μT-Kernel3.0
C-Frist（RL78 S3コア）　向け
構築手順書

Version.01.01.02

2023.2.3

目次

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μT-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 RL78 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μT-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	μT-Kernel3.00.00	RL78用に改変したソースコード
実機	C-First	マルツエレック製
搭載マイコン	RL78/G14 R5F104LE	ルネサスエレクトロニクス製
デバッガ	EZ-Emulator	C-Frist搭載（標準）
	E1/E2等	JTAGデバッガ（拡張用）

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-RL）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以下の通りです。

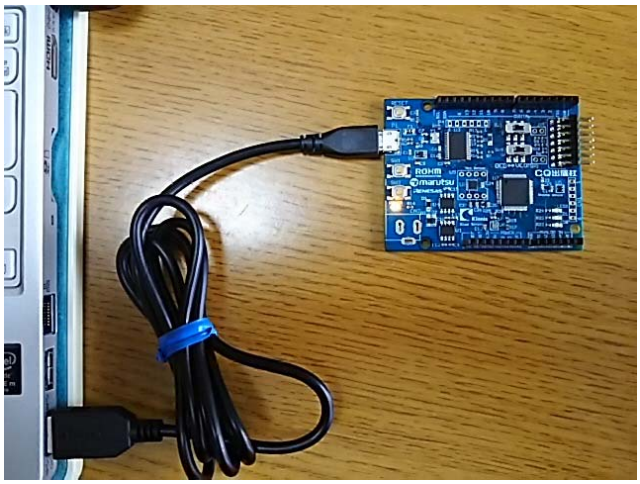


図 2.1　C-First のハードウェア構成（RL78 EZ Emulator 使用時）

2.2　デバッガ

ターゲットハードウェアとホスト PC を USB で接続して開発を行います。ターゲットハードウェアには RL78 EZ Emulator の IC が搭載されており、JTAG デバッガは必要ありません。ただし、RL78 EZ Emulator が使用する USB を仮想 COM ポートとして利用するのであれば、P2 にコネクタを実装し、そこに専用のプローブで JTAG デバッガを接続することも可能です。その場合は E1 や E2 等の JTAG デバッガが利用可能です。

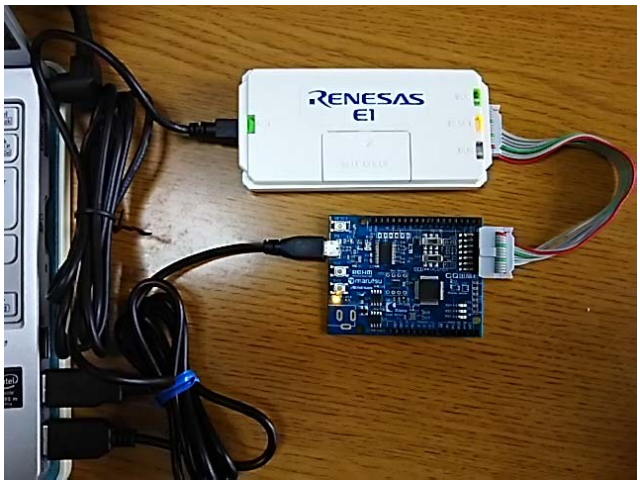


図 2.2　C-First のハードウェア構成（JTAG デバッガ使用時）

2.3　デバッグコンソール

JTAG デバッガを利用している場合に限り、USB を仮想 COM ポートの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μT-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、μT-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。

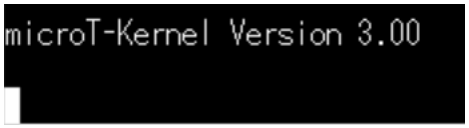


図 2.3　μT-Kernel の起動メッセージ例

なお、mtkernel_3/lib/libtm/libtm_conf.h における以下のマクロ名を 1(有効)から 0(無効)に変更することで仮想 COM ポートに使用している UART チャネルの制御関数を空関数とし、目的の UART チャネルをユーザシステムで別の用途に使用することが可能となります。

```
/* Select a communication port */
#define USE_COM_C_FIRST         (1)
```

【注意】

ターゲットハードウェアに R6 の抵抗が接続されていると、ホスト PC からのキー入力により、JTAG デバッガがブレークすることがあります。JTAG デバッガを使用し、USB をデバッグコンソール用の仮想 COM ポートとして使用する場合、R6 の抵抗を取り外す必要があります。

3.　開発環境の準備

本ソフトをビルド、実行するためには機器の準備と開発環境のインストールが必要になります。

3.1　ハードウェア

ターゲットハードウェアを用意し、USB ケーブルでホスト PC と接続します（図 2.1 参照）。その際、SW3 と SW4 のスライドスイッチは USB コネクタ側（OCD 側）にスライドしてください。USB ケーブルでホスト PC と接続する場合、電源供給は USB より行われます。

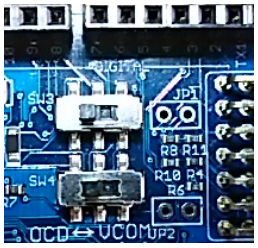


図 3.1　RL78 EZ Emulator 使用時の SW3/SW4 のスライドスイッチ

また、JTAG デバッガを利用される場合は、　ターゲットハードウェアと JTAG デバッガを用意し、増設の P2 コネクタよりホスト PC と接続します（図 2.2 参照）。その際、SW3 と SW4 のスライドスイッチは P2 コネクタ側（VCD 側）にスライドしてください。

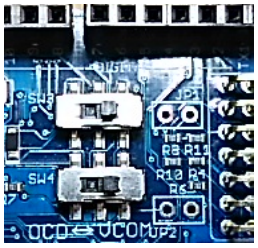


図 3.2　JTAG デバッガ使用時の SW3/SW4 のスライドスイッチ

JTAG デバッガを利用する場合、電源供給は JTAG デバッガから行うことも可能です。2.3 節で紹介したデバッグコンソールを利用するために USB ケーブルも接続しているのであれば、電源供給は USB より行います。一方、デバッグコンソールを利用せず、USB ケーブルを接続していないのであれば、JTAG デバッガより供給してください。

3.2　開発環境のインストール

CS+のインストーラの説明に従って、開発環境および必要であれば JTAG デバッガのドライバをインストールしてください。標準のインストールであれば、JTAG デバッガのドライバは自動的にインストールされます。また、C-First 付属の DVD を使用し、RL78 EZ Emulator のドライバをインストールしてください。

4.　μT-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。

4.1　μT-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μT-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/C_First.mtpj

