

RTOS-WinJ 1.0 Reference Manual

目 次

第一章 はじめに

1.1 RTOS-WinJ とは

第二章 概要

- 2.1 スレッドとスレッドの状態
- 2.2 スケジューリングと優先度
- 2.3 共用メモリデータベース
- 2.4 終了要求
- 2.5 デバッグトレース
- 2.6 その他
- 2.7 警告
- 2.8 プロセス間通信
 - 2.9.1 メッセージ
 - 2.9.2 シグナル

第三章 RTOS-WinJ ランタイムマネージャ

- 3.1 Windows からの RTOS-WinJ アプリケーションの起動
 - 3.1.1 パスワード保護
 - 3.1.2 RTOS-WinJ アプリケーションの起動と停止
- 3.2 出力メッセージの表示
- 3.3 スレッドステータスの表示と変更
- 3.4 メモリ内容の表示
- 3.5 名前付メモリ
- 3.6 ダイナミックデータエクスチェンジ (DDE)
- 3.7 パフォーマンスの表示
- 3.8 RTOS-WinJ システム情報の取得と操作

第四章 RTOS-WinJ トレースユーティリティ

- 4.1 トレースユーティリティを使用可能にする
- 4.2 トレースユーティリティの起動
- 4.3 開始・停止トリガの選択
 - 4.3.1 スタートおよびストップトリガイベント
- 4.4 トレースユーティリティ Settings タブ
- 4.5 Historical Trace File Information タブ

第五章 RTOS-WinJ 設定ウィザード

- 5.1 RTOS-WinJ 設定ウィザードの開始
- 5.2 システム設定
- 5.3 標準タイミングの設定
- 5.4 メモリボードの設定
- 5.5 Windows ブートメニューの選択
- 5.6 通信ポートの設定
- 5.7 現在の設定
- 5.8 リブートシステム

第六章 RTOS-WinJ アプリケーションの開発

- 6.1 RTOS-WinJ API
- 6.2 RTOS-WinJ プログラムの作成
- 6.3 システム関数の使用
- 6.4 コンパイラの設定

第七章 HyperShare アプリケーションの開発

- 7.1 HyperShare API

付録A RTOS-WinJ API リファレンス

A.1 メモリ

- A.1.1 `rwAlloc`
- A.1.2 `rwFree`
- A.1.3 `rwMapMemory`
- A.1.4 `rwUnmapMemory`

A.2 共用メモリ

- A.2.1 `rwLocateMemory`
- A.2.2 `rwNumberSharedBits`
- A.2.3 `rwNumberSharedBytes`
- A.2.4 `rwNumberSharedLongs`
- A.2.5 `rwNumberSharedWords`
- A.2.6 `rwSharedRam`
- A.2.7 `rwShareMemory`
- A.2.8 `rwUserSharedRam`
- A.2.9 `rwReadBit`
- A.2.10 `rwReadByte`
- A.2.11 `rwReadLong`
- A.2.12 `rwReadWord`
- A.2.13 `rwUnshareMemory`
- A.2.14 `rwWriteBit`
- A.2.15 `rwWriteByte`
- A.2.16 `rwWriteLong`
- A.2.17 `rwWriteWord`

A.3 リテンティブメモリ

- A.3.1 `rwRetentInit`
- A.3.2 `rwRetentRead`
- A.3.3 `rwRetentWrite`
- A.3.4 `rwRetentExit`
- A.3.5 `rwRetentQueryBatteryStatus`

A.4 ファイル I/O

- A.4.1 `rwCopyFile`
- A.4.2 `rwOpen`
- A.4.3 `rwClose`
- A.4.4 `rwRead`
- A.4.5 `rwRemoveFile`
- A.4.6 `rwRenameFile`
- A.4.7 `rwWrite`
- A.4.8 `rwTell`
- A.4.9 `rwSizeOfFile`

- A. 4. 10 rwSeek
- A. 4. 11 rwReadString
- A. 4. 12 rwWriteString

A. 5 タイミング

- A. 5. 1 rwDelay
- A. 5. 2 rwMicroClock
- A. 5. 3 rwMicroSleep
- A. 5. 4 rwSleep
- A. 5. 5 rwHardwareCounter
- A. 5. 6 rwGetTicks
- A. 5. 7 rwTime

A. 6 スレッドコントロール

- A. 6. 1 rwCreateProcess
- A. 6. 2 rwCreateProcessEx
- A. 6. 3 rwCreateThread
- A. 6. 4 rwCreateThreadEx
- A. 6. 5 rwEnterCriticalSection
- A. 6. 6 rwExitThread
- A. 6. 7 rwForceReady
- A. 6. 8 rwGetMainTid
- A. 6. 9 rwGetPriority
- A. 6. 10 rwGetThreadData
- A. 6. 11 rwGetTid
- A. 6. 12 rwKillThread
- A. 6. 13 rwLeaveCriticalSection
- A. 6. 14 rwLocateThread
- A. 6. 15 rwParentTid
- A. 6. 16 rwReleaseProcessor
- A. 6. 17 rwResume
- A. 6. 18 rwSeizeProcessor
- A. 6. 19 rwSetPriority
- A. 6. 20 rwSuspend
- A. 6. 21 rwThreadState
- A. 6. 22 rwYeild

- A. 7 スレッド間通信 : メッセージ**
 - A. 7. 1 rwCreceive
 - A. 7. 2 rwReceive
 - A. 7. 3 rwReply
 - A. 7. 4 rwSend
- A. 8 スレッド間通信 : セマフォ**
 - A. 8. 1 rwAllocSemaphore
 - A. 8. 2 rwCreateSemaphore
 - A. 8. 3 rwDestroySemaphore
 - A. 8. 4 rwFreeSemaphore
 - A. 8. 5 rwLocateSemaphore
- A. 9 スレッド間通信 : 新セマフォ関数**
 - A. 9. 1 rwOpenSemaphore
 - A. 9. 2 rwGetSemaphore
 - A. 9. 3 rwTrySemaphore
 - A. 9. 4 rwReleaseSemaphore
 - A. 9. 5 rwCloseSemaphore
- A. 10 スレッド間通信 : シグナル**
 - A. 10. 1 rwClearSignals
 - A. 10. 2 rwReadSignals
 - A. 10. 3 rwSignal
- A. 11 デバッグ**
 - A. 11. 1 rwDebug
 - A. 11. 2 rwPrint
- A. 12 割り込み**
 - A. 12. 1 rwInterruptAttach
 - A. 12. 2 rwInterruptDetach
 - A. 12. 3 rwInterruptDirwble
 - A. 12. 4 rwInterruptEnable
- A. 13 入出力**
 - A. 13. 1 rwInp, rwInpw, rwInpd
 - A. 13. 2 rwOutp, rwOutpw, rwOutpd
- A. 14 トレースデバッグ**
 - A. 14. 1 rwDebugTraceEvent
 - A. 14. 2 rwDumpDebugTrace
 - A. 14. 3 rwGetTraceBufferMode
 - A. 14. 4 rwGetTraceBufferSize
 - A. 14. 5 rwGetTraceState
 - A. 14. 6 rwGetTraceTriggers
 - A. 14. 7 rwSetTraceBufferMode
 - A. 14. 8 rwSetTraceBufferSize
 - A. 14. 9 rwSetTraceDumpFile

- A. 14. 10 rwSetTraceTriggers
- A. 14. 11 rwStartTraceMonitoring
- A. 14. 12 rwStopTracing
- A. 14. 13 rwTraceDataAvailable
- A. 15 **その他**
 - A. 15. 1 rwAttacrwystemTimer
 - A. 15. 2 rwDetacrwystemTimer
 - A. 15. 3 rwGetPrivateProfileString
 - A. 15. 4 rwGetSystemDirectory
 - A. 15. 5 rwGetWindowsDirectory
 - A. 15. 6 rwQuitRequest
 - A. 15. 7 rwSetGateSpeed
 - A. 15. 8 rwSystemCrash
 - A. 15. 9 rwTextOut
 - A. 15. 10 rwWritePrivateProfileString

付録B HyperShare API リファレンス

- B. 1 **システム**
 - B. 1. 1 rwGetRWInstallDir
 - B. 1. 2 rwGetLastError
 - B. 1. 3 rwStartApplication
 - B. 1. 4 rwStopApplication
- B. 2 **データアクセス**
 - B. 2. 1 rwAttacrwharedRam
 - B. 2. 2 rwDetacrwharedRam
 - B. 2. 3 rwNumberSharedBits
 - B. 2. 4 rwNumberSharedBytes
 - B. 2. 5 rwNumberSharedLongs
 - B. 2. 6 rwNumberSharedWords
 - B. 2. 7 rwNumberUserSharedBytes
 - B. 2. 8 rwNumberUserSharedLongs
 - B. 2. 9 rwNumberUserSharedWords
 - B. 2. 10 rwReadBit
 - B. 2. 11 rwReadByte
 - B. 2. 12 rwReadLong
 - B. 2. 13 rwReadUserByte
 - B. 2. 14 rwReadUserLong
 - B. 2. 15 rwReadUserword
 - B. 2. 16 rwReadWord
 - B. 2. 17 rwSharedRam

- B. 2. 18 rwUserSharedRam
- B. 2. 19 rwWriteBit
- B. 2. 20 rwWriteByte
- B. 2. 21 rwWriteLong
- B. 2. 22 rwWriteUserByte
- B. 2. 23 rwWriteUserLong
- B. 2. 24 rwWriteUserword
- B. 2. 25 rwWriteWord
- B. 3 スレッド関連**
 - B. 3. 1 rwAwait
 - B. 3. 2 rwKillThread
 - B. 3. 3 rwLocateMemory
 - B. 3. 4 rwLocateThread
 - B. 3. 5 rwMaxThreads
 - B. 3. 6 rwNameAttach
 - B. 3. 7 rwNameDetach
 - B. 3. 8 rwParentTid
 - B. 3. 9 rwReadSignals
 - B. 3. 10 rwSendTid
 - B. 3. 11 rwSignal
 - B. 3. 12 rwStackSize
 - B. 3. 13 rwStatusHeader
 - B. 3. 14 rwStatusString
 - B. 3. 15 rwSuspend
 - B. 3. 16 rwThreadName
 - B. 3. 17 rwThreadPriority
 - B. 3. 18 rwThreadState
- B. 4 ブールステータス**
 - B. 4. 1 rwKernelError
 - B. 4. 2 rwProcessExit
 - B. 4. 3 rwProcessFault
 - B. 4. 4 rwSystemActive
- B. 5 スレッド間通信 : セマフォ**
 - B. 5. 1 rwAllocSemaphoreEx
 - B. 5. 2 rwDestroySemaphoreEx
 - B. 5. 3 rwFreeSemaphoreEx

B. 6 スレッド間通信 : 新セマフォ関数

- B. 6. 1 rwCloseSemaphoreEx
- B. 6. 2 rwGetSemaphoreEx
- B. 6. 3 rwOpenSemaphoreEx
- B. 6. 4 rwReleaseSemaphoreEx
- B. 6. 5 rwTrySemaphoreEx

B. 7 デバッグトレースユーティリティ

- B. 7. 1 rwDumpDebugTrace
- B. 7. 2 rwFormatTraceFile
- B. 7. 3 rwFormatTraceFileCSV
- B. 7. 4 rwGetStartupTracing
- B. 7. 5 rwGetTraceBufferMode
- B. 7. 6 rwGetTraceBufferSize
- B. 7. 7 rwGetTraceDumpFile
- B. 7. 8 rwGetTraceState
- B. 7. 9 rwGetTraceTriggers
- B. 7. 10 rwSetStartupTracing
- B. 7. 11 rwSetTraceBufferMode
- B. 7. 12 rwSetTraceBufferSize
- B. 7. 13 rwSetTraceDumpFile
- B. 7. 14 rwSetTraceTriggers
- B. 7. 15 rwStartTraceMonitoring
- B. 7. 16 rwStopTracing

B. 8 その他

- B. 8. 1 rwAttachDebugWindow
- B. 8. 2 rwClearDebug
- B. 8. 3 rwDetachDebugWindow
- B. 8. 4 rwGetSystemTime

第一章 はじめに

ハードリアルタイム OS (RTOS) のアプリケーション開発者たちは、独自の RTOS 環境に対応できる開発ツールが限定されていたため、作業が制限されていました。必要なツール、ソフトウェア機能などの資源が限られているために開発に取り組む中でコストや開発範囲に限界が生じ、さらにはマーケットへの流通を遅らせていると RTOS を使用している開発者のほとんどが感じています。

Windows XP®オペレーティングシステム用のRTOS-WinJがあれば、Windowsの多様な開発環境を利用できるようになります。

数多くの利用可能な資源を使いながら、より低コストでより短期間にハードリアルタイムアプリケーションの開発が可能になりました。

商標について

本サイトに掲載されている商品またはサービスなどの名称は、各社の商標または登録商標です。

各社の商標または登録商標

※Microsoft, Windows XP, Windows, Microsoft Developers Studio, MS-DOS
Microsoft Visual C, Microsoft Visual Basic, Windows WDK, Microsoft Excel
の商標はマイクロソフト社の登録商標です。

1.1 RTOS-WinJとは

当社は航空宇宙用アプリケーションのデータ収集とコントロールソフトウェアに関する経験から、RTOS-WinJのコンセプトを提案いたします。

産業用コントロールのランタイム実行に対して、従来のリアルタイムマイクロカーネルターゲットシステムをサポートするものでした。

ハードリアルタイムアプリケーションを作成するために必要なツールをアプリケーション開発者に提供します。以下のRTOS-WinJの長所を利用して32ビットアプリケーションの開発が可能になります。

- ・ マルチスレッドのスケジューリングをベースとするプリエンティブなラウンドロビンの優先
- ・ 高速なスレッド間通信
- ・ アプリケーションスレッドのダイナミックスタート/ストップ
- ・ Windows XP標準アプリケーションと同時実行
- ・ ハードウェアへのダイレクトなアクセス (Windows WDK不要)
- ・ Windows XPアプリケーション対応の共用メモリアンターフェイス
- ・ 同時スレッド用シグナル
- ・ インタラプトマネジメントサービス
- ・ Microsoft標準開発ツールの使用
- ・ 高速で正確なタイマ

RTOS-WinJ の環境は独立型です。RTOS-WinJ のスレッドは RTOS-WinJ の環境で実行されるので、Windows XP のスケジューリングが立ち遅れても、影響されることはありません。Windows XP スケジューラは、RTOS-WinJ のスレッドを感知しないようになっています。RTOS-WinJ は、スケジューラ、一連のサービス、及び内部カーネルを持っています。

第二章 概要

2.1 スレッドとスレッドの状態

RTOS-WinJ システムは、最大 1024 の独立型実行スレッドから成り立っています。

スレッド ID により、個々のスレッドを識別することができます。

システムのために保護されている 5 つのスレッドには、\$Main、\$HyperGate、\$DebugText、\$Idle、及び\$NTtoRWSignal があります。

残りの 1019 のスレッドに関しては、自由に定義して利用可能です。

保護されているスレッドは以下のように定義されています。

- ・ \$Mainは、ユーザーのメインスレッドです。
- ・ \$HyperGateはWindows XPのサービス管理者であり、RTOS-WinJアプリケーションとWindows XP間のコミュニケーションを管理します。
RTOS-WinJの観点からだと、ファイルI/Oサーバーとして機能します。
- ・ \$DebugTextはバッファ付プリントステートメントを取り扱います。
- ・ \$Idle は、優先度が低い状態で実行するスレッドであり、優先度が高いスレッドが他に存在しない場合のみ実行されます。
- ・ \$NTtoRWSignalは、Windows XPからRTOS-WinJに対するシグナルを処理するスレッドです。

それぞれのアクティブスレッドの状態は、以下に記述した5つの内、常にいずれかの状態になります。

- | | |
|-----------|---|
| ・ READY | - スレッドはいつでも実行できます。 |
| ・ WAIT | - スレッドはスリープ状態です。(すなわちスレッドは <code>rwSleep</code> または <code>rwDelay</code> ステートメントを実行しています) |
| ・ SEND | - スレッドは他のスレッドにメッセージを送信し、返信を待っています。 |
| ・ RECEIVE | - スレッドは他のスレッドからのメッセージやシグナルを待っています。 |
| ・ SUSPEND | - 実行スレッドが一時中断されています。 |

その他のスレッドの状態は、エラー状態になった場合のみ表示されます。例えば FREE (スレッド ID が未定義の場合)、ZOMBIE (スレッド内で内部エラーが発生した場合、及びスレッドがシステムに正常に応答しなくなった場合) などです。

2.2 スケジューリングと優先度

RTOS-WinJ は、個々のスレッドに 0（最低）から 31（最高）までの優先度を指定します。システムがスレッドを実行する方法やタイミング及びスレッドを他のスレッドに切り換えるタイミングがスレッドの優先度の判断材料になります。

例えば、システムスレッドの優先度は、

- ・ \$Main は優先度 8 で実行します。
- ・ \$HyperGate は優先度 23 で実行します。
- ・ \$DebugText は優先度 8 で実行します。
- ・ \$Idle は優先度 0 で実行します。
- ・ \$NTtoRWSignal は優先度 23 で実行します。

優先度 n のスレッドが実行可能な状態の場合、 n より低いスレッドは実行されません。2つ以上の同じ優先度のスレッドが実行可能な状態で、それよりも高い優先度のスレッドが実行可能な状態ではない場合、既に実行可能なスレッドはシステムにおいて1000分の2秒毎に発生するプリエンティブなラウンドロビンで実行されます。

システムの複数のイベントはスレッドより先にスケジューラを実行させます。
例えば、

- ・ sleep、delay、または yield 関数の実行
- ・ ほかのスレッドへのメッセージ送信
- ・ メッセージまたはシグナル受信待ち
- ・ ISR（インタラプトサービスルーチン）からのシグナル受信
- ・ システムタイマの完了

2.3 共用メモリデータベース

RTOS-WinJユーザーデータ領域は、RTOS-WinJアプリケーションを実行するときは常に存在します。このデータ領域は、既に定義されたさまざまな種類の一連のグローバル変数および数値から構成されており、これらはユーザー専用です。

ほとんどのシステムでは、このデータ領域は、16K(16384)Bit、2K(2048)Byte、2K(2048)Word、2K(2048)Long、および汎用バッファとしてユーザーが使用可能な52K(53248)バイトのメモリ領域から構成されています。

rwNumberShared関数やrwUserSharedRam関数を呼び出して、ユーザーがデータの番号及びサイズを確認することを強くお勧めします。ユーザーデータ領域のBit、Byte、Word、及びLongは、変数の番号（例えば、Bit#3、Word#7など）を指定し、rwRead関数やrwWrite関数でアクセスします。

すべてのデータアクセス関数に関しては、**RTOS-WinJAPI**の中で詳しく説明します。

2.4 終了要求

rwQuitRequest関数によりRTOS-WinJアプリケーションは、システムのシャットダウンによるアプリケーション自身の終了するタイミングを決定します。

rwQuitRequestコール周辺に、RTOS-WinJアプリケーションのメインループを設定することをお勧めします。

アプリケーションが所定時間内（現在の設定は30秒）に、システムからの終了要求を認識して終了しない場合、RTOS-WinJはアプリケーションの状態にかかわらず、アプリケーションの実行を停止させます。

2.6 デバッグトレース

RTOS-WinJでは、デバッグをサポートするイベントトレース機能を提供しています。
RTOS-WinJトレースユーティリティによりカーネルのデバッグバージョンを実行し、さまざまな種類のイベントをリアルタイムで記録することができます。

トレース機能に関して、Pentiumプロセッサまたはそれ以上のプロセッサではマイクロ秒単位（百万分の1秒単位）の精度で、イベントをタイムスタンプします。（486 プロセッサでは千分の1秒単位）

トレース機能によって以下のイベントを記録します。

- Enter Interrupt
- Leave Interrupt
- Enter Kernel Call
- Leave Kernel Call
- Enter System Timer
- Leave System Timer
- Enter User Timer
- Leave User Timer
- Context Switch To RTOS-WinJ
- Context Switch To NT
- Enter Processor Fault
- Leave Processor Fault
- User Defined Event

RTOS-WinJトレースユーティリティは、トレース機能をコントロールすることができます。
また、RTOS-WinJ APIの関数を使用した実行中のRTOS-WinJアプリケーションのトレースをプログラムに基づいてコントロールすることができます。

- | | | |
|--------------------------|---|------------------------------------|
| ▪ rwTraceDataAvailable | - | トレースバッファの使用可能なレコード数を取得 |
| ▪ rwSetTraceBufferMode | - | トレースバッファのモード(one-shot/circular)を設定 |
| ▪ rwSetTraceBufferSize | - | トレースバッファのサイズを設定 |
| ▪ rwStartTraceMonitoring | - | スタートトリガの検索開始 |
| ▪ rwStopTracing | - | トレースの停止（トリガの検索停止） |
| ▪ rwSetTraceTriggers | - | トレースするイベントトリガを設定 |
| ▪ rwDebugTraceEvent | - | ユーザー定義のトレースイベントを設定 |
| ▪ rwDumpDebugTrace | - | トレースバッファ内容をファイルに出力 |
| ▪ rwSetTraceDumpFile | - | トレースバッファ内容を出力するファイルを設定 |
| ▪ rwGetTraceState | - | 現在のトレース状態を取得 |
| ▪ rwGetTraceBufferSize | - | トレースバッファのサイズを取得 |
| ▪ rwGetTraceBufferMode | - | 現在のバッファモードを取得 |
| ▪ rwGetTraceTriggers | - | 現在のスタートおよびストップトリガを取得 |

スタートおよびストップトリガはイベントのトレースをコントロールします。
ユーザーは、複数のスタートトリガとストップトリガを設定することができます。
(スタートトリガは最低1つ設定します。ストップトリガは設定しなくてもかまいません。)

トレースを開始するトリガを構成している要素は、トレース可能な通常のイベントです。
スタートトリガによりトレースが開始されると以下のイベントのいずれかが発生するまでトレース可能なすべてのイベントをトレースバッファに記録します。

- ・ バッファモードがone shot時のバッファフル
- ・ ストップトリガの発生
- ・ RTOS-WinJアプリケーションによるrwStopTracingの呼出し
- ・ WindowsアプリケーションによるHyperShare APIのrwStopTracingの呼出し
- ・ ユーザーによるRTOS-WinJトレースユーティリティからのトレース停止

デバッグトレースユーティリティ

RTOS-WinJトレースユーティリティを使用してRTOS-WinJシステム内のさまざまなイベントをリアルタイムでトレースすることができます。

ユーティリティはメモリ内のトレースバッファにイベントを保存してからディスクに出力します。

トレースセッションが終了すると、ユーティリティはイベントを検査します。

Pentiumプロセッサまたはそれ以上のプロセッサではマイクロ秒単位（百万分の1秒単位）の精度で、イベントをタイムスタンプします。（486プロセッサでは千分の1秒単位）

2.7 その他

RTOS-WinJは、多くのユーザーにとって必要なファイルI/Oやハードウェアアクセスをはじめとした多目的機能を備えています。ユーザーは、ハードウェアにアクセスするためにI/Oポートの読み込み・書き込みを行う場合があります。ハードウェア割込みを処理するインタラプトサービスルーチン (ISR) を登録する場合があります。

あるいはアドレスのマッピングやメモリの書き込みを行い、メモリにマップされたデバイスをアクセスすることもあります。

このようなトピックは、提供しているサンプル内において説明があります。

付録E サンプルプログラムをご覧になるか、インストールディレクトリのソースファイルを参照してください。

RTOS-WinJは、以下のような機能も提供しています。

- ・ **INIファイルサポート**

RTOS-WinJは、新しい2つのAPI関数によりWindows INIファイルのサポートを提供しています。

この新しい関数`rwGetPrivateProfileString`及び `rwWritePrivateProfileString`は、Win32 関数の`GetPrivateProfileString`及び`WritePrivateProfileString`と同じ機能です。

- ・ **ファイルハンドリング関数**

RTOS-WinJは、`rwCopyFile`、`rwRenameFile`及び`rwRemoveFile`を含むいくつかのファイル処理関数を提供しています。

- ・ **フロントエンドに依存しない実行**

RTOS-WinJは、Windows XPフロントエンドのコントロールとは独立して動作します。HyperShare APIの2つの関数`rwAttachDebugWindow`及び`rwDetachDebugWindow`によりRTOS-WinJアプリケーションの実行に影響することなくRTOS-WinJフロントエンドプログラムを開始及び停止することができます。

- ・ **Windows XPからのスレッドコントロール**

Windows XPフロントエンドは、HyperShare APIの関数により実行中のRTOS-WinJアプリケーションのスレッド操作を確実にコントロールします。スレッドの一時停止・終了・再開は、`rwResume`、`rwSuspend`、`rwSetPriority`及び`rwKillThread`によりスレッドの優先度を変更して行います。さらにRTOS-WinJアプリケーションの新しいプロセスの作成は、`rwCreateProcess`を使用します。

2.8 警告

コンピュータのBIOSシステムの一部としてのアドバンストパワーマネジメント機能が実行中のRTOS-WinJシステムと衝突してしまうことがあります。

従って、起こり得る衝突をあらかじめ避けるために、コンピュータのすべてのパワーマネジメントユーティリティを使用不可にしてからRTOS-WinJアプリケーションを起動してください。

2.9 プロセス間通信

2.9.1 メッセージ

メッセージとはスレッドから他のスレッドに送信されるユーザー定義のデータパケットです。

スレッドが他のスレッドにメッセージを送信する場合、受信側のスレッドがメッセージを受信・応答するまで送信側のスレッドはブロック（一時停止および待機）されます。

メッセージ受信待ちのスレッドは、他のスレッドがメッセージを送信するまでブロックされます。応答操作は、スレッドをブロックすることはありません。

以下の例では、スレッドBがスレッドAのメッセージ送信を待ち、メッセージにより実行を再開し、そのあとに応答しています。色付の文はRTOS-WinJの関数です。

次の例では、スレッドAがメッセージ送信後、スレッドBがそのメッセージを受信し、そのあとに応答しています。
スレッドBはブロックされません。

メッセージを受信するスレッドは、ピアツーピアモード (P2P) で動作し、指定のスレッドからのメッセージを受信することができます。もしくは、client/serverモードで動作し、他のスレッドからのメッセージを受信することもできます。

複数のスレッドが無限にブロックされる原因となる状況避けるよう十分に注意してください。スレッドAがスレッドBにメッセージを送信、スレッドBがスレッドCにメッセージを送信、スレッドCがスレッドAにメッセージを送信する状況になると、無限ブロッキングが容易に発生します。

2.9.2 シグナル

シグナルは、スレッド間通信の一種であり、メッセージより優先される基本的なノンブロッキング同期式通信方法です。

シグナルは、送信および受信スレッドがある点ではメッセージと非常に似ています。

(スレッドXが`rwSignal`関数を使用してスレッドYにシグナルを送信することは、スレッドXが`rwSend`関数を使用してスレッドYにメッセージを送信することと同じです。)

シグナルにはデータが存在しません。また、送信スレッドがブロックされません。この点がメッセージとは異なります。

第三章 RTOS-WinJランタイムマネージャ

RTOS-WinJ RunTime Manager (RTM) では、次にあげる機能を提供しています。

- ・ Windows XPユーザーレイヤ内からのRTOS-WinJアプリケーションの実行
- ・ RTOS-WinJアプリケーションからの出力メッセージのモニタリング
- ・ RTOS-WinJアプリケーション全スレッドの関数操作のステータスのモニタリング
- ・ RTOS-WinJ共有メモリのモニタリングおよび変更
- ・ DDEサーバとしての動作
- ・ プロセスの特徴的な機能の経過時間表示
- ・ RTOS-WinJのバージョン情報・リソースのヘルプ・設定オプションの表示

RTOS-WinJ ランタイムマネージャ (RTM) は、複数の RTOS-WinJ プロセスの識別・統計データを表示するグラフィカルユーザーインターフェイス (GUI) です。

RTM では RTOS-WinJ の共有メモリを全て検索・表示することが可能です。

3.1 WindowsからのRTOS-WinJアプリケーションの起動

スタートメニューの[すべてのプログラム]-[RTOS-WinJ]-[RTOS-WinJ Runtime Manager]を選択して RTM を起動することができます。あるいはコマンドプロンプトから RTM を起動することもできます。
起動時の RTOS-WinJ ランタイムマネージャの画面は、Threads タブの画面を表示します。

コマンドラインからの起動

引数を使用してRTMを起動することができます。起動方法は以下の通りです。

```
rwRTM.Exe [flag1 [arguments1]] [flag2 [argument2]]...
```

RTMのパラメータ(フラグ)及び引数

- ・ /s <pathAndFileName.exe>
- ・ /c <pathAndFileName.exe> <stackSize> <processName> <priority>
- ・ /noconfirm

引数で使用するパスは、絶対パスを使用してください。

/sスイッチ

RTOS-WinJプロセスが実行していない場合、RTOS-WinJアプリケーションを起動するときに使用します。

例えば、

```
rwrtm.exe /s "c:\Program Files\RTOS-WinJ\bin\rwsmp13.exe"
```

/cスイッチ

RTOS-WinJが既に実行中の場合、RTOS-WinJアプリケーションやプロセスを起動するときに使用します。

この場合、スタックサイズ、スレッド（プロセス）名、及びスレッドの優先度を指定します。

/cスイッチは、rwCreateProcess関数を実行します。

そのため、コマンドを入力する時にRTOS-WinJが実行されていなければなりません。

/cコマンドは、/sコマンドに続いて入力する場合があります。これは、/sコマンドがrwStartApplication関数を実行するためです。

```
rwrtm.exe /s "c:\Program Files\RTOS-WinJ\bin\rwsmp13.exe"  
/c "c:\Program Files\RTOS-WinJ\bin\rwsmp11.exe" 23000 rwsmp1 9
```

