RZ/A1H グループ
OS 移植層(OSPL) サンプルプログラム

R01AN1887JJ0160
Rev.1.60
2017.10.26

要旨

本アプリケーションノートでは、OS レスと OS ありを抽象化する OSPL のサンプルプログラムについて説明します。

OS 移植層(OSPL)サンプルプログラムの特長を以下に示します。

・OS レスと OS ありで共通の API を提供します（4.6）

・新しい OS に移植するときに変更するコードを局所化します（4.8.4）

・発生する非同期イベントのビットを自動的に割り当て、パイプラインを構成できます（4.5.6）

・キャッシュフラッシュの API を提供します（4.6.1(10)）

・タイマーハードウェアを使用し、時間に関する基本的な API を提供します（4.6.1(14)）

・CPU 使用率を計測する API を提供します（OS レス向けのみ）（4.6.1(15)）

・デバッグ機能を持ったエラー処理 API を提供します（4.6.1(17)）

・R_OSPL_NDEBUG を#define 定義するとデバッグ機能や一部の表明を無効にして高速かつコンパクトにします

対象デバイス

RZ/A1H グループ

RZ/A1M グループ

RZ/A1LU グループ

RZ/A1L グループ

本アプリケーションノートを他のマイコンへ適用する場合、そのマイコンの仕様にあわせて変更し、十分評価してください。

RSK ボードのプロジェクトについては、お問い合わせください。

本パッケージには、テスト用 OSPL も含んでいます。詳細は、R_OSPL_TEST_CODE の説明を参照してください（4.4.17）。

目次

1.　仕様

表 1.1 にサンプルが使用する周辺機能と用途を、図 1.1 にサンプルコード・OSPL サンプルドライバのテスト実行時の動作環境を示します。

表 1.1　使用する周辺機能と用途

周辺機能	用途
OS タイマー（OSTM0 または OSTM1）または マルチファンクション タイマー パルス ユニット 2（MUT2 - ch1, 2）	フリーランタイマー機能(4.6.15(2))の内部で使用。 R_OSPL_FTIMER_IS マクロで使用するチャンネルを選択できます。 サンプルドライバでも使用
割り込みコントローラ INTC（割込み ID：DMAINT0～3）	DMAC 割込み（サンプルドライバで使用）
ダイレクト メモリー アクセス コントローラ DMAC Ch(0～3)	DMA 転送（サンプルドライバで使用）

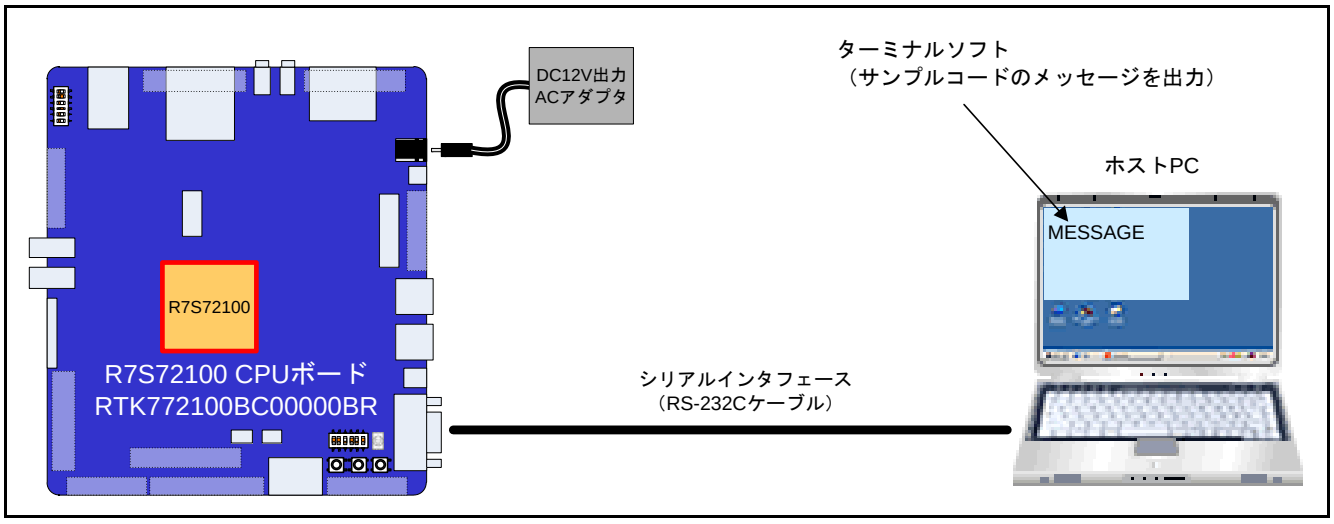


図 1.1　動作環境

2. 動作確認条件

本アプリケーションノートのサンプルコード、およびそのテストは、下記の条件で動作を確認しています。 ターゲット OS を変更する方法については、(4.8.1) ターゲットの選択 (use_list.h, mcu_board_select.h) を参照してください。

表 2.1　動作確認条件

項目		内容
使用マイコン		RZ/A1H
動作周波数		CPU クロック（Iφ）：400MHz 画像処理クロック（Gφ）：266.67MHz 内部バスクロック（Bφ）：133.33MHz 周辺クロック 1（P1φ）：66.67MHz 周辺クロック 0（P0φ）：33.33MHz
動作電圧		電源電圧（I/O）：3.3V 電源電圧（内部）：1.18V
ARM	統合開発環境	ARM®統合開発環境 ARM Development Studio 5（DS-5™）Version 5.16
	C コンパイラー	ARM C/C++ Compiler/Linker/Assembler Ver.5.03 [Build 102]
IAR	統合開発環境	IAR Embedded Workbench for ARM 7.80.4.12495
	C コンパイラー	
Renesas gcc	統合開発環境	e2 studio (Version: 5.3.0.023)
	C コンパイラー	GNUARM-NONE-EABI v16.01
動作モード		ブートモード 0 （CS0 空間 16 ビットブート）
ターミナルソフトの通信設定		・通信速度：115200bps ・データ長：8 ビット ・パリティ：なし ・ストップビット長：1 ビット ・フロー制御：なし
使用ボード		GENMAI ボード ・R7S72100 CPU ボード　RTK772100BC00000BR ・R7S72100 CPU ボード用オプションボード　RTK7721000B00000BR
OS		・OS レス ・CMSIS-RTOS RTX 4.80
使用デバイス （ボード上で使用する機能）		・シリアルインタフェース（Dsub-9 コネクタ）

3. 関連アプリケーションノート

本アプリケーションノートに関連するアプリケーションノートを以下に示します。併せて参照してください。

- RZ/A1H グループ 初期設定例（R01AN1646JJ）
- RZ/A1H グループ レジスター定義ヘッダーファイル iodefine.h（R01AN1860JJ）
- RZ/A1H グループ CMSIS-RTOS RTX BSP V2.06 (e2studio / KPIT GCC)（R01AN3104JJ）
- RZ/A1H グループ CMSIS-RTOS RTX BSP V2.03 ICCARM リリースノート（R01AN2990JJ）
- RZ/A1H グループ CMSIS-RTOS RTX BSP V2.03 リリースノート（R01AN2200JJ）
- RX ファミリ ボードサポートパッケージモジュール Firmware Integration Technology（R01AN1685JJ）

ドライバーの開発に関するガイドは、本書では扱っていません。

4.　ソフトウェア説明

4.1　　動作概要

図 4.1 にシステムブロック図を示します。

参考：(4.7) シーケンス図

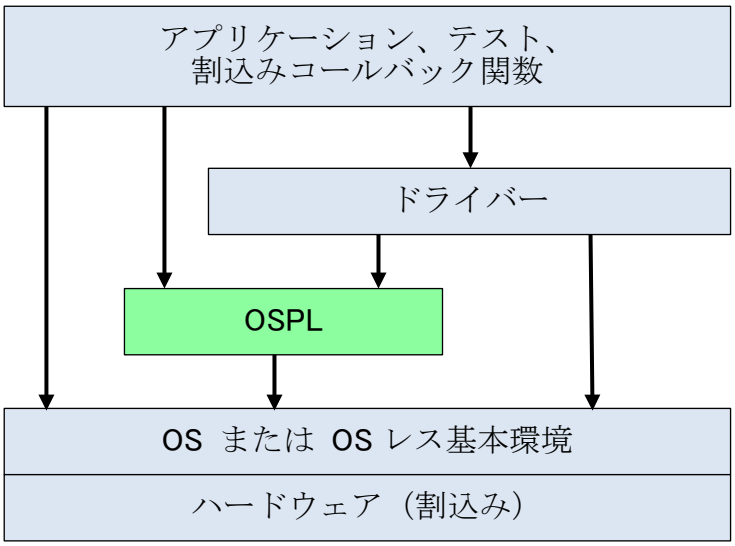


図 4.1　システムブロック図

OSPL は、OS レス環境（擬似マルチスレッド環境）と、OS（RTOS）あり環境に共通する API を提供します。OSPL を使用することで、OS レス環境と OS ありの環境の間で移植する事が簡単になります。

割込みに応答するドライバー（非同期処理の開始や待ちがあるドライバー）が対象です。

待ちが入る同期 API 関数と、それに対応する、非同期 API 関数の両方を提供し、使いやすさと擬似マルチスレッド対応の両立ができるようにすることを支援します。OSPL の推奨ドライバーAPI に対応していないドライバー（割込みハンドラー付きドライバー）を使用する場合でも、アプリケーションが直接 OSPL の API を呼び出すことで、移植性を高めます。

OSPL は、割込み応答処理、メモリー、時間、アイドル状態、エラー処理を扱います。スレッド間排他制御や、スレッド間通信は対象外です。 OS ありの環境で、それらの処理が必要になるときは、OS レス向けドライバーの上にかぶせる「ドライバー・ラッパー」から直接それぞれの OS 関数を呼び出します。

OSPL がサポートする機能を、表 4.1 に示します。

表 4.1 OSPL の機能

分類	機能	OS レス	OS あり	詳細
スレッド	スレッド管理	○	×	スレッド生成（OS レスは擬似スレッド） スレッドパラメータ設定、取得 スレッド ID 設定 ウェイト時の動作管理
		○	○	スレッド ID 取得
	ロック	○	○	ロック、ロックの解除
イベント	フラグ管理	○	○	スレッド付属フラグのセット、クリア、ウェイト 通常フラグのセット、クリア、チェック
	キュー	○	○	スレッド間でデータの転送
割込み	禁止・許可	○	○	全割込みの許可・禁止 コールバック関数の登録 割込みハンドラー登録 個別の割込みの許可・禁止・優先度設定
オブジェクト	ロック	○	○	ロック、ロック解除
メモリー	キャッシュ	○	○	キャッシュ操作
	アドレス変換	○	○	キャッシュ有効・無効領域のアドレス変換 仮想・物理アドレス変換
時間	ウェイト	○	○	指定時間ウェイト
	タイマー参照	○	○	フリーランタイマーの参照
デバッグ支援	CPU	○	×	CPU 使用率の測定・出力
	デバッグ	○	○	回復不能なエラーの管理 エラーブレーク 表明（契約プログラミング） エラー情報の保持・出力 ウォッチ、高速ログ、実行回数デバッグ スタック オーバーフロー チェック
コーディング支援	可読性向上	○	○	静的コード解析ツールの対応による可読性低下の回避

○：OSPL でサポートする機能

×：OSPL でサポートしない機能

4.1.1　使用準備

OSPL サンプルプログラムの実行準備を説明します。

1)　ホスト PC にてターミナルソフトを起動し、次のように設定します。(Tera Term の場合)

図 4.2　シリアルポートの設定

2)　サンプルプログラムを実行すると、次のようにターミナルに表示されます。

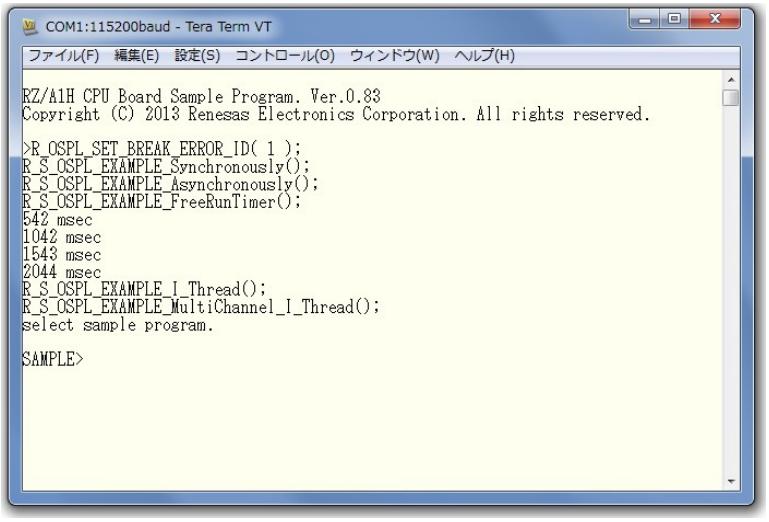


図 4.3　OSPL サンプルプログラムの実行結果

4.2　使用割込み一覧

表 4.2 に OSPL サンプルドライバのサンプルコードで使用する割込みを示します。 割込みで行われる処理の内容は、それぞれのドライバーの R_DRIVER_OnInterrupting 関数、および R_DRIVER_OnInterrupted 関数（関数名はドライバによって異なる）の処理の内容に一致します。

表 4.2　OSPL サンプルドライバのサンプルコードで使用する割込み

割込み(要因 ID)	優先度	処理概要
DMAINT0～3	7	DMAC から発生する割込みに応答する処理。
OSTM#TINT(#=0～1)	2	タイマー割込みに応答する処理。

4.3　基本型

表 4.3 にサンプルコードで使用する基本型を示します。

表 4.3　サンプルコードで使用する基本型

シンボル	内容
char_t	8 ビット文字
bool_t	論理型。値は true（1）, false（0）
bool8_t	8 ビット論理型。値は true（1）, false（0）
bool16_t	16 ビット論理型。値は true（1）, false（0）
bool32_t	32 ビット論理型。値は true（1）, false（0）
int_t	高速な整数、符号あり、本サンプルコードでは 32 ビット整数。
int8_t	8 ビット整数、符号あり（標準ライブラリーにて定義）
int16_t	16 ビット整数、符号あり（標準ライブラリーにて定義）
int32_t	32 ビット整数、符号あり（標準ライブラリーにて定義）
int64_t	64 ビット整数、符号あり（標準ライブラリーにて定義）
uint8_t	8 ビット整数、符号なし（標準ライブラリーにて定義）
uint16_t	16 ビット整数、符号なし（標準ライブラリーにて定義）
uint32_t	32 ビット整数、符号なし（標準ライブラリーにて定義）
uint64_t	64 ビット整数、符号なし（標準ライブラリーにて定義）
int_fast8_t	少なくとも 8 ビットある、高速な整数、符号あり
int_fast16_t	少なくとも 16 ビットある、高速な整数、符号あり
int_fast32_t	少なくとも 32 ビットある、高速な整数、符号あり
uint_fast8_t	少なくとも 8 ビットある、高速な整数、符号なし
uint_fast16_t	少なくとも 16 ビットある、高速な整数、符号なし
uint_fast32_t	少なくとも 32 ビットある、高速な整数、符号なし
uintptr_t	ポインターと同じビット数の符号なし整数型、または、物理アドレス
size_t	ポインターと同じビット数の符号なし整数型、または、バイトサイズ
ssize_t	ポインターと同じビット数の符号あり整数型
ptrdiff_t	ポインターと同じビット数の符号あり整数型、アドレスの差分
bit_flags_fast32_t	uint_fast32_t 型の ビットフラグ型（ビットフィールド型）
bit_flags32_t	uint32_t 型の ビットフラグ型（ビットフィールド型）
bit_flags16_t	uint16_t 型の ビットフラグ型（ビットフィールド型）
bit_flags8_t	uint8_t 型の ビットフラグ型（ビットフィールド型）
byte_t	１バイトの型
errnum_t	エラーコード、0=エラーなし。参考(4.4)
float32_t	32 ビット浮動小数 （"__ARM_NEON__"を指定時、標準ライブラリーにて定義）
float64_t	64 ビット浮動小数（標準ライブラリにて定義） （"__ARM_NEON__"を指定時、標準ライブラリーにて定義）
float128_t	128 ビット浮動小数

4.4　定数/列挙体/エラーコード

表 4.4　列挙体/定数/エラーコード一覧

章	型名	内容
4.4.1	-	バージョン
4.4.2	errnum_t	エラーコード
4.4.3	r_ospl_async_state_t	状態
4.4.4	r_ospl_wait_t	R_OSPL_THREAD_SetOnWait 関数のパラメーター
4.4.5	r_ospl_flush_t	R_OSPL_MEMORY_Flush 関数のパラメーター
4.4.6	r_ospl_axi_cache_attribute_t	AXI バスのキャッシュ属性
4.4.7	r_ospl_axi_protection_t	AXI バスの保護属性
4.4.8	r_ospl_async_type_t	非同期処理の種類
4.4.9	bsp_int_err_t	FIT*1 BSP のエラーコード
4.4.10	r_ospl_event_flags_t	イベントのビットフラグ、または、ID
4.4.11	r_ospl_table_flags_t	配列番号表のオプション
4.4.12	r_ospl_if_not_t	検索して見つからなかったときの処理方法
4.4.13	bsp_int_src_t	割込み番号
4.4.14	bsp_int_cb_t	割込みハンドラー
4.4.15	bsp_int_cmd_t	割込み関係の制御コマンド
4.4.16	mcu_lock_t	ロックするときに使うハードウェアのチャンネルの識別番号
4.4.17	-	その他の定数

4.4.1　バージョン

定数名	設定値	内容
R_OSPL_VERSION	155	OSPL のバージョン番号。 100 の位は、OSPL の仕様のバージョン番号。 10 の位、1 の位は、特定の OS、特定のボードでのマイナーバージョン番号。
R_OSPL_VERSION_STRING	"1.55	OSPL のバージョン番号の文字列。例："1.00"
R_OSPL_IS_PREEMPTION	0 または 1	プリエンプションに対応した RTOS であるかどうか。 OS レスのときは 0 です。 ラウンドロビンでなくとも割込みや API 呼び出しによってプリエンプションする場合は 1 です。 0 のときは、擬似マルチスレッドを構成する必要があります。
BSP_CFG_RTOS_USED	0 または 1	R_OSPL_IS_PREEMPTION と同じです。
BSP_CFG_PARAM_CHECKING_ENABLE	0 または 1	システム全般のパラメータチェックの有効無効 1 = R_OSPL_NDEBUG が定義されると無効になるもの以外は、すべてチェックする。（デフォルト） 0 = すべてチェックしない。
R_OSPL_FOR_FREE_RTOS	80102	対象の FreeRTOS のバージョン番号。 10000 の位が major、100 の位が minor、1 の位が build。

*1 「Board Support Package Module Using Firmware Integration Technology（R01AN1685EU0260）」を参照

R_OSPL_FOR_RTX	((4<<16)|74)	対象の RTX のバージョン番号。 osCMSIS_RTX の値と同じ
R_OSPL_FOR_RZ_A1_BSP	205	対象の RZ/A1H RTX BSP のバージョン番号の 100 倍
R_OSPL_FOR_RZ_A1_OS_LESS	101	対象の RZ/A1H OS レス サンプル のバージョン番号の 100 倍
R_OSPL_FOR_RZ_A1_INTC	101	対象の 割込み (INTC) サンプル のバージョン番号の 100 倍
R_OSPL_FOR_RZ_A1_CACHE	101	対象の キャッシュ制御 サンプル のバージョン番号の 100 倍

4.4.2　errnum_t 型 - エラーコード

定数名	設定値	内容
0	0	エラーなし
E_OTHERS	1	その他のエラー
E_FEW_ARRAY	2	固定長配列が不足したときのエラー
E_FEW_MEMORY	3	ヒープメモリー不足
E_FIFO_OVER	4	キューに入りきらなかったエラー
E_NOT_FOUND_SYMBOL	5	シンボルが定義されていない
E_NO_NEXT	6	次のリスト要素がない
E_ACCESS_DENIED	7	読み込み書き込み拒否エラー 別のスレッドにロックされている
E_NOT_IMPLEMENT_YET	9	未実装
E_ERRNO	0x0E(=14)	errno を参照
E_LIMITATION	0x0F(=15)	暫定の制限事項
E_STATE	0x10(=16)	現在の状態では実行できないというエラー
E_NOT_THREAD	0x11(=17)	スレッドではないエラー、割込みコンテキストから呼べない
E_PATH_NOT_FOUND	0x12(=18)	ファイルやフォルダーが見つからない
E_BAD_COMMAND_ID	0x16(=22)	コマンド ID が範囲外
E_TIME_OUT	0x17(=23)	タイムアウト
E_STACK_OVERFLOW	0x1C(=28)	スタック オーバーフロー
E_NO_DEBUG_TLS	0x1D(=29)	デバッグ用ワーク領域がない
E_EXIT_TEST	0x1E(=30)	テストの停止要求

4.4.3　r_ospl_async_state_t 型 - 状態

定数名	設定値	内容
R_OSPL_UNINITIALIZED	0	未初期化状態。 値は、グローバル変数の初期値と同じ 0
R_OSPL_RUNNABLE	1	非同期処理が呼び出せる状態
R_OSPL_RUNNING	2	非同期処理が起動中、または、割込みや入力を待機中
R_OSPL_INTERRUPTING	3	R_DRIVER_OnInterrupting 関数が呼び出せる状態
R_OSPL_INTERRUPTED	4	R_DRIVER_OnInterrupted 関数を呼び出す必要がある状態
R_OSPL_LOCKED	5	ドライバー以外が使用中

4.4.4　r_ospl_wait_t 型 - R_OSPL_THREAD_SetOnWait 関数のパラメーター

定数名	設定値	内容
R_OSPL_WAIT_POLLING	0	ポーリングで待つ
R_OSPL_WAIT_PM_THREAD	1	擬似マルチスレッドをしながら待つ。 待ちを行う関数を呼び出したときは、その引数に指定されたタイムアウト時間は無視され、関数からすぐに返ります。 待ちを行う関数の直後で、R_OSPL_THREAD_GetIsWaiting 関数(4.6.3(13))を呼び出してください。 ただし、タイムアウト時間が R_OSPL_INFINITE のときは、関数の中で待ち続けます。 これは、同期関数 R_DRIVER_Transfer の中から OS レス専用関数 R_OSPL_THREAD_GetIsWaiting を呼ぶと、OS レス依存になってしまうからです。 この設定のときは、CPU 使用率が常に 100% になります。

4.4.5　r_ospl_flush_t 型 - R_OSPL_MEMORY_Flush 関数のパラメーター

定数名	設定値	内容
R_OSPL_FLUSH_WRITEBACK_INVALIDATE	2	メモリーにライトバック（クリーン）してキャッシュを破棄する（L1 キャッシュのみ）
R_OSPL_FLUSH_WRITEBACK_INVALIDATE_2ND	8	メモリーにライトバック（クリーン）してキャッシュを破棄する（L2 キャッシュのみ）
R_OSPL_FLUSH_INVALIDATE	0	データ キャッシュを破棄する（L1 キャッシュのみ）

4.4.6　r_ospl_axi_cache_attribute_t 型 - AXI バスのキャッシュ属性

バスマスターが AXI バスにつながったキャッシュをどのように使うかの設定値。RZ/A1H では、R_OSPL_AXI_CACHE_ZERO 以外は、L2 キャッシュ PL310 の外にあるメモリーに対するキャッシュ制御の属性を設定するときに使います。 CPU が L2 キャッシュありでアクセスし、CPU 以外のバスマスターが L2 キャッシュにキャッシュなしでアクセスする場合、L2 キャッシュのフラッシュ（R_OSPL_FLUSH_WRITEBACK_INVALIDATE_2ND）が必要になります。

参考：「RZ/A1H グループユーザーズマニュアル　ハードウェア編」「5.8 AXI プロトコルの制御信号」

定数名	設定値	内容
R_OSPL_AXI_CACHE_ZERO	0	内部バス向け設定
R_OSPL_AXI_STRONGLY	0	ストロングリ オーダー メモリー
R_OSPL_AXI_DEVICE	1	デバイス メモリー
R_OSPL_AXI_UNCACHED	3	非キャッシュ・ノーマル アクセス
R_OSPL_AXI_WRITE_BACK_W	11	キャッシュあり、ライトバック、ライト時にアロケート
R_OSPL_AXI_WRITE_BACK	15	キャッシュあり、ライトバック、リード時とライト時にアロケート

4.4.7　r_ospl_axi_protection_t 型 - AXI バスの保護属性

RZ/A1H では、R_OSPL_AXI_CACHE_ZERO 以外は、AXI バスにつながった L2 キャッシュ PL310 の外にある SDRAM に対するセキュア属性になります。セキュア属性は、L2 キャッシュにヒットするかどうかに影響します。セキュア属性は、CPU の MMU(TTB)の NS ビットに対応します。

参考：「RZ/A1H グループユーザーズマニュアル　ハードウェア編」「5.8 AXI プロトコルの制御信号」

定数名	設定値	内容
R_OSPL_AXI_PROTECTION_ZERO	0	内部バス向け設定
R_OSPL_AXI_SECURE	0	セキュア
R_OSPL_AXI_NON_SECURE	2	非セキュア

4.4.8　r_ospl_async_type_t

定数名	設定値	内容
R_OSPL_ASYNC_TYPE_NORMAL	1	通常の非同期処理
R_OSPL_ASYNC_TYPE_FINALIZE	2	終了処理における非同期処理

4.4.9　bsp_int_err_t

定数名	設定値	内容
BSP_INT_SUCCESS	0	エラーなし
BSP_INT_ERR_NO_REGISTERED_CALLBACK	0x2101	コールバック関数が登録されていない
BSP_INT_ERR_INVALID_ARG	1	パラメーターエラー
BSP_INT_ERR_UNSUPPORTED	15	サポートしていない

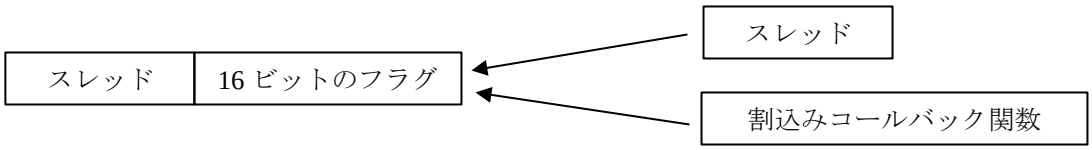
4.4.10　r_ospl_event_flags_t

スレッド付属イベントのビットフラグの ID を格納する整数型。

通常、r_ospl_event_flags_t 型の変数を持つ r_ospl_async_t 型の変数に通知先を設定して、その変数を処理対象に渡します。

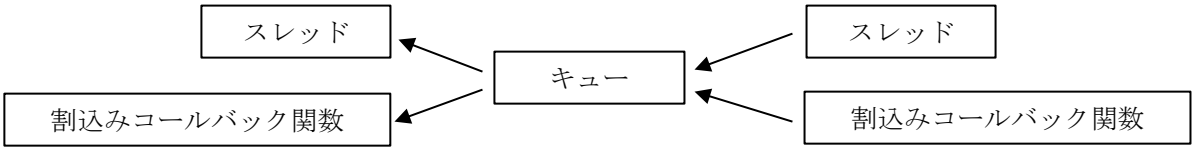
r_ospl_thread_id_t 型 の変数の値	r_ospl_event_flags_t 型 の変数の値
通知先のスレッドの ID	スレッド付属イベントのビット（＋ チェックビット）

スレッド付属イベント：



データの送信（非同期通信）をするときは、キュー（r_ospl_queue_id_t）も併用することがありますが、通常、キューはライブラリーの API の中に隠れるでしょう。

キュー：



4.4.11　r_ospl_table_flags_t

定数名	設定値	内容
R_OSPL_TABLE_I_LOCK	2	それぞれの関数の中で、I-ロック区間を作るかどうか
R_OSPL_TABLE_T_LOCK	1	それぞれの関数の中で、T-ロック区間を作るかどうか。

4.4.12　r_ospl_if_not_t

検索して見つからなかったときの処理方法の定数。

定数名	設定値	内容
R_OSPL_ALLOCATE_IF_NOT	1	キーが見つからなかったときは、未使用の配列番号を検索して追加します。配列が不足したら、追加せず、E_FEW_ARRAY エラーになります。キーが見つかったときは、その配列番号を出力します。
R_OSPL_ERROR_IF_NOT	0	キーが見つからなかったときは、E_NOT_FOUND_SYMBOL エラーにします。キーが見つかったときは、その配列番号を出力します。
R_OSPL_DO_NOTHING_IF_NOT	2	キーが見つからなかったときは、出力変数を入力時のままにします。キーが見つかったときは、その配列番号を出力します。表の中にキーがあるかどうかを判定するときに使います。
R_OSPL_OUTPUT_IF_NOT	3	キーが見つからなかったときは、未使用の配列番号を検索して追加します。不足したら、追加せず、E_FEW_ARRAY エラーになります。キーが見つかったときは、出力変数を入力時のままにします。 表の中にインデックスがあるかどうかを判定するときに使います。
R_OSPL_ALLOCATE_IF_EXIST_OR_IF_NOT	4	キーが見つかっても見つからなくても、未使用の配列番号を検索して追加します。 R_OSPL_TABLE_GetIndex 関数の in_Key 引数は無視します。不足したら、追加せず、E_FEW_ARRAY エラーになります。R_OSPL_ALLOCATE_IF_EXIST_OR_IF_NOT を使った後は、それ以外の r_ospl_if_not_t 型の定数は使用できません。 また、R_OSPL_TABLE_Free 関数も呼び出せません。R_OSPL_TABLE_InitConst を呼び出せば、使用できるようになります。

4.4.13　bsp_int_src_t

割込み番号の型。

bsp_int_src_t の内容となる型は、環境によって整数型または列挙型になります。

値は、BSP_INT_SRC_から始まります。

4.4.14　bsp_int_cb_t

割込みハンドラーの型。

bsp_int_cb_t の内容となる関数型の返り値や引数は、環境によって変わります。

OSPL 環境では、R_OSPL_CallInterruptCallback を経由してコールバックします。

4.4.15　bsp_int_cmd_t

定数名	設定値	内容
BSP_INT_CMD_INTERRUPT_ENABLE	0	割込みを許可します。 R_BSP_InterruptControl 関数の Parameter 引数には、FIT_NO_PTR を指定してください。
BSP_INT_CMD_INTERRUPT_DISABLE	1	割込みを禁止します。 R_BSP_InterruptControl 関数の Parameter 引数には、FIT_NO_PTR を指定してください。

4.4.16　mcu_lock_t

ロックするときに使うハードウェアのチャンネルの識別番号の列挙体。

命名規則は、BSP_LOCK_ から始まり、大文字のスネークケースです。

定義サンプル:

```
typedef enum {
    BSP_LOCK_DMAC0,
    BSP_LOCK_DMAC1,
          :
    BSP_NUM_LOCKS /* This entry is not a valid lock. It is used for sizing
       g_bsp_Locks[] array below. Do not touch! */
} mcu_lock_t;
```

4.4.17　その他の定数

定数名	設定値	内容
R_OSPL_NDEBUG	非定義 または 1	Debug 環境（非定義）か Release 環境（1）か。標準ライブラリーの NDEBUG と仕様は同じ OSPL 本体がデバッグ版、アプリケーションがリリース版でも動きます。 ライブラリー（コンパイル済みバイナリー）の内部からデバッグ版の OSPL を呼び出しているときは、デバッグ版で（R_OSPL_NDEBUG を定義して）OSPL 本体をコンパイルしてください。
R_OSPL_TEST_CODE	0 または 1	テスト用コードを有効にするかどうかのマクロ。 OSPL のパッケージの中の ospl フォルダーは、テスト用コードを含んでいます。 workspace¥src には、テストコードを含まない OSPL 本体があります。テスト用コードは、R_OSPL_TEST_CODE マクロによる分岐の中にあります。*_test.* ファイルはテスト用のソースファイルです。
R_OSPL_ERROR_BREAK	0 または 1	エラーブレーク機能を使用するかどうかのマクロ。
R_OSPL_PRINTF	（右記）	OSPL 内部から printf 出力するコードを有効にするかどうかの設定。 以下のいずれか。 ● R_OSPL_PRINTF_ENABLED ● R_OSPL_PRINTF_DISABLED ● R_OSPL_PRINTF_TO_INT_LOG

R_OSPL_INTERRUPT_HANDLER_IS	（右記）	割込みハンドラーの型の種類。 以下のいずれか。 ● R_OSPL_INTERRUPT_HANDLER_IS_GENERAL ➢ 割込みハンドラーは、引数がありません。 ➢ 割込みハンドラーは、返り値として何も返しません。 ➢ 割込みハンドラーの最後で呼び出さなければならない API はありません。 ● R_OSPL_INTERRUPT_HANDLER_IS_WITH_INT_SENSE ➢ 割込みハンドラーは、int_sence 引数を持ちます。 ● R_OSPL_INTERRUPT_HANDLER_IS_FOR_RTX ➢ RTX 用です。R_OSPL_INTERRUPT_HANDLER_IS_GENERAL と同じです。
R_OSPL_TLS_ERROR_CODE	0 または 1	TLS（スレッド ローカル ストレージ）に、エラーコードを格納するかどうか。 アプリケーションや使用するライブラリーによっては、1 である必要があります。
R_OSPL_STACK_CHECK_CODE	0 または 1	スタック チェックに関するコードを有効にするかどうかの設定。 1 に設定すると、IF マクロで条件判定する直前にスタック チェックを行います。 0 に設定すると、以下の関数や定数が無効になります。 ● R_OSPL_GET_STACK_POINTER ● R_OSPL_SET_END_OF_STACK ● R_OSPL_MOVE_END_OF_STACK ● R_OSPL_RESET_MIN_FREE_STACK_SIZE ● R_OSPL_GET_MIN_FREE_STACK_SIZE ● R_OSPL_STACK_CHECK_CANARY_VALUE OS によっては、実行中のスレッドが切り替わるときにスタック チェックを行いますが、その機能とは別の実装です。
R_OSPL_CPU_LOAD	0 または 1	CPU 使用率を計測するかどうかの設定。（OS レス専用）
R_OSPL_DEBUG_TOOL	0 または 1	デバッグツール機能を使用するかどうか
R_OSPL_TLS_EVENT_CODE	0 または 1	スレッド付属イベントのフラグの使用管理を有効にするかどうかの設定。 参考：(4.6.4(1)) R_OSPL_EVENT_Allocate
R_OSPL_DETECT_BAD_EVENT	0 または 1	スレッド付属イベントのフラグが重複していないことを、チェック ビットでチェックするかどうかの設定。
R_OSPL_EVENT_MUST_ALLOCATE	0 または 1	確保せずにイベントを使ったときに、R_OSPL_RaiseUnrecoverable 関数を呼び出すかどうかの設定。
R_OSPL_ALL_EVENT_ALLOCATE	0 または 1	R_OSPL_EVENT_Allocate に R_OSPL_EVENT_ALL_BITS を指定することをサポートしているかどうかの値。