図 4.1　CS+のプロジェクト・ファイル

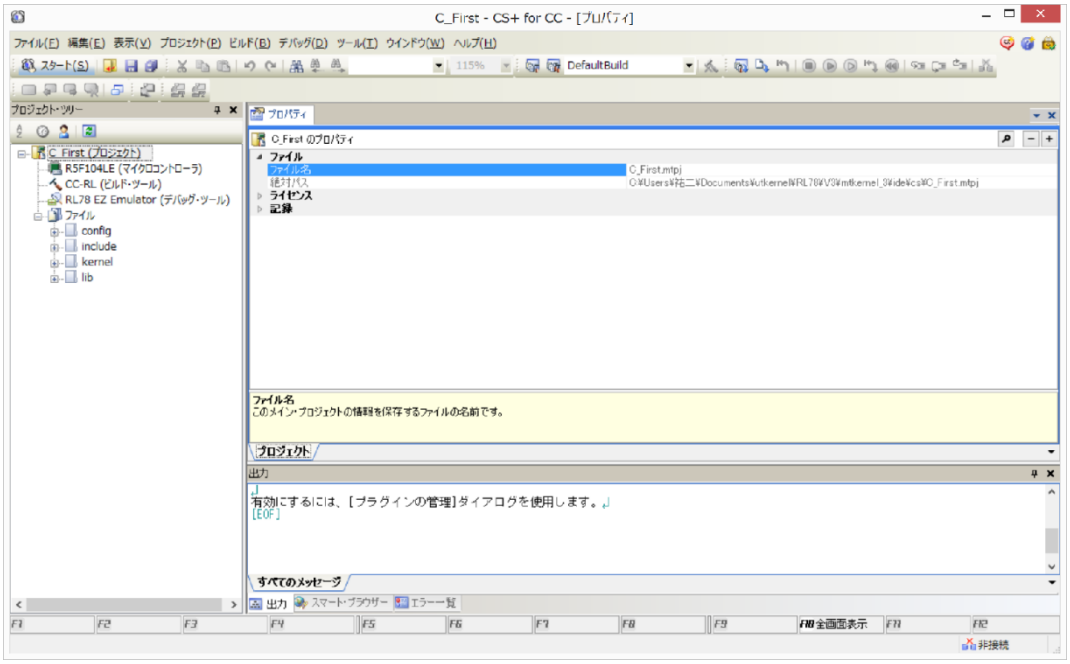


図 4.2　CS+起動直後の画面

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 RL78 EZ Emulator が選択されています。

通常（RL78 EZ Emulator を使用する場合）は変更する必要がありませんが、JTAG 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。

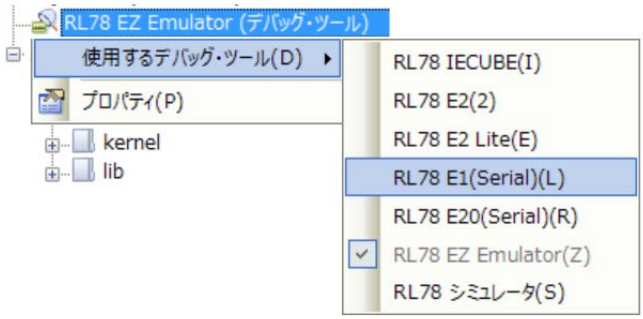


図 4.3　デバッグ・ツールの切り替え

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。

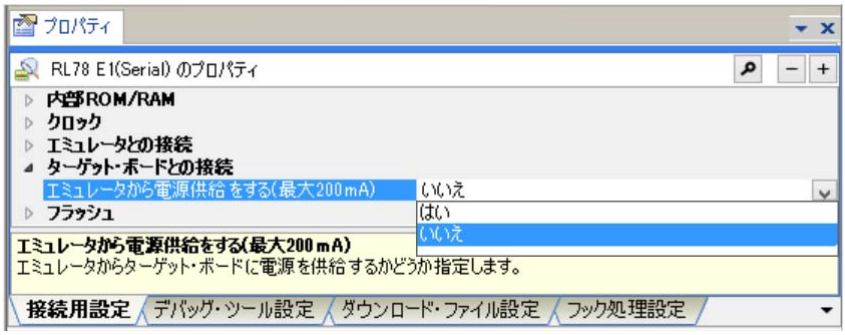


図 4.4　JTAG デバッガからの電源供給

(3) ビルドの開始

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

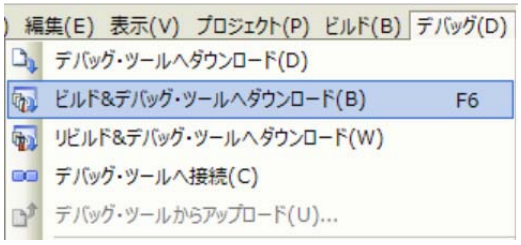


図 4.5　ビルドを開始

(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 C-First にダウンロードされます。

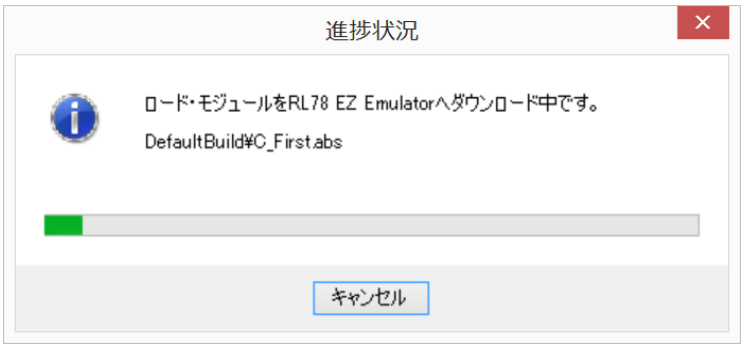


図 4.6　プログラムをロード中

(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。

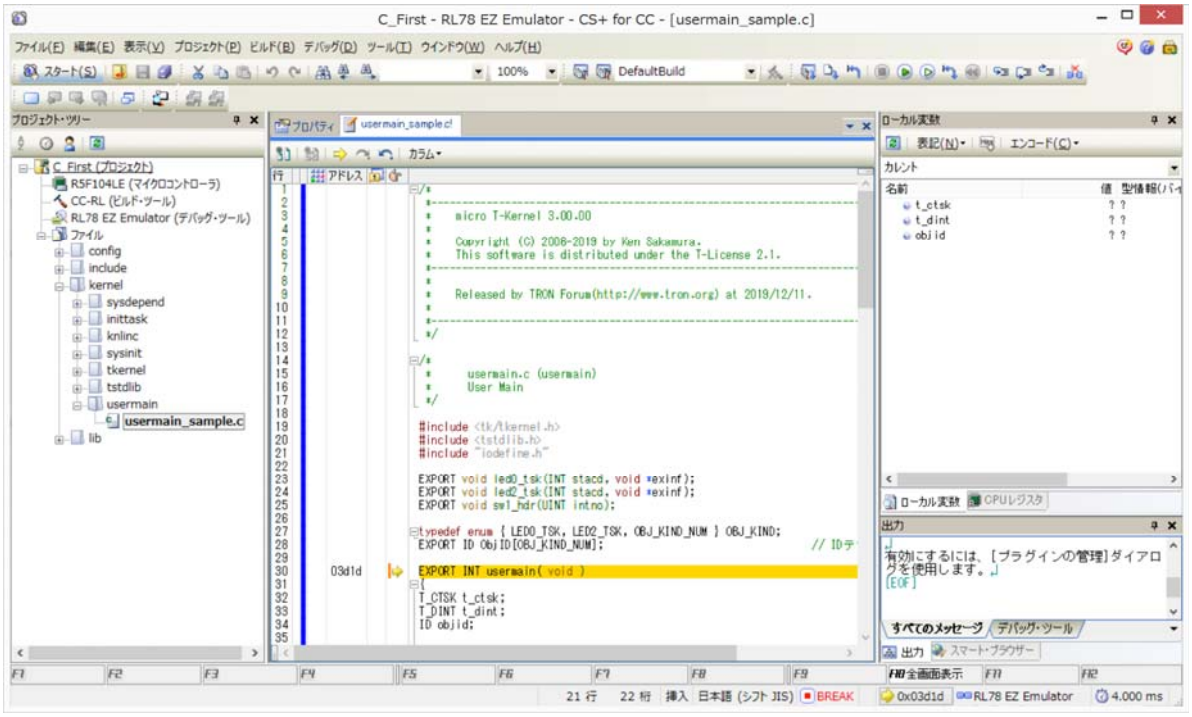


図 4.7 ロード直後の表示

(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μT-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、C-First 搭載の LED0 が 500ms の間隔で点灯・消灯を繰り返し、並行して SW1 の押し下げ毎に LED2 の点灯・消灯が切り替わるようになっています。