/noconfirmフラグ

ユーザーがRTMを使用してRTOS-WinJアプリケーションを起動する場合、確認ダイアログを表示しません。

/noconfirmフラグを使用している場合、“Are you sure you want to exit?” ダイアログは表示されません。

/noconfirmコマンドは、コマンドのあとに続くプロセスにだけに適応されます。

例えば、

```
rwrtm.exe /noconfirm /s "c:\Program Files\RTOS-WinJ\bin\rwsmp13.exe"
```

引数のはじめを示す/以外のデリミタ（区切り符号）は不要です。

しかし、ファイル名のパスにスペースがある場合は二重引用符が必要です。

3.1.1 パスワード保護

ThreadsタブのPasswordボタンをクリックして、スレッドを終了させるための Password Protectionポップアップダイアログボックスを表示して下さい。

パスワード保護を設定するには、

- ・ Change statusボタンをクリックしてPassword protection statusダイアログボックスを表示します。
- ・ Enable password protectionチェックボックスをチェックしOKをクリックして下さい。現在のパスワード保護ステータスがENABLEDにかわります。

パスワード保護ステータスがENABLEDの状態でChange Passwordボタンをクリックし、新しいパスワードの入力あるいは既存のパスワードの変更を行ってください。

Change Passwordダイアログボックスにおいて

- ・ 確認のため、古いパスワード（設定している場合）を入力してください。
- ・ 20文字以内で新しいパスワードを入力してください。
- ・ 確認のため、再度新しいパスワードを入力しOKをクリックしてください。

新しいパスワードの入力によりCurrent password statusがVALIDになります。Current password statusがVALIDの状態でパスワード保護を無効にする場合は、現在のパスワードを入力して確認することが必要です。

3.1.2 RTOS-WinJアプリケーションの起動と停止

RTOS-WinJアプリケーションを起動するには、RTOS-WinJ Application editボックスにおいて、プログラムの絶対パスとファイル名を入力します。

ユーザーはBrowseボタンを使用し、ファイルを選択することもできます。Startボタンで選択したアプリケーションを起動します。

選択したファイルは、HyperShare APIを使用していないRTOS-WinJアプリケーションでなくてはならないことに注意してください。

HyperShare APIを使用しているWindows XPアプリケーション及びRTOS-WinJアプリケーションは、RTMのユーザーインターフェイスから実行することはできません。

RTOS-WinJシステムが稼働している場合、Start RW Treadダイアログボックスが表示され、（メインスレッドである）新しいアプリケーション用のパラメータを入力するよう指示されます。Stack size、Thread Name（16文字以内）、及び0から31までのスレッドのPriority（優先度）を入力して下さい。

RTOS-WinJアプリケーションを停止するには、Stop RTOS-WinJボタンをクリックします。このボタンは全スクリーンに表示され、実行中のすべてのアプリケーションを停止し、RTOS-WinJシステムをシャットダウンするためのものです。

3.2 出力メッセージの表示

RTOS-WinJ RTMには、テキストウィンドウにプロセスの出力情報（デバッグ情報）を表示するアウトプット表示画面があります。RTMウィンドウにあるOutputタブをクリックして画面にアクセスします。現在実行中のすべてのプロセスがこの画面に表示されます。Print Displayボタンを使用して、現在表示されている画面の内容を印刷する事ができます。Clearボタンで画面をクリアにすることもできます。

3.3 スレッドステータスの表示と変更

スレッドステータスのモニタリング

RTOS-WinJ RTMでは、複数の処理とスレッドに関する識別・統計情報を同時に表示します。Threadsタブにアクセスし、プロセスやスレッドを確認してください。次にあげるような個々のスレッド情報を確認することができます。

- ・ スレッド名 (Thread name)
- ・ スレッド ID (TID : 0-1023)
- ・ 親スレッド ID (Parent TID)
- ・ スレッドの優先度 (0-31)
- ・ スレッドの状態 (READY, RECEIVE, SEND, SUSPEND, WAIT)
- ・ 関連スレッド ID (AssocTID: 送信によるブロックが発生していない場合、送信元の ID)
- ・ ペンディングシグナル (Signals)
- ・ スタックサイズ (Stack)

スレッド関数の操作

Threadsタブウィンドウに表示されているスレッドを右クリックし、スレッドを編集することができます。スレッドを終了・停止・再開するオプション及びスレッドの優先度を変更するオプションを表示したポップアップウィンドウが表示されます。

どのオプションを選択しても、選択したスレッドの編集に関する警告が表示されます。

3.4 メモリ内容の表示

RTOS-WinJ RTMでは、様々な型のユーザー共有メモリの内容を表示します。Memoryタブをクリックし、RTOS-WinJのメモリ内容を表示してください。

メモリ表示では、独特のコントロールをいくつか使用しています。このコントロールは、Data Display Optionsです。画面の表示内容の変更及び実際のメモリの値の変更を行うことができます。

Typeの選択

RTOS-WinJのメモリ及びユーザー共有メモリをTypeのドロップダウンリストから選択し表示することができます。

Sizeの選択

RTOS-WinJの4つのサイズのメモリを表示することができます。

Bit・Byte・Word・Longの4種類です。

Bitは16384個、Byte、Word、Longは2048個です。Sizeのドロップダウンリストから表示するサイズを選択してください。デフォルトはLongです。

Formatの選択

符号なし10進数、符号付10進数、16進数、2進数、8進数、またはBitのフォーマットを使用して、選択したメモリのタイプやサイズを表示することができます。

Formatのドロップダウンリストから表示したいメモリのフォーマットを選択して下さい。

メモリ内容の変更

アプリケーション実行中に、メモリ内容を変更することが可能です。メモリウィンドウの Offset列に表示されているメモリ変数のアドレスを右クリックしてください。ポップアップウィンドウには、変更するアドレスのオフセット値の選択が表示されます。

3.5 名前付メモリ

RTOS-WinJ RTMでは、様々なフォーマットでRTOS-WinJの名前付メモリの値を表示することができます。

Named Memory SegmentsタブをクリックしてRTOS-WinJの名前付メモリの内容を表示してください。選択された名前付メモリの内容は、ウィンドウの右側に表示されます。Contents Display Optionsを使用して選択したメモリの表示方法を変更することができます。

Sizeの選択

RTOS-WinJの4つのサイズのメモリを確認することが可能です。

Byte・Word・Long・Floatの4種類です。Byte、Word、Longは、2048個です。

Data sizeのドロップダウンリストから表示したいメモリのサイズを選択して下さい。

デフォルトはLongです。

Formatの選択

符号付10進数、16進数、2進数、8進数、Bit、またはFloatのフォーマットを使用して選択した名前付メモリを表示することができます。

Data Formatのドロップダウンリストから表示したいメモリのフォーマットを選択して下さい。

3.6 ダイナミックデータエクスチェンジ (DDE)

RTMは、ダイナミックデータエクスチェンジ (DDE) サーバーとして機能します。これは、他のアプリケーションがRTOS-WinJ共有メモリのデータを読み書きしてRTOS-WinJに関する情報をリレーするものです。RTMは、DDEMLとインターフェイスを取ることによってRTOS-WinJのデータを表示します。設定されたサービスネーム、トピック、アイテムの文字列を使用することによりDDEクライアントソフトウェアを書き込むことが可能であり、Microsoft Excelなどの既存のプログラムにネームを追加することも可能です。

DDEからリレーされる情報のフォーマットは、以下のもので構成されています。

service_name / topic_name! 'identifier'

identifierの前後には引用符が必要ですので注意してください。

サービスネーム

DDE通信を確立する時にRTMは以下のサービスネームをサポートします。

- ・ HyperLink
- ・ HyperLnk (8文字以内 : MS-DOSの実行可能ファイル名称の制限)
- ・ RWRTM (8文字以内 : MS-DOSの実行可能ファイル名称の制限)

3.7 パフォーマンスの表示

RTMでは一定時間、プロセスのパフォーマンスを表示します。

RTMのPerformanceタブをクリックしてシステムパフォーマンス画面にアクセスしてください。このシステムパフォーマンス画面では、その時点でのRTOS-WinJのプロセスに関する統計を表示します。Turn On Performance MonitoringをチェックするとRTOS-WinJ RTMは、RTOS-WinJに割り当てられているCPU時間をベースにした統計を表示します。以下のものは、絶えず更新されるフォーマットです。

- ・ % RTOS-WinJ CPU Urwge By Priority の棒グラフは、プロセスの優先度別にCPUを占めるRTOS-WinJの使用率を表示します。個々の棒グラフは0から31までの優先度を表します。
- ・ % RTOS-WinJ CPU Urwge の棒グラフは、現在実行中のRTOS-WinJプロセスが使用しているCPUの時間の割合を表します。
- ・ % RTOS-WinJ CPU Urwge History の折れ線グラフは、RTOS-WinJが使用したCPUの履歴を表します。

3.8 RTOS-WinJシステム情報の取得と操作

RTOS-WinJ RTMでは、RTOS-WinJのバージョン情報を表示するオプション画面があります。RTMウィンドウのOptionタブをクリックしてオプション画面にアクセスしてください。オプション画面では次のことができます。

- ・ AboutボタンのクリックでRTMに関する情報を示すaboutボックスを表示します。
- ・ HelpボタンのクリックでRTOS-WinJオンラインヘルプを表示します。
- ・ ConfigureボタンのクリックでRTOS-WinJ Configuration Wizardが起動します。
RTOS-WinJシステムの環境設定を確認・変更することができます。
- ・ SettingsボタンのクリックでRTOS-WinJ configuration Settingsのクイックビューを表示します。
- ・ TraceボタンのクリックでRTOS-WinJ Trace Utilityが起動します。

第四章 RTOS-WinJトレースユーティリティ

RTOS-WinJにはデバッグをサポートするイベントトレース機能があります。

カーネルのデバッグバージョンは、RTOS-WinJトレースユーティリティにより起動され、様々なイベントをリアルタイムで記録します。

トレース機能は、RTOS-WinJトレースユーティリティによりコントロールされ、**RTOS-WinJ API**機能を使用する実行中のRTOS-WinJアプリケーションをプログラムに基づいてコントロールします。

RTOS-WinJトレースユーティリティによりRTOS-WinJシステム内の様々なタイプのイベントをリアルタイムにデバックすることが可能です。

トレースユーティリティでは、メモリのトレースバッファにイベントを保存し、そのバッファをトレースセッション終了後、検査してディスクへ書き込みます。

このイベントは、Pentiumプロセッサまたはそれ以上のプロセッサではマイクロ秒単位（百万分の1秒単位）の精度で、イベントをタイムスタンプします。（486プロセッサでは千分の1秒単位）

4.1 トレースユーティリティを使用可能にする

Trace UtilityのメインウィンドウにあるEnable TraceチェックボックスをチェックすることによりRTOS-WinJシステムは、カーネルの特別なバージョンを実行するようになっています。このカーネルは、トレース可能なイベントをトラップおよび記録するフックを備えています。カーネルのトレースバージョンは、標準カーネルよりも実行速度が遅いため、デバッグを目的としたトレースセッションを実行する場合のみEnable Traceチェックボックスをチェックしてください。


注意 : RTOS-WinJアプリケーションを起動した時、RTOS-WinJシステムはカーネル標準バージョンとトレースバージョンの実行する・しないを決定します。そのため、システムがRTOS-WinJアプリケーションを実行している間は、トレースを実行することができません。実行中のRTOS-WinJアプリケーションを停止・再スタートするまでは、トレースによる影響はありません。

4.2 トレースユーティリティの起動

トレースカーネルは、イベントをトレースバッファに記録するタイミングを決定するためにスタートトリガ及びストップトリガの両方を使用します。カーネルは、イベントのモニタリングを開始するまでデータをトレースバッファに記録しません。次の 3 つの方法の内、1 つを使用してイベントのモニタリングを開始することが可能です。(これらの方法を行うには **Enable Tracing** チェックボックスをチェックしてください。)

- ・ Trace Utiliy ウィンドウの File メニューから Start Monitoring を選択します。



あるいは、Start Monitoring ボタン  をクリックします。

このオプションは、アプリケーションが実行中の場合のみ操作可能です。

- ・ Trace Utility のメインウィンドウの Auto Start Monitoring チェックボックスをチェックし、RTOS-WinJ アプリケーションを実行してください。
このオプションは、モニタリングの開始と共に RTOS-WinJ アプリケーションを起動します。これは、システムが起動してから最初の数秒内で起こるイベントを記録・検出するのに非常に有効です。
- ・ RTOS-WinJ API の関数 **rwStartTraceMonitoring** を使用している RTOS-WinJ アプリケーションを実行してください。


4.3 開始・停止トリガの選択

トレースが使用可能状態でモニタリングが開始されている時にRTOS-WinJアプリケーションが起動されるとカネールは、スタートトリガイベントの検索を開始します。スタートトリガは、イベントをトレースバッファに記録しはじめるタイミングを決定します。

もっともシンプルな形のスタートイベントは、Start Monitoringイベントです。このイベントは特別なイベントであり、イベントのモニタリングを開始すると同時にトレースバッファにイベントの記録が開始されます。

Start Monitoringイベントがスタートトリガとして設定していない場合は、設定したスタートトリガの1つが発生するまでトレースバッファに記録されるイベントはありません。例えば、スタートトリガのイベントとしてEnter Interruptを設定する場合があります。

イベントの記録が開始されると、以下のイベントの1つが発生するまで一連のイベントすべてをトレースバッファに記録します。

- ・ バッファモードが one shot 時のバッファフル
- ・ ストップトリガの発生
- ・ RTOS-WinJ アプリケーションによる `rwStopTracing` の呼出し
- ・ Windows アプリケーションによる **HyperShare API** の `rwStopTracing` の呼出し
- ・ RTOS-WinJ トレースユーティリティのメインウィンドウからの File → Stop Monitoring
- ・ メニューオプションの選択、ツールバーボタンのStop Monitoringの  のクリック

トレースが停止されると記録されたイベントを設定した出力ファイルに出力します。トレースユーティリティが自動的にバイナリ出力ファイルを GSV テキストファイルに変換後、適切なヘッダと共にテキストファイルを表示します。

注意 1 : モニタリングを開始してある状態でRTOS-WinJプロジェクトの実行を停止した場合、ディスクに出力されるファイルはありません。出力ファイルを作成するためには、モニタリングを停止してからRTOS-WinJプロジェクトの実行を停止してください。

注意 2 : モニタリング開始後、既に複数のイベントが記録されている時にRTOS-WinJアプリケーションのスレッド0においてプロセッサフォルトが発生した場合、アプリケーションは正常にシャットダウンし、出力ファイルは自動的にディスクへ出力されます。その後、ユーザーが出力ファイルを選択し、トレースユーティリティでファイルを表示することができます。ユーティリティは、このファイルをバイナリファイルであると認識し、表示するためにバイナリファイルをテキストファイルへ自動的に変換します。スレッド0以外のスレッドにおいてプロセッサフォルトが発生した場合、RTOS-WinJシステムは引き続き実行され、イベントのモニタリングや記録が通常通り続けられます。

4.3.1 スタートおよびストップトリガイベント

下の表は、トレースカネールが記録することができるイベントです。これらのイベントは、スタート及びストップトリガとしてすべて使用することができますが、特別なイベントであるStart Monitoringは、スタートトリガとしてのみ使用することができます。

イベント	説 明
Enter Interrupt	RTOS-WinJ割込みルーチンの開始
Leave Interrupt	RTOS-WinJ割込みルーチンの終了
Enter Kernel Call	RTOS-WinJAPI コールの開始
Leave Kernel Call	RTOS-WinJAPI コールの終了
Enter System Timer	RTOS-WinJシステムタイマーの開始
Leaves System Timer	RTOS-WinJシステムタイマーの終了
Enter User Timer	ユーザー付属タイマ (<code>rwAttacrwystemTimer</code>) の開始
Leave User Timer	ユーザー付属タイマ (<code>rwAttacrwystemTimer</code>) の終了
Context Switch To RTOS-WinJ	Windows XP→RTOS-WinJの切換え発生
Context Switch To NT	RTOS-WinJ→Windows XPの切換え発生
Enter Processor Fault	プロセッサフォルトルーチンの開始
Leave Processor Fault	プロセッサフォルトルーチンの終了
User Defined Event	RTOS-WinJアプリケーションの <code>rwDebugTraceEvent</code> の呼び出し

さらにRTOS-WinJ API及びHyperShare APIには、イベントのトレースや記録をコントロールすることができる多くの機能があります。

4.4 トレースユーティリティSettingsタブ

メニューオプション

Fileメニューには次のオプションがあります。

- | | |
|--------------------|---|
| ・ Start Monitoring | トリガ/イベントのモニタリングの開始 |
| ・ Stop Monitoring | トリガ/イベントのモニタリングの停止、イベントの記録の停止 |
| ・ Print File | 情報タブに表示されているテキストファイルの内容の印刷
(テキストファイルが表示されている場合に有効) |
| ・ Exit | アプリケーションの終了 |

Tool メニューの **Runtime Manager** オプションは RTOS-WinJ RTM アプリケーションを起動するものです。

Help メニューの About RTOS-WinJ Trace オプションはアプリケーションのバージョン情報を表示するものです。

トレースユーティリティの設定

トレースユーティリティメインウィンドウのBuffer InformationにModeとNumber of Recordがあります。

ModeはトレースバッファのモードをCircularまたはOne Shotから選択することができます。

Circularモードはバッファをサイクリックに使用します。バッファの空きが無くなると一番古いイベントを上書きし、引き続きイベントをモニタリングします。

One shotモードは、バッファに空きが無くなるとイベントのモニタリングを停止します。Number of Recordはレコード数によってトレースバッファのサイズを決定します。

また、Trace Utilityのメインウィンドウには、Debug Trace Informationにfilenameがあります。Browseボタンをクリックしてダイアログボックスを表示してください。

出力ファイルのファイル名を入力することができます。選択したファイル名は読み取り専用として表示されます。

トレースユーティリティトリガ

Start Eventsとは、イベントのトレースバッファへの記録開始となるトリガの選択肢リストです。Stop Eventsは、イベントのトレースバッファへの記録停止となるトリガの選択肢リストです。

4.5 Historical Trace File Informationタブ

Historical Trace File Informationタブでは、出力ファイルのパスとファイル名、ファイル作成日時、ならびにファイル内の総記録数を表示します。

出力ファイルには、記録No、記録したトリガの種類、トリガの詳細（トリガに関連している関数呼び出しなど）、記録が書き込まれた時のマイクロ秒単位での実行時間、記録が書き込まれた時に実行していたTID、その他の情報（関数終了時の戻り値など）があります。

他の出力ファイルを確認したい場合は **Clear Grid** ボタンをクリックして画面をクリアしてから **Open File** ボタンをクリックし、表示する出力ファイルを選択します。

第五章 RTOS-WinJ 設定ウィザード

RTOS-WinJ設定ウィザードはWindows XPのアプリケーションであり、RTOS-WinJをシステムに設定することができます。RTOS-WinJをインストールするとウィザード画面が表示され、システムをコントロールするWindows XPの変更を行うことができます。また、システムにインストールしたあとでも変更を行うためにRTOS-WinJ設定ウィザードを実行することができます。RTOS-WinJランタイムマネージャ(RTM)ウィンドウ上でOptionsタブをクリックしてください。このオプションウィンドウでconfigureボタンをクリックします。RTOS-WinJ設定ウィザードを実行する時は、管理ユーザー権限が与えられていることを確認してください。

5.1 RTOS-WinJ設定ウィザードの開始

RTOS-WinJのインストール中にRTOS-WinJ設定ウィザードは自動的に実行されます。表示されている値を直接変更するか、または値の右側にあるスピンコントロールを使用し変更します。

システム設定、標準タイミング設定、メモリボードの設定、Windowsのブートメニューの選択、及び通信用ポートの設定の各設定を行う事が可能です。

設定が終了すると現在の設定が表示されます。必要であれば、変更したいウィンドウに戻り変更してください。設定した内容が正しい場合はリブートシステムに進んでください。

RTOS-WinJをインストールしたあとに設定をする場合、RTOS-WinJ設定ウィザードのウィンドウのうちいくつかは、インストール中に表示されるウィンドウとは若干の違いがあることに注意してください。

インストール中の場合、Cancelボタンはどのウィンドウにも表示されません。

RTOS-WinJ RTMからウィザードを実行する場合はCancelボタンをクリックすると変更をキャンセルしRTMに戻ります。

Nextを押して、次のページに進んでください。

5.2 システム設定

System Settingsウィンドウには、Windows XPで使用可能な全メモリ量の物理RAM の表示があります。

Windows XPが全物理メモリの一部にのみアクセス可能なようにシステムが設定されている場合があります。このウィンドウに表示されているメモリ量は、アクセス可能なメモリのことではなく、システムにインストール済みの全物理RAMであるということを必ずご確認ください。

注意: このオプションがチェックされれば、ドライバメモリの設定も使用可能になります。
(以下をご覧ください。)

System Memory設定では、システム起動時にRTOS-WinJへ割り当てられるメモリ量をコントロールします。

- **Physical RAM**では、システムにインストールされている全メモリ量を表示します。
Windowsが全物理メモリの一部にのみしかアクセス可能なようにシステムが設定されている場合があります。表示されているメモリ量は、システムにインストールされている全物理RAMであるということをご確認ください。
- **RTOS-WinJ Memory**では、ユーザーのアプリケーション用にRTOS-WinJが使用可能なRAM量を設定することができます。デフォルト値は2MBですが、特別大きい場合や複雑なコントロールアプリケーションを実行しようとする場合は、値を上げることができます。
- **Driver Memory**は、TCP/IPサポートが可能である場合、イーサネットNICドライバに割り当てられるメモリ量です。（上図を参照してください）必要なメモリ量は、選択されるドライバの種類によって変わります。（ドライバ自体は、**Ethernet TCP/IP Configuration Wizard**を使用して選択・設定します）
ODIドライバ用のドライバメモリは1MB必要です。NE2000ドライバ用のドライバメモリは必要ありません。

注意： RTOS-WinJに割り当てられた総メモリ = (RTOS-WinJメモリ + ドライバメモリ) です。
このメモリは、システム起動中は保護され、どのような方法でもWindowsがアクセスすることはできません。

Initial Thread Stack Sizeは、アプリケーションによって作成された第1スレッドの初期スタックサイズを示します。アプリケーションによって作成されたその他全てのスレッドは、APIパラメータリストでスタックサイズの数値を入力しなくてはなりません。デフォルト値は32Kです。

RTOS-WinJ Signaling Interruptは、RTOS-WinJとWindows XP間の通信を扱うために使用されるシステム割込み (IRQ) を示します。この値はシステム内で未使用のIRQでなければなりません。この値は必ず選択してください。

IRQを選択せずに次のウィンドウに進もうとした場合や、Detailsボタンをクリックした場合、Windows XPシステム情報ウィンドウのサンプルが表示されます。この説明を読み、Nextを押して次に進んでください。

Windows XPのシステム情報ページが表示されます。IRQを選択してシステムが現在使用しているIRQを表示してください。

システムが現在使用しているIRQを確認し、が使用可能な割込みを見つけてください。

確認後、システム情報ウィンドウを閉じ、RTOS-WinJ signaling interruptに使用可能なIRQをセットしてください。

Nextを押して次のページに進みます。

Backを押すと前のページに戻ります。

インストール時には、Cancelボタンは表示されません。

RTMからウィザードを実行している場合にCancelを押した時は、全ての変更した内容をキャンセルしRTMに戻ります。

5.3 標準タイミングの設定

Standard Timing Settingsウィンドウでは、Windows XPとRTOS-WinJに与えられる処理時間の合計の値が表示されます。

この数値により、RTOS-WinJとWindows XPのパフォーマンスを変更することができます。標準チック率を変更するには、スライダーコントロールを設定する数値の位置までクリック・アンド・ドラッグします。

デフォルト値は250 μ sです。

Nextを押して次のページに進みます。

Backを押すと前のページに戻ります。

インストール時には、Cancelボタンは表示されません。

RTMからウィザードを実行している場合にCancelを押した時は、全ての変更した内容をキャンセルしRTMに戻ります。

5.4 メモリボードの設定

Memory Board settings ウィンドウでは、リアルタイムサブシステムカードの設定を表示します。

カードの[使用可／不可]を設定するためには、Retentive Memory Enabled ボックスをチェックします。

カードの使用可を設定した場合、SRAM ベースアドレス及び I/O ベースアドレスはカードのスペックシートを基本にして設定してください。

SRAM ベースアドレスのデフォルト値は D8000、I/O ベースアドレスのデフォルト値は 300 です。

Nextを押して次のページに進みます。

Backを押すと前のページに戻ります。

インストール時には、Cancelボタンは表示されません。

RTMからウィザードを実行している場合にCancelを押した時は、全ての変更した内容をキャンセルしRTMに戻ります。

5.5 Windows ブートメニューの選択

Windows Boot Menu Selections ウィンドウでは、RTOS-WinJ Enable を含むすべての RTOS-WinJ のメニュー選択肢が表示されます。

システムの再起動時、Windows XP のブートメニューに表示されている選択用メニューが表示されます。

システムを起動してRTOS-WinJを実行しようとする場合、ブートメニューが表示されている間にRTOS-WinJ Enabledのメニューアイテムを選択してください。

Nextを押して次のページに進みます。

Backを押すと前のページに戻ります。

インストール時には、Cancelボタンは表示されません。

RTMからウィザードを実行している場合にCancelを押した時は、全ての変更した内容をキャンセルしRTMに戻ります。

5.6 現在の設定

Current Settings ウィンドウは、ユーザーが選択した設定内容が表示されます。変更箇所がある場合は、変更しようとしている設定のウィンドウが表示されるまで Back ボタンをクリックし、現在の設定を変更することができます。変更後は Next ボタンをクリックして Current Settings ウィンドウに戻ってください。そして正しく変更されていることを確認してください。

インストール時に Finish ボタンをクリックすると **Reboot Your System** ウィンドウになります。

RTM からウィザードを実行してシステムメモリ設定を変更した場合のみ Finish ボタンをクリックすると Reboot Your System ウィンドウが表示されます。

システム設定を変更していない場合に Finish ボタンをクリックするとシステム設定以外の変更を保存して RTMに戻ります。

警告： Current Settings ウィンドウで Finish ボタンをクリックすると設定が保存されることに御注意ください。Finish ボタンをクリックすると変更前の設定に戻すことができなくなります。

5.8 リブートシステム

RTOS-WinJ 設定ウィザードの最後のウィンドウです。

Reboot Your System ウィンドウでは、システムを再起動して設定した内容を実行することができます。

RTOS-WinJ をインストール後に、この画面にて再起動を行わない場合はユーザーが再起動をしなければなりません。

警告： システムを再起動しない場合、正常に RTOS-WinJ のシステムを実行できない場合がありますので御注意ください。
インストール後に再起動することをお勧めします。

第六章 RTOS-WinJ アプリケーションの開発

RTOS-WinJ アプリケーションは、標準 C アプリケーションと同じ方法で作成してください。実行のコントロール及びユーザーが作成した他のサブルーチンの呼び出しを行う `main()` 関数を記述してください。

そしてコンパイルとリンクを行い、実行可能プログラム (.exe ファイル) を作成してください。

実行可能プログラム (.exe ファイル) は RTM ユーザーインターフェイスアプリケーションから実行することができます。

利用可能な関数については、[6.1 RTOS-WinJ API](#) をご参照ください。

Microsoft Developers Studio でコンパイルする RTOS-WinJ プログラムの作成方法の例に関しては、[6.2 RTOS-WinJ のプログラムを作成する](#) をご参照ください。

更に [6.3 システム関数の使用](#) 及び [6.4 コンパイラの設定](#) もご覧ください。

6.1 RTOS-WinJ API

- : インタラプトサービスルーチンにおいて使用可能
 × : インタラプトサービスルーチンにおいて使用不可

メモリ

- | | |
|------------------------------|----------------|
| × <code>rwAlloc</code> | メモリの割当て |
| × <code>rwFree</code> | メモリの開放 |
| × <code>rwMapMemory</code> | メモリのデータ領域へのマップ |
| × <code>rwUnmapMemory</code> | マップしたメモリの開放 |

共有メモリ

- | | |
|------------------------------------|-----------------|
| ○ <code>rwLocateMemory</code> | 名前付共有メモリの取得 |
| ○ <code>rwNumberSharedBits</code> | Bit 領域数の取得 |
| ○ <code>rwNumberSharedBytes</code> | Byte 領域数の取得 |
| ○ <code>rwNumberSharedLongs</code> | Long 領域数の取得 |
| ○ <code>rwNumberSharedWords</code> | Word 領域数の取得 |
| ○ <code>rwSharedRam</code> | 共有メモリ先頭ポインタの取得 |
| ○ <code>rwShareMemory</code> | 名前付共有メモリの作成 |
| ○ <code>rwUserSharedRam</code> | ユーザー領域先頭ポインタの取得 |
| ○ <code>rwReadBit</code> | Bit 領域からの読み込み |
| ○ <code>rwReadByte</code> | Byte 領域からの読み込み |
| ○ <code>rwReadLong</code> | Long 領域からの読み込み |
| ○ <code>rwReadWord</code> | Word 領域からの読み込み |
| ○ <code>rwUnshareMemory</code> | 名前付共有メモリ領域の開放 |
| ○ <code>rwWriteBit</code> | Bit 領域への書き込み |
| ○ <code>rwWriteByte</code> | Byte 領域への書き込み |
| ○ <code>rwWriteLong</code> | Long 領域への書き込み |
| ○ <code>rwWriteWord</code> | Word 領域への書き込み |