R_OSPL_WAIT_OWNER_DEBUG	0 または 1	待ちに入る関数を呼び出したスレッドを Int ログに記録するかどうかの設定（OS レス専用、デバッグ用）。 R_OSPL_IS_PREEMPTION = 0　に定義されている OSPL にのみ本定数は存在します。 R_OSPL_THREAD_SetOnWait 関数に R_OSPL_WAIT_PM_THREAD を渡した擬似マルチスレッド環境において、待ち状態に入っているスレッドが、再度 同じ待ちに入る関数、または、R_OSPL_THREAD_ExitWaiting 関数を呼び出さないと、期待通りに動かなくなるのですが、再度待ちに入るときにどの関数に入るべきかを調べるときに使います。
R_OSPL_DEBUG_THREAD_COUNT	2 以上	デバッグ・ワーク領域に格納できるスレッド数＋割込みレベル数の最大値。 R_OSPL_ERROR_BREAK = 0 かつ、R_OSPL_TLS_ERROR_CODE = 0 のときは、デバッグ・ワーク領域は作られません。
R_OSPL_LIBRARY_MAKING	0 または 1	ライブラリーを作成しているかどうかのマクロ。 ワーク領域の構造体のメンバー変数が参照できなくなります。R_OSPL_QUEUE_DEF などの *_DEF マクロが使えなくなります。INLINE マクロの定義内容に影響することがあります。
R_OSPL_FLAG32_ALL_BITS	0xFFFFFFFF	r_ospl_flag32_t 型のすべてのビット
R_OSPL_STACK_CHECK_CANARY_VALUE	0x57AC512E	スタック領域の端や、まだ使われていない範囲に書かれるカナリア値。
BSP_CFG_USER_LOCKING_ENABLED	0 または 1	ユーザー定義のロック・オブジェクトを使うかどうか。 0 = OSPL が提供する C-ロックを使うように、下記のマクロを定義する 1 = 下記のマクロをユーザーが定義する ● BSP_CFG_USER_LOCKING_TYPE = r_ospl_c_lock_t ● BSP_CFG_USER_LOCKING_HW_LOCK_FUNCTION = R_BSP_HardwareLock ● BSP_CFG_USER_LOCKING_HW_UNLOCK_FUNCTION = R_BSP_HardwareUnlock ● BSP_CFG_USER_LOCKING_SW_LOCK_FUNCTION = R_BSP_SoftwareLock ● BSP_CFG_USER_LOCKING_SW_UNLOCK_FUNCTION = R_BSP_SoftwareUnlock
R_OSPL_UNLOCKED_CHANNEL	0xFEE	ロックされていないチャンネル番号を使うときに渡す値。

R_OSPL_FTIMER_IS	（右記）	フリーランタイマー機能(4.6.15(2))の内部で使用するチャンネル。 サンプルの設定値： ● R_OSPL_FTIMER_IS_OSTM1 (OS レス) ● R_OSPL_FTIMER_IS_MTU2_1_2 (OS あり)
R_OSPL_FTIMER_IS_OSTM0	0	OS タイマー - チャンネル０。 R_OSPL_FTIMER_IS に指定する値の１つ
R_OSPL_FTIMER_IS_OSTM1	1	OS タイマー - チャンネル１。 R_OSPL_FTIMER_IS に指定する値の１つ
R_OSPL_FTIMER_IS_MTU2_1_2	2	MTU2 - チャンネル１と２のカスケード接続。 R_OSPL_FTIMER_IS に指定する値の１つ

4.5　構造体/共用体

表 4.5　構造体/共用体一覧

章	型名	概要
4.5.1	r_ospl_thread_id_t	スレッドへのポインター
4.5.2	r_ospl_thread_def_t	スレッドの定義
4.5.3	r_ospl_flag32_t	32 ビットのフラグ
4.5.4	r_ospl_event_group_id_t	内部で使われる、イベント フラグの型
4.5.5	r_ospl_event_status_t	イベントの情報
4.5.6	r_ospl_async_t	通知設定
4.5.7	r_ospl_async_status_t	ドライバーの状態と割込みステータス
4.5.8	r_ospl_queue_id_t	スレッド間通信ができるキュー
4.5.9	r_ospl_queue_def_t	キューの定義の型
4.5.10	r_ospl_queue_status_t	キューの状態（使用数など）
4.5.11	BSP_CFG_USER_LOCKING_TYPE	C-ロック
4.5.12	r_ospl_c_lock_t	C-ロック。内部用
4.5.13	r_ospl_table_t	配列番号表
4.5.14	r_ospl_table_status_t	配列番号表の状況を表す構造体
4.5.15	r_ospl_memory_spec_t	メモリーやキャッシュの仕様
4.5.16	r_ospl_ftimer_spec_t	フリーラン タイマーの精度
4.5.17	r_ospl_caller_t	割込みコールバック関数の管理
4.5.18	r_ospl_interrupt_t	割込みの発信元に関する構造体。 割込み番号など

4.5.1　r_ospl_thread_id_t

スレッドへのポインター型。

メンバー変数は参照/変更しないでください。

内部に 16 ビットのイベントを持ちます。

R_OSPL_THREAD_Create 関数で生成します。

OS ありの場合は、OS が定義するスレッドの型と OSPL が定義する r_ospl_thread_id_t 型は同じ型になります。R_OSPL_THREAD_GetCurrentId 関数以外は、OS の API を使用してください。

OS レスの場合は、プリエンプションできません（擬似マルチスレッド）。主に 16 ビット（16 種類）より多くのイベントを識別するためにスレッドを分けます。

r_ospl_thread_id_t 型の変数に、R_OSPL_THREAD_NULL を代入することができます。

4.5.2　r_ospl_thread_def_t

スレッドの定義に関する型。

メンバー変数は参照/変更しないでください。

R_OSPL_THREAD_DEF マクロで定義します。

4.5.3　r_ospl_flag32_t

32 ビットのフラグの型。

メンバー変数は参照/変更しないでください。

R_OSPL_FLAG32_InitConst 関数で初期化します。

4.5.4　r_ospl_event_group_id_t

スレッドとは異なる ID を持つ OSPL 内部で使われる、イベント フラグの型。

OS のイベント フラグが 16 ビットより多くのビットがあっても、OSPL では使いません。イベント クループへのライトは、スレッドだけでなく、割込みコンテキストから呼び出すことができます。

r_ospl_event_group_id_t 型は、スレッド付属イベントの機能がない OS において、OS が定義するイベント グループの型と同じ型として内部で定義されます。 R_OSPL_EVENT_GROUP_CODE = 0 のときは、r_ospl_event_group_id_t 型は定義されません。

OSPL のユーザーは、スレッド付属イベントの API を呼び出してください。 イベント グループは、その内部で使います。R_OSPL_EVENT_Allocate 関数の内部から R_OSPL_EVENT_GROUP_Create 関数を呼び出すことで生成されたイベント グループを確保します。 R_OSPL_EVENT_Free 関数の内部から R_OSPL_EVENT_GROUP_Delete 関数を呼び出すことで、イベント グループを削除できるように開放します。 生成・削除のタイミングは、移植層である R_OSPL_EVENT_GROUP_Create 関数と R_OSPL_EVENT_GROUP_Delete 関数の実装によります。 イベント グループを待つスレッドを変更することはできません。r_ospl_event_group_id_t 型の変数に、R_OSPL_EVENT_GROUP_NULL を代入することができます。

4.5.5　r_ospl_event_status_t

概　要	イベントの情報。
ヘッダー	r_ospl.h
説　明	R_OSPL_EVENT_GetStatus 関数で取得できます。 R_OSPL_DETECT_BAD_EVENT s = 1 のときのみ使えます。

メンバー変数

bit_flags32_t AllocatedEvents	確保されているイベントのビット。 参考：(4.6.4(1)) R_OSPL_EVENT_Allocate
bit_flags32_t UnexpectedEvents	予期しないでセットされたイベントのビット。

4.5.6　r_ospl_async_t

概　要	通知設定の型
ヘッダー	r_ospl.h
説　明	R_DRIVER_TransferStart 関数に指定します。 参考：R_OSPL_EVENT_Wait 関数 (4.6.4(4)) 通知の詳細は、別途キュー（r_ospl_queue_id_t）を内部に持つ API から取得することがあります。

メンバー変数

bit_flags_fast32_t　Flags	フラグド構造体パラメーター。（4.8.2） 指定するメンバー変数に合わせて、0 または、以下の論理和を指定。 ● R_F_OSPL_A_Thread ● R_F_OSPL_A_EventValue ● R_F_OSPL_I_Thread ● R_F_OSPL_I_EventValue ● R_F_OSPL_InterruptCallback ● R_F_OSPL_Delegate 下記のプリセット（メンバー変数の組み合わせ）を上記の値との論理和で指定することもできます。 ● R_F_OSPL_AsynchronousPreset 参照：R_OSPL_ASYNC_SetDefaultPreset 関数
void*　Delegate	アプリケーション定義の値。ドライバーおよび OSPL はアクセスしません。 割込みコールバック関数の引数から参照できます。 Flags メンバー変数に設定された R_F_OSPL_Delegate は無視されます。

r_ospl_thread_id_t A_Thread	非同期処理が完了したときの通知を受ける A-イベントを持つ A-スレッド。 設定するときは、Flags メンバー変数に R_F_OSPL_A_Thread を追加してください。 デフォルトは R_OSPL_THREAD_NULL。 R_OSPL_THREAD_NULL を指定したときは通知しない。 通常、R_OSPL_THREAD_GetCurrentId 関数の返り値。 通知を受信するには、通常、R_OSPL_EVENT_Wait 関数を呼び出します。
r_ospl_event_flags_t A_EventValue	A-イベントに設定する通知先。 設定するときは、Flags メンバー変数に R_F_OSPL_A_EventValue を追加してください。 通知先に設定するイベントフラグの値。 フラグは 16 ビットです。 デフォルトは、R_OSPL_UNUSED_FLAG です。 ただし、R_DRIVER_SetDefaultAsync 関数の内容によって変わります。 ドライバーの内部で、R_OSPL_EVENT_Allocate 関数や、R_OSPL_EVENT_Free 関数を呼び出す場合、ドライバーの内部で値は変化します。
r_ospl_thread_id_t I_Thread	割込み応答処理が必要である通知を受ける I-イベントを持つ I-スレッド。 通常、アプリケーションが設定する必要はありません。 設定するときは、Flags メンバー変数に R_F_OSPL_I_Thread を追加してください。 A_Thread と同じ値を設定可能。 デフォルトは R_OSPL_THREAD_NULL。 R_OSPL_THREAD_NULL を指定したときは、割込みコールバック関数の中から R_DRIVER_OnInterrupted 関数が自動的に呼び出されますが、そうでないときは、I-イベントを受信したスレッドが R_DRIVER_OnInterrupted 関数を呼び出してください。 OS ありでは、R_OSPL_THREAD_NULL を指定したとき、ドライバー内部で生成した I-スレッドに通知するドライバーの仕様である可能性があります。 R_DRIVER_SetDefaultAsync 関数の内容を確認してください。 参考：（4.7）シーケンス図
r_ospl_event_flags_t I_EventValue	I-イベントに設定するフラグパターン値。 通常、アプリケーションが設定する必要はありません。 設定するときは、Flags メンバー変数に R_F_OSPL_I_EventValue を追加してください。 通知先に設定するイベントフラグの値。 フラグは 16 ビットです。 デフォルトは、I_Thread == R_OSPL_THREAD_NULL なら 0、それ以外なら 0x0002 = R_OSPL_I_FLAG。 ただし、R_DRIVER_SetDefaultAsync 関数の内容によって変わります。

r_ospl_callback_t InterruptCallback	割込みコールバック関数。 内部の割込みに対してもコールバックされます。 通常、アプリケーションが設定する必要はありません。 設定するときは、Flags メンバー変数に R_F_OSPL_InterruptCallback を追加してください。 デフォルトは NULL。 NULL の場合、ドライバーが用意したデフォルトの割込みコールバック関数。 ドライバーが応答する割込みの、割込みコンテキストでコールバックされますが、そうでないドライバーもありえます。
errnum_t　ReturnValue	非同期処理の返り値。 非同期処理を開始したときに入っている値は破棄されます。 非同期処理が完了したら ReturnValue の値をチェックする必要があります。

4.5.7　r_ospl_async_status_t

概　要　OSPL が定義する、ドライバーの状態と、割込みステータスの構造体

ヘッダー　r_ospl.h

説　明　ドライバーが定義する r_driver_async_status_t 型には、存在しないメンバー変数や追加されたメンバー変数があるかもしれません。

メンバー変数

r_ospl_async_state_t State	ドライバーの状態。 非同期処理中や割込み処理中など。要 volatile
bool_t IsEnabledInterrupt	チャンネルに関わるすべての割込み線を許可しているかどうか。
r_ospl_flag32_t InterruptEnables	チャンネルに関わる割込み線のうち、割込み許可している線。
r_ospl_flag32_t InterruptFlags	割込み状態レジスターのコピー。 各ビットに対応するシンボルはドライバー内部で定義し、ドライバーによっては、非公開です。
r_ospl_flag32_t CancelFlags	停止処理の開始から完了までの状態を扱うドライバー内部のフラグ。
union　LockOwner	ロックしているオーナー。 R_OSPL_IS_PREEMPTION = 1 に定義されているときのみ、本メンバー変数は存在します。
r_ospl_thread_id_t LockOwner.Thread	ロックしているオーナー。スレッド型。 R_OSPL_THREAD_NULL=ロックしていない。
void* LockOwner.Context	ロックしているオーナー。コンテキスト型。 NULL=ロックしていない

4.5.8　r_ospl_queue_id_t

スレッド間通信ができる（排他制御がある）キューの型。

１つのキューの中にある要素はすべて同じサイズ。要素のサイズはそれぞれのキューごとに異なるようにできます。

メンバー変数は参照/変更しないでください。

R_OSPL_QUEUE_Create 関数(4.6.8(3))で初期化します。

r_ospl_queue_id_t 型の変数には、R_OSPL_QUEUE_NULL を代入することができます。

4.5.9　r_ospl_queue_def_t

キューの定義の型。

メンバー変数は参照/変更しないでください。

R_OSPL_QUEUE_DEF マクロで定義します。

4.5.10　r_ospl_queue_status_t

概　要	キューの状態（使用数など）。
ヘッダー	r_ospl.h
説　明	R_OSPL_QUEUE_GetStatus 関数で取得できます。

メンバー変数

int_fast32_t　UsedCount	確保されているキューの要素の数
int_fast32_t　MaxCount	キューに入れることができる最大の要素数

4.5.11　BSP_CFG_USER_LOCKING_TYPE

C-ロックに相当する型です。 ロックするときに内部変数の排他制御を行います。 C-ロックに成功したオーナー以外に対してアクセスを禁止する処理は、別途行ってください。

ハードウェアのチャンネルや、ソフトウェアの使用権を管理します。 待ちがある T-ロックの用途には使えません。

BSP_CFG_USER_LOCKING_TYPE は、BSP_CFG_USER_LOCKING_ENABLED が 0 のときは、r_ospl_c_lock_t と同じ型になります。1 のときは、BSP_CFG_USER_LOCKING_TYPE 型を定義してください。

ロックに成功したら、そのオーナーがある権利を有したことになります。ある権利とは、ロックする対象に対してある処理ができることです。 権利がなければ、たとえば、E_ACCESS_DENIED エラーになります。オーナーは、ロック対象の仕様によって、スレッドまたはコンテキストのどちらかです。

- ロックに成功したら、オーナーID をロック対象に記録します。
- ロックを解除する直前で、ロック対象からオーナーID を削除します。
- ロック中にロック対象を操作する API 関数の中で、オーナーID をチェックしてください。

同じ周辺機能を操作する別のドライバーがある環境では、OSPL は、その別のドライバーにロック状態の管理を移譲する実装の場合があります。

4.5.12　r_ospl_c_lock_t

C-ロックの型。 OSPL 内部用。 排他制御はしません。 API は、BSP_CFG_USER_LOCKING_TYPE を参照。

r_ospl_c_lock_t* 型の変数には、NULL を代入することができます。

4.5.13　r_ospl_table_t

配列番号表の型。

メンバー変数は参照/変更しないでください。

R_OSPL_TABLE_DEF、または、R_OSPL_TABLE_InitConst 関数で初期化します(4.6.1(12))。

キー（void* 型、または、uintptr_t 型の任意の値、NULL 可能）から、配列番号に変換します。配列番号は、0 ～ (n-1) の範囲の整数で、n は任意に指定できます。各関数の中で I-ロックや T-ロックをするかどうかを選ぶことができます。R_OSPL_NDEBUG が定義されていないとき、不正なリエントラントがされていないかチェックします。不正なリエントラントを検出したら、R_OSPL_RaiseUnrecoverable(E_STATE) を呼び出します。

r_ospl_thread_id_t 型から配列番号に変換すれば、スレッド ローカル ストレージと同等の目的に使うことができます。 配列番号表を固定長メモリプールの代わりとして使うこともできます。

4.5.14　r_ospl_table_status_t

概　要　　配列番号表の状況を表す構造体。
ヘッダー　　r_ospl.h
説　明　　(4.6.13(8)) R_OSPL_TABLE_GetStatus 関数で取得できます。
メンバー変数

int_fast32_t　Count	現在の格納されている要素の数
int_fast32_t　MaxCount	配列番号表に格納できる最大の要素数

4.5.15　r_ospl_memory_spec_t

概　要　　メモリーやキャッシュの仕様の構造体。
ヘッダー　　r_ospl.h
説　明
メンバー変数

uint_fast32_t CacheLineSize	キャッシュラインのサイズ（バイト）

4.5.16　r_ospl_ftimer_spec_t

概　要　　フリーラン タイマーの精度
ヘッダー　　r_ospl.h
説　明　　R_OSPL_FTIMER_InitializeIfNot 関数で取得できます。
1 ミリ秒の精度なら、msec_Numerator = 1、msec_Denominator = 1 になります。
1 マイクロ秒の精度なら、msec_Numerator = 1、msec_Denominator = 1000 になります。
カウント値は、MaxCount を超えたら 0 に回り込みます。
メンバー変数

uint32_t msec_Numerator	１カウントあたりのミリ秒数の分子
uint32_t msec_Denominator	１カウントあたりのミリ秒数の分母
uint32_t　MaxCount	カウント値の最大値
uint32_t ExtensionOfCount	経過の猶予期間のカウント R_OSPL_FTIMER_IsPast で Now 引数が TargetTime 引数＋ExtensionOfCount を超えたらエラーになります

4.5.17　r_ospl_caller_t

割込みコールバック関数の管理する構造体の型。

メンバー変数は参照/変更しないでください。

4.5.18　r_ospl_interrupt_t

概　要　　割込みの発信元に関する構造体。
ヘッダー　　r_ospl.h
説　明　　-
メンバー変数

bsp_int_src_t　IRQ_Num	割込み番号
int_fast32_t　ChannelNum	チャンネル番号
int_fast32_t　Type	ドライバー内部定義の番号
void*　Delegate	ドライバー内部定義のポインター

4.6　関数

4.6.1　一覧

章	分類
(1)	OSPL のバージョンに関する関数の一覧
(2)	r_ospl_thread_id_t 型 - スレッドに関する関数の一覧
(3)	スレッド付属イベントに関する関数の一覧
(4)	r_ospl_flag32_t 型 - フラグに関する関数の一覧
(5)	bit_flags_fast32_t 型 - ビットフラグに関する関数の一覧
(6)	r_ospl_async_t 型 - 通知に関する関数の一覧
(7)	r_ospl_queue_id_t 型 - キューに関する関数の一覧
(8)	全割込み禁止区間に関する関数の一覧
(9)	割込み処理に関する関数の一覧
(10)	BSP_CFG_USER_LOCKING_TYPE 型に関する関数の一覧
(11)	r_ospl_c_lock_t 型に関する関数の一覧
(12)	r_ospl_table_t - 配列番号表に関する関数の一覧
(13)	メモリーに関する関数の一覧
(14)	時間に関する関数の一覧
(15)	アイドル状態に関する関数の一覧
(16)	r_ospl_caller_t 型 - 割込みコールバック関数に関する関数の一覧
(17)	エラー処理、デバッグに関する関数の一覧
(18)	静的コード解析ツールのレビュー済みタグ
(19)	マルチコンパイラー対応
(20)	OSPL の下位層に関する関数の一覧
(21)	ドライバーの API に共通する関数仕様の一覧
(22)	ドライバーの下位層に共通する関数仕様の一覧

(1)　OSPL のバージョンに関する関数の一覧

章	関数名	概要
4.6.2(1)	R_OSPL_GetVersion	OSPL のバージョン番号を返します
4.6.2(2)	R_OSPL_IsPreemption	プリエンプションに対応しているかどうかを返します

(2)　r_ospl_thread_id_t 型 - スレッドに関する関数の一覧

章	関数名	概要
4.6.3(1)	R_OSPL_THREAD_DEF	スレッドのワーク領域や初期値を定義します（OS レス専用）
4.6.3(2)	R_OSPL_THREAD	スレッドのワーク領域を返します（OS レス専用）
4.6.3(3)	R_OSPL_THREAD_Create	擬似スレッドを生成します（OS レス専用）
4.6.3(4)	R_OSPL_THREAD_Destroy	擬似スレッドを削除します（OS レス専用）
4.6.3(5)	R_OSPL_THREAD_GetArgument	現在実行中のスレッドが生成されたときに渡されたパラメーターを返します（OS レス専用）
4.6.3(6)	R_OSPL_THREAD_SetCurrentId	現在実行中のスレッド ID を設定します（OS レス専用）
4.6.3(7)	R_OSPL_THREAD_GetCurrentId	現在実行中のスレッド ID を返します（OS レス、OS あり共通）
4.6.3(8)	R_OSPL_THREAD_GetMainId	メインスレッドの ID を返します（OS レス専用）
4.6.3(9)	R_OSPL_THREAD_SetDelegate	スレッド付属の変数に値を設定します（OS レス専用）

4.6.3(10)	R_OSPL_THREAD_GetDelegate	スレッド付属の変数の値を返します（OS レス専用）
4.6.3(11)	R_OSPL_THREAD_SetOnWait	現在のスレッドについて、待つときの動きを設定します
4.6.3(12)	R_OSPL_THREAD_GetOnWait	現在のスレッドについて、待つときの動きを取得します
4.6.3(13)	R_OSPL_THREAD_GetIsWaiting	現在のスレッドが、待っている状態かどうかを返します
4.6.3(14)	R_OSPL_THREAD_ExitWaiting	現在のスレッドが待っている状態なら、その状態を中止します

(3) スレッド付属イベントに関する関数の一覧

スレッドが持つ 16 ビットのフラグに対応する同期プリミティブ。

別のスレッドや割込みコールバック関数から、フラグの一部を立てる（ビットを 1 にする）ことで、スレッドに通知します。

章	関数名	概要
4.6.4(1)	R_OSPL_EVENT_Allocate	イベントを確保します。
4.6.4(2)	R_OSPL_EVENT_Set	１つまたはいくつかのビットを 1 に設定します
4.6.4(3)	R_OSPL_EVENT_Clear	１つまたはいくつかのビットを 0 に設定します
4.6.4(4)	R_OSPL_EVENT_Wait	16 ビットのフラグが立つまで待ち、受信したフラグをクリアします
4.6.4(5)	R_OSPL_EVENT_GetStatus	イベントの状態を取得します。
4.6.4(6)	R_OSPL_EVENT_Free	イベントを返却します。

32 ビットを使わない理由は、16 ビットしかない OS が存在するためと、下記のように上位 16 ビットにタイムアウトやチェック ビットを割当てられるようにするためです。

ビット	内容
31	Reserved
30	R_OSPL_TIMEOUT
29	Reserved
28	R_OSPL_UNUSED_FLAG
27:20	チェック ビット - イベントを確保するたびに変わる値
19:16	チェック ビット - 確保したイベントフラグのビット番号
15:3	スレッド付属のイベントフラグ
2	スレッド付属のイベントフラグ - R_OSPL_FINAL_A_FLAG
1	スレッド付属のイベントフラグ - R_OSPL_I_FLAG
0	スレッド付属のイベントフラグ - R_OSPL_A_FLAG

(4) r_ospl_flag32_t 型 - フラグに関する関数の一覧

割込みステータスなどを通知するための、32 ビットフラグの 1 段バッファー。

API は、アトミックではありません。 排他制御が必要です。

bit_flags32_t 型変数と同様に、いくつでも生成することができます。

章	関数名	概要
4.6.5(1)	R_OSPL_FLAG32_InitConst	32 ビットのフラグをすべての 0 にクリアします
4.6.5(2)	R_OSPL_FLAG32_Set	１つまたはいくつかのビットを 1 に設定します
4.6.5(3)	R_OSPL_FLAG32_Clear	１つまたはいくつかのビットを 0 に設定します
4.6.5(4)	R_OSPL_FLAG32_Get	32 ビットのフラグの値を取得します
4.6.5(5)	R_OSPL_FLAG32_GetAndClear	フラグの値を返し、すべてのビットを 0 にクリアします

(5)　bit_flags_fast32_t 型 - ビットフラグに関する関数の一覧

bit_flags_fast32_t 型は、uint_fast32_t 型の ビットフラグ型です。整数型をバイナリと見て、1 ビットずつのビットの集合とした型です。

章	関数名	概要
4.6.6(1)	IS_BIT_SET	指定したビットが 1 かどうかを評価します
4.6.6(2)	IS_ANY_BITS_SET	指定した複数のビットのどれかが 1 かどうかを評価します
4.6.6(3)	IS_ALL_BITS_SET	指定した複数のビットのすべてが 1 かどうかを評価します
4.6.6(4)	IS_BIT_NOT_SET	指定したビットが 0 かどうかを評価します
4.6.6(5)	IS_ANY_BITS_NOT_SET	指定した複数のビットのどれかが 0 かどうかを評価します
4.6.6(6)	IS_ALL_BITS_NOT_SET	指定した複数のビットのすべてが 0 かどうかを評価します

(6)　r_ospl_async_t 型 - 通知に関する関数の一覧

章	関数名	概要
4.6.7(1)	R_OSPL_ASYNC_SetDefaultPreset	各メンバー変数にプリセットの値を設定します。

(7)　r_ospl_queue_id_t 型 - キューに関する関数の一覧

章	関数名	概要
4.6.8(1)	R_OSPL_QUEUE_DEF	キューの属性とワーク領域を定義します
4.6.8(2)	R_OSPL_QUEUE	キューの属性とワーク領域を返します
4.6.8(3)	R_OSPL_QUEUE_Create	キューを初期化します
4.6.8(4)	R_OSPL_QUEUE_GetStatus	キューの状態（使用数など）を取得します
4.6.8(5)	R_OSPL_QUEUE_Allocate	キューからメモリー領域を確保します。待ち可能
4.6.8(6)	R_OSPL_QUEUE_Put	キューに入れます
4.6.8(7)	R_OSPL_QUEUE_Get	キューから取り出します。待ち可能
4.6.8(8)	R_OSPL_QUEUE_Free	メモリー領域をキューに返却します
4.6.8(9)	R_OSPL_GetQueueAsSingletonLock	シングルトンの生成と削除をするときに T-ロックするオブジェクトを返します。

(8)　全割込み禁止区間に関する関数の一覧

すべての割込みを禁止/許可することで、すべての割込みハンドラーと排他制御します。

ただし、NMI は対象外。 多重割込みについては、「4.8.3 多重割込みについて」 を参照。

章	関数名	概要
4.6.9(1)	R_OSPL_EnableAllInterrupt	すべての割込みを禁止していたことを解除します
4.6.9(2)	R_OSPL_DisableAllInterrupt	すべての割込みを禁止します
4.6.9(3)	R_OSPL_GetIsAllInterruptEnabled	すべての割込みが許可されているかどうかを返します

(9)　割込み処理に関する関数の一覧

章	関数名	概要
4.6.10(1)	R_BSP_InterruptWrite	割込みハンドラーを登録します
4.6.10(2)	R_BSP_InterruptRead	登録されている割込みハンドラーを返します
4.6.10(3)	R_BSP_InterruptControl	割込みを制御します
4.6.10(4)	R_OSPL_SetInterruptPriority	割込みの優先度を設定します
4.6.10(5)	R_OSPL_END_OF_INTERRUPT	割込みハンドラーの最後で呼び出すマクロ

(10) BSP_CFG_USER_LOCKING_TYPE 型に関する関数の一覧

章	関数名	概要
4.6.11(1)	R_OSPL_LockChannel	チャンネル番号で、ロックします。
4.6.11(2)	R_OSPL_UnlockChannel	チャンネル番号で、ロック解除します。
4.6.11(3)	R_BSP_HardwareLock	ハードウェアの識別番号で、ロックします。
4.6.11(4)	R_BSP_HardwareUnlock	ハードウェアの識別番号で、ロックを解除します。
4.6.11(5)	R_BSP_SoftwareLock	ロックします。
4.6.11(6)	R_BSP_SoftwareUnlock	ロックを解除します。

(11) r_ospl_c_lock_t 型に関する関数の一覧

章	関数名	概要
4.6.12(1)	R_OSPL_C_LOCK_InitConst	C-ロック オブジェクトを初期化します。
4.6.12(2)	R_OSPL_C_LOCK_Lock	ロックします。
4.6.12(3)	R_OSPL_C_LOCK_Unlock	ロックを解除します。

(12) r_ospl_table_t - 配列番号表に関する関数の一覧

章	関数名	概要
4.6.13(1)	R_OSPL_TABLE_DEF	配列番号表のワーク領域や初期値を宣言します
4.6.13(2)	R_OSPL_TABLE	配列番号表を返します
4.6.13(3)	R_OSPL_TABLE_InitConst	配列番号表を初期化します
4.6.13(4)	R_OSPL_TABLE_SIZE	配列番号表に使う領域のサイズを計算します
4.6.13(5)	R_OSPL_TABLE_GetIndex	キーから配列番号を返します
4.6.13(6)	R_OSPL_TABLE_Free	配列番号表から除外します（キー指定）
4.6.13(7)	R_OSPL_TABLE_FreeByIndex	配列番号表から除外します（配列番号指定）
4.6.13(8)	R_OSPL_TABLE_GetStatus	配列番号表の状況を取得します

(13) メモリーに関する関数の一覧

章	関数名	概要
4.6.14(1)	R_OSPL_MEMORY_Flush	キャッシュをフラッシュします
4.6.14(2)	R_OSPL_MEMORY_RangeFlush	キャッシュをフラッシュします。 仮想アドレスの範囲指定あり
4.6.14(3)	R_OSPL_MEMORY_GetLevelOfFlush	指定したメモリーをフラッシュするのに必要なレベルを取得します
4.6.14(4)	R_OSPL_MEMORY_GetMaxLevelOfFlush	すべてのメモリーをフラッシュするレベルを取得します
4.6.14(5)	R_OSPL_MEMORY_GetSpecification	メモリーやキャッシュの仕様を取得します。
4.6.14(6)	R_OSPL_ToPhysicalAddress	仮想アドレスから物理アドレスに変換します
4.6.14(7)	R_OSPL_ToCachedAddress	キャッシュ領域のアドレスに変換します
4.6.14(8)	R_OSPL_ToUncachedAddress	非キャッシュ領域のアドレスに変換します
4.6.14(9)	R_OSPL_MEMORY_Barrier	メモリー バリアを設定します。
4.6.14(10)	R_OSPL_InstructionSyncBarrier	命令同期バリアを設定します。
4.6.14(11)	R_OSPL_AXI_Get2ndCacheAttribute	アドレスから AXI バスの L2 キャッシュの属性を取得します

4.6.14(12)	R_OSPL_AXI_GetProtection	アドレスから AXI バスの保護属性を取得します

(14) 時間に関する関数の一覧

章	関数名	概要
4.6.15(1)	R_OSPL_Delay	指定した時間だけ待ちます
4.6.15(2)	R_OSPL_FTIMER_InitializeIfNot	フリーラン タイマーを起動します
4.6.15(3)	R_OSPL_FTIMER_Get	フリーラン タイマーの現在の時刻を返します
4.6.15(4)	R_OSPL_FTIMER_IsPast	指定した時刻を経過したかどうかを返します
4.6.15(5)	R_OSPL_FTIMER_TimeToCount	ミリ秒単位から、フリーランタイマーの単位に変換します
4.6.15(6)	R_OSPL_FTIMER_CountToTime	フリーランタイマーの単位から、ミリ秒単位に変換します
4.6.15(7)	R_OSPL_FTIMER_GetSpecification	フリーランタイマーの精度を取得します。

(15) アイドル状態に関する関数の一覧

主に、CPU 使用率の計測に使用します。

OS ありの環境で CPU 使用率を計測するときは、OSPL を使用する代わりに、最も優先度の低いアイドルスレッドを作成し、そのスレッドの中で変数をカウントアップし続け、周期的に入るタイマー割込みで、そのカウント値を参照します。 事前にアイドル スレッドだけが動くときのカウントアップ値を 100% の空きとして CPU 使用率の計算してください。また、ICE の機能で測定できる場合もあります。

章	関数名	概要
4.6.16(1)	R_OSPL_IDLE_Start_CPU_Load	CPU 使用率の計測を開始します
4.6.16(2)	R_OSPL_IDLE_Print_CPU_Load	CPU 使用率を printf 表示します

(16) r_ospl_caller_t 型 - 割込みコールバック関数に関する関数の一覧

章	関数名	概要
4.6.17(1)	R_OSPL_CallInterruptCallback	割込みコールバック関数を呼び出します。 ドライバーの OS 移植層から呼び出します
4.6.17(2)	r_ospl_callback_t	割込みコールバック関数の型

(17) エラー処理、デバッグに関する関数の一覧

章	関数名	概要
4.6.18(1)	CHK	エラーが発生していたら、無限ループに入ります
4.6.18(2)	R_OSPL_RaiseUnrecoverable	回復不能なエラーを発生させます
4.6.18(3)	R_DEBUG_BREAK	ブレークします
4.6.18(4)	R_DEBUG_BREAK_IF_ERROR	エラー状態ならブレークします
4.6.18(5)	IF	条件式が 0 以外（真）ならブレークしてエラー状態にします
4.6.18(6)	IF_D	デバッグ版のみ比較を行う、IF
4.6.18(7)	ASSERT_R	表明（契約プログラミング）を行う
4.6.18(8)	ASSERT_D	デバッグ版のみ有効な、ASSERT_R
4.6.18(9)	R_STATIC_ASSERT	静的に条件を満たしていないときにコンパイルエラーにする表明（契約プログラミング）を行う
4.6.18(10)	R_STATIC_ASSERT_GLOBAL	グローバルスコープでの R_STATIC_ASSERT
4.6.18(11)	R_NOOP	何もしない関数

4.6.18(12)	R_OSPL_MergeErrNum	終了処理で発生したエラーコードをマージします
4.6.18(13)	R_OSPL_SetErrNum	TLS に、エラーコードを格納します
4.6.18(14)	R_OSPL_GetErrNum	TLS に、格納されているエラーコードを返します
4.6.18(15)	R_OSPL_CLEAR_ERROR	エラー状態を解消します
4.6.18(16)	R_OSPL_NOTIFY_ERROR	エラーを他のスレッドに通知します
4.6.18(17)	R_OSPL_SET_BREAK_ERROR_ID	エラーが発生した瞬間にブレークするように設定します
4.6.18(18)	R_OSPL_GET_ERROR_ID	現在発生中のエラーについて、エラーが発生した順番の番号を返します
4.6.18(19)	R_OSPL_DEBUG_WORK_SIZE	デバッグ用ワーク領域のサイズを計算します
4.6.18(20)	R_OSPL_GetCurrentThreadError	現在のスレッドに対するエラーのデバッグ情報を返します
4.6.18(21)	R_OSPL_FreeCurrentThreadError	現在のスレッドに対するエラーのデバッグ情報を格納する領域を開放します
4.6.18(22)	R_OSPL_CHANGE_THREAD_LOCKED_COUNT	現在のスレッドだけがアクセスできるカウンターの値を変更します
4.6.18(23)	R_OSPL_GET_THREAD_LOCKED_COUNT	現在のスレッドだけがアクセスできるカウンターの値を返します
4.6.18(24)	R_OSPL_GET_STACK_POINTER	現在のスタック ポインターの値を返します
4.6.18(25)	R_OSPL_SET_END_OF_STACK	現在のスレッドのスタック領域の端を設定します
4.6.18(26)	R_OSPL_MOVE_END_OF_STACK	スタック領域の端を移動します。
4.6.18(27)	R_OSPL_CHECK_STACK_OVERFLOW	スタック オーバーフローが発生したかどうかをチェックします
4.6.18(28)	R_OSPL_RESET_MIN_FREE_STACK_SIZE	現在のスレッドが持つスタックの空きサイズの最小値をリセットします
4.6.18(29)	R_OSPL_GET_MIN_FREE_STACK_SIZE	現在のスレッドが持つスタックの空きサイズの最小値をカウントします
4.6.18(30)	R_OSPL_GET_MIN_STACK_POINTER	今まで最も多くスタック領域を使用した位置を探します。
4.6.18(31)	R_D_Add	ウォッチする整数変数またはポインター変数を登録します
4.6.18(32)	R_D_Watch	ウォッチします
4.6.18(33)	R_D_AddToIntLog	高速でログに記録します
4.6.18(34)	R_D_Counter	通過する回数をカウントします

