OpenRTM-aist-0.2.0
デベロッパーズガイド

National Institute of Advanced Industrial Science and Technology
AIST

独立行政法人 産業技術総合研究所知能システム研究部門・タスクインテリジェンス研究グループ

OpenRTM プロジェクトチーム

RT
MIDDLEWARE

目次

第1章 RT ミドルウエアとは **3**

1.1 RT ミドルウエアの目的 3

1.2 RT ミドルウエアの現状と課題 3

1.3 RT ミドルウエアの特長 4

1.4 RT ミドルウエア・RT コンポーネント 5

1.5 RT コンポーネント 6

1.6 RT コンポーネント、マネージャ、ネームサービス 8

第2章 RT コンポーネント入門 **9**

2.1 OpenRTM-aist のインストール 9

2.2 UNIX 系システムでのビルドとインストール 9

2.3 RTCLink による動作確認 18

2.4 Windows 系システムでのビルドとインストール 21

第3章 RT コンポーネントプログラミング **23**

3.1 RT コンポーネント開発の流れ 23

3.2 コンポーネントの作成 27

3.3 Hello RT World サンプル 27

3.4 `rtc-template` を使用したコンポーネントの作成 39

第4章 OpenRTM-aist ツール **55**

4.1 RTCLink 55

4.2 `rtc-template` 69

4.3 `rtm-config` 72

4.4 `rtm-naming` 75

4.5 `rtcd` 75

第5章 RTM Specification **77**

5.1 OpenRTM-aist と RTM Specification 77

5.2 RTM の標準化 79

5.3 RTM Specification Ver.0.1 81
5.4 OpenRTM-aist インターフェース定義 90
5.5 OpenRTM-aist の拡張オペレーションと標準化 98

第 6 章 OpenRTM クラスリファレンス 99

6.1 クラス RTM::RtcBase 99
6.2 構造体 RTM::RtcBase::ComponentStateMtx 128
6.3 クラス RTM::RtcBase::eq_name 130
6.4 構造体 RTM::RtcBase::InPorts 131
6.5 構造体 RTM::RtcBase::OutPorts 132
6.6 クラス RTM::RtcBase::ThreadState 133
6.7 クラス RTM::InPortBase 134
6.8 クラス テンプレート RTM::InPortAny< T > 137
6.9 クラス RTM::OutPortBase 142
6.10 クラス テンプレート RTM::OutPortAny< T > 146
6.11 クラス RTM::RtcManager 152
6.12 クラス RTM::RtcConfig 161
6.13 クラス RTM::RtcNaming 166

第 7 章 OpenRTM IDL リファレンス 175

7.1 インタフェース RTM::RTComponent 175
7.2 例外 RTM::RTComponent::IllegalTransition 182
7.3 例外 RTM::RTComponent::NoSuchName 183
7.4 例外 RTM::InPort::Disconnected 184
7.5 構造体 RTM::NamedValue 185
7.6 インタフェース RTM::OutPort 186
7.7 構造体 RTM::PortProfile 188
7.8 構造体 RTM::SubscriberProfile 189

第 8 章 OpenRTM 拡張 IDL リファレンス 191

8.1 インタフェース RTM::RTCBase 191
8.2 インタフェース RTM::RTCManager 206
8.3 構造体 RTM::Time 209
8.4 構造体 RTM::TimedBoolean 210
8.5 構造体 RTM::TimedBooleanSeq 211
8.6 構造体 RTM::TimedChar 212

8.7 構造体 RTM::TimedCharSeq 213
8.8 構造体 RTM::TimedDouble 214
8.9 構造体 RTM::TimedDoubleSeq 215
8.10 構造体 RTM::TimedFloat 216
8.11 構造体 RTM::TimedFloatSeq 217
8.12 構造体 RTM::TimedLong 218
8.13 構造体 RTM::TimedLongSeq 219
8.14 構造体 RTM::TimedOctet 220
8.15 構造体 RTM::TimedOctetSeq 221
8.16 構造体 RTM::TimedShort 222
8.17 構造体 RTM::TimedShortSeq 223
8.18 構造体 RTM::TimedState 224
8.19 構造体 RTM::TimedString 225
8.20 構造体 RTM::TimedStringSeq 226
8.21 構造体 RTM::TimedULong 227
8.22 構造体 RTM::TimedULongSeq 228
8.23 構造体 RTM::TimedUShort 229
8.24 構造体 RTM::TimedUShortSeq 230

付 録 A 依存パッケージ類のビルドについて 231

A.1 ACE のビルド 231
A.2 boost のビルド 234
A.3 omniORB のビルド 237

本書の構成

本書は OpenRTM-aist の公式解説書です。OpenRTM-aist は独立行政法人新エネルギー・産業技術総合開発機構 (NEDO) の 21 世紀ロボットチャレンジプログラム「ロボット機能発現のために必要な要素技術開発」において研究開発され、標準化作業が進められている RTM Specification のインターフェース仕様に準拠した実装で、独立行政法人産業技術総合研究所により開発されフリーで提供されています。OpenRTM-aist を使用することにより、ユーザは分散オブジェクト技術を利用したネットワーク・ロボットシステムを簡単に構築することが出来ます。

OpenRTM-aist にはロボットシステム構築に特化したミドルウエアです。しかしながら、現在開発されて間もないために、必要な機能が全て実装されているわけではありませんし、バグも多く含んでいると思われます。今後、ユーザの皆様からのフィードバックを得てより使いやすいミドルウエアに成長させたいと考えています。

第 1 章 RT ミドルウエアとは

RT ミドルウエアの概要を説明します。RT ミドルウエアの研究開発に至った背景や現在の課題などを示します。RT ミドルウエアとはどういうものなのか、そのコンセプト、特徴、基本的なアーキテクチャについて紹介します。

第 2 章 RT コンポーネント入門

RT ミドルウエアの実装のひとつである、OpenRTM-aist のインストール方法、基本的な使い方について説明します。OpenRTM-aist のインストールに必要なパッケージ類のインストール方法については、付録の「依存パッケージのビルドについて」もあわせてご参照ください。

第 3 章 RT コンポーネントプラグラミング

OpenRTM-aist におけるコンポーネントの形態や実行方法について紹介します。OpenRTM-aist を使用してコンポーネントを作成する方法について説明します。

第 4 章 OpenRTM-aist ツール

RT コンポーネントを視覚的に操作するためのツール RTCLink の基本的な使い方や、そのほか OpenRTM-aist に付属しているツール群について説明します。

第 5 章 RTM Specification

OpenrRTM-aist のインターフェース仕様である RTM Specification について説明します。

第 6 章 OpenRTM クラスリファレンス

OpenRTM-aist のクラスリファレンスです。

第 7 章 OpenRTM IDL リファレンス

RTM Specification に基づいた IDL インターフェース定義のリファレンスです。

第 8 章 OpenRTM 拡張 IDL リファレンス

RTM Specification から派生した、OpenRTM-aist 独自の IDL インターフェース定義のリファレンスです。

付録 依存パッケージのビルドについて

OpenRTM-aist をコンパイルするために必要なパッケージ類のビルドの仕方について説明します。

第1章　RTミドルウエアとは

1.1　RTミドルウエアの目的

インターネットの広がりとともに、ロボットやロボットシステムをネットワーク化しネットワーク上のリソースを活用して、更なる知能化を目指す研究・開発が盛んに行われています。しかしながら、これらのシステムの開発には通常膨大な開発技術者の投入及び、長い開発期間が必要で、ロボット製品として市場に提供できるだけのレベル、機能、価格のものはいまだほとんど世に出ていないのが現状です。

RTミドルウエアは、様々なロボット要素（RTコンポーネント）を通信ネットワークを介して自由に組み合わせることで、多様なネットワークロボットシステムの構築を可能にする、ネットワーク分散コンポーネント化技術による共通プラットフォームを確立することを目指しています。

ここでいうロボットシステムとは、必ずしも移動ロボットやヒューマノイドロボットといった単体のロボットのみを想定している訳ではなく、「ロボット技術を活用した、実世界に働きかける機能を持つネットワーク化された知能化システム」の総体としてのロボットシステムを指しています。例えばセンサ、アクチュエータを生活空間の中に分散配置させ、ネットワークを介して協調することにより生活支援や介護を実現するといった、一見ロボットには見えないがロボット的な技術を利用したシステムを広く包含しています。実際、センシング技術とセンサから得られる信号の処理、及びアクチュエータ等による現実世界への働きかけのフィードバックにより相互作用を行うシステムはロボット技術といってよいでしょう。日本ロボット工業会が主体となり、こうしたロボット技術の総称をRT(RobotTechnology)と呼ぶことが提言されています。

RTミドルウエアはこうした技術(RT)の共通プラットフォームを整備し、ロボット研究・開発の効率を高め、RTの適用範囲を広げ、さらには新たな市場を拓くことを目指して現在も開発が続けられています。ロボットシステムのソフトウェア開発においては、通常のソフトウェア開発と比べて、ロボットシステム特有の問題のためにソフトウェアの再利用性が低く開発効率が悪いといった問題が指摘されています。こうした背景から、ロボット技術要素をソフトウェアレベルでモジュール化し、その再利用性を高めるミドルウェアを研究・開発し多くのユーザに使ってもらい、さらには開発に参加してもらうことで、ロボットシステム開発に資する共通プラットフォームを提供することがRTミドルウエアの目的なのです。

1.2　RTミドルウエアの現状と課題

OpenrRTM-aistの開発は主に、我々、独立行政法人産業技術総合研究所・知能システム研究部門・タスクインテリジェンス研究グループにより行われています。基は、平成14年度から平成16年度まで、独立行政法人新エネルギー・産業技術統合開発機構(NEDO)の21世紀ロボットチャレンジプログラム「ロボット機能発現のために必要な要素技術開発」においてロボット

用分散ミドルウェアの研究開発プロジェクトによりスタートしました。この研究プロジェクトにおいて、分散ミドルウェアのインターフェースレベルでの仕様が策定され、プロトタイプ実装が完成しました。本書は、そのプロトタイプ実装 OpenRTM-aist-0.2.0 の使用法等について解説するための本です。

上記プロジェクトでは、RT ミドルウェアについてのコンセプトの議論及び、大枠でのインタフェースとこれに基づいたプロトタイプ実装が得られました。しかしながら、これはあくまでサンプルの仕様および実装なので、実システムでの使用に耐えうるレベルにまではいたっていないのが現状です。今後、多くのユーザからのフィードバックを基に実用的で詳細な仕様を策定し、リファレンスとなる実装を開発する必要があります。OpenRTM-aist-0.2.0 はその足がかりとなる実装であり、今後多くの改良・改善が必要なレベルのソフトウエアであることをご承知おきください。

1.3 RT ミドルウエアの特長

RT ミドルウエアの特長を以下に挙げます。

RT システムに特有な機能をサポートしている

従来のネットワークロボット、情報家電や情報化ハウスなど IT 分野を中心として類似のミドルウェアの提案がいくつかなされています。これらは、

- 個々の機器にコマンドを送る
- ストリーミングを行う

などが主体であり、各機器を個別にネットワーク越しに使うことを中心に考えられており機器間の密な連携までは考慮されていません。RT ミドルウエアでは、コマンドフローとデータフローを明示的に分離し機器間の密な連携を意識した設計となっています。ネットワーク上のコンポーネントは互いに他のコンポーネントをあたかも自分の一部の機能であるかのように使用しタスクを遂行することができます。また、CORBA、DCOM といった汎用的な分散オブジェクトミドルウェアも確立されていますが、RT ミドルウェアとそれらとの一番大きな違いは、コンポーネント自身がアクティブに動作すること、上記コンポーネント間の密な連携など、RT システムに固有な機能をサポートしていることです。

高いスケーラビリティを有したコンポーネント化が可能

旧来の単体のロボット単位におけるコンポーネント化では、それぞれのコンポーネントを組み合わせて新しいロボットシステムを構築することは困難でした。RT 関連技術を機能要素単位でモジュール化し部品化することにより、新しいロボットシステムの構築のみならず、ロボット以外の分野へも RT の活用が開かれます。さらに、RT ミドルウェアでは、同一のフレームワークで、装置レベルまでも広くカバーすることが可能です。この高いスケーラビリティにより、コンポーネント化の粒度を、RT を部品として提供しようとしているもの及びそれを利用しようとしているものが必要に応じて自由に選ぶことができ、粒度の異なるコンポーネントを自在に組み合わせるなどの応用面においても、幅広い選択が可能となります。

プラットフォーム、ネットワーク非依存

特定の計算機や OS、特定のネットワークに限定されては、自由な粒度でモジュール化を行うことは困難です。RT ミドルウェアの考え方は、特定のアーキテクチャに依存しないものですので、各 CPU や OS、ネットワークインタフェース (デバイス) に適用可能です。RTM では基本的に CORBA が使用可能なプラットフォームであれば、OS、言語を問わず実装することが出来ます。

既存のソフトウェア資産の再利用が可能である

RT ミドルウエアではソフトウエア資源の再利用が容易になり、導入の敷居が低くなるようにフレームワークやツール群などを用意しています。また、一旦完成したコンポーネントは、開発者以外にも再利用可能であり、インテグレータのアイデア次第で多種多様なシステムを実現できるでしょう。

国際標準に基づいた仕様策定

RT ミドルウェアは、ネットワーク通信のためのミドルウェアとして OMG (Object Management Group) の策定した CORBA を採用しています。また、RT ミドルウェア自体は、OMG の SDO モデルへのマッピング作業が行われており、標準化を意識した仕様策定を行っています。この標準化作業の過程で、ロボットの分散モジュール化およびその仕様の標準化は OMG で重要視され始め、MDA(Model Driven Architecture) に基づくロボットソフトウェア開発のための標準化作業も始まろうとしています。

1.4 RT ミドルウエア・RT コンポーネント

RT ミドルウエアとは、上述したように RT のためのプラットフォームでありそのためのミドルウエアをさします。図 1.1 は RT ミドルウエアの概観図です。RT ミドルウエアは、

- コンポーネントフレームワーク、
- 標準的に再利用されるソフトウエア部品群、
- ライブラリ群、
- 標準サービス群

などから構成されます。

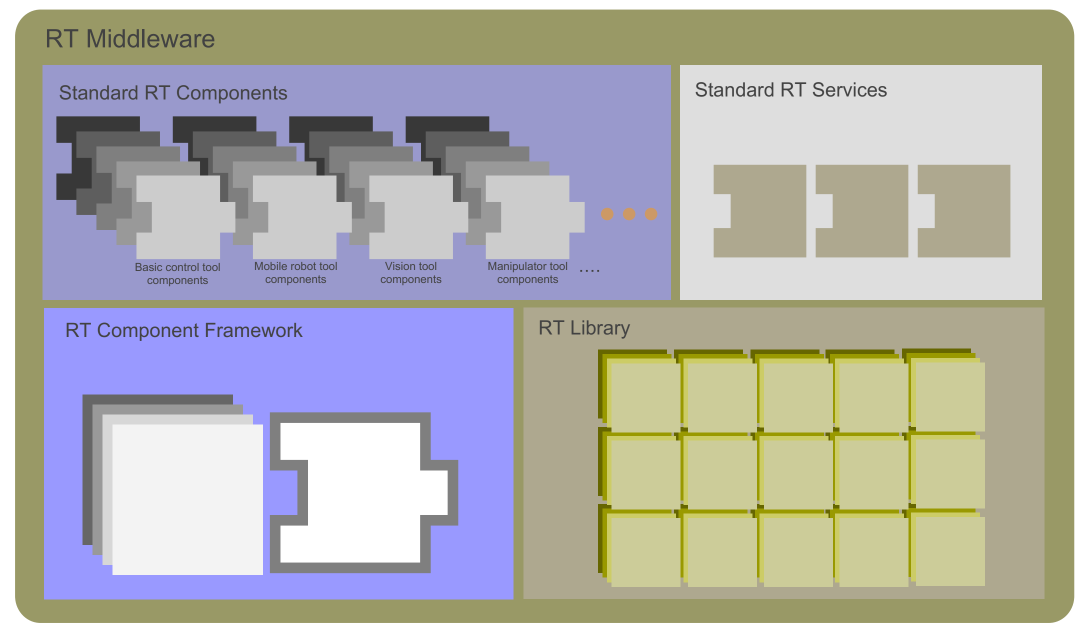


図 1.1: RT ミドルウエア

RT ミドルウエアのうち、RT システムを構成する主体となるソフトウエア部品群を RT コンポーネントと呼びます。RT コンポーネントは通常、上記の RT コンポーネントフレームワークを用いて作成されます。既存の RT コンポーネントを利用したり、自分で開発した RT コンポーネントを部品のように組み合わせることで目的とするシステムを作成します。RT コンポーネント群を使用するために必要な機能で、コンポーネント自身にないものはいつかのサービス群を利用することが出来ます。また、コンポーネントの作成を助けるライブラリなども提供することを想定しています。

以上の全てをまとめたものを RT ミドルウエアと呼びます。

1.5 RT コンポーネント

RT コンポーネントの基本的構造を説明します。図 1.2 は RT コンポーネントのアーキテクチャ・ブロック図です。RT コンポーネントの仕様ではネットワーク上に分散されたコンポーネントへの透過的アクセスを実現するために、分散オブジェクト技術を用いて実装されることを想定しています。

OpenRTM-aist では、OS・言語非依存性を重視して CORBA を用いて実装されています。

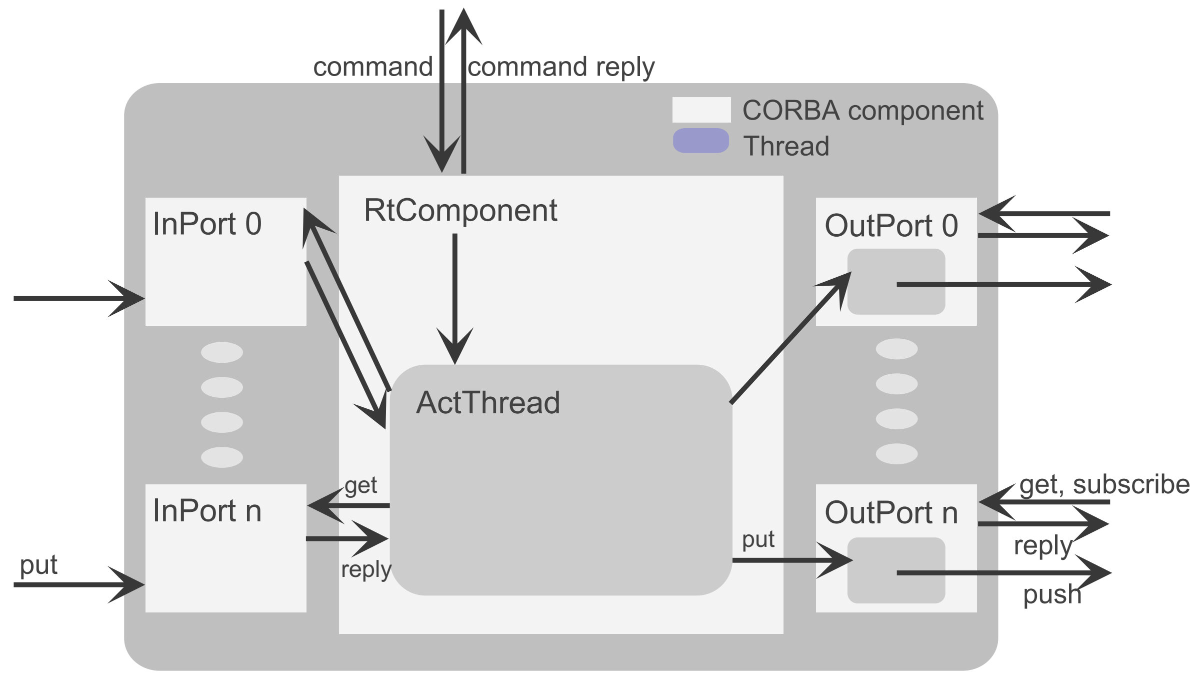


図 1.2: RT コンポーネント

通常の分散オブジェクトでは、データのやり取りは基本的にメソッド呼び出しを介して行われます。これに対して、ロボットの制御における入出力データストリームは、通常メソッドよりもデータ型が重要な意味を持ちます。

例えば、6 自由度を持つジョイスティックがあり、これが出力する 6 個の double 型データを用いて対象 (マニピュレータ、移動ロボット、ヒューマノイド等) を様々な方式 (位置制御、速度制御、インピーダンス制御等) で制御したいとします。単なる分散オブジェクトとしてシステムを構成するならば、ジョイスティックはあらゆるタイプの制御対象のメソッド (インターフェース) を予め知っていなければなりません。しかしながら、実際には送り手が 6 個の double 型データを送ることができ、受け手も 6 個の double 型データを処理して動作することができれば十分なのです。RT コンポーネントはこういったデータ交換のために、データ型が同じであれば接続可能な InPort/OutPort と呼ばれるデータポートを持っています。

RT コンポーネントはいくつかのオブジェクトから構成されており、大きく分けると以下の 3 つの部分から成ります。

RTComponent RT コンポーネントの本体であり、基本的なインターフェースおよび入出力を行う InPort/OutPort オブジェクトを 0〜n 個以上持つ。処理を行うコアロジックを 1 個持ち、外部または内部からのイベントに応じて内部状態を遷移させる。これを Activity と呼び、他とは独立したスレッドに割り当てられ以下に挙げる入出力とは独立に処理が行われる。

InPort 他の RT コンポーネントからの出力をを受け取りハンドリングする入力ポート。データのストリームを受け取る必要がある RT コンポーネントはこの InPort を持つ。コアロジックは連続的に送られてくるデータに対して処理を行うため、基本的に周期動作を行うコンポーネントのみがこの InPort を持つ。

OutPort 他の RT コンポーネントへ処理結果のデータストリームを渡す出力ポート。受け取る側のコンポーネントから値を取得する pull 型のデータ出力と、サブスクライブすることにより受け取り側へ能動的にデータを送る push 型の動作がある。

OpenRTM-aist では、上記の構造を持つ RT コンポーネントを簡単に作成できるフレームワークやツール群があります。これらを使用することで、既存のソフトウエア資産 (ロボット制御用ライブラリ等) にコンポーネントの皮に包むことができます。さらに、一旦コンポーネント化されたソフトウエアは、仕様が変わらない限り、最コンパイルすることなく他のコンポーネントと組み合わせシステムを構築することが出来るのです。

1.6 RT コンポーネント、マネージャ、ネームサービス

上述したように、RT コンポーネントは、ロボットシステムを構築するための様々な機能を持っていますが、RT コンポーネントだけでは意味のあるシステムを作ることは出来ません。幾つかのホスト上で動作するコンポーネントを検索し、接続、起動、停止するなど操作を行う必要があります。

OpenRTM-aist においては、RT コンポーネントのライフサイクルの管理、ネームサーバへの登録などを一括して行うオブジェクトとしてコンポーネントマネージャを提供しています。

コンポーネントとマネージャやネームサービスなどの関係は図 1.3 のようになっています。

コンポーネントは通常、ロード可能なライブラリ形式 (=モジュールと呼ぶ) で提供され、マネージャはこのモジュールをロードし、インスタンス化することでコンポーネントを生成します。

また、マネージャは生成したコンポーネントがネットワーク上の他のホストからも検索できるように、CORBA ネーミングサービスを使用して、ネームサーバへコンポーネントの名前とオブジェクトリファレンスを登録します。また、コンポーネントはマネージャを通してネームサーバに対して問い合わせることにより、他のコンポーネントを名前で検索することも出来ます。