リテンティブメモリ

- | | |
|---|------------------|
| × <code>rwRetentInit</code> | リテンティブメモリの初期化 |
| × <code>rwRetentRead</code> | リテンティブメモリからの読み込み |
| × <code>rwRetentWrite</code> | リテンティブメモリへの書き込み |
| × <code>rwRetentExit</code> | リテンティブメモリの開放 |
| × <code>rwRetentQueryBatteryStatus</code> | バッテリーの状態調査 |

ファイル I/O

× <code>rwCopyFile</code>	ファイルのコピー
× <code>rwOpen</code>	ファイルのオープン
× <code>rwClose</code>	ファイルのクローズ
× <code>rwRead</code>	ファイルからのデータの読み込み
× <code>rwRemoveFile</code>	ファイルの削除
× <code>rwRenameFile</code>	ファイル名の変更
× <code>rwWrite</code>	ファイルへのデータの書き込み
× <code>rwTell</code>	ファイル内の現在位置（オフセット）の取得
× <code>rwSizeOfFile</code>	ファイルサイズの取得
× <code>rwSeek</code>	ファイル内の指定オフセットへの移動
× <code>rwReadString</code>	ファイルからの文字列の読み込み
× <code>rwWriteString</code>	ファイルへの文字列の書き込み

タイミシング

× <code>rwDelay</code>	スレッドの一時停止
× <code>rwMicroClock</code>	システム起動時からの経過時間取得（マイクロ秒）
× <code>rwMicroSleep</code>	スレッドのスリープ（マイクロ秒）
× <code>rwSleep</code>	スレッドのスリープ（ミリ秒）
○ <code>rwHardwareCounter</code>	プロセッサのカウンタレジスタの参照
○ <code>rwGetTicks</code>	システムチック値の取得
○ <code>rwTime</code>	1970 年 1 月 1 日からの経過時間取得（秒）

スレッドコントロール

× <code>rwCreateProcess</code>	プロセスの作成
× <code>rwCreateProcessEx</code>	パラメータ付プロセスの作成
× <code>rwCreateThread</code>	スレッドの作成
× <code>rwCreateThreadEx</code>	パラメータ付スレッドの作成
× <code>rwEnterCriticalSection</code>	スレッドのスイッチング停止
× <code>rwExitThread</code>	スレッドの終了
× <code>rwForceReady</code>	ブロックされているスレッドの実行
○ <code>rwGetMainTid</code>	メインスレッドのスレッド ID 取得
× <code>rwGetPriority</code>	スレッド優先度の取得
○ <code>rwGetThreadData</code>	<code>rwCreateThreadEx</code> , <code>rwCreateProcessEx</code> からのパラメータの取得
○ <code>rwGetTid</code>	スレッド ID の取得
× <code>rwKillThread</code>	スレッドの実行終了
× <code>rwLeaveCriticalSection</code>	スイッチングの停止解除
× <code>rwLocateThread</code>	名前付スレッドのスレッド ID 取得
× <code>rwParentTid</code>	スレッドの親スレッド ID 取得
○ <code>rwReleaseProcessor</code>	CPU 資源の開放
× <code>rwResume</code>	スレッドの実行再開
○ <code>rwSeizeProcessor</code>	CPU 資源の独占（Windows XP 実行不可）
○ <code>rwSetPriority</code>	スレッド優先度の設定
× <code>rwSuspend</code>	スレッドの実行の一時停止

- `rwThreadState` スレッドの実行状態の取得
- × `rwYield` タイムスライス残部分の放棄

スレッド間通信 : メッセージ

- × `rwCreceive` メッセージ及びシグナルの受信（ブロック無）
- × `rwReceive` メッセージ及びシグナルの受信（ブロック有）
- × `rwReply` メッセージ送信に対する応答
- × `rwSend` メッセージの送信

スレッド間通信 : セマフォ

- × `rwAllocSemaphore` セマフォの割当て
- × `rwCreateSemaphore` 名前付セマフォの作成
- × `rwDestroySemaphore` セマフォの破棄
- × `rwFreeSemaphore` セマフォの開放
- × `rwLocateSemaphore` セマフォ作成スレッド ID の取得

スレッド間通信 : 新セマフォ関数

- × `rwOpenSemaphore` セマフォのオープン
- × `rwGetSemaphore` セマフォの所有権獲得（ブロック有）
- × `rwTrySemaphore` セマフォの所有権獲得（ブロック無）
- × `rwReleaseSemaphore` セマフォの所有権放棄
- × `rwCloseSemaphore` セマフォのクローズ

スレッド間通信 : シグナル

- `rwClearSignals` 未処理シグナルのクリア
- `rwReadSignals` 未処理シグナル数の取得
- `rwSignal` シグナルの送信

デバッグ

- × `rwDebug` デバッグ文字列の書込み
- × `rwPrint` デバッグ文字列の送信

割込み

× <code>rwInterruptAttach</code>	割込みサービスルーチン (ISR) のアタッチ
× <code>rwInterruptDetach</code>	割込みサービスルーチン (ISR) のデタッチ
○ <code>rwInterruptDirwble</code>	割込みの禁止
○ <code>rwInterruptEnable</code>	割込みの許可

入出力

○ <code>rwInpd</code>	I/O ポートからの Double Word の入力
○ <code>rwInp</code>	I/O ポートからの Byte の入力
○ <code>rwInpw</code>	I/O ポートからの Word の入力
○ <code>rwOutpd</code>	I/O ポートへの Double Word の出力
○ <code>rwOutp</code>	I/O ポートへの Byte の出力
○ <code>rwOutpw</code>	I/O ポートへの Word の出力

トレースデバッグ

○ <code>rwDebugTraceEvent</code>	トレースバッファへのイベントの格納
○ <code>rwDumpDebugTrace</code>	出力ファイルへのトレースバッファ内容出力
○ <code>rwGetTraceBufferMode</code>	トレースバッファモードの取得
○ <code>rwGetTraceBufferSize</code>	トレースバッファサイズの取得
○ <code>rwGetTraceState</code>	トレースモニタリング状態の取得
○ <code>rwGetTraceTriggers</code>	スタート・ストップトリガの取得
○ <code>rwSetTraceBufferMode</code>	トレースバッファモードの設定
○ <code>rwSetTraceBufferSize</code>	トレースバッファサイズの設定
○ <code>rwSetTraceDumpFile</code>	出力ファイルの設定
○ <code>rwSetTraceTriggers</code>	スタート・ストップトリガの設定
○ <code>rwStartTraceMonitoring</code>	トリガ・イベントのモニタリング開始
○ <code>rwStopTracing</code>	トリガ・イベントのモニタリング終了
○ <code>rwTraceDataAvailable</code>	トレースバッファ内の記録数の取得

その他

○ <code>rwAttacrwystemTimer</code>	システムタイマー関数のアタッチ
○ <code>rwDetacrwystemTimer</code>	システムタイマー関数のデタッチ
○ <code>rwGetPrivateProfileString</code>	初期化ファイル (INI ファイル) からの文字列取得
○ <code>rwGetSystemDirectory</code>	Windows システムディレクトリの文字列取得
○ <code>rwGetWindowsDirectory</code>	Windows ディレクトリの文字列取得
○ <code>rwQuitRequest</code>	シャットダウン要求の検出
× <code>rwSetGateSpeed</code>	メッセージチェックスピードの設定
× <code>rwSystemCrash</code>	Windows のクラッシュ検出
× <code>rwTextOut</code>	ブルースクリーンクラッシュ後のテキスト出力
× <code>rwWritePrivateProfileString</code>	初期化ファイル (INI ファイル) への文字列出力

6.2 RTOS-WinJ プログラムの作成

Microsoft Developers Studio 6.0 でコンパイルされる RTOS-WinJ のプログラムを作成します。(アスタリスクマーク(*)が付いているステップは重要であり、注意が必要です。間違えて実行すると深刻なエラーを引き起こす可能性があります。)

Microsoft Developers Studio を開いてください。

- ・ ファイルメニューから[新規作成]で[新規作成]ダイアログを開きます。
- ・ プロジェクトタブを開き、プロジェクト形式の一覧から Win Console Application を選択してください。(すべてのRTOS-WinJ アプリケーションは、Win32 となります)。
- ・ プロジェクト名とファイル式が保存されるフォルダの位置を入力して、OK をクリックしてください。新しいワークスペースが作成されます
- ・ [Win32 Console Application ステップ 1/1]ダイアログにて、[空のプロジェクト]を選択して、[終了]ボタンをクリックしてください。
(新規プロジェクト情報ダイアログの OK ボタンのクリックにより、新規プロジェクトが作成されます)
- * ・ [ビルドメニュー]の[アクティブな構成で設定]にて[プロジェクトの標準構成]ダイアログを開きます。
[プロジェクトの構成]で「xxx - Win32 Release」を選択して[OK]ボタンをクリックします。(デフォルトは Debug です。Debug で RTOS-WinJ のプロジェクトを作成しても機能しません。
※ xxx には、プロジェクト名が入ります。
- ・ [プロジェクトメニュー]の[設定]にて[プロジェクトの設定]ダイアログを開きます
[C/C++] タブの [カテゴリ] のドロップダウンリストから [コード生成]を選択し、[構造体メンバのアライメント]を1バイトに設定してください。(推奨)
- ・ [C/C++]タブの[カテゴリ]から[プリプロセッサ]を選択し、[インクルードファイルのパス]に RTOS-WinJ SDK で用意されているインクルードファイルのディレクトリを加えてください。

RTOS-WinJ SDK の標準セットアップでは、インクルードファイルのディレクトリは
“c:\Program Files\RTOS-WinJ\Inc” です。

- ・ [リンク]タブの[カテゴリ]から[一般]を選択し、[オブジェクト/ライブラリモジュール]を lib.c.lib, kernel32.lib, Hyperlib.lib の3つにしてください。
Hyperlib.lib は、フルパスで入力する必要があります。
通常は “c:\Program Files\RTOS-WinJ\Lib\HyperLib.lib” です。
RTOS-WinJ シリアルポート関数を使用する場合は rwSerialLib.lib も加えてください。

注意: パスの中にスペースが使用されている場合は、引用符で囲まれなければなりません。

- * ・ プロジェクトオプションの部分に[/fixed : no]を追加してください。
- * ・ [カテゴリ]から[アウトプット]を選択し、[エントリーポイントシンボル]に[rwMain]と入力して[OK]をクリックしてください。

以上で RTOS-WinJ アプリケーションを作成するためのプロジェクトの設定が完了しました。

アプリケーションを作成するためには：

- ・ [ファイル]メニューより [新規作成] で [新規作成] ダイアログを開きます。
- ・ [ファイル] タブを開いて、ファイル形式の一覧から [テキストファイル] を選択し、作成する C ソースコードの [ファイル名] を入力して [OK] ボタンをクリックしてください。

(ファイル名の最後には [.c] を必ず付けてください。)

- ・ ソースコードに [rwernel.h] をインクルードしてください。

下記のサンプルソースは、毎秒画面にメッセージを表示するプログラムです。

```
#include "rwKernel. H"

void main (void)
{
    while ( ! rwQuitRequest ())
    {
        rwSleep(1000);
        rwDebug (" your Mesrwge" );
    }
}
```

- ・ [ビルド] メニューより [ビルド] を選択し、ビルドを行ってください。
- ・ **動作確認事項**
 - ・ RTOS-WinJ Runtime Manager から RTOS-WinJ プログラムを起動します。
Windows の [スタートボタン] から [すべてのプログラム] - [Namatron RTOS-WinJ] - [RTOS-WinJ Runtime Manager] の順に選択してください。
 - ・ RTOS-WinJ RTM の画面にて [Browse] ボタンをクリックしてください。
ファイル選択の画面が表示されます。
 - ・ 作成したプロジェクトのディレクトリに進み、Release フォルダ内の exe ファイルを選択してください。
[開く] ボタンのクリックによりパスとファイル名が RTOS-WinJ Application に表示されますので RTM ウィンドウの [Start] ボタンをクリックしてください。
 - ・ サンプルプログラムを使用している場合は、RTM ウィンドウの [Output] タブを選択してください。
プログラムのメッセージが [Output] ウィンドウに表示されます。
 - ・ 以上で、非常に簡単な RTOS-WinJ プログラムの開発と実行を行いました。

6.3 システム関数の使用

RTOS-WinJ アプリケーションを開発しているユーザーは、Windows XP オペレーティングシステムにアクセスするシステムのランタイムライブラリ関数を呼び出さないように注意してください。（例：malloc, fopen 等）

オペレーティングシステムから独立している自己完結型のライブラリコールは、意図通りに実行されます。（例：strcpy, sprintf 等）

6.4 コンパイラの設定

Microsoft Visual C コンパイラを使用して RTOS-WinJ アプリケーションを作成します。

- ・ Win32 コンソールアプリケーションプロジェクトを作成してください。
- ・ [プロジェクト]メニューの[設定]により[プロジェクトの設定]ダイアログを開き、[C/C++]タブの[カテゴリ]から[プリプロセッサ]を選択してください。
- ・ RTOS-WinJ SDK ディレクトリを[インクルードファイルのパス]に追加してください。
(例 : C:\Program Files\RTOS-WinJ\Inc)
- ・ [Link]タブの[カテゴリ]から[一般]を選択してください。
- ・ [オブジェクト/ライブラリモジュール]を libc.lib, kernel32.lib と HyperLib.lib の3つにしてください。
(例 : libc.lib, kernel32.lib, "C:\Program Files\RTOS-WinJ\Lib\HyperLib.lib")
RTOS-WinJ シリアルポート関数を使用している場合は、rwSerialLib.lib もリンクしてください。パスに空白が含まれている場合は、引用符で囲ってください。
- ・ Microsoft Visual C++コンパイラ V6 の場合、[プロジェクト]メニューの[設定]により[プロジェクトの設定]ダイアログを開き、[Link]タブの[カテゴリ]から[一般]を選択し、/fixed : no をプロジェクトオプションに追加してください。
- ・ [Link]タブの[カテゴリ]から[アウトプット]を選択し、エントリーポイントシンボルの部分を rwMain に変更してください。
- ・ ソースコードの中に rwerenl.h をインクルードしてください。
- ・ Release version をコンパイルし、実行してください。Debug version ではないことにご注意ください。

第七章 HyperShare アプリケーションの開発

HyperShare ライブラリは、Windows アプリケーションが RTOS-WinJ のデータにアクセスし、RTOS-WinJ アプリケーションと相互に関連をもつことができる Windows の DLL です。このライブラリを使用することにより RTOS-WinJ アプリケーションの開始・停止が行えます。また、RTOS-WinJ データ領域の入出力、スレッド情報の取得、及び状態情報の取得も行えます。

HyperShare 関数を使用して C プログラムを作成する場合は、Windows ディレクトリにある HyperShare.dll を使用します。

(例 : C:\Winnt\HyperShare.dll)

インクルードファイルと DLL は下記の場所にあります。

C:\Program Files\RTOS-WinJ\Inc\HyperShare.h
C:\Program Files\RTOS-WinJ\Lib\HyperShare.lib

Microsoft Visual Basic 及び **ポーランド Delphi** のサンプルアプリケーションもご覧ください。

7.1 HyperShare API

HyperShare システム関数

<code>rwGetRWInstallDir</code>	RTOS-WinJ インストールディレクトリの取得
<code>rwGetLastError</code>	最後のエラー情報の取得
<code>rwStartApplication</code>	RTOS-WinJ アプリケーションの開始
<code>rwStopApplication</code>	RTOS-WinJ アプリケーションの停止

HyperShare データアクセス関数

<code>rwAttacrwharedRam</code>	共有メモリ領域のアタッチ
<code>rwDetacrwharedRam</code>	共有メモリ領域のデタッチ
<code>rwNumberSharedBits</code>	Bit 領域数の取得
<code>rwNumberSharedBytes</code>	Byte 領域数の取得
<code>rwNumberSharedLongs</code>	Long 領域数の取得
<code>rwNumberSharedWords</code>	Word 領域数の取得
<code>rwNumberUserSharedBytes</code>	ユーザーByte 領域数の取得
<code>rwNumberUserSharedLongs</code>	ユーザーLong 領域数の取得
<code>rwNumberUserSharedWords</code>	ユーザーWord 領域数の取得
<code>rwReadBit</code>	Bit 領域からの読み込み
<code>rwReadByte</code>	Byte 領域からの読み込み
<code>rwReadLong</code>	Long 領域からの読み込み
<code>rwReadUserByte</code>	ユーザーByte 領域からの読み込み
<code>rwReadUserLong</code>	ユーザーLong 領域からの読み込み
<code>rwReadUserword</code>	ユーザーWord 領域からの読み込み
<code>rwReadWord</code>	Word 領域からの読み込み
<code>rwSharedRam</code>	共有メモリ先頭ポインタの取得
<code>rwUse r SharedRam</code>	ユーザー領域先頭ポインタの取得
<code>rwWriteBit</code>	Bit 領域への書き込み
<code>rwWriteByte</code>	Byte 領域への書き込み
<code>rwWriteLong</code>	Long 領域への書き込み
<code>rwWriteUserByte</code>	ユーザーByte 領域への書き込み
<code>rwWriteUserLong</code>	ユーザーLong 領域への書き込み
<code>rwWriteUserword</code>	ユーザーWord 領域への書き込み
<code>rwWriteWord</code>	Word 領域への書き込み

HyperShare スレッド関連関数

<code>rwAwait</code>	Windows スレッド（プロセス）側でのシグナルの受信（ブロック有）
<code>rwCreateProcess</code>	Windows スレッド（プロセス）側からのプロセスの作成
<code>rwKillThread</code>	スレッドの実行終了
<code>rwLocateMemory</code>	名前付共有メモリの取得
<code>rwLocateThread</code>	名前付スレッドのスレッド ID 取得
<code>rwMaxThreads</code>	RTOS-WinJ 内最大可能スレッド数の取得
<code>rwNameAttach</code>	Windows スレッド（プロセス）名のアタッチ
<code>rwNameDetach</code>	Windows スレッド（プロセス）名のデタッチ
<code>rwParentTid</code>	スレッドの親スレッド ID 取得
<code>rwReadSignals</code>	未処理シグナル数の取得
<code>rwResume</code>	スレッドの実行再開
<code>rwSendTid</code>	送信ブロックしているスレッド ID の取得
<code>rwSetPriority</code>	スレッド優先度の設定
<code>rwSignal</code>	Windows スレッド（プロセス）側からのシグナル送信
<code>rwStackSize</code>	スタックサイズの取得
<code>rwStatusHeader</code>	スレッド情報のヘッダー文字列取得
<code>rwStatusString</code>	スレッド情報のステータス文字列取得
<code>rwSuspend</code>	スレッドの実行の一時停止
<code>rwThreadName</code>	スレッド名の取得
<code>rwThreadPriority</code>	スレッドの優先度取得
<code>rwThreadState</code>	スレッドの実行状態の取得

HyperShare ブーリアンステータス関数

<code>rwKernelError</code>	RTOS-WinJ 内部エラーの検出
<code>rwProcessExit</code>	RTOS-WinJ アプリケーションの終了検出
<code>rwProcessorFault</code>	プロセッサフォルトの検出
<code>rwSystemActive</code>	RTOS-WinJ システムの稼動検出

スレッド間通信 : セマフォ

<code>rwAllocSemaphoreEx</code>	セマフォの割当て
<code>rwCreateSemaphore</code>	名前付セマフォの作成
<code>rwDestroySemaphoreEx</code>	セマフォの破棄
<code>rwFreeSemaphoreEx</code>	セマフォの開放
<code>rwLocateSemaphore</code>	セマフォ作成スレッド ID の取得

スレッド間通信 : 新セマフォ関数

<code>rwCloseSemaphoreEx</code>	セマフォのクローズ
<code>rwGetSemaphoreEx</code>	セマフォの所有権獲得（ブロック有）
<code>rwOpenSemaphoreEx</code>	セマフォのオープン
<code>rwReleaseSemaphoreEx</code>	セマフォの所有権放棄
<code>rwTrySemaphoreEx</code>	セマフォの所有権獲得（ブロック無）

HyperShare デバッグトレースユーティリティ関数

<code>rwDumpDebugTrace</code>	出力ファイルへのトレースバッファ内容出力
<code>rwFormatTraceFile</code>	出力ファイルのアスキーフォーマット変換
<code>rwFormatTraceFileCSV</code>	出力ファイルの CSV フォーマット変換
<code>rwGetStartupTracing</code>	自動トレースモニタリングフラグの取得
<code>rwGetTraceBufferMode</code>	トレースバッファモードの取得
<code>rwGetTraceBufferSize</code>	トレースバッファサイズの取得
<code>rwGetTraceDumpFile</code>	出力ファイル名の取得
<code>rwGetTraceState</code>	トレースモニタリング状態の取得
<code>rwGetTraceTriggers</code>	スタート・ストップトリガの取得
<code>rwSetStartupTracing</code>	自動トレースモニタリングフラグの設定
<code>rwSetTraceBufferMode</code>	トレースバッファモードの設定
<code>rwSetTraceBufferSize</code>	トレースバッファサイズの設定
<code>rwSetTraceDumpFile</code>	出力ファイルの設定
<code>rwSetTraceTriggers</code>	スタート・ストップトリガの設定
<code>rwStartTraceMonitoring</code>	トリガ・イベントのモニタリング開始
<code>rwStopTracing</code>	トリガ・イベントのモニタリング終了
<code>rwTraceDataAvailable</code>	トレースバッファ内の記録数の取得

その他の HyperShare 関数

<code>rwAttachDebugWindow</code>	RTOS-WinJ RTM Output ウィンドウのアタッチ
<code>rwClearDebug</code>	RTOS-WinJ RTM Output ウィンドウのクリア
<code>rwDetachDebugWindow</code>	RTOS-WinJ RTM Output ウィンドウのデタッチ
<code>rwGetSystemTime</code>	システム時刻の取得

7.2 プログラミングサンプル

7.2.1 Microsoft Visual Basic

```
Declare Function StartHyperLink Lib "C:\Winnt\HyperShare.dll"
```

```
    (ByVal rwExe As String, ByVal Flagrws Long) As Integer
```

```
Declare Function StopHyperLink Lib "C:\Winnt\HyperShare.dll" () As Integer
```

付録 A : RTOS-WinJ API リファレンス

A.1 メモリ

A.1.1 `rwAlloc`

構 文

```
void* rwAlloc(long NumBytes)
```

引 数

型	引数名	説 明
long	NumBytes	割当てるバイト数

戻り値

割当てたメモリへのポインタ : 成功
NULL : 失敗

解 説

`rwAlloc` は、システムメモリプールからサイズ `NumBytes` で指定したメモリ領域を割当てます。

参 照

`rwFree`

A.1.2 rwFree

構 文

```
void rwFree(void* Block)
```

アーギュメント

型	引数名	説 明
void*	Block	RwAlloc にて割当てた領域のポインタ

戻り値

なし

解 説

rwFree は、ポインタで示すメモリ領域をシステムメモリプールに戻します。
メモリ領域は、rwAlloc 関数で割当てた領域でなければなりません。

参 照

rwAlloc

A.1.3 rwMapMemory

構 文

```
void* rwMapMemory(unsigned long Phys, unsigned long Size,
                  unsigned long AllowCache)
```

引 数

型	引数名	説 明
unsigned long	Phys	スレッドにマップする物理メモリアドレス
unsigned long	Size	メモリのサイズ (バイト単位)
unsigned long	AllowCache	TRUE : CPU メモリキャッシュ FALSE: インターフェイスカード (通常)

戻り値

マップされたメモリのポインタ : 成功
NULL : 失敗

解 説

rwMapMemory は、物理アドレスを呼び出しスレッドの仮想アドレス空間にマップし、そのポインタを返します。
呼び出しスレッドは、返されたポインタに read/write することによりこのメモリにアクセスすることができます。

参 照

rwUnmapMemory

A. 1. 4 rwUnmapMemory

構 文

```
long rwUnmapMemory(void* Mem, unsigned long Size)
```

引 数

型	引数名	説 明
void*	Mem	RwMapMemory で返されたポインタ
unsigned long	Size	メモリのサイズ (バイト単位)

戻り値

0 : 成功
-1 : 失敗

解 説

rwUnmapMemory は、rwMapMemory を使用してマップされた物理メモリをアンマップします。

参 照

rwMapMemory

A.2 共有メモリ

A.2.1 rwLocateMemory

構 文

```
void* rwLocateMemory(char* Name, unsigned long* Length)
```

引 数

型	引数名	説 明
char*	Name	共有メモリ領域の名前へのポインタ
unsigned long*	Length	メモリの長さが返される領域のポインタ

戻り値

共有メモリのポインタ : 成功
 NULL : 失敗

解 説

rwLocateMemory は、Name で指定された名前付共有メモリの位置を検索し、共有メモリ領域の長さとその領域へのポインタを返します。
 名前付共有メモリは、rwShareMemory 関数にて作成します。
 名前付共有メモリは、RTOS-WinJ プロセスや Windows プロセスによる情報の共有を可能にします。

参 照

rwShareMemory, rwUnshareMemory

A. 2. 2 rwNumberSharedBits

構 文

long rwNumberSharedBits(void)

引 数

なし

戻り値

RTOS-WinJ 共有メモリ領域に割当てられている Bit 数

解 説

rwNumberSharedBits は、RTOS-WinJ 共有メモリ領域に割当てられている使用可能な Bit 数を返します。

参 照

rwNumberShared, rwRead, rwWrite, rwUserSharedRam

A. 2. 3 `rwNumberSharedBytes`

構 文

`long rwNumberSharedBytes(void)`

引 数

なし

戻り値

RTOS-WinJ 共有メモリ領域に割り当てられている Byte 数

解 説

`rwNumberSharedBytes` は、RTOS-WinJ 共有メモリ領域に割り当てられている使用可能な Byte 数を返します。

参 照

`rwNumberShared`, `rwRead`, `rwWrite`, `rwUserSharedRam`

A. 2. 4 rwNumberSharedLongs

構 文

long rwNumberSharedLongs(void)

引 数

なし

戻り値

RTOS-WinJ 共有メモリ領域に割当てられている Long 数

解 説

rwNumberSharedLongs は、RTOS-WinJ 共有メモリ領域に割当てられている使用可能な Long 数を返します。

参 照

rwNumberShared, rwRead, rwWrite, rwUserSharedRam

A. 2. 5 `rwNumberSharedWords`

構 文

`long rwNumberSharedWords(void)`

引 数

なし

戻り値

RTOS-WinJ 共有メモリ領域に割り当てられている Word 数

解 説

`rwNumberSharedWords` は、RTOS-WinJ 共有メモリ領域に割り当てられている使用可能な Word 数を返します。

参 照

`rwNumberShared`, `rwRead`, `rwWrite`, `rwUserSharedRam`

A. 2. 6 rwSharedRam

構 文

`void* rwSharedRam(void)`

引 数

なし

戻り値

共有メモリへのポインタ

解 説

rwSharedRam は、Windows と共有するメモリの先頭ポインタを返します。

参 照

rwUserSharedRam, rwNumberShared, rwRead, rwWrite

A. 2. 7 rwSharedMemory

構 文

```
long rwSharedMemory(char* Name, void* p, long Length)
```

引 数

型	引数名	説 明
char*	Name	共有メモリ領域の名前へのポインタ
void*	p	名前付共有メモリ領域のポインタ
long	Length	共有メモリ領域の長さ

戻り値

1 : 成功
0 : 失敗

解 説

rwSharedMemory は、使用可能な共有メモリリソースとして名前付共有メモリ領域を作成します。

このメモリは、プログラム内に定義（例えば、配列として）をするか、または rwAlloc により動的割当てることができます。

作成されたこのメモリは、rwLocateMemory 関数により他の RTOS-WinJ のプロセスや Windows プロセスから取得することができ、取得したプロセスからアクセスすることが可能になります。

共有メモリへのアクセスには、セマフォやクリティカルセクションを使用して共有メモリ内のデータを保護しなければならない場合があります。

共有メモリリソースがなくなった場合には、必ず rwUnsharedMemory を呼び出してください。

参 照

rwUnshareMemory, rwLocateMemory

A. 2. 8 rwUserSharedRam

構 文

```
void* rwUserSharedRam(long* Size)
```

引 数

型	引数名	説 明
long*	Size	サイズが返される領域のポインタ

戻り値

共有メモリーユーザー領域の先頭ポインタ

解 説

rwUserSharedRam は、RTOS-WinJ 共有 RAM 領域のポインタと割当てられたメモリのサイズを返します。

このメモリは、RTOS-WinJ アプリケーション実行中にいつでもユーザーが使用可能な汎用バッファです。

参 照

rwNumberShared, rwRead, rwWrite

A. 2. 9 rwReadBit

構 文

long rwReadBit(long BitVar)

引 数

型	引数名	説 明
long	BitVar	読込む Bit 番号

戻り値

指定 Bit 番号の値 : 成功
-1 : エラー

解 説

rwReadBit は、RTOS-WinJ 共有メモリの指定 Bit 番号の値を返します。

参照

rwNumberShared, rwRead, rwWrite, rwUserSharedRam

A. 2. 10 rwReadByte

構 文

long rwReadByte(long ByteVar)