(18) 静的コード解析ツールのレビュー済みタグ

章	関数名	概要
4.6.19(1)	IS	MISRA 13.2 の警告に対処したコードを読みやすくします。
4.6.19(2)	R_OSPL_ReturnFalse	if ブロックを無効にするときに警告されるときの対策
4.6.19(3)	R_UNREFERENCED_VARIABLE	変数が参照されていないという警告を発生させないようにします
4.6.19(4)	R_UNREFERENCED_VARIABLE2	複数指定できる R_UNREFERENCED_VARIABLE
4.6.19(5)	R_UNREFERENCED_VARIABLE3	複数指定できる R_UNREFERENCED_VARIABLE
4.6.19(6)	R_UNREFERENCED_VARIABLE4	複数指定できる R_UNREFERENCED_VARIABLE

(19) マルチコンパイラー対応

章	マクロ名	概要
4.6.20(1)	R_OSPL_SECTION	関数や変数にセクション名を付けます。
4.6.20(2)	R_OSPL_ALIGNMENT	グローバル変数の先頭アドレスをアラインメントします。
4.6.20(3)	R_COUNT_OF	配列の要素数を返します。
4.6.20(4)	INLINE	C99 の inline 関数
4.6.20(5)	STATIC_INLINE	C99 の static inline 関数
4.6.20(6)	R_ADDRESS_Add	アドレスの値をバイト単位で加算します。
4.6.20(7)	R_OSPL_CountLeadingZeros	最上位ビットから 0 のビットの数を数えます。
4.6.20(8)	R_OSPL_IsSetBitsCount1	1 になっているビットの数が 1 つかどうかを返します。

(20) OSPL の下位層に関する関数の一覧

章	関数名	概要
4.6.21(1)	R_DebugBreak	ブレークするときに OSPL からコールバックされる関数
4.6.21(2)	R_OSPL_OnIdleDefault	デフォルトのアイドル状態コールバック関数（OS レス専用）
4.6.21(3)	R_OSPL_Start_T_Lock	OSPL 内部から T-ロックを開始するときにコールバックされる関数
4.6.21(4)	R_OSPL_End_T_Lock	OSPL 内部から T-ロックを終了するときにコールバックされる関数
4.6.21(5)	R_OSPL_EVENT_GROUP_Create	イベント グループを生成します
4.6.21(6)	R_OSPL_EVENT_GROUP_Delete	イベント グループを削除します

(21) ドライバーの API に共通する関数仕様の一覧

章	関数名	概要
4.6.22(1)	R_DRIVER_Transfer	周辺機能が持つ非同期処理を同期的に処理します
4.6.22(2)	R_DRIVER_TransferStart	周辺機能が持つ非同期処理を起動します
4.6.22(3)	R_DRIVER_OnInterrupting	割込みを受信します
4.6.22(4)	R_DRIVER_OnInterrupted	割込み応答処理を行います
4.6.22(5)	R_DRIVER_GetAsyncStatus	割込みや非同期処理の状況を示す構造体へのポインターを取得します。
4.6.22(6)	R_DRIVER_Initialize	ドライバーを初期化して使える状態にします
4.6.22(7)	R_DRIVER_Finalize	ドライバーの終了処理をします
4.6.22(8)	R_DRIVER_LockChannel	チャンネルをロックします
4.6.22(9)	R_DRIVER_UnlockChannel	チャンネルをロック解除します

(22) ドライバーの下位層に共通する関数仕様の一覧

章	関数名	概要
4.6.23(1)	R_DRIVER_SetDefaultAsync	r_ospl_async_t 型の構造体のデフォルト値を設定します。
4.6.23(2)	R_DRIVER_I_LOCK_Replace	I-ロックを行うオブジェクトを、統合型ドライバーの I-ロック・オブジェクトに置き換えます。
4.6.23(3)	R_DRIVER_DisableInterrupt	I-ロックを開始するために、割込みを禁止します。
4.6.23(4)	R_DRIVER_EnableInterrupt	I-ロックを終了するために、割込みを許可します。

4.6.2　OSPL のバージョンに関する関数

(1)　R_OSPL_GetVersion

概　要	OSPL のバージョン番号を返します。
ヘッダー	r_ospl.h
宣　言	int32_t　R_OSPL_GetVersion();
説　明	R_OSPL_VERSION マクロと同じ値です。

引　数		
	なし	
リターン値	OSPL のバージョン番号	

(2)　R_OSPL_IsPreemption

概　要	プリエンプションに対応した RTOS であるかどうかを返します。
ヘッダー	r_ospl.h
宣　言	bool_t　R_OSPL_IsPreemption();
説　明	R_OSPL_IS_PREEMPTION マクロと同じ値です。

引　数		
	なし	
リターン値	プリエンプションに対応した RTOS であるかどうか	

4.6.3　r_ospl_thread_id_t 型 - スレッドに関する関数

(1)　R_OSPL_THREAD_DEF

概　要	スレッドのワーク領域や初期値を定義します。（OS レス専用）
ヘッダー	r_ospl.h
宣　言	#define　R_OSPL_THREAD_DEF(Name)
説　明	R_OSPL_IS_PREEMPTION = 0 に定義されているときのみ本関数は存在します。 ライブラリーの中で本マクロを使用しないでください。OSPL の内容が変更されたときに、ライブラリーの再コンパイルが必要になってしまいます。

引　数		
	Name	スレッド名。" " で囲まない。
リターン値	なし	

(2)　R_OSPL_THREAD

概　要	スレッドのワーク領域や初期値を返します。（OS レス専用）
ヘッダー	r_ospl.h
宣　言	r_ospl_thread_def_t*　R_OSPL_THREAD(Name);
説　明	R_OSPL_IS_PREEMPTION = 0 に定義されているときのみ本関数は存在します。 ライブラリーの中で本マクロを使用しないでください。OSPL の内容が変更されたときに、ライブラリーの再コンパイルが必要になってしまいます。

引　数		
	Name	R_OSPL_THREAD_DEF で定義したスレッド名。" " で囲まない
リターン値	スレッドのワーク領域や初期値	

(3)　R_OSPL_THREAD_Create

概　要	擬似スレッドを生成します。（OS レス専用）
ヘッダー	r_ospl.h
宣　言	errnum_t　R_OSPL_THREAD_Create(r_ospl_thread_def_t* ThreadDef, void* Argument,　r_ospl_thread_id_t* out_ThreadId);
説　明	R_OSPL_IS_PREEMPTION = 0 に定義されているときのみ本関数は存在します。 主に、イベントを複数生成するために使用します。 スレッドを削除するときは、R_OSPL_THREAD_Destroy 関数を呼び出してください。

例：

```
r_ospl_thread_id_t  g_ThreadA_Id;
R_OSPL_THREAD_DEF( g_ThreadA_Def );

void  main()
{
    errnum_t  e;

    e= R_OSPL_THREAD_Create( R_OSPL_THREAD( g_ThreadA_Def ),
        NULL, &g_ThreadA_Id );
        IF(e){goto fin;}
}
```

引　数	r_ospl_thread_def_t* ThreadDef	R_OSPL_THREAD 関数が返すスレッドのワーク領域や初期値
	void* Argument	R_OSPL_THREAD_GetArgument 関数に渡すパラメーター
	r_ospl_thread_id_t* out_ThreadId	（出力） スレッド ID
リターン値	エラーコード。エラーなし＝0	

(4)　R_OSPL_THREAD_Destroy

概　要	擬似スレッドを削除します。（OS レス専用）
ヘッダー	r_ospl.h
宣　言	errnum_t　R_OSPL_THREAD_Destroy(r_ospl_thread_id_t*　in_out_ThreadId);
説　明	R_OSPL_IS_PREEMPTION = 0 に定義されているときのみ本関数は存在します。 in_out_ThreadId 引数が指すスレッド ID が R_OSPL_THREAD_NULL のときは、何もせず、0（エラーなし）を返します。 注意： スレッドを終了する直前で、R_OSPL_FreeCurrentThreadError を呼び出してください。

引　数	r_ospl_thread_id_t* in_out_ThreadId	削除するスレッドの ID
リターン値	エラーコード。エラーなし＝0	

(5)　R_OSPL_THREAD_GetArgument

概　要	現在実行中のスレッドが生成されたときに渡されたパラメーターを返します。（OS レス専用）
ヘッダー	r_ospl.h
宣　言	void*　R_OSPL_THREAD_GetArgument();
説　明	R_OSPL_IS_PREEMPTION = 0 に定義されているときのみ本関数は存在します。 本関数を呼び出す前に、メッセージループから R_OSPL_THREAD_SetCurrentId 関数を呼び出してください。

引　数	なし	
リターン値	R_OSPL_THREAD_Create 関数の Argument 引数	

(6)　R_OSPL_THREAD_SetCurrentId

概　要	現在実行中のスレッド ID を設定します。（OS レス専用）
ヘッダー	r_ospl.h
宣　言	void　R_OSPL_THREAD_SetCurrentId(r_ospl_thread_id_t ThreadId);
説　明	R_OSPL_IS_PREEMPTION = 0 に定義されているときのみ本関数は存在します。

擬似マルチスレッドのメッセージループから呼び出します。

例：

```
R_OSPL_THREAD_SetCurrentId( ChildThreadId );
ChildThreadFunction();
R_OSPL_THREAD_SetCurrentId( MainThreadId );
```

引　数	r_ospl_thread_id_t ThreadId	現在実行中にするスレッド ID
リターン値	なし	

(7)　R_OSPL_THREAD_GetCurrentId

概　要	現在実行中のスレッド ID を返します。
ヘッダー	r_ospl.h
宣　言	r_ospl_thread_id_t　R_OSPL_THREAD_GetCurrentId();
説　明	R_OSPL_IS_PREEMPTION の定義内容に関わらず使用することができます。 OS レスでは、R_OSPL_THREAD_SetCurrentId 関数に指定した スレッド ID が返ります。 割込みコンテキストのときは、R_OSPL_THREAD_NULL が返ります。

引　数	なし	
リターン値	現在実行中のスレッド ID	

(8)　R_OSPL_THREAD_GetMainId

概　要	メインスレッドの ID を返します。（OS レス専用）
ヘッダー	r_ospl.h
宣　言	r_ospl_thread_id_t　R_OSPL_THREAD_GetMainId();
説　明	R_OSPL_IS_PREEMPTION = 0 に定義されているときのみ本関数は存在します。 main 関数の最初の方で、メインスレッドの ID を指定した R_OSPL_THREAD_SetCurrentId 関数を呼び出してください。

引　数	なし	
リターン値	メインスレッドの ID	

(9)　R_OSPL_THREAD_SetDelegate

概　要	スレッド付属のポインター型変数に値を設定します。（OS レス専用）
ヘッダー	r_ospl.h
宣　言	void　R_OSPL_THREAD_SetDelegate(r_ospl_thread_id_t　in_ThreadId,　void* in_Delegate);
説　明	スレッド付属のポインター型変数は、初期値の設定以外で、OSPL からアクセスすることはありません。

引　数	r_ospl_thread_id_t in_ThreadId	スレッド ID
	void*　in_Delegate	設定する値
リターン値	なし	

(10)　R_OSPL_THREAD_GetDelegate

概　要	スレッド付属のポインター型変数の値を返します。（OS レス専用）
ヘッダー	r_ospl.h
宣　言	void*　R_OSPL_THREAD_GetDelegate(r_ospl_thread_id_t　in_ThreadId);
説　明	

引　数	r_ospl_thread_id_t in_ThreadId	スレッド ID

リターン値	スレッド付属のポインター型変数の値、初期値は NULL

(11) R_OSPL_THREAD_SetOnWait

概　要	現在のスレッドについて、待つときの動きを設定します。
ヘッダー	r_ospl.h
宣　言	errnum_t　R_OSPL_THREAD_SetOnWait(r_ospl_wait_t OnWait);
説　明	ドライバーから呼び出さないでください。 アプリケーションから呼び出すことができます。 マルチスレッドのコードを擬似マルチスレッドのコードに変更するときは、本関数で R_OSPL_WAIT_PM_THREAD を設定してください。 さらに、OS ありで R_OSPL_INFINITE で待っているコードは、OS レスではタイムアウト 0 に変更してください。OS ありで タイムアウトを設定しているコードは、タイムアウトを変更する必要はありません。 R_OSPL_IS_PREEMPTION = 0 の場合： 　初期値は、R_OSPL_WAIT_POLLING です。 　割込みコンテキストから呼び出したときは、E_STATE エラーになります。 R_OSPL_IS_PREEMPTION = 1 の場合： 　アプリケーションの互換のために存在します。 　引数は無視され、0 を返します。 参考：R_OSPL_THREAD_GetIsWaiting
引　数	r_ospl_wait_t OnWait　待つときの動き
リターン値	エラーコード、正常=0

(12) R_OSPL_THREAD_GetOnWait

概　要	現在のスレッドについて、待つときの動きを取得します。
ヘッダー	r_ospl.h
宣　言	r_ospl_wait_t　R_OSPL_THREAD_GetOnWait();
説　明	R_OSPL_IS_PREEMPTION = 0 の場合： 　初期値は、R_OSPL_WAIT_POLLING です。 　割込みコンテキストから呼び出したときは、R_OSPL_WAIT_POLLING が返ります。 R_OSPL_IS_PREEMPTION = 1 の場合： 　アプリケーションの互換のために存在します。 　R_OSPL_WAIT_POLLING を返します。 　ただし、待つときにポーリングするわけではありません。 参考：R_OSPL_THREAD_SetOnWait
引　数	なし
リターン値	待つときの動き

(13) R_OSPL_THREAD_GetIsWaiting

概　要	現在のスレッドが、待っている状態かどうかを返します。
ヘッダー	r_ospl.h
宣　言	bool_t　R_OSPL_THREAD_GetIsWaiting();
説　明	R_OSPL_IS_PREEMPTION = 0 の場合： 　R_OSPL_THREAD_SetOnWait 関数を使って R_OSPL_WAIT_PM_THREAD を設定すると待ちに入る関数の一部は、待つ状態であっても待たないですぐに関数呼び出しから返り、R_OSPL_THREAD_GetIsWaiting 関数は、true を返すようになります。

タイムアウトが 0 のときは、R_OSPL_THREAD_SetOnWait 関数によってどんな設定がされていても、タイムアウトすると
R_OSPL_THREAD_GetIsWaiting 関数は、 true を返すようになります。

タイムアウトが R_OSPL_INFINITE のときは、待ちから抜けないため、R_OSPL_THREAD_GetIsWaiting 関数を呼び出すことができなくなってしまいます。擬似マルチタスクに移植するときは、R_OSPL_INFINITE を 0 に置き換えてください。

割込みコンテキストから呼び出したときは、false を返します。
デバッグ版では、内部で ASSERT_D に引っかかります。

R_OSPL_IS_PREEMPTION = 1 の場合：
アプリケーションの互換のために存在します。
false を返します。

サンプル

```
e= R_OSPL_THREAD_SetOnWait( R_OSPL_WAIT_PM_THREAD );
     IF(e){goto fin;}
e= R_OSPL_Delay( 100 ); IF(e){goto fin;}
    if ( R_OSPL_THREAD_GetIsWaiting() ) { e=0; goto fin; }
    IF(e){goto fin;}
```

引　数	なし	
リターン値	待っている状態かどうか	

(14) R_OSPL_THREAD_ExitWaiting

概　要	現在のスレッドが待っている状態なら、その状態を中止します。
ヘッダー	r_ospl.h
宣　言	void　R_OSPL_THREAD_ExitWaiting();
説　明	R_OSPL_IS_PREEMPTION = 0 の場合：

R_OSPL_THREAD_GetIsWaiting 関数から true を返されたスレッドは、同じ場所の待ちがある関数を再び呼び出さなければなりませんが、待ちを中断することで、別の処理や別の待ちができるようになります。 タイムアウトを 0 で待ちを始めた場合は、中断は不要です。

OSPL の API のどこかで、必要な中断がされていないことが検出できたら、E_STATE エラーになりますが、検出できないこともあります。 そのときは、タイムアウトの時間が正しくなくなります。

割込みコンテキストから呼び出したときは、何もしません。 デバッグ版では、内部で ASSERT_D に引っかかります。

R_OSPL_IS_PREEMPTION = 1 の場合：
アプリケーションの互換のために存在します。
本関数は、何もしません。

参考：R_OSPL_WAIT_OWNER_DEBUG

引　数		
リターン値	なし	

4.6.4　スレッド付属イベントに関する関数

(1)　R_OSPL_EVENT_Allocate

概　要　現在のスレッドが受信するスレッド付属イベントのフラグのビットを確保します。

ヘッダー　r_ospl.h

宣　言
```
errnum_t　R_OSPL_EVENT_Allocate( r_ospl_thread_id_t*　out_ThreadId,
r_ospl_thread_id_t　in_ThreadId,　bit_flags32_t*　out_SetFlag,
r_ospl_event_flags_t　in_SetFlag );
```

説　明　すでに存在するイベント フラグに対して、設定と待ちができるように確保します。確保したイベントは、現在のスレッドでのみ受信できるようになります。 送信は、本関数が出力するチェック ビットを持っていれば、任意のスレッドや割込みコンテキストから送信できます。確保したイベント フラグを使わなくなったら、R_OSPL_EVENT_Free 関数で返却してください。
ライブラリ（バイナリー）の同期関数の中でスレッド付属イベントを使うときは、エラー検出ができるように、本関数を必ず呼び出してください。

通常、ブロッキング関数の中、ノンブロッキング関数の外で確保と返却を行います。

R_OSPL_DETECT_BAD_EVENT= 0 の場合は、out_SetFlag 引数にチェック ビットは付きません。

異なる種類のイベントを順次待つときは、一度イベントを返却して、再度確保してください。チェック ビットには、イベントを確保するたびに値が変わり、イベントを返却すると値が無効になる、ID を含んでおり、返却した古い ID を持つ送信元から送信されたら、エラーになります。 ID は、グローバル変数のカウンターを使って、1～255 を繰り返します。

確保しようとするイベントが、すでにシグナル状態になっているときは、E_ACCESS_DENIED エラーになります。

in_SetFlag 引数に R_OSPL_UNUSED_FLAG を指定した場合、確保されていないイベント フラグのビットの中から動的に選んで 1 つのビットを確保します。 ただし、R_OSPL_TLS_EVENT_CODE = 1 でなければ、R_OSPL_UNUSED_FLAG を指定できません。 R_OSPL_UNUSED_FLAG を指定した場合、下位 4 ビット（R_OSPL_A_FLAG など）を除く 12 ビット（ビット 15～4）のどれかを確保します。

in_SetFlag 引数に R_OSPL_EVENT_ALL_BITS を指定した場合、すべてのビットを確保します。 現在のスレッドは、フラグの重複のチェックができなくなりますが、フラグごとに確保する必要がなくなります。
すでに 1 つでもビットが確保されているときは、E_ACCESS_DENIED エラーになります。 *out_SetFlag にチェック ビットは付かなくなります。 ただし、R_OSPL_ALL_EVENT_ALLOCATE = 1 でなければ、R_OSPL_EVENT_ALL_BITS を指定できません。

引　数

r_ospl_thread_id_t* out_ThreadId	（出力） イベントの通知先のスレッドの ID ＝ 現在のスレッド
r_ospl_thread_id_t in_ThreadId	通知先のスレッドの ID
bit_flags32_t* out_SetFlag	（出力） 確保したビット + チェック ビット
r_ospl_event_flags_t in_SetFlag	確保するスレッド付属イベントのビット、または、R_OSPL_UNUSED_FLAG

リターン値	エラーコード、正常=0

(2)　R_OSPL_EVENT_Set

概　要　　１つまたはいくつかのビットを 1 に設定します。

ヘッダー　　r_ospl.h

宣　言　　void　R_OSPL_EVENT_Set(r_ospl_thread_id_t　in_ThreadId, r_ospl_event_flags_t　in_SetFlags);

説　明　　OS レスでは、内部に全割込み禁止区間があります。
OS ありのときは、R_OSPL_EVENT_Wait 関数で待っているスレッドがすぐに起床する可能性があります。
in_ThreadId = R_OSPL_THREAD_NULL のときは、何もしません。

R_OSPL_DETECT_BAD_EVENT が 1 のとき、in_SetFlags 引数にチェック ビットが付いていなければ、R_OSPL_RaiseUnrecoverable(E_OTHERS) を呼び出します。 チェック ビットは、R_OSPL_EVENT_Allocate で取得できます。

引　数	r_ospl_thread_id_t in_ThreadId	対象となるイベントを持つスレッドの ID
	r_ospl_event_flags_t in_SetFlags	設定するビットが 1 になったビットフラグ値
リターン値	なし	

(3)　R_OSPL_EVENT_Clear

概　要　　１つまたはいくつかのビットを 0 に設定します。

ヘッダー　　r_ospl.h

宣　言　　void　R_OSPL_EVENT_Clear(r_ospl_thread_id_t ThreadId, r_ospl_event_flags_t ClearFlags1);

説　明　　R_OSPL_EVENT_Wait 関数を使用したときは、本関数を呼び出す必要はありません。
イベントのすべてのビットフラグをクリアするときは、ClearFlags1 引数に R_OSPL_EVENT_ALL_BITS (=0x0000FFFF) を指定してください。
R_OSPL_EVENT_Set を呼び出して他のスレッドに通知したとき、呼び出し側のスレッドから R_OSPL_EVENT_Clear は呼び出さないでください。
OS レスでは、内部に全割込み禁止区間があります。
ThreadId = R_OSPL_THREAD_NULL のとき、または、ClearFlags1 引数が 0 のときは、何もしません。

ClearFlags1 引数に指定するスレッド付属イベントは、R_OSPL_EVENT_Allocate 関数で確保されている必要はありません。

引　数	r_ospl_thread_id_t ThreadId	対象となるイベントを持つスレッドの ID
	r_ospl_event_flags_t ClearFlags1	クリアするビットが 1 になったビットフラグ値
リターン値	なし	

(4)　R_OSPL_EVENT_Wait

概　要　　16 ビットのフラグが立つまで待ち、受信したフラグをクリアします。または、シグナル状態になるまで待ち、受信したら非シグナル状態に戻します。

ヘッダー　　r_ospl.h

宣　言　　errnum_t　R_OSPL_EVENT_Wait(r_ospl_event_flags_t　in_WaitgingFlags, bit_flags32_t* out_GotFlags, uint32_t　in_Timeout_msec);

説　明　　R_OSPL_EVENT_Set で立たせたイベント フラグが、指定したパターンになるまで、本関数の内部で待ちます。受信したフラグはクリアされます。

OS ありのときは、待っている間に他のスレッドが動くことができます。OS なしのときは、待っている間に他のスレッドが動くことができません。 in_Timeout_msec 引数に 0 を指定して、擬似マルチスレッドを構成してください。

非同期処理が完了したら、非同期処理の返り値 r_ospl_async_t::ReturnValue をチェックしてください。

*out_GotFlags の出力値には、WaitingFlags に指定していないフラグに関する値は含まれていません。

例：

```
    errnum_t       e;
    errnum_t       ee;
    r_ospl_async_t  async;

    async.A_EventValue = 0;  /* For fin block */

    async.Flags = R_F_OSPL_A_Thread | R_F_OSPL_A_EventValue;
    e= R_OSPL_EVENT_Allocate(
       &async.A_Thread,  R_OSPL_THREAD_GetCurrentId(),
       &async.A_EventValue,  R_OSPL_UNUSED_FLAG );
       IF(e){goto fin;}

    e= R_DRIVER_TransferStart( &async ); IF(e){goto fin;}

    for (;;) {
       bit_flags32_t  got_flags;

       async.ReturnValue = 0;
       R_OSPL_EVENT_Set( async.A_Thread,
          async.A_EventValue );
           /* 割込みコンテキストか別のスレッドから呼ぶ */

       e= R_OSPL_EVENT_Wait( async.A_EventValue, &got_flags,
          R_OSPL_INFINITE );
          IF(e){goto fin;}

       if ( IS_BIT_SET( got_flags, async.A_EventValue ) ) {
          e = async.ReturnValue; IF(e){goto fin;}
            :
          /* ここに応答処理を書く */
       }
       if ( IS_BIT_SET( got_flags, async_B.A_EventValue ) ) {
          e = async_B.ReturnValue; IF(e){goto fin;}
            :
          /* ここに応答処理を書く */
       }
    }

    e=0;
fin:
    ee= R_OSPL_EVENT_Free( &async.A_Thread,
```

```
        &async.A_EventValue );
        e= R_OSPL_MergeErrNum( e, ee );
    ee= R_OSPL_EVENT_Free( &async_B.A_Thread,
        &async_B.A_EventValue );
        e= R_OSPL_MergeErrNum( e, ee );
    return  e;
}
```

OSPL は、非同期処理の完了の通知方法としてイベント通知を推奨しています。コールバック関数の場合、応答するコンテキスト（スレッド、割込み）が不明、または、仕様依存になり、排他制御の問題が発生しやすくなるからです。また、イベント通知は、上記のようにモジュール間の連携に関するデータを、１つのスレッド関数のローカル変数にまとめることができ、多くの場合、排他制御のコードを書かずに済みます。割込みコンテキストの処理を減らせるため、リアルタイム性が高いです。 システム全体を見て、優先する処理のみコールバック関数に移動します。

I-スレッドが本関数を呼び出して、割込み応答処理のタイミングの通知を受信したら、R_DRIVER_OnInterrupted 関数を呼び出してください。
R_DRIVER_GetAsyncStatus 関数から取得できる r_ospl_async_state_t 型の状態が、R_OSPL_UNINITIALIZED のときは、メッセージループを終了することができます。

OS レスでは、内部に全割込み禁止区間がありますが、内部で待っているときは許可になります。全割込み禁止状態で呼び出すと、エラーになる可能性があります。

OS レスの場合は、待っている間に CPU 使用率を計測するための内部関数を何度も呼び出します。

in_WaitgingFlags 引数について
待つ 16 ビットのフラグの代わりに、以下の定数を指定できます。

シンボル	値	内容
R_OSPL_ANY_FLAG	0x00000000	すべてのフラグのうちどれか

タイムアウトと、in_Timeout_msec 引数について

in_Timeout_msec 引数には、タイムアウトの時間の他に以下のシンボルを指定できます。

シンボル	値	内容
R_OSPL_INFINITE	0xFFFFFFFF	受信するまで永遠に待ち続けます。値は環境によります。
R_OSPL_INFINITE_PSEUDO	0xFFFFFFFF または 0	擬似マルチスレッドに対応した R_OSPL_INFINITE。R_OSPL_IS_PREEMPTION が 1 なら 受信するまで永遠に待ち続けます。 R_OSPL_IS_PREEMPTION が 0 なら待つ状態になっても関数の内部で待つことはしません（in_Timeout_msec = 0 と同じ）。その代わり、R_OSPL_THREAD_GetIsWaiting

の返り値によって、別のスレッドの処理を実行するように分岐してください。

タイムアウトしたときは、*out_GotFlags の R_OSPL_TIMEOUT のビットが 1 になり、16 ビットのフラグの部分のビットは、実際のフラグのビットの値に関わらず、0 が得られます。

シンボル	値	内容
R_OSPL_TIMEOUT	0x40000000	タイムアウトした

OS なしのときは、待っている間に他のスレッドが動くことができないため、in_Timeout_msec 引数に 0 を指定して、擬似マルチスレッドを構成してください。
タイムアウトのときは、*out_GotFlags の R_OSPL_TIMEOUT のビットが 1 になり、16 ビットのフラグの部分のビットは、実際のフラグのビットの値に関わらず、0 が得られます。返り値は、in_Timeout_msec = 0 なら 0、それ以外なら E_TIME_OUT になります。

R_OSPL_IS_PREEMPTION = 0 に定義されているときは、R_OSPL_THREAD_SetOnWait 関数に、R_OSPL_WAIT_PM_THREAD を設定すると、待っている状態でも、R_OSPL_EVENT_Wait 関数からすぐに返ります。 このときの R_OSPL_EVENT_Wait 関数の返り値は、0 で、*out_GotFlags は 0x00000000 です。

in_WaitgingFlags 引数について：

OR 条件で待つには、in_WaitgingFlags 引数に、0、または、R_OSPL_ANY_FLAG を指定します。ビットを指定できないため、待っていないフラグが立っていた場合の対処も必要になります。

AND 条件で待つには、in_WaitgingFlags 引数が、「0 または R_OSPL_ANY_FLAG」以外を指定します。

AND と OR の複合条件で待つには、in_WaitgingFlags 引数に、0、または、R_OSPL_ANY_FLAG を指定します。 関数から返ったら、r_ospl_flag32_t 型の自動変数（ローカル変数）にフラグを転送し、*out_GotFlags で取得できる bit_flags_fast32_t 型の値をチェックしてください。 まだ条件を満たさないときは、再度 R_OSPL_EVENT_Wait 関数を呼び出してください。

引　数	r_ospl_event_flags_t in_WaitgingFlags	1 になるまで待つフラグ（AND 条件）、または、R_OSPL_ANY_FLAG
	bit_flags32_t* out_GotFlags	NULL 可。（出力） 16 ビットのフラグ、または、R_OSPL_TIMEOUT
	uint32_t in_Timeout_msec	タイムアウト（ミリ秒）、または、R_OSPL_INFINITE
リターン値	エラーコード。エラーなし＝0	

(5)　R_OSPL_EVENT_GetStatus

概　要	イベントの状態を取得します。
ヘッダー	r_ospl.h

宣　言　errnum_t　R_OSPL_EVENT_GetStatus(r_ospl_thread_id_t　in_ThreadId, r_ospl_event_status_t** out_Status);

説　明　R_OSPL_DETECT_BAD_EVENT = 1 のときのみ使えます。
取得した状態は、本関数を呼び出した瞬間の情報です。 他のスレッドが動作することで変化します。

サンプル：
```
r_ospl_event_status_t*  status;
e= R_OSPL_EVENT_GetStatus( R_OSPL_THREAD_GetCurrentId(),
&status ); IF(e){goto fin;}
```

引　数

r_ospl_thread_id_t in_ThreadId	対象となるイベントを持つスレッドの ID
r_ospl_event_status_t** out_Status	（出力） イベント内部の構造体へのポインター 参考：(4.5.5)

リターン値　エラーコード、正常=0

(6)　R_OSPL_EVENT_Free

概　要　R_OSPL_EVENT_Allocate 関数で確保したイベントのフラグのビットを返却します。

ヘッダー　r_ospl.h

宣　言　errnum_t　R_OSPL_EVENT_Free(r_ospl_thread_id_t*　in_out_ThreadId, r_ospl_event_flags_t*　in_out_SetFlag);

説　明　*in_out_ThreadId 引数と *in_out_SetFlag 引数は、R_OSPL_EVENT_Allocate 関数の in_ThreadId 引数と in_SetFlag 引数に指定した値に戻ります。
返却したイベントは、セットと受信ができないようになります。
現在のスレッドが確保したイベントのみ返却できます。
ライブラリ（バイナリー）の同期関数の中でスレッド付属イベントを使うときは、フラグの重複に関してエラー検出ができるように、本関数を必ず呼び出してください。

*in_out_SetFlag 引数に 0 を指定したときは、何もしません。
返却するフラグは、本関数の内部でクリアされます。

引　数

r_ospl_thread_id_t* in_out_ThreadId	（入出力） 通知先のスレッドの ID
r_ospl_event_flags_t* in_out_SetFlag	（入出力） 返却する通知先（＋チェック ビット）

リターン値　エラーコード、正常=0

4.6.5　r_ospl_flag32_t 型 - フラグに関する関数

(1)　R_OSPL_FLAG32_InitConst

概　要　32 ビットのフラグをすべての 0 にクリアします

ヘッダー　r_ospl.h

宣　言　void　R_OSPL_FLAG32_InitConst(r_ospl_flag32_t* self);

説　明　本関数は、スレッドセーフではありません。 必要なら別途排他制御を行ってください。
下記の処理を行います。
```
volatile bit_flags32_t  self->flags;
self->flags = 0;
```

引　数

r_ospl_flag32_t*　self	32 ビットのフラグ

リターン値　なし

(2)　R_OSPL_FLAG32_Set

概　要　　１つまたはいくつかのビットを 1 に設定します。

ヘッダー　　r_ospl.h

宣　言　　void　R_OSPL_FLAG32_Set(r_ospl_flag32_t* self, bit_flags32_t SetFlags);

説　明　　本関数は、スレッドセーフではありません。必要なら別途排他制御を行ってください。

下記の処理を行います。

|= の Read Modify Write により、アトミックではありません。

```
volatile bit_flags32_t  self->Flags;
bit_flags32_t           SetFlags;
self->Flags |= SetFlags;
```

引　数		
	r_ospl_flag32_t*　self	32 ビットのフラグ
	uint32_t　SetFlags	設定するビットが 1 になったビットフラグ値
リターン値	なし	

(3)　R_OSPL_FLAG32_Clear

概　要　　１つまたはいくつかのビットを 0 に設定します。

ヘッダー　　r_ospl.h

宣　言　　void　R_OSPL_FLAG32_Clear(r_ospl_flag32_t* self, bit_flags32_t ClearFlags1);

説　明　　本関数は、スレッドセーフではありません。必要なら別途排他制御を行ってください。

下記の処理を行います。

すべてクリアするときは、R_OSPL_FLAG32_ALL_BITS を指定してください。

&= の Read Modify Write により、アトミックではありません。

```
volatile bit_flags32_t  self->Flags;
bit_flags32_t           ClearFlags1;

self->Flags &= ~ClearFlags1;
```

引　数		
	r_ospl_flag32_t*　self	32 ビットのフラグ
	uint32_t　ClearFlags1	クリアするビットが 1 になったビットフラグ値
リターン値	なし	

(4)　R_OSPL_FLAG32_Get

概　要　　32 ビットのフラグの値を取得します。

ヘッダー　　r_ospl.h

宣　言　　bit_flags32_t　R_OSPL_FLAG32_Get(r_ospl_flag32_t* self);

説　明　　本関数は、スレッドセーフではありません。必要なら別途排他制御を行ってください。

イベントを受信するときは、本関数の代わりに R_OSPL_FLAG32_GetAndClear 関数を呼び出すか、本関数で取得したフラグの値を「Not 演算した」値を R_OSPL_FLAG32_Clear 関数に渡してください。

下記の処理を行います。

```
volatile bit_flags32_t  self->Flags;
bit_flags32_t           return_flags;

return_flags = self->Flags;

return  return_flags;
```

引　数		
	r_ospl_flag32_t*　self	32 ビットのフラグ
リターン値	フラグの値	

(5) R_OSPL_FLAG32_GetAndClear

概　要　フラグの値を返し、すべてのビットを 0 にクリアします。

ヘッダー　r_ospl.h

宣　言　bit_flags32_t　R_OSPL_FLAG32_GetAndClear(r_ospl_flag32_t* self);

説　明　本関数は、スレッドセーフではありません。必要なら別途排他制御を行ってください。
下記の処理を行います。
0 にクリアする直前で Set されることがあるため、アトミックではありません。

```
volatile bit_flags32_t  self->Flags;
bit_flags32_t          return_flags;

return_flags = self->Flags;
self->Flags = 0;

return  return_flags;
```

引　数	r_ospl_flag32_t*　self	32 ビットのフラグ
リターン値	フラグの値	

4.6.6 bit_flags_fast32_t 型 - ビットフラグに関する関数

(1) IS_BIT_SET

概　要　指定したビットが 1 かどうかを評価します。

ヘッダー　r_ospl.h

宣　言　bool_t　IS_BIT_SET(bit_flags_fast32_t Variable, bit_flags_fast32_t ConstValue);

説　明　ConstValue 引数の中にある、1 になっているビットが 1 個以外のとき、返り値は不定です。

引　数	bit_flags_fast32_t Variable	調べる対象となるビットフラグの値
	bit_flags_fast32_t ConstValue	調べるビットが 1 になったビットフラグの値
リターン値	指定したビットが 1 かどうか	