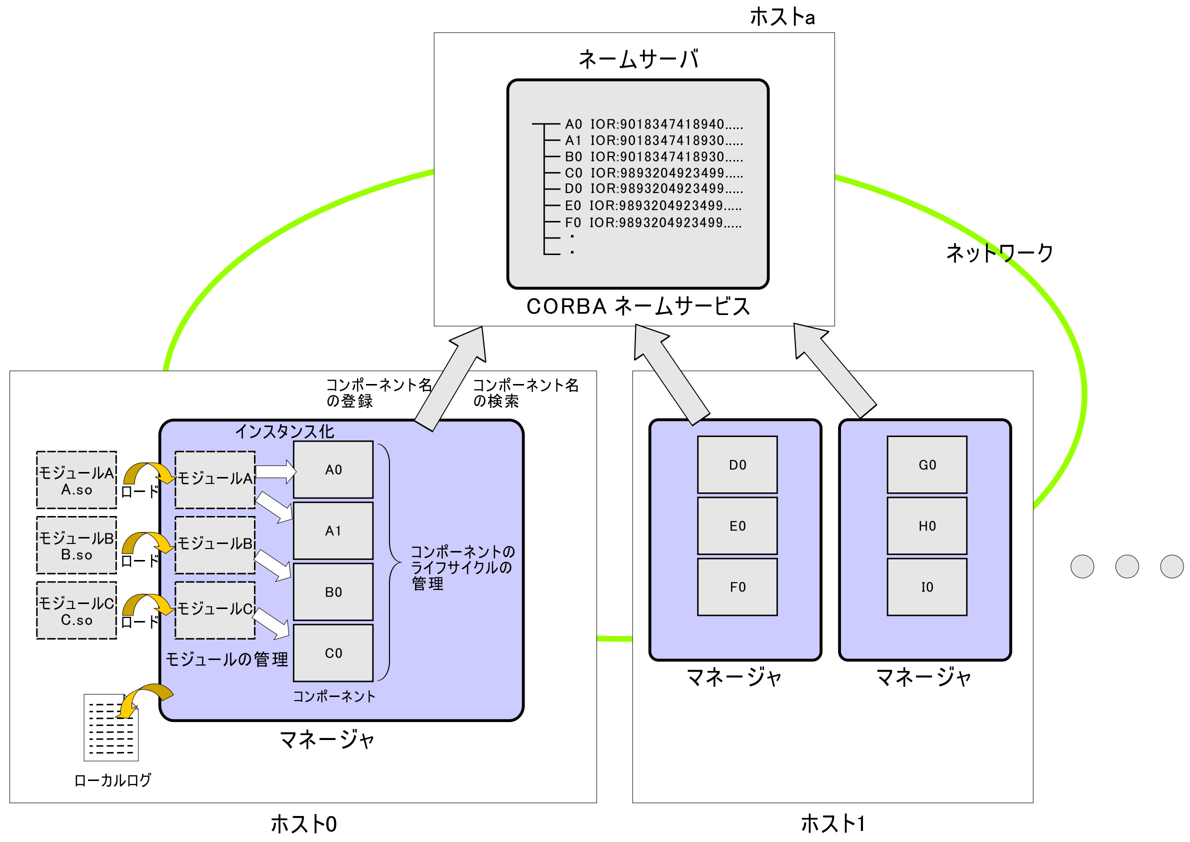


図 1.3: RT コンポーネント、マネージャ、ネームサービス

第2章　RTコンポーネント入門

2.1　OpenRTM-aist のインストール

ここからは、OpenRTM-aist をソースからビルドしインストールする手順を説明します。自分でソースからビルドしてインストールする場合には、以下の手順に従ってインストールを行ってください。パッケージを利用する場合には読み飛ばしても構いません。

2.2　UNIX 系システムでのビルドとインストール

2.2.1　動作条件

OpenRTM-aist のインストールには下記のパッケージが必要です。各パッケージのインストールは、それぞれのドキュメントに従って行ってください。なお、簡単なインストール方法については巻末の付録 A 「依存パッケージ類のビルドについて」にて説明していますので参照してください。

● 依存パッケージ一覧

ACE	The ADAPTIVE Communication Environment (ACE) 5.4 以上 URL: `http://www.cs.wustl.edu/~schmidt/ACE.html`
boost	boost C++ ライブラリ 1.30 以上 URL: `http://www.boost.org/`
ORB	現在のところ対応 ORB 実装は omniORB-4.0 以上のみとなっています。 omniORB – URL: `http://omniorb.sourceforge.net/`
Python	Python 2.2 以上。Python でコンポーネントを開発する場合には、Python の開発環境も必要になります。 URL: `http://www.python.org/`
gcc	gcc は version3 以降が望ましい。gcc2.95.x でもコンパイル可能ですが、オプションを指定する必要があります。(下記参照) URL: `http://gcc.gnu.org/`

これらのパッケージが、システムの標準的なディレクトリ (`/usr`，`/usr/local` 等) 下に正しくインストールされていると、OpenRTM-aist のコンパイルおよびインストール作業がよりスムーズにできるでしょう。標準のディレクトリとは、コマンド、ヘッダ、ライブラリそれぞれのインストールが下記のディレクトリにされているということです。

● 標準インストールディレクトリ

コマンド	/usr/bin, /usr/local/bin
ヘッダ	/usr/include/*package_name*, /usr/local/include/*package_name*
ライブラリ	/usr/lib, /usr/local/lib

最近の Linux では各ディストリビューションの標準パッケージ類が、/usr/ 以下の [bin|include|lib] に、それ以外の UNIX 系 OS では /usr/local/ あるいは /opt 以下の [bin|include|lib] の下にインストールされていることが多いとおもいます。OpenRTM-aist は autoconf, automake を使用してビルドされるので、上記以外の場所に依存しているパッケージがインストールされている場合でもビルド可能ですが、あまり推奨されません。後々の面倒を避けるためにも、標準的なディレクトリに各種パッケージをインストールしておくことをお勧めします。

また、debian GNU Linux や RedHat/Fedora などのディストリビューションでは、上記の依存パッケージ類はすでに deb パッケージや、rpm パッケージ化されているので、それらをインストールするほうがよいでしょう。

● 依存パッケージの入手先

debian GNU Linux	http://www.debian.org/distrib/packages を参照
	ace – ○, boost – ○, omniORB – ○, Python – ○
RedHat/Fedora 系	http://rpmfind.net/ を参照
	ace – ○, boost – ○, omniORB – ○, Python – ○
Vine Linux	http://vinelinux.org/ を参照
	ace – ×, boost – ×, omniORB – ×, Python – ○
FreeBSD	http://www.freebsd.org を参照
	ace – ○, boost – ○, omniORB – ○, Python – ○

Vine Linux 用 ace, boost, omniORB (+ omniORBpy) については、産総研で用意していますので、OpenRTM-aist の Web ページ (http://www.is.aist.go.jp/rt/) を参照して下さい。

2.2.2 ソースコードの展開

まずソースコード OpenRTM-aist-0.2.0.tar.gz を適当なディレクトリに展開します。自分のホームディレクトリでも構いませんし、ソースビルド用ディレクトリがあればそこで行っても構いません。

ただし、一般的にこれ以後の作業はインストール作業を除いて、一般ユーザ (root 以外) として行ったほうがよいでしょう。

```
> tar xvzf OpenRTM-aist-0.2.0.tar.gz
> cd OpenRTM-aist-0.2.0
```

2.2.3 ビルド

OpenRTM-aist はパッケージのビルドに autoconf, automake を使用していますので、ビルドは、他の autoconf, automake を使用したパッケージと同様に行うことができます。ただし、ORB を指定するオプションは必須ですので必ず指定してください。

```
> ./configure [options]
```

`./configure` に与えることのできるオプションの一部を以下に示します。

2.2.3.1 使用する ORB の指定

※現在は omniORB しかサポートされていません。他のオプションは将来のために予約されています。追加オプションとして ORB がインストールされているディレクトリを `=dir` の部分に指定することができます。

● ORB の選択オプション

`--with-omniorb=dir`	ORB に omniORB を使用します。
`--with-tao=dir`	ORB に TAO を使用します。
`--with-mico=dir`	ORB に MICO を使用します。
`--with-orbix=dir`	ORB に Orbix/E を使用します。
`--with-orbacus=dir`	ORB に ORBacus を使用します。

2.2.3.2 サポートする言語の指定

OpenRTM-aist でのコンポーネント開発は、C++言語での開発を前提としています。現在のところ、他にサポートする言語は、スクリプト言語である Python のみサポートしています。Python での開発を行う場合は、オプションを指定する必要があります。これを指定した場合は Python の開発環境がシステムにインストールされていなければなりません。また、Python のバージョンは 2.2 以上である必要があります。

Python が標準的なディレクトリ以外にインストールされている場合は `=dir` にインストール先のディレクトリを指定してください。ディレクトリの指定の仕方は `Python.h` が

python_dir/`python2.x/Python.h` にある場合、`--with-python=`*python_dir* のように指定します。

● **サポートする言語の選択オプション**

`--with-python=dir`	Python 言語のインターフェースを提供します。

2.2.3.3 gcc3.x 以前のコンパイラ（gcc2.95.x 等）を使用する場合

最近まで使用されていた gcc-2.95.x など、バージョン 3 未満の gcc ではテンプレート・クラスのインスタンス生成の深さが深すぎる場合にコンパイルエラーになる場合があります。このため、これを回避するオプション (`-ftemplate-depth-n`) を指定してこれを回避しなければなりません。gcc-2.95.x などのコンパイラを使用する場合は、以下のオプションを指定することでこのコンパイラオプションを自動的についかします。

● **コンパイラの選択オプション**

`--with-gcc2`	コンパイラに gcc-2.x を使用する。

2.2.3.4 ACE include パス, lib パス

ACE が標準的なディレクトリにインストールされていない場合以下のオプションでヘッダとライブラリのディレクトリを指定して下さい。

● **ACE ヘッダライブラリディレクトリ指定オプション**

`--with-ace-includes=dir`	ACE のヘッダディレクトリを指定します。
`--with-ace-lib=dir`	ACE のライブリディレクトリを指定します。

ACE はソースコードのアーカイブを展開すると `ACE_wrappers` というディレクトリを作成します。ヘッダ及びライブラリはどちらもは `ACE_wrappers/ace` 以下に展開、または作成されます。`ACE_wrppers` が仮に `/tmp/src/ACE_wrappers` にある場合、以下のように指定します。

```
./configure ... --with-ace-include=/tmp/src/ACE_wrappers \
                --with-ace-libs=/tmp/src/ACE_wrappers/ace ...
```

ここで、注意が必要なのは、ACE のヘッダ群は `#include "ace/ACE.h"` というようにサブディレクトリ `ace` を仮定して参照するため、`/tmp/src/ACE_wrappers` を指定しなければならないということです。これに対して、ライブラリディレクトリはライブラリがあるディレクトリを直接指定、つまり `/tmp/src/ACE_wrappers/ace` を指定する必要があるということです。

2.2.3.5 その他

その他使用可能なオプションは--help オプションで参照して下さい。

```
> ./configure --help
```

2.2.4 configure, make

システムの環境に適したオプションを指定して、configure を下のように実行します。最後に、どのようなコンパイルオプションが指定されるかが表示され、Makefile, ヘッダ、依存関係のファイルを生成して終了します。configure を行った後は configure が正常に終了したことを確認して下さい。

```
> ./configure --with-omniorb --with-python
        :(中略)
------------------------------------------------------------

  OpenRTM-aist will be compiled with the following environment.

------------------------------------------------------------
CXX: g++
CPPFLAGS: -I/usr/local/include -I/usr/local/include \
          -I/usr/local/include -I/usr/local/include
CXXFLAGS: -I/usr/local/include -O
BUILD_CFLAGS:
BUILD_CPPFLAGS:

LD: /usr/bin/ld
LIBS: -lpthread -lACE -lboost_regex -lomniORB4 -lomnithread -lomniDynamic4
LDFLAGS:  -L/usr/local/lib -L/usr/local/lib -L/usr/local/lib
LDSOLIBS:  -lACE -lboost_regex -lomniORB4 -lomnithread -lomniDynamic4

ORB: omniORB
IDLC: /usr/local/bin/../bin/omniidl
IDL_FLAGS: -bcxx -Wba -nf

WRAPPERS:
------------------------------------------------------------

configure: creating ./config.status
config.status: creating Makefile
        :(中略)
config.status: creating rtm/config_rtc.h
config.status: executing depfiles commands
```

config.status: executing depfiles commands が最後に実行されていなければ、configure に失敗しています。もし、エラーや Warning が出ている場合は config.log を見て原因を特定して対処してください。たいていは、依存パッケージがインストールされていない、インストールされているがヘッダやライブラリが見つからないなどの理由が多いようですので、オプションをよく調べてみてください。

configure が正常に終了した事を確認したら make を行います。数分から十数分程度でビルドが完了します。

```
> make
```

再帰的に全てのディレクトリをビルドします。インストールするためには少なくとも下記のディレクトリが正常にビルドされている必要があります。

● OpenRTM ディレクトリ構成

OpenRTM-aist-0.2.0/rtm	RT コンポーネントフレームワーク
OpenRTM-aist-0.2.0/rtm/idl	RT コンポーネント IDL
OpenRTM-aist-0.2.0/util/rtm-config	RTM コンフィギュレーションツール
OpenRTM-aist-0.2.0/util/rtm-naming	RTM ネーミングサービスラッパ
OpenRTM-aist-0.2.0/util/rtc-template	RTC テンプレートツール
OpenRTM-aist-0.2.0/util/rtc-link	コンポーネント操作 GUI

ビルドが正常に終了したら、ヘッダファイル、ライブラリ、ユーティリティコマンド群、ドキュメントをインストールします。システムディレクトリ (/usr や /usr/local) へインストールするには root になる必要があります。

```
> su
# make install
```

root になって make install とすると、ヘッダ類やライブラリがシステムのディレクトリにインストールされます。

2.2.5 サンプルのテスト

インストールが正常に終了したら、example/SimpleIO 以下のサンプルでテストを行っておくとよいでしょう。このサンプルは、コンソールから入力された数値を出力するコンポーネントと受け取った数値をコンソールに表示するコンポーネントを実行し、接続するだけの簡単なサンプルです。

```
> cd examples/SimpleIO/
> run.sh
```

run.sh ではターミナルウインドウとして kterm を仮定しています。もし、kterm ではなく他のターミナルウインドウを使用している場合には、実行シェルスクリプト run.sh を適宜書き換えて実行してください。

はじめに画面 kterm が 2 つ開き、数秒後にもうひとつ開きます。3 つ目の kterm が開くまで、必ず待ってください。ウインドウタイトルに ConsoleIn とある方で数字を打ち込みます。入力する数字は、`long int` に収まる範囲の数値を入力してください。数字を打ち込むと、ConsoleOut とウインドウタイトルのある方から、入力したのと同じ数字と時刻 (sec:nsec のフォーマットになっている) が現れます。ここまでできれば、OpenRTM-aist のメインのライブラリ `libRTC.so` が正しくビルドされていることがわかります。

2.2.6 コンポーネントテンプレートのテスト

次に、コンポーネントのテンプレートジェネレータを使用して、簡単なコンポーネントを作成するテストを行います。テンプレートジェネレータ `rtc-template` を使用します。コンポーネントを作成するデイレクトリ (任意のディレクトリで結構です。) を作成して、そこでコンポーネントを作成しコンパイルしてみます。

```
> cd [コンポーネントを作成するディレクトリ]
> mkdir SampleComponent
> cd SampleComponent
```

まずは、`rtc-template` のヘルプを見てみます。

```
> rtc-template --help

Usage: rtc-template [OPTIONS]
Options:
    [--help]                              Print this help.
    [--c++]                               Generate C++ template code.
    [--python]                            Generate Python tempalte code.
    [--output[=output\_file]]              Output base file name.
    [--module-name[=name]]                Your module name.
    [--module-desc[=description]]         Module description.
    [--module-version[=version]]          Module version.
    [--module-author[=author]]            Module author.
    [--module-category[=category]]        Module category.
    [--module-comp-type[=component\_type]]  Component type.
    [--module-act-type[=activity\_type]]    Component's activity type.
    [--module-max-inst[=max\_instance]]     Number of maximum instance.
    [--module-lang[=language]]            Language.
    [--module-inport[=PortName:Type]]     InPort's name and tyoe.
    [--module-outport[=PortName:Type]]    OutPort's name and type
        :
     中略
        :
Example:
    rtc-template --c++ --module-name=Sample --module-desc='Sample component' \
    --module-version=0.1 --module-author=DrSample --module-category=Generic \
    --module-comp-type=COMMUTATIVE --module-act-type=SPORADIC \
    --module-max-inst=10  \
    --module-inport=Ref:TimedFloat --module-inport=Sens:TimedFloat \
    --module-outport=Ctrl:TimedDouble --module-outport=Monitor:TimedShort
```

`rtc-template` に作成したいコンポーネントの設定を引数として渡すと、コンポーネントの

雛形のコードを作成します。ここでは、help の表示の最後に現れた ”Example:” 以下をコピーして試してみます。

```
> rtc-template --c++ --module-name=Sample --module-desc='Sample component' \
    --module-version=0.1 --module-author=DrSample --module-category=Generic \
    --module-comp-type=COMMUTATIVE --module-act-type=SPORADIC \
    --module-max-inst=10  \
    --module-inport=Ref:TimedFloat --module-inport=Sens:TimedFloat \
    --module-outport=Ctrl:TimedDouble --module-outport=Monitor:TimedShort
Sample.h was generated.
Sample.cpp was generated.
SampleComp.cpp was generated.
Makefile.Sample was generated.
> ls
Makefile.Sample         Sample.h
Sample.cpp              SampleComp.cpp
```

このように、コンポーネントの C++のソースコードと Makefile (`Makefile.Sample`) が作成されます。作成されたソースコードはこの Makefile をそのままでビルドできます。

```
> make -f Makefile.Sample
もしくは
> mv Makefile.Sample Makefile
> make
```

のように、`make` の `-f` オプションを使用し `Makefile.Sample` を直接していするか、`Makefile.Sample` を `Makefile` にリネームして make します。

```
> make -f Makefile.Sample
rm -f Sample.o
g++ `rtm-config --cflags` -c -o Sample.o Sample.cpp
     :
    中略
     :
g++ `rtm-config --libs` -o SampleComp Sample.o SampleComp.o
rm -f Sample.so
g++ -shared `rtm-config --libs` -o Sample.so Sample.o
> ls
Makefile.Sample  Sample.h  Sample.so*   SampleComp.cpp
Sample.cpp       Sample.o  SampleComp*  SampleComp.o
```

これで、ローダブルモジュール版のコンポーネント (`Sample.so`) と実行形式のコンポーネント (`SampleComp`) が作成されました。ここで、実行形式のコンポーネントを実行してみます。コンポーネントの実行にはコンフィギュレーションファイル (通常は `rtc.conf` という名前) が必要です。ここでは、簡易版のものをカレントディレクトリに作成します。

```
> cat > rtc.conf
NameServer        現在の PC のホスト名:ポート番号
^D (Ctrl+D)
```

ここでは、仮にホスト名：rtm.or.jp、 ポート番号:6789 とします。rtm.or.jp は今作業しているホスト名に読み替えてください。

```
> cat > rtc.conf
NameServer      rtm.or.jp:6789 (このように入力する)
^D (Ctrl+D)
> cat rtc.conf (確認)
NameServer      rtm.or.jp:6789
```

次に、CORBA のネーミングサービスを起動します。CORBA のネーミングサービスは、

```
> rtm-naming ポート番号
```

で起動できますので、先ほど `rtc.conf` で指定したポート番号を指定して起動します。

```
> rtm-naming 6789
Starting omniORB omniNames: ichi:9999
n-ando@ichi:/tmp/SampleComponent>
Fri Oct 29 17:12:51 2004:

Starting omniNames for the first time.
Wrote initial log file.
Read log file successfully.
Root context is IOR:010000002b00000049444c3a6f6d672e6f72672f436f734e616d696e672f
4e616d696e67436f6e746578744578743a312e300000010000000000000060000000010102000e00
00003135302e32392e39362e313638000f270b0000004e616d6553657276696365000200000000
000008000000010000000054544101000001c000000010000000100010001000000010001050901
0100010000009010100
Checkpointing Phase 1: Prepare.
Checkpointing Phase 2: Commit.
Checkpointing completed.
```

次に、コンポーネントを起動します。

```
> SampleComp -f rtc.conf
```

これで、コンポーネントが起動できました。この段階までできれば、OpenRTM-aist が正常にインストールされたことになります。

2.3 RTCLink による動作確認

前述の SimpleIO サンプルでは、コンポーネントの起動・接続を接続専用の別プログラムで行いました。しかしながら、開発段階で様々なコンポーネントを色々と組み合わせてテストするには、プログラムで接続などを行うのは非常に煩雑になり、RT コンポーネントのメリットを生かせません。OpenRTM-aist には、コンポーネントの起動・接続などを行うための GUI ツール (RTCLink) が用意されています。

ここでは、RTCLink を使用して、前述の ConsoleIn, ConsoleOut コンポーネントを接続しテストする方法を説明します。RTCLink の詳しい使用方法については後述する「4.1 RTCLink」を参照してください。

2.3.1 操作手順

まず、RTCLink を起動します。起動方法は、コンソール上で「rtc-link」と入力するだけです。

SimpleIO のディレクトリの中の、`ConsoleInComp` と `ConsoleOutComp` をそれぞれ別のウインドウで起動します (図 2.1、図 2.2)。

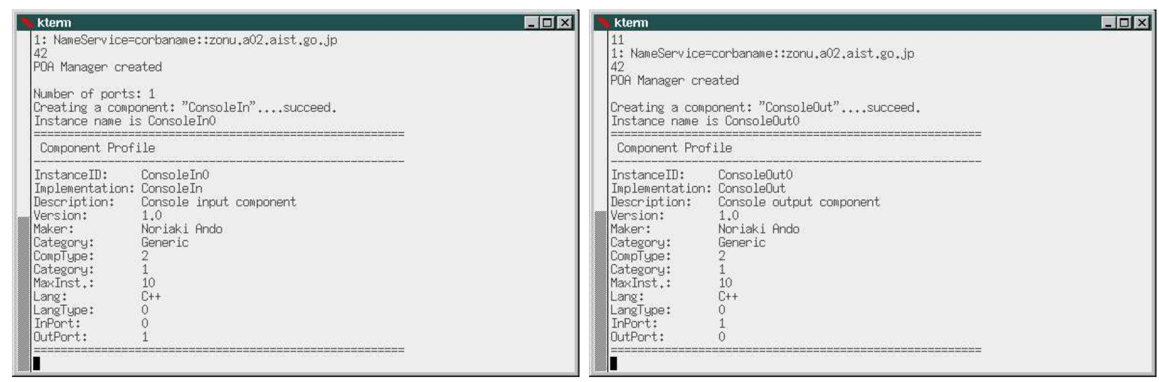


図 2.1: `ConsoleInComp` の起動　　図 2.2: `ConsoleOutComp` の起動

次に、RTCLink の左側のツリーウインドウをクリックしてツリーを展開します。ツリーの中に、ConsoleIn コンポーネントと、ConsoleOut コンポーネントがあることを確認してください (図 2.3)。コンポーネントは のアイコンで表示されています。

さらに、ツールバー上の ボタンを押してシステムドローウインドウを中央の部分に表示させます (図 2.4)。このウインドウの中でシステムを構築してゆきます。

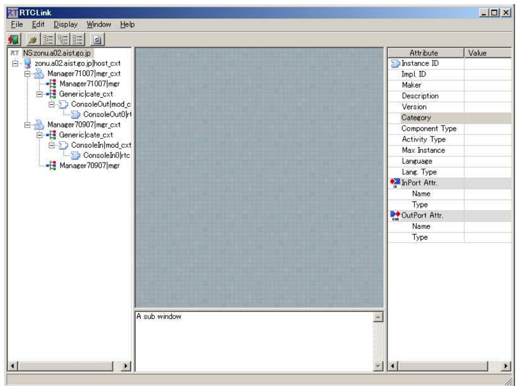


図 2.3: ツリーの展開

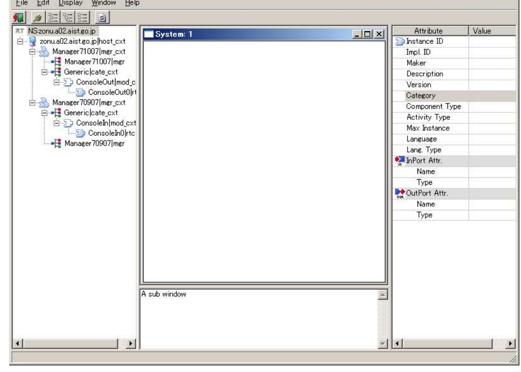


図 2.4: システムドローウインドウの表示

ツリーウインドウの中で のアイコンで表示されて ConsoleIn と ConsoleOut のコンポーネントをシステムドローウインドウへドラッグアンドドロップします。