引 数

型	引数名	説 明
long	ByteVar	読込む Byte 番号

戻り値

指定 Byte 番号の値 : 成功
0 : エラー

解 説

rwReadByte は、RTOS-WinJ 共有メモリの指定 Byte 番号の値を返します。

参 照

rwNumberShared, rwRead, rwWrite, rwUserSharedRam

A. 2. 11 rwReadLong

構 文

long rwReadLong(long LongVar)

引 数

型	引数名	説 明
Long	LongVar	読込む Long 番号

戻り値

指定 Long 番号の値 : 成功
0 : エラー

解 説

rwReadLong は、RTOS-WinJ 共有メモリの指定 Long 番号の値を返します。

参 照

rwNumberShared, rwRead, rwWrite, rwUserSharedRam

A. 2. 12 rwReadWord

構 文

```
long rwReadWord(long WordVar)
```

引 数

型	引数名	説 明
Long	WordVar	読み込む Word 番号

戻り値

指定 Word 番号の値 : 成功
 0 : エラー

解 説

rwReadWord は、RTOS-WinJ 共有メモリの指定 Word 番号の値を返します。

参 照

rwNumberShared, rwRead, rwWrite, rwUserSharedRam

A. 2. 13 rwUnshareMemory

構 文

long rwUnshareMemory(char* Name)

引 数

型	引数名	説 明
char*	Name	共有メモリの名前へのポインタ

戻り値

1 : 成功
0 : 失敗

解 説

rwUnshareMemory は、rwShareMemory 関数にて作成された使用可能な共有メモリリソースから指定した名前付共有メモリを削除します。
このメモリは、開放（デアロケート）できませんのでご注意ください。
rwUnshareMemory が呼び出された後でも、既に共有メモリへアクセスしているスレッドおよびプロセスにおけるそのアクセスは、保持されます。
rwUnshareMemory は、他のプロセスから検索できないように共有メモリリソースから単純にメモリを削除します。
アプリケーションがこのメモリを共用する必要性がなくなった場合、rwShareMemory 呼び出しに対応する rwUnshareMemory 呼び出しを行わなくてはならないことを忘れないでください。

参 照

rwShareMemory, rwLocateMemory

A.2.14 rwWriteBit

構 文

```
void rwWriteBit(long BitVar, long Value)
```

引 数

型	引数名	説 明
long	BitVar	書込む Bit 番号
long	Value	Bit 番号に書込む値

戻り値

なし

解 説

rwWriteBit は、指定 Bit 番号に値を書込みます。

参 照

rwNumberShared, rwRead, rwWrite, rwUserSharedRam

A. 2. 15 rwWriteByte

構 文

```
void rwWriteByte(long ByteVar, long Value)
```

引 数

型	引数名	説 明
long	ByteVar	書込む Byte 番号
long	Value	Byte 番号に書込む値

戻り値

なし

解 説

rwWriteByte は、指定 Byte 番号に値を書込みます。

参 照

rwNumberShared, rwRead, rwWrite, rwUserSharedRam

A. 2. 16 **rwWriteLong**

構 文

void rwWriteLong(long LongVar, long Value)

引 数

型	引数名	説 明
long	LongVar	書込む Long 番号
long	Value	Long 番号に書込む値

戻り値

なし

解 説

rwWriteLong は、指定 Long 番号に値を書込みます。

参 照

rwNumberShared, rwRead, rwWrite, rwUserSharedRam

A. 2.17 rwWriteWord

構 文

```
void rwWriteWord(long WordVar, long Value)
```

引 数

型	引数名	説 明
long	WordVar	書込む Word 番号
long	Value	Word 番号に書込む値

戻り値

なし

解 説

rwWriteWord は、指定 Word 番号に値を書込みます。

参 照

rwNumberShared, rwRead, rwWrite, rwUserSharedRam

A.3 リテンティブメモリ

A.3.1 rwRetentInit

構 文

```
int rwRetentInit(void)
```

引 数

なし

戻り値

RW_RETENT_OK	:	成功
RW_RETENT_NOT_INSTALLED	:	カードが検出できない
RW_RETENT_ERROR	:	カードの初期化に失敗

解 説

rwRetentInit は、リテンティブメモリの初期化を行います。

参 照

rwRetentExit, rwRetentRead, rwRetentWrite, kRetentQueryBatteryStatus

A.3.2 rwRetentRead

構 文

```
int rwRetentRead(unsigned long Offset, void* Data, int Length)
```

引 数

型	引数名	説 明
unsigned long	Offset	読込むリテンティブメモリのアドレス
void*	Data	データ格納領域のポインタ
int	Length	読込むデータ長 (バイト単位)

戻り値

RW_RETENT_OK : 成功
 RW_RETENT_ERROR : 読みみエラー

解 説

rwRetentRead は、リテンティブメモリからデータを読込みます。
 この関数は、データ格納領域にリテンティブメモリの Offset で指定された
 アドレスから Length で指定されたバイト数のデータを読込みます。

参 照

rwRetentExit, rwRetentInit, rwRetentWrite, kRetentQueryBatteryStatus

A.3.3 rwRetentWrite

構 文

```
int rwRetentWrite(unsigned long Offset, void* Data, int Length)
```

引 数

型	引数名	説 明
unsigned long	Offset	書込むリテンティブメモリのアドレス
void*	Data	データ格納領域のポインタ
Int	Length	書込むデータ長（バイト単位）

戻り値

RW_RETENT_OK : 成功
 RW_RETENT_ERROR : 書込みエラー

解 説

rwRetentWrite は、リテンティブメモリにデータを書込みます。
 この関数は、データ格納領域のデータをリテンティブメモリの Offset で指定されたアドレスに Length で指定されたバイト数で書き込みます。

参 照

rwRetentExit, rwRetentInit, rwRetentRead, rwRetentQueryBatteryStatus

A.3.4 rwRetentExit

構 文

```
int rwRetentExit(void)
```

引 数

なし

戻り値

RW_RETENT_OK

解 説

rwRetentExit は、リテンティブメモリへのアクセスを終了するときに使用します。

この関数はリテンティブメモリをアンマップします。

参 照

rwRetentInit, rwRetentRead, rwRetentWrite,
rwRetentQueryBatteryStatus

A. 3.5 rwRetentQueryBatteryStatus

構 文

```
int rwRetentQueryBatteryStatus(int iWhichBattery)
```

引 数

型	引数名	説 明
int	iWhichBattery	問合せるバッテリーの種類 RW_RETENT_INTERNAL_BATTERY RW_RETENT_EXTERNAL_BATTERY

戻り値

RW_RETENT_BATTERY_OK : 成功

RW_RETENT_ERROR : リアルタイムサブシステムカードがインストールされていない場合、あるいはバッテリー状態のデータが無効な場合

RW_RETENT_BATTERY_LOW : バッテリー量がわずかな場合

解 説

rwRetentQueryBatteryStatus は、リアルタイムサブシステムカード (BBSRAM) の内部あるいは外部バッテリーの状態を測定するために使用します。

注意： 関数 rwRetentInit () を呼び出してから rwRetentQueryStatus () を呼び出さなければなりません。

A.4 ファイル I/O

A.4.1 rwCopyFile

構 文

```
long rwCopyFile(char* SrcName, char* DestName, long FailIfExists)
```

引 数

型	引数名	説 明
char*	SrcName	コピー元のファイル名へのポインタ
char*	DestName	コピー先のファイル名へのポインタ
long	FailIfExists	デスティネーション存在フラグ 0 : DestName が存在する場合ファイルを 上書きする 1 : DestName が存在する場合に関数呼 び出し失敗

戻り値

1 : 成功
0 : 失敗

解 説

rwCopyFile は、DestName によって指定されたファイル名のファイルに SrcName で指定したファイルをコピーします。
コピーするファイル名のファイルがすでに存在するか、FailIfExists パラメータが 0 でない場合、この関数は失敗します。
FailIfExists を 0 とする場合、DestName で指定したファイルがすでに存在しているなら上書きされます。

参 照

rwRenameFile, rwRemoveFile も参照してください。

A. 4. 2 rwOpen

構 文

```
long rwOpen(char* Filename, long Mode)
```

引 数

型	引数名	説 明
char*	Filename	開くファイルのネーム
long	Mode	次の 1 つ : READ_FILE WRITE_FILE READWRITE_FILE

戻り値

失敗 : ファイルを開かなかった場合、0 を返します
 成功 : 開いたファイルへのハンドル。

解 説

rwOpen は、Windows XP ディスクファイルを開きます。
 ファイルへのパスはドライブレターを含むフルパスを使用します。
 (例 : C:¥¥directiry¥¥file.ext)

参 照

rwClose, rwRead, rwWrite, rwTell, rwSeek, rwSizeOfFile も参照してください。

A. 4. 3 **rwClose**

構 文

long rwClose(long FileHandle)

引 数

型	引数名	説 明
long	FileHandle	RwOpen によって戻されたファイルハンドル

戻り値

0 : 成功
-1 : 失敗

解 説

rwClose は、rwOpen によって開かれたファイルを閉じます。

参 照

rwClose, rwRead, rwWrite, rwTell, rwSeek, rwSizeOfFile も参照してください

A. 4. 4 **rwRead**

構 文

long rwRead(void* Buffer, long Size, long Count, long FileHandle)

引 数

型	引数名	説 明
void*	Buffer	ディスクファイルからデータリード用バッファを示すポインタ
long	Size	記録のサイズ（バイト単位）
long	Count	読取るレコードの数
long	FileHandle	読取るファイルのハンドル

戻り値

読み取られる全レコード数。
0 の戻り値は EOF を示します。

解 説

rwRead は、Size で指定したサイズのカウンタデータ記録を読み取り、この記録をバッファに設定します。

参 照

rwOpen, rwWrite, rwTell, rwSeek, rwSizeOfFile, rwReadString も参照してください。

A. 4.5 `rwRemoveFile`

構 文

`long rwRemoveFile(char* FileName)`

引 数

型	引数名	説 明
char*	FileName	消去するファイルのネーム

戻り値

0 : 失敗
1 : 成功

解 説

`rwRemoveFile` は、`FileName` で指定したファイルを消去します。

参 照

`rwCopyFile`, `rwRenameFile` も参照してください。

A. 4. 6 `rwRenameFile`

構 文

```
long rwRenameFile(char* OldName, char* NewName)
```

引 数

型	引数名	説 明
char*	OldName	リネームするファイルネーム
char*	NewName	ファイルの新しいネーム

戻り値

0 : 失敗
1 : 成功

解 説

`rwRenameFile` は、OldName で指定したファイルを NewName で指定したファイル名に変更します。
新しいファイルネームがすでに存在する場合、この関数は失敗します。

参 照

`rwCopyFile`, `rwRemoveFile` も参照してください。

A.4.7 rwWrite

構 文

```
long rwWrite(void* Buffer, long Size, long Count, long FileHandle)
```

引 数

型	引数名	説 明
void*	Buffer	書込むデータのバッファを示すポインタ
long	Size	レコードサイズ（バイト単位）
long	Count	書込むレコード数
long	FileHandle	書込みファイルのハンドル

戻り値

書込まれたレコード総数。

解 説

rwWrite は、バッファから Size で指定したサイズのカウンタデータ記録をディスクファイルに書き込みます。

参 照

rwOpen, rwRead, rwWrite, rwTell, rwSeek, rwSizeOfFile も参照してください。

A. 4. 8 **rwTell**

構 文

long rwTell(long FileHandle)

引 数

型	引数名	説 明
long	FileHandle	ファイルのハンドル

戻り値

ファイルの始めからの現在のオフセット（バイト単位）。

解 説

rwTell は、ファイルの始めから現在のオフセットを戻します。

参 照

rwOpen, rwRead, rwWrite, rwTell, rwSeek, rwSizeOfFile も参照してください。

A. 4.9 rwSizeOfFile

構 文

```
long rwSizeOfFile(long FileHandle)
```

引 数

型	引数名	説 明
long	FileHandle	読むファイルのハンドル

戻り値

ファイル内のバイト総数 : 成功
-1 : エラー

解 説

rwSizeOfFile は、ファイル中のバイト総数を戻します。

参 照

rwOpen, rwRead, rwWrite, rwTell, rwSeek, rwSizeOfFile も参照してください。

A. 4. 10 **rwSeek**

構 文

long rwSeek(long FileHandle, long Offset, long Origin)

引 数

型	引数名	説 明
long	FileHandle	ファイルのハンドル
long	Offset	元の値に関係した位置
long	Origin	次のうち 1 つ : SEEK_BOF (ファイルの始め) SEEK_EOF (ファイルの終わり) SEEK_CURRENT (現在地から)

戻り値

0 : 成功
-1 : 失敗

解 説

rwSeek は、ファイル内の希望するオフセットへ、現在のファイル位置を移動します。

注意 :
SEEK_EOF を使用する場合、あるいは SEEK_CURRENT の位置から、ファイルの始めに向けてシークする場合は、ネガティブオフセットを使います

参 照

rwOpen, rwRead, rwWrite, rwTell, rwSeek, rwSizeOfFile も参照してください。

A. 4. 11 `rwReadString`

構 文

```
void* rwReadString(void* Buffer, long Size, long FileHandle)
```

引 数

型	引数名	説 明
void*	Buffer	ディスクファイルから読取られるデータ用バッファポインタ
long	Size	バッファサイズ (バイト単位)
long	FileHandle	読取りファイルのハンドル

戻り値

バッファを示すポインタ : 成功
 NULL ポインタ : 失敗

解 説

`rwReadString` は、ファイルからデータの文字列を読み取ります。
 この関数は CR/LF に遭遇するか、バイト数が最大まで書き込まれるまで、
 ファイルからデータを読み込みます。

参 照

`rwOpen`, `rwRead`, `rwWrite`, `rwTell`, `rwSeek`, `rwSizeOfFile` も参照してください。

A. 4.12 rwWriteString

構 文

```
long rwWriteString(void* Buffer, long FileHandle)
```

引 数

型	引数名	説 明
void*	Buffer	ディスクファイルに書込まれるデータのバッファポインタ
long	FileHandle	書込みファイルのハンドル

戻り値

ファイルに書き込まれたバイト総数

解 説

rwWriteString は、NULL 文字が終端の文字列のデータをファイルに書き込みます。

参 照

rwOpen, rwWrite, rwTell, rwSeek, rwSizeOfFile, rwRead, rwReadString も参照してください。

A.5 タイミング

A.5.1 rwDelay

構 文

```
void rwDelay(long Delay)
```

引 数

型	引数名	説 明
long	Delay	遅らせるシステムチック数

戻り値

なし

解 説

rwDelay は、指定したシステムチックの値だけ現在のスレッドの実行を一時中断します。

注意： このシステムチックのサイズは、現在は1ミリセカンドです。

参 照

rwSleep, rwYield も参照してください。

A.5.2 rwMicroClock

構 文

```
_int64 rwMicroClock(void)
```

引 数

なし

戻り値

システムが起動した時間からの μ sec 数

解 説

rwMicroClock は、システムが起動した時間（システムブートタイム）からのミリセカンド数を返します。

Pentium プロセッサにおいて RwMicroClock のみが正常に、あるいはより正確に機能します。486 機種では、この関数は 0 を返します。RwMicroClock は、1 μ セカンド以内の正確さをもつプロセッサ・ハードウェア・カウンタを使用しています。

参 照

rwMicroSleep も参照してください。

A.5.3 rwMicroSleep

構 文

```
void rwMicroSleep(unsigned long microseconds, long CritFlag)
```

引 数

型	引数名	説 明
unsigned long	Microseconds	スリープするマイクロセカンド数
long	CritFlag	スリープする際に重要なセクションを使用する

戻り値

なし

解 説

rwMicroSleep は、スレッド／プロセスが 1 μ sec 以内の間スリープできるようにするものです。スリープタイムは、マイクロセカンド単位で測定されます。この関数のみが Pentium、あるいは改良型のプロセッサで正常に実行されます。486 機種では、すぐにこの関数は戻ります。

この関数は十分な注意を払って取り扱う必要があります。

この関数は、 μ sec 間隔でスレッドの呼び出しを遅らせるために、特別なプロセッサハードウェアカウンタを使用します。

しかしこのスレッドはほかのスレッドに対するタイムスライスをやめることはしません。この呼び出しスレッドは、待機状態のままになります。

つまり、このスリープは、ビジー状態による遅れということです。

特に I/O ハードウェアの要件を満たすために、ごく短い時間（通常は 10 から 20 マイクロセカンド以下）遅らせる必要がある場合、ユーザーは、この関数を使用します。CritFlag は、インタラプトを防ぐため、あるいは遅らせている間コンテキストの切り替えを防ぐために使用されます。

CritFlag へ伝えられるゼロではない数値によって、重要なセクション内で遅れさせます。このアクションによって、遅らせる時間の正確性を高めることができますが、このシステムの（スリープ時間の長さによる）インタラプトの潜在性を増す影響も概して考えられます。

A. 5.5 rwHardwareCounter

構 文

```
void rwHardwareCounter(_int64* Ticks)
```

引 数

型	引数名	説 明
_int64*	Ticks	カウンタ値を受信する 64 ビット整数を示すポインタ

戻り値

なし

解 説

rwHardwareCounter は、プロセッサの高性能カウンタが存在すれば、その現在位置を獲得します。これは、プロセッサへのクロックパルス毎に増加する 64 ビットローリングカウンタです。

参 照

rwGetTicks も参照してください。

A.5.6 rwGetTicks

構 文

```
unsigned long rwGetTicks(void)
```

引 数

なし

戻り値

システム クロックチックの現在の数値

解 説

rwGetTicks は、システムクロックチックの現在の数値である符号なしの数値を返します。

クロックチックは、1 秒に 1000 回生じます。

この数は現在日時に直接関連付けられるものではありません。

間隔のタイミングに、秒単位以上の正確性を与えるように装備されているものです。

rwGetTicks に対する 2 つの呼び出しが行われた場合、そして戻り値が互いから引かれる場合は、その違いの絶対値がコール間の 1msc の数となります。

この戻り値はフリーランニングの 32 ビットカウンタですので、カウンタがゼロを送り込んでいるときは、ユーザーは注意が必要です。

参 照

rwHardwareCounter も参照してください。

A.5.7 rwTime

構 文

```
void rwTime(unsigned long* Seconds, unsigned long* milliseconds)
```

引 数

型	引数名	説 明
unsigned long*	Seconds	秒のデスティネーションを示すポインタ
unsigned long*	milliseconds	ミリ秒を示すポインタ

戻り値

なし

解 説

1970 年 1 月深夜以来 rwTime は、セカンド（秒）の数を返します。
 この数値はすでに現地時間に変更されています。
 こうしてこの数値は 1 日のうちの時刻を得る（とらえる、受ける）ために、
 gmtime 及び asctime に伝えられます。
 現地時間やタイムゾーン情報を要する関数に、この数値を伝える必要はありません。
 ミリ秒のパラメータは、秒のパラメータが戻す秒を過ぎて、ミリ秒（0 から 999）の数を返します。
 このタイムスタンプは、NT システム使用できる gmtime 及び asctime 関数
 と互換性があり、データのログ収集や、ネイティブ NT アプリケーションが
 読み取ることのできるタイムスタンプ情報のために使用される場合があります。

A. 6 スレッドコントロール

A. 6.1 rwCreateProcess

構 文

```
long rwCreateProcess(char* Filename, long StackSize, char* name,
                    long Priority)
```

引 数

型	引数名	説 明
char*	Filename	完全パス (full path) プロセスを作成する際に使用される exe のファイルネームを示すポインタ
long	StackSize	プロセス用のバイト単位でのスタックのサイズ (最小 2K)
char*	char*	新たなプロセス用のスレッドネームを示すポインタ
long	Priority	作成されたプロセスのスレッドの優先度 (0 から 31)

戻り値

```
新たなプロセスのスレッド ID      : 成功
-1                                : 失敗
```

解 説

rwCreateProcess は、RTOS-WinJ システムに新たなプロセスを作成します。プロセスにはカプセル化したスレッドがありますので、システム内で標準スレッドのように正確に機能します。

違うことが2つだけあり、まず、このプロセスが別々にコンパイル及びリンクされている実行可能ファイル (この実行可能ファイル内のメインの () 関数で始まる) からロードされているということ、そしてグローバルデータが、スレッド間のようにプロセス間で直接アクセスが不可能であるということです。

ポインタはプロセスからプロセスへ (共有メモリや rwSend から) 伝えることが可能であり、そのポインタは有効のままです。

通常の RTOS-WinJ アプリケーションプログラムのように、プロセスは正確に書き込まれます。

例えば、そのプロセスは void main(void) 関数で開始し、rwQuitRequest()

が、このシステムがシャットダウン中であることを判断することを確認する必要があります、ということです。

A. 6.2 rwCreateProcess

構 文

```
long rwCreateProcessEx(char* Filename, long StackSize, char* Name,
                      long Priority, void* Param)
```

引 数

型	引数名	説 明
char*	Filename	実行可能ファイルの完全パスとファイル ルネームポインタ
long	StackSize	バイト単位のプロセス用スタックのサ イズ（最小 2K）
char*	char*	新たなプロセス用のスレッドネームポ インタ
long	Priority	作成されたプロセスのスレッドの優先 度（0 から 31）
void*	Param	プロセスに伝えられたユーザーパラメ ータポインタ

戻り値

新たなプロセスのスレッド ID : 成功
-1 : 失敗

解 説

rwCreateProcessEx は、RTOS-WinJ システムの新たなプロセスを作成します。

プロセスにはカプセル化したスレッドがありますので、システム内で標準スレッドのように正確に機能します。

プロセスにはカプセル化しているスレッドがありますので、システムにおいて標準スレッドのように正確に機能します。

違うことが2つだけあり、まずこのプロセスが別々にコンパイル及びリンクされている実行可能ファイル（この実行可能ファイル内のメインの（）関数で始まる）からロードされているということ、そしてグローバルデータが、スレッド間のようにプロセス間で直接アクセスが不可能であるという事です。ポインタはプロセスからプロセスへ（共有メモリや rwSend から）伝えることが可能であり、そのポインタは有効のままです。

通常の RTOS-WinJ アプリケーションプログラムのように、プロセスは正確に書き込まれます。

例えば、そのプロセスは void main(void)関数で開始し、rwQuitRequest()

が、このシステムがシャットダウン中であることを判断することを確認する必要があります、ということです。

rwCreateProcessEx は、ユーザーがパラメータを作成したプロセスに伝えることができるという点において、rwCreateProcess と異なります。

このパラメータは、作成したプロセス内（あるいは RTOS-WinJ のすべてのプロセスやスレッド内）において、rwGetThreadData 関数から検索されることがあります。

参 照

rwGetThreadData, rwCreateThreadEx も参照してください。

A. 6.3 rwCreateThread

構 文

```
long rwCreateThread(void* Address, long StackSize, char* ThreadName,
                    long Priority)
```

引 数

型	引数名	説 明
void*	Address	関数を示すポインタ
long	StackSize	スレッドのスタック用に割り当てるメモリ（最小 2K）
char*	ThreadName	このスレッドを認識する際に使用されるスレッド名（最大 16 文字）
long	Priority	スレッドを実行する際の優先度（0 から 31）

戻り値

```
新たなプロセスのスレッド ID      : 成功
-1                                : 失敗
```

解 説

rwCreateThread は、RTOS-WinJ にて実行する新たなスレッドを作成します。スレッドの作成失敗にて考えられる理由が考えられます：

- ・ ThreadName がすでに存在している。
- ・ 最大スレッドが実行中である。
- ・ メモリが不足している。

このスレッドをカプセル化する関数は、次のように定義される必要があります。例えば、パラメータや戻り値がないという場合、

```
void ThreadFunc(void)
{
}
```

パラメータは rwCreateThreadEx API への呼び出し（コール）から、この関数に伝えられる場合があります。しかし、このパラメータは直接この関数に伝えられませんが、rwGetThreadData API への呼び出し（コール）からこの関数が検索します。（rwCreateThreadEx を参照してください）。スレッド関数が戻る場合は、rwKillThread がスレッド ID によって呼び出されるかのように、このスレ

ット自体が破壊されます。

参 照

rwExitThread, rwKillThread, rwLocateThread, rwCreateThreadEx も
参照してください。

A. 6. 4 rwCreateThreadEx

構文

```
long rwCreateThreadEx(void* Address, long StackSize, char* ThreadName,
                     long Priority, void* Param)
```

引 数

型	引数名	説 明
void*	Address	関数を示すポインタ
long	StackSize	バイト単位のスレッド用スタックのサイズ（最小 2K）
char*	ThreadName	新たなスレッド用のスレッドネームを示すポインタ
long	Priority	作成されたスレッドの優先度（0 から 3 1）
void*	Param	スレッドに伝えられたユーザーパラメータを示すポインタ

戻り値

```
スレッド ID      : 成功
-1               : 失敗
```

解説

rwCreateThreadEx は、RTOS-WinJ で実行するスレッドを作成します。

この関数は数値を戻すか、変数を持ってはいけません。

例えば :

```
#include "rwnel.h"

typedef struct
{
    long Data1, Data2, Data3;
} MyStructType;

void ThreadFunc(void)
{
    MyStructType* p;
    p = rwGetThreadData(rwGetTid());
    //Thread code goes here ...
}

void main(void)
```

```

{
    long tid;
    MyStructType p;
    //Fill in p

    p.Data1 = 1;
    p.Data2 = 2;
    p.Data3 = 3;
    //Create the Thread
    tid = rwCreateThreadEx(ThreadFunc, MIN_STACK_SIZE,
                           "AthreadName", 8, p);
    while(!rwQuitRequest())
    {
        rwSleep(100);
    } //while
} //main

```

rwCreateThreadEx は rwCreateThread 関数と異なる点は、ユーザーが作成したスレッドにパラメータを渡すことを可能にしている点です。
このパラメータは rwGetThreadData 関数から検索されます。

参 照

rwGetThreadData も参照してください。

A. 6.5 rwEnterCriticalSection

構 文

```
void rwEnterCriticalSection(long* ps)
```

引 数

型	引数名	説 明
long*	ps	システム情報を保存する際に使用される一時的なローカル変数

戻り値

なし

解 説

rwEnterCriticalSection は、RTOS-WinJ におけるスレッドの切り替えを防ぎます。

この関数は、Windows がタスクを実行したり切り替えしないようにするもので、またブロックのインタラプトが発生しないようにもします。

このような理由がありますので、使用する際は注意が必要です。

rwEnterCriticalSection は、このマシン（機種）でもっとも重要なコードのセクションのために、短時間使用されるのみです。 数 100 ミリセカンドの間、重要なこのセクションにシステムが存続する場合、クロックが遅れる可能性があり、また、タイムアウト監視のきっかけとなる NT デバイスドライバもあります。

rwEnterCriticalSection と共に rwSeizeProcessor を使用する必要はありません。

参 照

rwLeaveCriticalSection, rwSeizeProcessor, rwReleaseProcessor も参照してください。

A. 6. 6 rwExitThread

構 文

void rwExitThread (void)

引 数

なし

戻り値

なし

解 説

rwExitThread は、現在のスレッドを終了させます。

参 照

rwKillThread も参照してください。

A. 6. 7 rwForceReady

構 文

```
int rwForceReady(long tid)
```

引 数

型	引数名	説 明
long	tid	スレッド ID

戻り値

```
0      : 失敗
1      : 成功
```

解 説

rwForceReady は、スレッドをいつでも使える状態にさせます。
 スレッドが Send, Receive, Wait, Suspend block の場合は、次のスケジュールのタイムスライスにおいて、スレッドは強制的に実行再開されます。
 rwSend, rwReceive では失敗 (0) を返します。

A. 6. 8 rwGetMainTid

構 文

long rwGetMainTid(void)

引 数

なし

戻り値

メインスレッドのスレッド ID

解 説

rwGetMainTid は、メインスレッドのスレッド ID を戻します。

メインスレッドは、RTOS-WinJ アプリケーションのメイン()関数を実行する初期スレッドです。

参 照

rwGetTid, rwLocateThread も参照してください。

A. 6. 9 rwGetPriority

構 文

```
int rwGetPriority(long tid)
```

引 数

型	引数名	説 明
long	tid	優先度を獲得するためのスレッド ID

戻り値

スレッドの優先度 : 成功
0 : 失敗

解 説

rwGetPriority は、tid で指定したスレッドの優先度を返します。

参 照

rwSetPriority も参照してください。

A. 6.10 rwGetThreadData

構 文

```
void* rwGetThreadData(long tid)
```

引 数

型	引数名	説 明
long	tid	パラメータが検索されるためのスレッドのスレッド ID

戻り値

スレッド作成パラメータへのポインタ : 成功
 NULL : 失敗

解 説

rwGetThreadData は、tid にて獲得したスレッドの rwCreateThreadEx あるいは rwCreateProcessEx に送るユーザーパラメータを検索します。
 (rwCreateThreadEx を例として参照してください。)
 このスレッドが rwCreateThreadEx あるいは rwCreateProcess から作成された場合は、この関数は NULL を戻します。

A. 6. 11 rwGetTid

構 文

long rwGetTid(void)

引 数

なし

戻り値

現在のスレッドのスレッド ID

解 説

rwGetTid は、現在のスレッドのスレッド ID を戻します。

参 照

rwGetMainTid, rwLocateThread も参照してください。

A. 6. 12 rwKillThread

構 文

```
int rwKillThread(long Tid)
```

引 数

型	引数名	説 明
long	tid	終了されるスレッドの ID

戻り値

TRUE : 成功
FALSE : 失敗

解 説

rwKillThread は、実行中の指定したスレッドを無条件に終了します。
この関数は、終了させるスレッドが未知の状態にある可能性もあるので、注意して使用してください。