(2) IS_ANY_BITS_SET

概　要　指定した複数のビットのどれかが 1 かどうかを評価します。

ヘッダー　r_ospl.h

宣　言　bool_t　IS_ANY_BITS_SET(bit_flags_fast32_t Variable, bit_flags_fast32_t ConstValue);

説　明　ConstValue 引数の中にある、1 になっているビットが 0 個のとき、返り値は不定です。

引　数	bit_flags_fast32_t Variable	調べる対象となるビットフラグの値
	bit_flags_fast32_t ConstValue	調べるビットが 1 になったビットフラグの値
リターン値	指定した複数のビットのどれかが 1 かどうか	

(3) IS_ALL_BITS_SET

概　要　指定した複数のビットのすべてが 1 かどうかを評価します。

ヘッダー　r_ospl.h

宣　言　bool_t　IS_ALL_BITS_SET(bit_flags_fast32_t Variable, bit_flags_fast32_t ConstValue);

説　明　ConstValue 引数の中にある、1 になっているビットが 0 個のとき、返り値は不定です。

引　数	bit_flags_fast32_t Variable	調べる対象となるビットフラグの値
	bit_flags_fast32_t ConstValue	調べるビットが 1 になったビットフラグの値
リターン値	指定した複数のビットのすべてが 1 かどうか	

(4)　IS_BIT_NOT_SET

概　要　　指定したビットが 0 かどうかを評価します。
ヘッダー　　r_ospl.h
宣　言　　bool_t　IS_BIT_NOT_SET(bit_flags_fast32_t Variable, bit_flags_fast32_t ConstValue);
説　明　　ConstValue 引数の中にある、1 になっているビットが 1 個以外のとき、返り値は不定です。

引　数	bit_flags_fast32_t Variable	調べる対象となるビットフラグの値
	bit_flags_fast32_t ConstValue	調べるビットが 1 になったビットフラグの値
リターン値	指定したビットが 0 かどうか	

(5)　IS_ANY_BITS_NOT_SET

概　要　　指定した複数のビットのどれかが 0 かどうかを評価します。
ヘッダー　　r_ospl.h
宣　言　　bool_t　IS_ANY_BITS_NOT_SET(bit_flags_fast32_t Variable, bit_flags_fast32_t ConstValue);
説　明　　ConstValue 引数の中にある、1 になっているビットが 0 個のとき、返り値は不定です。

引　数	bit_flags_fast32_t Variable	調べる対象となるビットフラグの値
	bit_flags_fast32_t ConstValue	調べるビットが 1 になったビットフラグの値
リターン値	指定した複数のビットのどれかが 0 かどうか	

(6)　IS_ALL_BITS_NOT_SET

概　要　　指定した複数のビットのすべてが 0 かどうかを評価します。
ヘッダー　　r_ospl.h
宣　言　　bool_t　IS_ALL_BITS_NOT_SET(bit_flags_fast32_t Variable, bit_flags_fast32_t ConstValue);
説　明　　ConstValue 引数の中にある、1 になっているビットが 0 個のとき、返り値は不定です。

引　数	bit_flags_fast32_t Variable	調べる対象となるビットフラグの値
	bit_flags_fast32_t ConstValue	調べるビットが 1 になったビットフラグの値
リターン値	指定した複数のビットのすべてが 0 かどうか	

4.6.7　r_ospl_async_t 型 - 通知に関する関数

(1)　R_OSPL_ASYNC_SetDefaultPreset

概　要　　r_ospl_async_t 型構造体にプリセット フラグがあれば、各メンバー変数にプリセットの値を設定します。
ヘッダー　　r_ospl.h
宣　言　　void　R_OSPL_ASYNC_SetDefaultPreset(r_ospl_async_t* in_out_Async);

説　明　　R_DRIVER_SetDefaultAsync 関数から呼び出してください。
以下のプリセット フラグから、メンバー変数を設定します。
プリセット フラグが設定されていなければ、何もしません。
メンバー変数に対応するフラグが設定されていたら、そのメンバー変数にプリセットの値は設定されません。
プリセット フラグが設定されていたら、メンバー変数に対応するフラグをセットし、プリセット フラグはクリアします。

プリセット フラグ	内容
R_F_OSPL_AsynchronousPreset	現在のスレッドへの新規非同期通知。 また、内部で R_OSPL_EVENT_Allocate を呼び出します。 R_F_OSPL_AsynchronousPreset を指定したモジュールは、in_out_Async->A_EventValue に出力されたイベント対して R_OSPL_EVENT_Free を呼び出してください。 エラーが発生したら、R_OSPL_RaiseUnrecoverable を呼び出します。 以下のように設定します。 in_out_Async->A_Thread は、現在のスレッド。 in_out_Async->A_EventValue は、空いているビット（R_OSPL_UNUSED_FLAG で得られるビット）＋チェックビット。 他は、設定しません。

引　数	r_ospl_async_t* in_out_Async	（入出力） r_ospl_async_t 型構造体
リターン値	なし	

4.6.8　r_ospl_queue_id_t 型 - キューに関する関数

(1)　R_OSPL_QUEUE_DEF

概　要　　キューの属性とワーク領域を定義します。
ヘッダー　　r_ospl.h
宣　言　　#define　R_OSPL_QUEUE_DEF(Name, MaxCount, Type)
説　明　　ライブラリーの中で本マクロを使用しないでください。
OSPL の内容や OS を変更したときに、ライブラリーの再コンパイルが必要になってしまいます。

引　数	Name	定義する変数名。 "" で囲まない
	MaxCount	キューに入れることができる最大の要素数
	Type	要素の型
リターン値	なし	

(2)　R_OSPL_QUEUE

概　要　　キューの属性の初期値とワーク領域を返します。
ヘッダー　　r_ospl.h
宣　言　　r_ospl_queue_def_t*　R_OSPL_QUEUE(Name);
説　明　　ライブラリーの中で本マクロを使用しないでください。
OSPL の内容や OS を変更したときに、ライブラリーの再コンパイルが必要になってしまいます。

引　数	Name	R_OSPL_QUEUE_DEF で定義したキューの名前。" " で囲まない
リターン値	キューの属性の初期値とワーク領域	

(3)　R_OSPL_QUEUE_Create

概　要　キューを初期化します。

ヘッダー　r_ospl.h

宣　言　errnum_t　R_OSPL_QUEUE_Create(r_ospl_queue_id_t* out_self, r_ospl_queue_def_t*　in_QueueDefine);

説　明　ライブラリーの中から本関数を呼び出さないでください。 本関数はドライバーの移植層から呼び出し、生成したキューをドライバーに渡してください。
OSPL では、（CMSIS に合わせて）終了処理関数が無いので、同じ in_QueueDefine に対して Create は 1 回しかできません。 OS によっては、そのような制限はありません。
r_ospl_queue_id_t 型変数のアドレスを out_self に指定してください。
キューの内部変数は、in_QueueDefine に格納されます。

引　数	r_ospl_queue_id_t* out_self	（出力） キュー
	r_ospl_queue_def_t* in_QueueDefine	R_OSPL_QUEUE 関数が返すキューの属性とワーク領域
リターン値	エラーコード、正常=0	

(4)　R_OSPL_QUEUE_GetStatus

概　要　キューの状態（使用数など）を取得します。

ヘッダー　r_ospl.h

宣　言　errnum_t　R_OSPL_QUEUE_GetStatus(r_ospl_queue_id_t self,　const r_ospl_queue_status_t** out_Status);

説　明　取得した状態は、本関数を呼び出した瞬間の情報です。 他のスレッドが動作することで変化します。
out_Status 引数のポインター型については、R_DRIVER_GetAsyncStatus 関数を参照。

引　数	r_ospl_queue_id_t self	キュー
	r_ospl_queue_status_t** out_Status	（出力） キューの状態の構造体へのポインター
リターン値	エラーコード、正常=0	

(5)　R_OSPL_QUEUE_Allocate

概　要　キューからメモリー領域を確保します。待ち可能。

ヘッダー　r_ospl.h

宣　言　errnum_t　R_OSPL_QUEUE_Allocate(r_ospl_queue_id_t self,　void* out_Address, uint32_t　in_Timeout_msec);

説　明　割込みコンテキストでは、in_Timeout_msec = 0 以外を指定するとエラーになります。
タイムアウトになったときは、in_Timeout_msec = 0 を指定した場合はエラーにはならず、*out_Address = NULL になります。 in_Timeout_msec に 0 以外を指定した場合は E_TIME_OUT エラーになります。

OS レス版では、擬似マルチスレッドに対応しています。
参照：R_OSPL_THREAD_GetIsWaiting 関数。

引　数	r_ospl_queue_id_t self	キュー

	void* out_Address	（入力）確保したメモリー領域のアドレスを格納するポインターのアドレス。（出力） 確保したメモリー領域のアドレス
	uint32_t in_Timeout_msec	タイムアウト（ミリ秒）、または、R_OSPL_INFINITE
リターン値	エラーコード、正常=0	

(6) R_OSPL_QUEUE_Put

概　要　キューに入れます。

ヘッダー　r_ospl.h

宣　言　errnum_t　R_OSPL_QUEUE_Put(r_ospl_queue_id_t self,　void*　in_Address);

説　明　R_OSPL_QUEUE_Allocate 関数を呼び出してから、R_OSPL_QUEUE_Put するまでの間に、他のスレッドがキューに入れたり、キューから取り出したりしても、正しく動作します。
キューに入れたメッセージは、R_OSPL_QUEUE_Get 関数を呼び出すスレッドで受信します。

引　数	r_ospl_queue_id_t self	キュー
	void*　in_Address	R_OSPL_QUEUE_Allocate 関数で確保したメモリー領域のアドレス
リターン値	エラーコード、正常=0	

(7) R_OSPL_QUEUE_Get

概　要　キューから取り出します。待ち可能。

ヘッダー　r_ospl.h

宣　言　errnum_t　R_OSPL_QUEUE_Get(r_ospl_queue_id_t self,　void* out_Address, uint32_t　in_Timeout_msec);

説　明　要素へのアクセスが終了したら R_OSPL_QUEUE_Free 関数を呼び出してください。 R_OSPL_QUEUE_Free 関数を呼び出した後は、要素のメモリー領域にアクセスしないでください。

割込みコンテキストでは、in_Timeout_msec = 0 以外を指定すると E_NOT_THREAD エラーになります。割込みコンテキストでは、キューにデータが入るまで待つことはできません。

タイムアウトになったときは、in_Timeout_msec = 0 を指定した場合はエラーにはならず、*out_Address = NULL になります。 in_Timeout_msec != 0 を指定した場合は E_TIME_OUT エラーになります。

スレッドがキューの待ちに入ってしまって別のイベントを受け取れなくなることを防ぐには、in_Timeout_msec = 0 を指定するだけでなく、イベントを併用して、次の順序で関数を呼び出してください。

送信側：
1. R_OSPL_QUEUE_Allocate
2. R_OSPL_QUEUE_Put
3. R_OSPL_EVENT_Set

受信側：
1. R_OSPL_EVENT_Wait
2. R_OSPL_QUEUE_Get
3. R_OSPL_QUEUE_Free

OS レス版では、擬似マルチスレッドに対応しています。
参照：R_OSPL_THREAD_GetIsWaiting 関数。

引　数	r_ospl_queue_id_t self	キュー
	void* out_Address	（入力）キューから取り出したメモリー領域のアドレスを格納するポインターのアドレス。（出力） キューから取り出したメモリー領域のアドレス
	uint32_t in_Timeout_msec	タイムアウト（ミリ秒）、または、R_OSPL_INFINITE
リターン値	エラーコード、正常=0	

(8)　R_OSPL_QUEUE_Free

概　要　メモリー領域をキューに返却します。
ヘッダー　r_ospl.h
宣　言　errnum_t　R_OSPL_QUEUE_Free(r_ospl_queue_id_t self,　void*　in_Address);
説　明　R_OSPL_QUEUE_Get 関数を呼び出してから、R_OSPL_QUEUE_Free するまでの間に、他のスレッドがキューに入れたり、キューから取り出したりしても、正しく動作します。

本関数を呼び出すと、キューの要素が 1 つ空きます。

引　数	r_ospl_queue_id_t self	キュー
	void* in_Address	R_OSPL_QUEUE_Get 関数で取り出したメモリー領域のアドレス
リターン値	エラーコード、正常=0	

(9)　R_OSPL_GetQueueAsSingletonLock

概　要　シングルトンの生成と削除をするときに T-ロックするオブジェクトを返します。
ヘッダー　r_ospl.h
宣　言　r_ospl_queue_id_t　R_OSPL_GetQueueAsSingletonLock(void);
説　明　返されるロック オブジェクト（キュー）はシステム全体で 1 つです。 本関数の初回の呼び出しのときだけ、関数の内部全体で割込みを禁止し、ロック オブジェクトを生成します。 返り値のキューに格納できる要素数の最大を 1 に設定することで、排他制御をしています。 返り値のキューに対して R_OSPL_QUEUE_Put や R_OSPL_QUEUE_Get は呼び出さないでください。 OSPL は T-ロック専用の API 関数を用意していません。

シングルトン オブジェクト（クラス オブジェクト）の初期化または終了処理は、その要求が複数のスレッドから同時に要求されても正しく動作するために、T-ロックが必要です。

サンプル：

```
errnum_t  CreateSingletonObject()
{
    r_ospl_queue_id_t  signleton_lock_ID;
    void*              signleton_lock = NULL;

    signleton_lock_ID = R_OSPL_GetQueueAsSingletonLock();
    e= R_OSPL_QUEUE_Allocate( signleton_lock_ID,
           &signleton_lock,  R_OSPL_INFINITE );
       IF(e){goto fin;}  /* Start of T-Lock */
    if ( gs_SingletonObject == NULL ) {
```

```
            gs_SingletonObject = ...;

        }
        e=0;
    fin:
        if ( signleton_lock != NULL ) {
            ee= R_OSPL_QUEUE_Free( signleton_lock_ID,
                    signleton_lock );  /* End of T-Lock */
                e= R_OSPL_MergeErrNum( e, ee );
        }
        return  e;
    }
```

引　数	なし	
リターン値	ロック オブジェクトとして使う、キュー	

4.6.9　全割込み禁止区間に関する関数

(1)　R_OSPL_EnableAllInterrupt

概　要	すべての割込みを禁止していたことを解除します。
ヘッダー	r_ospl.h
宣　言	void　R_OSPL_EnableAllInterrupt();
説　明	ドライバー使用者は、本関数を呼び出さないでください。 全割込み禁止区間の終了時に呼び出します。 呼び出し元ですでに禁止していたときは、解除しないでください。 NMI は対象外です。

引　数	なし	
リターン値	なし	

(2)　R_OSPL_DisableAllInterrupt

概　要	すべての割込みを禁止します。
ヘッダー	r_ospl.h
宣　言	bool_t　R_OSPL_DisableAllInterrupt();
説　明	ドライバー使用者は、本関数を呼び出さないでください。 全割込み禁止区間の開始時に呼び出します。 NMI は対象外です。

例：

```
 void  Func()
 {
    bool_t  was_all_enabled = false;

    was_all_enabled = R_OSPL_DisableAllInterrupt();

    /* 全割込み禁止区間 */

    if ( was_all_enabled )
       { R_OSPL_EnableAllInterrupt(); }
 }
```

引　数	なし	
リターン値	今まで許可されていたかどうか	

(3) R_OSPL_GetIsAllInterruptEnabled

概　要	すべての割込みが許可されているかどうかを返します。
ヘッダー	r_ospl.h
宣　言	bool_t　R_OSPL_GetIsAllInterruptEnabled();
説　明	-

引　数		
	なし	
リターン値	すべての割込みが許可されているかどうか	

4.6.10　割込み処理に関する関数

(1) R_BSP_InterruptWrite

概　要	割込みハンドラーを登録します
ヘッダー	platform.h or mcu_interrupts.h
宣　言	bsp_int_err_t　R_BSP_InterruptWrite(bsp_int_src_t　in_IRQ_Num, bsp_int_cb_t in_Callback);
説　明	-

引　数		
	bsp_int_src_t　in_IRQ_Num	割込み番号
	bsp_int_cb_t　in_Callback	割込みハンドラーとする関数、または、FIT_NO_FUNC
リターン値	エラーコード、正常=BSP_INT_SUCCESS	

(2) R_BSP_InterruptRead

概　要	登録されている割込みハンドラーを返します
ヘッダー	platform.h or mcu_interrupts.h
宣　言	bsp_int_err_t　R_BSP_InterruptRead(bsp_int_src_t　in_IRQ_Num, bsp_int_cb_t* out_Callback);
説　明	-

引　数		
	bsp_int_src_t　in_IRQ_Num	割込み番号
	bsp_int_cb_t*　out_Callback	割込みハンドラーとしている関数
リターン値	エラーコード、正常=BSP_INT_SUCCESS	

(3) R_BSP_InterruptControl

概　要	割込みを制御します
ヘッダー	platform.h or mcu_interrupts.h
宣　言	bsp_int_err_t　R_BSP_InterruptControl(bsp_int_src_t　in_IRQ_Num, bsp_int_cmd_t　in_Command, void*　in_Parameter);
説　明	参照：(4.4.15) bsp_int_cmd_t

引　数		
	bsp_int_src_t　in_IRQ_Num	割込み番号
	bsp_int_cmd_t in_Command	制御コマンド
	void*　in_Parameter	参照：(4.4.15) bsp_int_cmd_t
リターン値	エラーコード、正常=BSP_INT_SUCCESS	

(4) R_OSPL_SetInterruptPriority

概　要	指定した割込み線に対する優先度を設定します
ヘッダー	r_ospl.h
宣　言	errnum_t　R_OSPL_SetInterruptPriority(bsp_int_src_t　in_IRQ_Num, int_fast32_t in_Priority);
説　明	割込みコンテキストでは、指定した優先度以下の割込みは禁止状態になります。多重割込みをサポートしないときは、優先度を最大に設定してください。

引　数	bsp_int_src_t　in_IRQ_Num	割込み番号
	int_fast32_t　in_Priority	優先度、値が小さいほど優先
リターン値	エラーコード、正常=0	

(5)　R_OSPL_END_OF_INTERRUPT

概　要　割込みハンドラーの最後で呼び出すマクロ。

ヘッダー　r_ospl.h

宣　言　#define　R_OSPL_END_OF_INTERRUPT()

説　明　R_OSPL_INTERRUPT_ARGUMENTS の内容によっては、内部で、R_UNREFERENCED_VARIABLE を呼び出します。

R_OSPL_INTERRUPT_ARGUMENTS は、環境（割込み API）によって割込みハンドラーの引数が異なるとき、それらを共通化するときに使います。

サンプル：

```
static void  R_DRIVER_IRQ_Handler2(
       R_OSPL_INTERRUPT_ARGUMENTS )
{
    R_DRIVER_IRQ_HandlerN( 2 );  /* 2 = channel */
    R_OSPL_END_OF_INTERRUPT();
}
```

引　数	なし	
リターン値	なし	

4.6.11　BSP_CFG_USER_LOCKING_TYPE 型に関する関数

(1)　R_OSPL_LockChannel

概　要　チャンネル番号で、ロックします。

ヘッダー　r_ospl.h

宣　言　errnum_t　R_OSPL_LockChannel(int_fast32_t　in_ChannelNum, int_fast32_t* out_ChannelNum,　mcu_lock_t　in_HardwareIndexCh0,　mcu_lock_t in_HardwareIndexChMax);

説　明　本関数は、R_DRIVER_Initialize 関数や R_DRIVER_LockChannel 関数の内部から呼び出します。
本関数の内部で、R_BSP_HardwareLock を呼び出します。

in_ChannelNum 引数に R_OSPL_UNLOCKED_CHANNEL (=0xFEE) を指定した場合、ロックされていないチャンネルをロックします。 このとき、out_ChannelNum に NULL を指定すると、R_OSPL_NDEBUG = 1 のときのみ E_OTHERS エラーになります。

チャンネル番号が 0 以外から始まるときは、最小のチャンネル番号を引いた値を、in_ChannelNum 引数に指定してください。 また、*out_ChannelNum に最小のチャンネル番号を足した値が、ロックしたチャンネル番号になります。

本関数は、対象となる周辺機能のチャンネルに対して、初期化をしません。

すでにロック済みだったときは、E_ACCESS_DENIED エラーになります。チャンネル番号が範囲外だったときは、R_OSPL_NDEBUG = 1 のときのみ E_FEW_ARRAY エラーになります。in_ChannelNum = R_OSPL_UNLOCKED_CHANNEL のとき、すべてのチャンネルが使用済みだったときは、E_FEW_ARRAY エラーになります。

引　数	int_fast32_t in_ChannelNum	ロックするチャンネル番号、または、R_OSPL_UNLOCKED_CHANNEL
	int_fast32_t* out_ChannelNum	（出力） ロックしたチャンネル番号、NULL 可能
	mcu_lock_t in_HardwareIndexCh0	チャンネル番号 0 のハードウェアの識別番号
	mcu_lock_t in_HardwareIndexChMax	チャンネル番号最大のハードウェアの識別番号
リターン値	エラーコード、正常=0	

(2)　R_OSPL_UnlockChannel

概　要　チャンネル番号で、ロック解除します。

ヘッダー　r_ospl.h

宣　言　errnum_t　R_OSPL_UnlockChannel(int_fast32_t　in_ChannelNum,　errnum_t　e, mcu_lock_t　in_HardwareIndexCh0,　mcu_lock_t　in_HardwareIndexMax);

説　明　本関数は、R_DRIVER_Finalize 関数や R_DRIVER_UnlockChannel 関数の内部から呼び出します。本関数の内部で、R_BSP_HardwareUnlock を呼び出します。

チャンネル番号が 0 以外から始まるときは、最小のチャンネル番号を引いた値を、in_ChannelNum 引数に指定してください。

本関数は、対象となる周辺機能のチャンネルに対して、終了処理をしません。

すでにロック解除されていたときは、E_ACCESS_DENIED エラーになります。in_ChannelNum 引数が R_OSPL_UNLOCKED_CHANNEL のとき、0 未満のとき、チャンネル数以上だったときは、何もせず、0（エラーなし）を返します（引数 e=0 の場合）。

引　数	int_fast32_t in_ChannelNum	チャンネル番号
	errnum_t　e	これまでに発生したエラーコード。 エラー無し=0
	mcu_lock_t in_HardwareIndexCh0	チャンネル番号 0 のハードウェアの識別番号
	mcu_lock_t in_HardwareIndexMax	チャンネル番号最大のハードウェアの識別番号
リターン値	エラーコード または e、0=「成功かつ e=0」	

(3)　R_BSP_HardwareLock

概　要　ハードウェアの識別番号で、ロックします。

ヘッダー　r_ospl.h または locking.h または platform.h

宣　言　bool_t　R_BSP_HardwareLock(mcu_lock_t　in_HardwareIndex);

説　明　複数のチャンネルがあるときは、R_DRIVER_LockChannel 関数を呼び出す方が便利です。

BSP_CFG_USER_LOCKING_ENABLED が 0 なら、内部で以下の順に呼び出します。

- R_OSPL_Start_T_Lock（R_OSPL_IS_PREEMPTION = 1 の場合のみ）
- R_OSPL_C_LOCK_Lock
- R_OSPL_End_T_Lock（R_OSPL_IS_PREEMPTION = 1 の場合のみ）

ただし、RZ/A1H の RTX BSP 環境で、BSP_CFG_USER_LOCKING_ENABLED

が 0 なら、DMAC の識別番号が指定された場合のみ、本関数の内部で、RTX BSP の DMA ドライバーの DMA_Alloc を呼び出します。

BSP_CFG_USER_LOCKING_ENABLED が 1 なら、BSP_CFG_USER_LOCKING_HW_LOCK_FUNCTION マクロの内容の関数が呼ばれます。

引　数	mcu_lock_t in_HardwareIndex	ハードウェアの識別番号
リターン値	成功したかどうか	

(4)　R_BSP_HardwareUnlock

概　要　ハードウェアの識別番号で、ロックを解除します。

ヘッダー　r_ospl.h または locking.h または platform.h

宣　言　bool_t　R_BSP_HardwareUnlock(mcu_lock_t　in_HardwareIndex);

説　明　複数のチャンネルがあるときは、R_DRIVER_UnlockChannel 関数を呼び出す方が便利です。

すでにロック解除されていたときは、E_ACCESS_DENIED エラーになります。

BSP_CFG_USER_LOCKING_ENABLED が 0 なら、内部で以下の順に呼び出します。

- R_OSPL_Start_T_Lock（R_OSPL_IS_PREEMPTION = 1 の場合のみ）
- R_OSPL_C_LOCK_Unlock
- R_OSPL_End_T_Lock（R_OSPL_IS_PREEMPTION = 1 の場合のみ）

ただし、RZ/A1H の RTX BSP 環境で、BSP_CFG_USER_LOCKING_ENABLED が 0 なら、DMAC の識別番号が指定された場合のみ、本関数の内部で、RTX BSP の DMA ドライバーの DMA_Free を呼び出します。

ハードウェアの識別番号が範囲外だったときは、R_OSPL_RaiseUnrecoverable 関数が呼ばれます。

BSP_CFG_USER_LOCKING_ENABLED が 1 なら、BSP_CFG_USER_LOCKING_HW_UNLOCK_FUNCTION マクロの内容の関数が呼ばれます。

引　数	mcu_lock_t in_HardwareIndex	ハードウェアの識別番号
リターン値	成功したかどうか	

(5)　R_BSP_SoftwareLock

概　要　ロック オブジェクトで、ロックします。

ヘッダー　r_ospl.h または locking.h または platform.h

宣　言　bool_t　R_BSP_SoftwareLock(BSP_CFG_USER_LOCKING_TYPE* LockObject);

説　明　BSP_CFG_USER_LOCKING_ENABLED が 0 なら、内部で以下の順に呼び出します。

- R_OSPL_Start_T_Lock （R_OSPL_IS_PREEMPTION = 1 の場合のみ）
- R_OSPL_C_LOCK_Lock
- R_OSPL_End_T_Lock （R_OSPL_IS_PREEMPTION = 1 の場合のみ）

1 なら、BSP_CFG_USER_LOCKING_SW_LOCK_FUNCTION マクロの内容の関数が呼ばれます。

引　数	BSP_CFG_USER_LOCKING_TYPE* LockObject	ロック オブジェクト
リターン値	成功したかどうか	

サンプル：

```
/* (*1) オーナーがスレッドのとき */
/* (*2) オーナーがコンテキストのとき */
BSP_CFG_USER_LOCKING_TYPE  lock;  /* 0 初期化グローバル変数 */
r_ospl_thread_id_t        lock_owner_thread = R_OSPL_THREAD_NULL;  /* (*1) */
r_ospl_c_lock_t*          lock_owner_object = NULL;  /* (*2) */


    /* 初期化やオープン関数の中で */
    b= R_BSP_SoftwareLock( &lock ); IF(!b){e=E_ACCESS_DENIED; goto fin;}
    lock_owner_thread = R_OSPL_THREAD_GetCurrentId();  /* (*1) */
    lock_owner_object = &lock;                        /* (*2) */


    /* それぞれの API 関数の開始時に、ロックしているか（権利があるか）どうかをチェックする */
    IF ( lock_owner_thread != R_OSPL_THREAD_GetCurrentId() )  /* (*1) */
       { e=E_ACCESS_DENIED; goto fin; }
    IF ( lock_owner_object != &lock )                        /* (*2) */
       { e=E_ACCESS_DENIED; goto fin; }

fin:
    /* 終了処理やクローズ関数の中で */
    if ( lock_owner_thread == R_OSPL_THREAD_GetCurrentId() )  /* (*1) */
    /* if ( lock_owner_object == &lock ) */                  /* (*2) */
    {
       lock_owner_thread = R_OSPL_THREAD_NULL;  /* (*1) */
       lock_owner_object = NULL;  /* (*2) */
       ee= R_BSP_SoftwareUnlock( &lock ); e=R_OSPL_MergeErrNum( e, ee );
    }
```

(6)　R_BSP_SoftwareUnlock

概　要	ロック オブジェクトで、ロックを解除します。
ヘッダー	r_ospl.h または locking.h または platform.h
宣　言	bool_t　R_BSP_SoftwareUnlock(BSP_CFG_USER_LOCKING_TYPE* LockObject);
説　明	すでにロック解除されていたときは、E_ACCESS_DENIED エラーになります。 LockObject = NULL のときは、何もせず、true を返します。 BSP_CFG_USER_LOCKING_ENABLED が 0 なら、内部で以下の順に呼び出します。 ● R_OSPL_Start_T_Lock （R_OSPL_IS_PREEMPTION = 1 の場合のみ） ● R_OSPL_C_LOCK_Unlock ● R_OSPL_End_T_Lock （R_OSPL_IS_PREEMPTION = 1 の場合のみ） 1 なら、BSP_CFG_USER_LOCKING_SW_UNLOCK_FUNCTION マクロの内容の関数が呼ばれます。

引　数	BSP_CFG_USER_LOCKING_TYPE* LockObject	ロック オブジェクト
リターン値	成功したかどうか	

4.6.12　r_ospl_c_lock_t 型に関する関数

(1)　R_OSPL_C_LOCK_InitConst

概　要　C-ロック・オブジェクトを初期化します。
ヘッダー　r_ospl.h
宣　言　void　R_OSPL_C_LOCK_InitConst(r_ospl_c_lock_t* self);
説　明　0 で初期化されるグローバル変数や静的変数の場合、本関数を呼び出す必要はありません。

引　数	r_ospl_c_lock_t* self	C-ロック オブジェクト
リターン値	なし	

(2)　R_OSPL_C_LOCK_Lock

概　要　ロックできる状態なら、ロックします。
ヘッダー　r_ospl.h
宣　言　errnum_t　R_OSPL_C_LOCK_Lock(r_ospl_c_lock_t* self);
説　明　ロックしたオーナーであっても、すでにロック済みだったら、E_ACCESS_DENIED エラーになります。

OS ありでは、オーナーを管理する変数を変更する処理と同じロック区間の中から呼び出してください。 本関数は、内部で排他制御を行いません。

割込みコンテキストから呼び出すと、E_NOT_THREAD エラーになります。

引　数	r_ospl_c_lock_t* self	C-ロック オブジェクト
リターン値	エラーコード、正常=0	

(3)　R_OSPL_C_LOCK_Unlock

概　要　ロック解除します。
ヘッダー　r_ospl.h
宣　言　errnum_t　R_OSPL_C_LOCK_Unlock(r_ospl_c_lock_t* self);
説　明　ロックされていないときに本関数を呼び出すと、何もせず、E_ACCESS_DENIED エラーになります。
self = NULL のときは、何もせず、0（エラーなし）を返します。
割込みコンテキストから呼び出すと、E_NOT_THREAD エラーになります。
本関数は、内部で排他制御を行いません。

OS ありでは、オーナーを管理する変数を変更する処理と同じロック区間の中から呼び出してください。

引　数	r_ospl_c_lock_t* self	C-ロック オブジェクト
リターン値	エラーコード、正常=0	

4.6.13　r_ospl_table_t - 配列番号表に関する関数

(1)　R_OSPL_TABLE_DEF

概　要　配列番号表のワーク領域や初期値を宣言します。
ヘッダー　r_ospl.h
宣　言　#define　R_OSPL_TABLE_DEF(Name, MaxCount, Flags)

説　明　　ライブラリーの中で本マクロを使用しないでください。
OSPL の内容や OS を変更したときに、ライブラリーの再コンパイルが必要になってしまいます。
本マクロは、グローバル スコープにのみ記述できます。
本マクロで宣言した配列番号表は、R_OSPL_TABLE で取得できます。

引　数		
	Name	配列番号表の名前。" " で囲まない
	MaxCount	配列番号表の最大要素数
	Flags	参照：(4.4.11) r_ospl_table_flags_t
リターン値	なし	

(2)　R_OSPL_TABLE

概　要　　配列番号表を返します。
ヘッダー　　r_ospl.h
宣　言　　r_ospl_table_t*　R_OSPL_TABLE(Name);
説　明　　ライブラリーの中で本マクロを使用しないでください。
OSPL の内容や OS を変更したときに、ライブラリーの再コンパイルが必要になってしまいます。

引　数		
	Name	R_OSPL_TABLE_DEF で定義した配列番号表の名前。" " で囲まない
リターン値	配列番号表	

(3)　R_OSPL_TABLE_InitConst

概　要　　配列番号表を初期化します。
ヘッダー　　r_ospl.h
宣　言　　void　R_OSPL_TABLE_InitConst(r_ospl_table_t*　self,　void*　in_Area,　uint32_t　in_AreaSize,　r_ospl_table_flags_t　in_Flags);
説　明　　R_OSPL_TABLE_DEF で宣言した変数に対しては、本関数を呼び出す必要はありません。R_OSPL_TABLE_DEF で宣言していない変数に対しては、本関数を呼び出す必要があります。

引　数		
	r_ospl_table_t*　self	配列番号表
	void*　in_Area	配列番号表に使う領域の先頭アドレス
	uint32_t　in_AreaSize	配列番号表に使う領域のサイズ（バイト） 参照：R_OSPL_TABLE_SIZE
	r_ospl_table_flags_t in_Flags	オプション。 0 または r_ospl_table_flags_t (4.4.11) の論理和。
リターン値	なし	

(4)　R_OSPL_TABLE_SIZE

概　要　　配列番号表に使う領域のサイズを計算します。
ヘッダー　　r_ospl.h
宣　言　　#define　R_OSPL_TABLE_SIZE(int_fast32_t　in_MaxCount)
説　明　　0 ～ in_MaxCount - 1 の配列番号を管理するようになります。

引　数		
	int_fast32_t　in_MaxCount	配列の要素数
リターン値	配列番号表に使う領域のサイズ（バイト）	

(5)　R_OSPL_TABLE_GetIndex

概　要　　キーから配列番号を返します。
ヘッダー　　r_ospl.h

宣　言　errnum_t　R_OSPL_TABLE_GetIndex(r_ospl_table_t*　self,　void*　in_Key, int_fast32_t*　out_Index,　r_ospl_if_not_t　in_TypeOfIfNot);

説　明　in_TypeOfIfNot = R_OSPL_DO_NOTHING_IF_NOT のときは、*out_Index = R_OSPL_NO_INDEX に設定してから呼び出すと、登録されていないことが判定できるようになります。
in_TypeOfIfNot = R_OSPL_OUTPUT_IF_NOT のときは、*out_Index = R_OSPL_NO_INDEX に設定してから呼び出すと、すでに登録されていることが判定できるようになります。

定数名	設定値	内容
R_OSPL_NO_INDEX	-1	該当する配列番号がない

引　数	r_ospl_table_t*　self	配列番号表
	void*　in_Key	キー、ここに渡された値（アドレス）をキーとします
	int_fast32_t*　out_Index	（出力）　配列番号
	r_ospl_if_not_t　in_TypeOfIfNot	登録されていないときの動作 参照：r_ospl_if_not_t (4.4.12)
リターン値	エラーコード、正常=0	

(6)　R_OSPL_TABLE_Free

概　要　キーと配列番号を切り離し、配列番号表から除外します。（キー指定）

ヘッダー　r_ospl.h

宣　言　void　R_OSPL_TABLE_Free(r_ospl_table_t*　self,　void*　in_Key);

説　明　すでに、除外されているときでも、エラーになりません。
一度でも R_OSPL_ALLOCATE_IF_EXIST_OR_IF_NOT で配列番号を取得した後は、R_OSPL_RaiseUnrecoverable(E_STATE) を呼び出します。

引　数	r_ospl_table_t*　self	配列番号表
	void*　in_Key	キー
リターン値	なし	

(7)　R_OSPL_TABLE_FreeByIndex

概　要　キーと配列番号を切り離し、配列番号表から除外します。　（配列番号指定）

ヘッダー　r_ospl.h

宣　言　void　R_OSPL_TABLE_FreeByIndex(r_ospl_table_t*　self,　int_fast32_t in_Index);