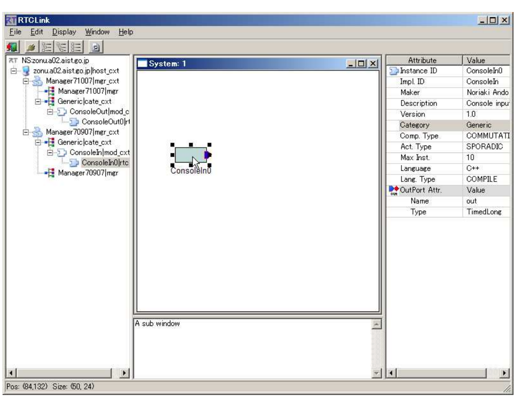


図 2.5: ConsoleIn コンポーネントの配置

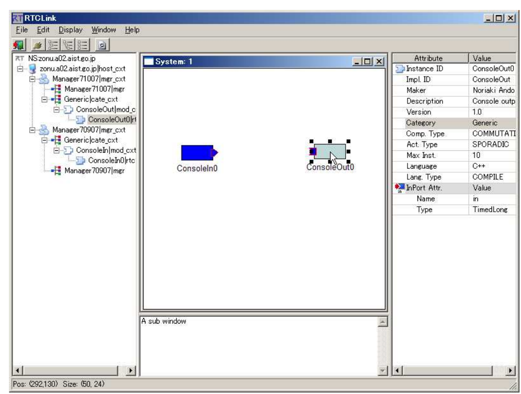


図 2.6: ConsoleOut コンポーネントの配置

図 2.7 のように ConsoleOut コンポーネントの OutPort をクリックし、ConsoleIn コンポーネントの InPort へドラッグします。これで、ConsoleOut コンポーネントの OutPort と ConsoleIn コンポーネントの InPort が接続されます。

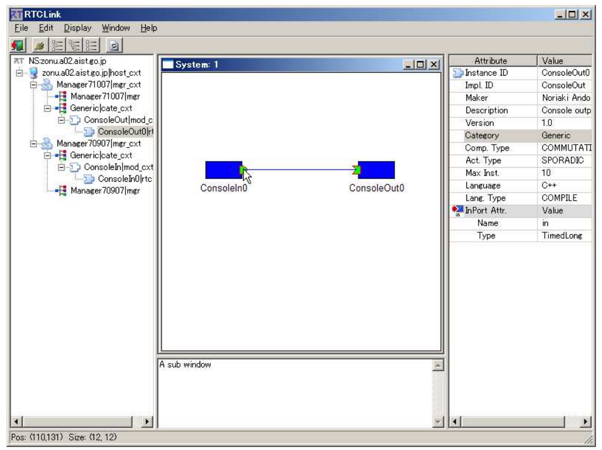


図 2.7: InPort と OutPort の接続

次に、システムドローウインドウ上の ConsoleIn コンポーネントを右クリックしコンテキストメニューを出します。「Start」を選択すると ConsoleIn コンポーネントがスタートします (図 2.8)。同様に ConsoleOut コンポーネントコンポーネントもスタートさせます (図 2.9)。

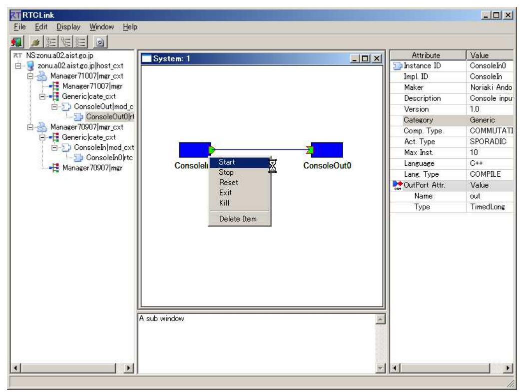


図 2.8: ConsoleIn のスタート

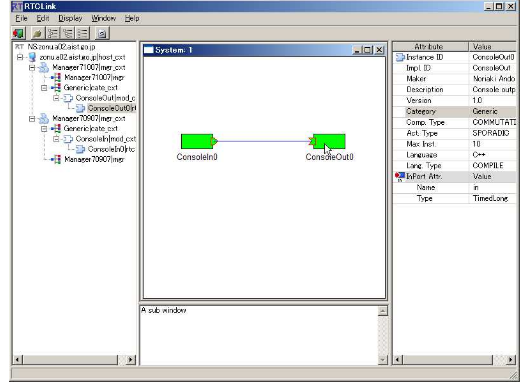


図 2.9: ConsoleOut のスタート

ConsoleIn コンポーネントをスタートさせると図 2.10 のように数値の入力を求めてきます。適当な数値を入力します。入力する数値は long int 型に入力範囲の数値にしてください。ここで入力した数値が、ConsoleOut コンポーネントへ送られて表示されます。

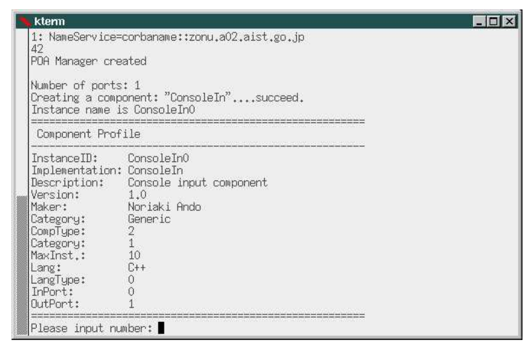


図 2.10: ConsoleIn のスタート直後

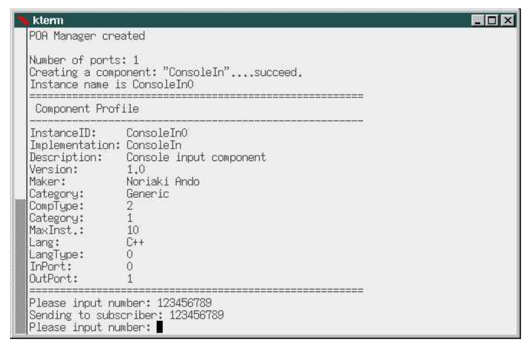


図 2.11: ConsoleIn に数値を与える

ConsoleOut コンポーネントのほうでは、図 2.12 のようにスタート直後から数値と時刻が出力されますが、ConsoleIn への入力がない間は 0 を表示し続けます。ConsoleIn へ数値を入力すると、その数値が図 2.13 のように表示されます。

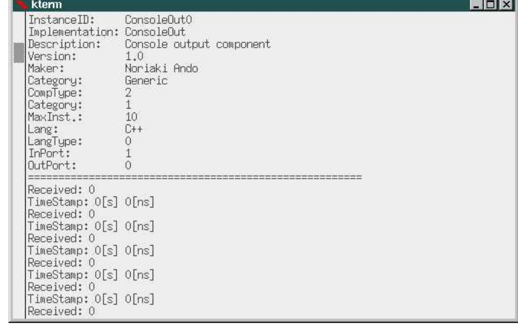


図 2.12: ConsoleOut のスタート直後

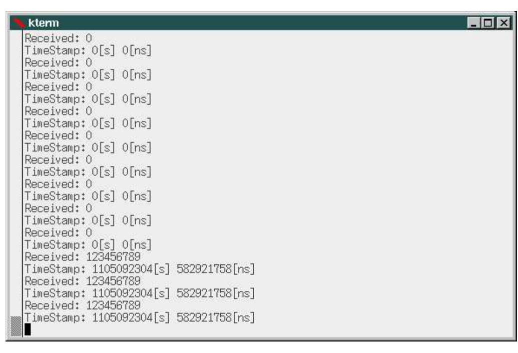


図 2.13: ConsoleOut で数値が表示される

これらが正常に動作すれば、OpenRTM-aist は正常に動作していることになります。

2.4 Windows 系システムでのビルドとインストール

将来的には対応する予定ですが、現在は未対応です。

第3章　RTコンポーネントプログラミング

3.1　RTコンポーネント開発の流れ

OpenRTM-aist は、コンポーネントを開発したいユーザ (コンポーネントデベロッパ) が持つ既存のソフトウエア資産、あるいは新たに作成したソフトエアを容易に RT コンポーネント化するためのフレームワークを提供します。

コンポーネント作成の大まかな流れは図 3.1 のようになります。

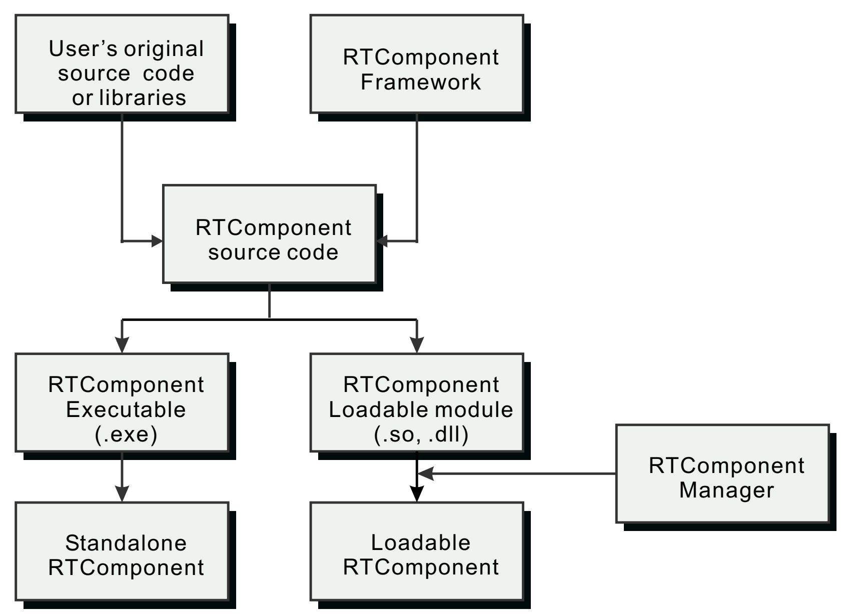


図 3.1: RT コンポーネントの開発フロー

コンポーネントデベロッパは、既存のソフトウエア資産のライブラリ関数・クラスライブラリ等をコンポーネントフレームワークに埋め込みコンポーネントを作成します。こうすることで、既存のソフトウエア資源をソフトウエア部品である RT コンポーネントとして作成しておき、様々な場面で再利用することが出来るようになります。作成された RT コンポーネントは、ネットワーク上の適切な場所に配置して、分散オブジェクトとしてネットワーク上の任意の場所から利用することができます。

図 3.1 に示すように、RT コンポーネントフレームワークに則って作成された RT コンポーネントは大きく分けて 2 つの実行形式のバイナリファイルとして作成することが出来ます。スタンドアロン RT コンポーネント (Standalone RTComponent) は、単一ファイルでそのまま実行できる実行バイナリ形式です。ローダブルモジュール RT コンポーネント (Loadable Module RTComponent) は動的にロード可能なローダブルモジュール形式のバイナリファイルです。RT コンポーネントはこれらの 2 つの形式で作成、配布、実行することができます。

3.1.1 コンポーネントの形態

上でも述べたように、RT コンポーネントは、単体で実行可能な「スタンドアロン RT コンポーネント (Standalone RTComponent) 」および、動的にロードされ実行される「ローダブルモジュール RT コンポーネント (Loadable Module RTComponent) 」の 2 つの形式として生成することができます。「スタンドアロンコンポーネント」および「ローダブルモジュールコンポーネント」は以下のような特徴があります。

3.1.1.1 スタンドアロンコンポーネント

単体で実行可能なバイナリ形式のコンポーネントです。1 プロセスにつき原則として 1 種類のコンポーネント (複数のインスタンスを持つことができる。) に対応します。他のコンポーネントとの通信は CORBA 経由で行うため、速度的には不利ですが、他のコンポーネントがエラーで停止しても、影響を被らないため安定性では有利となります。

また、コンポーネントの起動、停止はプロセスのそれと同義ですので、他のコンポーネントに影響を与えることなくプロセスごと起動や停止することが出来ます。

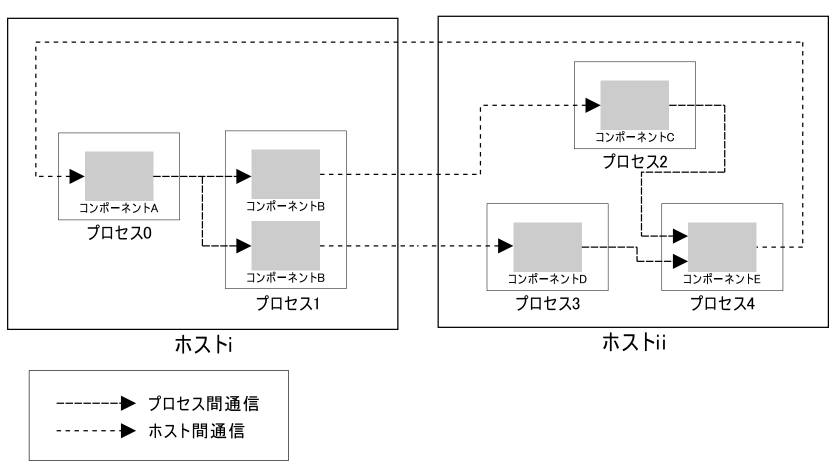


図 3.2: スタンドアロンコンポーネント

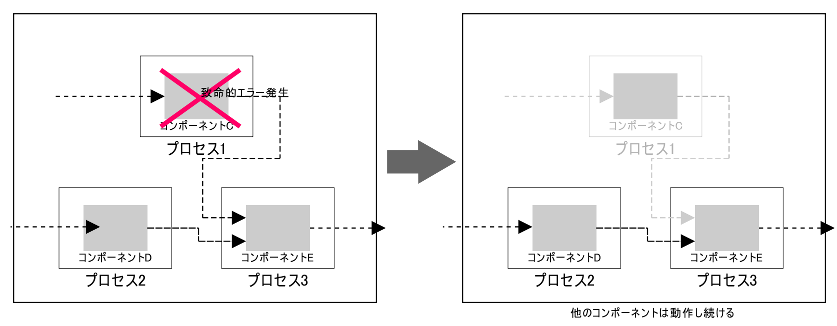


図 3.3: スタンドアロンコンポーネントのエラーの影響

3.1.1.2 ローダブルモジュールコンポーネント

マネージャにより動的にロードされ、実体化されるコンポーネントです。

ローダブルモジュールコンポーネント RT コンポーネントは通常、共有ライブラリの形態をとります。共有ライブラリとは、実行時に動的にロードされるライブラリのことです。通常 UNIX 系では拡張子が `.so` (HP-UX では `.sl`)、Windows では `.dll` となっています。ダイナミックロード・コンポーネントを作成する際に書くプログラムには `main` 関数がないのでそのままでは実行できません。実行はコンポーネントサーバが、ダイナミックロードコンポーネントを動的にロードし、コンポーネントサーバ内のコンポーネントマネージャがコンポーネントを生成し実行します。

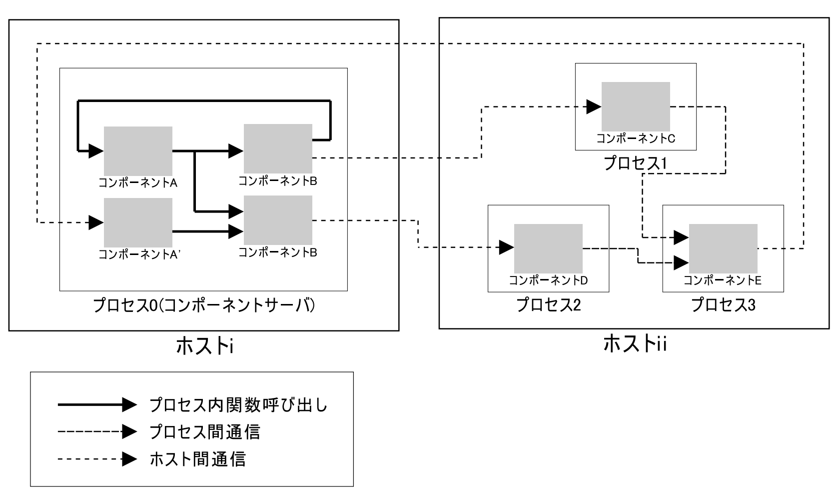


図 3.4: ローダブルモジュールコンポーネント

ローダブルモジュールコンポーネントは、1 つのプロセスに対して複数の種類の複数のインスタンスを持つことができます (図 3.4)。CORBA 実装の多くは同一プロセス内での CORBA オペレーション呼び出しを高速化する機構が組み込まれているものが多く、密接に連携させたいコンポーネントを同一のサーバ上に生成し接続すると、パフォーマンスの向上が見込める場合があります。

ただし、これは使用している CORBA 実装に依る所が大きく、具体的にどの程度高速化されるかは一概には言えませんが、速度的制約の厳しいシステムを構成する場合にはこうした構成方法が有効となる場合があるでしょう。

ローダブルモジュール RT コンポーネントはスタンドアロン RT コンポーネントとは異なり、プロセス内の 1 つのコンポーネントがセグメント違反等の致命的エラーを起こして停止すると、同一プロセス内の全てのコンポーネントも停止します (図 3.5)。したがって、安定でないコンポーネントを他のコンポーネントと同時に動作させると、安定に動作しているコンポーネントも巻き添えにして停止するなど、全体として不安定になるデメリットがあります。

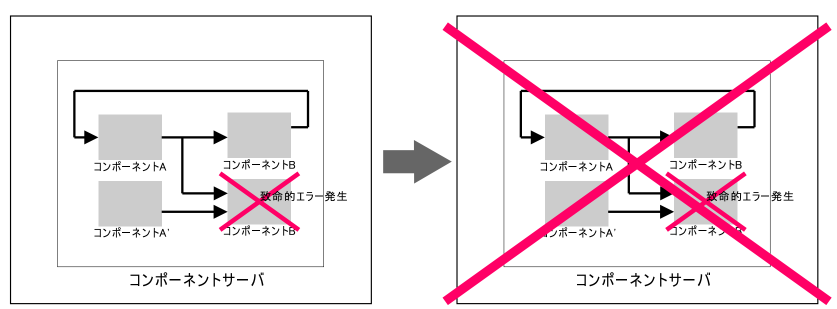


図 3.5: ローダブルモジュールコンポーネントのエラーの影響

一旦インターフェース等の仕様が確定し、十分にデバッグされ安定に動作するコンポーネントが完成すれば、そのコンポーネントを利用するユーザは、コンポーネントのソースコードに手つけることなしに、他のコンポーネントと組み合わせ、ロボットシステムを構築することができます。これが、RT コンポーネントを利用する大きなメリットのひとつです。コンポーネントデベロッパはソースコード、ローダブルモジュール形式、実行ファイル形式の任意の形式で自分が作成したコンポーネントを配布することができます。

3.1.2 モジュールとコンポーネント

OpenRTM-aist では、モジュールとコンポーネントを以下のように区別しています。

● モジュールとコンポーネント

モジュール	RT コンポーネントのモデルを定義するクラス。
コンポーネント	モジュールのクラスから実体化され、実際に配置されるソフトウエア構成部品すなわち RT コンポーネントそのもの。

C++のプログラム的解釈をすると、モジュール=クラスであり、コンポーネント=インスタンスというイメージになります。コンポーネントはモジュールから生成されるものであり、A というモジュールに対して、コンポーネント (A0, A1, ... , An) は複数存在することができます (図 3.6) 。

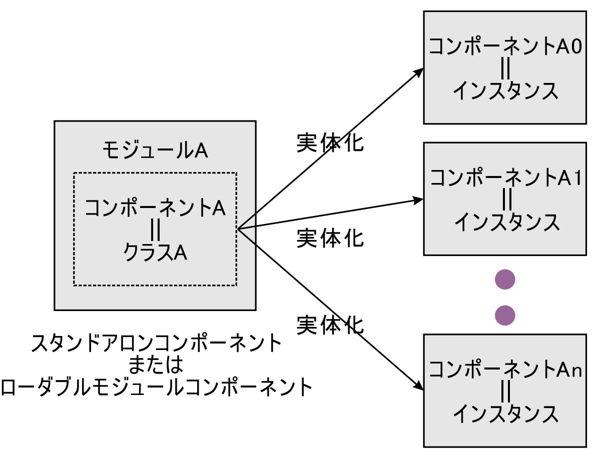


図 3.6: モジュールとコンポーネント

あくまで、処理を実行しているのはコンポーネントであり、モジュールはそのコンポーネントの鋳型という関係です。この鋳型を作成するための基底クラスが OpenRTM-aist のコンポーネント作成のためのフレームワークとして用意されています。コンポーネントデベロッパは、この基底クラスを継承して自分のコンポーネントを作成します。

3.2 コンポーネントの作成

ここからは、実際に簡単なコンポーネントを作成しながら、コンポーネント作成の手順を説明してゆきます。なお、以下で紹介するサンプルのソースコードは、配布ソースコードの `example` ディレクトリの下にあります。

3.3 Hello RT World サンプル

まず始めに、"Hello RT World" コンポーネントを作成します。

この "Hello RT World" コンポーネントは、起動されるとコンソールに"Hello RT World!!!" と表示し続けるだけのコンポーネントです。それ以外には何もしない簡単なコンポーネントです。ここでは、コンポーネントフレームワークの最も基本となる部分について説明します。

コンポーネント作成のために必要なファイルは次の 4 つです。

● HelloWorld コンポーネント作成に必要なファイル

`HelloRTWorld.h`	HelloRTWorld コンポーネントのヘッダファイル。
`HelloRTWorld.cpp`	HelloRTWorld コンポーネントのソースファイル。
`HelloRTWorldComp.cpp`	HelloRTWorld コンポーネントのスタンドアロンコンポーネントのためのソースファイル。
`Makefile`	ビルドするための Makefile。

`HelloRTWorld.h` と `HelloRTWorld.cpp` でコンポーネントのクラスである HelloRTWorld クラスを定義・実装します。HelloRTWorld クラスはコンポーネントの実態となるクラスで、コンポーネントの基底クラスである `RtcBase` クラスを継承して作成します。

また、ここでは HelloRTWorld コンポーネントをスタンドアロンコンポーネントとして実装しますので、コンポーネントを実体化し実行するためのソースコードとして、`HelloRTWorldComp.cpp` を作成します。`HelloRTWorldComp.cpp` をコンパイルして HelloRTWorld クラスのオブジェクトファイルとリンクしたものを `HelloRTWorldComp` とし、これを実行することで HelloRTWorld コンポーネントを実体化・実行します。

同時に、ここではローダブルモジュールも作成します。作成の仕方は `Makefile` 作成の項で述べます。

3.3.1 ヘッダファイル

まずは、ヘッダを定義します。ヘッダは上でも述べたように、`HelloRTWorld` クラスを定義するもので `RtcBase` クラスを継承します。そのほかにも、コンポーネントクラスを作成するた

めの定型的宣言がいくつもあります。では、HelloRTWorld.h を見てみましょう。

HelloRTWorld.h

```
#ifndef __HELLORTWORLD_h__
#define __HELLORTWORLD_h__

#include <rtm/RtcBase.h>
#include <rtm/RtcManager.h>
#include <rtm/RtcInPort.h>
#include <rtm/RtcOutPort.h>

using namespace RTM;

static RtcModuleProfSpec hellortworld_spec[] =
  {

  {RTC_MODULE_NAME, "HelloRTWorld"},
  {RTC_MODULE_DESC, "Hello RT world component"},
  {RTC_MODULE_VERSION, "0.1"},
  {RTC_MODULE_AUTHOR, "DrSample"},
  {RTC_MODULE_CATEGORY, "Generic"},
  {RTC_MODULE_COMP_TYPE, "COMMUTATIVE"},
  {RTC_MODULE_ACT_TYPE, "SPORADIC"},
  {RTC_MODULE_MAX_INST, "10"},
  {RTC_MODULE_LANG, "C++"},
  {RTC_MODULE_LANG_TYPE, "COMPILE"},
  {RTC_MODULE_SPEC_END, NULL}
  };


class HelloRTWorld
  : public RTM::RtcBase
{
 public:
  HelloRTWorld(RtcManager* manager);
  virtual RtmRes rtc_active_do();
};


extern "C" {
  RtcBase* HelloRTWorldNew(RtcManager* manager);
  void HelloRTWorldDelete(RtcBase* p);
  void HelloRTWorldInit(RtcManager* manager);
};
#endif // __HELLORTWORLD_h__
```

行っていることは以下のとおりです。

- インクルードガード
- 各種ヘッダのインクルード
- namespace の使用宣言
- モジュールのプロファイルリストの定義
- HelloWorld クラス宣言
- ファクトリ関数と初期化関数の宣言