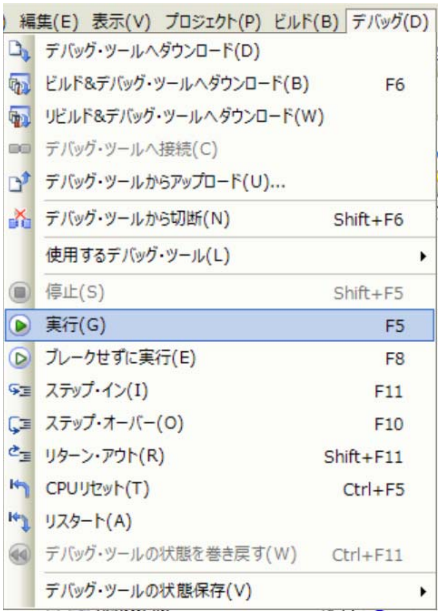


図 4.8　プログラムの実行

(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。

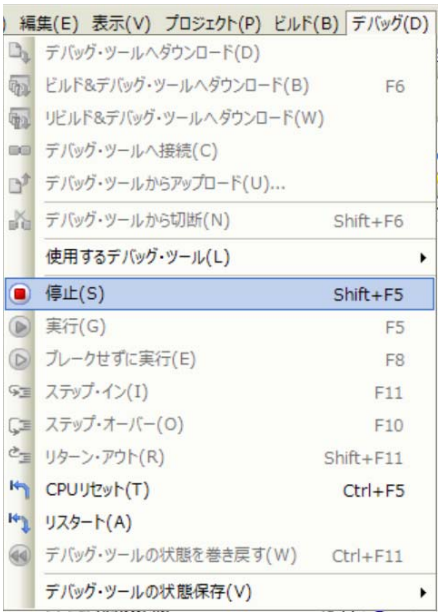


図 4.9　プログラムの停止

(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 RL78 EZ Emulator や 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。

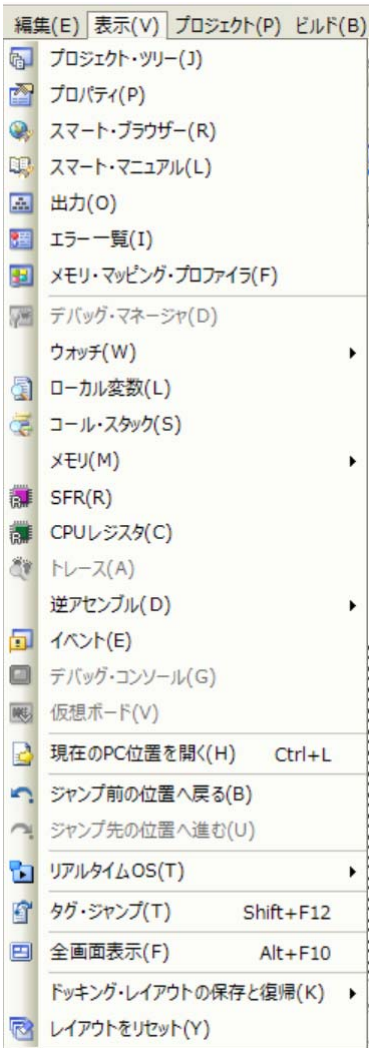


図 4.10　表示・メニュー

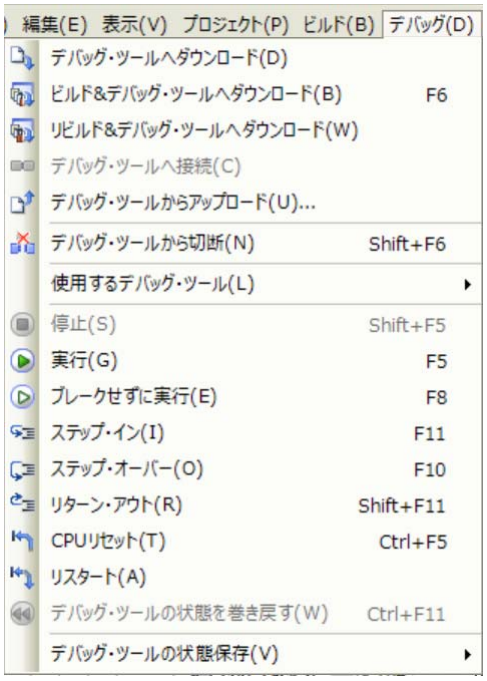


図 4.11　デバッグ・メニュー

4.3　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-RL（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-RL のプロパティから変更することができます。

図 4.12　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、C 言語のカテゴリ、C 言語の規格のオプションは C99 に固定となります。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。

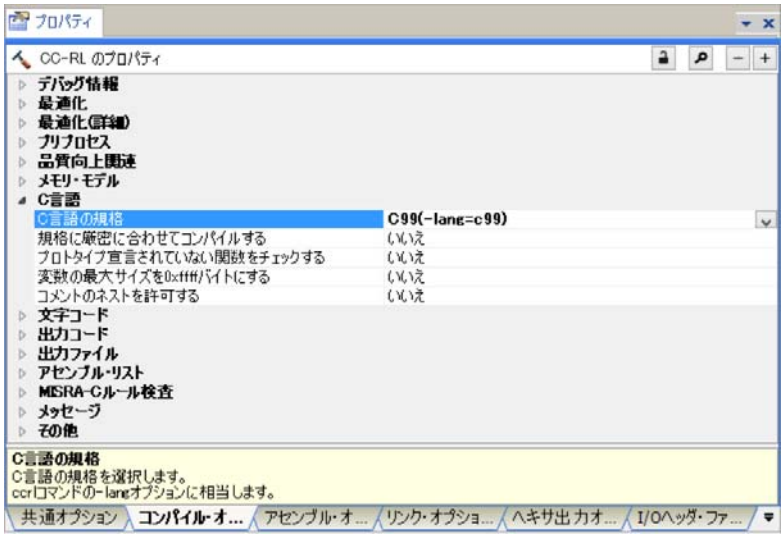


図 4.13　C 言語の規格

(2) メモリ・モデル

本ソフトでは、コンパイル・オプションのタブ、メモリ・モデルのカテゴリ、メモリ・モデルのオプションはスモール・モデルに固定となります。このオプションを変更した場合、μT-Kernel3.0 は正常に動作しません。これは OS 内部で各種ポインタのサイズを２バイトで取り扱っているためです。タスク切り替えの際も、CPU 内部レジスタの CS と ES はコンテキストの対象外としています。

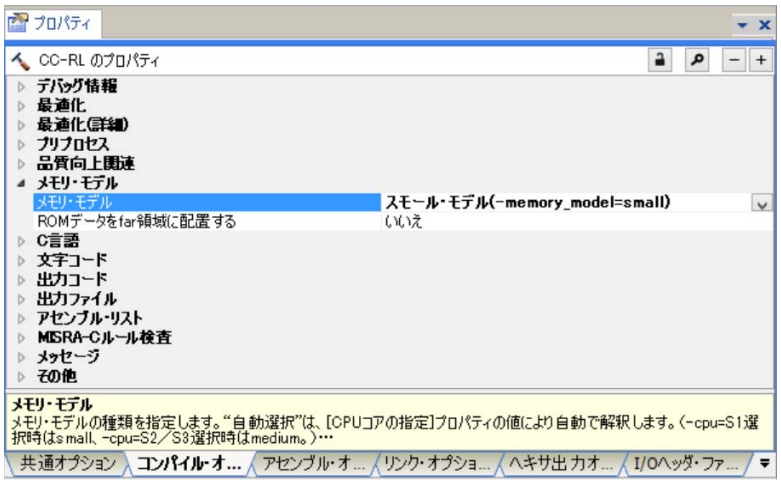


図 4.14　メモリ・モデル

(3) 定義マクロ

本ソフトでは、システム構成を定義するための３つの定義マクロ（C 言語は３個、アセンブリ言語は２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、プリプロセスのカテゴリ、定義マクロで指定します。アセンブリ言語の定義マクロは、アセンブル・オプションのタブ、プリプロセスのカテゴリ、定義マクロで指定します。