説　明　一度も R_OSPL_ALLOCATE_IF_EXIST_OR_IF_NOT で配列番号を取得していないときは、R_OSPL_RaiseUnrecoverable(E_STATE) を呼び出します。
すでに、除外されているときは、
R_OSPL_RaiseUnrecoverable(E_NOT_FOUND_SYMBOL) を呼び出します。

引　数	r_ospl_table_t*　self	配列番号表
	int_fast32_t　in_Index	配列番号
リターン値	なし	

(8)　R_OSPL_TABLE_GetStatus

概　要　配列番号表の状況（を示す構造体へのポインター）を取得します。

ヘッダー　r_ospl.h

宣　言　errnum_t　R_OSPL_TABLE_GetStatus(r_ospl_table_t*　self,　const r_ospl_table_status_t** out_Status);

説　明　out_Status 引数に指定するポインター変数には、const 修飾子が必要です。

本関数を呼ばなくても、*out_Status の値は更新されます。

引　数	r_ospl_table_t*　self	配列番号表
	const r_ospl_table_status_t** out_Status	（出力）　配列番号表の状況を示す構造体へのポインター。　参考 (4.5.14) r_ospl_table_status_t
リターン値	なし	

4.6.14　メモリーに関する関数

(1)　R_OSPL_MEMORY_Flush

概　要　キャッシュをフラッシュします。

ヘッダー　r_ospl.h

宣　言　void　R_OSPL_MEMORY_Flush(r_ospl_flush_t　in_FlushType);

説　明　ハードウェアがアクセスするデータ領域がキャッシュ領域のときは、キャッシュを制御しないドライバーでは、キャッシュ上に作成した入出力バッファーをフラッシュしてから、ドライバーの関数を呼び出してください。
キャッシュを制御するドライバーかどうかは、ドライバーの仕様によります。

RZ/A1H では、in_FlushType 引数に、
R_OSPL_FLUSH_WRITEBACK_INVALIDATE か
R_OSPL_FLUSH_WRITEBACK_INVALIDATE_2ND を指定できます。ただし、
R_OSPL_FLUSH_WRITEBACK_INVALIDATE_2ND は通常使いません。

引　数	r_ospl_flush_t in_FlushType	フラッシュ操作の種類
リターン値	なし	

(2)　R_OSPL_MEMORY_RangeFlush

概　要　キャッシュをフラッシュします。 仮想アドレスの範囲指定あり。

ヘッダー　r_ospl.h

宣　言　errnum_t　R_OSPL_MEMORY_RangeFlush(r_ospl_flush_t　in_FlushType, void* in_StartAddress, size_t　in_Length);

説　明　in_StartAddress 引数と in_Length 引数は、キャッシュラインのサイズに合わせてください。合っていないときは、E_OTHERS エラーになります。
参考：R_OSPL_MEMORY_GetSpecification

ハードウェアがライトして CPU がリードするデータ領域がキャッシュ領域のときに、データ領域に対応するキャッシュ領域を無効化
（R_OSPL_FLUSH_WRITEBACK_INVALIDATE、または、
R_OSPL_FLUSH_INVALIDATE）をせずにハードウェアを起動したときは、ハードウェアがライトを完了した後で、キャッシュ領域を無効化してからリードしてください。（キャッシュを制御しないドライバーの場合のみ）

RZ/A1H では、in_FlushType 引数に、R_OSPL_FLUSH_INVALIDATE のみ指定できます。

引　数	r_ospl_flush_t in_FlushType	フラッシュ操作の種類
	void* in_StartAddress	フラッシュする範囲の先頭の仮想アドレス
	size_t　in_Length	フラッシュする範囲のサイズ（バイト）
リターン値	エラーコード、正常=0	

(3)　R_OSPL_MEMORY_GetLevelOfFlush

概　要　指定したメモリーをフラッシュするのに必要なレベルを取得します

ヘッダー　r_ospl.h

宣　言　errnum_t　R_OSPL_MEMORY_GetLevelOfFlush(void* in_Address, int_fast32_t* out_Level);

説　明

引　数		
	void* in_Address	フラッシュするメモリーの中のアドレス
	int_fast32_t* out_Level	（出力）0=フラッシュ不要、1=L1 キャッシュのみ、2=L1, L2 キャッシュの両方
リターン値	エラーコード、正常=0	

(4)　R_OSPL_MEMORY_GetMaxLevelOfFlush

概　要　すべてのメモリーをフラッシュするのに必要なキャッシュのレベルを取得します。

ヘッダー　r_ospl.h

宣　言　int_fast32_t　R_OSPL_MEMORY_GetMaxLevelOfFlush();

説　明　L2 キャッシュが存在しても、フラッシュが不要であれば 1 を返します。

引　数		
	なし	
リターン値	キャッシュのレベル。 1=L1 キャッシュ、2=L2 キャッシュ	

(5)　R_OSPL_MEMORY_GetSpecification

概　要　メモリーやキャッシュの仕様を取得します。

ヘッダー　r_ospl.h

宣　言　void　R_OSPL_MEMORY_GetSpecification(r_ospl_memory_spec_t* out_MemorySpec);

説　明

引　数		
	r_ospl_memory_spec_t* out_MemorySpec	メモリーやキャッシュの仕様
リターン値	なし	

(6)　R_OSPL_ToPhysicalAddress

概　要　仮想アドレスから物理アドレスに変換します。

ヘッダー　r_ospl.h

宣　言　errnum_t　R_OSPL_ToPhysicalAddress(void* in_Address, uintptr_t* out_PhysicalAddress);

説　明　例： 物理アドレスが整数型の場合

```
uintptr_t  physical_address;

e= R_OSPL_ToPhysicalAddress( address, &physical_address );
IF(e){goto fin;}
```

例： 物理アドレスがポインター型の場合

```
uintptr_t  physical_address;
void*      pointer;

e= R_OSPL_ToPhysicalAddress( address, &physical_address );
IF(e){goto fin;}
pointer = (void*) physical_address;
```

参考：(4.8.9) キャッシュ領域と非キャッシュ領域の配置パターン（RZ/A1）

引　数		
	void*　in_Address	仮想アドレス
	uintptr_t* out_PhysicalAddress	（出力）物理アドレス
リターン値	エラーコード。エラーなし＝0	

(7) R_OSPL_ToCachedAddress

概　要	キャッシュ領域のアドレスに変換します。
ヘッダー	r_ospl.h
宣　言	errnum_t　R_OSPL_ToCachedAddress(void* in_Address, void* out_CachedAddress);
説　明	out_CachedAddress 引数は、ポインターへのアドレスを指定してください。 ミラー領域がない場合、in_Address に非キャッシュ領域のアドレスを指定すると、E_ACCESS_DENIED エラーになります。 E_ACCESS_DENIED エラーになったとき、in_Address 引数に渡されたアドレスに対応する変数が何かは、map ファイルから分かるかもしれません。 ミラー領域がないメモリーマップにした環境では、アドレス変換の関数を呼び出す必要はありませんが、指定したアドレスがキャッシュ領域か非キャッシュ領域かをチェックすることに使えます。 参考：(4.8.9) キャッシュ領域と非キャッシュ領域の配置パターン（RZ/A1）

引　数		
	void*　in_Address	仮想アドレス
	void* out_CachedAddress	（出力）キャッシュ領域の仮想アドレス
リターン値	エラーコード。エラーなし＝0	

(8) R_OSPL_ToUncachedAddress

概　要	非キャッシュ領域のアドレスに変換します。
ヘッダー	r_ospl.h
宣　言	errnum_t　R_OSPL_ToUncachedAddress(void* in_Address, void* out_UncachedAddress);
説　明	out_UncachedAddress 引数は、ポインターへのアドレスを指定してください。 ミラー領域がない場合、in_Address に非キャッシュ領域のアドレスを指定すると、E_ACCESS_DENIED エラーになります。 E_ACCESS_DENIED エラーになったとき、in_Address 引数に渡されたアドレスに対応する変数が何かは、map ファイルから分かるかもしれません。 ミラー領域がないメモリーマップにした環境では、アドレス変換の関数を呼び出す必要はありませんが、指定したアドレスがキャッシュ領域か非キャッシュ領域かをチェックすることに使えます。 参考：(4.8.9) キャッシュ領域と非キャッシュ領域の配置パターン（RZ/A1）

引　数		
	void*　in_Address	仮想アドレス
	void* out_UncachedAddress	（出力）非キャッシュ領域の仮想アドレス
リターン値	エラーコード。エラーなし＝0	

(9) R_OSPL_MEMORY_Barrier

概　要	メモリー バリアを設定します。
ヘッダー	r_ospl.h
宣　言	void　R_OSPL_MEMORY_Barrier(void);
説　明	ARM では、DSB アセンブリ命令になります。

引　数		
	なし	
リターン値	なし	

(10) R_OSPL_InstructionSyncBarrier

概　要	命令同期バリアを設定します。
ヘッダー	r_ospl.h
宣　言	void　R_OSPL_InstructionSyncBarrier(void);

説　明	ARM では、ISB アセンブリ命令になります。

引　数	なし	
リターン値	なし	

(11) R_OSPL_AXI_Get2ndCacheAttribute

概　要	アドレスから AXI バスの L2 キャッシュの属性を取得します
ヘッダー	r_ospl.h
宣　言	errnum_t　R_OSPL_AXI_Get2ndCacheAttribute(uintptr_t　in_PhysicalAddress, r_ospl_axi_cache_attribute_t*　out_CacheAttribute);
説　明	メモリーの種類によってキャッシュ属性が異なるときに使用します。

引　数	uintptr_t in_PhysicalAddress	メモリー領域の中の物理アドレス
	r_ospl_axi_cache_attribute_t* out_CacheAttribute	（出力）AXI バスの L2 キャッシュの属性
リターン値	エラーコード、正常=0	

(12) R_OSPL_AXI_GetProtection

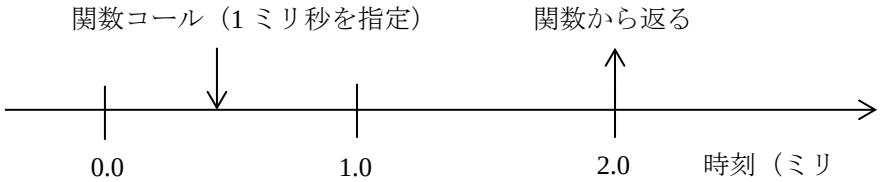
概　要	アドレスから AXI バスの保護属性を取得します
ヘッダー	r_ospl.h
宣　言	errnum_t　R_OSPL_AXI_GetProtection(uintptr_t　in_PhysicalAddress, r_ospl_axi_protection_t*　out_Protection);
説　明	メモリーの種類によって保護属性が異なるときに使用します。

引　数	uintptr_t in_PhysicalAddress	メモリー領域の中の物理アドレス
	r_ospl_axi_protection_t* out_Protection	（出力）AXI バスの保護属性
リターン値	エラーコード、正常=0	

4.6.15　時間に関する関数

(1) R_OSPL_Delay

概　要	指定した時間だけ待ちます。
ヘッダー	r_ospl.h
宣　言	errnum_t　R_OSPL_Delay(uint32_t　in_DelayTime_msec);
説　明	OS レスでは、フリーランタイマーを参照します。 OS ありでは、OS の機能を呼び出します。 タイマーの精度によって、指定した時間より長く待つ場合があります。 たとえば、精度が 1 ミリ秒で、1 ミリ秒待つ指定がされた場合、0.4 ミリ秒の時刻で呼ばれたら、1.6 ミリ秒後の、2.0 ミリ秒の時刻になったら返ります。



in_DelayTime_msec 引数に、R_OSPL_MAX_TIME_OUT を超える値を指定したときは、R_OSPL_MAX_TIME_OUT ミリ秒だけ待ってから、E_TIME_OUT エラーを返します。デバッグ版なら（R_OSPL_NDEBUG が定義されていなかったら）、待つ前に、内部で ASSERT_D に引っかかります。R_OSPL_MAX_TIME_OUT ミリ

秒以上待つときは、OS ありでは周期ハンドラーを使ってください。OS レスでは R_OSPL_FTIMER_IsPast をオーバーフローに注意しながら使ってください。

割込みハンドラーの中から R_OSPL_Delay を呼び出すと、R_OSPL_RaiseUnrecoverable (E_NOT_THREAD) が呼ばれます。　もし、R_OSPL_RaiseUnrecoverable から返ると、R_OSPL_Delay 関数は、E_NOT_THREAD エラーを返します。割込みハンドラーの中で待つときは、R_OSPL_FTIMER_IsPast をオーバーフローに注意しながら使ってください。ただし、優先度が最高のスレッドさえ動けなくなります。

OS レスの場合は、待っている間に r_ospl_idle_callback_t 型の関数を何度もコールバックします。擬似マルチスレッドを構成するときは、本関数の代わりに R_OSPL_FTIMER_IsPast 関数を使用してください。
OS によっては、割込みハンドラーの中から本関数と同様の関数を呼び出すと、その関数はエラーになります。その環境では、本関数はポーリングします。

R_OSPL_IS_PREEMPTION = 0 に定義されているときは、R_OSPL_THREAD_SetOnWait 関数に、R_OSPL_WAIT_PM_THREAD を設定すると、待っている状態でも、R_OSPL_Delay 関数からすぐに返ります。 このときの R_OSPL_Delay 関数の返り値は、0 です。待っている状態かどうかは、R_OSPL_THREAD_GetIsWaiting 関数で分かります。

引　数	uint32_t in_DelayTime_msec	待つ時間（ミリ秒、最大 R_OSPL_MAX_TIME_OUT = 65533）
リターン値	なし	

(2)　R_OSPL_FTIMER_InitializeIfNot

概　要　フリーラン タイマーを起動します。

ヘッダー　r_ospl.h

宣　言　errnum_t　R_OSPL_FTIMER_InitializeIfNot(r_ospl_ftimer_spec_t* out_Specification);

説　明　フリーラン タイマーは、停止しないタイマーです。
オーバーフローしたら 0 に戻ります。
割込みハンドラーの中でもカウントアップします。
割込みは使いません。

R_OSPL_FTIMER_IS マクロで、使用するタイマーを選択できます。

すでに起動されているときは、起動処理は行わず、out_Specification 引数に出力して、エラーにはなりません。
タイムアウトを指定する OSPL の API 関数や R_OSPL_Delay 関数を呼び出したら、それらの関数の中から本関数が呼ばれます。
内部に全割込み禁止区間があります。

引　数	r_ospl_ftimer_spec_t* out_Specification	NULL 可、（出力） フリーラン タイマーの精度
リターン値	エラーコード。エラーなし＝0	

(3)　R_OSPL_FTIMER_Get

概　要　フリーラン タイマーの現在の時刻を返します。

ヘッダー　r_ospl.h

宣　言　uint32_t　R_OSPL_FTIMER_Get(void);

説　明　本関数を呼び出す前に R_OSPL_FTIMER_InitializeIfNot 関数を呼び出しておいてください。
時間が経過したかどうかを判定するときは、R_OSPL_FTIMER_IsPast 関数を呼び出してください。

例：

```
errnum_t            e;
r_ospl_ftimer_spec_t  ts;
uint32_t            start;
uint32_t            end;

e= R_OSPL_FTIMER_InitializeIfNot( &ts ); IF(e){goto fin;}
start = R_OSPL_FTIMER_Get();

/* 計測区間 */

end = R_OSPL_FTIMER_Get();
printf( "%d msec¥n", R_OSPL_FTIMER_CountToTime(
    &ts, end - start ) );
```

引　数	なし	
リターン値	フリーラン タイマーの現在のクロックカウント値	

(4)　R_OSPL_FTIMER_IsPast

概　要　フリーラン タイマーが指定した時刻を経過したかどうかを返します。

ヘッダー　r_ospl.h

宣　言　errnum_t　R_OSPL_FTIMER_IsPast(r_ospl_ftimer_spec_t* out_Specification,
uint32_t Now, uint32_t TargetTime, bool_t* out_IsPast);

説　明　フリーランタイマーの時刻と TargetTime に指定した時刻が同じときは、false を返します。超えてから true になります。 （これは、開始時刻＋経過時間の時刻では、タイマーの精度より小さい時間だけ経過時間より短いことがあるからです）

Now 引数は、実際の現在の時刻でなくても構いません。 複数のモジュール間で同じ時刻に対する処理ができるようにするためには、それらのモジュールに対して同じ時刻を渡さなければならないからです。

ターゲット時刻から猶予期間（r_ospl_ftimer_spec_t::ExtensionOfCount）を超えた時刻を Now 引数に指定すると、E_TIME_OUT エラーになります。 よって、本関数を繰り返し呼び出す間隔は、猶予期間より短くしてください。 また、カウンターがオーバーフローする時間以上の先の時刻はターゲットにできません。 これに関連するエラーが検出できるよう、現在時刻＋最大カウント数／２ を超える値を指定したら、E_TIME_OUT エラーになります。 out_Specification 引数の中にある猶予期間はアプリケーションが変更することができます。 ただし、猶予期間を長くしすぎると、エラー検出できる可能性が低くなります。 詳しくは、下記「境界値について」を参照してください。

最大カウント数／２ を超えた先の時刻を過ぎたかどうかを判定するときは、一度 TargetTime に最大カウント数／２ を設定し、その時刻を超えたと判断したときに、残りの時間を TargetTime に加算することを繰り返してください。 （下記サンプルを参照）

指定した時刻を過ぎるまでポーリング（ループ）している間は、優先順位の低いスレッドが動かなくります。 割込みハンドラーでポーリング（ループ）すると、どのスレッドも動かなくなります。

R_OSPL_Delay 関数を呼び出せば、OS ありの環境では、本関数の代わりに他のスレッドが動くことができますが、OS レスでは他のスレッドは動きません（割込みは動きます）。

例：300 ミリ秒経過したかどうかを判断する。

```
errnum_t            e;
r_ospl_ftimer_spec_t  ts;
uint32_t            target;
bool_t              is_past;

e= R_OSPL_FTIMER_InitializeIfNot( &ts ); IF(e){goto fin;}

target = R_OSPL_FTIMER_Get() +
    R_OSPL_FTIMER_TimeToCount( &ts, 300 );
   /* or  previous_target +
       R_OSPL_FTIMER_TimeToCount( &ts, 300 ); */

/* ... */

e= R_OSPL_FTIMER_IsPast( &ts, R_OSPL_FTIMER_Get(),
   target, &is_past );
   IF(e){goto fin;}
```

例：1 時間経過したかどうかを判断する。

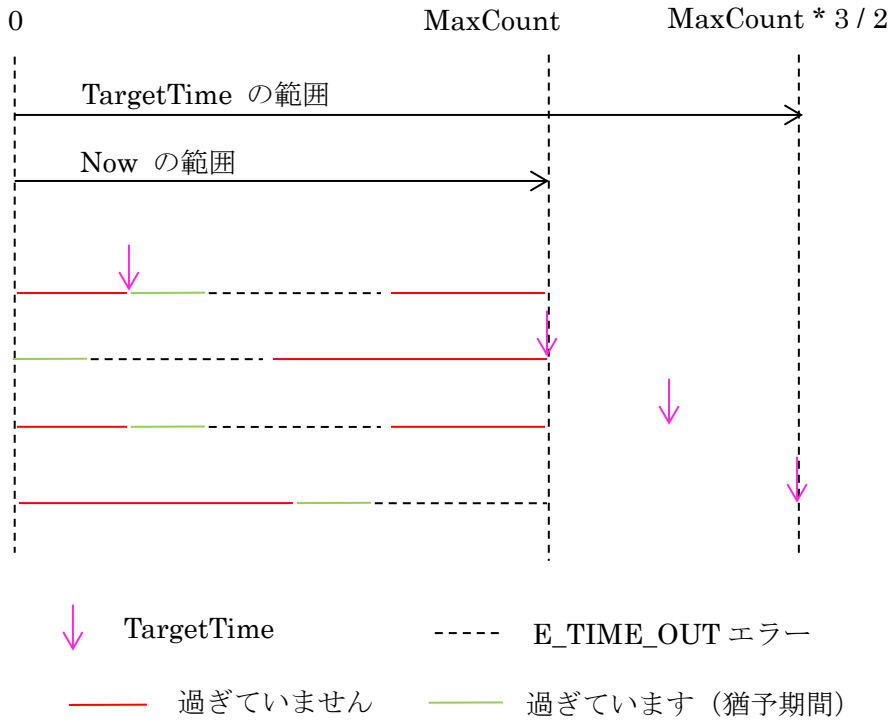
```
errnum_t            e;
r_ospl_ftimer_spec_t  ts;
uint32_t            target;
bool_t              is_past;
const uint32_t       interval_time = 60 * 1000;  /* msec
 */
uint32_t            interval_count;
uint32_t            elapsed_time;

e= R_OSPL_FTIMER_InitializeIfNot( &ts ); IF(e){goto fin;}

if ( ts.msec_Numerator >= ts.msec_Denominator ) {
   printf( "Max time is UINT32_MAX/2 msec¥n" );
} else {
   printf( "Max time is %d msec¥n",
     R_OSPL_FTIMER_CountToTime( &ts, ts.MaxCount / 2 ) );
}

ASSERT_R( ts.msec_Numerator >= ts.msec_Denominator  ||
   R_OSPL_FTIMER_CountToTime( &ts, ts.MaxCount / 2 ) >=
   interval_time,
   e=E_LIMITATION; goto fin );

interval_count = R_OSPL_FTIMER_TimeToCount( &ts,
    interval_time );
elapsed_time = 0;
target = R_OSPL_FTIMER_Get() + interval_count;

/* ... */
```

```
    e= R_OSPL_FTIMER_IsPast( &ts, R_OSPL_FTIMER_Get(),
target,
       &is_past );
       IF(e){goto fin;}
       /* この関数を繰り返し呼び出す間隔は、*/
       /* ts.ExtensionOfCount 以内であること */
    if ( is_past ) {
       elapsed_time += 1;
       if ( elapsed_time >= 60 ) {
          ...
       } else {
          target += interval_count;
       }
    }
```

境界値について：

r_ospl_ftimer_spec_t::MaxCount が uint32_t の最大値のときは、0 に回り込んだ時でも正しく判定できるよう、TargetTime が r_ospl_ftimer_spec_t::MaxCount より大きいときは、フリーランタイマーが、TargetTime - (MaxCount + 1) を超えたかどうかで判定します。

r_ospl_ftimer_spec_t::MaxCount が uint32_t の最大値ではないときは、TargetTime 引数に、r_ospl_ftimer_spec_t::MaxCount を超える値を指定することができます。ただし、現在の時刻＋ r_ospl_ftimer_spec_t ::MaxCount / 2 を超える時刻を指定したら、E_TIME_OUT エラーになります。



引　数

r_ospl_ftimer_spec_t* out_Specification	フリーラン タイマーの精度
uint32_t Now	現在の時刻のカウント値

	uint32_t TargetTime	ターゲット時刻のカウント値
	bool_t* out_IsPast	（出力） ターゲット時刻を過ぎたかどうか
リターン値	エラーコード、正常=0、E_TIME_OUT など	

(5) R_OSPL_FTIMER_TimeToCount

概　要　ミリ秒単位から、フリーランタイマーの単位に変換します。

ヘッダー　r_ospl.h

宣　言　uint32_t　R_OSPL_FTIMER_TimeToCount(r_ospl_ftimer_spec_t* ts, uint32_t msec);

説　明　端数は切り上げられます。（指定された時間以上待つため）
下記の計算をします。

```
( msec * ts->msec_Denominator + ts->msec_Numerator - 1 ) / ts->msec_Numerator
```

ts->msec_Numerator より ts->msec_Denominator が大きいときは、オーバーフローしないように注意してください。

引　数	r_ospl_ftimer_spec_t* ts	フリーラン タイマーの精度
	uint32_t msec	ミリ秒単位の値
リターン値	フリーランタイマーの単位の値	

(6) R_OSPL_FTIMER_CountToTime

概　要　フリーランタイマーの単位から、ミリ秒単位に変換します。

ヘッダー　r_ospl.h

宣　言　uint32_t　R_OSPL_FTIMER_CountToTime(r_ospl_ftimer_spec_t* ts, uint32_t Count);

説　明　端数は切り下げられます。（Count = r_ospl_ftimer_spec_t::MaxCount のときにオーバーフローしないようにするため）
下記の計算をします。

```
( Count * ts->msec_Numerator ) / ts->msec_Denominator
```

引　数	r_ospl_ftimer_spec_t* ts	フリーラン タイマーの精度
	uint32_t Count	フリーランタイマーの単位の値
リターン値	ミリ秒単位の値	

(7) R_OSPL_FTIMER_GetSpecification

概　要　フリーランタイマーの精度を取得します。

ヘッダー　r_ospl.h

宣　言　void　R_OSPL_FTIMER_GetSpecification(r_ospl_ftimer_spec_t* out_Specification);

説　明　R_OSPL_FTIMER_IS マクロに設定できる値を追加したときや、OS を変更したときは、本関数の定義の変更を検討してください。

引　数	r_ospl_ftimer_spec_t* out_Specification	NULL 可、（出力） フリーラン タイマーの精度
リターン値	なし	

4.6.16　アイドル状態に関する関数

(1) R_OSPL_IDLE_Start_CPU_Load

概　要　CPU 使用率の計測を開始します。（OS レス専用）

ヘッダー　r_ospl_os_less.h
宣　言　void　R_OSPL_IDLE_Start_CPU_Load(int32_t Interval_msec);
説　明　本関数を呼び出すと、R_OSPL_OnIdleDefault 関数から定期的に CPU 使用率を printf 表示するようになります。
R_OSPL_CPU_LOAD を 1 にしてコンパイルする必要があります。
すぐに表示させたいときは、R_OSPL_IDLE_Print_CPU_Load 関数を呼び出します。
また、R_OSPL_WAIT_POLLING に設定してある必要があります。

引　数	int32_t　Interval_msec	定期的に表示する間隔（ミリ秒）、0=定期的に表示しない
リターン値	なし	

(2)　R_OSPL_IDLE_Print_CPU_Load

概　要　CPU 使用率を printf 表示します。（OS レス専用）
ヘッダー　r_ospl.h
宣　言　void　R_OSPL_IDLE_Print_CPU_Load(bool_t IsPrintNow);
説　明　R_OSPL_CPU_LOAD を 1 にしてコンパイルする必要があります。
R_OSPL_IDLE_Start_CPU_Load 関数を呼び出してから、または、その関数の Interval_msec 引数に指定した時刻になってから、本関数を呼び出すまでの CPU 使用率を表示します。ただし、一度でも R_OSPL_IDLE_Start_CPU_Load 関数を呼び出しておく必要があります。
IsPrintNow = false のときは、Interval_msec 引数に指定した時刻を過ぎた時だけ表示します。
R_OSPL_IDLE_Start_CPU_Load 関数の Interval_msec 引数が 0 のときは、前回 printf 表示したときから現在までの CPU 利用率を表示します。
アプリケーション開発者が本関数をカスタマイズすることができます。

R_OSPL_WAIT_POLLING に設定していなければ、常に CPU 使用率が 100% になります。

引　数	bool_t　IsPrintNow	定期的に表示するタイミングでなくても、すぐに表示するかどうか
リターン値	なし	

4.6.17　r_ospl_caller_t 型 - 割込みコールバック関数に関する関数

(1)　R_OSPL_CallInterruptCallback

概　要　割込みコールバック関数を呼び出します。
ヘッダー　r_ospl.h
宣　言　void　R_OSPL_CallInterruptCallback(r_ospl_caller_t* self,　r_ospl_interrupt_t* InterruptSource);
説　明　self に登録されている bsp_int_cb_t 型の関数を呼び出します。
ドライバーの OS 移植層にある割込みハンドラーから呼び出します。
self 引数に渡す値は、割込みが入る前にドライバー本体からドライバー移植層に渡されます。

引　数	r_ospl_caller_t*　self	割込み処理に関するドライバー内部のパラメーター
	r_ospl_interrupt_t* InterruptSource	割込み発信元
リターン値	なし	

(2)　r_ospl_callback_t

概　要　割込みコールバック関数の型。
ヘッダー　r_ospl.h

宣　言　　typedef errnum_t　(* r_ospl_callback_t)(const r_ospl_interrupt_t* InterruptSource,
　　const r_ospl_caller_t* Caller);

説　明　　割込みコンテキストで実行し、割込みハンドラーから呼び出される「割込みコールバック関数」の型。

r_ospl_async_t::.InterruptCallback にユーザー定義の割込みコールバック関数を指定して置き換えることができますが、必要になることはまずありません。

割込みコールバック関数は、通常、ドライバーから提供されるデフォルトの割込みコールバック関数を使用し、ユーザー定義の割込みコールバックが必要になることはまずありません。割込みが発生したときに実行する処理（イベントドリブンの処理）は、R_OSPL_EVENT_Wait 関数を呼び出すコードの直後に記述してください。

ユーザー定義の割込みコールバック関数から、任意のイベントを発生させたりすることができますが、下記にあるように、必要な処理も実行する必要があります。

非同期処理が完了したかどうかは、R_DRIVER_GetAsyncStatus 関数から参照できる r_ospl_async_state_t 型の変数が R_OSPL_RUNNABLE に戻ったかどうかで確認できます。

割込みコールバック関数の中から、割込み専用の return （IRET） を記述する必要はありません。 割込みコールバック関数を呼び出す割込みハンドラーが、必要な時に IRET を実行します。

割込みの種類ごとに割込みコールバック関数を別に分けることはできません。 その代わり、割込みコールバック関数を呼び出す割込みハンドラーがドライバーの下位の移植層にあり、そこは割込み番号ごとに分かれているので、そこに処理を記述することはできます。

割込みコールバック関数で必要な処理：
r_ospl_async_t::I_Thread == R_OSPL_THREAD_NULL のときは、R_DRIVER_OnInterrupting 関数を呼び出した後、R_DRIVER_OnInterrupted 関数を呼び出します。（R_DRIVER_OnInterrupted 関数が存在しないときは、呼び出しません。）

r_ospl_async_t::I_Thread != R_OSPL_THREAD_NULL のときは、R_DRIVER_OnInterrupting 関数を呼び出した後、I-イベントを発生させます。I-イベントを受信したスレッドは、R_DRIVER_OnInterrupted 関数を呼び出してください。 （R_DRIVER_OnInterrupted 関数が存在しないときは、I-イベントを発生させません。）

引　数		
	r_ospl_interrupt_t* InterruptSource	割込み発信元、型は OS またはボード定義
	r_ospl_caller_t*　Caller	割込み処理に関するドライバー内部のパラメーター
リターン値	エラーコード。エラーなし＝0。 r_ospl_async_t::ReturnValue に入れる値	

4.6.18　エラー処理、デバッグに関する関数

(1)　CHK

概　要　　エラーが発生していたら、無限ループに入ります。 デバッグ用です。
ヘッダー　　r_ospl.h
宣　言　　void　CHK(errnum_t　e);

説　明　　e == 0 なら、何もせず返ります。 そうでないなら、すべての割込みを禁止して無限ループに入ります。

サンプル A :
```
e= FunctionX(); CHK(e);
```

サンプル B :
```
CHK( FunctionX() );
```

引　数	errnum_t　e	チェックするエラーコード。
リターン値	なし	

(2)　R_OSPL_RaiseUnrecoverable

概　要　　回復不能なエラーを発生させます。
ヘッダー　　r_ospl.h
宣　言　　void　R_OSPL_RaiseUnrecoverable(errnum_t e);
説　明　　回復不能なエラーとは、ヒープ領域の破壊やハードウェアの反応なしなど、プロセスやシステムが自力で回復できないエラーです。 OS やシステムコントローラー（ソフトウェアリセット等）でのみ復帰できるエラーです。

たとえば、回復処理をしているときにエラーが発生したら、本関数を呼び出します。

ユーザーが内容を変更できる関数です。デフォルトでは、R_DebugBreak 関数を呼び出してから、無限ループに入ります。

引　数	errnum_t　e	エラーコード
リターン値	なし	

(3)　R_DEBUG_BREAK

概　要　　ブレークします。
ヘッダー　　r_ospl.h
宣　言　　#define　R_DEBUG_BREAK()
説　明　　ブレークは、R_DebugBreak 関数を呼び出すことで行われます。
R_OSPL_ERROR_BREAK マクロの設定に影響されません。

引　数	なし	
リターン値	なし	

(4)　R_DEBUG_BREAK_IF_ERROR

概　要　　エラー状態ならブレークします。
ヘッダー　　r_ospl.h
宣　言　　#define　R_DEBUG_BREAK_IF_ERROR()
説　明　　R_OSPL_ERROR_BREAK マクロが 0 のときは、本マクロを呼び出しても、何もしません。以下は、R_OSPL_ERROR_BREAK マクロを 1 に設定したときのみ有効です。

現在のスレッドのエラー状態をチェックします。
それぞれのスレッドの最後から呼び出してください。
ブレークは、R_DebugBreak 関数を呼び出すことで行われます。

エラーが発生していたら、次のようなメッセージを printf します。error_ID を

R_OSPL_SET_BREAK_ERROR_ID に設定してください。
```
<ERROR error_ID="0x1" file="../src/api.c(336)"/>
```

引　数	なし	
リターン値	なし	

(5)　IF

概　要	条件式が 0 以外（真）ならブレークしてエラー状態にします。
ヘッダー	r_ospl.h
宣　言	#define　IF (Condition)
説　明	R_OSPL_ERROR_BREAK マクロが 0 のときは、通常の if と同じです。 以下は、R_OSPL_ERROR_BREAK マクロを 1 に設定したときのみ有効です。 エラーが発生した場所を特定する作業を支援します。 例： `e= TestFunction(); IF(e){goto fin;}` ガード節や関数呼び出しの直後によくある、エラーが発生したかどうかを判定する if 文の代わりに IF マクロを記述すれば、エラーが発生したときにすぐにブレークするようになります。 エラーの原因となる状況（変数値）やエラーが発生する詳細な場所（コールスタック）をすぐに確認できるようになり、デバッグ作業の効率が上がります。 また、IF マクロは、例外的な（異常系の）処理ブロックであるとすぐに判断できるため、そこを読み飛ばせば、コードを読む効率が上がります。 ブレークさせるには、R_OSPL_SET_BREAK_ERROR_ID 関数を呼び出しておきます。 エラー状態かどうかや、エラーの発生回数は、スレッド ローカル ストレージに格納されます。リリース版では、エラー状態かどうかや、エラーの発生回数の変数はなくなります。エラーコードは、呼び出し元の自動変数など OSPL の外で管理してください。 R_OSPL_CLEAR_ERROR 関数を呼び出すと、エラー状態が解消されます。エラー状態のまま、R_DEBUG_BREAK_IF_ERROR マクロを呼び出すと、ブレークします。 R_OSPL_STACK_CHECK_CODE マクロが 1 のときは、IF マクロの内部で R_OSPL_CHECK_STACK_OVERFLOW が呼ばれます。

引　数	Condition	条件式
リターン値	なし	

(6) IF_D

概　要　デバッグ版のみ比較を行う、IF。

ヘッダー　r_ospl.h

宣　言　#define　IF_D (Condition)

説　明　リリース版では、条件式の評価結果は常に偽になります。
リリース版は、標準ライブラリー同様、R_OSPL_NDEBUG マクロが定義されている場合です。

引　数	Condition	条件式
リターン値	なし	

(7) ASSERT_R

概　要　表明（契約プログラミング）を行う。

ヘッダー　r_ospl.h

宣　言　#define　ASSERT_R(in_Condition, in_StatementsForError)

説　明　R_OSPL_ERROR_BREAK マクロが 0 のとき：
条件式が 0 （偽）なら、in_StatementsForError を実行します。

R_OSPL_ERROR_BREAK マクロが 1 のとき：
条件式が 0 （偽）なら、IF と同様にブレークしてエラー状態になり、in_StatementForError を実行します。
R_OSPL_STACK_CHECK_CODE マクロが 1 のときは、ASSERT_R マクロの内部で R_OSPL_CHECK_STACK_OVERFLOW が呼ばれます。

条件式が 0 （偽）のときに、何もしないときは、in_StatementsForError 引数を R_NOOP() にしてください。

in_Condition 引数に false を指定すると警告されるときは、R_OSPL_ReturnFalse 関数を呼び出してください。

サンプル：

```
ASSERT_R( size <= sizeof( buffer ), goto fin );
```

引　数	in_Condition	真になるべき条件式
	in_StatementsForError	エラー時に実行する文。 ; で区切った複文も可能
リターン値	なし	