まず、インクルードガードですが、ヘッダの二重インクルード防止する決まり文句です。最後の対応する `#endif` を忘れないでください。

RTM に関する各種ヘッダをインクルードします。コンポーネントデベロッパが作成するコンポーネントは、必ず `RtcBase` クラス (もしくはそのサブクラス) を継承することになります。したがって、`RtcBase.h` (もしくはそのサブクラスのヘッダ) をインクルードする必要があります。また、RT コンポーネントのコンストラクタは `RtcManager` のポインタを受け取るので、`RtcManager.h` をインクルードするか、`RtcManager` のクラス宣言が必要となります。

RTM 関連のクラスは、すべて `RTM` という名前空間の下に存在しています。しがたって、これらのクラスや関数を使用する場合には、名前解決演算子`::`を使用して `RTM::RtcBase` と指定

するか、using namespace を使用して名前空間を取り込む必要があります。ここでは、名前空間 RTM を取り込んでいます。

次に、モジュールプロファイルリストを宣言しています。モジュールには、モジュール名、作成者、バージョンといった、モジュール (およびコンポーネント) に関する情報を関連付ける必要があります。モジュールプロファイルには、以下の情報を格納します。

●コンポーネントプロファイル

RTC_MODULE_NAME	モジュール名。モジュールの名前を指定します。(HelloRTWorld)
RTC_MODULE_DESC	モジュール概要。モジュールの短い説明を指定します。(Hello RT world component)
RTC_MODULE_VERSION	モジュールバージョン。モジュールのバージョンをを指定します。(0.1)
RTC_MODULE_AUTHOR	モジュール作成者。モジュールの作成者を指定します。(DrSample)
RTC_MODULE_CATEGORY	モジュールカテゴリ。モジュールのカテゴリを指定します。(Generic)
RTC_MODULE_COMP_TYPE	コンポーネント型。モジュールのが生成するコンポーネントの型を指定します。(COMMUTATIVE)
RTC_MODULE_ACT_TYPE	アクティビティ型。モジュールのアクティビティ型を指定します。(SPORADIC)
RTC_MODULE_MAX_INST	最大インスタンス数。モジュールが生成する最大のコンポーネント数を指定します。(10)
RTC_MODULE_LANG	モジュール記述言語名。モジュールの記述言語を指定します。(C++)
RTC_MODULE_LANG_TYPE	モジュール記述言語型。モジュールの記述言語型を指定します。(COMPILE)
RTC_MODULE_SPEC_END	リスト終了マーカ。このリストの終わりを意味します。リストの最後に必ず必要です。

RTC_MODULE_NAME:モジュール名

このモジュールの名前を英数字で記述します。Motor とか Sensor といった一般的な名詞は極力避け、できるだけ具体的な名前を使用するようにしてください。

デフォルトでは、このモジュール名を利用してコンポーネントに名前がつけられます。たとえば、モジュール名 HelloRTWorld というモジュールからは、HelloRTWorld0, HelloRTWorld1, ... というコンポーネントが生成されます。モジュール名に後ろにつく名前は同一マネージャ上に存在するモジュールに対して、0 から順番に付加されます。このようにして、コンポーネントのインスタンスごとにつけられた名前は、ネーミングサービスに登録され、以後コンポーネントを識別するために使用されます。この名前付けの規則はユーザが任意に変えることが出来ますが、その方法についてはここでは割愛します。

RTC_MODULE_DESC:モジュール概要

モジュールの概要を記述します。わかりやすい概要を簡潔に記述してください。

RTC_MODULE_VERSION:モジュールバージョン

モジュールのバージョン番号を記述してください。バージョン番号のつけ方の作法としては、メジャーバージョンとマイナーバージョンをつけるやり方が一般的です。大幅な機能の改訂、後方互換性が無くなるなどの変更は、メジャーバージョンアップとし、それらに満たない小さな改訂はマイナーバージョンとするなどです。小さなバグフィックスなどは、マイナーバージョンアップとせず修正版としてリリースすることが多いようです。

RTC_MODULE_AUTHOR:モジュール作成者・ベンダ名

このモジュールを作成した人の氏名および所属を記述します。

RTC_MODULE_CATEGORY:モジュールカテゴリ

モジュールのカテゴリを記述します。カテゴリに関しては、特に決まりを設けていませんが、今後規定が設けられる可能性があります。今のところは、自由に名前をつけてもかまいません。

RTC_MODULE_COMP_TYPE:コンポーネント型

コンポーネントはその存在形態により以下の 3 種類の型に分けられます。

● コンポーネント型

STATIC	静的に存在するコンポーネント。コンポーネントはモジュールの初期化と同時に生成され、以後動的に生成されたり削除されたりしません。ハードウエアと深く結びついたコンポーネントは、コンポーネントの数が物理的デバイスの数に制限されます。このように、動的な生成が意味を成さないコンポーネントは STATIC 型に当たります。
UNIQUE	このコンポーネントは、動的に生成したり削除したりすることができます。しかしながら、それぞれのコンポーネントは内部に固有の状態を持っており、必ずしも可換ではありません。
COMMUTATIVE	このコンポーネントは、動的に生成したり削除したりすることができます。また、内部に固有の状態を持たないので、生成されたコンポーネントは可換です。

自分が作成したいコンポーネントの型にあわせて、STATIC，UNIQUE，COMMUTATIVE のいず

れかをダブルクオートで囲んで指定します。このコンポーネントは COMMUTATIVE 型に設定されています。現在のところ、この設定はコンポーネントの挙動に影響を与えません。

RTC_MODULE_ACT_TYPE:アクティビティ型

コンポーネントのアクティビティは、動作形態の種類により以下の 3 種類に分けられます。(OpenRTM-aist-0.2.0 の実装ではこれらの宣言は使用されていません。リアルタイム周期処理を実現したい場合は、現在はユーザがその処理を実装する必要がありますが、将来的にはリアルタイム OS 上でのリアルタイムな周期実行をサポートする予定です。)

●アクティビティ型

PERIODIC	Periodic(=周期的な) アクティビティ。メインのアクティビティは一定時間ごとに繰り返し処理されます。リアルタイム OS 上では、正確な周期処理が可能です。
SPORADIC	Sporadic(=散発的な) アクティビティ。メインのアクティビティは繰り返し処理は行われますが、その周期は不定となります。例えば、センサの値が変化したときだけ、外部にそれを知らせるコンポーネント等は Sporadic なコンポーネントになるでしょう。
EVENT_DRIVEN	外部からのイベントに応じて処理を行うコンポーネント。この場合の外部とは主に、他のコンポーネントから発行される CORBA オペレーションの事を指します。

このコンポーネントでは、SPORADIC に設定されています。

RTC_MODULE_MAX_INST:最大インスタンス数

生成できる最大のインスタンス数 (=コンポーネント数) を指定します。STATIC なコンポーネントでは、モジュール初期化時に最大数のインスタンスが生成されます。それ以外のコンポーネントでは、この数値以上のインスタンスを生成することは出来ません。このコンポーネントはインスタンス化可能な最大数は 10 に設定されています。

RTC_MODULE_LANG:モジュール記述言語名とその型

ここには、このコンポーネントを記述するプログラミング言語およびその言語の型を指定します。

●モジュール記述言語と型

モジュール記述言語名	‘‘C++’’, ‘‘Python’’, ‘‘Ruby’’, ‘‘Tcl’’
モジュール記述言語型	‘‘COMPILE’’, ‘‘SCRIPT’’

現在サポートしている言語は、C++と Python のみです。モジュール記述言語型はそれぞれ、C++ であれば “COMPILE”, Python であれば “SCRIPT” を指定します。

これらコンポーネントのプロファイルの指定の仕方は、今後変更される可能性があります。XML 形式のプロファイル記述ファイルに記述し読み込むことでプロファイルを設定する方法が検討されています。

3.3.2 ソースファイル

次に、処理の本体と、ファクトリ関数、初期化関数の実装のソースコードを作成します。`HelloRTWorld.h` で定義したクラスのメソッドを実装してゆきます。

では、ソースコードを見てみましょう。

HelloRTWorld.cpp

```
#include "HelloRTWorld.h"
#include <iostream>

using namespace std;

HelloRTWorld::HelloRTWorld(RtcManager* manager)
  : RtcBase(manager)

{
}

RtmRes HelloRTWorld::rtc_active_do()
{
  std::cout << "Hello RT World!!!" << std::endl;
  return RTM_OK;
}

extern "C" {

  RtcBase* HelloRTWorldNew(RtcManager* manager)
  {
    return new HelloRTWorld(manager);
  }


  void HelloRTWorldDelete(RtcBase* p)
  {
    delete (HelloRTWorld *)p;
    return;
  }


  void HelloRTWorldInit(RtcManager* manager)
  {
    RtcModuleProfile profile(hellortworld_spec);
    manager->registerComponent(profile, HelloRTWorldNew, HelloRTWorldDelete);
  }
};
```

コンストラクタでは、コンポーネントマネージャへのポインタを受け取り、このマネージャのポインタを基底クラス `RtcBase` に渡して RtcBase を初期化しています。`RtcBase` の初期化を忘れずに行ってください。

次に、唯一のメソッド `RtmRes HelloRTWorld::rtc_active_do()` があります。これが RT コンポーネントのメインの処理部分です。

このメインの処理の部分をコンポーネントの**アクティビティ**とよびます。コンポーネントのコンポーネントのアクティビティはいくつかの状態を持ちその状態間を規則にしたがって遷移します。`rtc_active_do()` というメソッドは、コンポーネントが起動し、そのコンポーネント

のアクティビティが ACTIVE 状態にあるときループ実行されます。

コンポーネントのアクティビティの状態には以下のものがあります。

●コンポーネントアクティビティ状態

UNKNOWN：	下記に定義される状態以外の状態。通常この状態には成り得ない。
BORN：	インスタンス生成中の状態。
INITIALIZING：	内部状態を初期化する初期状態。
READY：	活動し外部からのオペレーションを受け付ける状態。
STARTING：	READY 状態から ACTIVE 状態へ移行するときに通過する過渡状態。
ACTIVE：	活性状態となりメインとなる処理を実行する状態。
STOPPING：	ACTIVE 状態から READY 状態へ移行するときに通過する過渡状態。
ABORTING：	活性状態時にエラーを捕捉し ERROR 状態へと移行する過渡状態。
ERROR：	復帰可能なエラー状態。
FATAL_ERROR：	致命的エラー状態。復帰することはできない。
EXITING：	コンポーネント終了状態。リソースの解放等を行う。

これらのアクティビティの状態は図 3.7 の状態遷移図に従って遷移します。

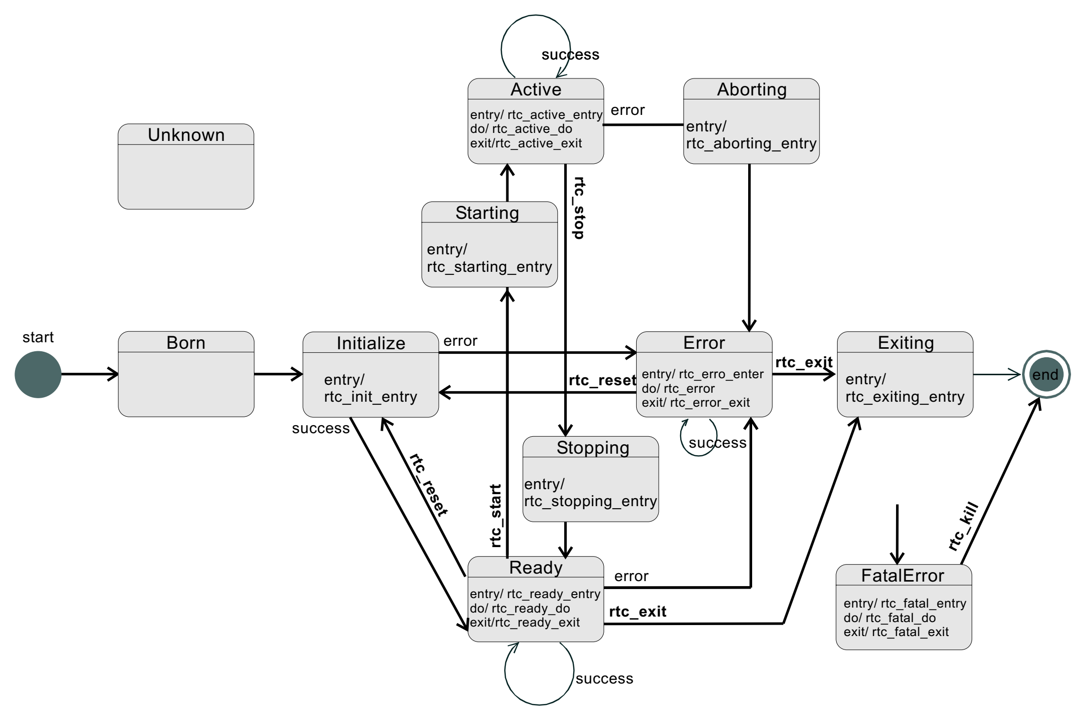


図 3.7: RT コンポーネント状態遷移図

それぞれの状態において、対応するメソッドが実行されます。たとえば、RTC_READY 状態であれば rtc_ready_???() が実行されます。??? の部分はその状態のどのタイミングで実行されるかを示しています。

entry はその状態に入るときに一度だけ実行されます。exit はその状態から出るときに一度だけ実行されます。do はその状態にいる間周期実行されます。

●状態遷移詳細

entry	：その状態に進入するときに一度だけ実行される。
do	：その状態にある間繰り返し実行される。
exit	：その状態から出るときに実行される。

これらの状態は大きく分けて、定常状態と過渡状態に分けられます。

●定常状態と過渡状態

定常状態	READY, ACTIVE, ERROR, FATAL_ERROR
過渡状態	BORN, INITIALIZING, STARTING STOPPING, ABORTING, EXITING
その他	UNKNOWN

定常状態は、一旦その状態に入ると、何らかのアクション (活性化する、非活性化する、再初期化する、強制終了する) があるまで、その状態に留まります。その間、rtc_xxx_do() をループ実行し続けます。定常状態では、その状態から出るときと入るときに rtc_xxx_entry(), rtc_xxx_exit() が実行されます。

これに対して過渡状態は、その状態に入るとき rtc_xxx_entry() を一度だけ処理し、すぐに次の状態に遷移します。したがって、過渡状態に対応するメソッドには rtc_xxx_do() と rtc_xxx_exit() がありません。

RT コンポーネントの基底クラス、RtcBase では上記の状態それぞれの場合に処理するメソッドが virtual 定義されています。したがって、コンポーネントデベロッパは、自分作成するコンポーネントに必要だと思う状態についての処理を、対応するメソッドを再定義することで、それぞれのタイミングで処理を実行させることができます。

基底クラス、RtcBase に定義されているメソッドは以下のとおりです。

RtcBase
+ virtual RtmRes rtc_init_entry()
+ virtual RtmRes rtc_ready_entry()
+ virtual RtmRes rtc_ready_do()
+ virtual RtmRes rtc_ready_exit()
+ virtual RtmRes rtc_starting_entry()
+ virtual RtmRes rtc_active_entry()
+ virtual RtmRes rtc_active_do()
+ virtual RtmRes rtc_active_exit()
+ virtual RtmRes rtc_stopping_entry()
+ virtual RtmRes rtc_aborting_entry()
+ virtual RtmRes rtc_error_entry()
+ virtual RtmRes rtc_error_do()
+ virtual RtmRes rtc_error_exit()
+ virtual RtmRes rtc_fatal_entry()
+ virtual RtmRes rtc_fatal_do()
+ virtual RtmRes rtc_fatal_exit()
+ virtual RtmRes rtc_exiting_entry()

この、HelloRTWorld の例では、単にコンポーネントが活性状態の時にコンソールに”HelloRTWorld!!!” と表示し続けるだけの処理なので、rtc_active_do() に

HelloRTWorld::rtc_active_do()

```
RtmRes HelloRTWorld::rtc_active_do()
{
  std::cout << "Hello RT World!!!" << std::endl;
  return RTM_OK;
}
```

と記述します。ここで、戻り値 RtmRes (RTM result) が RTM_OK となっています。

RtmRes には、RTM_OK，RTM_ERR，RTM_WARNING，RTM_FATAL_ERR の 4 種類の値があり、意味は以下のようになっています。

RtmRes

RTM_OK	正常終了。現在の状態を保持する。
RTM_ERR	エラー終了。Error 状態へ移行する。Active 状態のときは Aborting を通過してから Error 状態へ遷移する。
RTM_WARNING	警告状態。何らかの異常があったことを呼び出し側に伝えるが、現在の状態を保持する。
RTM_FATAL_ERR	致命的エラー終了。FatalError 状態へ移行する。Active 状態のときは Aborting を通過してから Error 状態へ遷移する。

このように、戻り値 RtmRes は値によりコンポーネントの状態を変化させる効果を持ちます。ここでは、常に RTM_OK を返します。

3.3.3 コンポーネント実行ファイル

上記のコンポーネントを単体の実行ファイルとしてビルドするための main 関数を含むソースコードを作成します。ここでは、コンポーネントマネージャを作成し、このマネージャ上でコンポーネントを生成・アクティブ状態にします。ソースコード HelloRTWorldComp.cpp は次のようになります。

main 関数では、マネージャオブジェクトを生成し、初期化、アクティブ化、コンポーネントの生成、マネージャの実行を行っています。

マネージャはコンストラクタの引数としてコマンドライン引数をとるので、main のコマンドライン引数をそのまま与えています。

initManager() でマネージャの初期化、activateManager でアクティブ化とマネージャのネーミングサービスへの登録を行います。

initModuleProc はマネージャへのポインタを引数として取る関数へのポインタを与えています。この与えられた関数をマネージャは適切なタイミングで実行します。最後に runManager() でマネージャが起動します。

MyModuleInit() では、コンポーネントの初期化関数 HelloRTWorldInit() を呼び出しコンポーネントを初期化しています。その後、マネージャ上で HelloRTWorld コンポーネントを作成するために createComponent() を呼び出し、コンポーネントのインスタンスを 1 つ生成しています。このとき、 createComponent() の第 1 引数には、コンポーネントのモジュール名 (ここでは HelloRTWorld)、第 2 引数にはカテゴリ名を与えています。

最後に、このコンポーネントに対して rtc_start() を呼び出し、コンポーネントの状態を ACTIVE 状態にしています。usleep(5000) はコンポーネントが生成され READY 状態になるまで待つためのウェイトです。この例はサンプルなので、コンポーネントの生成、アクティブ化までを MyModuleInit() にて行っていますが、通常は外部から何らかの方法 (rtc-link を用いる等) で、生成・アクティブ化を行います。

HelloRTWorldComp.cpp

```
#include <rtm/RtcManager.h>
#include <string>
#include "HelloRTWorld.h"


void MyModuleInit(RtcManager* manager)
{
  HelloRTWorldInit(manager);

  std::string name;
  RtcBase* comp;
  comp = manager->createComponent("HelloRTWorld", "Generic", name);
  usleep(5000);
  comp->rtc_start();

}
```

```
int main (int argc, char** argv)
{
  RTM::RtcManager manager(argc, argv);
  // Initialize manager
  manager.initManager();
  // Activate manager and register to naming service
  manager.activateManager();
  // Initialize my module on this maneger
  manager.initModuleProc(MyModuleInit);
  // Main loop
  manager.runManager();
  return 0;
}
```

3.3.4 Makefile の作成とビルド

上記の HelloWorld.cpp，HelloWorld.h，HelloWorldComp.cpp の 3 つのソースを合わせて、スタンドアロンコンポーネントとローダブルモジュールをビルドします。コンポーネントをビルドするための Makefile はおおよそ以下のようになります。

CXX_FLAGS および LD_FLAGS は OpenRTM をビルドした環境により異なります。ここでは、omniORB を使用して OpenRTM-aist がビルドされたものとしてそれぞれ、下記のようなコンパイル・リンクオプションとなっていますが、ACE、omniORB、boost 等の依存パッケージのインストール先によりこれらのオプションは変わってきます。自分の環境に合わせた適切なオプションを指定する必要があります。後の例ではここで示す例より、より汎用的な方法について解説しますが、ここではこういったコンパイル・リンクオプションが必要なことを理解してください。

Makefile

```
CXXFLAGS = -I/usr/local/include -I/usr/local/include/rtm/idl
LDFLAGS  = -L/usr/local/lib -lpthread -lACE -lboost_regex -lomniORB4 \
           -lomnithread -lomniDynamic4 -lRTC
SHFLAGS  = -shared
.SUFFIXES: .cpp .o .so

all: HelloRTWorldComp HelloRTWorld.so

.cpp.o:
rm -f $@
$(CXX) $(CXXFLAGS) -c -o $@ $<

.o.so:
rm -f $@
$(CXX) $(SHFLAGS) -o $@ $< $(LDFLAGS)

HelloRTWorldComp: HelloRTWorld.o HelloRTWorldComp.o
$(CXX) -o $@ HelloRTWorld.o HelloRTWorldComp.o $(LDFLAGS)

clean:
rm -f *~ *.o *.so *Comp
```

なお、この Makefile では .o.so ルールも指定されており、これによりローダブルモジュール

コンポーネント `HelloWorld.so` も同時に生成するようになっています。このコンポーネントは、OpenRTM-aist 付属のコンポーネントサーバ `rtcd` にロードして使用することができます。詳細については各種ツールの使用法を参照してください。

3.3.5 実行・テスト

上記の Makefile でソースをビルドして、実行ファイル `HelloWorldComp` を作成します。この実行ファイルを `-f rtc.conf` などのコンポーネント設定ファイルを引数として与えて実行します。実行すると図 3.8 のようになります。

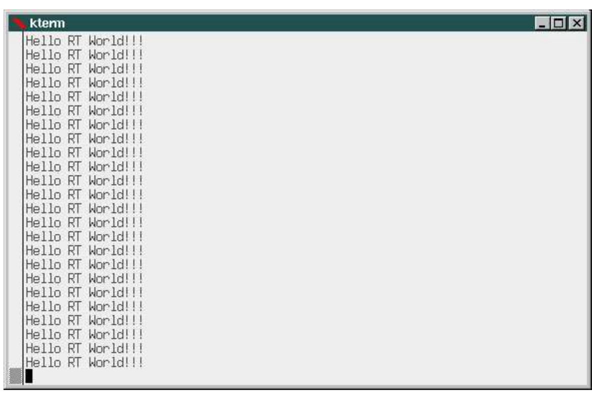


図 3.8: HelloRTWorldComp の実行

3.4 rtc-template を使用したコンポーネントの作成

これまで見てきたように、コンポーネント化するためのコードというのはほとんど決まりきったものになります。OpenRTM-aist にはこれらのコードを自動的に生成するコードジェネレータ rtc-template が付属しています。ここからは、この rtc-template を使用してコンポーネントを作成する方法を説明します。

このコマンドは、コンポーネントの名前やタイプ、InPort/OutPort の名前や型を与えてやると、RT コンポーネントの C++(や Python) のソースコードの雛形を出力します。コンポーネントデベロッパは、出力された雛形のクラスの各アクティビティに対応するメソッドに行いたい処理を追加するだけで、コンポーネントの挙動を記述することができます。

3.4.1 コンポーネントの仕様を決める

まずはじめに、作成したいコンポーネントの仕様を決定します。コンポーネントを作成するために以下の項目をあらかじめ決める必要があります。

- コンポーネントモジュール名
- コンポーネントカテゴリ名
- コンポーネントタイプ
- コンポーネントのアクティビティのタイプ
- InPort の数、名前、変数型
- OutPort の数、名前、変数型

「コンポーネントモジュール名」、「コンポーネントカテゴリ名」、「コンポーネントタイプ」、「コンポーネントのアクティビティのタイプ」はそれぞれ HelloRTWorld サンプルのヘッダにおいて RtcModuleProfile 構造体に指定した、RTC_MODULE_NAME, RTC_MODULE_CATEGORY, RTC_MODULE_COMP_TYPE, RTC_MODULE_ACT_TYPE に対応します。

コンポーネントモジュール名

コンポーネント名は、他のアプリケーション、コンポーネント等からそのコンポーネントを探すときに使用します。名前は、他のコンポーネントと衝突しないように、できるだけ具体的な名前をつけたほうがよいでしょう。名前の付け方については特にガイドラインなどはありませんが、ユーザが増加し何らかのガイドラインを設ける必要が発生した場合には、以後のバージョンにおいてガイドラインを設ける可能性もあります。