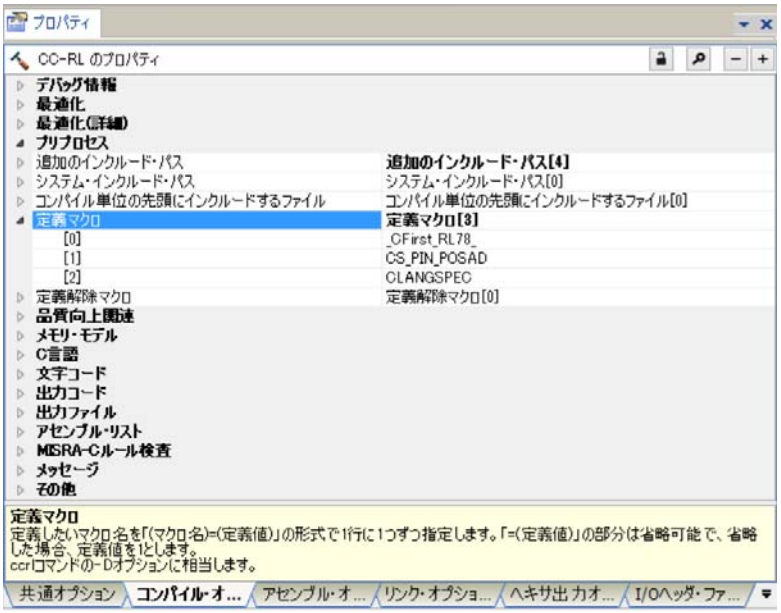


図 4.15　C 言語の定義マクロ

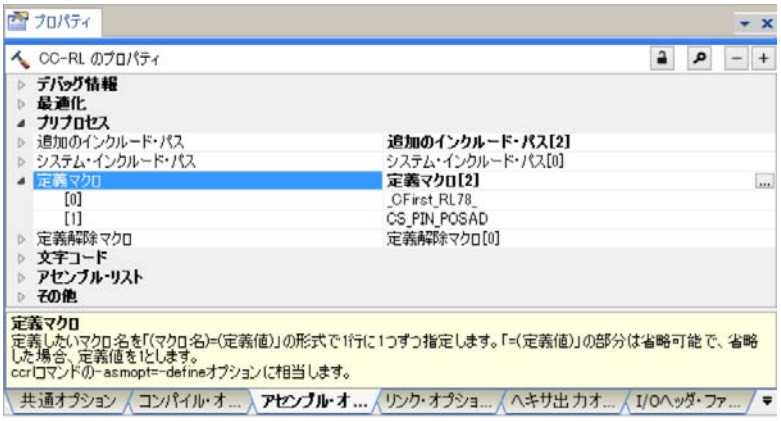


図 4.16　アセンブリ言語の定義マクロ

・「_CFirst_RL78_」定義マクロ

「_CFirst_RL78_」は、ビルドする本ソフトが C-First 用のμT-Kernel3.0 であることを意味する定義マクロです。この定義マクロを削除した場合、正しくビルドできません。必ず「_CFirst_RL78_」を定義した状態でビルドを行ってください。なお、「_CFirst_RL78_」定義マクロは、C 言語、アセンブリ言語の両方に必要となります。

・「CS_PIN_POSAD」定義マクロ

「CS_PIN_POSAD」は、第 5 章で説明するパートナーOS 対応デバッグプラグインを使用する際に必要な定義マクロです。詳細は第 5 章で説明します。なお、「CS_PIN_POSAD」定義マクロは、C 言語、アセンブリ言語の両方に必要となります。

・「CLANGSPEC」定義マクロ

「CLANGSPEC」は、第 7 章で説明するμT-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定義マクロです。デフォルトは未指定です。詳細は第 7 章で説明します。

(4) 最適化方法

本ソフトでは、最適化リンカが持つ最適化方法のオプションを有効にしています。最適化リンカの最適化方法は、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μT-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μT-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設定できない場合、本オプションの設定値は変更しても構いません。

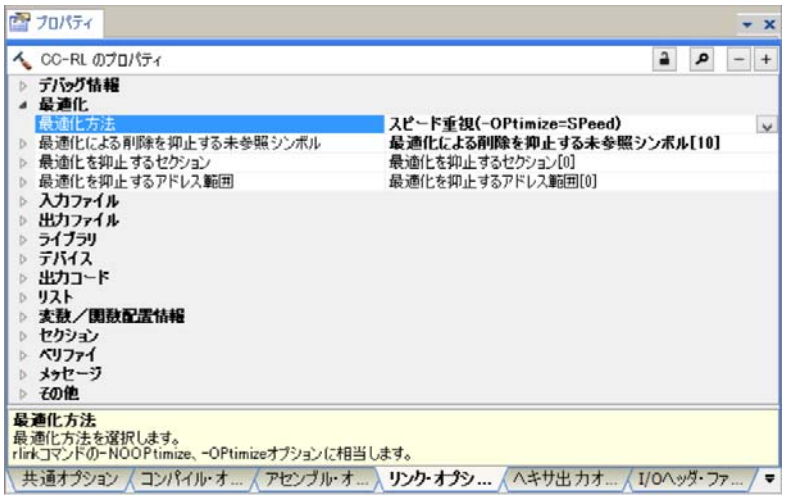


図 4.17　最適化リンカの最適化方法

(5) 警告抑止

本ソフトでは、コンパイラが出力する警告メッセージの内、以下の警告を抑止しています。もし、ユーザシステムにおいて、以下の警告を抑止すると不都合が生じるのであれば、警告抑止のオプションを削除しも構いません。ただし、削除するとビルド時に目的の警告が数多く表示されますが、μT-Kernel3.0 の動作に問題はありません。

W0523082：偶数アライメントのオブジェクトを指すポインタが奇数番地を保持しています。

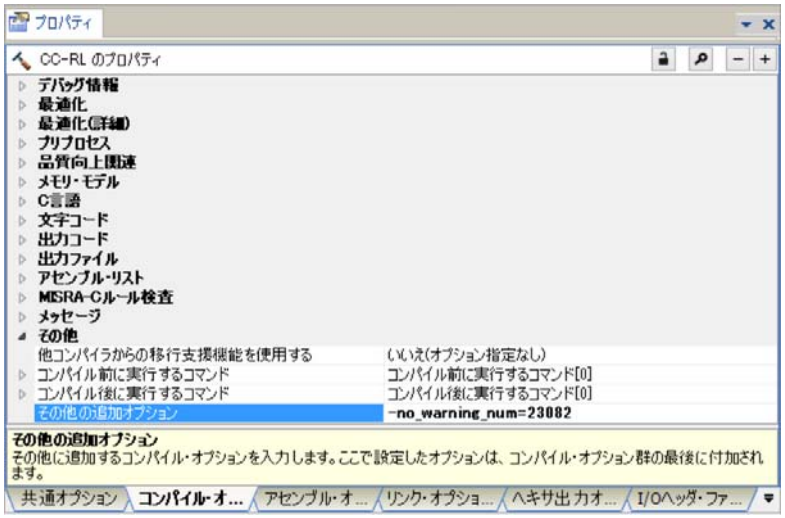


図 4.18　その他の追加オプション

また、CC-RL の V1.12 からは以下の警告も抑止しています。

W0511187："バージョン"のオプション"文字列"の評価期間の有効期限が切れています。暗黙に"-Olite"指定に変更します。引き続き"文字列"を利用したい場合は製品の購入を検討ください。明示的に"-Olite"か"-Onothing"を指定することで，この警告は消えます。