(8) ASSERT_D

概　要　デバッグ版のみ有効な、ASSERT_R。

ヘッダー　r_ospl.h

宣　言　#define　ASSERT_D(Condition, StatementForError)

説　明　リリース版では、何もしなくなります。
リリース版は、標準ライブラリー同様、R_OSPL_NDEBUG マクロが定義されている場合です。

引　数	Condition	真になるべき条件式
	StatementsForError	エラー時に実行する文。 ; で区切った複文も可能
リターン値	なし	

(9) R_STATIC_ASSERT

概　要　静的に条件を満たしていないときにコンパイルエラーにする表明（契約プログラミング）を行う。

ヘッダー　r_ospl.h
宣　言　#define　R_STATIC_ASSERT(ConstantExpression, StringLiteral)
説　明　グローバルスコープでは、R_STATIC_ASSERT_GLOBAL を使用してください。
C++0x の static_assert と同じですが、GSCE の命名規則に違反するため、OSPL では別のシンボルに変更しています。
アセンブリ命令は何も出力されません。

引　数		
	ConstantExpression	条件式 （コンパイル時に答が出る式であること）
	StringLiteral	"" を指定してください。 無視されます
リターン値	なし	

(10) R_STATIC_ASSERT_GLOBAL

概　要　グローバルスコープでの R_STATIC_ASSERT
ヘッダー　r_ospl.h
宣　言　#define　R_STATIC_ASSERT_GLOBAL(ConstantExpression, StringLiteral)
説　明　アセンブリ命令は何も出力されません。

引　数		
	ConstantExpression	条件式 （コンパイル時に答が出る式であること）
	StringLiteral	"" を指定してください。 無視されます
リターン値	なし	

(11) R_NOOP

概　要　何もしない関数。
ヘッダー　r_ospl.h
宣　言　void　R_NOOP();
説　明　ASSERT_R の StatementsForError 引数に指定できます。

引　数		
	なし	
リターン値	なし	

(12) R_OSPL_MergeErrNum

概　要　エラーが発生しているときでも実行する、終了処理で発生したエラーコードをマージします。
ヘッダー　r_ospl.h
宣　言　errnum_t　R_OSPL_MergeErrNum(errnum_t CurrentError, errnum_t AppendError);
説　明　エラーが発生しているときに別のエラーが発生したら、そのエラーコードは破棄されます。
CurrentError != 0 のときは、CurrentError が返ります。
CurrentError == 0 のときは、AppendError が返ります。
ユーザーが内容を変更できる関数です。

サンプル

```
ee= Sample();
e= R_OSPL_MergeErrNum( e, ee );
return  e;
```

引　数		
	errnum_t CurrentError	今までのエラーコード
	errnum_t AppendError	終了処理のエラーコード
リターン値	マージしたエラーコード	

(13) R_OSPL_SetErrNum

概　要　TLS（スレッド ローカル ストレージ）に、エラーコードを格納します。

ヘッダー　r_ospl.h
宣　言　void　R_OSPL_SetErrNum(errnum_t e);
説　明　通常、エラーコードは、返り値で通知しますが、それができない API 関数は、R_OSPL_SetErrNum 関数でエラーコードを設定する API 関数の仕様にすることができます。
R_OSPL_TLS_ERROR_CODE マクロを 1 に設定したときのみ本関数は存在します。
すでに TLS にエラーコードが格納されていたら、何もしません。
R_OSPL_SetErrNum 関数を呼び出しただけでは、エラー状態にはなりません。参考：R_OSPL_GET_ERROR_ID

引　数	errnum_t e	発生したエラーのエラーコード
リターン値	なし	

(14) R_OSPL_GetErrNum

概　要　TLS（スレッド ローカル ストレージ）に、格納されているエラーコードを返します。
ヘッダー　r_ospl.h
宣　言　errnum_t　R_OSPL_GetErrNum();
説　明　通常、エラーコードは、返り値で通知しますが、それができない API 関数は、R_OSPL_GetErrNum 関数でエラーコードを取得できる API 関数の仕様である可能性があります。
R_OSPL_TLS_ERROR_CODE マクロを 1 に設定したときのみ本関数は存在します。R_OSPL_CLEAR_ERROR を呼び出すと、0 が返るようになります。

引　数	なし	
リターン値	エラーコード	

(15) R_OSPL_CLEAR_ERROR

概　要　エラー状態を解消します。
ヘッダー　r_ospl.h
宣　言　void　R_OSPL_CLEAR_ERROR();
説　明　R_OSPL_ERROR_BREAK マクロが 0 のとき、かつ、R_OSPL_TLS_ERROR_CODE マクロが 0 のときは、本関数を呼び出しても、何もしません。
ただし、R_OSPL_ERROR_BREAK マクロが 0 のライブラリーと 1 のアプリケーションをリンクするとき、アプリケーションにコールバックした関数の中で発生した エラーをライブラリーがクリアする必要があるため、R_OSPL_CLEAR_ERROR 関数の呼び出しがなくなることはありません。

以下は、R_OSPL_ERROR_BREAK マクロを 1 に設定したときのみ有効です。

エラー状態かどうかは、スレッド ローカル ストレージに格納されます。
本関数を呼び出すと、R_OSPL_GetErrNum 関数が 0 を返すようになります。

エラーをクリアしないまま動作させた場合、下記の影響があります。
- R_DEBUG_BREAK_IF_ERROR マクロでブレークするようになる
- 何番目のエラーであるかをカウントアップしなくなるため、R_OSPL_SET_BREAK_ERROR_ID 関数の使い勝手が悪くなる。

引　数	なし	
リターン値	なし	

(16) R_OSPL_NOTIFY_ERROR

概　要　　エラーを他のスレッドに通知して、現在のスレッドのエラー状態を解消します。

ヘッダー　　r_ospl.h

宣　言　　void　R_OSPL_NOTIFY_ERROR(errnum_t*　in_out_ErrorNumInOtherThread, errnum_t　in_NewErrorNum);

説　明　　本関数は、*in_out_ErrorNumInOtherThread = in_NewErrorNum という単純な代入により通知を行います。 スレッドの内部ステータスにはアクセスしません。

R_OSPL_DEBUG_TOOL と R_OSPL_ERROR_NOTIFICATION_WATCH を 1 に設定すると、in_NewErrorNum != 0 のときに Int ログにエラーコードが記録されます。

サンプル：

```
r_ospl_async_t  async;

R_OSPL_NOTIFY_ERROR( /*Set*/ &async.ReturnValue,  e );
```

R_OSPL_ERROR_NOTIFICATION_WATCH を 0 に設定したときは、上記のサンプルは、下記のコードと同じです。

サンプル：

```
r_ospl_async_t  async;

async.ReturnValue = e;
if ( e != 0 ) {
    R_OSPL_CLEAR_ERROR();
}
```

引　数

errnum_t* in_out_ErrorNumInOtherThread	現在のスレッドのエラーコードを格納する変数のアドレス
errnum_t　in_NewErrorNum	現在のスレッドのエラーコード

リターン値　　なし

(17) R_OSPL_SET_BREAK_ERROR_ID

概　要　　エラーが発生した瞬間にブレークするように設定します。

ヘッダー　　r_ospl.h

宣　言　　void　R_OSPL_SET_BREAK_ERROR_ID(int_fast32_t ErrorID);

説　明　　R_OSPL_ERROR_BREAK マクロが 0 のときは、本関数を呼び出しても、何もしません。エラーが発生している状態かどうかは、errnum_t 型変数の値を確認してください。

以下は、R_OSPL_ERROR_BREAK マクロを 1 に設定したときのみ有効です。

ブレークさせるには、R_DebugBreak 関数にブレークポイントを張る必要があります。
設定すべき番号は R_DEBUG_BREAK_IF_ERROR マクロまたは R_OSPL_GET_ERROR_ID マクロから知ることができます。
マルチスレッドの環境では、すべてのスレッドで発生したエラーの順番の番号を指定します。

起動して初めて発生したエラーでブレークするように設定するには、次のようにしてください。

```
R_OSPL_SET_BREAK_ERROR_ID( 1 );
```

何度かエラー復帰した後で、次に発生したエラーでブレークするように設定するには、次のようにしてください。

```
R_OSPL_SET_BREAK_ERROR_ID( R_OSPL_GET_ERROR_ID() + 1 );
```

引　数	int_fast32_t　ErrorID	何番目のエラーでブレークさせるか
リターン値	なし	

(18) R_OSPL_GET_ERROR_ID

概　要	現在発生中のエラーについて、エラーが発生した順番の番号を返します。
ヘッダー	r_ospl.h
宣　言	int_fast32_t　R_OSPL_GET_ERROR_ID();
説　明	R_OSPL_ERROR_BREAK マクロが 0 のときは、本関数を呼び出しても、何もしません。以下は、R_OSPL_ERROR_BREAK マクロを 1 に設定したときのみ有効です。 一度もエラーが発生していなかったら、 0 が返ります。 初めてエラーが発生したら 1 が返ります。 その状態から、R_OSPL_CLEAR_ERROR を呼び出し、その後でエラーが発生したら 2 が返ります。 R_OSPL_CLEAR_ERROR によってエラーがクリアされても 0 が返るようにはなりません。 エラーが発生した順番の番号は、システム全体（全スレッド）の通し番号です。 次のマクロによって、エラーが発生します。 IF, IF_D, ASSERT_R, ASSERT_D エラーが発生した順番の番号を、R_OSPL_SET_BREAK_ERROR_ID マクロに設定しておけば、エラーが発生した瞬間にブレークできるようになります。

引　数	なし	
リターン値	何番目のエラーが発生しているか	

(19) R_OSPL_DEBUG_WORK_SIZE

概　要	デバッグ用ワーク領域のサイズを計算します。
ヘッダー	r_ospl.h
宣　言	#define　R_OSPL_DEBUG_WORK_SIZE(int_fast32_t ThreadMaxCount)
説　明	ThreadMaxCount は、割込みコンテキストの分も１つとして含めます。 たとえば、スレッドが最大２のときは、割込みコンテキストの分も合わせて、３を指定します。

引　数	int_fast32_t ThreadMaxCount	スレッドの最大数＋割込みのレベル数
リターン値	デバッグ用ワーク領域のサイズ（バイト）	

(20) R_OSPL_GetCurrentThreadError

概　要	現在のスレッドに対するエラーのデバッグ情報を返します。
ヘッダー	r_ospl.h
宣　言	r_ospl_error_t*　R_OSPL_GetCurrentThreadError(void);
説　明	返り値のメンバー変数は変更しないでください。

引　数	なし	
リターン値	デバッグ情報	

(21) R_OSPL_FreeCurrentThreadError

概　要	現在のスレッドに対するエラーのデバッグ情報を格納する領域を開放します。
ヘッダー	r_ospl.h
宣　言	void　R_OSPL_FreeCurrentThreadError(void);
説　明	OSPL のエラー処理に関する API を呼び出したスレッドの数が、R_OSPL_DEBUG_THREAD_COUNT を超えたときに、R_OSPL_GetCurrentThreadError 関数でエラーになりますが、本関数を呼び出さなければ、数が増えるだけになります。 スレッドの最後で本関数を呼び出してください。 本関数を呼び出すと、R_OSPL_EVENT_Allocate 関数でイベントを確保したかどうかをチェックする情報も消え、領域を確保していない状態になります。 現在のスレッドにエラーのデバッグ情報が関連付けられていないときは、本関数は何もしません。

引　数	なし	
リターン値	なし	

(22) R_OSPL_CHANGE_THREAD_LOCKED_COUNT

概　要	現在のスレッドだけがアクセスできるカウンターの値を変更します。
ヘッダー	r_ospl.h
宣　言	void　R_OSPL_CHANGE_THREAD_LOCKED_COUNT(r_ospl_thread_id_t ThreadID,　int_fast32_t Plus);
説　明	ドライバーが本関数を呼び出します。 OSPL 内部から呼び出しません。 R_OSPL_ERROR_BREAK マクロを 1 に設定したときのみ有効です。 R_OSPL_ERROR_BREAK マクロが 0 のときは、何もしません。

引　数	r_ospl_thread_id_t ThreadID	スレッド ID
	int_fast32_t Plus	カウンター値を増加する値。マイナスなら減少する
リターン値	なし	

(23) R_OSPL_GET_THREAD_LOCKED_COUNT

概　要	現在のスレッドだけがアクセスできるカウンターの値を返します。
ヘッダー	r_ospl.h
宣　言	int_fast32_t　R_OSPL_GET_THREAD_LOCKED_COUNT(r_ospl_thread_id_t ThreadID);
説　明	R_OSPL_ERROR_BREAK マクロを 1 に設定したときのみ有効です。 R_OSPL_ERROR_BREAK マクロが 0 のときは、0 を返します。

引　数	r_ospl_thread_id_t ThreadID	スレッド ID
リターン値	ロックしているオブジェクトのカウンター値	

(24) R_OSPL_GET_STACK_POINTER

概　要	デバッグ用に、現在のスタック ポインターの値を返します。
ヘッダー	r_ospl.h
宣　言	uint8_t*　R_OSPL_GET_STACK_POINTER();
説　明	R_OSPL_STACK_CHECK_CODE マクロが 0 のときは、本関数は存在しません。 R_OSPL_STACK_CHECK_CODE マクロを 1 に設定したときのみ有効です。

引　数	なし	
リターン値	現在のスタック ポインターの値	

(25) R_OSPL_SET_END_OF_STACK

概　要	デバッグ用に、現在のスレッドのスタック領域の端を設定します。
ヘッダー	r_ospl.h
宣　言	void　R_OSPL_SET_END_OF_STACK(void* in_EndOfStackAddress);
説　明	本関数は、スタック領域の端にカナリア値 R_OSPL_STACK_CHECK_CANARY_VALUE を書くことも行います。 本関数が呼ばれるまで、スタックのチェックは無効になっています。 NULL を指定すると、スタックのチェックは無効になります。 本関数を呼び出した後で、必要なら R_OSPL_RESET_MIN_FREE_STACK_SIZE を呼び出してください。 OS レスでは、すべての擬似スレッドと割込みコンテキストでスタック領域を共有します。 OS ありでは、スレッドごとと割込みコンテキストでスタック領域が分かれます。 本関数は、現在のスレッド、または、現在の割込みコンテキストが使用するスタック領域に対して設定を行います。 R_OSPL_STACK_CHECK_CODE マクロが 0 のときは、本関数は存在しません。 R_OSPL_STACK_CHECK_CODE マクロを 1 に設定したときのみ有効です。

引　数	void* in_EndOfStackAddress	スタック領域の端のアドレス、または、NULL
リターン値	なし	

(26) R_OSPL_MOVE_END_OF_STACK

概　要	現在のスレッドのスタック領域の端を移動し、カナリア値を埋めます。
ヘッダー	r_ospl.h
宣　言	void　R_OSPL_MOVE_END_OF_STACK(void* in_EndOfStackAddress);
説　明	デバッグ用に、現在のスレッドのスタック領域の端を移動し、広がった範囲にカナリア値を埋めます。 標準ライブラリーの種類によっては、ヒープ領域を広げるなどの影響によってスタック領域のサイズが変化しますが、それに対応します。 R_OSPL_SET_END_OF_STACK との違いは、スタック領域の端を移動することでスタック領域が広がったときは、広がった範囲にカナリア値 R_OSPL_STACK_CHECK_CANARY_VALUE を埋め込むことも行うことです。 埋め込むことで、R_OSPL_GET_MIN_STACK_POINTER が正しく動作します。

引　数	void* in_EndOfStackAddress	新しいスタック領域の端のアドレス
リターン値	なし	

(27) R_OSPL_CHECK_STACK_OVERFLOW

概　要	スタック オーバーフローが発生したかどうかをチェックします。
ヘッダー	r_ospl.h
宣　言	void　R_OSPL_CHECK_STACK_OVERFLOW();
説　明	スタック オーバーフローが発生していたら、R_OSPL_RaiseUnrecoverable 関数を呼び出します。

スタックのチェックは、カナリア値が上書きされていないかどうかで行います。
本関数は、IF マクロや ASSERT_R マクロなどから呼ばれます。

R_OSPL_STACK_CHECK_CODE マクロが 0 のときは、本関数は存在しません。
R_OSPL_STACK_CHECK_CODE マクロを 1 に設定したときのみ有効です。

R_OSPL_SET_END_OF_STACK でスタックの端を設定しないと何もしません。

スタック オーバーフローが発生したときは、R_OSPL_SET_END_OF_STACK 関数でブレークさせると、スタック サイズを変更するコードの場所が分かります。

OS の API が呼び出せないとき（OS ありの環境で割込み禁止状態のとき）は、チェックを行いません。

引　数	なし	
リターン値	なし	

(28) R_OSPL_RESET_MIN_FREE_STACK_SIZE

概　要	現在のスレッドが持つスタックの空きサイズの最小値をリセットします。
ヘッダー	r_ospl.h
宣　言	errnum_t　R_OSPL_RESET_MIN_FREE_STACK_SIZE();
説　明	R_OSPL_SET_END_OF_STACK で指定したスタックの端から、現在のスレッドまたは割込みコンテキストのスタック ポインターの間に、R_OSPL_STACK_CHECK_CANARY_VALUE を埋めます。 R_OSPL_STACK_CHECK_CODE マクロが 0 のときは、本関数は存在しません。 R_OSPL_STACK_CHECK_CODE マクロを 1 に設定したときのみ有効です。

引　数	なし	
リターン値	エラーコード、正常=0	

(29) R_OSPL_GET_MIN_FREE_STACK_SIZE

概　要	現在のスレッドが持つスタックの空きサイズの最小値をカウントします。
ヘッダー	r_ospl.h
宣　言	size_t　R_OSPL_GET_MIN_FREE_STACK_SIZE();
説　明	スタックの空きサイズの最小値とは、R_OSPL_RESET_MIN_FREE_STACK_SIZE 関数を呼び出してから本関数を呼び出すまでの間で、最も少なくなったときのスタックの空きサイズです。 スタックの空きサイズの最小値をカウントする方法は、現在のスレッドまたは割込みコンテキストが持つスタック領域の中で、R_OSPL_STACK_CHECK_CANARY_VALUE からスタックの中の値が変わっていない領域のサイズをカウントすることです。 特定の処理を実行したときに使用したスタックのサイズを計測するときは、R_OSPL_GET_MIN_STACK_POINTER から計算してください。 (30)を参照。 R_OSPL_STACK_CHECK_CODE マクロが 0 のときは、本関数は存在しません。 R_OSPL_STACK_CHECK_CODE マクロを 1 に設定したときのみ有効です。

引　数	なし	
リターン値	現在のスレッドの最小残りサイズ	

(30) R_OSPL_GET_MIN_STACK_POINTER

概　要	今まで最も多くスタック領域を使用した位置を探します。
ヘッダー	r_ospl.h
宣　言	void*　R_OSPL_GET_MIN_STACK_POINTER();
説　明	現在のスレッドまたは割込みコンテキストが持つスタック領域のうち、今までの最も多く使用した位置を探すために、スタック領域の中をリードして、リードした値が R_OSPL_STACK_CHECK_CANARY_VALUE と異なったら、そのアドレスを返します。 スタックの使用量は、before-address から after-min-address を引くことで計算できます。ただし、before-address は、ターゲットの処理を行う前のスタック ポインターの値、after-min-address は、ターゲットの処理を行った後の最も多くスタック領域を使用した位置です。この方法は、ヒープ領域が拡張したか何かの理由で、R_OSPL_MOVE_END_OF_STACK でスタック領域の端を動かしたときでも正しく計算します。 R_OSPL_STACK_CHECK_CODE マクロが 0 のときは、本関数は存在しません。 R_OSPL_STACK_CHECK_CODE マクロを 1 に設定したときのみ有効です。 R_OSPL_SET_END_OF_STACK でスタックの端を設定しないと NULL が返ります。

引　数	なし	
リターン値	今まで最も多くスタック領域を使用した位置	

(31) R_D_Add

概　要	ウォッチする整数変数またはポインター変数を登録します。
ヘッダー	r_ospl_debug.h
宣　言	void　R_D_Add(int_fast32_t IndexNum, void* Address, uint32_t BreakValue, bool_t IsPrintf);
説　明	R_D_Add と R_D_Watch はウォッチに関する API です。 R_OSPL_DEBUG_TOOL を 1 に設定したときのみ、このデバッグツール機能が使えます。 例： `R_D_Add( 0, &var, 0xB0, true );  // 0xB0 may be not hit` R_OSPL_DEBUG_TOOL マクロが 0 のときは、本関数は存在しません。 R_OSPL_DEBUG_TOOL マクロを 1 に設定したときのみ有効です。

引　数	int_fast32_t IndexNum	ウォッチ番号、0 以上
	void* Address	ウォッチする 32 ビット整数変数またはポインター変数のアドレス
	uint32_t BreakValue	R_D_Watch を呼び出したときにブレークする変数の値
	bool_t IsPrintf	R_D_Watch を呼び出したときに値を printf 表示するかどうか
リターン値	なし	

(32) R_D_Watch

概　要	ウォッチします。
ヘッダー	r_ospl_debug.h
宣　言	void　R_D_Watch(int_fast32_t IndexNum);
説　明	R_D_Add と R_D_Watch はウォッチに関する API です。 R_OSPL_DEBUG_TOOL を 1 に設定したときのみ、このデバッグツール機能が使えます。

R_D_Add で登録した変数の値を printf 表示するか、指定しておいた値になったらブレークします。
本関数は、登録している変数のスコープ外から呼び出すこともできるため、スコープを気にすることなく、多くの場所に呼び出すコードをばらまくことができます。
例：

```
R_D_Watch( 0 );
printf( "Line: %d", __LINE__ );  // ここの場所を示す値を表示
```

R_OSPL_DEBUG_TOOL マクロが 0 のときは、本関数は存在しません。
R_OSPL_DEBUG_TOOL マクロを 1 に設定したときのみ有効です。

デバッグ用ツールの内部データが破壊されるときは、r_ospl_debug.o オブジェクトファイルのグローバル変数を、安全なアドレス（メモリーマップ）に配置してください。

引　数	int_fast32_t IndexNum	ウォッチ番号、0 以上
リターン値	なし	

(33) R_D_AddToIntLog

概　要　高速でログに記録します。
ヘッダー　r_ospl_debug.h
宣　言　void　R_D_AddToIntLog(int_fast32_t Value);
説　明　R_D_AddToIntLog と g_IntLog と g_IntLogLength は Int ログに関する API です。
R_OSPL_DEBUG_TOOL を 1 に設定したときのみ、このデバッグツール機能が使えます。
g_IntLog の最大要素数を超えたときは、先頭からログを上書きします。
表示したい変数の値を記録するだけでなく、コードの場所を識別するための値や、現在の時間も記録するとよいでしょう。

g_IntLog と g_IntLogLength と g_DebugVar は外部リンケージのグローバル変数です。g_DebugVar 変数は R_D_AddToIntLog 関数がアクセスしないグローバル変数です。スコープを超えた変数を参照する条件付きブレークをするときに使えます。また、CPU が最後に通過した位置を調べるときにも使えます。

```
int_fast32_t   g_IntLog[100];
int_fast32_t   g_IntLogLength;
int_fast32_t   g_DebugVar[10];
```

R_OSPL_DEBUG_TOOL マクロが 0 のときは、本関数と上記のグローバル変数は存在しません。g_IntLog と g_IntLogLength 変数もなくなります。
R_OSPL_DEBUG_TOOL マクロを 1 に設定したときのみ有効です。

デバッグ用ツールの内部データが破壊されるときは、r_ospl_debug.o オブジェクトファイルのグローバル変数を、安全なアドレス（メモリーマップ）に配置してください。

引　数	int_fast32_t Value	記録する値
リターン値	なし	

(34) R_D_Counter

概　要　通過する回数をカウントし、指定した回数になったかどうかを返します。
ヘッダー　r_ospl_debug.h
宣　言　bool_t　R_D_Counter(int_fast32_t* in_out_Counter, int_fast32_t TargetCount, char* Label);

説　明　　R_OSPL_DEBUG_TOOL を 1 に設定したときのみ、このデバッグツール機能が使えます。
TargetCount = 0 を設定して実行すると、通過するたびに通過回数を printf 出力します。通過回数を確認し、TargetCount に確認した回数を指定してプログラムを再起動すると、指定したカウンター値になったときに true を返します。printf 出力が多すぎるときは、Label = NULL にします。 1 回目だけカウンターのアドレスが printf 出力されるので、デバッガーでカウンター値を確認してください。

例：
```
{ static int tc;  if ( R_D_Counter( &tc, 0, "A" ) ) {
R_DEBUG_BREAK(); }}
```

引　数		
	int_fast32_t* in_out_Counter	（入出力） カウンター
	int_fast32_t TargetCount	カウンターと比較する値
	char* Label	printf 出力するラベル、 NULL=printf 出力しない
リターン値	カウンターの値が TargetCount になったかどうか	

4.6.19　静的コード解析ツールのレビュー済みタグ

(1)　IS

概　要　　MISRA 13.2 の警告に対処したコードを読みやすくします。
ヘッダー　　r_static_an_tag.h
宣　言　　#define　IS(bool_value)
説　明　　「0 以外」という二重否定表現を回避します。
IS マクロは、言語仕様に bool 型があればその型にキャストすることに相当します。
例：
```
bool_t  is_condition = ( x > 0 );
if ( IS( is_condition ) ) { ... }
/* if ( is_condition != 0 ) ) { ... } */
```

静的コード解析ツールに警告されてから記述してください。

引　数	bool_value	評価結果がブール型になる式
リターン値	bool_value != 0 の結果	

(2)　R_OSPL_ReturnFalse

概　要　　if ブロックを無効にするときに警告されるときの対策。
ヘッダー　　r_ospl.h
宣　言　　int_t　R_OSPL_ReturnFalse(void);
説　明
```
#define  IF_D( Condition )  if ( false )
```
のように記述して、警告されるときは、
```
#define  IF_D( Condition )  if ( R_OSPL_ReturnFalse() )
```
のようにします。

引　数	なし	
リターン値	0	

(3)　R_UNREFERENCED_VARIABLE

概　要　　変数が参照されていなという警告を発生させないようにします。
ヘッダー　　r_static_an_tag.h
宣　言　　#define　R_UNREFERENCED_VARIABLE(Variable)
説　明　　警告されてから記述してください。

引　数	Variable	警告を発生させないようにする変数
リターン値	なし	

(4)　R_UNREFERENCED_VARIABLE2

概　要　複数の変数が指定できる R_UNREFERENCED_VARIABLE です。
ヘッダー　r_static_an_tag.h
宣　言　#define　R_UNREFERENCED_VARIABLE2(Variable, Variable2)
説　明

引　数	Variable	警告を発生させないようにする変数
	Variable2	警告を発生させないようにする変数
リターン値	なし	

(5)　R_UNREFERENCED_VARIABLE3

概　要　複数の変数が指定できる R_UNREFERENCED_VARIABLE です。
ヘッダー　r_static_an_tag.h
宣　言　#define　R_UNREFERENCED_VARIABLE3(Variable, Variable2, Variable3)
説　明

引　数	Variable	警告を発生させないようにする変数
	Variable2	警告を発生させないようにする変数
	Variable3	警告を発生させないようにする変数
リターン値	なし	

(6)　R_UNREFERENCED_VARIABLE4

概　要　複数の変数が指定できる R_UNREFERENCED_VARIABLE です。
ヘッダー　r_static_an_tag.h
宣　言　#define　R_UNREFERENCED_VARIABLE4(Variable, Variable2, Variable3, Variable4)
説　明

引　数	Variable	警告を発生させないようにする変数
	Variable2	警告を発生させないようにする変数
	Variable3	警告を発生させないようにする変数
	Variable4	警告を発生させないようにする変数
リターン値	なし	

4.6.20　マルチコンパイラー対応

(1)　R_OSPL_SECTION

概　要　関数や変数にセクション名を付けます。
ヘッダー　r_ospl.h
宣　言　#define　R_OSPL_SECTION(SectionName, Declaration)
#define　R_OSPL_SECTION_FOR_ZERO_INIT(SectionName, Declaration)
#define　R_OSPL_SECTION_FOR_INLINE(SectionName, Declaration)
説　明　メモリーマップ上のどこに関数や変数を置くかを設定するために、関数や変数にセクション名を付けます。
初期値は、R_OSPL_SECTION マクロの外に記述してください。
R_OSPL_SECTION_FOR_ZERO_INIT は、初期値なし変数に使います。
R_OSPL_SECTION_FOR_INLINE は、インライン関数に使います。

サンプル：

```
int  NormalFunction( int a )
```

```
{
    return  a + 1;
}

R_OSPL_SECTION( "CODE_BASIC_SECTION",
int  SectionFunction( int a )
)
{
    return  a + 1;
}

R_OSPL_SECTION( "DATA_BASIC_SECTION",
int  g_Variable_Data[100]
)
= { 1 };
```

引　数	SectionName	セクション名
	Declaration	関数または変数の宣言
リターン値	関数または変数の宣言	

(2)　R_OSPL_ALIGNMENT

概　要　　グローバル変数の先頭アドレスをアラインメントします。

ヘッダー　　r_ospl.h

宣　言　　#define　R_OSPL_ALIGNMENT(ByteCount, Declaration)

説　明　　初期値は、R_OSPL_ALIGNMENT マクロの外に記述してください。

サンプル：

```
R_OSPL_ALIGNMENT( 0x100,
extern const int  g_Variable_Const2[4] );

R_OSPL_ALIGNMENT( 0x100,
const int  g_Variable_Const2[4] ) = { 0x01, 0x02, 0x03,
0x04 };
```

引　数	ByteCount	アラインメントの値（バイト）
	Declaration	変数の宣言
リターン値	変数の宣言	

(3)　R_COUNT_OF

概　要　　配列の要素数を返します。

ヘッダー　　r_ospl.h

宣　言　　#define　R_COUNT_OF(Array)

説　明　　サンプル：

```
uint32_t  array[10];
R_COUNT_OF( array )  // = 10
```

Array 引数に配列として使っているポインターを指定しないでください。

サンプル：

```
uint32_t   array[10];
func( array );

void  func( uint32_t array[] )  /* array はポインター */
{
    R_COUNT_OF( array )  // NG
}
```

引　数	Array	配列
リターン値	配列の要素数	

(4)　INLINE

INLINE マクロは、C99 の inline 仕様（static も extern もない inline）と同じ仕様です。

(5)　STATIC_INLINE

STATIC_INLINE マクロは、C99 の static inline 仕様と同じ仕様です。

(6)　R_ADDRESS_Add

概　要　　アドレス（ポインター）の値をバイト単位で加算します。

ヘッダー　　r_ospl.h

宣　言　　void*　R_ADDRESS_Add(void*　in_BaseAddress,　int_fast32_t　in_Offset);

説　明　　
```
R_ADDRESS_Add( base_address, +offset )
```
は、次の計算を行います。
```
( ( (uint8_t*) base_address ) + offset )
```
ポインターの型によらず、in_Offset 引数はバイト単位になります。
uint8_t* 型から uint8_t* 型へのキャストが冗長に感じなくなるため、キャストのし忘れを防ぎます。
MISRA-C 2004-17.4 では配列で記述することが求められますが、バイト単位で加算ができません。
返り値のアドレスを参照する前に、配列番号と同様に、アクセス可能な範囲のチェックが必要です。

引　数	void*　in_BaseAddress	加算される前のアドレス
	int_fast32_t　in_Offset	加算する値。マイナス可能
リターン値	加算された後のアドレス	

(7)　R_OSPL_CountLeadingZeros

概　要　　最上位ビット(MSB)から 0 になっているビットの数を数えます。

ヘッダー　　r_ospl.h

宣　言　　int_fast32_t　R_OSPL_CountLeadingZeros(bit_flags32_t　in_BitFlags);

説　明　　in_BitFlags 引数 = 0 のとき、返り値は 32 です。
コンパイラー固有関数に存在する環境では、R_OSPL_CountLeadingZeros はマクロになります。
サンプル：
```
R_OSPL_CountLeadingZeros( 0x20000000 ) ==  2;
R_OSPL_CountLeadingZeros( 0x02000000 ) ==  6;
R_OSPL_CountLeadingZeros( 0x00000000 ) == 32;
int_fast32_t  leading_bit_num = 31 -
    R_OSPL_CountLeadingZeros( flags );
```

引　数	bit_flags32_t　in_BitFlags	調べる対象となるビットフラグの値
リターン値	ビットの数	

(8)　R_OSPL_IsSetBitsCount1

概　要　　値が 1 になっているビットの数が 1 つかどうかを返します。

ヘッダー　　r_ospl.h

宣　言　　bool_t　R_OSPL_IsSetBitsCount1(uint32_t　in_BitFlags);

説　明　　サンプル：

```
R_OSPL_IsSetBitsCount1( 0x00000001 ) == true;
R_OSPL_IsSetBitsCount1( 0x00000002 ) == true;
R_OSPL_IsSetBitsCount1( 0x00000003 ) == false;
R_OSPL_IsSetBitsCount1( 0x00000000 ) == false;
```

引　数	bit_flags32_t　in_BitFlags	調べる対象となるビットフラグの値
リターン値	1 になっているビットの数が 1 つかどうか	

4.6.21　OSPL の下位層に関する関数

(1)　R_DebugBreak

概　要　ブレークするときに、OSPL からコールバックされる関数。

ヘッダー　r_ospl.h

宣　言　void　R_DebugBreak(char_t* File, int_fast32_t Line);

説　明　本関数にブレークポイントを張ってください。
リリース版では、File = NULL、Line = 0 になります。
File = NULL のときは、Line はエラーコードになります。
アプリケーション開発者が直接本関数を修正することができます。
本関数の中から、OSPL のエラー処理の機能を使用しないでください。

もし、デバッグ環境や printf 出力するシリアルが使えなくても、LED、または GPIO とオシロスコープが使えれば、シリアル出力を使わなくても情報を得ることができます。その方法は、R_DebugBreak 関数の中から LED を制御する関数と一定時間待つ関数（R_OSPL_Delay など）を呼び出し、モールス信号のように R_DebugBreak 関数の引数を表示することです。 たとえば、次のように表示します。

- 開始は、１秒点灯、１秒消灯
- １なら 0.5 秒点灯 0.5 秒消灯の表示
- ０なら 0.2 秒点灯 0.8 秒消灯の表示
- シフト演算で次の桁の１／０を表示していき、２進数を表示する

引　数	char_t*　File	ブレークしているファイル名、または NULL
	int_fast32_t　Line	ブレークしている行番号、または エラーコード
リターン値	なし	

(2)　R_OSPL_OnIdleDefault

概　要　デフォルトのアイドル状態コールバック関数

ヘッダー　r_ospl.h

宣　言　errnum_t　R_OSPL_OnIdleDefault(r_ospl_idle_event_t IdleEvent);

説　明　アイドル状態のときに何度もコールバックされます。

R_OSPL_CPU_LOAD を 1 にして、R_OSPL_IDLE_Start_CPU_Load 関数を呼び出すと、本関数から定期的に R_OSPL_IDLE_Print_CPU_Load 関数を呼び出して CPU 使用率を printf 表示するようになります。 ただし、待ちに入る API を呼び出さなかったときは、定期的には表示されません。

アプリケーション開発者が本関数をカスタマイズすることができます。

引　数	r_ospl_idle_event_t IdleEvent	アイドル状態に関するイベントの種類
リターン値	エラーコード。エラーなし＝0	

(3)　R_OSPL_Start_T_Lock

概　要	OSPL 内部から T-ロックを開始するときにコールバックされる関数
ヘッダー	-
宣　言	errnum_t　R_OSPL_Start_T_Lock(void);
説　明	OSPL 内部の１つの同期オブジェクトを使用してロックします。 アプリケーションから呼び出すことはできません。 本関数の中から、OSPL のエラー処理の機能を使用しないでください。 T-ロックが開始されている状態で、本関数がコールバックされることはありません。