参 照

rwExitThread, rwLocateThread を参照してください。

A. 6. 13 rwLeaveCriticalSection

構 文

```
void rwLeaveCriticalSection(long ps)
```

引 数

型	引数名	説 明
long	ps	rwEnterCriticalSection への呼び出しからの数値

戻り値

なし

解 説

rwLeaveCriticalSection は、通常操作へのタスクの切り替えを戻します。

参 照

rwEnterCriticalSection, rwSeizeProcessor, rwReleaseProcessor も参照してください。

A. 6. 14 rwLocateThread

構 文

long rwLocateThread(char* ThreadName)

引 数

型	引数名	説 明
char*	ThreadName	スレッド名に一致する null 終了文字列を示すポインタ

戻り値

指定したスレッド名を照合するスレッドの ID
そのスレッド名が現在のスレッド名のどれとも適合しなかった場合、-1 を返します。

解 説

rwLocateThread は実行中のスレッドが他のスレッドのスレッド ID を調べるときに使用します。
実行中のスレッドがこの後に、他のスレッドと通信する際にメッセージングやシグナルを使用する場合があります。

参 照

rwCreateThread も参照してください。

A. 6. 15 rwParentTid

構 文

```
long rwParentTid(long tid)
```

引 数

型	引数名	説 明
long	tid	ペアレントを獲得するためのスレッド ID

戻り値

-1 : 失敗
 ペアレントスレッドの TID : 成功

解 説

rwParentTid は、tid にて指定されたスレッドのペアレントスレッド ID を検索します。

rwCreateThread、あるいは rwCreateThreadEx 関数を使用して作成される場合は、スレッドはペアレントを備えています。

そのペアレントは、元々プロセスを始めるものとして定義されます。

(例 : RTOS-WinJ において最初に作成されるプロセス／スレッド、あるいは rwCreateProcess、または rwCreateProcessEx によって作成されるプロセス)

例えばプロセス A が rwCreateThread にてスレッド 1 を作成し、その後スレッド 1 が rwCreateThread にてスレッド 2 を作成する場合は、スレッド 2 のペアレントは引き続きプロセス A であり、これは両方のスレッドのペアレントとなります。

参 照

rwGetTid も参照してください。

A. 6. 16 rwReleaseProcessor

構 文

void rwReleaseProcessor(void)

引 数

なし

戻り値

なし

解 説

rwReleaseProcessor は、rwSeizeProcessor の後に呼び出されるものであり、これによって CPU が Windows にサービスすることが可能になります。詳細は rwSeizeProcessor を参照してください。

参 照

rwSeizeProcessor, kEnterCriticalSection, rwLeaveCriticalSection も参照してください。

A. 6.17 rwResume

構 文

```
int rwResume(long Tid)
```

引 数

型	引数名	説 明
long	Tid	実行開始（再使用可能）するスレッドの ID

戻り値

TRUE : 成功
FALSE : 失敗

解 説

rwResume は以前中断していたスレッドの実行を再び可能にします。

参 照

rwSuspend, rwKillThread も参照してください。

A. 6. 18 rwSeizeProcessor

構 文

```
void rwSeizeProcessor(void)
```

引 数

なし

戻り値

なし

解 説

rwSeizeProcessor は通常インタラプトルーチンにおいて使用しているのを、rwReleaseProcessor がシグナルを送ったスレッドで呼び出されるまでは、Windows に CPU リソース機能を装備させません。

参 照

rwReleaseProcessor, rwEnterCriticalSection, rwLeaveCriticalSection も参照してください。

A. 6. 19 `rwSetPriority`

構 文

```
int rwSetPriority(long tid, long pri)
```

引 数

型	引数名	説 明
long	tid	優先度設定を行うスレッド ID
long	pri	新しい優先度

戻り値

```
1      : 成功
0      : 失敗
```

解 説

`rwSetPriority` は、tid で指定したスレッドの優先度を設定します。
 新しい優先度は、すべて 0 と定数 `MAXIMUM_PRIORITY` の間にある必要があります。
`MAXIMUM_PRIORITY` は 31 と定義されます。

参 照

`rwGetPriority` も参照してください。

A. 6. 20 rwSuspend

構 文

```
int rwSuspend(long Tid)
```

引 数

型	引数名	説 明
long	Tid	一時中断するスレッドの ID

戻り値

TRUE : 成功
FALSE : 失敗

解 説

rwSuspend は、指定したスレッドの実行を一時中断します。
このスレッドは、rwResume が呼び出されるまで中断したままになります。

A. 6. 21 rwThreadState

構 文

long rwThreadState(long Tid)

引 数

型	引数名	説 明
long	Tid	スレッドの ID

戻り値

指定したスレッドの実行状態
この Tid が無効な場合、-1 は戻ります。

解 説

rwThreadState は実行状態を戻します。 次の説明に適合するスレッドステート定義に rwThreadState を使用してアクセスする際に、rwSystem.h ファイルが含まれる必要があります。

定数	解説
STATE_READY	スレッドが使用可能状態で、タイムスライスを待っている
STATE_RECEIVE	スレッドがメッセージあるいはシグナルを待つてブロックされている
STATE_SEND	スレッドがメッセージを送信し、返答をまっている
STATE_SUSPEND	スレッドがほかのスレッドによって一時中断されている
STATE_WAIT	スレッドがスリープ状態 (rwSleep) あるいは遅れている (rwDelay)
STATE_FREE	スレッドが実行中ではない

参 照

rwSuspendThread, rwResumeThread も参照してください。

A. 6. 22 rwYield

構 文

void rwYield(void)

引 数

なし

戻り値

なし

解 説

rwYield は、現在のスレッドのタイムスライスの残りを、次に使用可能な同じ優先度のスレッドにリリースします。

参 照

rwSleep, rwDelay も参照してください。

A.7 スレッド間通信、メッセージ

A.7.1 rwCreceive

構 文

```
long rwCreceive(long tid, void* Data, long Length)
```

引 数

型	引数名	説 明
long	tid	受信するスレッド ID あるいは定数 ANYTID
void*	Data	データを置くバッファへのポインタ
long	Length	最長受信データ

戻り値

送信 (rwSend) スレッドのスレッド ID : メッセージ到達時成功
 定数 SIGNALED : シグナル到達時成功
 -1 : 使用可能メッセージ / シグナルなし

解 説

rwCreceive と rwReceive は、重要な違いが1つありますが非常に似ているものです。rwCreceive は、メッセージを待ちながらブロックできません。RwCreceive は条件付きで受信します。
 例えば rwCreceive は1つが中断していると（すでに送信されている）、メッセージやシグナルを受信するということです。メッセージやシグナルが中断中でなければ、rwCreceive は失敗コード (-1) とともにすぐに戻ります。

参 照

rwReceive も参照してください。

A.7.2 rwReceive

構文

```
long rwReceive(long Tid, void* Mesrwge, long MsgLentgh)
```

引 数

型	引数名	説 明
long	Tid	メッセージが受けられる際のスレッドの ID
void*	Mesrwge	到達メッセージを保存する場所のアドレス
long	MsgLentgh	最長到達メッセージ

戻り値

到達データパケットがメッセージである時 : 送信 (rwSend) スレッド
のスレッド ID

到達データパケットがシグナルである時 : 定数 SIGNED

-1 : Tidが無効である場合

解説

rwReceive はほかのスレッドから到達メッセージおよびシグナルを受け入れ、あるいはインタラプトルーチンからシグナルを受け入れます。

rwReceive を呼び出すこのスレッドは、到達メッセージやシグナルがそれを送信するまでブロックします。

メッセージ

指定のスレッド ID は、あるスレッドからメッセージを受け入れるときのみ使用され、あるいは定数 ANYTID はスレッドすべてからのメッセージを受け入れる際に使用される場合があります。

注意：スレッド ID が指定されている場合でさえも、どのソースからの
シグナルもスレッドによって引き続き受信されます。

メッセージを受信する全てのスレッドは、送信スレッドに対して返信 (rwReply) する必要があります (あるいは、送信スレッドは必ずブロックします)。

シグナル

シグナルは返信を必要としません。シグナルはメッセージより先に発生します。

中断中のすべてのシグナルは、メッセージが受信される前に受信されなければなりません。

参 照

rwSend, rwReply, rwSignal, rwClearSignals, rwReadSignals も参照してください。

A.7.3 rwReply

構 文

```
long rwReply(long Tid, void* Reply, long ReplyLength)
```

引 数

型	引数名	説 明
long	Tid	返信するスレッドの ID
void*	Reply	返信を示すポインタ
long	ReplyLength	返信の長さ（バイト単位）

戻り値

送信スレッドのスレッド ID
 スレッドが返信を受信する前に終了されたり、あるいはスレッドがメッセージを現在のスレッドに送信しなかった場合、-1 は戻されます。

解 説

rwReply は、メッセージ（シグナルではありません）をすでに送信したスレッドに返信します。送信スレッドはその返信を受け付け、そして非ブロック化します。
 rwReply を呼び出すスレッドはブロックしません。

注意： rwReply は、rwReceive がメッセージを受信したあとに発生しなければなりません。
 スレッドが rwReply にシグナルしか送らない場合は、実は不正で、要求はされません。

参 照

rwSend, rwReceive も参照してください。

A.7.4 rwSend

構文

```
long rwSend(long Tid, void* Meswrge, long MsgLength, void* Reply,
            long ReplyLength)
```

引 数

型	引数名	説 明
long	Tid	デスティネーションスレッドの ID
void*	Meswrge	送信するメッセージを示すポインタ
long	MsgLength	返信を受信する場所を示すポインタ
void*	Reply	送信するメッセージを示すポインタ
long	ReplyLength	返信で受け付ける最大バイト数

戻り値

メッセージを受信するスレッドの ID	:	成功
スレッドが存在しない場合や、返信前に終了した場合、-1	:	失敗

解説

rwSend は指定されたスレッドにメッセージを送ります。
送信先スレッドがメッセージを受信し (rwReceive)、応答 (rwReply) を返すまで送信元スレッドはブロックされます。

参 照

rwReceive, rwReply も参照してください。

A.8 スレッド間通信、セマフォ

A.8.1 rwAllocSemaphore

構 文

```
long rwAllocSemaphore(long tid)
```

引 数

型	引数名	説 明
long	tid	セマフォのスレッド ID

戻り値

0 : 成功
-1 : 失敗

解 説

rwAllocSemaphore は、すでに rwCreateSemaphore 関数と共に作成されたセマフォを割り当てます。一旦割り当てられると、スレッドが同じセマフォを割り当てようとする場合、ほかの NT や RTOS-WinJ スレッドは、最初の割り当てが rwFreeSemaphore からそのセマフォをフリーにするまでブロックします。

このようにして、共有リソースへのアクセスは、rwAllocSemaphore/rwFreeSemahore の2つにおいてこのリソースを送り込み・回り込みすることによって、コントロールする場合があります。

このセマフォを認識するスレッド ID は、rwCreateSemaphore または rwLocateSemaphore によって戻されたスレッド ID です。

参 照

rwCreateSemaphore, rwDestroySemaphore, rwFreeSemaphore, rwLocateSemaphore も参照してください。

A. 8.2 rwCreateSemaphore

構 文

long rwCreateSemaphore(char* Name, unsigned long Priority)

引 数

型	引数名	説 明
char*	Name	セマフォのネームを示すポインタ
unsigned long	Priority	セマフォの優先度

戻り値

セマフォのスレッド ID : 成功
 -1 : 失敗

解 説

rwCreateSemaphore は、共有の共通リソースへのアクセスをコントロールするための名前付きセマフォを作成します。(例えば共有メモリ領域)。作成したあと、このセマフォは、共有リソースにアクセスするために RTOS-WinJ や NT スレッドによって割り当てられ、フリーにされます。

セマフォを作成することで、セマフォと同じ名前のもので RTOS-WinJ スレッドが作成されるので、通常のスレッドネームとの衝突を防ぐように注意を払う必要があります。

さらにセマフォは rwDestroySemaphore 関数を使用することによって、これ以上必要性がなくなった際に破棄されます。

参 照

rwDestroySemaphore, rwAllocSemaphore, rwFreeSemaphore, rwLocateSemaphore も参照してください。

A. 8.3 rwDestroySemaphore

構 文

long rwDestroySemaphore(long tid)

引 数

型	引数名	説 明
long	tid	セマフォのスレッド ID

戻り値

0 : 成功
 -1 : 失敗

解 説

rwDestroySemaphore は、rwCreateSemaphore 関数と共に以前作成されたセマフォを破棄します。

このセマフォを認識するスレッド ID は、rwCreateSemaphore または rwLocateSemaphore が戻したスレッド ID です。

セマフォは、アプリケーションでは必要性がなくなった時は常に破棄しなくてはなりません。

このセマフォはフリーになるまで実際破棄されませんが、スレッドを破棄できるのは、セマフォを割り当てた同じスレッドです。

参 照

rwCreateSemaphore, rwAllocSemaphore, rwFreeSemaphore, rwLocateSemaphore も参照してください。

A. 8. 4 **rwFreeSemaphore**

構 文

long rwFreeSemaphore(long tid)

引 数

型	引数名	説 明
long	tid	セマフォのスレッド ID

戻り値

0 : 成功
 -1 : 失敗

解 説

rwFreeSemaphore は、rwAllocSemaphore 関数ですでに割り当てられたセマフォをフリーにします。 このセマフォをフリーにすれば、他の NT や RTOS-WinJ スレッドがセマフォを割り当てることができます。
 （そして rwAllocSemaphore をすでに呼び出したスレッドを非ブロック化します。）このセマフォを認識するスレッド ID は、rwCreateSemaphore または rwLocateSemaphore が戻したスレッド ID です。

参 照

rwCreateSemaphore, rwDestroySemaphore, rwAllocSemaphore, rwLocateSemaphore も参照してください。

A. 8. 5 rwLocateSemaphore

構 文

long rwLocateSemaphore(char* Name)

引 数

型	引数名	説 明
char*	Name	セマフォのネームを示すポインタ

戻り値

セマフォのスレッド ID : 成功
 -1 : 失敗

解 説

rwLocateSemaphore は、以前作成されたセマフォの位置を
 rwCreateSemaphore 関数から確認します。(そしてそのセマフォのスレッド
 IDを戻します)。このスレッド IDは、rwAllocSemaphoreと rwFreeSemaphore
 関数からセマフォを使用する必要があります。

セマフォのスレッド IDを獲得するためにセマフォを作成しなかった
 RTOS-WinJ あるいは NT スレッドが rwLocateSemaphore を使用します。

参 照

rwCreateSemaphore, rwDestroySemaphore, rwAllocSemaphore,
 rwFreeSemaphore も参照してください。

A.9 スレッド間通信、新セマフォ関数

A.9.1 rwOpenSemaphore

構 文

Semaphore* rwOpenSemaphore(char* Name)

引 数

型	引数名	説 明
char*	Name	セマフォ名へのポインタ (最大 16 字)

戻り値

セマフォを示すポインタ : 成功
 NULL : 失敗

解 説

rwOpenSemaphore は、パラメータとして Name で指定したセマフォを作成したり、開きます。この Name は、NULL で終了する長さ最大 16 文字 ASCII 文字列です。(NULL の終端バイトを含みません)。

指定した名称のセマフォがすでに存在している場合、この関数はセマフォの使用カウントを増やし、ポインタをセマフォに戻します。

このセマフォがまだ存在していない場合は、セマフォを作成して初期化し、そしてポインタをセマフォに戻します。

rwCreateSemaphore と共に作成されたセマフォと一緒に rwOpenSemaphore を使用しないでください。

作成したセマフォや rwOpenSemaphore と共に開いたセマフォを操作するためには、rwTrySemaphore, rwGetSemaphore, rwReleaseSemaphore, そして rwCloseSemaphore だけを使用してください。

スレッドを開くにあたって必要性のなくなったセマフォ用に rwCloseSemaphore を呼び出してください。

使用にあたりスレッドが開かれたそのスレッドを必要とする限り、セマフォは開いたままにします。

処理から抜ける際には、スレッドは開いたセマフォすべてを常に閉じる必要があります。

参 照

rwTrySemaphore, rwGetSemaphore, rwReleaseSemaphore, rwCloseSemaphore も参照してください。

A. 9.2 rwGetSemaphore

構 文

long rwGetSemaphore (Semaphore* semaphore)

引 数

型	引数名	説 明
Semaphore*	semaphore	rwOpenSemaphore が戻したセマフォのポインタ

戻り値

1 : 成功
0 : 失敗

解 説

rwGetSemaphore は、パラメータとして伝えられたポインタに関連するセマフォの所有権を獲得しようと試みます。

このセマフォがすでに他のスレッドによって所有されている場合は、この呼び出し (caller) は、現在の所有スレッドがそのセマフォをリリースするまでブロックします。

2つ以上のスレッドが1つのセマフォを待っている場合、優先度の高いスレッドが最初に所有権を主張してきます。

同じ優先度のマルチスレッドが待機中の場合、もっとも長く待機していたスレッドが最初に所有権を獲得します。

セマフォの所有権の必要性がなくなった時、この呼び出しスレッドは rwReleaseSemaphore を呼び出す必要があります。

参 照

rwTrySemaphore, rwOpenSemaphore, rwReleaseSemaphore, rwCloseSemaphore も参照してください。

A. 9.3 rwTrySemaphore

構 文

long rwTrySemaphore (Semaphore* semaphore)

引 数

型	引数名	説 明
Semaphore*	semaphore	rwOpenSemaphore が戻したセマフォのポインタ

戻り値

- 1 : 成功（所有権は獲得されています）
- 0 : 失敗（セマフォはすでに所有されています）

解 説

rwTrySemaphore は、パラメータとして伝えられたポインタに関連するセマフォの所有権を獲得しようと試みます。そのセマフォがすでに他のスレッドによって所有されている場合は、この関数は0の戻り値を返し、すぐに戻ります。このセマフォが現在所有されていない場合は、この関数はrwGetSemaphore 関数と同じように、正確にそのセマフォの所有権を獲得します。

このセマフォの所有権に必要性がなくなった時、この呼び出しスレッドはrwReleaseSemaphore 関数を呼び出す必要があります。

参 照

rwReleaseSemaphore, rwGetSemaphore, rwOpenSemaphore, rwCloseSemaphore も参照してください。

A. 9. 4 `rwReleaseSemaphore`

構 文

`long rwReleaseSemaphore (Semaphore* semaphore)`

引 数

型	引数名	説 明
Semaphore*	semaphore	<code>rwOpenSemaphore</code> が戻したセマフォのポインタ

戻り値

1 : 成功
0 : 失敗

解 説

`rwReleaseSemaphore` は、`rwGetSemaphore` や `rwTrySemaphore` 関数の使用を以前許可されたセマフォの所有権をリリースします。

`rwGetSemaphore` や `rwTrySemaphore` の所有権のあるスレッドは、他のスレッドがセマフォが保護しているリソースを使用している場合、その不必要なブロッキングを防いだらできるだけ早めにセマフォをリリースしなければなりません。

参 照

`rwTrySemaphore`, `rwGetSemaphore`, `rwOpenSemaphore`, `rwCloseSemaphore` も参照してください。

A. 9.5 rwCloseSemaphore

構 文

long rwCloseSemaphore (Semaphore* semaphore)

引 数

型	引数名	説 明
Semaphore*	semaphore	rwOpenSemaphore が戻したセマフォのポインタ

戻り値

1 : 成功
0 : 失敗

解 説

rwCloseSemaphore は、rwOpenSemaphore で開かれたセマフォを閉じます。
そのセマフォを開こうとするスレッドが他にない場合、セマフォを閉じることでそれを破棄します。そのセマフォを開こうとするスレッドがある場合は、rwCloseSemaphore は、そのセマフォの使用カウントを減少させます。

参 照

rwTrySemaphore, rwOpenSemaphore, rwReleaseSemaphore, rwGetSemaphore も参照してください。

A.10 スレッド間通信、シグナル

A.10.1 `rwClearSignals`

構 文

```
unsigned long rwClearSignals(long Tid)
```

引 数

型	引数名	説 明
long	Tid	スレッドの ID

戻り値

一時中断しているシグナルの数

解 説

`rwClearSignals` は、Tid にて指定したスレッドにて一時中断しているシグナル全てを削除し、中断中のシグナルの数を戻します。

参 照

`rwSignal`, `rwReadSignals`, `rwReceive` も参照してください。

A. 10.2 rwReadSignals

構 文

```
unsigned long rwReadSignals(long Tid)
```

引 数

型	引数名	説 明
long	Tid	中断中シグナルへクエリーするスレッドの ID

戻り値

Tid で指定したスレッドで一時中断しているシグナルの数

解 説

rwReadSignals は、指定したスレッドの一時中断中のシグナル数を戻します。

参 照

rwSignal, rwClearSignals, rwReceive も参照してください。

A. 10.3 `rwSignal`

構 文

`long rwSignal(long Tid)`

引 数

型	引数名	説 明
long	Tid	シグナルを発信するスレッドの ID

戻り値

シグナルが送信されたスレッド ID : 成功
-1 : 失敗

解 説

`rwSignal` は、指定したスレッドへシグナルを発信します。この呼び出しスレッドはブロックしません。スレッドで一時中断できるシグナルの最大数は 4 2 9 4 9 6 7 2 9 6（2 の 3 2 乗）です。

参 照

`rwReceive`, `rwClearSignals`, `rwReadSignals` も参照してください。

A.11 デバッグ

A.11.1 rwDebug

構 文

```
long rwDebug(char* Format, ... )
```

引 数

型	引数名	説 明
char*	Format	ステートメントをフォーマットする標準 C プリント f を使用
...	...	フォーマット文字列に対するアーギュメント

戻り値

0 : 成功
-1 : 失敗

解 説

rwDebug は、完了するまで呼び出しスレッドを遅らせるダイレクトプリント構造です。呼び出しがフォーマット化されたテキストの文字列をアウトプット（出力）ウィンドウに送信します。このウィンドウは Windows で実行しているものです。このアクションによって、プログラマーはフォーマットされた出力（アウトプット）をウィンドウに送信することができます。

注意： アウトプット（出力）文字列の長さの最大値は255バイトです。

警告： リアルタイムパフォーマンスが重要である時には、この関数を使用することは避けてください。rwDebug は目的をデバッグするための開発ツールとしてのみ使用してください。この関数によって、デバッグメッセージをすぐに（待たずに）表示することができますが、Windows が、リアルタイムに処理を行っているものを中断させるメッセージを送るまでこの関数は戻りません。ご使用のリアルタイムアプリケーションに深刻な影響の少ないプリント関数に関しては、rwPrint を参照してください。表示には多少の時間がかかり、ご使用のリアルタイムアプリケーションにある程度の影響を及ぼすことにご注意ください。

参 照

rwPrint も参照してください。

A.11.2 rwPrint

構 文

```
long rwPrint(char* Format, ... )
```

引 数

型	引数名	説 明
char*	Format	ステートメントをフォーマットする標準 C プリント f を使用
...	...	フォーマット文字列に対するアーギュメント

戻り値

0 : 成功
-1 : 失敗

解 説

rwPrint は、フォーマットされたテキストの文字列をアウトプット（出力）ウィンドウに送信します。このウィンドウは Windows で実行しているものです。このテキスト文字列はバッファに置かれ（最大 64 メッセージがバッファに置かれます）、システムタイムが可能な場合、出力（アウトプット）します。このバッファリングは、データのプリントが呼び出しスレッドのタイミングを中断しないようにします。

注意： 出力（アウトプット）文字列の長さの最大値は 255 バイトです。

参 照

rwDebug も参照してください。

A.12 インタラプト

A.12.1 `rwInterruptAttach`

構 文

```
int rwInterruptAttach(long IRQNumber, void* ISRAddress)
```

引 数

型	引数名	説 明
long	IRQNumber	インタラプトの数（0 から 15）
void*	ISRAddress	IRQ が発生した際呼び出すインタラプトサービスルーチン

戻り値

TRUE : 成功
 FALSE : 失敗

解 説

`rwInterruptAttach` は、関数をハードウェアインタラプトに付属させます。このハードウェアインタラプトが生じたとき、`ISRAddress` で指定された関数が呼び出されます。この関数は、現在スレッドがブロック化できるいかなる関数も呼び出しません。

ISR 関数ができるだけ素早く実行することは重要であり、その後、システムを通常操作に戻してください。重要度の低い操作を遅らせるために、ISR は（`rwSignal` を使用して）待機スレッドを発信し、できるだけ素早く ISR を抜ける必要があります。

rwInterruptAttach への呼び出しに続くインタラプトを常に使用不可にしてください。RTOS-WinJ システムがこの承認を実行していく中で、ISR は直接インタラプトコントローラを承認しません。ISR はインタラプトを引き起こすハードウェア上で、インタラプト要求をリセットする役割を担っています。

注意：浮動小数点の算術がインタラプトルーチン内で実行されている時は、手動で浮動小数点コンテキストを保護してください。

ISR を以下のように定義してください。つまり、パラメータも戻り値もないということです。

```
void InterruptFunc(void)
```

参 照

rwInterruptDetach, rwInterruptEnable, rwInterruptDirwble も参照してください。

A. 12.2 `rwInterruptDetach`

構 文

```
int rwInterruptDetach(long IRQNumber)
```

引 数

型	引数名	説 明
long	IRQNumber	ハードウェアインタラプトの数（0 から 15）

戻り値

TRUE : 成功
FALSE : 失敗

解 説

`rwInterruptDetach` は、呼び出し（関数の）`rwInterruptAttach` が以前付属したハードウェアインタラプトをから ISR 関数を分離します。
この関数の実行後に、ハードウェアインタラプトが起こった時（発生したとき）、ISR は RTOS-WinJ から呼び出されません。

参 照

`rwInterruptAttach`, `rwInterruptEnable`, `rwInterruptDirwble` も参照してください。

A. 12.3 rwInterruptDirwble

構 文

```
int rwInterruptDirwble(long IRQNumber)
```

引 数

型	引数名	説 明
long	IRQNumber	使用不可能にするハードウェアインタラプトの数（0 から 15）

戻り値

TRUE : 成功
FALSE : 失敗

解 説

rwInterruptDirwble は、割込み発生時に指定されたハードウェアインタラプトを使用不可能にしますが、ISR 関数を分離しません。

参 照

rwInterruptEnable, rwInterruptAttach, rwInterruptDetach も参照してください。

A. 12.4 `rwInterruptEnable`

構 文

```
int rwInterruptEnable(long IRQNumber)
```

引 数

型	引数名	説 明
long	IRQNumber	使用可能にするハードウェアインタラプトの数

戻り値

TRUE : 成功
FALSE : 失敗

解 説

`rwInterruptEnable` は、インタラプトコントローラー上で指定されたインタラプトを使用可能にします。ISR はハードウェアでのインタラプトをリセットする義務があります。

参 照

`rwInterruptAttach`, `rwInterruptDetach`, `rwInterruptDirwble` も参照してください。

A.13 インプット／アウトプット（入力／出力）

A.13.1 rwInp, rwInpw, rwInpd

構 文

```
int rwInp(unsigned short port)
unsigned short rwInpw(unsigned short port)
unsigned long rwInpd(unsigned short port)
```

引 数

型	引数名	説 明
unsigned short	Port	読み取るための I/O ポート

戻り値

指定した I/O ポートの位置から読み取られたバイト、ワード、ダブルワード

解 説

rwInp, rwInpw, rwInpd は、指定した I/O ポートからバイト、ワード、ダブルワードを読み取ります。

参 照

rwOutp, rwOutpw, rwOutpd も参照してください。

A. 13.2 rwOutp, rwOutpw, rwOutpd

構 文

```
int rwOutp(unsigned short Port, int Value)
unsigned short rwOutpw(unsigned short Port,
                        unsigned short Value)
unsigned long rwOutpd(unsigned short Port, unsigned long Value)
```

引 数

型	引数名	説 明
unsigned short	Port	I/O ポート
int	Value	I/O に書き込むための数値
unsigned short	Value	I/O に書き込むための数値
unsigned long	Value	I/O に書き込むための数値

戻り値

出力された数値

解 説

rwOutp, rwOutpw, rwOutpd は、指定した I/O ポートのロケーションへバイト、ワード、ダブルワードのデータをアウトプットします。

参 照

rwInp, rwInpw, rwInpd も参照してください。

A.14 トレースデバッグ

A.14.1 rwDebugTraceEvent

構 文

```
#include "rwernel.h"
#include "tracefunc.h"
void rwDebugTraceEvent(unsigned long EventCode,
                        unsigned long FunctionCode,
                        unsigned long Parameter)
```

引 数

型	引数名	説 明
unsigned long	EventCode	RW_TRACE_EVENT_USER_EVENT に設定される必要がある
unsigned long	FunctionCode	ユーザー定義コード
unsigned long	Parameter	ユーザー定義パラメータ

戻り値

なし

解 説

rwDebugTraceEvent によって、パラメータはトレースバッファにおいてトレースイベントを挿入することができます。この関数がユーザープログラムの中で呼び出されるときに、トレースイベントはデバッグトレースバッファに挿入されます。この関数コード及びパラメータの引数は、ユーザーが希望するあらゆる 32 ビットに設定される可能性があります。