4.4　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、μT-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 4.1　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 4.2　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。

4.5 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_C-First 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/r5f104le/sysdef.h
mtkernel_3/include/sys/sysdepend/cpu/r5f104le/sysdef.inc

(4) プロファイル

【TK_SUPPORT_LOWPOWER】
mtkernel_3/include/sys/sysdepend/cfirst_rl78/profile.h
mtkernel_3/include/sys/sysdepend/cfirst_rl78/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/rl78s3/profile.h

【TK_SUPPORT_UTC、TK_SUPPORT_TRONTIME】
mtkernel_3/include/sys/profile.h

【TK_SUPPORT_DSNAME、TK_SUPPORT_DBGSPT、TK_MAX_TSKPRI】

TK_SUPPORT_DSNAME	⇒	config.h(inc)の USE_OBJECT_NAME で指定
TK_SUPPORT_DBGSPT	⇒	config.h(inc)の USE_DBGSPT で指定
TK_MAX_TSKPRI	⇒	config.h(inc)の CFN_MAX_TSKPRI で指定

5.　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μT-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。

5.1　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 5.1　パートナーOS 対応デバッグプラグインのインストーラ

5.2　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。

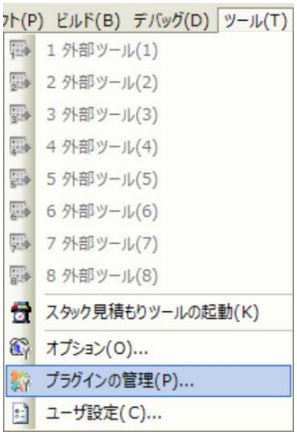


図 5.2　プラグインの管理

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、チェックしてプログライン機能を有効にしてください。

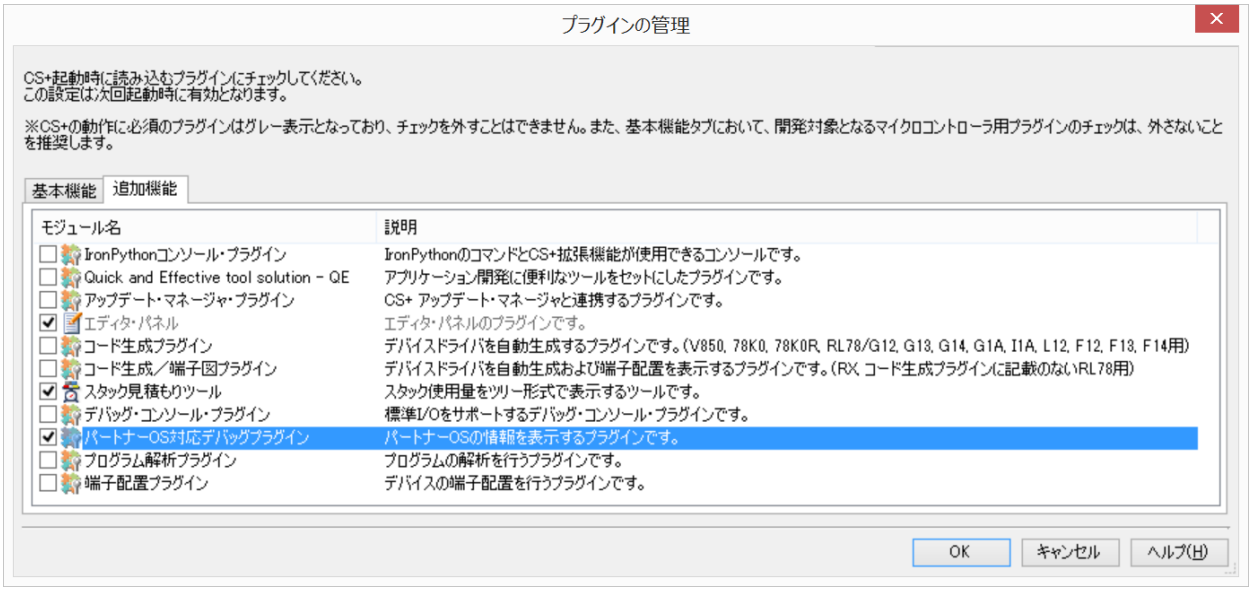


図 5.3　「プラグインの管理」ダイアログ

5.3　OS 定義ファイル

μT-Kernel のパートナーOS 対応デバッグプラグインは、本来 RX ファミリ用に開発されており、RL78 ファミリには対応していません。本機能をμT-Kernel3.0 に対応させるためには、RL78 ファミリ用の 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。

RL78 用の OS 定義ファイルは、μT-Kernel3.0 の展開フォルダの mtkernel_3/others フォルダにファイル名「UT-RL.mtod」として格納されています。

図 5.4　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。

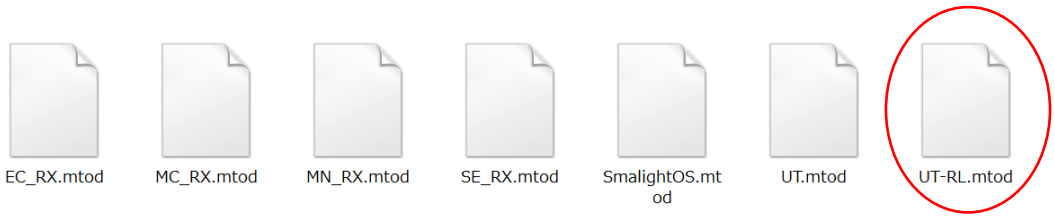


図 5.5　OS 定義ファイルの格納先フォルダ

5.4　定義マクロ

パートナーOS 対応デバッグプラグインを利用するためには「CS_PIN_POSAD」の定義マクロが C 言語及びアセンブリ言語に必要となります（図 4.15、図 4.16 参照）。

なお、パートナーOS 対応デバッグプラグインを利用するとシステムの ROM 領域と RAM 領域を共に約 30 バイト程度（カーネルオブジェクトの個数に依存して）消費します。パートナーOS 対応デバッグプラグインが不要であれば、「CS_PIN_POSAD」の定義マクロは削除することを推奨します。

5.5　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。

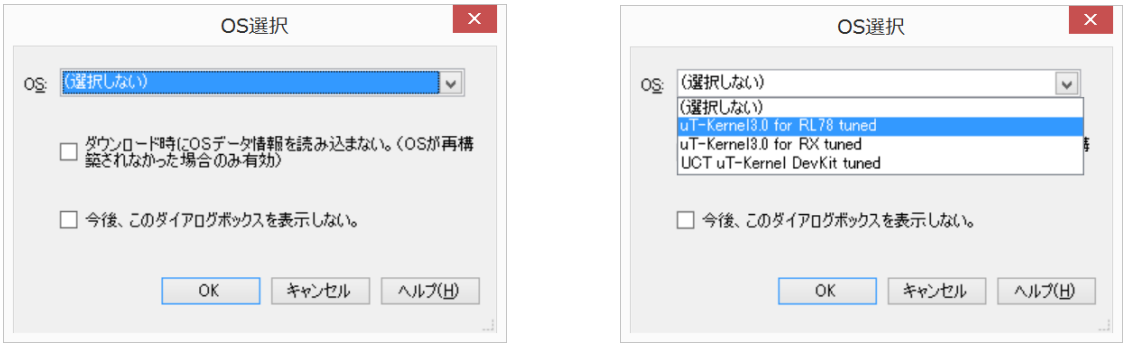


図 5.6　「OS 選択」のダイアログ

ドロップダウンで「uT-Kernel3.0 for RL78 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 5.7　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。