引　数	なし	
リターン値	エラーコード。エラーなし＝0	

(4)　R_OSPL_End_T_Lock

概　要	OSPL 内部から T-ロックを終了するときにコールバックされる関数
ヘッダー	-
宣　言	void　R_OSPL_End_T_Lock(void);
説　明	アプリケーションから呼び出すことはできません。 本関数の中から、OSPL のエラー処理の機能を使用しないでください。 T-ロックが開始されていない状態で、本関数がコールバックされることはありません。

引　数	なし	
リターン値	なし	

(5)　R_OSPL_EVENT_GROUP_Create

概　要	イベント グループを生成します。
ヘッダー	r_ospl.h
宣　言	errnum_t　R_OSPL_EVENT_GROUP_Create(volatile r_ospl_event_group_id_t* out_EventGroupId,　r_ospl_thread_id_t　in_ThreadId);
説　明	アプリケーションから呼び出すことはできません。 R_OSPL_EVENT_GROUP_CODE = 1 のときのみ、定義があります。 参考：r_ospl_event_group_id_t　(4.5.4)

引　数	r_ospl_event_group_id_t * out_EventGroupId	（出力） 生成されたイベント グループの ID
	r_ospl_thread_id_t in_ThreadId	生成されたイベント グループを待つスレッドの ID
リターン値	エラーコード。エラーなし＝0	

(6)　R_OSPL_EVENT_GROUP_Delete

概　要	イベント グループを削除します。
ヘッダー	r_ospl.h
宣　言	errnum_t　R_OSPL_EVENT_GROUP_Delete(volatile r_ospl_event_group_id_t in_EventGroupId,　r_ospl_thread_id_t　in_ThreadId);
説　明	アプリケーションから呼び出すことはできません。 R_OSPL_EVENT_GROUP_CODE = 1 のときのみ、定義があります。

引　数	r_ospl_event_group_id_t in_EventGroupId	イベント グループの ID
	r_ospl_thread_id_t in_ThreadId	in_EventGroupID 引数に指定したイベント グループを待つスレッドの ID
リターン値	エラーコード。エラーなし＝0	

4.6.22　ドライバーの API に共通する関数仕様

(1)　R_DRIVER_Transfer

概　要	周辺機能が持つ非同期処理を同期的に処理します。
ヘッダー	（ドライバーのヘッダー）
宣　言	errnum_t　R_DRIVER_Transfer(int_fast32_t ChannelNum, ...);
説　明	A-スレッドから呼び出します。 R_DRIVER_TransferStart 関数の同期版（ブロッキング関数）です。 内部で待ちが入りますが、OS ありのときと、OS レスの環境で他の（擬似）スレッドが動く必要がないときに、使用できます。 R_DRIVER_Transfer は、プレースホルダーです。 実際の関数名は、ドライバーによります。例： R_FS_Read、R_VSYNC_Wait、R_TOUCH_Read CPU と並列にハードウェアや通信などが動く、非同期処理を起動し、続きの処理をすぐに行えるように、処理の完了まで本関数の内部で待ちます。もしくは、入力待ちをします。 内部で使用するイベントは、内部で R_OSPL_EVENT_Allocate 関数を呼び出して確保します。これにより、並列して動作する別の処理のイベントと衝突していないかチェックを行います。 OS レスの場合は、待っている間に r_ospl_idle_callback_t 型の関数を何度もコールバックします。 OS なしの場合や、OS のスレッド数（スタックメモリーの総量など）を減らしたいときは、R_DRIVER_TransferStart 関数を使用して、擬似マルチスレッドを構成してください。

引　数		
	int_fast32_t ChannelNum	チャンネル番号、API 関数によって変えてもよい
	...	処理のパラメーター、API 関数によって変えてもよい
リターン値	エラーコード。エラーなし＝0、API 関数によって変えてもよい	

(2)　R_DRIVER_TransferStart

概　要	周辺機能が持つ非同期処理を起動します。
ヘッダー	（ドライバーのヘッダー）
宣　言	errnum_t　R_DRIVER_TransferStart(int_fast32_t ChannelNum, ..., r_ospl_async_t* Async);
説　明	A-スレッドから呼び出します。 R_DRIVER_Transfer 関数の非同期版（ノンブロッキング関数）です。 R_DRIVER_TransferStart は、プレースホルダーです。 実際の関数名は、ドライバーによります。例： R_FS_ReadStart, R_SOUND_Play, R_VSYNC_WaitStart、R_TOUCH_ReadStart 非同期処理の完了を受信する方法については、r_ospl_async_t 型構造体の説明を参照してください。 **非同期処理が完了したら、r_ospl_async_t::ReturnValue メンバー変数の値をチェックしてください。**

非同期処理が終わるまで、Async 引数の構造体の領域を保持し、A-イベントや I-イベントに指定したフラグは他の目的に使わないでください。
Async 引数に指定した構造体を保持している間は、他の非同期処理の Async 引数に指定しないでください。
ドライバーによっては、別の Async 構造体変数を Async 引数に指定すれば、複数の非同期処理の要求を開始することができるかもしれません。できなければエラーになります。

入力装置（ハードウェア）から連続した入力をロストしないよう、R_DRIVER_TransferStart 関数で入力の受付を開始した後で入力（A-イベント）を受信しても、継続して非同期入力処理が行われる仕様のドライバーが考えられます。 その場合は、入力の受付を終了するまで、A-イベントを受信した後も Async 引数に指定した構造体を保持し続け、A-イベントや I-イベントに指定したフラグは他の目的に使わないでください。

参考：4.7.1、4.7.4 シーケンス図

（本関数は、旧、R_DRIVER_TransferAsync 関数です。）

引　数	int_fast32_t ChannelNum	チャンネル番号、API 関数によって変えてもよい
	...	処理のパラメーター、API 関数によって変えてもよい
	r_ospl_async_t*　Async	（入出力） 通知設定、NULL 不可
リターン値	エラーコード。エラーなし＝0、API 関数によって変えてもよい	

(3)　R_DRIVER_OnInterrupting

概　要　　割込みを受信します。
ヘッダー　　（ドライバーのヘッダー）
宣　言　　errnum_t　R_DRIVER_OnInterrupting(const r_ospl_interrupt_t* InterruptSource);
説　明　　本関数は、割込みステータスレジスターから、r_driver_async_status_t::InterruptFlags 変数に割込み通知を伝達し、割込みをクリアします。

R_DRIVER_OnInterrupting は、プレースホルダーです。 実際の関数名は、ドライバーによります。

通常、ドライバーが提供するデフォルトの割込みコールバック関数から本関数が自動的に呼び出されます。

割込み応答処理は、通常 R_DRIVER_OnInterrupted 関数の中で行われますが、本関数で割込み応答処理も行われるドライバーの仕様もあります。

引　数	r_ospl_interrupt_t* InterruptSource	割込み発信元、API 関数によって変えてもよい
リターン値	エラーコード。エラーなし＝0、API 関数によって変えてもよい	

(4)　R_DRIVER_OnInterrupted

概　要　　割込み応答処理を行います。
ヘッダー　　（ドライバーのヘッダー）
宣　言　　errnum_t　R_DRIVER_OnInterrupted(int_fast32_t ChannelNum);

説　明　　R_DRIVER_OnInterrupting 関数によって 1 に設定された r_driver_async_status_t::InterruptFlags 変数のビットを 0 にクリアして、割込み応答処理を行います。

R_DRIVER_OnInterrupted は、プレースホルダーです。 実際の関数名は、ドライバーによります。

非同期処理が完了したときは、A-イベントが発生します。 A-イベントを受信したら、
r_ospl_async_t::ReturnValue メンバー変数の値をチェックしてください。

r_ospl_async_t::I_Thread == R_OSPL_THREAD_NULL に設定して非同期処理を開始したときは、ドライバーが提供するデフォルトの割込みコールバック関数から本関数が自動的に呼び出されます。

r_ospl_async_t::I_Thread != R_OSPL_THREAD_NULL に設定して非同期処理を開始したときは、デフォルトの割込みコールバック関数の中から I-イベントが発生します。I-イベントを受信したら、本関数を呼び出してください。

割込みが発生していないときは、本関数の中で何もせず、0（エラーなし）を返します。

割込み応答処理がほとんどないときは、本関数が提供されないことがあります。

引　数	int_fast32_t ChannelNum	チャンネル番号、API 関数によって変えてもよい
リターン値	エラーコード。エラーなし＝0、API 関数によって変えてもよい	

(5)　R_DRIVER_GetAsyncStatus

概　要　　割込みや非同期処理の状況を示す構造体へのポインターを取得します。

ヘッダー　　（ドライバーのヘッダー）

宣　言　　errnum_t　R_DRIVER_GetAsyncStatus(int_fast32_t ChannelNum, const r_driver_async_status_t** out_Status);

説　明　　R_DRIVER_GetAsyncStatus は、プレースホルダーです。 実際の関数名は、ドライバーによります。

out_Status 引数に指定するポインター変数には、const 修飾子が必要です。
スレッド、または、割込みコールバック関数のどちらからでも呼び出すことができます。未初期化の状態でも呼び出すことができます。

**out_Status の値は、アプリケーションから変更することはできませんが、ドライバーによって変更されます。メンバー変数によって、本関数を呼び出さなくても変更されるものと、呼び出さなければ変更されないものがあります。 それぞれのメンバー変数の仕様によります。

引　数	int_fast32_t ChannelNum	チャンネル番号、API 関数によって変えてもよい
	r_driver_async_status_t** out_Status	（出力） 割込みや非同期処理の状況を示す構造体へのポインター。ドライバー定義
リターン値	エラーコード。エラーなし＝0、API 関数によって変えてもよい	

(6)　R_DRIVER_Initialize

概　要　　ドライバーを初期化して使える状態にします。

ヘッダー　（ドライバーのヘッダー）

宣　言　errnum_t　R_DRIVER_Initialize(int_fast32_t ChannelNum,　r_driver_config_t in_out_Config);

説　明　ここでは、チャンネル使用管理（ロック）についてのみ説明します。

本関数を呼び出すと、初期化したチャンネルをロックします。本関数の代わりに、R_DRIVER_LockChannel 関数を呼び出すと、初期化をせずにチャンネルをロックします。 初期化をしないでロックをすると、空いているチャンネルのリストから除外され、別のドライバーを競合せずに使えるようになります。

ロックに失敗したら、E_ACCESS_DENIED エラーや E_FEW_ARRAY エラーなどを返します。

本関数を使用しているモジュールは、競合していたときやチャンネルが不足していたときに、エラーにするか、機能制限をする必要があります。

チャンネル番号を指定して初期化するには、次のようにします。

```
 e= R_DRIVER_Initialize( 1, NULL );
    IF(e){goto fin;}
```

空いているチャンネル番号を使って初期化し、そのチャンネル番号を取得するには、次のようにします。

```
    int_fast32_t      channel_num = R_OSPL_UNLOCKED_CHANNEL;
    r_driver_config_t  config;

    e= R_DRIVER_Initialize( channel_num, &config );
       IF(e){goto fin;}
    channel_num = config.ChannelNum;
```

引　数

int_fast32_t ChannelNum	ロックして初期化するチャンネル番号、または、R_OSPL_UNLOCKED_CHANNEL。 API 関数によって変えてもよい。
r_driver_config_t in_out_Config	（入出力） 初期化に関するすべての設定、NULL 可能。 API 関数によって変えてもよい。

リターン値　エラーコード。エラーなし＝0、API 関数によって変えてもよい

(7)　R_DRIVER_Finalize

概　要　ドライバーの終了処理をします。

ヘッダー　（ドライバーのヘッダー）

宣　言　errnum_t　R_DRIVER_Finalize(int_fast32_t ChannelNum,　errnum_t　e);

説　明　ここでは、チャンネル使用管理（ロック）についてのみ説明します。

本関数を呼び出すと、終了処理をして、チャンネルをロック解除します。

R_DRIVER_Initialize を呼び出したオーナー（スレッド、または、コンテキスト）と異なるオーナーから R_DRIVER_Finalize を呼び出したときは、終了処理もロック解除も行いません。

引　数

int_fast32_t ChannelNum	終了処理をしてロック解除するチャンネル番号。 API 関数によって変えてもよい。
errnum_t e	これまでに発生したエラーコード。 エラー無し=0。 API 関数によって変えてもよい

リターン値	エラーコード または e、0=「成功かつ e=0」、API 関数によって変えてもよい

(8) R_DRIVER_LockChannel

概　要	対象のユニットの中にある、チャンネルをロックします。（使用状態にします）
ヘッダー	（ドライバーのヘッダー）
宣　言	errnum_t　R_DRIVER_LockChannel(int_fast32_t const　ChannelNum, int_fast32_t* out_ChannelNum);
説　明	内部で R_OSPL_LockChannel を呼び出してロックし、本関数を持つドライバー以外でチャンネルを使えるようにします。 R_DRIVER_LockChannel 関数を持つドライバーを使うときは、通常の初期化関数（例：R_DRIVER_Initialize）を呼び出してください。 本関数は、対象となる周辺機能のチャンネルに対して、初期化をしません。 ChannelNum 引数に R_OSPL_UNLOCKED_CHANNEL(=0xFEE) を指定した場合、ロックされていないチャンネルをロックします。 このとき、out_ChannelNum に NULL を指定できません。

引　数		
	int_fast32_t ChannelNum	ロックするチャンネル番号、または、R_OSPL_UNLOCKED_CHANNEL。 API 関数によって変えてもよい。
	int_fast32_t* out_ChannelNum	（出力）ロックしたチャンネル番号、NULL 可
リターン値	エラーコード。エラーなし＝0、API 関数によって変えてもよい	

(9) R_DRIVER_UnlockChannel

概　要	対象のユニットの中にある、チャンネルをロック解除します。（未使用状態にします）
ヘッダー	（ドライバーのヘッダー）
宣　言	errnum_t　R_DRIVER_UnlockChannel(int_fast32_t const　ChannelNum, errnum_t e);
説　明	内部で R_OSPL_UnlockChannel を呼び出してロックを解除し、本関数を持つドライバーでチャンネルを使うことができるようにします。 本関数は、対象となる周辺機能のチャンネルに対して、終了処理をしません。 すでにロック解除されていたときは、E_ACCESS_DENIED エラーになります。 ChannelNum 引数が R_OSPL_UNLOCKED_CHANNEL のとき、0 未満のとき、または、チャンネル数以上だったときは、何もせず、0（エラーなし）を返します（引数 e=0 の場合）。

引　数		
	int_fast32_t ChannelNum	ロック解除するチャンネル番号。 API 関数によって変えてもよい。
	errnum_t e	これまでに発生したエラーコード。 エラー無し=0。 API 関数によって変えてもよい
リターン値	エラーコード または e、0=「成功かつ e=0」、API 関数によって変えてもよい	

4.6.23　ドライバーの下位層に共通する関数仕様

(1) R_DRIVER_SetDefaultAsync

概　要	r_ospl_async_t 型の構造体のデフォルト値を設定します。
ヘッダー	（ドライバーの下位層のヘッダー）
宣　言	void　R_DRIVER_SetDefaultAsync(r_ospl_async_t* Async, r_ospl_async_type_t AsyncType);

説　明	システムに応じて本関数によるデフォルト設定を変えることができます。 R_DRIVER_SetDefaultAsync は、プレースホルダーです。 実際の関数名は、ドライバーによります。 内部で、R_OSPL_ASYNC_SetDefaultPreset 関数を呼び出します。 例： R_FS_SetDefaultAsync	
引　数	r_ospl_async_t* Async	（入出力） 通知設定、NULL 不可
	r_ospl_async_type_t AsyncType	非同期処理の種類
リターン値	なし	

(2)　R_DRIVER_I_LOCK_Replace

概　要	I-ロックを行うオブジェクトを、統合型ドライバーの I-ロック・オブジェクトに置き換えます。	
ヘッダー	（ドライバーの下位層のヘッダー）	
宣　言	bool_t　R_DRIVER_I_LOCK_Replace(int_fast32_t ChannelNum,　void* i_lock, r_ospl_i_lock_vtable_t* i_lock_v_table);	
説　明	統合型ドライバーに統合される可能性があるドライバーが提供する API です。 本関数は統合型ドライバーから呼ばれ、アプリケーションから呼び出しません。 I-ロックによって割込みの許可や禁止を統合型ドライバーに集めます。	
引　数	int_fast32_t ChannelNum	チャンネル番号
	void* i_lock	i_lock_v_table 引数のメンバー関数の第 1 引数に渡す値、または NULL
	r_ospl_i_lock_vtable_t* i_lock_v_table	I-ロックに関する VTable、または NULL
リターン値	置き換えたかどうか	

(3)　R_DRIVER_DisableInterrupt

概　要	I-ロックを開始するために、割込みを禁止します。	
ヘッダー	（ドライバーの下位層のヘッダー）	
宣　言	bool_t　R_DRIVER_DisableInterrupt(int_fast32_t ChannelNum);	
説　明	統合型ドライバーに統合される可能性があるドライバーが提供する API です。 本関数は統合型ドライバーから呼ばれ、アプリケーションから呼び出しません。	
引　数	int_fast32_t ChannelNum	チャンネル番号
リターン値	今まで割込み許可されていたかどうか	

(4)　R_DRIVER_EnableInterrupt

概　要	I-ロックを終了するために、割込みを許可します。	
ヘッダー	（ドライバーの下位層のヘッダー）	
宣　言	void　R_DRIVER_EnableInterrupt(int_fast32_t ChannelNum);	
説　明	統合型ドライバーに統合される可能性があるドライバーが提供する API です。 本関数は統合型ドライバーから呼ばれ、アプリケーションから呼び出しません。	
引　数	int_fast32_t ChannelNum	チャンネル番号
リターン値	なし	

4.6.24　その他の関数

(1)　非推奨関数

以下の関数は、古い OSPL の仕様との互換のため、または他の仕様との互換のためにあり、使用は非推奨です。

R_OSPL_EVENT_Get、R_OSPL_EVENT_OBJECT_Create、R_BSP_InterruptsEnable、R_BSP_InterruptsDisable

(2) 半内部関数

以下の関数は、サンプル ドライバーの共通処理を抽出しただけで互換性を保証していない関数、または、内部デバッグ用にドライバーなどから呼び出せるようにしている関数です。

R_OSPL_ASYNC_CopyExceptAThread、R_OSPL_I_LOCK_LockStub、R_OSPL_I_LOCK_UnlockStub、R_OSPL_I_LOCK_RequestFinalizeStub、R_OSPL_LockCurrentThreadError、R_OSPL_UnlockCurrentThreadError

(3) Print 関数

以下の関数は、内部状態を表示し、最終製品で呼び出すことは非推奨です。

R_OSPL_TABLE_Print、R_OSPL_QUEUE_Print

4.7　シーケンス図

4.7.1　同期型 割込み応答処理（割込みコンテキスト応答版）

図 4.4 に同期型関数を使用したときの割込み応答処理のシーケンス図を示します。 OS レスでは、割込みコールバック関数（割込みコンテキスト）から、R_DRIVER_OnInterrupted 関数を呼び出します。

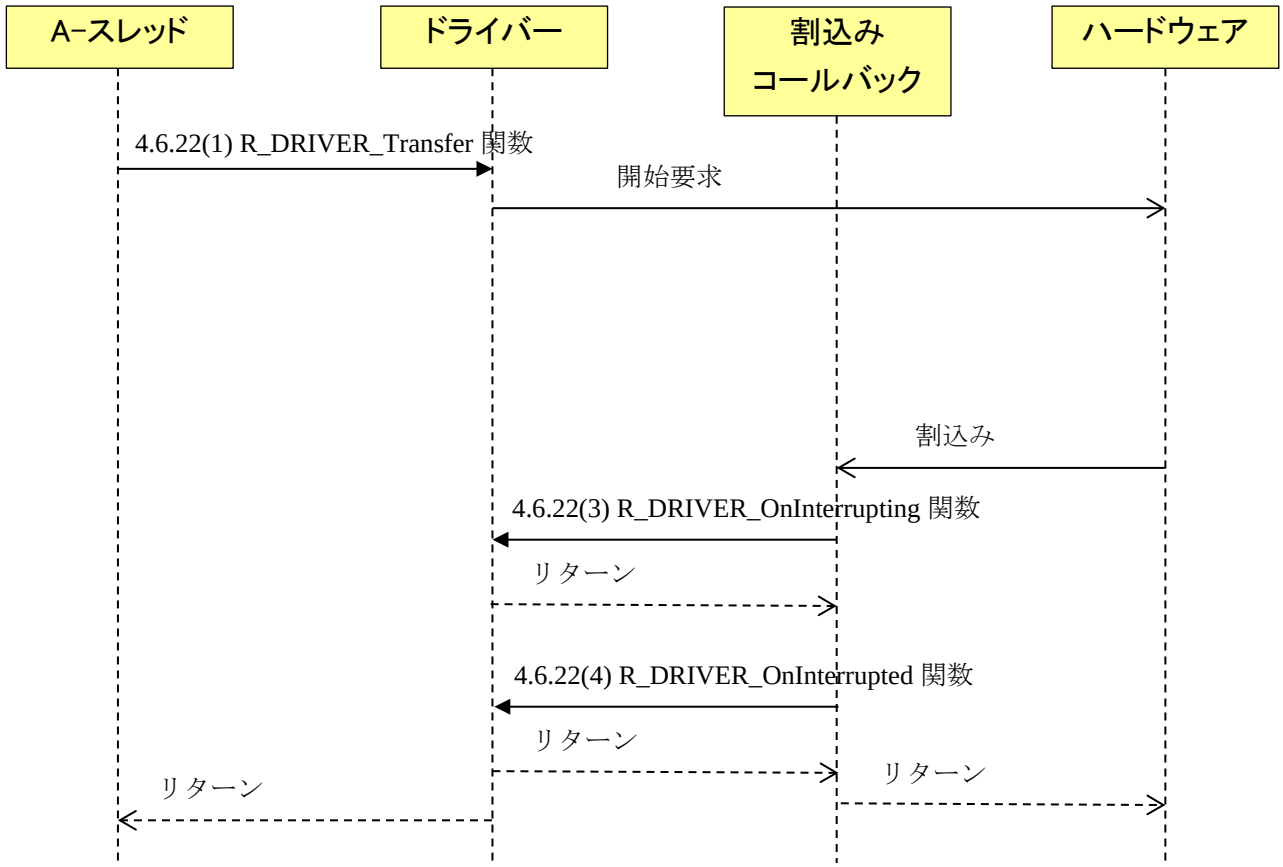


図 4.4　同期型関数を使用したときの割込み応答処理（割込みコンテキスト応答版）

4.7.2　同期型 割込み応答処理（A-スレッド応答版）

図 4.5 に同期型関数を使用したときの割込み応答処理のシーケンス図を示します。OS ありの環境で、ドライバー内の割込み応答処理が長くなるときは、他の優先度の高いスレッドが動けるように、割込みコールバック関数（割込みコンテキスト）からではなく、A-スレッドが実行している R_DRIVER_Transfer 関数の中から R_DRIVER_OnInterrupted 関数を呼び出す仕様のドライバーの場合があります。

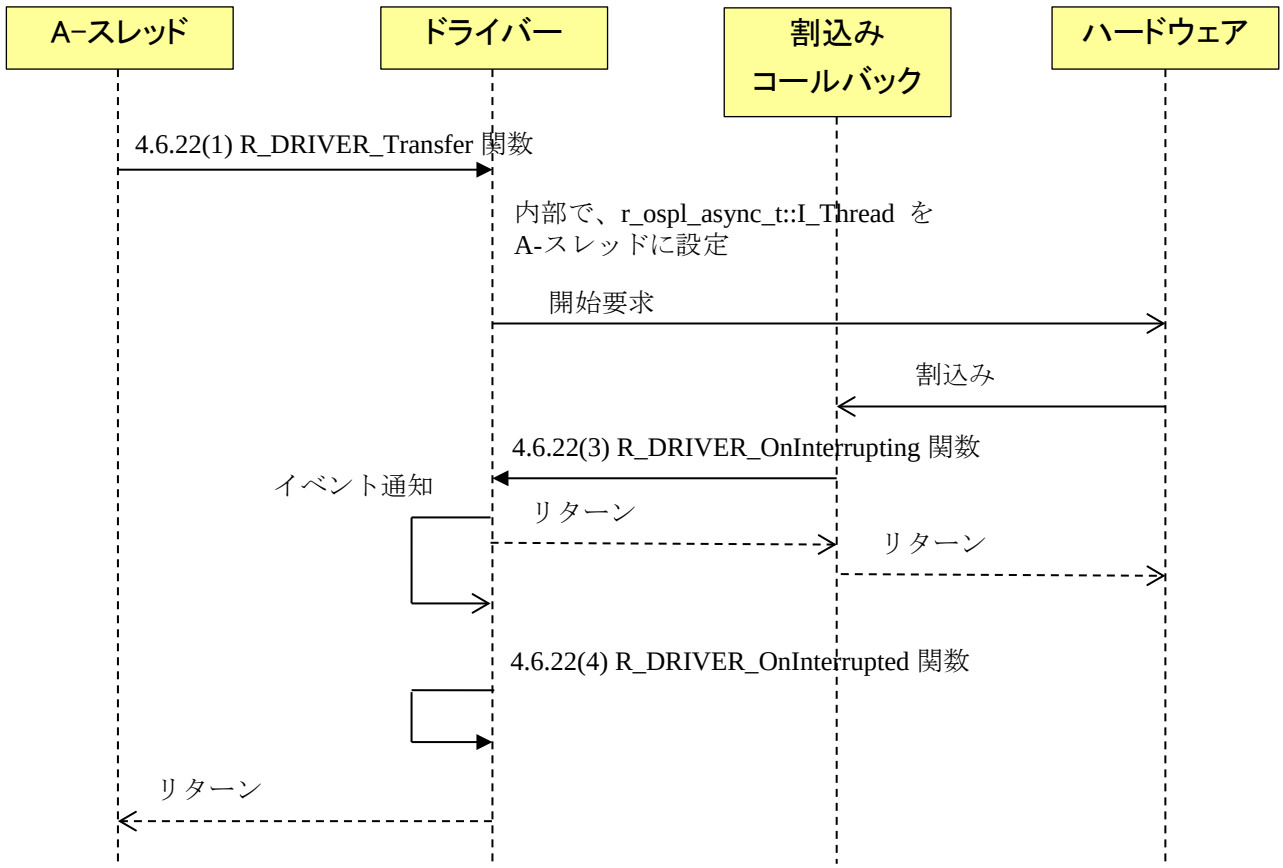


図 4.5　同期型関数を使用したときの割込み応答処理（A-スレッド応答版）

4.7.3　非同期型 割込み応答処理（I-スレッドなし）

図 4.6 に非同期型関数を使用したときの割込み応答処理（r_ospl_async_t::I_Thread==R_OSPL_THREAD_NULL のとき）のシーケンス図を示します。

OS ありの場合、r_ospl_async_t::I_Thread==R_OSPL_THREAD_NULL のときでも、ドライバー内部にある I-スレッドを用いて、(4.7.4) 非同期型 割込み応答処理（I-スレッドあり）とほぼ同じ動きになる仕様のドライバーの場合もあります。

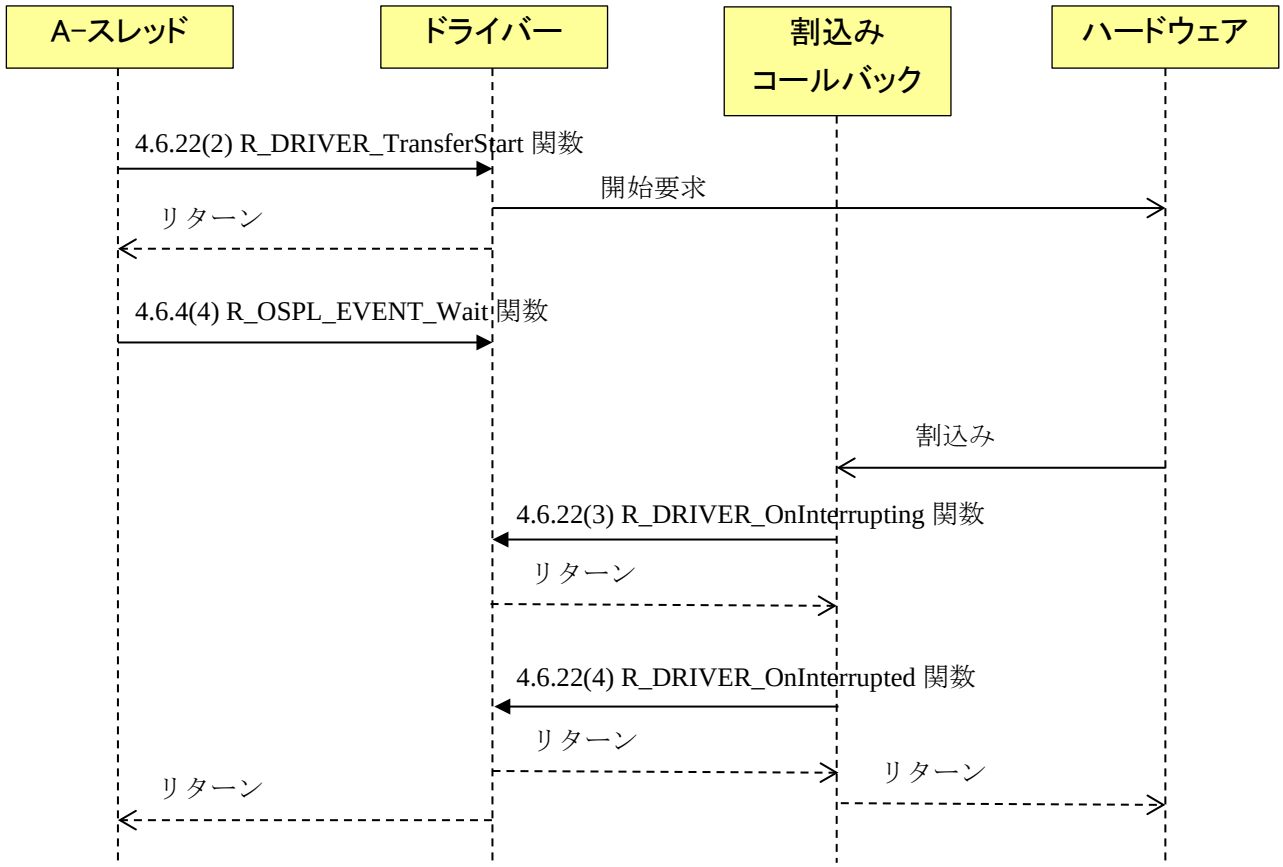


図 4.6　非同期型関数を使用したときの割込み応答処理
（r_ospl_async_t::I_Thread==R_OSPL_THREAD_NULL のとき）

4.7.4　非同期型 割込み応答処理（I-スレッドあり）

図 4.7 に非同期型関数を使用したときの割込み応答処理（r_ospl_async_t::I_Thread !=R_OSPL_THREAD_NULL のとき）のフローチャートを示します。

OS ありの場合、r_ospl_async_t::I_Thread==R_OSPL_THREAD_NULL のときでも、ドライバー内部にある I-スレッドを用いた動きになる仕様のドライバーの場合もありますが、その場合でも、優先度を高くしたいときは、優先度の高いスレッドを r_ospl_async_t::I_Thread に明示的に指定した方がよいでしょう。

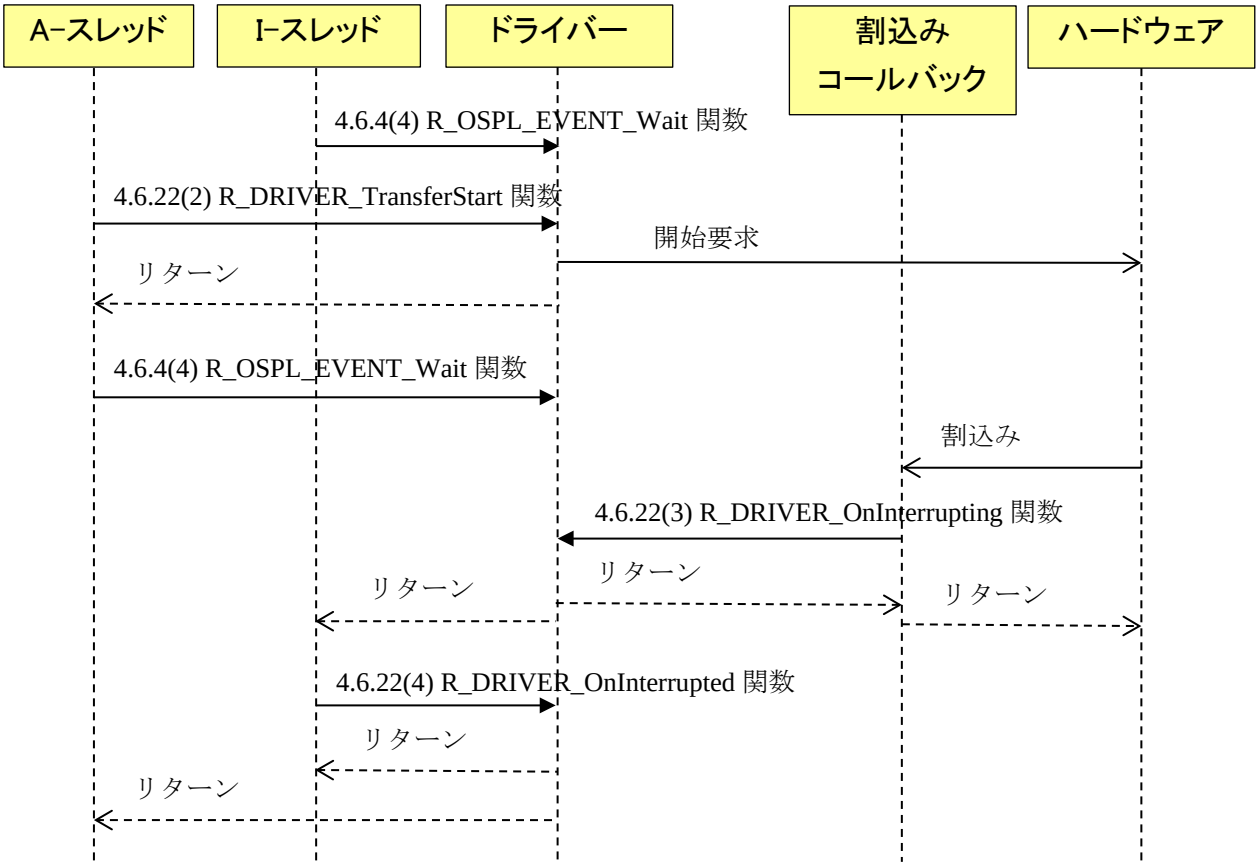


図 4.7　非同期型関数を使用したときの割込み応答処理
（r_ospl_async_t::I_Thread !=R_OSPL_THREAD_NULL のとき）

4.8　補足

4.8.1　ターゲットの選択 (use_list.h, mcu_board_select.h)

OSPL は、ターゲットの OS や、ベースとなるソフトウェア（BSP など）に複数対応しています。 ターゲットを変更するときは、use_list.h ファイル、および mcu_board_select.h ファイルの中に記述されているマクロの定義の有無や定義内容を変更してください。

ただし、すべての組合せに対応しているわけではありません。 テストで使用した組み合わせは以下の通りです。 列挙してあるシンボルは、値なしで定義した #define シンボル、または、#define 定義の値です。

- TARGET_RZA1H,　USE_LIST_RZA1H_OS_LESS
- TARGET_RZA1H,　USE_LIST_RZA1H_BSP,　USE_LIST_CMSIS,　USE_LIST_RTX

4.8.2　フラグド構造体パラメーター

構造体の中の Flags メンバー変数をビットフラグとして使い、ビットが 1 であれば、対応するメンバー変数を有効にするというコーディングパターンです。ビットが 0 ならば、メンバー変数の値は、デフォルトの値が設定されたものと同じとします。バージョンアップしたら構造体のメンバーが増えた場合でも、バイナリ互換にできます。

```
FuncA_ConfigClass  config;

config.Flags  = F_FuncA_Param1 | F_FuncA_Param2;
config.Param1 = 10;
config.Param2 = 2;
FuncA( &config );
```