これら 2 つの数値と一緒にこの記録は、RTOS-WinJ デバッグトレースユーティリティプログラムのデバッグトレースにおいて表示されます。

注意 1 : EventCode は、RW_TRACE_EVENT_USER_EVENT (例 : 0x0400) に常に設定されている必要があります。

他の数値はすべてによって、不適当なトレースイベントがトレースバッファ内に発生することになります。

注意 2 : この関数を使用するときは、他のインクルードファイルである tracefunc.h が必要ですが、これには必ず trace.h が含まれます。

A. 14. 2 `rwDumpDebugTrace`

構 文

`rwDumpDebugTrace(long ClearBuffer)`

引 数

型	引数名	説 明
long	ClearBuffer	トレースバッファをクリアにするフラグ

戻り値

1 : 成功
0 : 失敗

解 説

`rwDempDebugTrace` は、トレースバッファをダンプファイルにダンプします。このダンプファイルのネームは `rwSetTraceDumpFile` コールを使って指定します。

`rwDumpDebugTrace` 関数が呼び出されたときは、常にこのダンプファイルのコンテンツは上書きされます。ユーザーがこのダンプファイル保護したい場合は、`rwSetTraceDumpFile` への新しい呼び出しを作成し、このファイルネームをダンプして変更するか、あるいは現在のダンプファイルを異なったファイルネームにコピーしてから、再び `rwDumpDebugTrace` を呼び出してください。

このダンプファイルはバイナリーフォーマットの中にあります。RTOS-WinJ デバッグトレースユーティリティプログラムで、このトレースファイルをフォーマットして人が読み取れるフォーマットに直します。(HyperShare 関数の `rwFormatTraceFile` および `rwFormatTraceFileCSV` も参照してください。)

ClearBuffer フラグは、rwDumpDebugTrace 関数が、ダンプ後にトレースバッファをクリアにするかどうかを表示します。ゼロではない数値は、ダンプされた後そのトレースバッファが空にされる必要があることを表示します。rwStartTraceMonitoring への呼び出しは、モニタリングを開始したときに、トレースバッファを自動的にクリアにします。

rwDumpDebugTrace は、モニタリングが実行していないときに唯一呼び出すことが可能です。(rwGetTraceState は、RW_TRACE_STATE_DORMANT を戻します。)

RW_TRACE_STATE_DORMANT は、trace.h で定義され、

```
#define RW_TRACE_STATE_DORMANT 1
```

となります。

トレースモニタリングを実行している間に、トレースファイルをダンプしようとする場合は (例: rwStartTraceMonitoring がすでに呼び出されていて、ストップトリガやフルバッファ、あるいは rwStopTracing コールがトレーシングを中止していないなど)、rwDumpDebugTrace は失敗します。

注意: この関数を使用するときは、他のインクルードファイルである tracefunc.h が必要ですが、これには必ず trace.h が含まれます。

参 照

rwSetTraceDumpFile, rwFormatTraceFile も参照してください。

A. 14.3 rwGetTraceBufferMode

構 文

```
#include "rwerne.h"
#include "tracefunc.h"
long rwGetTraceBufferMode(long* Mode)
```

引 数

型	引数名	説 明
long	Mode	現在のバッファモード用の保管位置

戻り値

1 : 成功
0 : 失敗

解 説

rwGetTraceBufferMode は、イベントをバッファする際に使用される現在モードを検索します。このモードは以下のように trace.h で定義されます。

```
#define RW_TRACE_BUFFER_ONESHOT 1
#define RW_TRACE_BUFFER_CIRCULAR 2
```

ワンショットバッファは、一旦バッファがフルになるとモニタリングイベントを中止します。イベントにおいて、フルのバッファは更なるストップトリガとして機能します。

循環バッファは、バッファがフルになった後もモニタリングイベントを続行します。もっとも古いイベントが循環式で上書きされます。

参 照

rwSetTraceBufferMode も参照してください。

注意： この関数を使用するときは、他のインクルードファイルである tracefunc.h が必要ですが、これには必ず trace.h が含まれます。

A. 14.4 `rwGetTraceBufferSize`

構 文

```
#include "rwerne.h"
#include "tracefunc.h"
long rwGetTraceBufferSize(long* BufferSize)
```

引 数

型	引数名	説 明
long*	BufferSize	現在のバッファサイズ用の保管位置

戻り値

1 : 成功
0 : 失敗

解 説

`rwGetTraceBufferSize` は、いくつかのエントリ（記録）においてトレースバッファのサイズを検索します。トレーシングが実行されている間、システム内で1つの記録がイベントごとに使用されます。これは、実際に使用可能である記録数とは同じではありません。（`rwTraceDataAvailable` を参照してください。）

注意：この関数を使用するときは、他のインクルードファイルである `tracefunc.h` が必要ですが、これには必ず `trace.h` が含まれます。

参 照

`rwSetTraceBufferSize` も参照してください。

A. 14.5 rwGetTraceState

構 文

```
#include "rwerne.h"
#include "tracefunc.h"
long rwGetTraceState(long* State)
```

引 数

型	引数名	説 明
long*	State	現在の状態を置くアドレスの場所

戻り値

```
1      : 成功
0      : 失敗
```

解 説

rwGetTraceState は、トレースモニタリングの現在の状態を獲得します。この異なったステートは、以下のように trace.h において定義されます。

```
#define RW_TRACE_STATE_NOTREADY 0
    //system still coming up, not ready to trace
#define RW_TRACE_STATE_DORMANT 1
    //tracing not active, tracing can be configured
#define RW_TRACE_STATE_MONITORING 2
    //rwStartTraceMonitoring called, looking for
    // start triggers
#define RW_TRACE_STATE_TRIGGERED 4
    //currently recording, start trigger has occurred
#define RW_TRACE_STATE_PAUSED 8
    //internal state, monitoring temporarily paused
```

注意：この関数を使用するときは、他のインクルードファイルである `tracefunc.h` が必要ですが、これには必ず `trace.h` が含まれます。

参 照

`rwSetTraceState` も参照してください。

A. 14. 6 `rwGetTraceTriggers`

構 文

```
#include "rwerne.h"
#include "tracefunc.h"

long rwGetTraceTriggers(unsigned long* StartTrigger,
                        unsigned long* Stoptrigger)
```

引 数

型	引数名	説 明
unsigned long*	StartTrigger	スタートトリガを置く保管位置
unsigned long*	StopTrigger	ストップトリガを置く保管位置

戻り値

1 : 成功
0 : 失敗

解 説

`rwGetTraceTriggers` は、イベントをトレースバッファに記録を開始（及び停止）させるトリガを獲得します。トレーストリガに関する詳細情報は `rwSetTraceTriggers` を参照してください。

注意：この関数を使用するときは、他のインクルードファイルである `tracefunc.h` が必要ですが、これには必ず `trace.H` が含まれます。

参 照

`rwSetTraceTriggers` も参照してください。

A. 14. 7 rwSetTraceBufferMode

構 文

```
#include "rwerne.h"
#include "tracefunc.h"
long rwSetTraceBufferMode(long Mode)
```

引 数

型	引数名	説 明
long	Mode	バッファリングモードを設定する

戻り値

1 : 成功
0 : 失敗

解 説

rwSetTraceBufferMode は、バッファリングイベント用モードを設定します。
このモードは以下のように trace.h において定義されます。

```
#define RW_TRACE_BUFFER_ONESHOT 1
#define RW_TRACE_BUFFER_CIRCULAR 2
```

ワンショットバッファは、一旦バッファがフルになるとモニタリングイベントを中止します。イベントトレーシングにおいて、フルのバッファは更なるストップトリガとして機能します。

循環バッファは、バッファがフルになった後もモニタリングイベントを続行します。 もっとも古いイベントが循環式で上書きされます。

注意： この関数を使用するときは、他のインクルードファイルである tracefunc.h が必要ですが、これには必ず trace.h が含まれます。

参 照

rwSetTraceBufferSize も参照してください。

A. 14. 8 rwSetTraceBufferSize

構 文

```
#include "rwerne.h"
#include "tracefunc.h"
```

long rwSetTraceBufferSize(long BufferSize)

引 数

型	引数名	説 明
long	BufferSize	エントリ数の中のトレースバッファのサイズ

戻り値

1 : 成功
0 : 失敗

解 説

rwSetTraceBufferSize は、エントリ（記録）の数におけるトレースバッファのサイズを設定します。トレーシングを実行している間にシステムにおいて1つの記録がイベントごとに使用されます。

rwSetTraceBufferSize は、モニタリングを実行していないときに唯一成功します。（例：rwGetTraceState への呼び出しは RW_TRACE_STATE_DORMANT を戻します。）

注意：この関数を使用するときは、他のインクルードファイルである tracefunc.h が必要ですが、これには必ず trace.h が含まれます。

参 照

rwSetTraceBufferMode も参照してください。

A. 14. 9 `rwSetTraceDumpFile`

構 文

```
#include "rwerne.h"
#include "tracefunc.h"
long rwSetTraceDumpFile (char* filename)
```

引 数

型	引数名	説 明
char*	filename	新しいトレースダンプのファイルネーム

戻り値

1 : 成功
0 : 失敗

解 説

`rwSetTraceDumpFile` は、デバッグトレースダンプファイル用のファイルネームを設定します。`rwDumpDebugTrace` が呼び出されるときに、トレースバッファは `rwSetTraceDumpFile` が指定したファイルにダンプされます。

注意： フルパスネームにて指定しなければなりません。

`rwSetTraceDumpFile` は、モニタリングを実行しているときに唯一成功します。(`rwGetTraceState` は、`RW_TRACE_STATE_DORMANT` を戻します。)

注意： この関数を使用するときは、他のインクルードファイルである `tracefunc.h` が必要ですが、これには必ず `trace.h` が含まれます。

参 照

`rwDumpDebugTrace`, `rwGetRraceDumpFile` も参照してください。

A. 14. 10 rwSetTraceTriggers

構 文

```
#include "rwerne.h"
#include "tracefunc.h"
long rwSetTraceTriggers(unsigned long StartTrigger,
                        unsigned long StopTrigger)
```

引 数

型	引数名	説 明
unsigned long	StartTrigger	モニタリングイベントを開始するトリガ
unsigned long	StopTrigger	モニタリングイベントを停止するトリガ

戻り値

1 : 成功
0 : 失敗

解 説

rwSetTraceTriggers は、イベントをトレースバッファに記録開始（及び停止）させるトリガを設定します。2つ以上のトリガは、スタートおよびストップトリガの両方用に設定される場合もあります。このマルチトリガ、あるいはトリガ識別子をシング 32 ット数値と一緒に設定してください。トリガの中には両方のトリガにおいて無効なものもあります。このトリガ識別子は以下のように trace.h において定義されます。

トリガ	数値	注釈
#define RW_TRACE_EVENT_ENTER_INTERRUPT	0x0001	1
#define RW_TRACE_EVENT_LEAVE_INTERRUPT	0x0002	1
#define RW_TRACE_EVENT_ENTER_FUNCTION	0x0004	1
#define RW_TRACE_EVENT_LEAVE_FUNCTION	0x0008	1
#define RW_TRACE_EVENT_ENTER_SYSTEM_TIMER	0x0010	1
#define RW_TRACE_EVENT_LEAVE_SYSTEM_TIMER	0x0020	1
#define RW_TRACE_EVENT_CALL_USER_TIMER	0x0040	1
#define RW_TRACE_EVENT_RETURN_USER_TIMER	0x0080	1
#define RW_TRACE_EVENT_ENTER_PROCESSOR_FAULT	0x0100	3
#define RW_TRACE_EVENT_LEAVE_PROCESSOR_FAULT	0x0200	3
#define RW_TRACE_EVENT_USER_EVENT	0x0400	1
#define RW_TRACE_EVENT_START_MONITORING	0x0800	2
#define RW_TRACE_EVENT_FAULT_LOCATION	0x1000	4
#define RW_TRACE_EVENT_SWITCH_TO_NT	0x2000	1
#define RW_TRACE_EVENT_SWITCH_TO_RW	0x4000	1
#define RW_TRACE_EVENT_NT_FAULT	0x8000	4

注意：

1. スタート あるいは ストップ
2. スタートのみ
3. ストップのみ
4. Hyper 内部にて使用しないでください

このコメントによって、トリガが有効な場所を認識します。継続的なモニタリング用のスタートトリガは、RW_TRACE_EVENT_START_MONITORING です。

このイベントは、rwStartTraceMonitoring() コールが作成されるとすぐに記録するイベントを誘発します。

RW_TRACE_EVENT_SWITCH_TO_RW も、有用なスタートトリガです。

NT から RW へのイベントの記録を次のコンテキストスイッチに記録を開始します。

既知の外部 RTOS-WinJ ハードウェアインタラプトが予期される場合は、RW_TRACE_EVENT_ENTER_INTERRUPT はスタートトリガ用の良い候補となります。(ストップトリガ用の停止条件に設定しようとしているもの)

(ストップトリガではない) 0 がストップトリガ用に指定される場合は、トレーシングは `rwStopTracing` コールを使用して停止したままの場合があります。

`rwSetTraceTriggers` は、モニタリングを実行していないときに唯一成功します。(`rwGetTraceState` は `RW_TRACE_STATE_DORMANT` を戻します。)

注意：この関数を使用するときは、他のインクルードファイルである `tracefunc.h` が必要ですが、これには必ず `trace.h` が含まれます。

参 照

`rwGetTraceTriggers`, `rwStartTraceMonitoring` も参照してください。

A. 14. 11 rwStartTraceMonitoring

構 文

```
#include "rwerne.h"
#include "tracefunc.h"
long rwStartTraceMonitoring(void)
```

引 数

なし

戻り値

1	:	成功
0	:	失敗

解 説

rwStartTraceMonitoring は、トリガ／イベント用モニタリングを開始します。このスタートトリガイイベントが一旦発生すると、その後すべてのイベントがさらにトレースバッファに記録されます。ストップトリガが一旦発生すると、(あるいはトレースバッファがワンショットトレースバッファでフルになると)、トレースモニタリング(及びイベントの記録)は終了します。このストップトリガ、あるいはフルになったバッファは、言い換えると自動的の rwStopTracing コールということです。rwStartTraceMonitoring が呼び出されたあとは、その時点までに検出されるイベントもトリガもありません。ストップトリガが一旦発生すると、そのバッファはワンショットトレースバッファでフルになります。あるいは rwStopTracing が呼び出されます。その後 rwStartTraceMonitoring が再び呼び出されるまで、検出・記録されるイベントやトリガはありません。

rwStartTraceMonitoring への呼び出しは、モニタリングが開始するときに、自動的にトレースバッファをクリアにします。

注意 1 : モニタリングが `rwStartTraceMonitoring` への呼び出しで動作する前に `kSetTraceBufferMode`, `rwSetTraceBufferSize` および `rwSetTraceTriggers` を呼び出す必要があります。

注意 2 : この関数を使用するときは、他のインクルードファイルである `tracefunc.h` が必要ですが、これには必ず `trace.h` が含まれます。

参 照

`rwSetTraceBufferMode`, `rwSetTraceBufferSize`, `rwSetTraceTriggers`, `rwStopTracing` も参照してください。

A. 14. 12 rwStopTracing

構 文

```
#include "rwerne.h"
#include "tracefunc.h"
long rwStopTracing (void)
```

引 数

なし

戻り値

1	:	成功
0	:	失敗

解 説

rwStopTracing は、トリガ／イベント用のモニタリングを停止し、スタートトリガがすでに発生している場合は、イベントの記録を停止します。ストップトリガは、要するに、rwStopTracing への呼び出しであり、これはストップトリガが発生するときにシステムが自動的に初期化するものです。ストップトリガ用に 0 が指定される場合は、その後 rwStopTracing コールを使用することが、モニタリングを停止する唯一の方法となります。

注意：この関数を使用するときは、他のインクルードファイルである tracefunc.h が必要ですが、これには必ず trace.h が含まれます。

参 照

rwStartTraceMonitoring も参照してください。

A. 14. 13 rwTraceDataAvailable

構 文

```
#include "rwerne.h"
#include "tracefunc.h"
long rwTraceDataAvailable (void)
```

引 数

なし

戻り値

0	:	失敗
記録の数	:	成功

解 説

rwTraceDataAvailable は、デバッグトレースバッファにおける記録の数を返します。

この記録の数は、トレースステートが RW_TRACE_STATE_DORMANT（例：1）ではない場合、0 として返されます。スタートトリガとストップトリガがすでに実行されている場合は、（あるいは rwStopTracing 関数コールがすでに作成されている場合は）、トレースバッファに記録が存在します。トレースバッファをディスクに書き込むために rwDumpDebugTrace が呼び出されます。

注意：この関数を使用するときは、他のインクルードファイルである tracefunc.h が必要ですが、これには必ず trace.h が含まれます。

参 照

rwDumpDebugTrace も参照してください。

A. 15 その他

A. 15.1 rwAttacrwystemTimer

構 文

```
int rwAttacrwystemTimer(void* func, void* param)
```

引 数

型	引数名	説 明
void*	func	呼び出す関数を示すポインタ
void*	param	関数へ伝えるパラメータ (NULL となる場合あり)

戻り値

1 : 成功
0 : 失敗

解 説

rwAttacrwystemTimer は、獲得した関数をシステムタイマーに付属させます。この関数はインタラプトコンテキストの中で呼び出され、インタラプトルーチンの内部で使用を許可されたこれらの関数だけの内部で獲得した関数を呼び出さなければなりません。

与えられた関数ポインタの関数ヘッダ用のシンタックスは以下にあげるようになる必要があります。

```
void FuncName(void* param)
```

rwAttacrwystemtime rに伝えられたこのパラメータの引数は、獲得した関数に伝えられます。このシステムタイマーは、RTOS-WinJ 設定ウィザードから設定された Standard Timing Timer 設定を倍にする速度で発動します。このウィザード内での有効なタイマー設定は 25, 50, 100 そして 250 マイクロセカンドです。そのため、この獲得した関数は、50, 100, 200, 500 マイクロセカンド毎にそれぞれ呼び出されるようになります。

注意：浮動小数点算術がインタラプトルーチン内で実行されている時は、手動で浮動小数点コンテキストを保護してください。

例

//以下の例では、システムタイマーを基本にしたスクエアウェーブの発生を説明しています。

```
long WaveState;
void SysTimer(void*context)
{
    if(WaveState) {
        rwOutp(DATA_PORT, 0x01);
        WaveState = 0;
    } else {
        rwOutp(DATA_PORT, 0x00);
        WaveState = 1;
    } //else
} //SysTimer

void main(void)
{
    WaveState = 0;
    rwAttacrwystemTimer(SysTimer, NULL);
    while(!rwQuitRequest()) {
    } //while
    kDetacrwystemTimer(SysTimer);
} //main
```

参 照

rwDetacrwystemTimer も参照してください。

A. 15.2 rwDetacrwystemTimer

構 文

```
int rwDetacrwystemTimer(void* func)
```

引 数

型	引数名	説 明
void*	func	呼び出す関数を示すポインタ

戻り値

1 : 成功
0 : 失敗

解 説

rwDetacrwystemTimer は、システムタイマーから獲得した関数を分離します。この関数のポインタは、rwAttacrwystemTimer に与えられるのと同じものとなります。そのスレッドが、付属関数の呼び出しを必要としなくなった時に、スレッド／プロセスが rwDetacrwystemTimer を呼び出すようになります。スレッド／プロセスが抜ける前に rwDetacrwystemTimer が常に呼び出されます。

参 照

rwAttacrwystemTimer も参照してください。

A. 15.3 rwGetPrivateProfileString

構 文

```
long rwGetPrivateProfileString(char* SectionName,
                               char* KeyName,
                               char* Default,
                               char* Destination,
                               long Size,
                               char* FileName)
```

引 数

型	引数名	説 明
char*	SectionName	セクション名を示すポインタ
char*	KeyName	キー名を示すポインタ
char*	Default	デフォルト文字列を示すポインタ
char*	Destination	デスティネーションバッファを示すポインタ
long	Size	デスティネーションバッファのサイズ
char*	FileName	ファイル名の初期化を示すポインタ

戻り値

0 : 失敗
 バッファにコピーされた文字数、終了 null 文字を含まない : 成功

注意 : SectionName あるいは KeyName が NULL である場合、そして与えられた（獲得した）デスティネーションバッファが小さすぎて要求された文字列をホールドできない場合、この文字列は切り捨てられ、その後 null 文字が続きます。そして戻り値は Size - 1 と等しくなります。

SectionName あるいは KeyName が NULL である場合、そして与えられた（獲得した）デスティネーションバッファが小さすぎて全ての文字列をホールドできない場合、最後の文字列が切り捨てられ、2つの null 文字が続きます。この場合、その戻り値は Size - 2 と等しくなります。

解 説

rwGetPrivateProfileString は、初期化ファイルの指定したセクションから文字列を検索します。この関数は WindowrwPI 関数である GetPrivateProfileString と同じものです。rwGetPrivateProfile は、実際 Win32 関数コールの GetPrivateProfileString に変換されます。詳細に関し

ては、Microsoft Win32API リファレンスマニュアルの
GetPrivateProfileString を参照してください。

参 照

rwWritePrivateProfileString も参照してください。

A. 15.4 rwGetSystemDirectory

構 文

```
void rwGetSystemDirectory(char* Buffer, int Length)
```

引 数

型	引数名	説 明
char*	Buffer	Windows システムディレクトリを検索するバッファを示すポインタ
int	Length	バッファの長さ（バイト単位）

戻り値

なし

解 説

rwGetSystemDirectory は、Windows システムディレクトリを含むテキスト文字列を、獲得した（与えられた）バッファのポインタにコピーします。

参 照

rwGetWindowsDirectory も参照してください。

A. 15.5 rwGetWindowsDirectory

構 文

```
void rwGetWindowsDirectory(void* Buffer)
```

引 数

型	引数名	説 明
void*	Buffer	Windows ディレクトリを検索するバッファを示すポインタ

戻り値

なし

解 説

rwGetWindowsDirectory は、Windows ディレクトリを含むテキスト文字列をバッファにコピーします。

参 照

rwGetSystemDirectory も参照してください。

A. 15.6 rwQuitRequest

構 文

```
int rwQuitRequest(void)
```

引 数

なし

戻り値

TRUE : システムがシャットダウン中の場合
FALSE : システムがシャットダウン中ではない場合

解 説

rwQuitRequest は、RTOS-WinJ がシャットダウンを要求された場合に TRUE を返します。

注意 : rwQuitRequest が TRUE を戻した後に送信される rwDebug あるいは rwPrint メッセージは、Windows のメッセージウィンドウには表示されません。

参 照

なし

A. 15.7 rwSetGateSpeed

構 文

long rwSetGateSpeed(long NewSpeed)

引 数

型	引数名	説 明
long	NewSpeed	Windows が管理者にサーブする際のミリ秒単位の新しいスピードは新しい要求をチェックする。最小値は 1。

戻り値

前のゲートスピード

解 説

rwSetGateSpeed は、メッセージをチェックするのに Windows が管理者（HyperGate）にサービスする際の速度を設定します。システムが開始するときに、この速度は 10 ミリ秒のデフォルトに設定されます。スレッドが I/O ファイルを実行したり、rwDebug や rwPrint からプリントしたりする時、ゲートスピードを下げると情報量が増えます。しかし通常的环境下ではゲートスピードを下げると RTOS-WinJ アプリケーションの情報量は減ります。

参 照

なし

A. 15.8 rwSystemCrash

構 文

```
int rwSystemCrash(void)
```

引 数

なし

戻り値

TRUE : Windows がクラッシュした場合
FALSE : Windows がクラッシュしなかった場合

解 説

Windows はクラッシュ（ブルースクリーン）したと RTOS-WinJ が判断している場合、rwSystemCrash は TRUE を返します。この関数によって RTOS-WinJ アプリケーションの適切なシャットダウンが可能です。

A. 15.9 rwTextOut

構 文

```
void rwTextOut(char* s)
```

引 数

型	引数名	説 明
char*	s	プリントするテキストを示すポインタ

戻り値

なし

解 説

rwTextOut は、NT ブルースクリーン状態でクラッシュした後、テキスト文字列をこのスクリーンにプリントします。このブルースクリーン状態でのクラッシュは、rwSystemCrash 関数から検出される場合もあります。HrwTextOut は、ブルースクリーンが発生したあと、唯一呼び出されるものです。この関数は重大なシステムクラッシュのイベントにおいて、ユーザーへの重要なデータをダンプする際に使用されます。

参 照

rwSystemCrash も参照してください。

A. 15. 10 rwWritePrivateProfileString

構 文

```
long rwWritePrivateProfileString(char* SectionName,
                                char* KeyName,
                                char* String,
                                char* FileName)
```

引 数

型	引数名	説 明
char*	SectionName	セクション名を示すポインタ
char*	KeyName	キー名を示すポインタ
char*	String	書き込む文字列を示すポインタ
char*	FileName	初期化ファイル名を示すポインタ

戻り値

0 : コピー失敗、あるいはもっとも最近にアクセスした初期化ファイルのクラッシュしたバージョンを点減
 >0 : 文字列のコピー成功

解 説

rwWritePrivateProfileString は、文字列を初期化ファイルの指定セクションに書き込みます。この関数は、WidnowrwPI 関数である WritePrivateProfileString と同じものです。rwWritePrivateProfileString は、Win32 関数コールである WritePrivateProfileString へ実際解釈されます。更なる詳細は、Microsoft Win32 API リファレンスマニュアルの WritePrivateProfile を参照してください。

参 照

rwGetprivateProfileString も参照してください。

B.1 システム

B.1.1 rwGetRWInstallDir

構 文

```
int rwGetRWInstallDir(char* Path)
```

引 数

型	引数名	説 明
char*	Path	パス用ロケーション

戻り値

1 : 成功
0 : 失敗

解 説

rwGetRWInstallDir は、システムがパスにインストールされているパスのロケーションを検索します。このパスは、通常
C:\Program Files\Nemasoft\RTOS-WinJ で、インストール時に変更します。

B.1.2 rwGetLastError

構 文

```
long rwGetLastError(void)
```

引 数

なし

戻り値

- | | | |
|---|---|--------------------------------|
| 0 | : | エラーなし |
| 1 | : | ドライバをスタートするように使用可能にする |
| 2 | : | システムドライバとコミュニケーションするよう使用不可能にする |
| 4 | : | シェアードメモリへ添付ができない |

解 説

rwGetLastError は、システムでおこった最後のエラーをリターンします。

注：このコールは現在、rwAttacrwharedRam ファンクションの失敗のみサポートします。

参 照

rwAttacrwharedRam も参照してください。

B.1.3 rwStartApplication

構 文

```
long rwStartApplication(char* rwExe, HWND DdgHwnd)
```

引 数

型	引数名	説 明
char*	rwExe	の実行可能なアプリケーションが開始される。
HWND	DbgHwnd	すべての rwDebug 列を受領する Windows のハンドル（これは現在リストボックスへハンドルされなければならない。

戻り値

1 : アプリケーションがスタートする時
0 : エラー

解 説

rwStartApplication は、供給されたアプリケーションをスタートします。

参 照

rwStopApplication も参照してください。

B.1.4 rwStopApplication

構 文

`void rwStopApplication(void)`

引 数

なし

戻り値

なし

解 説

rwStopApplication は、現在動作しているアプリケーションをストップします。

参 照

rwStartApplication も参照してください。

B.2 データアクセス

B.2.1 rwAttacrwharedRam

構 文

```
long rwAttacrwharedRam(void)
```

引 数

なし

戻り値

TRUE : 成功
FALSE : 失敗

解 説

rwAttacrwharedRam は、ユーザーアプリケーションがシェアードメモリ領域へアクセスできるようにし、rwRead あるいは rwWrite がシェアードメモリ領域のユーザー部分へ呼び出される前に必要となります。また、ユーザーはメモリアクセスが完了した時に rwDetacrwharedRam をコールしなければなりません。

注：もしコーリングアプリケーションが既に rwStartApplication をコールした場合は、このファンクションは必要ではありません。

参 照

rwDetacrwharedRam も参照してください。

B.2.2 rwDetacrwharedRam

構 文

```
long rwDetacrwharedRam(void)
```

引 数

なし

戻り値

TRUE : 成功
FALSE : 失敗

解 説

rwDetacrwharedRam はのシェアードメモリ領域へアクセスが完了する時に、コールされるファンクションです。

参 照

rwAttacrwharedRam も参照してください。

B.2.3 `rwNumberSharedBits`

構 文

```
long rwNumberSharedBits(void)
```

引 数

なし

戻り値

シェアードメモリ領域中に割り当てられたビット数

解 説

`rwNumberSharedBits` は、シェアードメモリ領域中に割り当てられ、使用可能なビット数をリターンします。

参 照

`rwNumberShared`, `rwRead`, `rwWrite`, `rwUserSharedRam` も参照してください。

B. 2. 4 rwNumberSharedBytes

構 文

```
long rwNumberSharedBytes(void)
```

引 数

なし

戻り値

シェアードメモリ領域に割り当てられたバイト数

解 説

rwNumberSharedBytes はシェアードメモリ領域内で割り当てられ、使用可能なバイト数をリターンします。

参 照

rwNumberShared, rwRead, rwWrite, rwUserSharedRam も参照してください。

B. 2. 5 rwNumberSharedLongs

構 文

long rwNumberSharedLongs(void)