● 例: コンポーネントモジュール名

PA10	マニピュレータ
NittaFTSensor	力トルクセンサ
ForceControllerForPA10	PA10 を力制御するコントローラコンポーネント
:	:

コンポーネントカテゴリ名

コンポーネントのカテゴリをあらわす名前を指定します。名前は、そのコンポーネントのカテゴリを表すのに適切な名前をつけたほうがよいでしょう。名前の付け方については特にガイドラインなどはありませんが、ユーザが増加し何らかのガイドラインを設ける必要が発生した場合には、以後のバージョンにおいてガイドラインを設ける可能性もあります。

● 例: コンポーネントカテゴリ名

Manipulator	マニピュレータ
FTSensor	力トルクセンサ
Controller	コントローラ
:	:

コンポーネントタイプ

コンポーネントのタイプとは、生成されるコンポーネントのインスタンスの形式を指します。コンポーネントタイプには以下のものがあります。

● コンポーネントタイプ

STATIC	コンポーネントはマネージャに登録されると同時にインスタンス化され、新たに生成することはできない。ハードウエアに密接に関係するコンポーネント等はこのタイプにするとハードウエアとコンポーネントの対応がとりやすい。
UNIQUE	コンポーネントは動的に生成・削除するができるが、component0 と component1 は異なる内部状態を持ち交換可能ではない。
COMMUTATIVE	コンポーネントは、互いに交換可能。ソフトウエアのロジックのみのコンポーネントはこのタイプになる。

コンポーネントのアクティビティのタイプ

コンポーネントの内部の活動の形式により以下のタイプがあります。

● コンポーネントアクティビティタイプ

PERIODIC	コンポーネントの活動は一定周期で行われます。ただし、動作周期を守れるか否かは、OS に依存します。リアルタイム OS(ART-LINUX) を使用すれば一定周期動作を行わせることはできますが、非リアルタイム OS では厳密な周期動作をさせることは不可能です。
SPORADIC	コンポーネントの活動の周期は一定ではないが、繰り返し行われます。
EVENT_DRIVEN	外部からのオペレーションにより受動的に動作します。

InPort の数、名前、変数型

InPort とは外部からのデータを受け取る口に当たります。コンポーネントは複数の InPort を持つことができます。InPort には名前をつけ、外部のアプリケーションやコンポーネントはその名前により InPort を特定します。

名前

● 例: InPort の名前

"velocity"	速度
"reference"	目標値
"position"	位置

変数型

InPort には幾つかの決まった変数型のものがあらかじめ用意されています。型の種類は値をひとつしか保持しないものと、配列として複数の値を保持できるシーケンス型があります。

● InPort/OutPort の変数型

TimedShort	タイムスタンプつき符号付 Short Int 型
TimedLong	タイムスタンプつき符号付 Long Int 型
TimedUShort	タイムスタンプつき符号なし Short Int 型
TimedULong	タイムスタンプつき符号なし Long Int 型
TimedFloat	タイムスタンプつき Float 型
TimedDouble	タイムスタンプつき Double 型
TimedChar	タイムスタンプつき Char 型
TimedBoolean	タイムスタンプつき Boolean 型
TimedOctet	タイムスタンプつき Octet 型
TimedString	タイムスタンプつき String 型

● **InPort/OutPort の変数型 (シーケンス)**

TimedShortSeq	タイムスタンプつき符号付 Short Int シーケンス型
TimedLongSeq	タイムスタンプつき符号付 Long Int シーケンス型
TimedUShortSeq	タイムスタンプつき符号なし Short Int シーケンス型
TimedULongSeq	タイムスタンプつき符号なし Long Int シーケンス型
TimedFloatSeq	タイムスタンプつき Float シーケンス型
TimedDoubleSeq	タイムスタンプつき Double シーケンス型
TimedCharSeq	タイムスタンプつき Char シーケンス型
TimedBooleanSeq	タイムスタンプつき Boolean シーケンス型
TimedOctetSeq	タイムスタンプつき Octet シーケンス型
TimedStringSeq	タイムスタンプつき String シーケンス型

OpenRTM-aist においてはこれらの型を CORBA 経由で交換可能な型として `RTCDataType.idl` 内で定義しています。ユーザは必要ならば、`RTCDataType.idl` に構造体を追加することで、新たな型を使用することが出来ますが、ここではその詳細は割愛します。。

OutPort の数、名前、変数型

OutPort とは外部からのデータを受け取る口に当たります。コンポーネントは複数の OutPort を持つことができます。OutPort には名前をつけることができ、外部のアプリケーションやコンポーネントはその名前により OutPort を特定します。

名前

● **例: InPort の名前**

"velocity"	速度
"reference"	目標値
"position"	位置

変数型

OutPort には InPort と同様の決まった変数型のものがあらかじめ用意されています。型の種類は InPort と同じで、同じ型同士の InPort と OutPort のみがデータをやり取りできます (3.9)。

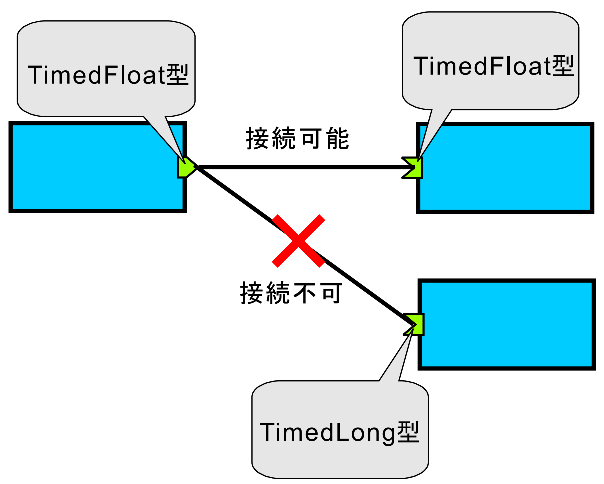


図 3.9: InPort と OutPort の型と接続

3.4.2 rtc-template を使用して雛形を作成する

上記の項目を決めたら、コンポーネントのテンプレートジェネレータ: `rtc-template` を使用してコンポーネントの雛形を作成することができます。

ここでは以下の仮定に基づき、コンポーネントを作成していきます。

アクティビティの処理

マニピュレータ制御用のライブラリが存在し、ライブラリの関数には以下のようなものがあると仮定します。

●例: マニピュレータ制御用ライブラリ関数の例

manipulator_init()	:	マニピュレータの初期化
manipulator_on()	:	マニピュレータのサーボを ON にする
manipualtor_off()	:	マニピュレータのサーボを OFF にする
manipulator_setpos()	:	マニピュレータの手先の目標値を設定する
manipulator_getpos()	:	マニピュレータの手先の位置姿勢を取得する
manipulator_emgstop()	:	マニピュレータを緊急停止させる
manipulator_destroy()	:	マニピュレータのリソースを開放する

●コンポーネントの仕様

コンポーネントモジュール名	MyManipulator
コンポーネントカテゴリ名	Manipulator
コンポーネントタイプ	STATIC
コンポーネントアクティビティタイプ	PERIODIC
InPort の数	1 個: 目標値を受け取る
InPort の名前	"pos"
InPort の変数型	TimedFloatSeq
OutPort の数	1 個: 現在位置を出力する
OutPort の名前	"pos"
OutPort の変数型	TimedFloatSeq

rtc-template で雛形を作成

コンポーネントを作成するデイレクトリ (任意のディレクトリで結構です。) を作成して、そこでコンポーネントを作成します。

```
> cd
> mkdir MyManipulator
> cd MyManipulator
```

まずは help を見てみます。

```
> rtc-template --help

Usage: rtc-template [OPTIONS]
Options:
    [--help]                              Print this help.
    [--c++]                               Generate C++ template code.
    [--python]                            Generate Python tempalte code.
    [--output[=output_file]]              Output base file name.
    [--module-name[=name]]                Your module name.
    [--module-desc[=description]]         Module description.
    [--module-version[=version]]          Module version.
    [--module-author[=author]]            Module author.
    [--module-category[=category]]        Module category.
    [--module-comp-type[=component_type]] Component type.
    [--module-act-type[=activity_type]]   Component's activity type.
    [--module-max-inst[=max_instance]]    Number of maximum instance.
    [--module-lang[=language]]            Language.
    [--module-inport[=PortName:Type]]     InPort's name and tyoe.
    [--module-outport[=PortName:Type]]    OutPort's name and type
       :
     中略
       :
Example:
    rtc-template --c++ --module-name=Sample --module-desc='Sample component' \
    --module-version=0.1 --module-author=DrSample --module-category=Generic \
    --module-comp-type=COMMUTATIVE --module-act-type=SPORADIC \
    --module-max-inst=10  \
    --module-inport=Ref:TimedFloat --module-inport=Sens:TimedFloat \
    --module-outport=Ctrl:TimedDouble --module-outport=Monitor:TimedShort
```

rtc-template に作成したいコンポーネントの設定を引数として渡すと、コンポーネントの雛形のソースコードを作成します。ここでは、上記の仕様にしたがって、オプションを指定し雛形を作成します。

```
> rtc-template --c++ --module-name=MyManipulator \
    --module-desc='My simple manipulator' \
    --module-version=0.1 --module-author=MyName --module-category=Manipulator \
    --module-comp-type=STATIC --module-act-type=PERIODIC \
    --module-max-inst=1  \
    --module-inport=posin:TimedFloatSeq --module-outport=posout:TimedFloatSeq
MyManipulator.h was generated.
MyManipulator.cpp was generated.
MyManipulatorComp.cpp was generated.
Makefile.MyManipulator was generated.
> ls
Makefile.MyManipulator  MyManipulator.h       MyManipulatorComp.cpp
MyManipulator.cpp       MyManipulator.h
MyManipulator.cpp       MyManipulatorComp.cpp
```

このように、コンポーネントの C++のコードと Makefile が作成されます。ここで、

```
> make -f Makefile.MyManipulator
もしくは
> mv Makefile.MyManipulator Makefile
```

として make してみます。

```
> make
rm -f MyManipulator.o
g++ `rtm-config --cflags` -c -o MyManipulator.o MyManipulator.cpp
     :
    中略
     :
g++ `rtm-config --libs` -o MyManipulatorComp MyManipulator.o MyManipulatorComp.o
rm -f MyManipulator.so
g++ -shared `rtm-config --libs` -o MyManipulator.so MyManipulator.o
> ls
Makefile.MyManipulator  MyManipulator.o     MyManipulatorComp.cpp
MyManipulator.cpp       MyManipulator.so*   MyManipulatorComp.o
MyManipulator.h         MyManipulatorComp*
```

これで、ローダブルモジュール版のコンポーネント (MyManipulator.so) と実行形式のコンポーネント (MyManipulatorComp) が作成されました。しかし、まだアクティビティに何も記述していないので、このコンポーネントは何も仕事をしません。コンポーネントに行わせたい処理を作成された雛形に埋め込んでいくことで、コンポーネントを作成していきます。

3.4.3 コンポーネントのソースコードを見てみる

rtc-template によって生成されたコンポーネントのソースコードを見てみてください。HelloRTWorld コンポーネントで作成したのとほぼ同じで、コンポーネント名、InPort/OutPort の部分が指定したもので置き換えられたソースコードになっています。

● rtc-template により生成されるファイル

MyManipulator.h	MyManipulator コンポーネントのヘッダファイル。
MyManipulator.cpp	MyManipulator コンポーネントのソースファイル。
MyManipulatorComp.cpp	MyManipulator コンポーネントのスタンドアロンコンポーネントのためのソースファイル。
Makefile	ビルドするための Makefile。

たとえば、ヘッダ MyManipulator.h には以下の記述があるはずです。

MyManipulator.h

```
class MyManipulator
  : public RTM::RtcBase
{
 public:
  MyManipulator(RtcManager* manager);

  //  virtual RtmRes rtc_ready_entry();
  //  virtual RtmRes rtc_ready_do();
  //  virtual RtmRes rtc_ready_exit();
  //  virtual RtmRes rtc_active_entry();
  virtual RtmRes rtc_active_do();
  //  virtual RtmRes rtc_active_exit();
  //  virtual RtmRes rtc_error_entry();
  //  virtual RtmRes rtc_error_do();
  //  virtual RtmRes rtc_error_exit();
  //  virtual RtmRes rtc_fatal_entry();
  //  virtual RtmRes rtc_fatal_do();
  //  virtual RtmRes rtc_fatal_exit();
  //  virtual RtmRes rtc_init_entry();
  //  virtual RtmRes rtc_starting_entry();
  //  virtual RtmRes rtc_stopping_entry();
  //  virtual RtmRes rtc_aborting_entry();
  //  virtual RtmRes rtc_exiting_entry();

  TimedFloatSeq m_posin;
  InPortAny<TimedFloatSeq> m_posIn;
  TimedFloatSeq m_posout;
  OutPortAny<TimedFloatSeq> m_posOut;

};
```

たくさん並んでいる rtc_xxx_yyy() というメソッドは状態に対応するメソッドです。各状態遷移については、図 3.7 (p.33) の状態遷移図および説明を参照してください。

3.4.4 使用するアクティビティを決める

さて、図 3.7 の状態遷移図を念頭において、アクティビティにおいて行う処理を決めていきます。埋め込むライブラリの動作と状態の意味を考えながら、各状態で実際行いたい処理を割り当ててください。INITIALIZING 状態では初期化、STARTING 状態では ACTIVE 状態に入る直

前に行う処理、ACTIVE 状態ではメインとなる周期処理、STOPPING 状態では ACTIVE 状態から待機 (READY) 状態へ移行するときに行うべき処理、といった形で対応させてゆきます。

これから作成する MyManipulator コンポーネントは各状態で以下のような処理を実行するようにします。

● 処理の内容

rtc_init_entry()	ライブラリの初期化処理を行うために manipulator_init() を呼び出す。 manipulator_init() は初期化が成功したかどうかを戻り値 (bool 型) で知らせるものと仮定する。
rtc_starting_entry()	マニピュレータのサーボを ON にするために、manipulator_on() を呼び出す。 manipulator_on() はサーボ ON の処理が成功したかどうかを戻り値 (bool 型) で知らせるものと仮定する。
rtc_stopping_entry()	マニピュレータのサーボを OFF にするために、manipulator_off() を呼び出す。 manipulator_off() はサーボ OFF の処理が成功したかどうかを戻り値 (bool 型) で知らせるものと仮定する。
rtc_active_do()	マニピュレータに InPort から入ってきた位置指令を manipulator_setpos() で設定し、現在位置を manipulator_getpos() で取得し OutPort に出力する。
rtc_aborting_entry()	RTC_ACTIVE 状態で何らかのエラーが生じ緊急停止をさせるため manipulator_emgstop() を呼び出す。
rtc_exiting_entry()	コンポーネントの終了処理。リソースを開放するために manipulator_destroy() を呼び出す。

3.4.5 実装

以上のように各状態で行うことを決定したら、それぞれの状態に対応するメソッドを実装していきます。

安全なコンポーネントを作成するためには、エラーを適切に処理することが大変重要となります。どのような場合に ERROR 状態に遷移させ、どのような場合には FATAL_ERR 状態に遷移させるかをよく考えて実装しましょう。

状態: RTC_INITIALIZING, メソッド: rtc_init_entry() の実装

MyManipulator.h

```
virtual RtmRes rtc_init_entry();
```

出力された雛形ではコメントをはずすだけです。

MyManipulator.cpp

```
RtmRes MyComponent::rtc_init_entry()
{
  if (!manipulator_init())
  {
    return RTM_ERR;
  }
  return RTM_OK;
}
```

ここでは、manipulator_init() の戻り値を見て、true であれば、RTM_OK を返しています。状態に対応するメソッドでは、戻り値 RtmRes に以下の値をとります。

● 終了状態と戻り値

RTM_OK：	正常終了
RTM_ERR：	エラー終了
RTM_FATAL_ERR：	致命的エラー終了

正常終了では、そのまま次の状態へ遷移します。この場合では、 RTC_READY 状態に遷移します。エラー終了するとエラー状態に入ります。エラー状態では、外部から再度初期化することにより RTC_INITIALIZE 状態に戻ることができます。致命的エラー終了の場合は、復帰できません。逆に、復帰不可能なエラーとして処理したい場合には、致命的エラー状態へ遷移させます。

状態: RTC_STARTING, メソッド rtc_starting_entry() の実装

MyManipulator.h

```
virtual RtmRes rtc_starting_entry();
```

出力された雛形ではコメントをはずすだけです。

MyManipulator.cpp

```
RtmRes MyComponent::rtc_starting_entry()
{
  if (!manipulator_on())
  {
    return RTM_ERR;
  }
  return RTM_OK;
}
```

manipulator_on() の戻り値に応じて RTM_OK または RTM_ERR を返します。

状態: RTC_STOPPING, メソッド: rtc_stopping_entry() の実装

MyManipulator.h

```
virtual RtmRes rtc_stopping_entry();
```

出力された雛形ではコメントをはずすだけです。

MyManipulator.cpp

```
RtmRes MyComponent::rtc_stopping_entry()
{
  if (!manipulator_off())
  {
    return RTM_ERR;
  }
  return RTM_OK;
}
```

manipulator_off() の戻り値に応じて RTM_OK または RTM_ERR を返します。

状態: RTC_ACTIVE, メソッド: rtc_active_do() の実装

MyManipulator.h

```
virtual RtmRes rtc_active_do();
```

出力された雛形ではコメントをはずすだけです。

次に rtc_active_do() を実装していきます。

rtc_active_do() ではマニピュレータに InPort から入ってきた位置指令を manipulator_setpos() で設定し、現在位置を manipulator_getpos() で取得し OutPort に出力します。InPort は m_posIn.read() によりデータを読み込みます。

MyManipulator.cpp

```
RtmRes MyComponent::rtc_active_do()
{
  float pos_in[6];
  float pos_out[6];

  // InPort からデータ読み込み
  m_posin = m_posIn.read();

  // InPort に規定数のデータが入っているか確認
  if (m_posin.data.length() == 6)
  {
    for (int i = 0; i < 6; i++)
    {
       pos_in[i] = m_posin.data[i];
    }
    // 位置データをセット
    if (!manipulator_setpos(pos_in))
    {
      return RTM_ERR;
    }
  }

  // 現在位置を取得
  if (!manipulator_getpos(pos_out))
  {
    return RTM_ERR+
  }

  // OutPort に出力
  m_posout.data.length(6);
  for (int i = 0; i < 6; i++)
  {
     m_posout.data[i] = pos_out[i];
  }
  m_posOut.write(m_posout);

  return RTM_OK;
}
```

TimedFloatSeq 型 (とその他のシーケンス型) は構造体で、時間 `tm` とデータ `data` のメンバを持ちます。`data` はシーケンス型になっていて、

● InPort/OutPort データ型へのアクセス

`m_poin.data[1]`	添え字によるアクセス
`m_posin.data.length()`	要素数の取得
`m_posin.data.length(10)`	領域の確保

などが行えます。OutPort も同様で、`manipulator_getpos()` でマニピュレータの手先位置を配列として取得したあとに、`m_posout` にデータをコピーし、最後に

```
m_posOut.write(m_posout);
```

により OutPort に出力しています。InPort, OutPort のメソッドの詳細はマニュアルもしくはソースコードを見てください。

状態: RTC_ABORTING, メソッド: rtc_aborting_entry() の実装

MyManipulator.h

```
    virtual RtmRes rtc_aborting_entry();
```

出力された雛形ではコメントをはずすだけです。

MyManipulator.cpp

```
RtmRes MyComponent::rtc_aborting_entry()
{
  if (!manipulator_emgstop())
  {
    return RTM_FATAL_ERR;
  }
  return RTM_OK;
}
```

RTC_ACTIVE 状態でエラーが発生した場合、緊急停止しなければならないと仮定します。そこで、rtc_aborting_entry() で manipulator_emgstop() を呼び出します。戻り値に応じて RTM_OK または RTM_ERR を返します。

状態: RTC_EXITING, メソッド: rtc_exiting_entry() の実装

MyManipulator.h

```
virtual RtmRes rtc_exiting_entry();
```

出力された雛形ではコメントをはずすだけです。

MyManipulator.cpp

```
RtmRes MyComponent::rtc_exiting_entry()
{
  manipulator_destroy();
  return RTM_OK;
}
```

リソースを開放します。この状態では終了する他ないため RTM_OK しか返しません。

3.4.6 make および実行

ソースが完成したら、再び make します。manipulator ライブラリをリンクするために Makefile.MyManipulator を書き換える必要があるかもしれません。

```
> make -f Makefile.MyManipulator
```

実行形式のコンポーネントを実行してみます。コンポーネントの実行にはコンフィギュレーションファイル (通常は rtc.conf という名前) が必要です。正式なコンフィギュレーションファイルは OpenRTM のソースの `etc/rtc.conf.sample` を参考にしてください。ここでは、簡易版のものをカレントディレクトリに作成します。

```
NameServer        現在の PC のホスト名:ポート番号
```

ここで、仮にホスト名：rtm.or.jp ポート番号:6789 とします。

```
> cat > rtc.conf
NameServer        rtm.or.jp:6789 (このように入力する)
\verb|^|D(Ctrl+D)
> cat rtc.conf (確認)
NameServer        rtm.or.jp:6789
```

次に、CORBA のネーミングサービスを起動します。CORBA のネーミングサービスは、

```
> rtm-naming ポート番号
```

で起動できますので、先ほど rtc.conf で指定したポート番号を指定して起動してください。

```
> rtm-naming 6789
Starting omniORB omniNames: ichi:9999
n-ando@ichi:/tmp/SampleComponent>
Fri Oct 29 17:12:51 2004:

Starting omniNames for the first time.
Wrote initial log file.
Read log file successfully.
Root context is IOR:010000002b00000049444c3a6f6d672e6f72672f436f734e616d696e672f
4e616d696e67436f6e746578744578743a312e3000000100000000000000060000000010102000e00
00003135302e32392e39362e313638000f270b0000004e616d6553657276696365000200000000000
000008000000001000000005454410100000001c0000000100000001000100010000000100010509001
01000100000009010100
Checkpointing Phase 1: Prepare.
Checkpointing Phase 2: Commit.
Checkpointing completed.
```

次に、コンポーネントを起動します。このコンポーネントは、位置指令を受け取ることで初めて動作するコンポーネントですので、起動しただけでは何も起こりません。上記と同様の手順で、位置指令を与えるコンポーネントと、現在位置を受け取るコンポーネントなどを作成しそれらを接続することで、システムを構成します。

```
> MyManipulatorComp -f rtc.conf
```

第4章　OpenRTM-aist ツール

本章では、OpenRTM-aist に付属のツールについて説明します。OpenRTM-aist には開発をサポートするためのツール群がいくつか用意されています。まだ、基本的なツールしかありませんが、今後ユーザの要望に沿う形で増やして行く予定です。

4.1　RTCLink

RT コンポーネントを組み合わせてシステムを構築するにはいくつかの方法があります。

- GUI による方法
- XML による方法
- スクリプトによる方法
- コンポーネントからの直接アクセス
- 通常のアプリケーションからの直接アクセス

このうち、もっとも直感的でわかりやすいシステム構築法はこれから説明する GUI(RTCLink) による構築方法です。この方法はコンポーネントの InPort/OutPort をマウスによるドラッグアンドドロップで接続し、GUI からの操作でコンポーネントの ON/OFF を行うことによりシステムを組み立て動作させることが出来ます。直感的でわかりやすく、システムの組み換えも簡単に行えるため、コンポーネントのテスト段階やシステム構築のテスト段階により適している方法です。

4.1.1　動作条件

RTCLink は Python のツールキットのひとつである wxPython で記述されています。したがって、Python と wxPython が動作する環境であればどの OS 上でも動作させることが出来ます。