Flags |= F_FuncA_Param3 が無いため、config.Param3 はデフォルト値。

4.8.3　多重割込みについて

全割込み禁止区間では、多重割込みをサポートしている環境であっても、どの割込みも入りません。

ただし、タイマーのみ入る実装の場合もありますが、ドライバーやアプリケーションのコーディングには影響しません。

あるドライバーA の I-ロック区間の中や、割込みコールバック関数の中では、割込み優先度が高い ドライバーB の割込みが多重割込みとして入ることが可能です。 ただし、ドライバーB は、割込み区間の中から ドライバーA の API を（間接的であっても）呼び出すことができません。

R_OSPL_DisableAllInterrupt 関数では、現在の割込みマスク レベルを退避してから、割込みマスク レベルを最大にします。R_OSPL_EnableAllInterrupt 関数では、退避しておいた割込みマスク レベルに戻します。

4.8.4　OS 移植ガイド

OS ありの OSPL を新しい OS に移植するときは、既存の OS ありの OSPL をベースに移植してください。

OS レスの OSPL を新しいボードに移植するときは、既存の OS レスの OSPL をベースに移植してください。

すべての API を提供しなくても、OSPL（のサブセット） を名乗ることができます。

付属のテストプログラムを使う場合、そのテストプログラムも移植が必要です。

ドライバーの開発に関するガイドは、本書では扱っていません。

(1)　OSPL の内部関数、ドライバー内部の移植層

新しい OS や新しい OS レスのボードに移植するときは、下記の OSPL 内部関数と、ドライバー内部の移植層を変更する必要があります。

- R_BSP_InterruptWrite 関数
- R_BSP_InterruptRead 関数
- R_BSP_InterruptControl 関数

新しい OS に移植するときは、次の関数も変更する必要があります。

- R_OSPL_Start_T_Lock 関数
- R_OSPL_End_T_Lock 関数

(2)　スレッド

スレッドに関する関数（4.6.1(2)）は、R_OSPL_THREAD_GetCurrentId 関数のみ、OS が異なっていても共通に使用できる API です。それ以外の関数は OS によって異なるため、共通の API を提供しません。

OSPL を新しい OS に移植するときは、R_OSPL_THREAD_GetCurrentId 関数の内容を変更してください。

OS がある環境から OS レスに移植するときは、並列して動く関数について擬似マルチスレッドを構成してください。

(3)　イベント

イベントに関する関数（4.6.1(3)、4.6.1(6)）は、OS が異なっていても共通に使用できる API です。

OSPL を新しい OS に移植するときは、OSPL の API 関数の内容を変更してください。

(4)　全割込み禁止区間

全割込み禁止区間に関する関数（4.6.1(8)）は、OS が異なっていても共通に使用できる API です。

OSPL を新しい OS やコンパイラーに移植するときは、OSPL の API 関数の内容を変更してください。

(5)　メモリー関係

メモリーに関する関数（4.6.1(13)）は、OS が異なっていても共通に使用できる API です。

OSPL を新しい OS やボードに移植するときは、OSPL の API 関数の内容を変更してください。

メモリーマップを変更したときも、OSPL の API 関数の内容を変更してください。 キャッシュ領域と非キャッシュ領域の配置については、(4.8.9) を参照してください。

AXI バスについては、「RZ/A1H グループユーザーズマニュアル　ハードウェア編」の「5.8 AXI プロトコルの制御信号」を参照してください。

(6) **時間関係**

時間に関する関数（4.6.1(14)）は、OS が異なっていても共通に使用できる API です。

OSPL を新しい OS やボードに移植するときは、OSPL の API 関数の内容を変更してください。

タイマーのハードウェアを使用する必要があります。ソフトウェアのみでは実装できません。

OS やアプリケーションが使っているタイマーとは、別のタイマーを使用してください。 使用すると、内部でタイマーを参照する OS の API 関数から戻らなくなることがあります。

OS によって使えるタイマーが無くなったときに限り、フリーランタイマーの実装に OS のタイマーを使って構いません。そのとき、R_OSPL_FTIMER_IS マクロの値として、R_OSPL_FTIMER_IS_OS_TIMER_INTERRUPT を定義してください。

(7) **アイドル状態関係**

アイドル状態に関する関数（4.6.1(15)）は、OS レス専用の API です。

新しいボードに移植するときでも、OSPL の API 関数の内容を変更する必要はありません。

OS ありの環境では、優先度を最も低くしたスレッドがアイドル状態の処理に相当します。

(8) **割込みコールバック関連**

割込みコールバック関数に関する関数（4.6.1(16)）は、OS が異なっていても共通に使用できる API です。

OSPL を新しい OS やボードに移植するときは、ドライバーの下位層の移植層に存在する割込みハンドラーが異なるため、割込みハンドラーから割込みコールバック関数を呼び出すコードを変更する必要があります。

(9) **キュー関連**

キューに関する関数（4.6.1(7)）は、OS が異なっていても共通に使用できる API です。

OSPL を新しい OS に移植するときは、OSPL の API 関数の内容を変更してください。

(10) **マルチコンパイラー対応**

マルチコンパイラー対応（4.6.1(19)）は、コンパイラーによって異なる関数や構文の差分を吸収します。

OSPL を新しいコンパイラーに移植するときは、内容を変更してください。

4.8.5　アプリケーション移植ガイド

OSPL の一部の機能に RTOS 関連の機能があります。RTOS の機能に関して、アプリケーションを移植する際の参考として、対応する API を示します。

OSPL と CMSIS-RTOS でほぼ同じ機能であるのは、次の通りです。 また、対応するμITRON の機能も示します。

CMSIS	OSPL	μITRON
Signal	スレッド付属イベント *1	イベントフラグ、または、 タスク付属イベントフラグ（実装独自）
Mail Queue	r_ospl_queue_id_t	固定長メモリプール ＋ データキュー、または、メールボックス
Generic Wait	R_OSPL_Delay	dly_tsk

*1 イベントの衝突を検出しやすくするために、確保・解放の API が追加されています

CMSIS に存在する機能に近い OSPL の機能は、次の通りです。アプリケーションの移植、または、ラッパーの開発が必要になります。

CMSIS	OSPL
Thread	スレッドの生成と削除 ＋ r_ospl_thread_id_t
Timer	スレッドの生成と削除 ＋ フリーランタイマー ＋ R_OSPL_Delay
Mutex	r_ospl_c_lock_t、または、r_ospl_queue_id_t *2
Semaphore	r_ospl_c_lock_t、r_ospl_queue_id_t *2
Message Queue	r_ospl_queue_id_t
Memory Pool	r_ospl_table_t

*2 キューの送信側のスレッドと受信側のスレッドでアクセスするデータを分けることで、排他制御の代わりをします。

4.8.6　インライン関数の本体 inline_body.c について

インライン関数が展開されなかったときに、関数の実体を自動的に生成するコンパイラーと、ユーザーが明示的に生成する必要があるコンパイラーがあります。 コンパイラーのオプションによっては、すべて展開されないこともあります。 展開されなかったときの関数の実体がなければ、リンカーは以下のようなエラーメッセージを出力します。

```
undefined reference to `R_OSPL_THREAD_ExitWaiting'
```

ospl¥porting¥inline_body.c ファイルの #include "r_ospl.h" より下に、インライン関数の定義があるヘッダーファイルをインクルードするコードを書くことで、明示的に関数の実体を生成することができます。 inline_body.c ファイルの先頭で R_OSPL_MAKE_INLINE_BODY が定義してあり、それによって、関数の実体を常に生成するように、INLINE マクロ 4.6.20(4) の定義内容が変わります。

コンパイラーによっては C99 のインライン関数仕様に準拠していないのに、__STDC_VERSION__ >= 199901L が真になってしまい、INLINE マクロの定義の前にある #if が期待通り判定できないことがあります。 その場合は、#if でコンパイラーの種類を判定しないで INLINE マクロを定義するとよいでしょう。

4.8.7　フットプリントの最小化

OSPL にあるデバッグ機能などを外すことで、必要なメモリーサイズ（フットプリント）を減らすことができます。

下記のように設定し、一旦 OSPL の *.c ファイルを削除、サンプル アプリケーションを 1 つだけにして、リビルドしたときにリンクエラーになった関数や変数だけ *.c にコピーしていくと、必要最小限のコードとデータになります。

- R_OSPL_NDEBUG = 1
- R_OSPL_ERROR_BREAK = 0
- R_OSPL_TLS_ERROR_CODE = 0 （R_OSPL_GetErrNum 関数を使っているときは 1 ）
- R_OSPL_DEBUG_TOOL = 0
- R_OSPL_EVENT_OBJECT_CODE = 0
- R_OSPL_DETECT_BAD_EVENT = 0
- R_OSPL_TLS_EVENT_CODE = 0
- R_OSPL_CPU_LOAD = 0 （OS レス版のみ存在）

4.8.8　割込みハンドラー付きドライバーの使用方法

OSPL は、アプリケーションが割込みハンドラーを作成する仕様になっているドライバーを、「割込みハンドラー付きドライバー」と呼んでいます。 このドライバーは、排他制御をアプリケーションが行う仕様になっている可能性があります。

本書に書かれたドライバーの API（例：4.6.22 R_OSPL_SetInterruptPriority）は、OS を抽象化して利用することで、OS の種類や OS の有無に依存しないようになっていますが、割込みハンドラー付きドライバーは、OS の種類や OS の有無に依存する部分を、ドライバーの外に出すことで、OS の種類や OS の有無に依存しないようになっています。

もし、アプリケーションやミドルウェアが、割込みハンドラー付きドライバーを使用する場合、次の点に注意する必要があります。

- OS の有無に関わらず、割込みハンドラーからデータを受け取るために、グローバル変数が必要になります。 その際、volatile 修飾子を忘れないようにする必要があります。イベントを受信する関数で応答処理を記述する場合、最低限必要な volatile のグローバル変数は、スレッドやイベントに関わるものと、割込みクリア後（割込みハンドラー終了後）に取得できなくなる引数やレジスター値です
- OS の有無に関わらず、必要なら、I-ロック区間による排他制御を記述してください（用語集を参照）
- OS ありでも使えるようにするときは、最高の優先度のスレッドがすぐに動けるように、割込みハンドラーでの処理を減らす必要があります。 具体的には、割込みハンドラーの中ではイベントの設定のみ行い、割込みに応答する処理は、イベントを受信する関数に記述してください。 また、非同期処理を開始する前に、イベントをクリアしてください。（OS レス限定使用やユニットテストであれば、この件の対応は不要です）
- OS ありの環境に移植するときは、T-ロック区間の中からドライバーの API を呼び出すことで排他制御をするか、現在のスレッドが期待する単一のスレッドであることをチェックしてからドライバーの API を呼び出すように、アプリケーションを記述してください。

割込みハンドラー付きドライバーを使うときの OS 依存対策例：

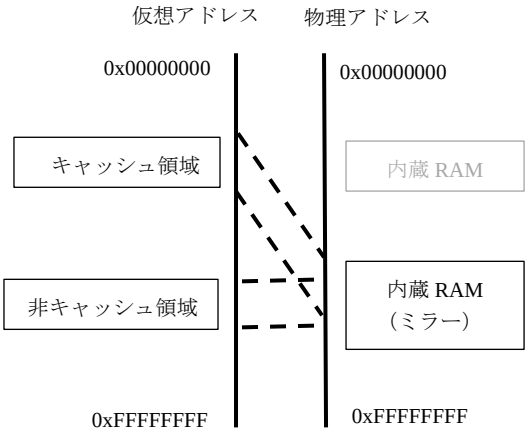
```
enum { R_DRIVER_EVENT_A = 0x0010 };
volatile r_ospl_thread_id_t    g_App_Thread;
volatile r_driver_interrupt_t  g_App_InterruptParameter;

int  main()
{
    g_App_Thread = R_OSPL_THREAD_GetCurrentId();
    R_OSPL_EVENT_Clear( g_App_Thread, R_DRIVER_EVENT_A | R_DRIVER_EVENT_B );
    R_DRIVER_RegisterISR( R_DRIVER_INTERRUPT, App_InterruptCallback );
    R_DRIVER_RegisterISR( R_DRIVER_ERROR_INTERRUPT,
        App_ErrorInterruptCallback );

    for (;;) {
        e= R_OSPL_EVENT_Wait( R_OSPL_ANY_FLAG, &got_flags,
            R_OSPL_INFINITE ); IF(e){goto fin;}

        /* Start I-Lock for R_DRIVER */

        #if R_OSPL_IS_PREEMPTION
            /* Start T-Lock for R_DRIVER */
        #endif

        if ( IS_BIT_SET( got_flags, R_DRIVER_EVENT_A ) ) {
            e = R_DRIVER_GetAsyncResult(); IF(e){goto fin;}
              :

            /* 割込みハンドラーに記述していた処理はここに記述する */
        }
```

```
        if ( IS_BIT_SET( got_flags, R_DRIVER_EVENT_B ) ) {
            e = R_DRIVER_GetAsyncResult(); IF(e){goto fin;}
              :
        }

        #if R_OSPL_IS_PREEMPTION
            /* End T-Lock and/or I-Lock  for R_DRIVER */
        #endif

        /* End I-Lock for R_DRIVER */
    }
}

void  App_InterruptCallback( int Argument )
{
    R_DRIVER_GetParameterOnInterrupting( Argument,
        &g_App_InterruptParameter );
    R_OSPL_EVENT_Set( g_App_Thread, R_DRIVER_EVENT_A );
}

void  App_ErrorInterruptCallback()
{
    R_OSPL_RaiseUnrecoverable( ERROR_CODE );
}
```

4.8.9　キャッシュ領域と非キャッシュ領域の配置パターン（RZ/A1）

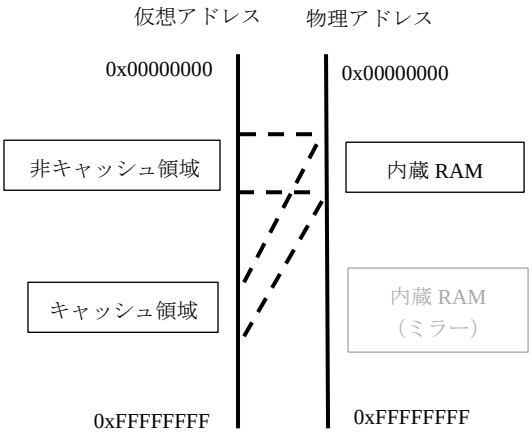
CPU の TTB (MMU) に設定したキャッシュ領域と非キャッシュ領域の配置関係を変更したときは、r_ospl_memory.c ファイルで定義されている関数の定義も、それに合わせて変更する必要があります。 ただし、配置関係にはいくつかのパターンがあり、そのパターンは、GS_RZ_A1_MMU_TYPE の値を変更するだけで変更することができます。 以下にサポートしているパターンを示します。

1. GS_RZ_A1_MMU_TYPE_IS_SHIFTED_MIRROR を選択した場合、キャッシュ領域と非キャッシュ領域の仮想アドレス（ソフトウェアのポインター）は、内蔵 RAM のミラー領域にマップされます。 非キャッシュ領域の仮想アドレスと物理アドレスは一致します。 RZ/A1M では、内蔵 RAM のミラー領域がないので、これを選択することはできません。



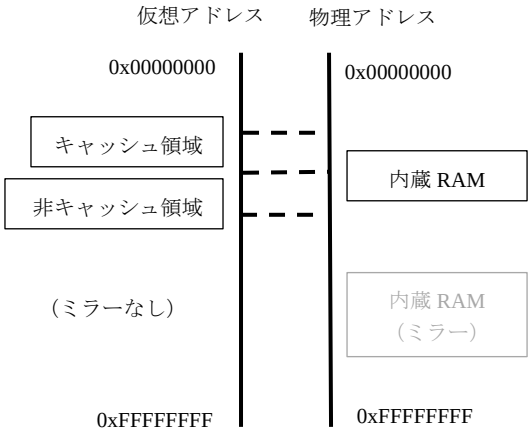
1. GS_RZ_A1_MMU_TYPE_IS_SHIFTED_MIRROR

2. GS_RZ_A1_MMU_TYPE_IS_ REVERSED _MIRROR を選択した場合、キャッシュ領域と非キャッシュ領域の仮想アドレス（ソフトウェアのポインター）は、内蔵 RAM（ミラー領域ではない）にマップされます。 非キャッシュ領域の仮想アドレスと物理アドレスは一致します。



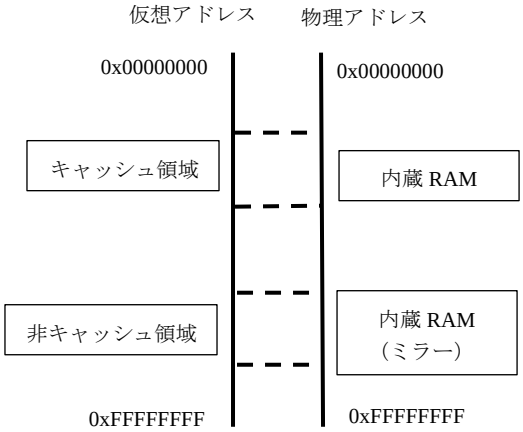
2. GS_RZ_A1_MMU_TYPE_IS_REVERSED_MIRROR

3. GS_RZ_A1_MMU_TYPE_IS_CACHE_UNCACHE を選択した場合、キャッシュ領域と非キャッシュ領域の仮想アドレス（ソフトウェアのポインター）は、内蔵 RAM（ミラー領域ではない）の前半と後半にマップされます。 仮想アドレスと物理アドレスは一致します。 仮想と物理のアドレス変換はできません。



3. GS_RZ_A1_MMU_TYPE_IS_CACHE_UNCACHE

4. GS_RZ_A1_MMU_TYPE_IS_CROSSED_MIRROR を選択した場合、キャッシュ領域の仮想アドレス（ソフトウェアのポインター）は内蔵 RAM（ミラー領域ではない）に、非キャッシュ領域の仮想アドレスは、内蔵 RAM のミラー領域にマップされます。 仮想アドレスと物理アドレスは一致します。 RZ/A1H では、内蔵 RAM オフセット アドレスが一致する最初と最後の 1MB ずつだけアドレス変換を許可しています。 RZ/A1M では、内蔵 RAM のミラー領域がないので、これを選択することはできません。



4. GS_RZ_A1_MMU_TYPE_IS_CROSSED_MIRROR

4.9　用語集

A-イベント	非同期処理が完了したときの通知を受け、A-スレッドを起床させるイベント。
A-スレッド	アプリケーションが実行するスレッド。 メインスレッドなど。 r_ospl_async_t::A_Thread に指定したスレッド。
C-ロック	１つのスレッドがコンテキスト（ドライバーやミドルウェアやハードウェアの現在の状態）を占有すること。 競合したら待たないでエラーにする。 複数のスレッドが同時にロックやアンロックを試みようとする可能性があるため、ロックやロック解除するときはロック状態を保持する内部変数に対して内部で排他制御をする。 T-ロック は関数単位でロックすることが多いが、C-ロック は初期化～終了処理、または、オープン～クローズの間をロックすることが多く、その場合、マルチスレッド環境（または擬似マルチスレッド環境）で排他制御をしなくても、自スレッドが設定したコンテキスト（設定値）が次の関数呼び出しのときにも設定されていることが期待できる。 参考：(4.5.12) r_ospl_c_lock_t、(4.6.22(6)) R_DRIVER_Initialize、T-ロック、I-ロック
I-イベント	割込み応答処理が必要である通知を受け、I-スレッドを起床させるイベント。
I-スレッド	割込み応答処理を実行するスレッド。r_ospl_async_t::I_Thread に指定したスレッド。 R_DRIVER_OnInterrupted 関数を呼び出す処理を行う。 OS ありのときは、ドライバー内部で生成したスレッド。 OS レスのときは、A-スレッドと同じスレッドで、ユーザーが擬似マルチスレッドするが、R_DRIVER_Transfer 関数の中では、I-スレッドが行う処理を、A-スレッドで行う。 IST（Interrupt Service Thread）とほぼ同じ役割。
I-ロック	ドライバーが対象とする周辺機能の割込み禁止をすることで、スレッドで実行する処理と、割込みコンテキスト（割込みハンドラー）で実行する処理の間で排他制御をすること。 たとえば、スレッドから周辺機能を停止する処理と、割込みハンドラーから周辺機能の続きを開始する処理が同時に発生するときに、ドライバー内部の状態変数と周辺機能の状態（レジスターの設定値）に矛盾がないことを維持するために必要になることがある。 割込み禁止状態（I-ロック区間）で割込みが入ったときは、通常、ペンディングされ、許可したときに割込みベクターにジャンプする。 参考：T-ロック、C-ロック
T-ロック	ミューテックスなどを使って、スレッドとスレッドの間で排他制御を行うために、資源や変数をロックすること。 OS レスでは不要。 OS ありでも C-ロックによって T-ロックの対象を占有する構成なら不要。 参考：I-ロック、C-ロック

TLS	スレッド ローカル ストレージ
イベント	参照：4.6.1(3) スレッド付属イベントに関する関数の一覧
擬似マルチスレッド	１つのメッセージループから、それぞれのスレッドのコンテキストで行う処理を実行することで、マルチスレッドをエミュレートすること。タイム シェアリングしなくても、高い優先度のスレッドが割込める OS は、擬似マルチスレッドではない。 擬似マルチスレッドに対応できるモジュールは、API 関数の内部で待つことができない。 同期関数の中で待っている間に他のスレッドのコンテキストの処理を実行する必要があるときは、次のようにして、メッセージループ（メイン関数）までリターンする必要がある。 ・ 待ち（ポーリング）を行う直前までのステップの関数と、ポーリング以後のステップの関数に分離 ・ ポーリングをしないチェック処理、または、タイムアウト０の待ちに変更 ・ 関数のローカル変数と関数の番号（または関数ポインター）を擬似スレッドのコンテキストに保存し、R_OSPL_THREAD_SetDelegate でそのコンテキストを登録する OSPL では、待ちが入る同期処理 API 関数と、それに対応する待ちがない非同期処理 API 関数の両方を提供することを推奨する。 参考：(4.6.3(13)) R_OSPL_THREAD_GetIsWaiting
コンテキスト	現在の設定値を格納する領域。 OSPL ではスレッドに関するコンテキストを指す。割込みコンテキストは、CPU が割込みのモードに入っているときのコンテキスト。 コンテキストはスレッドまたは割込みに関連付けられるが、１つのスレッドや割込みに固定されるとは限らない。 １つのチャンネルでも、そのチャンネルに対して並行して複数の一連の処理を行う場合、コンテキストを扱うコードが必要。 ただし、処理が並行して行われるのは、スレッド間だけでなく割込みの各段（多重割込みの段）でも行われることがあることに注意すること。
スレッド ローカル ストレージ	同じグローバル変数にアクセスしても、スレッドごとにメモリー領域が異なる変数。 スレッド番号を配列番号に持つ配列として実装されることがある。 この場合、TLS とは呼ばないことがある。 参考：(4.5.13) r_ospl_table_t
チャンネル	周辺機能ユニットがもつ複数のチャンネル。
通知設定	4.5.6 r_ospl_async_t
割込みコールバック関数	割込みハンドラーからコールバックされる関数。
割込みハンドラー	OS やボードの API を用いて割込みが発生したときに直接呼ばれる関数。 OS やボードの API の違いを吸収してから、割込みコールバック関数を呼び出す。

ISR（Interrupt Service Routines）のこと。
参考：(4.8.8) 割込みハンドラー付きドライバーの使用方法

5. サンプルコード

サンプルコードは、ルネサス エレクトロニクスホームページから入手してください。

6. 参考ドキュメント

ユーザーズマニュアル：ハードウェア

RZ/A1H グループ ユーザーズマニュアル ハードウェア編
（最新版をルネサス エレクトロニクスホームページから入手してください。）

R7S72100 CPU ボード RTK772100BC00000BR（GENMAI）ユーザーズマニュアル
（最新版をルネサス エレクトロニクスホームページから入手してください。）

R7S72100 CPU ボード用オプションボード RTK7721000B00000BR（GENMAI）ユーザーズマニュアル
（最新版をルネサス エレクトロニクスホームページから入手してください。）

ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition Issue C
（最新版を ARM ホームページから入手してください。）

ARM Generic Interrupt Controller Architecture Specification Architecture version 1.0
（最新版を ARM ホームページから入手してください。）

テクニカルアップデート／テクニカルニュース
（最新の情報をルネサス エレクトロニクスホームページから入手してください。）

ユーザーズマニュアル：開発環境

ARM ソフトウェア開発ツール（ARM Compiler toolchain、ARM DS-5 等）に関しては、ARM ホームページから入手してください。
（最新版を ARM ホームページから入手してください。）

ホームページとサポート窓口

ルネサス エレクトロニクスホームページ
http://japan.renesas.com/

お問合せ先
http://japan.renesas.com/contact/

改訂記録

Doc. Rev.	Version	発行日	改訂内容
1.60	1.60	2017.10.26	追加： use_list.h, mcu_board_select.h ：環境の選択 R_OSPL_FOR_* ：環境のバージョン FreeRTOS 対応 R_OSPL_INTERRUPT_HANDLER_IS R_OSPL_END_OF_INTERRUPT R_OSPL_THREAD_Destroy R_OSPL_THREAD_SetDelegate R_OSPL_THREAD_GetDelegate R_OSPL_NOTIFY_ERROR R_OSPL_FreeCurrentThreadError R_OSPL_INFINITE_PSEUDO R_OSPL_ALL_EVENT_ALLOCATE r_ospl_event_group_id_t R_OSPL_EVENT_GROUP_Create R_OSPL_EVENT_GROUP_Delete R_OSPL_QUEUE_NULL R_OSPL_GetQueueAsSingletonLock R_OSPL_TABLE_GetStatus R_OSPL_MOVE_END_OF_STACK R_OSPL_GET_MIN_STACK_POINTER CHK ：簡易エラーチェック 4.8.6 インライン関数についての資料 4.8.9 キャッシュ領域と非キャッシュ領域の配置についての資料 削除： イベント オブジェクト 修正： 各種コンパイル時の警告に対処 シンボルの中の語の修正（sentinel → canary） （modify_thread → change_thread） 意図しない I-ロックがかかってしまう問題に対処（R_OSPL_NDEBUG が非定義のときのみ）
1.01	0.96	2016.02.29	追加： R_OSPL_EVENT_Allocate R_OSPL_EVENT_Free R_OSPL_UNUSED_FLAG R_OSPL_DETECT_BAD_EVENT R_OSPL_ASYNC_SetDefaultPreset イベント オブジェクト 配列番号表 スタック チェック R_ADDRESS_Add R_OSPL_MEMORY_GetMaxLevelOfFlush R_OSPL_CountLeadingZeros R_OSPL_SET_DEBUG_WORK を廃止し、 R_OSPL_DEBUG_THREAD_COUNT を追加。

			R_OSPL_THREAD_INTERRUPT を廃止し、R_OSPL_THREAD_NULL を追加。 r_ospl_queue_t から r_ospl_queue_id_t に変更。
	0.90	2015.07.31	r_ospl_queue_t、r_ospl_axi_cache_attribute_t を追加
1.00	0.89	2014.07.30	R_OSPL_SetInterruptPriority を追加。
0.88	0.88	2014.06.20	OSPL の初期化の API を追加。 NDEBUG を R_OSPL_NDEBUG に分離。 時間に関する仕様を詳細に定義。 OS レスでの待ち方を設定する API（R_OSPL_THREAD_SetOnWait 関数など）を追加。 E_NOT_THREAD エラーを追加。 R_OSPL_FLUSH_INVALIDATE を追加。 C-ロックに関する API を追加。
	0.87	2014.03.18	ビットをクリアする各種関数の引数のビットの意味を反転。 r_ospl_interrupt_t 型構造体のメンバー変数を追加。 仕様を追加：R_OSPL_TLS_ERROR_CODE、R_OSPL_DEBUG_TOOL、R_OSPL_THREAD_INTERRUPT、R_OSPL_FINAL_A_FLAG 説明を追加：R_DRIVER_SetDefaultAsync、NDEBUG
	0.80	2013.12.27	初版発行

製品ご使用上の注意事項

ここでは、マイコン製品全体に適用する「使用上の注意事項」について説明します。個別の使用上の注意事項については、本ドキュメントおよびテクニカルアップデートを参照してください。

1. 未使用端子の処理

   【注意】未使用端子は、本文の「未使用端子の処理」に従って処理してください。

   CMOS製品の入力端子のインピーダンスは、一般に、ハイインピーダンスとなっています。未使用端子を開放状態で動作させると、誘導現象により、LSI周辺のノイズが印加され、LSI内部で貫通電流が流れたり、入力信号と認識されて誤動作を起こす恐れがあります。未使用端子は、本文「未使用端子の処理」で説明する指示に従い処理してください。

2. 電源投入時の処置

   【注意】電源投入時は，製品の状態は不定です。

   電源投入時には、LSIの内部回路の状態は不確定であり、レジスターの設定や各端子の状態は不定です。

   外部リセット端子でリセットする製品の場合、電源投入からリセットが有効になるまでの期間、端子の状態は保証できません。

   同様に、内蔵パワーオンリセット機能を使用してリセットする製品の場合、電源投入からリセットのかかる一定電圧に達するまでの期間、端子の状態は保証できません。

3. リザーブアドレス（予約領域）のアクセス禁止

   【注意】リザーブアドレス（予約領域）のアクセスを禁止します。

   アドレス領域には、将来の機能拡張用に割り付けられているリザーブアドレス（予約領域）があります。これらのアドレスをアクセスしたときの動作については、保証できませんので、アクセスしないようにしてください。

4. クロックについて

   【注意】リセット時は、クロックが安定した後、リセットを解除してください。

   プログラム実行中のクロック切り替え時は、切り替え先クロックが安定した後に切り替えてください。

   リセット時、外部発振子（または外部発振回路）を用いたクロックで動作を開始するシステムでは、クロックが十分安定した後、リセットを解除してください。また、プログラムの途中で外部発振子

   （または外部発振回路）を用いたクロックに切り替える場合は、切り替え先のクロックが十分安定してから切り替えてください。

5. 製品間の相違について

   【注意】型名の異なる製品に変更する場合は、製品型名ごとにシステム評価試験を実施してください。

   同じグループのマイコンでも型名が違うと、内部ROM、レイアウトパターンの相違などにより、電気的特性の範囲で、特性値、動作マージン、ノイズ耐量、ノイズ輻射量などが異なる場合があります。型名が違う製品に変更する場合は、個々の製品ごとにシステム評価試験を実施してください。

ご注意書き

1. 本資料に記載された回路、ソフトウェアおよびこれらに関連する情報は、半導体製品の動作例、応用例を説明するものです。お客様の機器・システムの設計において、回路、ソフトウェアおよびこれらに関連する情報を使用する場合には、お客様の責任において行ってください。これらの使用に起因して生じた損害（お客様または第三者いずれに生じた損害も含みます。以下同じです。）に関し、当社は、一切その責任を負いません。
2. 当社製品、本資料に記載された製品データ、図、表、プログラム、アルゴリズム、応用回路例等の情報の使用に起因して発生した第三者の特許権、著作権その他の知的財産権に対する侵害またはこれらに関する紛争について、当社は、何らの保証を行うものではなく、また責任を負うものではありません。
3. 当社は、本資料に基づき当社または第三者の特許権、著作権その他の知的財産権を何ら許諾するものではありません。
4. 当社製品を、全部または一部を問わず、改造、改変、複製、その他の不適切に使用しないでください。かかる改造、改変、複製等により生じた損害に関し、当社は、一切その責任を負いません。
5. 当社は、当社製品の品質水準を「標準水準」および「高品質水準」に分類しており、各品質水準は、以下に示す用途に製品が使用されることを意図しております。

   標準水準： コンピュータ、OA機器、通信機器、計測機器、AV機器、
   家電、工作機械、パーソナル機器、産業用ロボット等

   高品質水準： 輸送機器（自動車、電車、船舶等）、交通制御（信号）、大規模通信機器、
   金融端末基幹システム、各種安全制御装置等

   当社製品は、直接生命・身体に危害を及ぼす可能性のある機器・システム（生命維持装置、人体に埋め込み使用するもの等）、もしくは多大な物的損害を発生させるおそれのある機器・システム（宇宙、海底中継器、原子力制御システム、航空機制御システム、プラント基幹システム、軍事機器等）に使用されることを意図しておらず、これらの用途に使用することはできません。たとえ、意図しない用途に当社製品を使用したことにより損害が生じても、当社は一切その責任を負いません。
6. 当社製品をご使用の際は、最新の製品情報（データシート、ユーザーズマニュアル、アプリケーションノート、信頼性ハンドブックに記載の「半導体デバイスの使用上の一般的な注意事項」等）をご確認の上、当社が指定する最大定格、動作電源電圧範囲、放熱特性、実装条件その他指定条件の範囲内でご使用ください。指定条件の範囲を超えて当社製品をご使用された場合の故障、誤動作の不具合および事故につきましては、当社は、一切その責任を負いません。
7. 当社は、当社製品の品質および信頼性の向上に努めていますが、半導体製品はある確率で故障が発生したり、使用条件によっては誤動作したりする場合があります。また、当社製品は耐放射線設計を行っておりません。仮に当社製品の故障または誤動作が生じた場合であっても、人身事故、火災事故その他社会的損害等を生じさせないよう、お客様の責任において、冗長設計、延焼対策設計、誤動作防止設計等の安全設計およびエージング処理等、お客様の機器・システムとしての出荷保証を行ってください。特に、マイコンソフトウェアは、単独での検証は困難なため、お客様の機器・システムとしての安全検証をお客様の責任で行ってください。
8. 当社製品の環境適合性等の詳細につきましては、製品個別に必ず当社営業窓口までお問合せください。ご使用に際しては、特定の物質の含有・使用を規制するRoHS指令等、適用される環境関連法令を十分調査のうえ、かかる法令に適合するようご使用ください。かかる法令を遵守しないことにより生じた損害に関して、当社は、一切その責任を負いません。
9. 当社製品および技術を国内外の法令および規則により製造・使用・販売を禁止されている機器・システムに使用することはできません。また、当社製品および技術を、(1)核兵器、化学兵器、生物兵器等の大量破壊兵器およびこれらを運搬することができるミサイル（無人航空機を含みます。）の開発、設計、製造、使用もしくは貯蔵等の目的、(2)通常兵器の開発、設計、製造または使用の目的、または(3)その他の国際的な平和および安全の維持の妨げとなる目的で、自ら使用せず、かつ、第三者に使用、販売、譲渡、輸出、賃貸もしくは使用許諾しないでください。

   当社製品および技術を輸出、販売または移転等する場合は、「外国為替及び外国貿易法」その他日本国および適用される外国の輸出管理関連法規を遵守し、それらの定めるところに従い必要な手続きを行ってください。
10. お客様の転売、貸与等により、本書（本ご注意書きを含みます。）記載の諸条件に抵触して当社製品が使用され、その使用から損害が生じた場合、当社は一切その責任を負わず、お客様にかかる使用に基づく当社への請求につき当社を免責いただきます。
11. 本資料の全部または一部を当社の文書による事前の承諾を得ることなく転載または複製することを禁じます。
12. 本資料に記載された情報または当社製品に関し、ご不明点がある場合には、当社営業にお問い合わせください。

注1. 本資料において使用されている「当社」とは、ルネサス エレクトロニクス株式会社およびルネサス エレクトロニクス株式会社がその総株主の議決権の過半数を直接または間接に保有する会社をいいます。

注2. 本資料において使用されている「当社製品」とは、注1において定義された当社の開発、製造製品をいいます。

(Rev.3.0-1 2016.11)

ルネサス エレクトロニクス株式会社

■営業お問合せ窓口

http://www.renesas.com

※営業お問合せ窓口の住所は変更になることがあります。最新情報につきましては、弊社ホームページをご覧ください。

ルネサス エレクトロニクス株式会社　〒135-0061　東京都江東区豊洲3-2-24（豊洲フォレシア）

■技術的なお問合せおよび資料のご請求は下記へどうぞ。
総合お問合せ窓口：https://www.renesas.com/contact/