引 数

なし

戻り値

シェアードメモリ領域に割り当てられた long 数

解 説

rwNumberSharedLongs はシェアードメモリ領域内で割り当てられ、使用可能な long 数をリターンします。

参 照

rwNumberShared, rwRead, rwWrite, rwUserSharedRam も参照してください。

B.2.6 `rwNumberSharedWords`

構 文

```
long rwNumberSharedWords(void)
```

引 数

なし

戻り値

シェアードメモリ領域に割り当てられたワード数

解 説

`rwNumberSharedWords` はシェアードメモリ領域内で割り当てられ、使用可能なワード数をリターンします。

参 照

`rwNumberShared`, `rwRead`, `rwWrite`, `rwUserSharedRam` も参照してください。

B. 2. 7 rwNumberUserSharedBytes

構 文

long rwNumberUserSharedBytes(void)

引 数

なし

戻り値

シェアードメモリ領域のユーザーシェアード部中に割り当てられたバイト数

解 説

rwNumberUserSharedLongs は、シェアードメモリ領域のユーザーシェアード部中に割り当てられ、使用可能なバイト数をリターンします。

参 照

rwNumberUserShared, rwReadUser, rwWriteUser, rwUserSharedRam も参照してください。

B. 2. 8 rwNumberUserSharedLongs

構 文

long rwNumberUserSharedLongs(void)

引 数

なし

戻り値

シェアードメモリ領域のユーザーシェアード部中に割り当てられたロング数

解 説

rwNumberUserSharedLongs は、シェアードメモリ領域のユーザーシェアード部中に割り当てられ、使用可能なロング数をリターンします。

参 照

rwNumberUserShared, rwReadUser, rwWriteUser, rwUserSharedRam も参照してください。

B. 2. 9 rwNumberUserSharedWords

構 文

```
long rwNumberUserSharedWords(void)
```

引 数

なし

戻り値

シェアードメモリ領域のユーザーシェアード部中に割り当てられたワード数

解 説

rwNumberUserSharedLongs は、シェアードメモリ領域のユーザーシェアード部中に割り当てられ、使用可能なワード数をリターンします。

参 照

rwNumberUserShared, rwReadUser, rwWriteUser, rwUserSharedRam も参照してください。

B.2.10 rwReadBit

構 文

long rwReadBit(long BitVar)

引 数

型	引数名	説 明
long	BitVar	読み取り用ビット数

戻り値

指定されたビットの値

-1 : エラー

解 説

rwReadBit は、シェアードメモリ領域中の指定されたビットの値をリターンします。

注：このファンクションは、rwAttacrwharedRam がコールされた後にのみ使用可能です。

参 照

rwAttacrwharedRam, rwDetacrwharedRam, rwNumberShared, rwRead, rwWrite, rwUserSharedRam も参照してください。

B.2.11 `rwReadByte`

構 文

`long rwReadByte(long ByteVar)`

引 数

型	引数名	説 明
long	ByteVar	読み取り用バイト数

戻り値

指定されたバイトの値
0 : エラー

解 説

`rwReadByte` はシェアードメモリ領域中で見つかった、指定されたバイトの値をリターンします。

注：このファンクションは、`rwAttacrwharedRam` がコールされた後にのみ使用可能です。

参 照

`rwAttacrwharedRam`, `rwDetacrwharedRam`, `rwNumberShared`, `rwRead`, `rwWrite`, `rwUserSharedRam` も参照してください。

B. 2. 12 rwReadLong

構 文

long rwReadLong(long LongVar)

引 数

型	引数名	説 明
long	LongVar	読み取り用 Long 数

戻り値

指定された Long の値

0 : エラー

解 説

rwReadLong はシェアードメモリ領域中で見つかった、指定された long の値をリターンします。

注：このファンクションは、rwAttacrwharedRam がコールされた後にのみ使用可能です。

参 照

rwAttacrwharedRam, rwDetacrwharedRam, rwNumberShared, rwRead, rwWrite, rwUserSharedRam も参照してください。

B. 2. 13 rwReadUserByte

構 文

long rwReadUserByte(long ByteVar)

引 数

型	引数名	説 明
long	ByteVar	読み取り用バイト数

戻り値

指定された Long の値
0 : エラー

解 説

rwReadUserByte は、シェアードメモリ領域のユーザーシェアード部中に見つかった、指定されたバイトの値をリターンします。

注：このファンクションは、rwAttacrwharedRam がコールされた後にのみ使用可能です。

参 照

rwAttacrwharedRam, rwDetacrwharedRam, rwNumberUserShared, rwReadUser, rwWriteUser, rwUserSharedRam も参照してください。

B. 2. 14 rwReadUserLong

構 文

long rwReadUserLong(long LongVar)

引 数

型	引数名	説 明
long	LongVar	読み取り用ロング数

戻り値

指定されたロング数の値

0 : エラー

解 説

rwReadUserLong は、シェアードメモリ領域のユーザーシェアード部中に見つかった、指定されたロングの値をリターンします。

注：このファンクションは、rwAttacrwharedRam がコールされた後にのみ使用可能です。

参 照

rwAttacrwharedRam, rwDetacrwharedRam, rwNumberUserShared, rwReadUser, rwWriteUser, rwUserSharedRam も参照してください。

B. 2. 15 rwReadUserword

構 文

long rwReaduserword(long WordVar)

引 数

型	引数名	説 明
long	WordVar	読み取り用ワード数

戻り値

指定されたワード数の値

0 : エラー

解 説

rwReadUserword は、シェアードメモリ領域のユーザーシェアード部中に見つかった、指定されたワードの値をリターンします。

注：このファンクションは、rwAttacrwharedRam がコールされた後にのみ使用可能です。

参 照

rwAttacrwharedRam, rwDetacrwharedRam, rwNumberUserShared, rwReadUser, rwWriteUser, rwUserSharedRam も参照してください。

B.2.16 rwReadWord

構 文

long rwReadWord(long WordVar)

引 数

型	引数名	説 明
long	WordVar	読み取り用ワード数

戻り値

指定されたワード数の値

0 : エラー

解 説

rwReadWord は、シェアードメモリ領域のユーザーシェアード部中に見つかった、指定されたワードの値をリターンします。

注：このファンクションは、rwAttacrwharedRam がコールされた後にのみ使用可能です。

参 照

rwAttacrwharedRam, rwDetacrwharedRam, rwNumberShared, rwRead, rwWrite, rwUserSharedRam も参照してください。

B. 2. 17 rwSharedRam

構 文

`void* rwSharedRam(void)`

引 数

なし

戻り値

シェアードメモリセグメントへのポインタ

解 説

rwSharedRam は、Windows と共有される最初のメモリセクションへのポインタをリターンします。

参 照

rwUserSharedRam, rwNumberShared, rwRead, rwWrite も参照してください。

B. 2. 18 **rwUserSharedRam**

構 文

void* rwUserSharedRam(long* Size)

引 数

型	引数名	説 明
long*	Size	セグメントのサイズへのポインタ

戻り値

割り当てられたシェアード RAM 領域へのポインタ

解 説

rwUserSharedRam は、割り当てられたメモリの現サイズと同様、シェアード RAM 領域へのポインタを返します。
このメモリブロックは、アプリケーションが動作時はいつでもユーザーに対して使用可能な、一般的な目的のバッファです。

注：このファンクションは、rwAttacrwharedRam がコールされた後にのみ使用可能です。

参 照

rwAttacrwharedRam, rwDetacrwharedRam, rwNumberShared, rwRead, rwWrite も参照してください。

B.2.19 rwWriteBit

構 文

```
void rwWriteBit(long BitVar, long Value)
```

引 数

型	引数名	説 明
long	BitVar	書き込むためのビット数
long	Value	書き込む値

戻り値

なし

解 説

rwWriteBit は、シェアードメモリ領域にある指定されたビット数へ Value の値を書き込みます。

注：このファンクションは、rwAttacrwharedRam がコールされた後にのみ使用可能です。

参 照

rwAttacrwharedRam, rwDetacrwharedRam, rwNumberShared, rwRead, rwWrite, rwUserSharedRam も参照してください。

B. 2. 20 rwWriteByte

構 文

```
void rwWriteByte(long ByteVar, long Value)
```

引 数

型	引数名	説 明
long	ByteVar	書き込むためのバイト数
long	Value	書き込む値

戻り値

なし

解 説

rwWriteByte は、シェアードメモリ領域にある指定されたバイト数へ Value の値を書き込みます。

注：このファンクションは、rwAttacrwharedRam がコールされた後にのみ使用可能です。

参 照

rwAttacrwharedRam, rwDetacrwharedRam, rwNumberShared, rwRead, rwWrite, rwUserSharedRam も参照してください。

B.2.21 rwWriteLong

構 文

```
void rwWriteLong(long LongVar, long Value)
```

引 数

型	引数名	説 明
long	LongVar	書き込むためのロング数
long	Value	書き込む値

戻り値

なし

解 説

rwWriteLong は、シェアードメモリ領域にある指定されたロング数へ Value の値を書き込みます。

注：このファンクションは、rwAttacrwharedRam がコールされた後にのみ使用可能です。

参 照

rwAttacrwharedRam, rwDetacrwharedRam, rwNumberShared, rwRead, rwWrite, rwUserSharedRam も参照してください。

B. 2. 22 rwWriteUserByte

構 文

```
void rwWriteUserByte(long ByteVar, long Value)
```

引 数

型	引数名	説 明
long	ByteVar	書き込むためのバイト数
long	Value	書き込む値

戻り値

なし

解 説

rwWriteUserByte は、シェアードメモリ領域のユーザーシェアード部中の指定されたバイトへ、Value の値を書き込みます。

注：このファンクションは、rwAttacrwharedRam がコールされた後にのみ使用可能です。

参 照

rwAttacrwharedRam, rwDetacrwharedRam, rwNumberUserShared, rwReadUser, rwWriteUser, rwUserSharedRam も参照してください。

B. 2. 23 rwWriteUserLong

構 文

```
void rwWriteUserLong(long LongVar, long Value)
```

引 数

型	引数名	説 明
long	LongVar	書き込むためのロング数
long	Value	書き込む値

戻り値

なし

解 説

rwWriteUserLong は、シェアードメモリ領域のユーザーシェアード部中の指定されたロングへ、Value の値を書き込みます。

注：このファンクションは、rwAttacrwharedRam がコールされた後にのみ使用可能です。

参 照

rwAttacrwharedRam, rwDetacrwharedRam, rwNumberUserShared, rwReadUser, rwWriteUser, rwUserSharedRam も参照してください。

B. 2. 24 rwWriteUserword

構 文

```
void rwWriteUserword(long WordVar, long Value)
```

引 数

型	引数名	説 明
long	WordVar	書き込むためのワード数
long	Value	書き込む値

戻り値

なし

解 説

rwWriteUserword は、シェアードメモリ領域のユーザーシェアード部中の指定されたワードへ、Value の値を書き込みます。

注：このファンクションは、rwAttacrwharedRam がコールされた後にのみ使用可能です。

参 照

rwAttacrwharedRam, rwDetacrwharedRam, rwNumberUserShared, rwReadUser, rwWriteUser, rwUserSharedRam も参照してください。

B.2.25 rwWriteWord

構 文

```
void rwWriteWord(long WordVar, long Value)
```

引 数

型	引数名	説 明
long	WordVar	書き込むためのワード数
long	Value	書き込む値

戻り値

なし

解 説

rwWriteWord は、シェアードメモリ領域中の指定されたワード数へ Value の値を書き込みます。

注：このファンクションは、rwAttacrwharedRam がコールされた後にのみ使用可能です。

参 照

rwAttacrwharedRam, rwDetacrwharedRam, rwNumberShared, rwRead, rwWrite, rwUserSharedRam も参照してください。

B.3 スレッド関連

B.3.1 rwAwait

構 文

```
long rwAwait(long Mytid)
```

引 数

型	引数名	説 明
long	Mytid	rwNameAttach からリターンしたスレッド ID

戻り値

一定のシグナル : 成功
 -1 : 失敗

解 説

rwAwait は、rwSignal の機能によってスレッドによってシグナルが送信されるまで、NT スレッド/プロセスをブロックします。
 NT スレッドは、rwAwait へ渡す有効なスレッド ID を受け取るために rwNameAttach をコールします。
 スレッドは、NT スレッドが rwNameAttach をコールするまで NT スレッドを見つけるために、rwLocateThread を使用します。

参 照

rwSignal も参照してください。

B.3.2 rwKillThread

構 文

```
long rwKillThread(long tid)
```

引 数

型	引数名	説 明
long	tid	強制終了するためのスレッドの TID

戻り値

1 : 成功
0 : 失敗

解 説

rwKillThread は、TID に関係するスレッドを強制終了します。このファンクションを使用する時、任意にスレッドを強制終了することが予想外の副作用を起こす原因を及ぼしますので注意してください。

B.3.3 rwLocateMemory

構 文

```
void* rwLocateMemory(char* Name, unsigned long* Length)
```

引 数

型	引数名	説 明
char*	Name	名前のついたシェアードメモリ領域へのポインタ
unsigned long*	Length	length がリターンする場所へのポインタ

戻り値

メモリへのポインタ : 成功
 NULL (0) : 失敗

解 説

rwLocateMemory は、指定されたシェアードメモリ領域を見つけ、シェアードメモリ領域の長さと領域へのポインタをリターンします。

指定されたシェアードメモリ領域は、スレッドから rwSharedMemory (API を参照下さい) にてクリエートされます。
 そうして NT スレッド/プロセスは、rwLocateMemory をコールすることによりシェアードメモリ領域へのアクセスを可能にします。
 指定したシェアードメモリ領域は、プロセスと NT プロセスがシェアードメモリリソースを介して情報を共有することを可能にするために使用されます。

B.3.4 rwLocateThread

構 文

long rwLocateThread(char* Name)

引 数

型	引数名	説 明
char*	Name	スレッドの名前へのポインタ

戻り値

スレッド ID : 成功
 -1 : 失敗

解 説

rwLocateThread は、NT プロセス/スレッドがスレッドを配置し、そのスレッド ID を獲得できるようにします。NT プロセス/スレッドはスレッド ID を獲得後はスレッドであるかのように、rwSignal を介してそのスレッドへシグナルを送ります。

参 照

rwSignal も参照してください。

B.3.5 rwMaxThreads

構 文

long rwMaxThreads(void)

引 数

なし

戻り値

システムにより許容できるスレッドの最大数

解 説

rwMaxThreads は、システムにより許容できるスレッドの最大数をリターンします。この数は内部にてセットされ、ユーザーにより変更は出来ません。

参 照

rwThreadState, rwThreadPriority, rwReadSignals, rwStackSize, rwSendTid, rwThreadName, rwStatusString, rwStatusHeader も参照してください。

B.3.6 rwNameAttach

構 文

```
long rwNameAttach(char* Name, long Priority)
```

引 数

型	引数名	説 明
char*	Name	NT プロセス用スレッドの明確な名前
long	Priority	NT プロセス用スレッドの明確なプライオリティ

戻り値

NT スレッドのスレッド ID : 成功
 -1 : 失敗

解 説

rwNameAttach は NT プロセスかスレッドがネームヘアタッチし、システムの下のスレッド ID を受け取ることを可能にします。
 このスレッド ID を使用して NT プロセスは、rwSignal と rwAwait の機能によってヘシグナルを送信、あるいはからシグナルを受け取ります。
 NT プロセスはまた、rwAllocSemaphoreEx 及び rwFreeSemaphoreEx 機能によって、セマフォを割り付けフリーにします。
 スレッドは通常行なわれるように rwLocateThread を使用して、NT スレッドを見つけます。

NT プロセスはそれが存在する前に、常に rwNameDetach をコールしなければなりません。

参 照

rwSignal, rwAwait, rwAllocSemaphoreEx, rwFreeSemaphoreEx, rwNameDetach も参照してください。

9.2.1 rwNameDetach

構 文

long rwNameDetach(long tid)

引 数

型	引数名	説 明
long	tid	rwNameAttach によりリターンしたスレッド ID

戻り値

1 : 成功
0 : 失敗

解 説

rwNameDetach は、から NT スレッド/プロセスを分離します。のもとで NT プロセスを表示するスレッドは、終了します。rwNameAttach をコールするすべての NT スレッドやプロセスは、終了する前に rwNameDetach をコールしなければなりません。

参 照

rwNameAttach も参照してください。

B.3.7 rwParentTid

構 文

```
long rwParentTid(long tid)
```

引 数

型	引数名	説 明
long	tid	ペアレントを得るスレッドの TID

戻り値

ペアレント TID : 成功
 -1 : 失敗

解 説

rwParentTidは、tidで指定されたスレッドのペアレントTIDを検索します。
 ペアレントTIDは、送信元プロセスのTIDです。
 スレッド用に rwCreateProcess, rwStartApplication で作成され、
 rwParentTidは -1 をリターンします。
 プロセスはシステム内でペアレントを持ちません。しかし、もしプロセスが
 rwCreateThread を持つスレッドを作成する場合、rwParentTidはスレッドを
 作成したプロセスのTIDを返します。
 もし子スレッドが別のスレッドを作成する場合、新しいスレッドのペアレン
 トは作成するスレッドのペアレントです。
 (すなわちこれは依然として、両方のスレッド用に組織しているプロセスで
 す)。

B.3.8 rwReadSignals

構 文

unsigned long rwReadSignals(long Tid)

引 数

型	引数名	説 明
long	Tid	スレッドの ID

戻り値

与えられたスレッド用に中断しているシグナルの数

解 説

rwReadSignals は、任意のスレッドに中断しているシグナルの数をリターンします。

参 照

rwMaxThread, rwThreadsState, rwThreadPriority, rwStackSize, rwSendTid, rwThreadName, rwStatusString, rwStatusHeader も参照してください。

B.3.9 rwSendTid

構 文

long rwSendTid(long Tid)

引 数

型	引数名	説 明
long	Tid	スレッドの ID

戻り値

提供されたスレッドがセンドブロックであるスレッドのスレッド ID
 -1 : 失敗

解 説

rwSendTid は、提供されたスレッドがセンドブロックであるスレッドのスレッド ID をリターンします。

参 照

rwMaxThread, rwThreadsState, rwThreadPriority, rwReadSignals, rwStackSize, rwThreadName, rwStatusString, rwStatusHeader も参照してください。

9.2.2 rwSignal

構 文

long rwSignal(long Tid)

引 数

型	引数名	説 明
long	Tid	シグナルされるためのスレッド ID

戻り値

シグナルしたスレッドのスレッド ID : 成功
-1 : 失敗

解 説

rwSignal は、非ブロッキングシグナルを NT スレッド/プロセスからスレッドへ送ります。rwSignal へ渡されたスレッド ID は、rwLocateThread をコールすることにより得られ、NT スレッド/プロセスからのネームによるスレッドを配置します。スレッドは、rwReceive をコールすることによりシグナルを待つか、rwReadSignals をコールすることでシグナルをチェックします。

参 照

rwLocateThread も参照してください。

B.3.10 rwStackSize

構 文

long rwStackSize(long Tid)

引 数

型	引数名	説 明
long	Tid	スレッドの ID

戻り値

与えられたスレッドのバイト単位でのスタックサイズ
-1 : 失敗

解 説

rwStackSize は、与えられたスレッドのバイト単位でのスタックサイズを戻します。

参 照

rwMaxThreads, rwThreadState, rwThreadPriority, rwReadSignals, rwSendTid, rwThreadName, rwStatusString, rwStatusHeader も参照してください。

B.3.11 rwStatusHeader

構 文

```
void rwStatusHeader(char* String, long MaxLen)
```

引 数

型	引数名	説 明
char*	String	ステータスヘッダ文字列の送り先のバッファ
long	MaxLen	送り先の文字列へコピーする最大文字数

戻り値

なし

解 説

rwStatusHeader はコピーするためのバイトの最大数として MaxLen を使用しながら、ステータス情報ヘッダテキスト文字列を、与えられた送り先のバッファにコピーします。ヘッダ文字列のデフォルトの長さは 51 バイトです。

参 照

rwMaxThreads, rwThreadState, rwThreadPriority, rwReadSignals, rwStackSize, rwSendTid, rwThreadName, rwStatusString も参照してください。

B.3.12 rwStatusString

構 文

```
long rwStatusString(long Tid, char* String, long MaxLen)
```

引 数

型	引数名	説 明
long	Tid	スレッド ID
char*	String	ステータス情報の送り先のバッファ
long	MaxLen	送り先の文字列の最大の長さ

戻り値

1 : 成功
0 または -1 : 失敗

解 説

rwStatusString はコピーするバイトの最大数として MaxLen を使用しながら、与えられたスレッドに関するステータス情報を含むテキストの文字列を、送り先のバッファにコピーします。ステータス文字列のデフォルトの長さは 51 バイトです。

参 照

rwMaxThreads, rwThreadState, rwThreadPriority,
rwReadSignals, rwStackSize, rwSendTid, rwThreadName, rwStatusHeader
も参照してください。

B.3.13 rwSuspend

構 文

```
int rwSuspend(long Tid)
```

引 数

型	引数名	説 明
long	Tid	スレッド ID

戻り値

TRUE : スレッドが一時中断された場合
FALSE : スレッドが一時中断されなかった場合

解 説

rwSuspend は指定されたスレッドの実行を一時中断します。このスレッドは rwResume が呼び出されるまで中断されたままになります。

参 照

rwResume, rwKillThread も参照してください。

B.3.14 rwThreadName

構 文

```
void rwThreadName(long Tid, char* Name, int MaxLen)
```

引 数

型	引数名	説 明
long	Tid	スレッド ID
char*	Name	スレッドネームの送り先のバッファ
int	MaxLen	スレッドネームの最大の長さ

戻り値

なし

解 説

rwThreadName はコピーするバイトの最大数として MaxLen を使用しながら、与えられたスレッド ID に関連するスレッドのネームを、送り先のバッファにコピーします。
RTOS-WinJ のスレッドネームの最大文字数は 16 文字です。

参 照

rwMaxThreads, rwThreadState, rwThreadPriority, rwReadSignals, rwStackSize, rwSendTid, rwStatusString, rwStatusHeader も参照してください。

B.3.15 rwThreadPriority

構 文

long rwThreadPriority(long Tid)

引 数

型	引数名	説 明
long	Tid	スレッド ID

戻り値

与えられたスレッドの優先度
 -1 : エラー

解 説

rwThreadPriority は与えられたスレッドの優先度を戻します。

参 照

rwMaxThreads, rwThreadState, rwReadSignals, rwStackSize, rwSendTid, rwThreadName, rwStatusString, rwStatusHeader も参照してください。

B.3.16 rwThreadState

構 文

long rwThreadState(long Tid)

引 数

型	引数名	説 明
long	Tid	スレッド ID

戻り値

与えられたスレッドの有効な状態

0	Free	獲得した ID におけるスレッドは無し
1	Ready	いつでもスレッド実行できる状態
2	Receive	スレッドが他のスレッドからメッセージやシグナルを待っている
3	Send	スレッドが他のスレッドにメッセージを送信済みでその返信を待っている
4	Suspend	このスレッドの実行が明らかに一時中断されている
5	Zombie	獲得したスレッドで発生するエラー
6	Wait	スレッドはスリープしている
-1	On error	

解 説

rwThreadState は与えられたスレッドの状態を戻します。

参 照

rwMaxThreads, rwThreadPriority, rwReadSignals, rwStackSize, rwSendTid, rwThreadName, rwStatusStrng, rwStatusHeader も参照してください。

B.4 ブーリーンステータス

B.4.1 rwKernelError

構 文

long rwKernelError(void)

引 数

なし

戻り値

TRUE : RTOS-WinJ システムが何らかのエラーを起こしてしまった場合
FALSE : エラー

解 説

rwKernelError は、RTOS-WinJ 内部のエラーステータスビットの数値を返します。

参 照

rwSystemActive, rwProcessExit, rwProcessorFault も参照してください。

B.4.2 rwProcessExit

構 文

long rwProcessExit(void)

引 数

なし

戻り値

TRUE :
システムが RTOS-WinJ の下で実行中のアプリケーションを判断した場合

解 説

rwProcessExit は、RTOS-WinJ アプリケーションの Exit ステータスビットの
数値を戻します。

参 照

rwSystemActive, rwKernelExit, rwProcessorFault も参照してください。

B.4.3 rwProcessorFault

構 文

long rwProcessorFault(void)

引 数

なし

戻り値

TRUE :
システムがプロセッサの不具合がRTOS-WinJ で生じたと判断する場合

解 説

rwProcessorFault は、RTOS-WinJ のプロセッサの不具合ステータスビットの数値を返します。

参 照

rwSystemActive, rwKernelExit, rwProcessExit も参照してください。

B.4.4 rwSystemActive

構 文

long rwSystemActive(void)

引 数

なし

戻り値

TRUE : RTOS-WinJ システムがアクティブな場合
FALSE : エラー

解 説

rwSystemActive は、RTOS-WinJ システムがアクティブな場合、及び RTOS-WinJ アプリケーションが実行中の場合は TRUE です。

参 照

rwKernelError, rwProcessExit, rwProcessorFault も参照してください。

B.5 スレッド間通信、セマフォ

B.5.1 `rwAllocSemaphoreEx`

構 文

```
long rwAllocSemaphoreEx(long NTtid, long RWtid)
```

引 数

型	引数名	説 明
long	NTtid	rwNameAttach から戻されるスレッド ID
long	RWtid	セマフォスレッド ID

戻り値

```
0          : 成功
-1         : 失敗
```

解 説

`rwAllocSemaphoreEx` は、以前に `rwCreateSemaphore` 関数で作成されたセマフォを割り当てます。

一度割り当てられると、最初の割り付けルーチンが `rwFreeSemaphoreEx` 関数によってセマフォをフリーにするまで、スレッドが同じセマフォを割り付ける場合は NT、あるいは RTOS-WinJ スレッドをブロックします。

このようにして、共有リソースへのアクセスは、`rwAllocSemaphoreEx`/`rwFreeSemaphoreEx` の 2 つの関数にてリソースを保護することによってコントロールします。

セマフォを認識するこのセマフォスレッド ID は、`rwCreateSemaphore` あるいは `rwLocateSemaphore` が戻すスレッド ID です。

NT スレッドは、まず最初に `rwNameAttach` を呼び出して、`rwAllocSemaphoreEx` を呼び出す前に有効な RTOS-WinJ スレッド ID を獲得しなければなりません。

参 照

`rwCreateSemaphore`, `rwDestroySemaphoreEx`, `rwFreeSemaphoreEx`, `rwLocateSemaphore` も参照してください。

B.5.2 rwDestroySemaphoreEx

構 文

```
long rwDestorySemaphoreEx(long NTtid, long RWtid)
```

引 数

型	引数名	説 明
long	NTtid	rwNameAttach から戻されるスレッド ID
long	RWtid	セマフォのスレッド ID

戻り値

```
0          : 成功
-1         : 失敗
```

解 説

rwDestroySemaphoreEx は、以前に rwCreateSemaphore 関数にて作成されたセマフォを破棄します。

セマフォを認識するこのスレッド ID は、rwCreateSemaphore あるいは rwLocateSemaphore が戻すスレッド ID です。

アプリケーションにとって必要性がなくなった時は、セマフォは常に破棄されなければなりません。

もし破壊するスレッドがセマフォを割り当てたのと同じスレッドではない限り、このセマフォはフリーになるまで実際は破棄されません。

NT スレッドは、まず最初に rwNameAttach を呼び出して、rwDestroySemaphoreEx を呼び出す前に、有効な RTOS-WinJ スレッド ID を獲得しなければなりません。

参 照

rwCreateSemaphor, rwAllocSemaphoreEx, rwFreeSemaphoreEx, rwLocateSemaphore も参照してください。

B.5.3 rwFreeSemaphoreEx

構 文

```
long rwFreeSemaphoreEx(long NTtid, long RWtid)
```

引 数

型	引数名	説 明
long	NTtid	rwNameAttach から戻されるスレッド ID
long	RWtid	セマフォのスレッド ID

戻り値

```
0          : 成功
-1         : 失敗
```

解 説

rwFreeSemaphoreEx は、以前に rwAllocSemaphoreEx 関数にて割り当てられたセマフォをフリーにします。

このセマフォをフリーにすることで、ほかの NT や RTOS-WinJ スレッドがこのセマフォを割り当てることができます。

(そして rwAllocSemaphoreEx を呼び出しているスレッドを非ブロック化します)。

このセマフォを認識するスレッド ID は、rwCreateSemaphore あるいは rwLocateSemaphore が戻したスレッド ID のことです。

NT スレッドは、まず最初に rwNameAttach を呼び出して、rwAllocSemaphoreEx を呼び出す前に有効な RTOS-WinJ スレッド ID を獲得する必要があります。

参 照

rwCreateSemaphore, rwDestroySemaphoreEx, rwAllocSemaphoreEx, rwLocateSemaphore も参照してください。

B.6 スレッド間通信、新セマフォ関数

B.6.1 rwCloseSemaphoreEx

構 文

```
long rwCloseSemaphoreEx(long myTid, unsigned long semaphore)
```

引 数

型	引数名	説 明
long	myTid	rwNameAttach が戻した TID
unsigned long	semaphore	rwOpenSemaphoreEx が戻したセマフォ ID

戻り値

```
1          : 成功
0          : 失敗
```

解 説

rwCloseSemaphoreEx は、rwOpenSemaphoreEx で開かれたセマフォを閉じます。他のスレッドがセマフォを開いてない場合、この関数は閉じたあとにセマフォを破棄します。

他のスレッドがセマフォを開いている場合、rwCloseSemaphoreEx のセマフォをカウント処理を減少させます。

rwCloseSemaphoreEx は rwCloseSemaphore と類似しているもので、RTOS-WinJAPI の中に存在します。

Windows コンテキストからのセマフォの同じ種類(タイプ)のものにおいて、この関数は操作されます。従って RTOS-WinJ 及び Windows のスレッドは、このようなタイプのセマフォ使用して同時性を保つことができます。