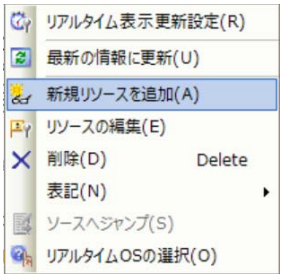
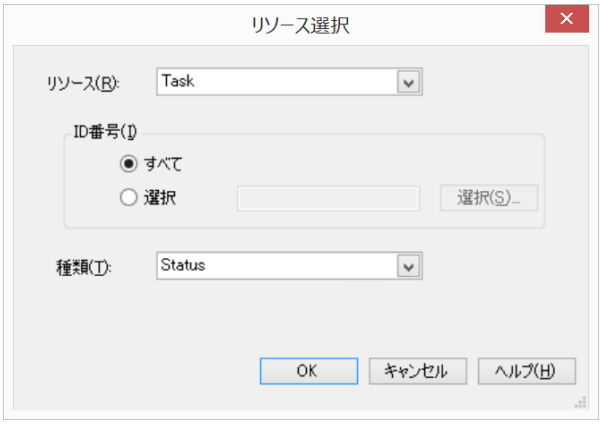
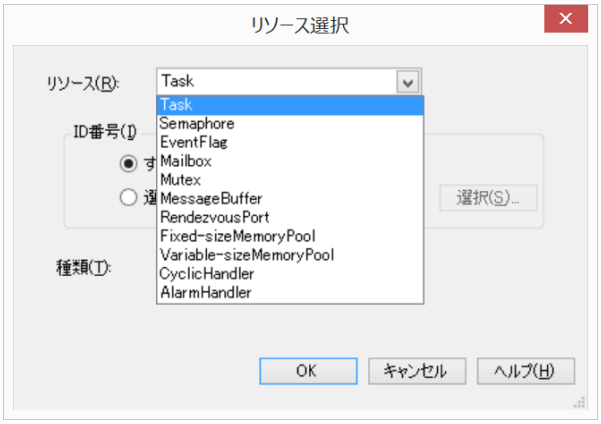
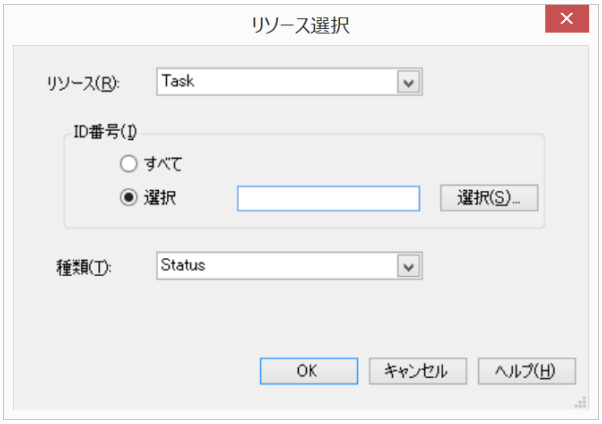


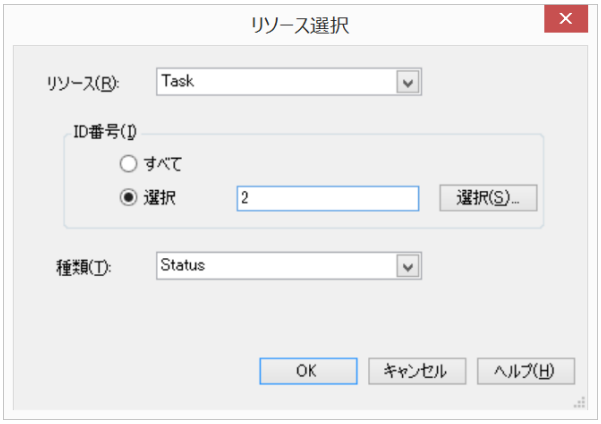
図 5.8　リソース情報の追加・削除

「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。









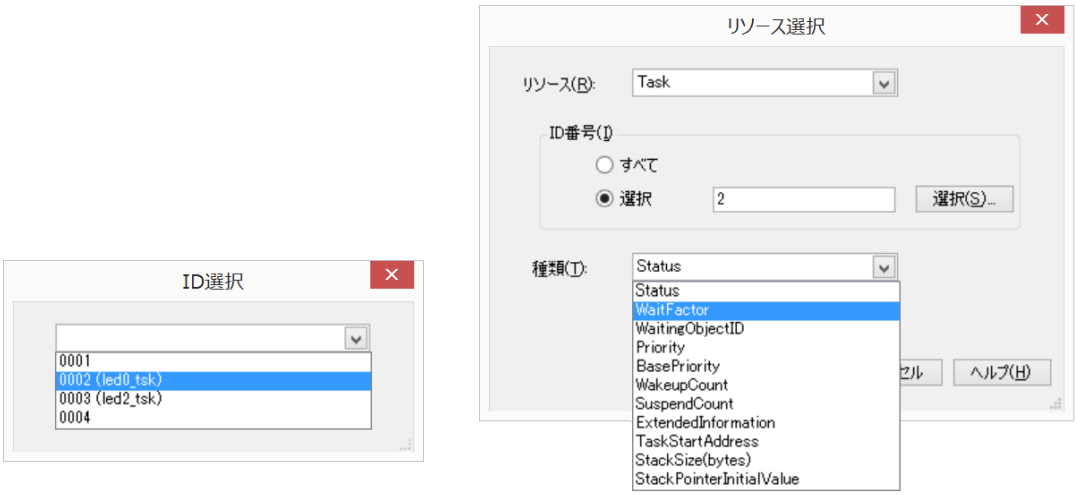


図 5.9　「リソース選択」、「ID 選択」のダイアログ

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。

図 5.10　リソースを修正したリソース情報パネル

6.　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_C-First 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。

6.1　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 6.1　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。

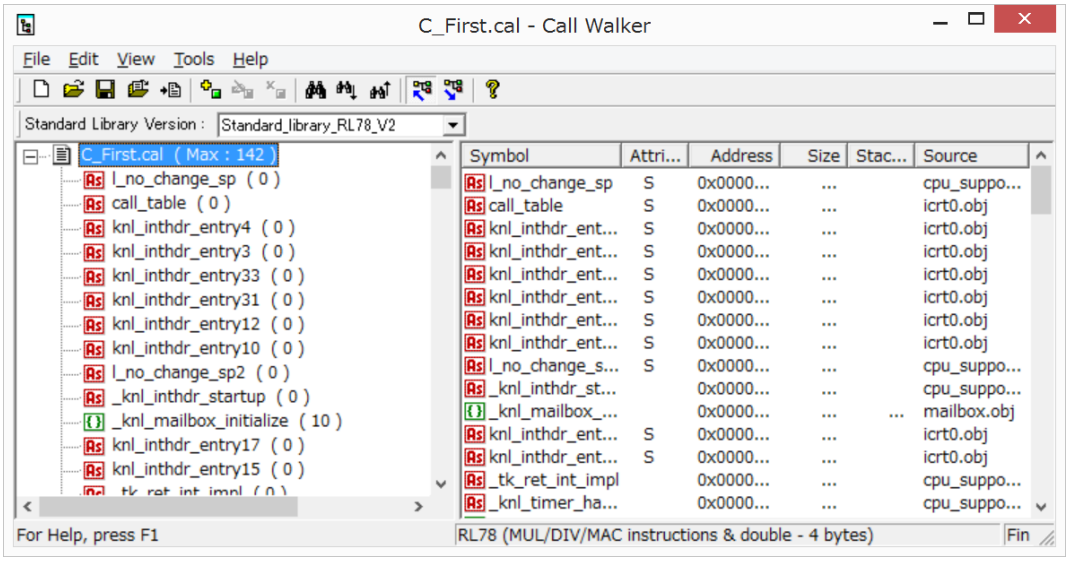


図 6.2　スタック見積もりツールの表示

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。

・エラーなしでビルドが完了しているか

スタック見積もりツールの起動には、プロジェクトに登録されているソースファイルのコンパイル結果として、アセンブリ言語のソースファイルが必要となります。アセンブリ言語のソースファイルはビルド時に生成するオプションを設定してありますから、ビルドがエラーなしで完了し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。