動作条件は以下のとおりです。

● **RTCLink 動作条件**

Python	Python 2.3 以上
wxPython	wxPython 2.5.1.5u
OpenRTM	aist-0.2.0 以上

4.1.2 起動

RTCLink を起動させると図 4.1 のようなウインドウが現れます。

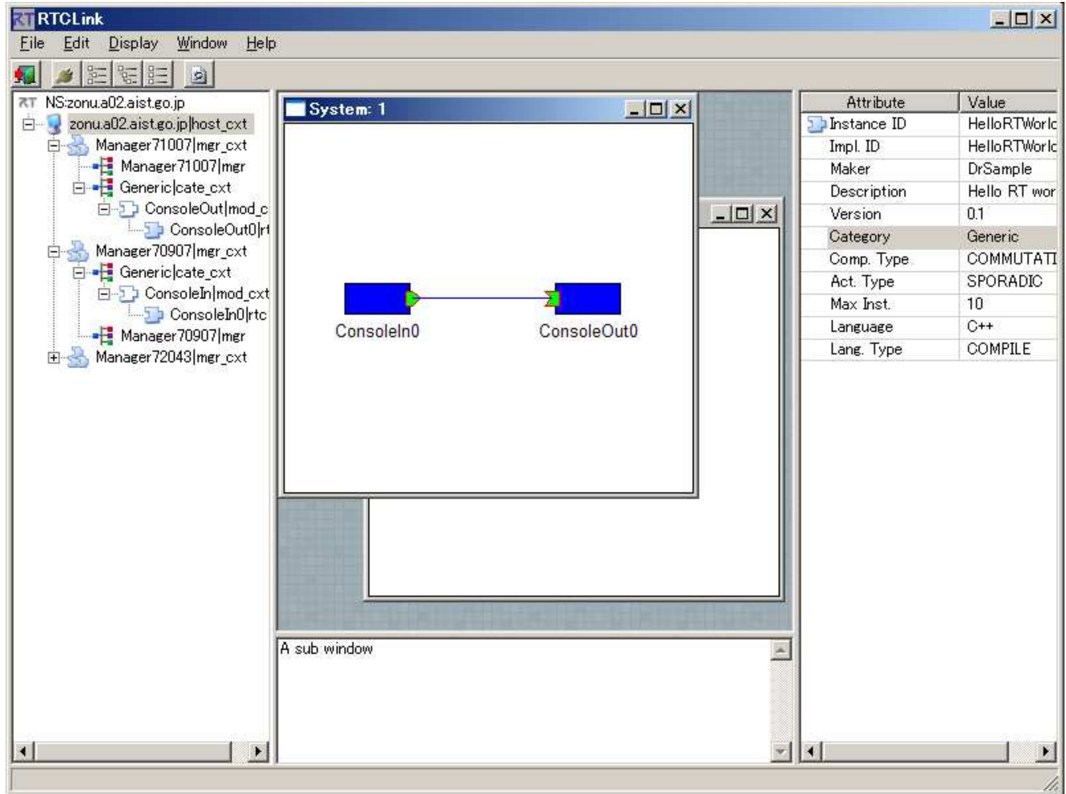


図 4.1: RTCLink

RTCLink は大きく分けて以下の 4 つのウインドウから構成されています。

- ネーミングウインドウ
- ログウインドウ
- プロファイルウインドウ
- システムドローウインドウ

4.1.3 ネーミングウインドウ

左側のウインドウはネーミングサービスに登録されているコンポーネントネームツリーを表示するウインドウです。コンポーネントは起動されると所定のネーミングサービス (設定ファイル：通常は `rtc.conf` に記述されているネーミングサービス) に自分の名前を登録します。ネーミングウインドウではその名前を見ることが出来ます。

名前の登録の仕方には 2 つの種類があります。

● 名前の登録の仕方

ロングネーム　ホストコンテキスト/マネージャコンテキスト/カテゴリコンテキスト/モジュールコンテキスト/コンポーネントからなる階層により定義される名前。一意性が保証され、名前が衝突することはない。デフォルトではこの名前がネームサービスに登録されます。

エイリアス　コンポーネントの別名。ユーザが任意の階層構造で名前を登録することが出来る。もちろん、階層化しない登録も可能。ただし、一意性は保証されないためユーザが名前の衝突を起こさないように考慮する必要がある。もし名前の衝突が発生した場合には、古いオブジェクトは新しいオブジェクトで上書きされる。

エイリアスの登録の仕方は `RtcBase` クラスリファレンス `appendAlias()` の項を参照してください。

4.1.3.1　ネームサーバへの接続

起動時、RTCLink はデフォルトのネームサーバへ接続します。デフォルトのネームサーバは、最後に接続したネームサーバ (履歴ファイルが存在する場合) もしくは、ローカルホスト上のネームサーバになります。

現在接続しているネームサーバはツリーのルートアイテムのラベルとして表示されています。

現在と異なるネームサーバへ接続したい場合には、

● ネームサーバへの接続

ツールバー上の ボタンを押す
メニューバーの「File」-「Connect Name Server」を選択する
ネーミングツリーのルートアイテム上で右クリックし 「Connect」 を選択する

のいずれかの方法でネームサーバ接続ダイアログを表示させます (図 4.2)。

図 4.2: ネームサーバ接続ダイアログ

接続したいネームサーバのアドレスとポート番号を `name_server:port` の形式で入力し OK を押すと接続されます。ポート番号が省略された場合 omniORB のデフォルトのネームサービスポート番号が使用されます。

4.1.3.2 アイテムとコンテキストメニュー

ネーミングツリー上で表示されるアイテムは以下のとおりです。

	ホストコンテキスト
	マネージャコンテキスト
	カテゴリコンテキスト
	モジュールコンテキスト
	コンポーネント

ネーミングツリーのアイテム上で右クリックをすると、アイテム毎のコンテキストメニューが表示されます。

各アイテムにおけるコンテキストメニューは以下のようになっています。

● マネージャコンテキスト (未実装)

Create	コンポーネントの生成
Load	モジュールのロード
Delete	ネーミングコンテキストの削除

● モジュールコンテキスト (未実装)

Create	コンポーネントの生成
Delete	ネーミングコンテキストの削除

● コンポーネント

Start	コンポーネントの起動
Stop	コンポーネントの停止
Reset	コンポーネントのリセット
Exit	コンポーネントの終了
Kill	コンポーネントの強制終了
Delete	ネーミングコンテキストの削除
Profile	コンポーネントプロファイル

4.1.4 ログウインドウ

OpenRTM ではネームサービスのネームスペース内にグローバルなログ収集コンポーネント (GlobalLogger) をトップレベルにおくことが出来ます。さらに、コンポーネントから出力され

るログメッセージをこのログ収集コンポーネントに集めることが出来ます。ログウインドウはこの GlobalLogger に入力されたログメッセージを表示することで、全てのコンポーネントが出力するログメッセージを常時監視することが出来ます。

4.1.5 プロファイルウインドウ

コンポーネントのプロファイルを表示するウインドウです。表示されるプロファイルは以下のとおりです。

Instance ID	コンポーネントインスタンス ID (コンポーネント名)
Implementation ID	コンポーネント実装 ID (モジュール名)
Maker	コンポーネントの作成者
Description	コンポーネントの概要説明
Version	コンポーネントのバージョン
Category	コンポーネントのカテゴリ
Component Type	コンポーネントタイプ
Activity Type	アクティビティタイプ
Max Instance	最大のインスタンス数
Language	コンポーネント記述言語
Language Type	コンポーネント記述言語タイプ
InPort Profile	InPort のプロファイル
OutPort Profile	OutPort のプロファイル

4.1.6 システムドローウインドウ

中央部はシステム構築のためのシステムドローウインドウです。マルチウインドウ (UNIX ではタブウインドウ) 表示となっています。

● システムドローウインドウの表示

- ツールバー上の ボタンを押す
- メニューバーの「File」-「New System」を選択する

システムドローウインドウは、ツールバーの ボタンを押すか、メニューバーの「File」-「New System」を選択することで新規に開くことが出来ます。ここにネーミングウインドウからコンポーネントをドラッグアンドドロップし、システムを組み立ててゆきます。

4.1.7 システムドローウインドウ上でのシステム構築

4.1.7.1 コンポーネントの配置

ネーミングツリーウインドウをクリックしてツリーを展開してゆきます。

ツリー画面上に当該コンポーネント名がない場合は、以下の点を確認してください。

- ネーミングサービスが起動しているかどうか？ (起動及びポート番号の確認)
- コンポーネントが起動しているかどうか？

ツリー画面上のコンポーネント名 () をマウスの左ボタンを押してドラッグし、システムドローウインドウにドロップします。すると、コンポーネントブロックが表示されます。

起動されていないコンポーネントをドラッグ&ドロップした場合には、黒いブロックが表示されます。

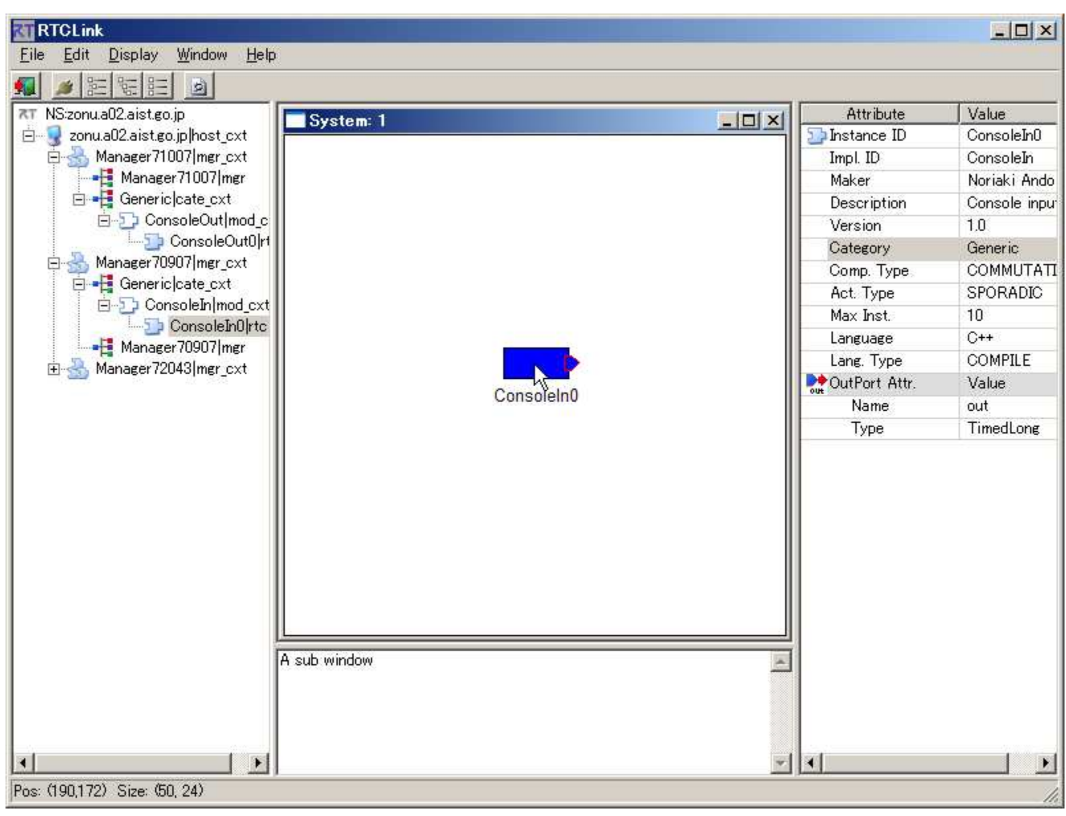


図 4.3: ドラッグアンドドロップによるコンポーネントの配置。 前回のシステム構築時に、コンポーネントを接続したまま RtcLink を終了した場合は、当該コンポーネントを配置した時に、前回の接続状態が再現されます。

4.1.7.2 RT コンポーネントの選択・移動・削除

選択 選択したいコンポーネントをマウスの左ボタンでクリックします (図 4.4)。複数の RT コンポーネントを選択するには [Shift キー] + [マウスの左ボタン] でクリックし連続選択します (図 4.5)。コンポーネントが選択されて選択色になります。

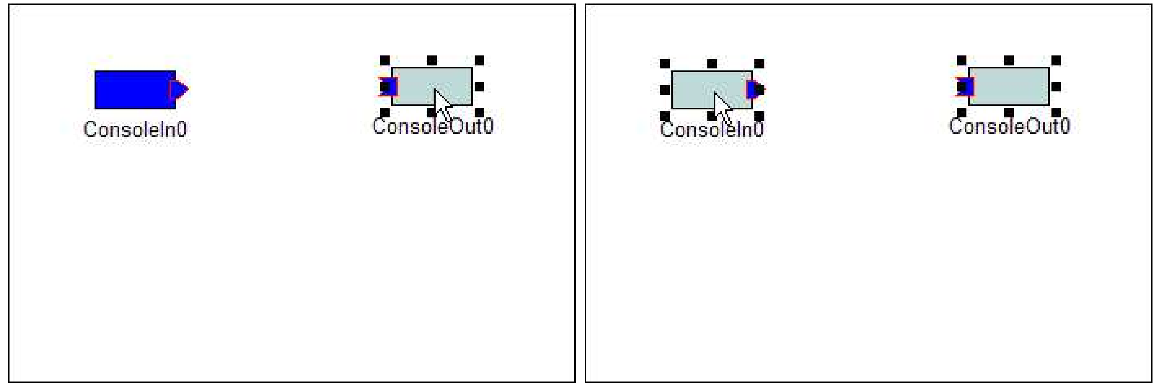


図 4.4: コンポーネントの選択　　図 4.5: 複数コンポーネントの選択

移動 選択色のコンポーネントをマウスの左ボタンでドラッグすると移動することができます(図 4.6)。複数選択している場合は、選択色のコンポーネントのどれかを [マウスの左ボタン] でドラッグすると選択されたすべてのコンポーネントを同時に移動することができます。

選択解除 RT コンポーネント以外の領域で [マウスの左ボタン] をクリックすると、選択が解除されます (図 4.7)。

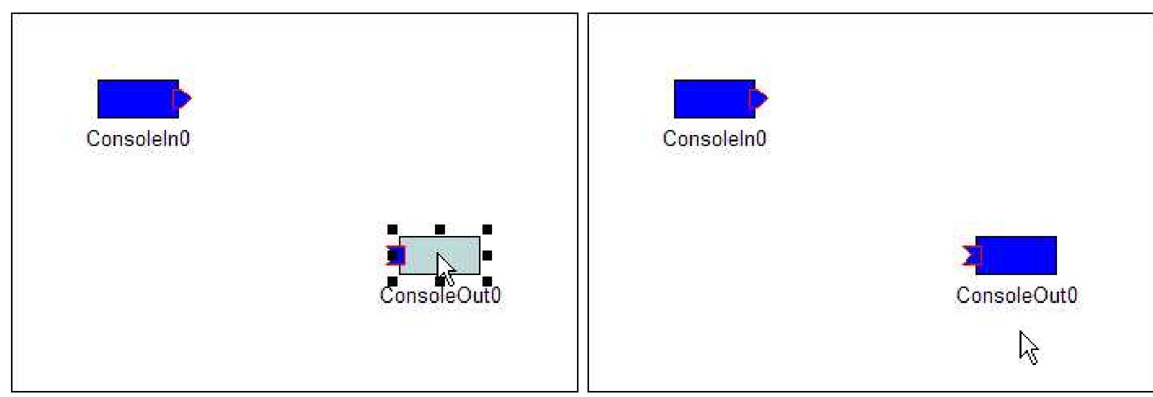


図 4.6: RT コンポーネントの移動　　図 4.7: コンポーネントの選択解除

削除 RT コンポーネント上で [マウスの右ボタン] をクリックするとコンテキストメニューが表示されます。メニュー内の'Delete Item' を選択または Delete キーを押すと、マウスカーソルの下の RT コンポーネントが System 画面上から削除される (図 4.8)。複数のコンポーネントを削除する場合は、最初に削除対象の RT コンポーネントを選択状態にします。次にコンポーネント外の領域で [マウスの右ボタン] をクリックするとコンテキストメニューが表示されますので、メニュー内の'Delete to Selected Item' を選択または Delete キーを押すと選択色のコンポーネントがシステムドローウインドウ上から削除されます。この削除は、システムドローウインドウ上からの削除ですので、コンポーネントそのものが削除されるわけではありません。

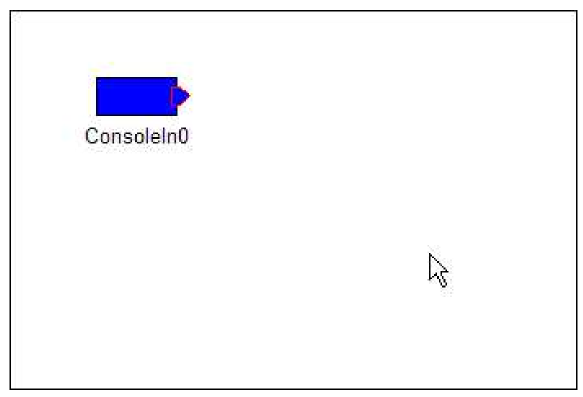


図 4.8: コンポーネントの削除

左右反転 コンポーネントを [マウスの中ボタン] でクリックすると、コンポーネントが左右反転表示され Inport と OutPort の位置が入れ替わります (図 4.9)。

回転 コンポーネントを [Shift キー] + [マウスの中ボタン] でクリックすると、インポートとアウトポートの位置が上下方向に入れ替わる (図 4.10)。

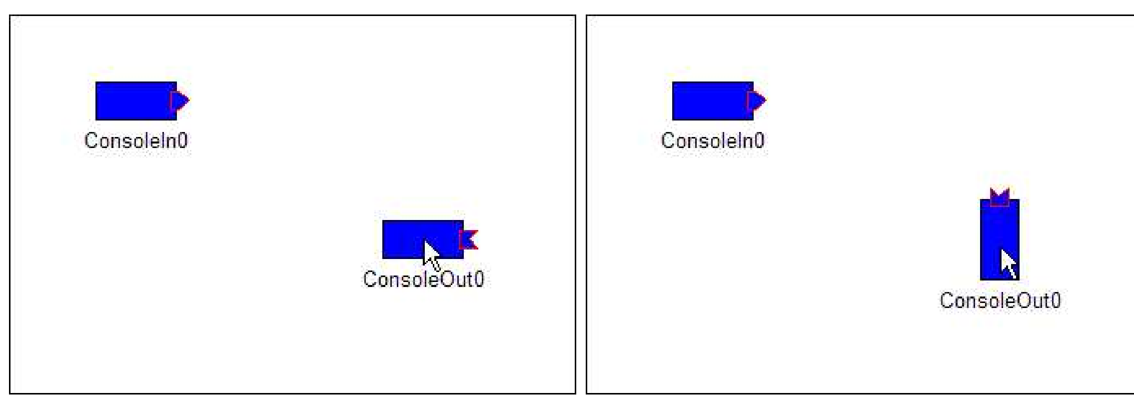


図 4.9: コンポーネントの左右反転　　図 4.10: コンポーネントの回転

拡大縮小 コンポーネントが選択状態の時に表示される四隅の黒い点を [マウスの左ボタン] でドラッグすると拡大縮小を行うことができます (図 4.11)。

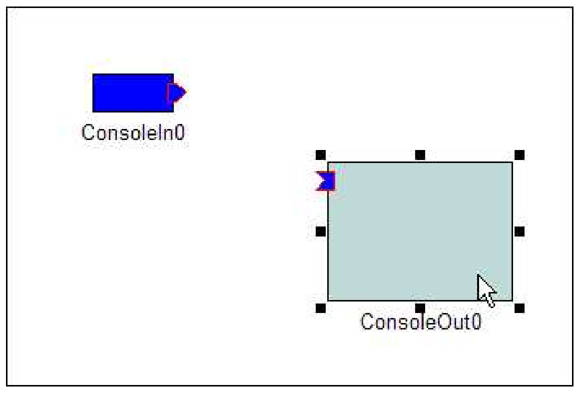


図 4.11: コンポーネントの拡大縮小

4.1.7.3 InPort/OutPort の名前と型情報の表示

InPort/OutPort を [マウスの右ボタン] でクリックすると、ポートの名前と型情報が表示されます (図 4.12)。

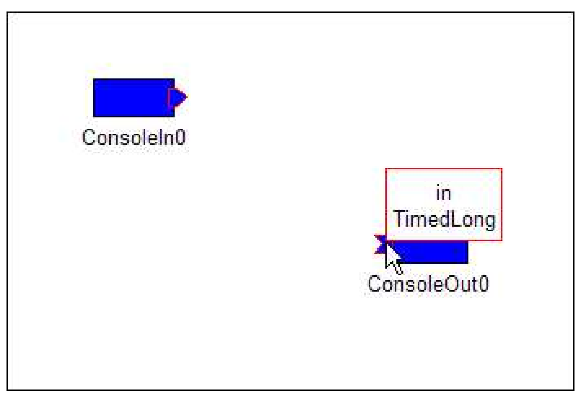


図 4.12: InPort/OutPort の名前と型情報の表示

4.1.7.4 コンポーネントの接続と接続解除

RT コンポーネントを接続する方法は、以下の 2 通りあります。

InPort/OutPort をクリックして接続 接続元のインポートを [マウスの左ボタン] でクリック (図 4.13) した後で、接続先のアウトポートを [マウスの左ボタン] でクリック (図 **??**) する。インポートとアウトポートが線で接続されて接続色になります。

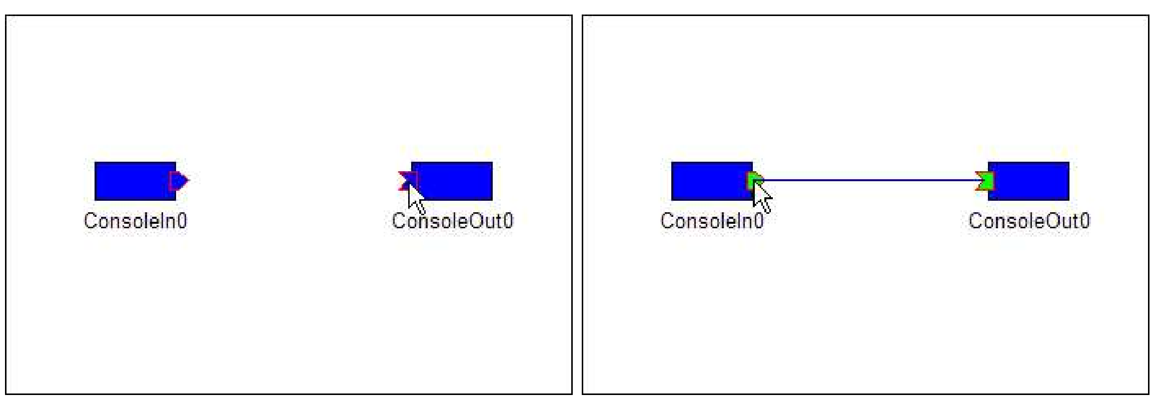


図 4.13: コンポーネントの接続 1(a)　　図 4.14: コンポーネントの接続 1(b)

InPort/OutPort 間をドラッグアンドドロップで接続 接続元のインポートを [マウスの左ボタン] でクリックしたままの状態で、接続先のアウトポートへマウスカーソルをドラッグ (図 4.15) して、ボタンを離します。インポートとアウトポートが線で接続されて接続色になります (図 4.16)。

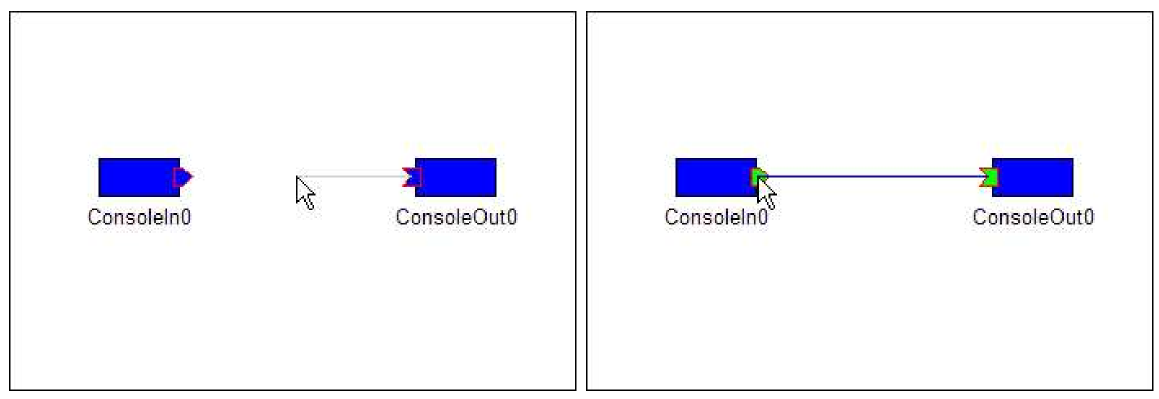


図 4.15: コンポーネントの接続 2(a)　図 4.16: コンポーネントの接続 2(b)

