler address */
VB      dsname[OBJECT_NAME_LENGTH]; /* Object name */

```

③ アラームハンドラ生成情報 (T_CALM の内容)

```

TIMFP  almhdr;                /* Alarm handler address */
VB      dsname[OBJECT_NAME_LENGTH]; /* Object name */

```

④ 割込みハンドラ定義情報 (T_DINT の内容)

```

INTFP  inthdr;                /* Interrupt handler address */

```

⑤ その他の生成情報 (T_CSEM、T_CFLG、T_CMBX、T_CMTX、T_CMBF、T_CMPF、T_CMPL の内容)

```

VB      dsname[OBJECT_NAME_LENGTH]; /* Object name */

```

⑥ サブシステム定義情報 (T_DSSY の内容)

```

SVCFP  svchdr;                /* Extended SVC handler address */

```

⑦ デバイスドライバ定義情報

```

OPNFP  openfn;                /* Open function */
CLSFP  closefn;               /* Close function */
EXCFP  execfn;                /* Execute function */
WAIFP  waitfn;                /* Completion wait function */
ABTFP  abortfn;               /* Abort function */
EVTFP  eventfn;               /* Event function */

```

6.2.4 μ T-Kernel/OS システムコール

【 μ T-Kernel3.0 仕様】

```
IMPORT ER tk_snd_mbx( ID mbxid, T_MSG *pk_msg );
IMPORT ER tk_rcv_mbx( ID mbxid, T_MSG **ppk_msg, TMO tmout );
IMPORT ER tk_get_mpl( ID mplid, SZ blksize, void **p_blk, TMO tmout );
IMPORT ER tk_get_mpf( ID mpfid, void **p_blf, TMO tmout );
```

【C 言語規格準拠 (「CLANGSPEC」定義マクロ指定時)】

```
IMPORT ER tk_snd_mbx( ID mbxid, void *pk_msg );
IMPORT ER tk_rcv_mbx( ID mbxid, void *ppk_msg, TMO tmout );
IMPORT ER tk_get_mpl( ID mplid, SZ blksize, void *p_blk, TMO tmout );
IMPORT ER tk_get_mpf( ID mpfid, void *p_blf, TMO tmout );
```

6.2.5 μ T-Kernel/SM システムコール

【 μ T-Kernel3.0 仕様】

T_LDEV の内容

```
UB devnm[L_DEVNM]; /* Physical device name */
IMPORT ID tk_opn_dev( CONST UB *devnm, UINT omode );
IMPORT ID tk_get_dev( ID devid, UB *devnm );
IMPORT ID tk_ref_dev( CONST UB *devnm, T_RDEV *pk_rdev );
IMPORT ID tk_def_dev( CONST UB *devnm, CONST T_DDEV *pk_ddev, T_IDEV *pk_iddev );
```

【C 言語規格準拠 (「CLANGSPEC」定義マクロ指定時)】

T_LDEV の内容

```
VB devnm[L_DEVNM]; /* Physical device name */
IMPORT ID tk_opn_dev( CONST VB *devnm, UINT omode );
IMPORT ID tk_get_dev( ID devid, VB *devnm );
IMPORT ID tk_ref_dev( CONST VB *devnm, T_RDEV *pk_rdev );
IMPORT ID tk_def_dev( CONST VB *devnm, CONST T_DDEV *pk_ddev, T_IDEV *pk_iddev );
```

6.2.6 μ T-Kernel/DS システムコール

【 μ T-Kernel3.0 仕様】

```
IMPORT ER td_ref_dsname( UINT type, ID id, UB *dsname );
IMPORT ER td_set_dsname( UINT type, ID id, CONST UB *dsname );
```

【C 言語規格準拠 (「CLANGSPEC」定義マクロ指定時)】

```
IMPORT ER td_ref_dsname( UINT type, ID id, VB *dsname );
IMPORT ER td_set_dsname( UINT type, ID id, CONST VB *dsname );
```

7. アプリケーションプログラムの作成

アプリケーションプログラムは、OS とは別にアプリ用のディレクトリを作成して、そこにソースコードを置き、OS と一括でコンパイル、リンクすることを推奨します。

公開している μ T-Kernel3.0 を改変したソースコードにはサンプルのアプリケーションが含まれています。/kernel/usermain ディレクトリが、サンプルのアプリケーションのディレクトリであり、これをユーザのアプリケーションのディレクトリに置き換えても構いません。

アプリケーションには usermain 関数を定義します。OS は起動後に初期タスクから usermain 関数を実行します。詳細は μ T-Kernel3.0 共通実装仕様書「5.2.3 ユーザ定義メイン関数 usermain」を参照してください。

アプリケーションから OS の機能を使用する場合は、以下のようにヘッダファイルのインクルードを行います。

```
#include <tk/tkernel.h>
```

また、T-Monitor 互換ライブラリを使用する場合は、さらに以下のインクルードが必要であります。

```
#include <tm/tmonitor.h>
```

μ T-Kernel3.0 の機能については μ T-Kernel3.0 仕様書を参照してください。

8. 問い合わせ先

本実装に関する問い合わせや他のRXファミリへのポーティングに関する相談は以下のメールアドレス宛にお願い致します。

yuji_katori@yahoo.co.jp

トロンフォーラム学術・教育WGメンバ
鹿取 祐二（かとり ゆうじ）

なお、上記のメールアドレスは余儀なく変更される場合がありますが、その際はご了承ください。

以上