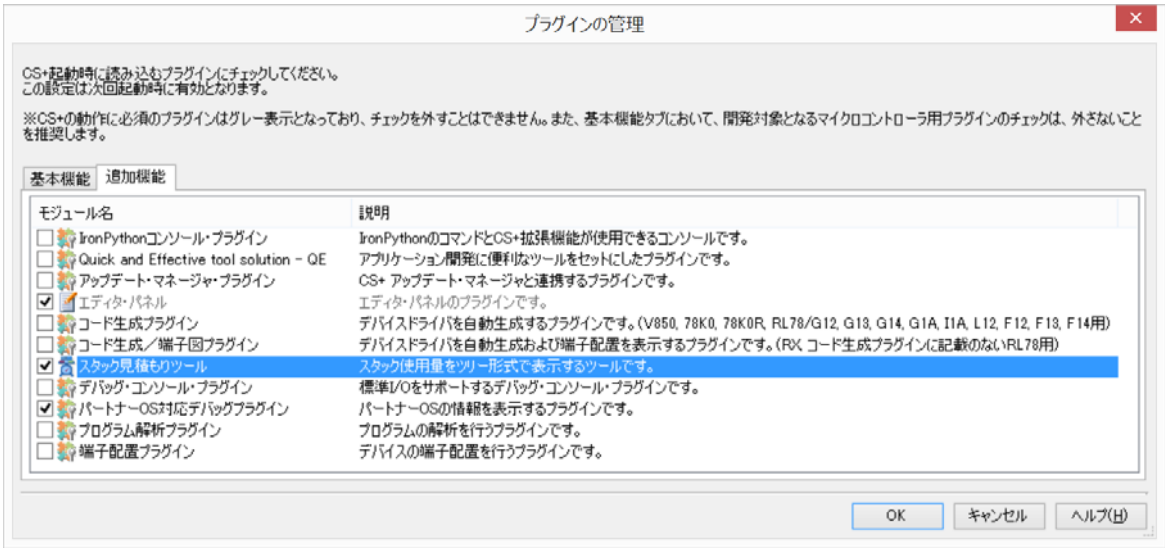


図 6.3　「プラグインの管理」ダイアログ

6.2　リアルタイム OS オプションの設定

デフォルトの状態で表示されたスタック解析結果から、μT-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、不必要なシンボル（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μT-Kernel3.0 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。

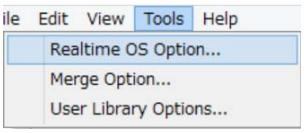


図 6.4　リアルタイム OS オプション

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。

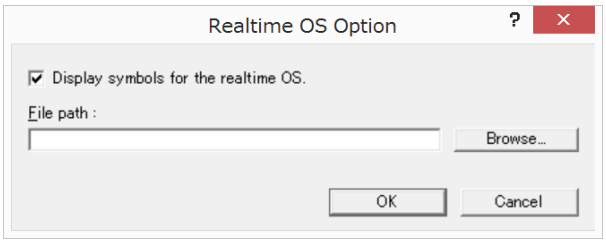


図 6.5　「Realtime OS Option」ダイアログ

RTOS ライブラリデータファイルは、μT-Kernel3.0 の展開フォルダ直下の mtkernel_3/others フォルダに CC-RL コンパイラのバージョン対応毎のフォルダにファイル名「utkernel.csv」として格納されています。V1.10.00 以降も同様です。

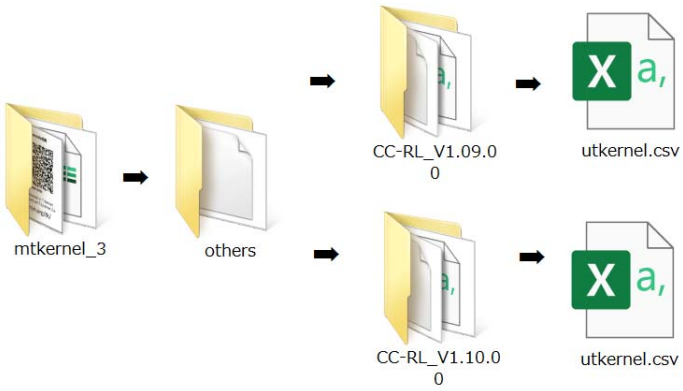


図 6.6　RTOS ライブラリデータファイル

使用している CC-RL 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-RL の V1.09.00 から V1.12.00 に対応しています。