コンポーネントの接続を解除する 接続を解除したい線を [マウスの左ボタン] でクリックし選択 (図 4.17) した後で、[Delete キー] を押下する (図 4.18)。

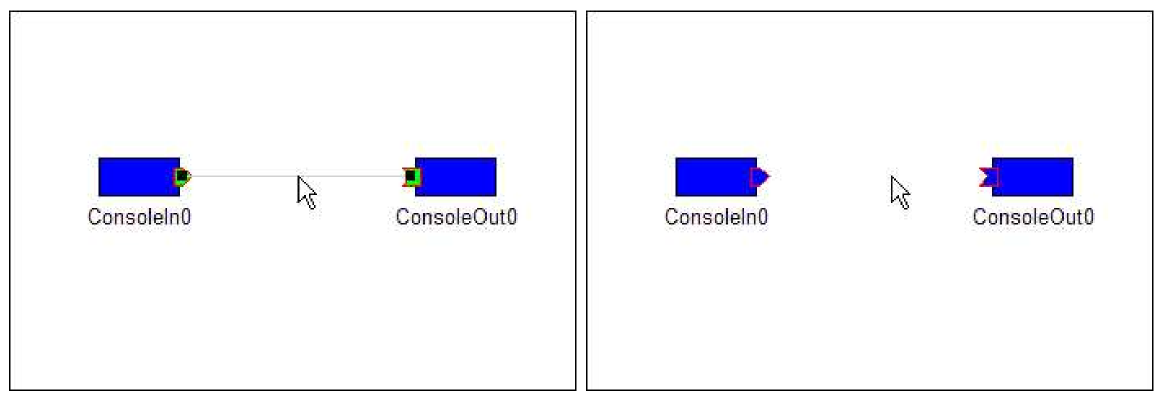


図 4.17: 接続の解除 (a)　図 4.18: 接続の解除 (b)

接続した線を移動する ポートを接続した線が折れ線になっている時は、線を移動することができます。線を [マウスの左ボタン] でクリックすると、移動可能な線の上に赤丸が表示されます。赤丸印を [マウスの左ボタン] でドラッグすると線を移動することができます。水平線は上下方向に、垂直線は左右方向に移動できます (図 4.19)。

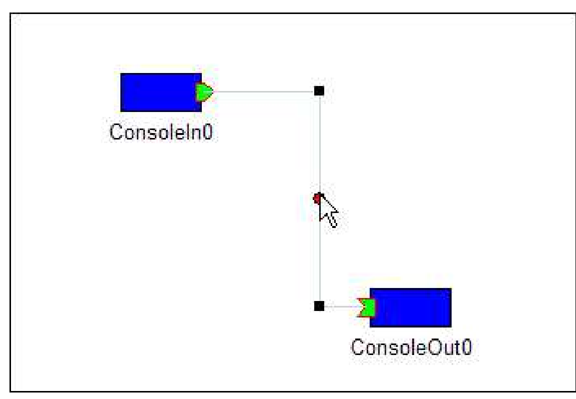


図 4.19: 折れ線の移動

4.1.7.5 コンポーネントの Start/Stop 等

コンポーネントブロック上で右クリックをすると、コンテキストメニューが表示されます (図 4.20)。

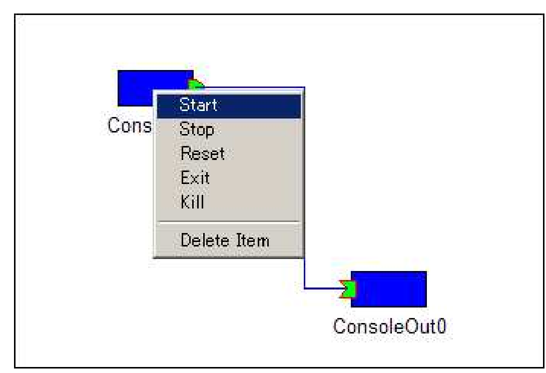


図 4.20: コンポーネントの Start

コンテキストメニューには以下ような項目があります。このメニューからコンポーネントの Start, Stop 等の操作を行うことが出来ます。Start している状態で Start メニューを選んだり、Stop している状態で Stop メニューを選択することも出来ますが、その場合は何も起こりません。

●コンポーネントコンテキストメニュー

Start	コンポーネントの起動
Stop	コンポーネントの停止
Reset	コンポーネントのリセット
Exit	コンポーネントの終了
Kill	コンポーネントの強制終了
Delete Item	コンポーネントの削除

4.1.8 システムのセーブとロード

RTCLink ではシステムドローウィンドウに構成したシステムの状態を保存したり、保存したシステムを復元することが出来ます。

4.1.8.1 構築したシステムの保存

システムの保存を行うには、以下の 2 通りの方法があります。

Save System

システムドローウィンドウ上に構成したシステムをデフォルト名 (システムドローウィンドウのタイトル) で XML ファイルに保存します。例えば、ウィンドウのタイトルが「System:1」

の場合は、System1.xml というファイル名になります。

● **構築したシステムをデフォルト名で保存**

メニューバーの「File」　「Save System」を選択する
システムドローウィンドウで [右クリック] しメニューから「Save System」を選択する

Save System as

システムドローウィンドウ上に構成したシステムをユーザ任意のファイル名で保存します。名前を付けて保存したい場合は、

● **構築したシステムを名前を付けて保存**

メニューバーの「File」　「Save System As」を選択する
システムドローウィンドウで [右クリック] しメニューから「Save System As」を選択する

Save System as の場合は、ファイルセーブ・ダイアログが表示されます (図 4.21)。

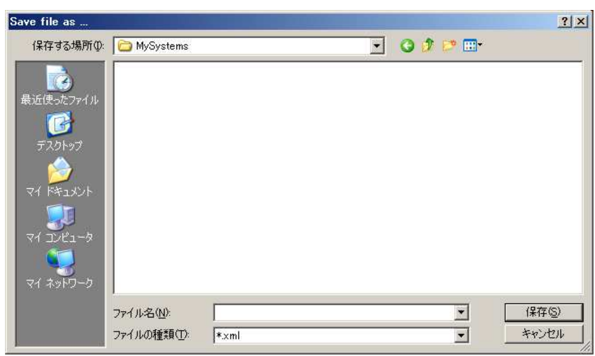


図 4.21: ファイルセーブ・ダイアログ

ファイルセーブ・ダイアログに任意のファイル名を入力し OK を押すと XML ファイルに保存されます。

4.1.8.2　保存したシステムの復元

XML ファイルに保存したシステム構成を復元するには、

● **保存したシステムを復元**

メニューバーの「File」　「Open System」を選択する
システムドローウィンドウで [右クリック] しメニューから「Open System」を選択する

ファイルオープン・ダイアログが表示されます (図 4.22)。

図 4.22: ファイルオープン・ダイアログ

ファイルオープン・ダイアログにユーザ任意のファイル名を入力し OK を押すと前回保存したシステムが白いコンポーネントブロックで表示されます (図 4.23)。ここでは、読み込んだシステム構成で正しいのか確認する段階となります。

ファイルオープン・ダイアログが表示されます (図 4.22)。

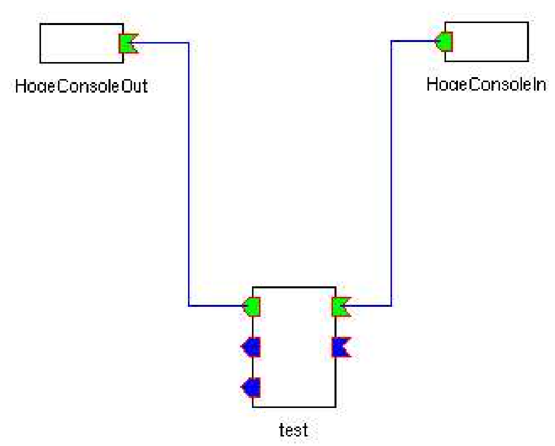


図 4.23: ロードされた直後のシステム

システム構成の確認の段階で、図 4.24 の様にコンポーネントブロックが黒い場合は、コンポーネントが起動していない状態です。コンポーネントを起動後、もう一度保存したシステムの復元を行ってください。

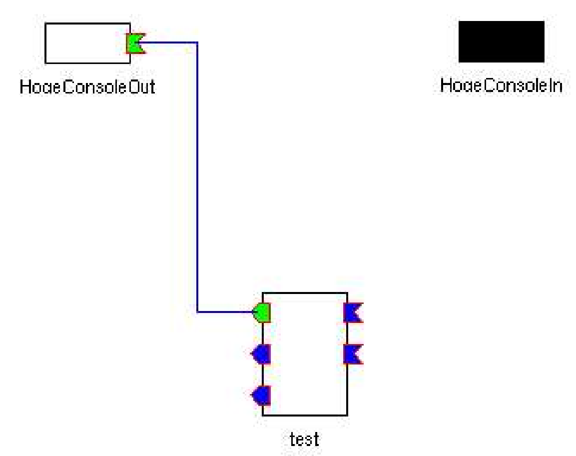


図 4.24: コンポーネントが起動していない状態

次に、コンポーネントの接続を行います。システムドローウィンドウのバックグランドで [右クリック] を行うとコンテキストメニューが表示されます。メニュー内の「Connect」を選択してください。コンポーネントブロックが青色に変わり Start/Stop 等の操作が可能となります。

● システム構成の確認時のコンテキストメニュー

Connect	コンポーネントの接続
Delete	表示しているシステム構成の削除

4.1.9 リフレッシュ

コンポーネントを接続したまま RTCLink を終了したり、コンポーネントを再起動した時等に、RTCLink 上のシステム構成と実際のコンポーネントの接続状態が異なる時があります。その際は、システムドローウィンドウのバックグランド上で [右クリック] を行いコンテキストメニュー内の「Refresh」を選択してください。

リフレッシュを行うと、コンポーネントの状態に合わせコンポーネントブロックの色を再表示し、システムドローウィンドウ上のシステム構成にあわせて接続や切断を行います。システムドローウィンドウ上のシステム構成にない接続が検出された場合は、システム構成にあわせて再接続 (接続や切断) を行うかダイアログで確認が行われます。

4.2 rtc-template

rtc-template はコンポーネントの雛形を生成する簡単なコードジェネレータです。

rtc-template には現在のところ以下の機能があります。

● rtc-template の機能

- RtcBase を継承したクラスの雛形の生成
- コンポーネントのプロファイルの埋め込み
- InPort の生成と登録
- OutPort の生成と登録

RTCBase インターフェースを継承した独自のインターフェースを持ったコンポーネントの生成にはまだ対応していません。今後のバージョンで対応する予定です。

使用できるオプションは --help オプションで表示することが出来ます。

```
> rtc-template --help

Usage: rtc-template [OPTIONS]
Options:
    [--help]                                  Print this help.
    [--c++]                                   Generate C++ template code.
    [--python]                                Generate Python tempalte code.
    [--output[=output_file]]                  Output base file name.
    [--module-name[=name]]                    Your module name.
    [--module-desc[=description]]             Module description.
    [--module-version[=version]]              Module version.
    [--module-author[=author]]                Module author.
    [--module-category[=category]]            Module category.
    [--module-comp-type[=component_type]]     Component type.
    [--module-act-type[=activity_type]]       Component's activity type.
    [--module-max-inst[=max_instance]]        Number of maximum instance.
    [--module-lang[=language]]                Language.
    [--module-inport[=PortName:Type]]         InPort's name and tyoe.
    [--module-outport[=PortName:Type]]        OutPort's name and type
     :
```

4.2.1 --help

rtc-template のヘルプを表示します。各オプションの意味と指定方法についても記述されていますので、参照してください。

4.2.2 `--c++`

C++ のコードを生成します。このオプションが指定された場合、コンポーネントのモジュール名 (後述) が Hoge の場合、以下のファイルが生成されます。

● 生成されるコード

Hoge.h	Hoge コンポーネントのヘッダファイル
Hoge.cpp	Hoge コンポーネントの実装ファイル
HogeComp.cpp	Hoge コンポーネントのスタンドアロン版実装ファイル
Makefile.Hoge	Hoge コンポーネントの Makefile

4.2.3 `--python`

Python のコードを生成します。コンポーネントのモジュール名 (後述) が Hoge の場合、以下のファイルが生成されます。

● 生成されるコード

Hoge.py	Hoge コンポーネントの Python 版実装ファイル

4.2.4 `--output`

コンポーネントのモジュール名に関わらず、生成するファイルの名前を指定します。`--output=Foo` を指定した場合、場合、以下のファイルが生成されます。

● 生成されるコード

Foo.h	コンポーネントのヘッダファイル
Foo.cpp	コンポーネントの実装ファイル
FooComp.cpp	コンポーネントのスタンドアロン版実装ファイル
Makefile.Foo	コンポーネントの Makefile

このオプションは `--c++` が指定されている場合にのみ有効です。

4.2.5 `--module-name`

モジュールの名前を指定します。ここで指定した名前は、クラス名にも使用されるので、C++ のクラスメイトして有効な名前を指定してください。

4.2.6 --module-desc

このコンポーネントモジュールの概要説明を記述します。ダブルクオートで囲めば、空白を含めることも出来ます。

4.2.7 --module-version

バージョン番号を指定します。

4.2.8 --module-author

コンポーネントの作成者を記述します。

4.2.9 --module-category

コンポーネントのカテゴリを記述します。

4.2.10 --module-comp-type

コンポーネントタイプを指定します。コンポーネントタイプについては、3.4 節の「コンポーネントタイプ」の項を参照してください。

4.2.11 --module-act-type

コンポーネントアクティビティタイプを指定します。コンポーネントアクティビティタイプについては、3.4 節の「コンポーネントアクティビティタイプ」の項を参照してください。

4.2.12 --module-max-inst

生成可能な最大のインスタンス数を数値で指定します。

4.2.13 --module-lang

コンポーネント記述言語を指定します。このオプションは --c++, --python の指定によって上書きされますので、指定しなくとも構いません。

4.2.14 --module-inport

このコンポーネントが持つ InPort をしてします。InPort を複数持つ場合には、このオプションを複数指定できます。

● **InPort 指定例**

pos:TimedFloatSeq	"pos" という名前を持つ、時間付き Float のシーケンス型
vel:TimedFloat	"pos" という名前を持つ、時間付き Float 型
num:TimedShort	"num" という名前を持つ、時間付き Short 型

InPort に付ける名前と型をコロン「:」で区切って指定します。指定可能な型については、3.4 節の「変数型」の項を参照してください。

4.2.15 --module-outport

このコンポーネントが持つ OutPort をしてします。OutPort を複数持つ場合には、このオプションを複数指定できます。指定の仕方については、InPort と同じです。

4.3 rtm-config

rtm-config はビルド時の各種設定情報を取得するためのコマンドです。

```
> ./rtm-config --help
Usage: rtm-config [OPTIONS]
Options:
        [--prefix[=DIR]]
        [--exec-prefix[=DIR]]
        [--version]
        [--libs]
        [--cflags]
        [--libdir]
        [--orb]
        [--idlc]
        [--idlflags]
```

4.3.1 --prefix

OpenRTM-aist のインストールディレクトリのプリフィクスを取得します。

```
> ./rtm-config --prefix
/usr/local
```

4.3.2 --exec-prefix

OpenRTM-aist の実行ファイルがインストールされたディレクトリのプリフィクスを取得します。

```
> ./rtm-config --exec-prefix
/usr/local
```

プリフィクス + bin が実際のディレクトリです。

4.3.3 --version

OpenRTM-aist のバージョン番号を取得します。

```
> ./rtm-config --version
aist-0.2.0
```

上記の例からわかるように、OpenRTM-aist では aist-x.x.x というバージョン番号をとります。

4.3.4 --libs

リンク時に必要なライブラリへのパスとライブラリをリンカへ渡すオプションの形式で取得します。Makefile を記述する際に、ライブラリの指定を rtm-config を利用して行うことで、移植性のある Makefile を記述することが出来ます。

```
> ./rtm-config --libs
-L/usr/local/lib -L/usr/local/lib -L/usr/local/lib -lpthread -lACE \
-lboost_regex -lomniORB4 -lomnithread -lomniDynamic4 -lRTC
```

4.3.5 --cflags

コンパイル時に必要なコンパイルオプションをコンパイラへ渡すオプションの形式で取得します。Makefile を記述する際に、コンパイルオプションの指定を rtm-config を利用して行うことで、移植性のある Makefile を記述することが出来ます

```
> ./rtm-config --cflags
-I/usr/local/include -I/usr/local/include -I/usr/local/include \
-I/usr/local/include -I/usr/local/include -I/usr/local/include/rtm/idl
```

4.3.6 --libdir

OpenRTM-aist の雑多なファイルをインストールしておくディレクトリを取得します。

```
> ./rtm-config --libdir
/usr/local/lib/OpenRTM
```

4.3.7 --orb

OpenRTM-aist がどの ORB でビルドされているかを指定します。下記の例では、OpenRTM-aist が omniORB でビルドされていることを示しています。

```
> ./rtm-config --orb
omniORB
```

4.3.8 --idlc

OpenRTM-aist がビルド時に使用した IDL コンパイラコマンドのフルパスを取得します。ユーザがコンポーネントに新たなインターフェースを追加する際には、このオプションで取得した IDL コンパイラを使用しなければなりません。

```
> ./rtm-config --idlc
/usr/local/bin/../bin/omniidl
```

4.3.9 --idlflags

OpenRTM-aist がビルド時に使用した IDL コンパイラコマンドへ与えたオプションを所得します。

```
> ./rtm-config --idlflags
-bcxx -Wba -nf
```

4.3.10 rtm-config の使用法

rtm-config を使用することにより、マシン毎に異なった設定でビルドされた OpenRTM がインストールされていても、その設定情報を容易に取得することができます。主な利用方法は、マシン毎に異なるコンパイルオプション、リンクオプションを rtm-config により取得し Makefile で使用することです。Makefile 内でバッククオートで rtm-config --libs や rtm-config --cflags を実行しコンパイルオプション、リンクオプションを設定すれば、他のマシンでも Makefile を書き換えることなく使用することが出来ます。

4.4 rtm-naming

rtm-naming は ORB 毎に異なるネームサーバーに対する起動ラッパです。現在のところ omniORB にしか対応していませんが、今後 OpenRTM-aist が他の ORB に対応した際に、同じコマンドオプションでネームサービスを起動できるようにするために用意されています。

4.5 rtcd

rtcd はコンポーネントを持たないコンポーネントマネージャだけからなる実行ファイルです。rtcd を起動して、後からローダブルモジュールコンポーネントをロードして使用します。

第5章　RTM Specification

OpenRTM-aist は 独立行政法人新エネルギー・産業技術統合開発機構 (NEDO) の 21 世紀ロボットチャレンジプログラム「ロボット機能発現のために必要な要素技術開発」において標準化作業が進められている RTM Specification のインターフェース仕様に準拠する実装のひとつとして開発されました。

プロジェクトでは、ロボットのための標準的なプラットフォームとなるようなミドルウエアを開発するために、インターフェースレベルでの仕様策定と標準化作業が行われ RTM Specification としてまとめられました。

本章では OpenRTM-aist のインターフェース仕様の基となっている RTM Specification について、仕様と実装の関係を説明します。また、IDL のインターフェース定義を示しながら、RTM Specification インターフェース仕様と OpenRTM-aist により独自に拡張されたインターフェースをそれぞれ示しながら、これらの違いについて説明します。

5.1　OpenRTM-aist と RTM Specification

RTM は独立行政法人新エネルギー・産業技術統合開発機構 (NEDO) の 21 世紀ロボットチャレンジプログラム「ロボット機能発現のために必要な要素技術開発」に於いて、ネットワークロボットのプラットフォームとなるミドルウエア技術を開発する目的の元に、独立行政法人産業技術総合研究所 (National Institute of Advanced Industrial Science and Technology: AIST)、松下電工株式会社 (Matsushita Electric Works, Ltd.: MEW) と日本ロボット工業会 (Japan Robot Association: JARA) の 3 者 (図 5.1) により、平成 14 年度から平成 16 年度までの 3 年間のプロジェクトとして研究開発が行われました。

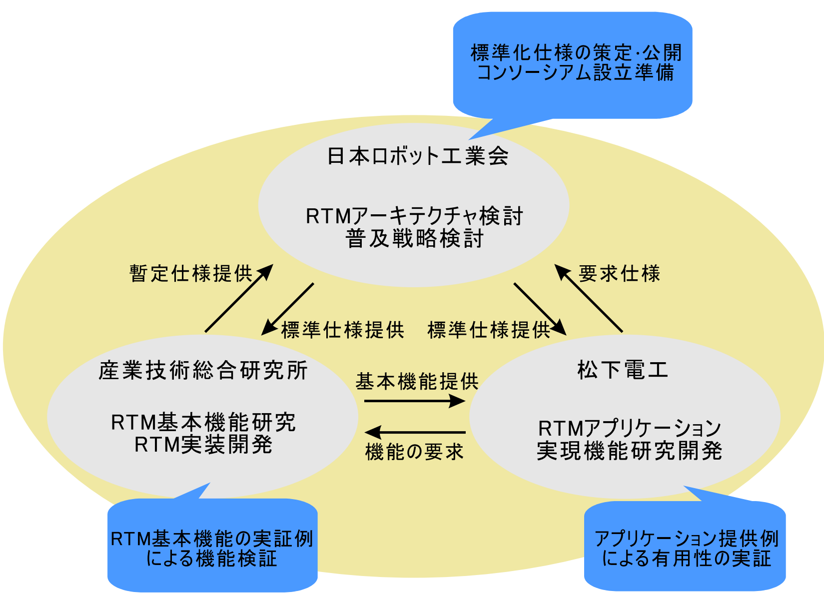


図 5.1: RTM 研究組織

RTM の標準化作業に向けた研究開発においては、様々なメーカー、コンポーネントベンダ、研究者が開発したコンポーネントが相互接続できるよう、インターフェースレベルでの標準化が進められました。同一のインターフェースを持つコンポーネントであれば、実装の如何に関わらず様々なコンポーネントが同一システムにおいて協調できることを目指しています。

この標準仕様事態はオープンなものであり、この仕様に基づいて誰でもこの実装を作成することが出来ます。実装されたものは、作成者が任意のライセンスに基づいて配布・販売することが出来ます。OpenRTM-aist は独立行政法人産業技術総合研究所がフリーで提供する RTM Specification に基づいた実装のひとつにあたります (図 5.2)。現在のところ、この RTM インターフェース仕様に準拠した RTM 実装でフリーなものは OpenRTM-aist しかありません。RTM Specification はコンポーネントの基本的なインターフェース、すなわちコンポーネントオブジェクト、Inport/Outport オブジェクトについての基本的なインターフェースを規定しているに過ぎません。OpenRTM-aist では標準インターフェース仕様に準拠したコンポーネントとともに、これを容易に開発するためのフレームワーク、各種サービス群をいったいとして配布しています。

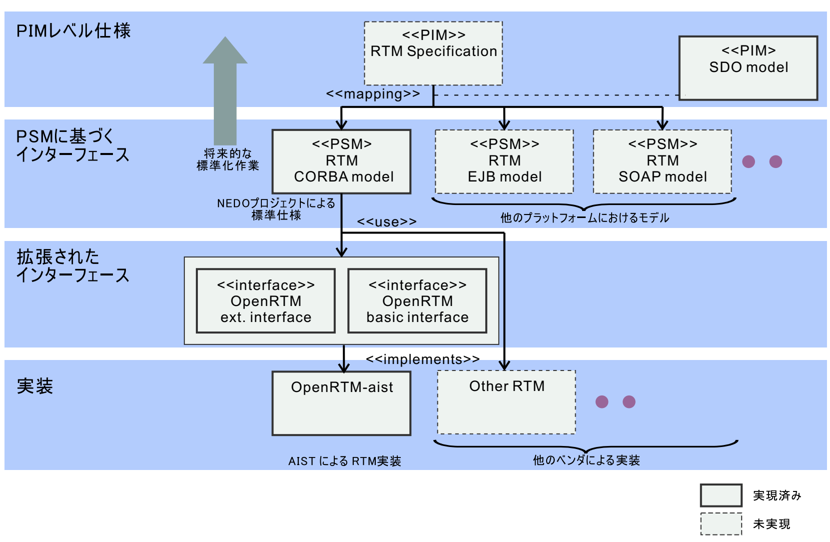


図 5.2: RTM Specification と OpenRTM-aist の関係

InPort/OutPort のデータ型、マネージャなど、コンポーネント運用上必要であるが、そのアプリケーションによりまた実装者により様々な形態が考えられ、かつコンポーネントの相互接続性に関与しない部分については基本仕様からは除かれています。たとえば、OpenRTM-aist においてはコンポーネント運用上の必要性から独自にコンポーネントマネージャCORBA オブジェクト、InPort/OutPort 基本データ型、コンポーネントプロファイルなどの拡張 CORBA オブジェクトを含んでいます。また、コンポーネントオブジェクトのインターフェースについても、コンポーネントの複合化やリアルタイム動作のための基本的なコンポーネントインターフェース `RTComponent` を継承する形でいくつかの拡張を行っており、そのインターフェースは `RTCBase` として定義されています。