参 照

rwTrySemaphoreEx, rwOpenSemaphoreEx, rwReleaseSemaphoreEx, rwGetSemaphoreEx も参照してください。

B. 6. 2 rwGetSemaphoreEx

構 文

```
long rwGetSemaphoreEx(long myTid, unsigned long semaphore)
```

引 数

型	引数名	説 明
long	myTid	rwNameAttach が戻した TID
unsigned long	semaphore	rwOpenSemaphoreEx が戻したセマフォ ID

戻り値

1 : 成功
0 : 失敗

解 説

rwGetSemaphoreEx は、パラメータとして伝えられた ID に関連のあるセマフォの所有権を獲得しようとします。
このセマフォをすでに他のスレッドが所有している場合、現在の所有者がこのセマフォをリリースするまでブロックします。
二つ以上のスレッドがセマフォを待っている場合、優先度の一番高いスレッドがまず最初に所有権を要求します。
同じ優先度の複数のスレッドが待機している場合は、もっとも長く待機しているスレッドが、最初に所有権を獲得します。

この呼び出しスレッドは、このセマフォの所有権の必要性がなくなったときに、rwReleaseSemaphoreEx 関数を呼び出します。

rwGetSemaphoreEx は、rwGetSemaphore 関数と類似しているもので、RTOS-WinJAPI に存在します。
Windows コンテキストからの同じタイプのセマフォでこの関数は操作されません。このようにして RTOS-WinJ 及び Windows のスレッドは、このようなセマフォを使用して同時性を保つことができます。

参 照

rwTrySemaphoreEx, rwOpenSemaphoreEx, rwReleaseSemaphoreEx, rwCloseSemaphoreEx も参照してください。

B. 6. 3 rwOpenSemaphoreEx

構 文

```
unsigned long rwOpenSemaphoreEx(long myTid, char* Name)
```

引 数

型	引数名	説 明
long	myTid	rwNameAttach が戻した TID
char*	Name	セマフォのネームを示すポインタ（最大 16 文字）

戻り値

セマフォ ID : 成功
 NULL : 失敗

解 説

rwOpenSemaphoreEx は、パラメータとして渡された名前に関連するセマフォを開いたり、作成しようと試みます。
 この名前は、NULL で終了する長さ最大 16 文字 ASCII 文字列です（NULL 終了バイトを含みません）。セマフォがすでに存在している場合は、この関数はセマフォの使用カウントを増やし、セマフォ関連の ID を戻します。セマフォがまだ存在していない場合は、この関数はセマフォを作成して初期化し、セマフォ関連の ID を戻します。

RwCreateSemaphore にて作成されたセマフォと同時に rwOpenSemaphoreEx を使用しないでください。
 RwCloseSemaphoreEx にて作成もしくは開かれたセマフォを操作する場合は、rwTrySemaphoreEx, rwGetSemaphoreEx, rwReleaseSemaphoreEx, rwOpenSemaphoreEx 関数のみ使用してください。

オーブンスレッドにおいて必要性がなくなったすべてのセマフォ用に rwCloseSemaphoreEx を呼び出してください。スレッドが開かれたセマフォを使用する必要がある限り、ユーザーはセマフォを開いたままにしておきます。スレッドは抜ける前に開いているすべてのセマフォを常に閉じる必要があります。

rwOpenSemaphoreEx は rwOpenSemaphore 関数と類似したもので RTOS-WinJAPI に存在しています。Windows コンテキストからのセマフォの同じタイプにおいてこの関数は操作されます。このようにして RTOS-WinJ および Windows のスレッドは、こういったセマフォを使用して同時性を保つことができます。

参 照

rwTrySemaphoreEx, rwGetSemaphoreEx, rwReleaseSemaphoreEx, rwCloseSemaphoreEx も参照してください。

B. 6. 4 `rwReleaseSemaphoreEx`

構 文

```
long rwReleaseSemaphoreEx(long myTid, unsigned long semaphore)
```

引 数

型	引数名	説 明
long	myTid	rwNameAttach が戻した TID
unsigned long	semaphore	rwOpenSemaphoreEx が戻したセマフォ ID

戻り値

```
1          : 成功
0          : 失敗
```

解 説

`rwReleaseSemaphoreEx` は、`rwGetSemaphoreEx` あるいは `rwTrySemaphoreEx` 関数を使用して以前に与えられたセマフォの所有権を開放します。

`rwGetSemaphoreEx` あるいは `rwTrySemaphoreEx` を使用してセマフォの所有権を獲得するすべてのスレッドは、セマフォによって保護されるリソースを使用する他のスレッドが不必要にブロッキングしないようにさせたら、できるだけすぐにこのセマフォを開放する必要があります。

`rwReleaseSemaphoreEx` は、`rwReleaseSemaphore` 関数と類似しているもので、RTOS-WinJAPI に存在しています。Windows コンテキストからの同じタイプのセマフォにおいてこの関数は操作されます。このようにして、RTOS-WinJ および Windows のスレッドは、こういったセマフォを使用して同時性を保つことができます。

参 照

`rwTrySemaphoreEx`, `rwOpenSemaphoreEx`, `rwGetSemaphoreEx`, `rwCloseSemaphoreEx` も参照してください。

B. 6. 5 rwTrySemaphoreEx

構 文

long rwTrySemaphoreEx(long myTid, unsigned long semaphore)

引 数

型	引数名	説 明
long	myTid	rwNameAttach が戻した TID
unsigned long	semaphore	rwOpenSemaphoreEx が戻したセマフォ ID

戻り値

1 : 成功
0 : 失敗

解 説

rwTrySemaphoreEx は、パラメータとして伝えられた ID に関連するセマフォの所有権を獲得しようとします。
このセマフォをすでに他のスレッドが所有している場合は、この関数は 0 の戻り値ですぐに戻ります。
セマフォが現在所有されていない場合は、この関数は rwGetSemaphore 関数として正確にセマフォの所有権を獲得します。

この呼び出しスレッドは、セマフォの所有権の必要性がなくなったときに、rwReleaseSemaphoreEx 関数を呼び出します。

rwTrySemaphoreEx は rwTrySemaphore と類似しているもので、RTOS-WinJ API に存在しています。Windows からのセマフォの同じタイプにおいて操作されます。このようにして RTOS-WinJ および Windows は、こういったセマフォを使用して同時性を保つことができます。

参 照

rwGetSemaphoreEx, rwOpenSemaphoreEx, rwReleaseSemaphoreEx, rwCloseSemaphoreEx も参照してください。

B.7 デバッグトレースユーティリティ

B.7.1 `rwDumpDebugTrace`

構 文

```
long rwDumpDebugTrace(long ClearBuffer)
```

引 数

型	引数名	説 明
long	ClearBuffer	トレースバッファをクリアするフラグ

戻り値

1 : 成功
0 : 失敗

解 説

`rwDumpDebugTrace` は、トレースバッファをダンプファイルにコピーします。
`rwSetTraceDumpFile` への呼び出しがダンプファイルネームを指定します。
`rwDumpDebugFile` 関数が呼び出されるたびに、ダンプファイルのコンテンツは上書きされます。

このダンプファイルを保存するためには、`rwSetTraceDumpFile` を呼び出し新しいダンプファイル名へ変更するか、`rwDumpDebugTrace` を再び呼び出す前に現在のダンプファイルを異なったファイル名へコピーします。

ダンプファイルはバイナリフォーマットにあることに注意してください。この `rwFormatTraceFile` コールを使用して、このトレースファイルを人間が判読可能な ASCII フォーマットへフォーマットしてください。

ClearBuffer フラグが、`rwDumpDebugTrace` 関数が、ダンプ後にトレースバッファをクリアにします。0 ではない数値は、ダンプが発生した後トレースバッファが空になる必要があることを示します。(`rwTraceDataAvailable` は false を戻します。) ClearBuffer のために 0 が渡される場合、このデータはダンプ後にトレースバッファに残ります。`rwStartTraceMonitoring` への呼び出しは、モニタリングが開始されるときに、自動的にトレースバッファをクリアにします。

rwDumpDebugTrace は、モニタリングがアクティブではないときに唯一呼び出される可能性があります。（例、rwGetTraceState は RW_TRACE_STATE_DORMANT を返します。）RW_TRACE_STATE_DORMANT は、trace.hにおいて次のように定義されます。

```
#define RW_TRACE_STATE_DORMANT 1
```

トレースモニタリングがアクティブである間に、ユーザーがトレースファイルをダンプしようとする場合（例、rwStartTraceMonitoring は呼び出されていて、ストップトリガ、あるいはフルのバッファ、rwStopTracing コールがトレーシングを停止していない）、rwDumpDebugTrace は失敗します。

参 照

rwSetTraceDumpFile, rwFormatTraceFile も参照してください。

B.7.2 rwFormatTraceFile

構 文

```
long rwFormatTraceFile(char* infile, char* outfile)
```

引 数

型	引数名	説 明
char*	infile	バイナリダンプファイルのファイルネーム
char*	outfile	初期化済み ASCII ファイル用ファイルネーム

戻り値

1 : 成功
0 : 失敗

解 説

rwFormatTraceFile は、二進ファイルを人間が判読可能な ASCII にフォーマットします。このフルパスは、それぞれのファイルに提供される必要があります。

infile は現在のファイル(例、以前記録された一つである場合があります。)になる必要がない rwDumpDebugTrace によってアウトプットされるファイルのことです。

rwFormatTraceFile は、機能させるために RTOS-WinJ システムがアクティブになることを要求しません。

フォーマットされた出力ファイルを生成するために、この関数は適切なインプットファイルが存在するように要求します。

参 照

rwDumpDebugTrace, rwSetTraceDumpFile も参照してください。

B.7.3 rwFormatTraceFileCSV

構 文

```
long rwFormatTraceFileCSV(char* Dumpfile, char* outfile)
```

引 数

型	引数名	説 明
char*	Dumpfile	バイナリダンプファイルのファイルネーム
char*	outfile	初期化済み ASCII CSV ファイル用ファイルネーム

戻り値

1 : 成功
0 : 失敗

解 説

rwFormatTraceFileCSV は、バイナリファイル、ダンプファイル、ASCII、人間／コンピュータが判読可能な、コンマで区分された書式を Outfile にフォーマットします。

このフルパスは、それぞれのファイル用に提供されます。 このダンプファイルは rwDumpDebugTrace がアウトプットしたファイルのことです。

このダンプファイルは現在のファイルになる必要がありません。
(例、Dumpfile は以前に記録されたファイルになる場合があります)。
rwFormatTraceFileCSV は、RTOS-WinJ システムがアクティブになるように要求しませんが、初期化済みのアウトプットファイルが発生させるために、適切なインプットファイルが存在するように要求します。
この CSV ファイルのフォーマットは次のようになります。
(示されるように、このイベントシンボリック定数はファイルの中には記録されません。異なったイベントレコードのタイプのフォーマットを区別するためにここに含まれています。)

Number Of Records, Time scale
 RW_TRACE_EVENT_ENTER_INTERRUPT
 RW_TRACE_EVENT_LEAVE_INTERRUPT
 Description, Interruptm, Timestamp, Active Tid, N/A
 RW_TRACE_EVENT_ENTER_FUNCTION
 Description, Function Name, Timestamp, Calling Tid, N/A
 RW_TRACE_EVENT_LEAVE_FUNCTION
 Description, Function Name, TimeStamp, Calling Tid, ReturnValue
 RW_TRACE_EVENT_ENTER_SYSTEM_TIMER :
 RW_TRACE_EVENT_LEAVE_SYSTEM_TIMER :
 Description, N/A, Timestamp, Calling Tid, N/A
 RW_TRACE_EVENT_CALL_USER_TIMER
 RW_TRACE_EVENT_RETURN_USER_TIMER
 RW_TRACE_EVENT_SWITCH_TO_NT
 RW_TRACE_EVENT_SWITCH_TO_RW
 Description, N/A, Timestamp, Active Tid, N/A
 RW_TRACE_EVENT_NT_FAULT
 RW_TRACE_EVENT_ENTER_PROCESSOR_FAULT
 RW_TRACE_EVENT_LEAVE_PROCESSOR_FAUL
 Description, Fault Number, Timestamp, Faulting Tid, Fault Address
 RW_TRACE_EVENT_FAULT_LOCATION
 Description, Error Code, Timestamp, Active Tid, Code Address
 RW_TRACE_EVENT_USER_EVENT
 Description, User Code, imestamp, Calling Tid, User Parameter

二つのイベント記録及びマイクロセカンドタイムスタンプを持つトレース CSV ファイルは次のように表されます。

2, Microseconds
 Enter Kernel Call, rwSend, 0, 5, N/A
 Enter Interrupt, 0, 3, 5, N/A

参 照

rwDumpDebugTrace, rwFormatTraceFile も参照してください。

B.7.4 rwGetStartupTracing

構 文

long rwGetStartupTracing(long* Trace)

引 数

型	引数名	説 明
long*	Trace	フラグ用保管位置

戻り値

1 : 成功
0 : 失敗

解 説

rwGetStartupTracing は、トレース内の自動立ち上げトレースモニタリングフラグの現在数値を検索します。自動立ち上げトレースモニタリングの詳細に関しては rwSetStartupTracing を参照してください。

参 照

rwSetStartupTracing も参照してください。

B.7.5 rwGetTraceBufferMode

構 文

```
long rwGetTraceBufferMode(long* Mode)
```

引 数

型	引数名	説 明
long*	Mode	現在バッファモードの位置

戻り値

```
1          : 成功
0          : 失敗
```

解 説

rwGetTraceBufferMode は、使用される現在のモードの位置を、Mode 内のバッファイベントに戻します。このモードは trace.h にて次のように定義されます。

```
#define RW_TRACE_BUFFER_ONESHOT 1
#define RW_TRACE_BUFFER_CIRCULAR 2
```

ONESHOT バッファは、バッファが一旦フルになるとイベントのモニタリングを停止します。

イベントにおいては、そのフルバッファの示すものは、他のストップトリガのように機能します。サーキュラバッファは、バッファがフルになった後に、イベントのモニタリングを続けます。

もっとも古いイベントは、循環式で単純に上書きされます。

参 照

rwSetTraceBufferMode も参照してください。

B.7.6 rwGetTraceBufferSize

構 文

```
long rwGetTraceBufferSize(long* BufferSize)
```

引 数

型	引数名	説 明
long*	BufferSize	現在のバッファサイズ

戻り値

1 : 成功
0 : 失敗

解 説

rwGetTraceBufferSize は、BufferSize におけるエントリ（記録）の数の中のトレースバッファのサイズを戻します。
アクティブなトレーシングは、システム内でイベントごとに一つのレコードを使用します。

参 照

rwSetTraceBufferSize も参照してください。

B.7.7 rwGetTraceDumpFile

構 文

long rwGetTraceDumpFile(char* Filename, long len)

引 数

型	引数名	説 明
char*	Filename	トレースダンプファイルネーム
long	len	バッファファイルネームの長さ

戻り値

1 : 成功
0 : 失敗

解 説

rwGetTraceDumpFile は、現在のトレースダンプファイルのネームを検索します。

トレースダンプファイルネームに関する更に詳しい情報はrwSetTraceDumpFileを参照してください。

参 照

rwDumpDebugTrace, rwSetTraceDumpFile も参照してください。

B.7.8 rwGetTraceState

構 文

```
long rwGetTraceState(long* State)
```

引 数

型	引数名	説 明
long*	State	現在のステートの位置

戻り値

```
1          : 成功
0          : 失敗
```

解 説

rwGetTraceState は、トレースモニタリングの現在ステートの位置を戻します。さまざまな状態は、trace.h に以下のように定義されています。

```
#define RW_TRACE_STATE_NOTREADY 0
//引き続きシステム立ち上げ中、トレース使用は不可

#define RW_TRACE_STATE_DORMANT 1
//トレーシングはアクティブではなく、設定される可能性あり

#define RW_TRACE_STATE_MONITORING 2
//rwStartTraceMonitoring はスタートトリガを検索中に呼ばれる

#define RW_TRACE_STATE_TRIGGERED 4
//記録、スタートトリガが発生している

#define RW_TRACE_STATE_PAUSED 8
//内部ステート、モニタリングは一時的に休止
```

すべての設定関数（例、rwSetTraceTriggers, rwSetTraceBufferSize）は、トレースステートが RW_TRACE_STATE_DORMANT である間に呼び出されなければなりません。

B.7.9 rwGetTraceTriggers

構 文

```
long rwGetTraceTriggers(unsigned long* StartTrigger,
                        unsigned long* StopTrigger)
```

引 数

型	引数名	説 明
unsigned long*	StartTrigger	スタートトリガを置く位置
unsigned long*	StopTrigger	ストップトリガを置く位置

戻り値

1 : 成功
0 : 失敗

解 説

rwGetTraceTriggers は、イベントの記録を開始（および停止）させるトリガの位置を StartTrigger および StopTrigger のトレースバッファに戻します。トレーストリガに関する更に詳しい情報は rwSetTraceTriggers を参照してください。

参 照

rwSetTraceTriggers も参照してください。

B. 7. 10 rwSetStartupTracing

構 文

```
long rwSetStartupTracing(long Trace)
```

引 数

型	引数名	説 明
long	Trace	使用可能なトレーシングスタートアップを示すフラグ

戻り値

1 : 成功
0 : 失敗

解 説

rwSetStartupTracing は、システム起動時のトレースモニタリングを使用可能にするフラグを設定あるいはクリアにします。 Trace の数値が 1 の場合、rwStartTraceMonitoring は、システムが rwStartApplication が起動するときに自動的に呼び出されます。

必ず、必要なすべての設定関数を呼び出してから、システムが起動してから最初の数秒間のうちに発生するプロセッサの不具合をデバッグしようとする際に rwStartApplication を呼び出すようにしてください。

(例、rwSetTraceDumpFile)

rwSetStartupTracing は、Trace と共に呼び出してゼロに設定すると、自動トレースモニタリングは使用不可となり、トレースモニタリングを開始するために rwStartTraceMonitoring を呼び出す必要があります。

参 照

rwStartTraceMonitoring も参照してください。

B.7.11 `rwSetTraceBufferMode`

構 文

```
long rwSetTraceBufferMode(long Mode)
```

引 数

型	引数名	説 明
long	Mode	バッファリングモードを設定

戻り値

```
1          : 成功
0          : 失敗
```

解 説

`rwSetTraceBufferMode` は、Mode を使用してバッファリングイベント用モードを設定します。このモードは `trace.h` において次のように定義されます。

```
#define RW_TRACE_BUFFER_ONESHOT 1
#define RW_TRACE_BUFFER_CIRCULAR 2
```

ONESHOT バッファは、一旦バッファがフルになると、モニタリングイベントを停止します。
 イベントにおいて、このフルのバッファは他の（更なる）ストップトリガとして作動します。最も古いイベントが循環式で単純に上書きされます。
`rwSetTraceBufferMode` は、モニタリングがアクティブではないとき（例、`rwGetTraceState` は `RW_TRACE_STATE_DORMANT` を戻します）、あるいは RTOS-WinJ システムがダウンしている時に、唯一呼び出される可能性があるものです。

参 照

`rwSetTraceBufferSize` も参照してください。

B. 7. 12 rwSetTraceBufferSize

構 文

long rwSetTraceBufferSize(long BufferSize)

引 数

型	引数名	説 明
long	BufferSize	エントリーの数の中のトレースバッファのサイズ

戻り値

1 : 成功
0 : 失敗

解 説

rwSetTraceBufferSize は、BufferSize を使用してエントリーの数の中のトレースバッファのサイズを設定します。アクティブなトレーシングはシステム内のイベントごとに一つの記録を使用します。rwSetTraceBufferSize は、モニタリングがアクティブではないとき（例、rwGetTraceState は RW_TRACE_STATE_DORMANT を戻します）、あるいは RTOS-WinJ システムがダウンしている時に、唯一呼び出される可能性のあるものです。

参 照

rwSetTraceBufferMode も参照してください。

B. 7. 13 `rwSetTraceDumpFile`

構 文

```
long rwSetTraceDumpFile(char* filename)
```

引 数

型	引数名	説 明
char*	filename	新しいトレースダンプファイルネーム

戻り値

```
1          : 成功
0          : 失敗
```

解 説

`rwSetTraceDumpFile` は、ファイルネームを使用してデバッグ トレース ダンプファイル用のファイルネームを設定します。このトレースバッファは、`rwDumpDebugTrace` が呼び出されるときに、`rwSetTraceDumpFile` が指定したファイルにダンプされます。フルパスのネームは必ずダンプファイルネームとして指定されなければなりません。

参 照

`rwDumpDebugTrace`, `rwGetTraceDumpFile` も参照してください。

B. 7. 14 rwSetTraceTriggers

構 文

```
long _cdecl rwSetTraceTriggers(unsigned long StartTrigger,
                               unsigned long StopTrigger)
```

引 数

型	引数名	説 明
unsigned long	StartTrigger	モニタリングイベントを開始させるトリガ
unsigned long	StopTrigger	モニタリングイベントを停止させるトリガ

戻り値

```
1          : 成功
0          : 失敗
```

解 説

rwSetTraceTriggers は、トレースバッファへのイベントの記録を開始（および停止）させるトリガを設定します。二つ以上のトリガが、スタートおよびストップトリガ両方用に設定される場合があります。この関数は複数のトリガ、あるいはトリガ識別子を一緒にシングル 32 ビット値に設定します。トリガの中には両方のトリガにおいて有効ではないものもあります。このトリガ識別子は trace.h において次のように定義されます。

```
#define RW_TRACE_EVENT_ENTER_INTERRUPT 0x0001
//開始または停止
#define RW_TRACE_EVENT_LEAVE_INTERRUPT 0x0002
//開始または停止
#define RW_TRACE_EVENT_ENTER_FUNCTION 0x0004
//開始または停止
#define RW_TRACE_EVENT_LEAVE_FUNCTION 0x0008
//開始または停止
#define RW_TRACE_EVENT_ENTER_SYSTEM_TIMER 0x0010
//開始または停止
#define RW_TRACE_EVENT_LEAVE_SYSTEM_TIMER 0x0020
//開始または停止
#define RW_TRACE_EVENT_CALL_USER_TIMER 0x0040
//開始または停止
#define RW_TRACE_EVENT_RETURN_USER_TIMER 0x0080
//開始または停止
#define RW_TRACE_EVENT_ENTER_PROCESSOR_FAULT 0x0100
//停止のみ
#define RW_TRACE_EVENT_LEAVE_PROCESSOR_FAULT 0x0200
//停止のみ
```

```
#define RW_TRACE_EVENT_USER_EVENT 0x0400
//開始または停止
#define RW_TRACE_EVENT_START_MONITORING 0x0800
//開始のみ
#define RW_TRACE_EVENT_FAULT_LOCATION 0x1000
//使用しないでください、内部で使用
#define RW_TRACE_EVENT_SWITCH_TO_NT 0x2000
//開始または停止
#define RW_TRACE_EVENT_SWITCH_TO_RW 0x4000
//開始または停止
#define RW_TRACE_EVENT_NT_FAULT 0x8000
//使用しないでください、内部で使用
```

このコメントはトリガが有効である場所を識別します。 継続する
モニタリング用の良好なスタートトリガは、識別子
RW_TRACE_EVENT_START_MONITORING です。

この識別子は、rwStartTraceMonitoring 関数コールが作成されるとできる
だけすぐに、イベントの記録をトリガします。

この識別子 RW_TRACE_EVENT_SWITCH_TO_RW は、有用なスタートトリガでもあり
ます。この識別子は、次のコンテキストが Windows から RTOS-WinJ へスイッ
チバックすると、イベントの記録を開始します。

既知の外部 RTOS-WinJ ハードウェアインタラプトが予想される場合は、
RW_TRACE_EVENT_ENTER_INTERRUPT は良好なスタートトリガとなる条件を有
しています。（あるいはユーザーが捕らえようとしているものによっては、
ストップトリガになる場合もあります。）ストップトリガ用に 0 が指定され
る場合、トレーシングは rwStopTracing 関数コールをしようして停止された
ままであることもあります。

rwSetTraceTriggers は、モニタリングがアクティブではないとき（例、
rwGetTraceState が RW_TRACE_STATE_DORMANT を戻す）、あるいは RTOS-WinJ
システムがダウンしているときは、唯一呼び出される可能性があるものです。

参 照

rwGetTraceTriggers, rwStartTraceMonitoring, rwStopTracing も参照して
ください。

B.7.15 rwStartTraceMonitoring

構 文

```
long rwStartTraceMonitoring(void)
```

引 数

なし

戻り値

1	:	成功
0	:	失敗

解 説

rwStartTraceMonitoring は、トリガ／イベント用のモニタリングを開始します。スタートトリガイイベントが一旦発生すると、ほかの全てのイベントがトレースバッファに記録されます。 ストップトリガが一旦発生すると（あるいは one-shot トレースバッファでトレースバッファがフルになると）、トレースモニタリング（あるいはイベントの記録）が停止します。ストップトリガあるいはフルのバッファは、実際には、自動 rwStopTracing 関数です。RwStartTraceMonitoring が呼び出されるまで検出されるイベントやトリガはない、ということに注意してください。このストップトリガが一旦発生すると、このバッファは one-shot バッファでフルになり、あるいは rwStopTracing は呼び出され、その後 rwStartTraceMonitoring が再び呼び出されるまで検出されたり記録されるイベントやトリガは他にありません。

rwStartTraceMonitoring への呼び出しは、モニタリングが開始されるときに自動的にトレースバッファをクリアします。

注意: rwSetTraceBufferMode, rwSetTraceBufferSize, rwSetTraceTriggers を呼び出してから、rwStartTraceMonitoring への呼び出しとともにモニタリングを開始してください。

参 照

rwSetTraceBufferMode, rwSetTraceBufferSize, rwSetTraceTriggers, rwStopTracing も参照してください。

B.7.16 rwStopTracing

構 文

```
long rwStopTracing(void)
```

引 数

なし

戻り値

1	:	成功
0	:	失敗

解 説

rwStopTracing はトリガ／イベント用モニタリングを停止し、スタートトリガがすでに発生している場合は記録イベントを停止します。ストップトリガは、実際にはストップトリガイイベントが発生するときにシステムが自動的に開始する rwStopTracing への呼び出しのことです。

注意：ストップトリガ用に 0 が指定されている場合、モニタリングを停止する唯一の方法は、rwStopTracing コールを使用することです。

参 照

rwStartTraceMonitoring も参照してください。

B. 8 その他

B. 8.1 rwAttachDebugWindow

構 文

long rwAttachDebugWindow (HWND hwnd)

引 数

型	引数名	説 明
HWND	hwnd	メッセージ送信先のリストボックスウィンドウハンドル

戻り値

1 : 成功
0 : 失敗

解 説

rwAttachDebugWindow は、指定ウィンドウへの RTOS-WinJ システム内でプリントされるすべてのデバッグステートメントを送信するように、RTOS-WinJ システムに知らせます。

(RTOS-WinJAPI の rwDebug と rwPrint を参照してください。) このウィンドウハンドルは、リストボックスへのハンドルとなります。

rwStartApplication が自動的に rwAttachDebugWindow コールを実行するので、ユーザーのフロントエンド内で rwStartApplication が使用される場合、i\It はこの関数を呼び出す必要はありません。この関数は、フロントエンド Windows アプリケーションによって使用されますが、このアプリケーションは RTOS-WinJ システムがアクティブである間に開始あるいは停止される場合があります。

このプログラムは、開始したときに rwSystemActive を呼び出します。

rwSystemActive が、RTOS-WinJ システムはすでに実行中であることを示す場合は、Windows プログラムは rwAttachDebugWindow を呼び出して、実行中のシステムからデバッグメッセージを検索することもあります。 このフロントエンドプログラムは、抜け出るときに rwDetachDebugWindow を呼び出します。

参 照

rwDetachDebugWindow も参照してください。

B. 8. 2 rwClearDebug

構 文

void rwClearDebug(void)

引 数

なし

戻り値

なし

解 説

rwClearDebug は、RTOS-WinJ RunTime アプリケーションのデバッグリストをクリアにします。

参 照

なし

B. 8. 3 `rwDetachDebugWindow`

構 文

```
long rwDetachDebugWondow (HWND hwnd)
```

引 数

型	引数名	説 明
HWND	hwnd	RW から分離するリストボックスウィンドウハンドル

戻り値

```
1          : 成功
0          : 失敗
```

解 説

`rwDetachDebugWindow` は、指定ウィンドウへの RTOS-WinJ (RTOS-WinJAPI の `rwDebug` 及び `rwPrint` を参照してください) システム内でプリントされるデバッグステートメントの送信を停止することを、RTOS-WinJ システムに知らせます。このウィンドウハンドルは、`rwAttachDebugWindow` あるいは `rwStartApplication` が以前与えたハンドルとなります。RTOS-WinJ の 4.3 バージョンを始め、二つ以上のデバッグメッセージを検索する場合があることに注意してください。

参 照

`rwAttachDebugWindow` も参照してください。

B. 8. 4 rwGetSystemTime

構 文

unsigned long rwGetSystemTime(void)

引 数

なし

戻り値

システムが起動してからの時間を表す現在のミリセカンドタイムスタンプ
カウンタ

解 説

rwGetSystemTime は、現在の RTOS-WinJ システムタイムを戻しますが、これは RTOS-WinJ システムが起動してからのミリセカンドの数のことであり、通常は相対的な時間を測定するために使用します。

参 照

なし

お問い合わせ先

info@hybridj.co.jp

TEL : 048-959-9487