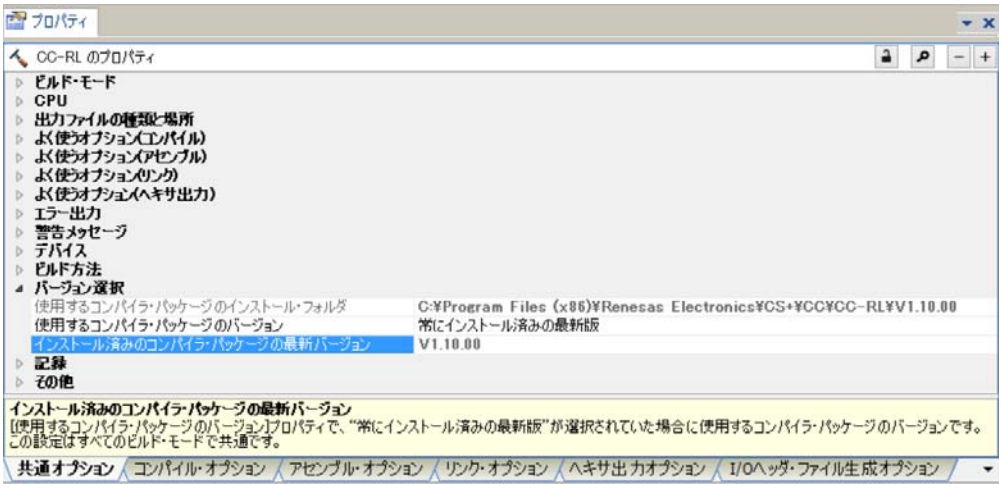


図 6.7　CC-RL コンパイラのバージョン情報

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

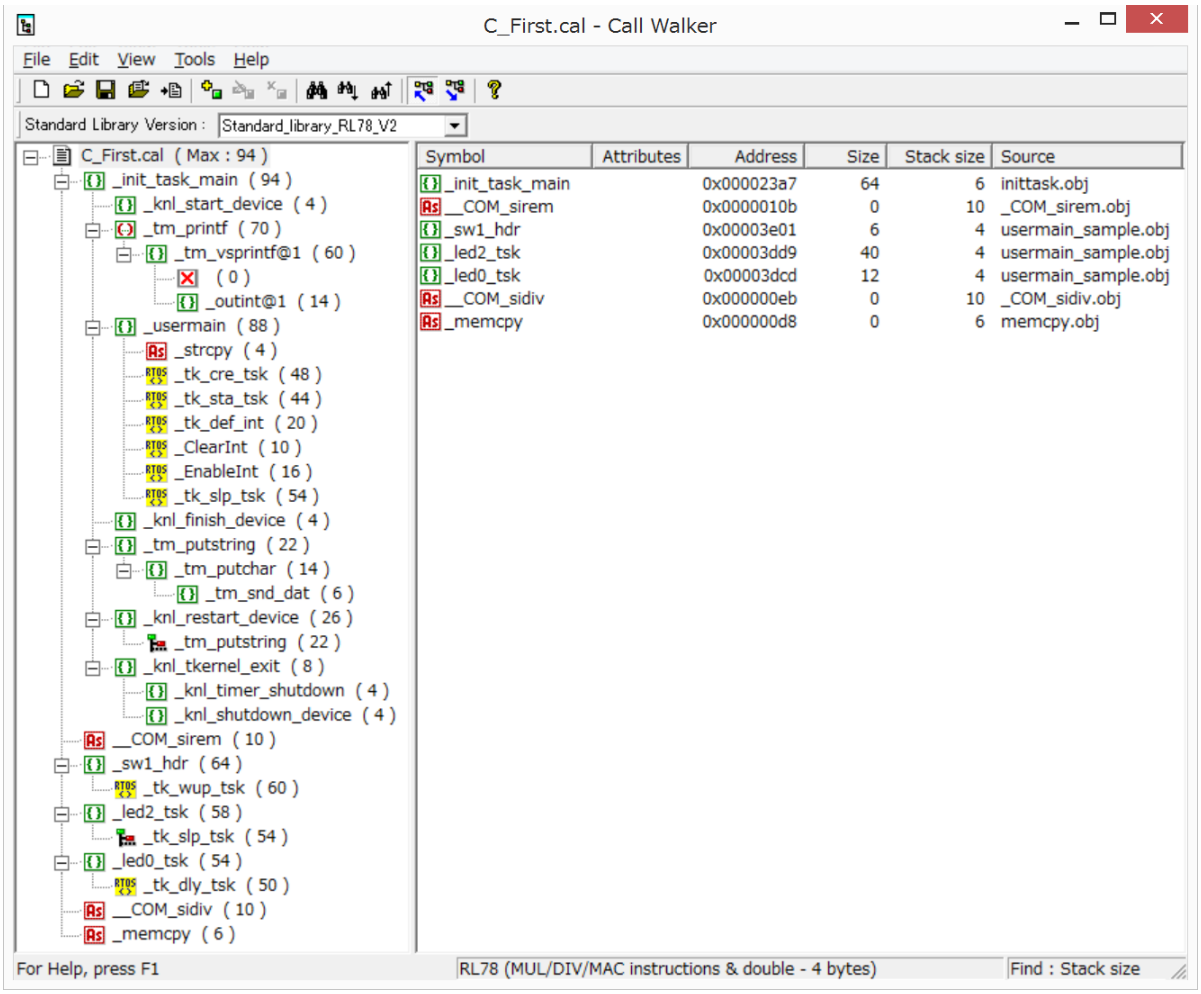


図 6.8　オプション指定後のスタック解析結果

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 94 バイト
・led0_tsk が独自に使用するスタックサイズは 54 バイト
・led2_tsk が独自に使用するスタックサイズは 58 バイト
・sw1_hdr 割込みハンドラが独自に使用するスタックサイズは 64 バイト
となります。

6.3　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。

6.3.1　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_CSTK.stksz)で指定します。

(2) テンポラリスタック

現在のコンテキストを破棄して、別のタスクコンテキストに移行するまでの間に使用するスタックであり、mtkernel_3/config/config.h(inc) の CFN_TMP_STACK_SIZE で指定します。

(3) 例外スタック

割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。

6.3.2　OS 管理外割込み使用時のスタックサイズ

OS 管理外割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 管理外割込みは割込み優先度 0 だけですが、コンフィギュレーションによっては割込み優先度 1 や 2 も OS 管理外割込みとすることが可能です。

結果、OS 管理外割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。

　割込み優先度 0 の中で最も多くスタックを消費する OS 管理外割込み関数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 管理外割込み関数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 管理外割込み関数が独自に使用するサイズ

割込み優先度が同一の OS 管理外割込み関数は、ネストして実行されることがありません（割込み優先度 0 だけは多重割込みを許可することが許されていません)。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 管理外割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。

OS 管理外割込みはレジスタバンクが使用できるため、OS 管理外割込み関数が独自に使用するスタックサイズはそれほど大きくはならないはずですが、可能あれば 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。

6.3.3 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 ＋ 12 バイト + OS 管理外割込みのスタックサイズ

計算式の 12 バイトはタスクコンテキストの退避に使用するスタックサイズです。

図 6.8 の例であれば、led0_tsk と led2_tsk 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と仮定しています）。

・led0_tsk：54 バイト ＋ 12 バイト ＋ 0 バイト ＝ 66 バイト
・led2_tsk：56 バイト ＋ 12 バイト ＋ 0 バイト ＝ 68 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 6.8 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：94 バイト ＋ 12 バイト ＋ 0 バイト ＝ 106 バイト

6.3.4 テンポラリスタックのサイズ

テンポラリスタックのサイズは、プロファイル TK_SUPPORT_LOWPOWER の設定値によって変化します。

【プロファイル TK_SUPPORT_LOWPOWER が FALSE の場合】

12 バイト + OS 管理外割込みのスタックサイズ

【プロファイル TK_SUPPORT_LOWPOWER が TRUE の場合】

low_power 関数が独自に使用するサイズ ＋ 12 バイト + OS 管理外割込みのスタックサイズ

プロファイル TK_SUPPORT_LOWPOWER が TRUE の場合、システムアイドリング時（実行するタスクが存在しない時）に low_power 関数が呼び出されます。従って、テンポラリスタックに low_power 関数が独自に使用するスタックサイズを加算する必要があります。

6.3.5　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 ～ 3 ですが、コンフィギュレーションによって割込み優先度 2 ～ 3 や割込み優先度 3 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 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
＋ 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 3 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
＋ 12 バイト＊(割込みのネスト数-1)　＋ 2 バイト
＋ OS 管理外割込みのスタックサイズ

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 12 バイトのスタックサイズを消費します。従って、12 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。ただし、タスク実行中はタスクスタックを使用するため、割込みネスト数は 1 を減算することになります。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。加算している最後の 2 バイトは SP の退避に必要なサイズです。

なお、デフォルトの設定では、割込み優先度 1 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 1 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 114 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の中に独自に使用するサイズが 94 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 20 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 94 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
　114 バイト
・スタックサイズの消費が 94 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 ＋ 20 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 6.8 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 3 に sw1_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のようになります（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 1	: 114 バイト（システムタイマ割込み）
＋	割込み優先度 2	: 0 バイト
＋	割込み優先度 3	: 64 バイト
＋	割込みのネスト数	: 14 バイト（12*(2-1)+2⇒割込みネスト数は 2）
＋	OS 管理外割込み	: 0 バイト
	合計	: 192 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。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 34 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
　54 バイト

・スタックサイズの消費が 34 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 ＋ 20 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。

7. C 言語規格準拠の仕様

7.1 C 言語規格に準拠していない部分

μT-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

μT-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

μT-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言う意味にはならず、言語仕様では引数の個数や型のチェックを行わないと言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。

7.2 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、μT-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μT-Kernel3.0 の API と同じです。

(1) T-Monitor のライブラリ

【μT-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, ... );
```

(2) 高速ロック・マルチロックライブラリ

【μT-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 )
```

(3) カーネルオブジェクト生成時のパラメータパケット

【μT-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_CSEM、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 */
```

【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 */
```

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 */
```

(4) μT-Kernel/OS システムコール

【μT-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 blksz, 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 blksz, void *p_blk, TMO tmout );
IMPORT ER tk_get_mpf( ID mpfid, void *p_blf, TMO tmout );
```

(5) μT-Kernel/SM システムコール

【μT-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_idev );
```

【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_idev );
```

8.　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μT-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μT-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>
```

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

9.　問い合わせ先

本実装に関する問い合わせ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