5.2 RTM の標準化

上記 RTM プロジェクトでは、オブジェクト指向技術の標準化団体である OMG (Object Management Group) に対して、RTM を OMG 標準として認定されるよう活動を行っています。

5.2.1 OMG (Object Management Group)

OMG (Objet Management Group: オブジェクトマネージメントグループ) は 1989 年に発足したオブジェクト指向技術の標準化団体で、現在では世界から 800 以上のソフトウェアベンダ、ディベロッパ、一般企業が参画するソフトウエア業界最大の団体です。オブジェクト指向の分散アプリケーション構築を可能にするフレームワークの仕様を提案することを目的として標準化活動を行っています。

OMG の代表的な仕事としては、CORBA などの分散オブジェクトプラットフォームの仕様策定や、UML (Unified Modeling Language) といった、オブジェクト指向のソフトウェア開発における、プログラム設計図の統一表記法の策定[1]などがあります。

[1]UML 自体は Rational Software 社の Grady Booch 氏、James Rumbaugh 氏、Ivar Jacobson 氏の 3 人によって開発されたオブジェクト指向設計の表記法ですが、1997 年 11 月に OMG が UML を標準認定したことにより、事実上業界のオブジェクト指向設計表記法の標準となりました。

5.2.2 MDA (Model Driven Architecture)

OMG が掲げたアーキテクチャに、MDA (Model Driven Architecture) というものがあります。日本語では一般に「モデル駆動アーキテクチャ」と訳されます。

現在のソフトウエアの移植性、相互運用において、同じ機能を持つソフトウェアであっても、プラットフォームが異なると様々な実装があり、本来再利用が可能なはずのソフトウェアがプラットフォームが変わると相互運用が困難になることが大きな問題となっています。

MDA は、モデリング主導のシステム開発およびライフサイクル管理を実現するためのアーキテクチャです。MDA の考え方は、全てのシステムはまずモデルにより定義、デザインされ、モデルが定義されることで特定言語や製品へのマッピングは自動的に決定され、システム管理、インテグレーションもモデルを中心として行うことで、システム開発の効率向上を図ろうというものです。

モデルの形で仕様を定めれば、それよりも下流の詳細設計やコーディングといった工程が大幅に自動化されるので、これが実現すればシステム構築の工数を大幅に削減できます。最終的には、モデルから実装コードの自動生成なども視野に入れているようですが、まだ実現にまでは至っていないようです。しかしながら、この考えも 1980 年代の CASE ツールが登場したときと同じだとの批判もあるのが事実です。

しかしながら、MDA で取り扱うモデルは、プラットフォームが変わっても透過性の高い再利用可能なシステムデザインの知見であり、システム開発に関わる関係者がシステムについて議論できる具体的な対象を与えてくれるという意味においては、有効な手法であるといえそうです。

MDA に基づいた開発では、

1. OS や言語、ミドルウェアなどに依存しないモデル「PIM (Platform Independent Model) を作成する。
2. PIM を基に、OS や言語、ミドルウェアなどに特化したモデル「PSM (Platform Specific Model)」を作成する。
3. PSM を基に、プログラムを生成する

という流れで開発が進められるということになっています。

5.2.3 PIM (Platform Independent Model)

PIM とはプラットフォームに依存しないモデルであり、ここでのプラットフォームとは OS や言語、ミドルウェアなどを指します。RTM プロジェクトにおいて策定された仕様は CORBA を前提とした IDL ファイルと周辺の仕様として提供されています。したがって、RTM 仕様はプラットフォームに依存するという意味で PIM とはなっていません。しかしながら、インターフェース設計に関わる考え方を PIM に一般化することは可能ですので、将来的には PIM レベルで標準化が行われる可能性もあります。PIM レベルでの標準化が行われれば、他のプラットフォーム (JEB, DCOM, SOAP, XML-RPC) などへのポーティングも可能になるでしょう (図 5.2)。

5.2.4 PSM (Platform Specific Model)

PSM とはプラットフォームに依存したモデルであり、例えば CORBA をプラットフォームと規定した場合 IDL インターフェースレベルでのモデル化になります。RTM では、仕様を CORBA の IDL レベルでのインターフェースの統一化による標準化を行ってきており、現在得られている仕様は PSM であるといえます (図 5.2)。

5.3 RTM Specification Ver.0.1

5.3.1 RTMBase.idl

RTMBase.idl では全体に共通する雑多な定義を行っています。

RTMBase.idl

```
module RTM {

  typedef short RtmRes;

  const RtmRes RTM_OK        = 0;
  const RtmRes RTM_ERR       = 1;
  const RtmRes RTM_WARNING   = 2;
  const RtmRes RTM_FATAL_ERR = 4;

  struct NamedValue {
    string name;
    any value;
  };

  typedef sequence<NamedValue> NVList;

  struct Time
  {
    unsigned long sec;    // sec
    unsigned long nsec;   // nano sec
  };
};
```

RTM のオペレーションの戻り値として RtmRes を定義しています。オペレーションが正常に実行されたかどうかを戻り値として返す場合、RtmRes を戻り値として、以下の値を定義しています。

● オペレーションの戻り値

```
const RtmRes RTM_OK        = 0;  正常終了
const RtmRes RTM_ERR       = 1;  エラー終了
const RtmRes RTM_WARNING   = 2;  警告
const RtmRes RTM_FATAL_ERR = 4;  致命的エラー終了
```

任意の型の値を格納する構造体として、NamedValue を定義しています。また、NamedValue のシーケンス型として NVList を定義しています。

時間を格納する構造体として Time 構造体を定義しています。Time 構造体は 秒単位の sec 及びナノ秒単位の nsec のメンバから構成されます。主に、InPort/OutPort でやり取りされるデータのタイムスタンプに使用されます。

5.3.2 RTComponent

以下に RTComponent のインターフェース IDL を示します。

RTComponent.idl (1)

```
#include "RTMBase.idl"
#include "RTCInPort.idl"
#include "RTCOutPort.idl"

module RTM {
  interface RTComponent
  //    : NamedObject, PropertySet
  {
    readonly attribute string instance_id;
    readonly attribute string implementation_id;
    readonly attribute string description;
    readonly attribute string version;
    readonly attribute string maker;
    readonly attribute string category;

    typedef short ComponentState;

    const ComponentState RTC_UNKNOWN        = 0;
    const ComponentState RTC_BORN           = 1;
    const ComponentState RTC_INITIALIZING   = 2;
    const ComponentState RTC_READY          = 3;
    const ComponentState RTC_STARTING       = 4;
    const ComponentState RTC_ACTIVE         = 5;
    const ComponentState RTC_STOPPING       = 6;
    const ComponentState RTC_ABORTING       = 7;
    const ComponentState RTC_ERROR          = 8;
    const ComponentState RTC_FATAL_ERROR    = 9;
    const ComponentState RTC_EXITING        = 10;

    exception IllegalTransition {};
```

RT コンポーネントの本体となるオブジェクトの宣言で `interface RTComponent` インターフェースとして定義されています。RTM のインターフェース定義は全て `RTM` 名前空間の下で定義されます。

プロファイルの attribute

コンポーネントプロファイル情報の以下の項目を `attribute` として定義しています。

● コンポーネントのプロファイル情報

`readonly attribute string instance_id;`	インスタンス ID
`readonly attribute string implementation_id;`	インプリメンテーション ID
`readonly attribute string description;`	概要説明
`readonly attribute string version;`	バージョン
`readonly attribute string maker;`	作成者
`readonly attribute string category;`	カテゴリ

5.3.3 コンポーネントアクティビティ状態

コンポーネントアクティビティの状態として、CORBA の short 型変数として ComponentState を typedef し、各状態を表す const 定数として以下の 11 の状態を定義しています。

● コンポーネントアクティビティ状態

```
const ComponentState RTC_UNKNOWN        = 0;   UNKNOWN 状態
const ComponentState RTC_BORN           = 1;   BORN 状態
const ComponentState RTC_INITIALIZING   = 2;   INITIALIZING 状態
const ComponentState RTC_READY          = 3;   READY 状態
const ComponentState RTC_STARTING       = 4;   STARTING 状態
const ComponentState RTC_ACTIVE         = 5;   ACTIVE 状態
const ComponentState RTC_STOPPING       = 6;   STOPPING 状態
const ComponentState RTC_ABORTING       = 7;   ABORTING 状態
const ComponentState RTC_ERROR          = 8;   ERROR 状態
const ComponentState RTC_FATAL_ERROR    = 9;   FATAL_ERROR 状態
const ComponentState RTC_EXITING        = 10;  EXITING 状態
```

上記の状態は図 5.3 の状態遷移に従います。

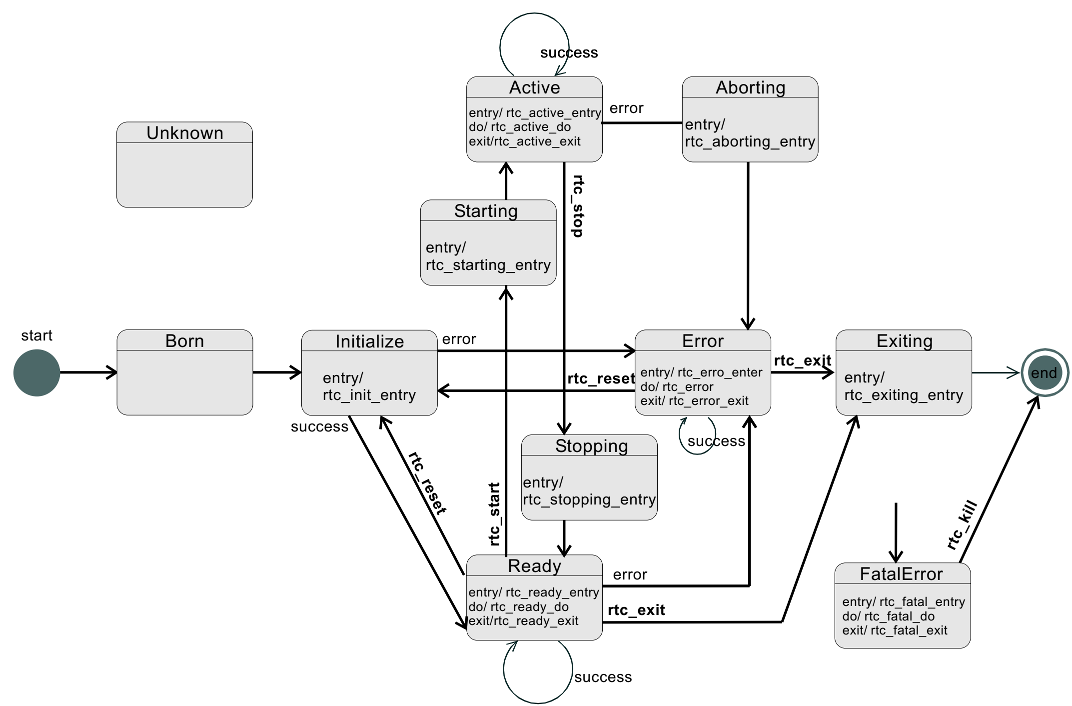


図 5.3: RT コンポーネント状態遷移図

RTComponent.idl (2)

```
    RtmRes rtc_worker();

    RtmRes rtc_start() raises (IllegalTransition);
    RtmRes rtc_stop() raises (IllegalTransition);
    RtmRes rtc_reset() raises (IllegalTransition);
    RtmRes rtc_exit() raises (IllegalTransition);
    RtmRes rtc_kill();

    readonly attribute OutPort rtc_state;

    exception NoSuchName { string name; };
    readonly attribute InPortList inports;
    InPort get_inport(in string name) raises (NoSuchName);
    readonly attribute OutPortList outports;
    OutPort get_outport(in string name) raises (NoSuchName);

    //! Return connector port object reference list
    //    readonly attribute ConnectorList connectors;
    //    OutPort get_connector(in string name) raises (NoSuchName);

    //    readonly attribute PropertySet rtc_settings;
  };

  typedef sequence<RTComponent> RTComponentList;

};  // end of module RTM
```

rtc_worker() オペレーションはコンポーネントのアクティビティ本体が実行されるオペレーションです。通常は、内部から実行され外部から呼び出されることはありませんが、複合コンポーネントを構成する場合などに外部の同期を取るコンポーネントなどから呼び出されることもあります。

rtc_start()-rtc_kill() オペレーションはコンポーネントの状態を遷移させるためのオペレーションとして定義されています。コンポーネントアクティビティの状態遷移は図 5.3 にしたがって行われます。

呼び出されるオペレーションが現在の状態において適切でなかった場合、IllegalTransition が発生します。

rtc_state はコンポーネントの状態を出力する OutPort オブジェクトのリファレンスを返します。この OutPort を用いてコンポーネントの状態を pull 型または push 型データ転送で取得することが出来ます。

● **コンポーネントアクティビティと状態遷移オペレーション**

`RtmRes rtc_worker();`	アクティビティ本体
`RtmRes rtc_start() raises (IllegalTransition);`	ACTIVE 状態へ遷移
`RtmRes rtc_stop() raises (IllegalTransition);`	READY 状態へ遷移
`RtmRes rtc_reset() raises (IllegalTransition);`	INITIALIZING 状態へ遷移
`RtmRes rtc_exit() raises (IllegalTransition);`	EXITING 状態へ遷移
`RtmRes rtc_kill();`	FATAL_ERROR 状態のとき EXITING 状態へ遷移
`readonly attribute OutPort rtc_state;`	コンポーネント状態 OutPort

コンポーネントは 1 つ以上のデータ入出力オブジェクト InPort/OutPort を持つことが出来ます。以下のオペレーションを使用しこれらのオブジェクトリファレンスを取得することが出来ます。

● InPort/OutPort

`exception NoSuchName { string name;};`	例外定義
`readonly attribute InPortList inports;`	InPortList を取得
`InPort get_inport(in string name)` `raises (NoSuchName);`	名前を指定し InPort を取得
`readonly attribute OutPortList outports;`	OutPortList を取得
`OutPort get_outport(in string name)` `raises (NoSuchName);`	名前を指定し OutPort を取得

最後に、コンポーネントのオブジェクトリファレンスをシーケンスとして保存する `RTComponentList` を定義しています。複数のコンポーネントをまとめて扱う場合に使用します。

5.3.4 RTCInPort.idl

RTCInPort.idl

```
#include "RTMBase.idl"

module RTM {
  typedef string SubscriptionID;

  struct PortProfile
  {
    string name;
    CORBA::TypeCode port_type;
    NVList properties;
  };

  interface InPort
  {
    exception Disconnected{};

    void put(in any data) raises(Disconnected);

    readonly attribute PortProfile profile;
  };

  typedef sequence<InPort> InPortList;
  typedef sequence<PortProfile> PortProfileList;

}; // end of module RTM
```

まず、CORBA string 型として `SubscriptionID` を定義しています。`SubscriptionID` は OutPort をサブスクライブする際の識別 ID で、通常は UUID を使用します。

構造体 `PortProfile` は InPort/OutPort のプロファイル情報を格納する構造体です。トップレベルには、ポート名: `name` とポート型: `port_type` のみをもち、その他の情報は `NamedValue` のシーケンス: `NVList` として保持します。

InPort のインターフェース宣言では、InPort にデータを送る `put()` インターフェースがあり、これにより InPort にデータを渡すことが出来ます。すでに、接続が切断されている OutPort から put を呼び出すと `Disconnected` 例外が発生します。InPort のプロファイルデータはメンバの `profile` に格納されています。

5.3.5 RTCOutPort.idl

RTCOutPort.idl

```
#include "RTMBase.idl"
#include "RTCInPort.idl"

module RTM {
  typedef short SubscriptionType;

  const SubscriptionType OPS_ONCE               = 0;
  const SubscriptionType OPS_PERIODIC           = 1;
  const SubscriptionType OPS_NEW                = 2;
  const SubscriptionType OPS_TRIGGERED          = 3;
  const SubscriptionType OSP_PERIODIC_NEW       = 4;
  const SubscriptionType OPS_NEW_PERIODIC       = 5;
  const SubscriptionType OPS_PERIODIC_TRIGGERED  = 6;
  const SubscriptionType OPS_TRIGGERRED_PERIODIC = 7;

  struct SubscriberProfile
  {
    SubscriptionType subscription_type;
    boolean event_base;
    NVList properties;
  };

  interface OutPort
  {
    any get();
    RtmRes subscribe(in InPort in_port, out SubscriptionID id,
                     in SubscriberProfile profile);
    RtmRes unsubscribe(in SubscriptionID id);
    readonly attribute InPortList inports;
    readonly attribute PortProfile profile;
  };

  typedef sequence<OutPort> OutPortList;
}; // end of module RTM
```

まずはじめに、OutPort をサブスクライブする際のモード `SubscriptionType` を CORBA short 型で `typedef` しています。

サブスクリプション型定数は以下のように定義されています。

● サブスクリプション型

OPS_ONCE	= 0;	一度だけデータ取得する
OPS_PERIODIC	= 1;	一定周期でデータ送信
OPS_NEW	= 2;	データ更新時にデータ送信
OPS_TRIGGERED	= 3;	トリガによりデータ送信
OSP_PERIODIC_NEW	= 4;	一定周期でデータ更新時のみ送信
OPS_NEW_PERIODIC	= 5;	データ更新時に送信・一定周期で送信
OPS_PERIODIC_TRIGGERED	= 6;	一定周期でトリガ時のみ送信
OPS_TRIGGERRED_PERIODIC	= 7;	トリガ時に送信・一定周期で送信

`OPS_ONCE` ではサブスクライブ後一度だけデータを取得して接続が切断されます。

`OPS_PERIODIC` は指定した周期でデータ送信をサブスクライバに送信します。

`OPS_NEW` は OutPort のデータが更新され次第すぐにサブスクライバにデータが送信されます。

`OPS_TRIGGERED` では、予め設定されたトリガによりトリガが発生したときにデータが送信されます。トリガとは、ポートごとに設定される送信条件で、例えば「OutPort の値が 4.0 以上になったら送信」などのような条件のことです。

OSP_PERIODIC_NEW はデータ更新時のみ、予め設定された周期のタイミングでデータが送信されます。周期が Δt [s] に設定されたとき、データ送信のタイミングは、データが更新されてから最も近い $n\Delta t$ [s] に行われます。

OPS_NEW_PERIODIC では、データ更新時にすぐにデータは送信されますが、それ以降はデータが更新され続ける限りは一定周期で送信されます。

OPS_PERIODIC_TRIGGERED では、 OPS_PERIODIC_NEW の送信条件がトリガに変わるだけで、そのほかの動作は OPS_PERIODIC_NEW と同じになります。

OPS_TRIGGERRED_PERIODIC では、OPS_NEW_PERIODIC の送信条件がトリガに変わるだけで、そのほかの動作は OPS_NEW_PERIODIC と同じになります。

SubscriptionType を含めたサブスクライブに関するサブスクライバのプロファイルを定義しているのが、次の SubscriberProfile です。

```
struct SubscriberProfile
{
  SubscriptionType subscription_type;
  boolean event_base;
  NVList properties;
};
```

SubscriptionType subscription_type に上記の SubscriptionType を指定します。サブスクリプションがイベントに依存するか否かを CORBA Boolean 型で event_base に指定します。サブスクリプションに関するその他のプロパティは NVList properties に指定します。

OutPort をサブスクライブする場合には、この構造体に適切なデータをセットして、OutPort の subscribe() オペレーションを呼び出さなければなりません。

次に OutPort インターフェースです。

```
interface OutPort
{
  any get();
  RtmRes subscribe(in InPort in_port, out SubscriptionID id,
                   in SubscriberProfile profile);
  RtmRes unsubscribe(in SubscriptionID id);
  readonly attribute InPortList inports;
  readonly attribute PortProfile profile;
};

typedef sequence<OutPort> OutPortList;
```

get() オペレーションでは、現在の OutPort の値を Any 型で取得することが出来ます。

定期的にデータを push 型で送ってほしい場合には subscribe() オペレーションでサブスクライブします。引数には、サブスクライブする InPort のオブジェクトリファレンスと、サブスクライバのプロファイル構造体を指定します。subscribe() が成功すると RtmRes に RTM_OK

が戻り値として返り、`SubscriptionID id` に UUID (Universally Unique IDentifier) を返します。

サブスクライブをやめるときにはこの UUID を指定して `unsubscribe()` オペレーションを発行します。

OutPort はサブスクライブしている InPort のリスト `InPortList inports` を `attribute` として保持しています。また、OutPort 自身のプロファイルとして `PortProfile profile` を `attribute` として保持しています。

5.4 OpenRTM-aist インターフェース定義

OpenRTM-aist では、RTM Specification に基づき、いくつかのインターフェースの拡張と、新たなオブジェクトインターフェースの定義を追加しています。

● OpenRTM-aist インターフェース定義 IDL ファイル

`RTCBase.idl`	RTComponent を拡張するインターフェースを定義
`RTCProfile.idl`	RTComponent が持つプロファイルを拡張し構造体として定義
`RTCDataType.idl`	InPort/OutPort のための標準的なデータタイプを定義
`RTCMnager.idl`	RTComponent のライフサイクルを管理するマネージャ

5.4.1 RTCBase.idl

OpenRTM-aist では RTComponent インターフェースを拡張したインターフェース定義 `RTCBase` を提供します。拡張インターフェースは主に、複合コンポーネントを構成するために必要なインターフェースを定義しています。

RTCBase.idl

```
#include "RTComponent.idl"
#include "RTCProfile.idl"

module RTM {

  interface RTCBase;
  typedef sequence<RTCBase> RTCBaseList;

  interface RTCBase
    : RTComponent
  {
    RtmRes rtc_ready_entry();
    RtmRes rtc_ready_do();
    RtmRes rtc_ready_exit();
    RtmRes rtc_active_entry();
    RtmRes rtc_active_do();
    RtmRes rtc_active_exit();
    RtmRes rtc_error_entry();
    RtmRes rtc_error_do();
    RtmRes rtc_error_exit();
    RtmRes rtc_fatal_entry();
    RtmRes rtc_fatal_do();
    RtmRes rtc_fatal_exit();
    RtmRes rtc_init_entry();
    RtmRes rtc_starting_entry();
    RtmRes rtc_stopping_entry();
    RtmRes rtc_aborting_entry();
    RtmRes rtc_exiting_entry();

    RtmRes rtc_stop_thread();
    RtmRes rtc_start_thread();

    RtmRes rtc_set_parent(in RTCBase comp);
    RtmRes rtc_add_component(in RTCBase comp);
    RtmRes rtc_delete_component(in RTCBase comp);
    RtmRes rtc_replace_component(in RTCBase comp1, in RTCBase comp2);
    RtmRes rtc_replace_component_by_name(in string name1, in string name2);
    RTCBaseList rtc_components();
    RTCBase rtc_get_component(in string name);

    RtmRes rtc_attach_inport(in InPort in_port);
    RtmRes rtc_attach_inport_by_name(in RTCBase comp, in string name);
    RtmRes rtc_detatch_inport(in InPort in_port);
    RtmRes rtc_detatch_inport_by_name(in string name);
```

```
    RtmRes rtc_attach_outport(in OutPort out_port);
    RtmRes rtc_attach_outport_by_name(in RTCBase comp, in string name);
    RtmRes rtc_detatch_outport(in OutPort out_port);
    RtmRes rtc_detatch_outport_by_name(in string name);

    readonly attribute RTCProfile profile;
  };
};
```

コンポーネントアクティビティの各状態に対応するメソッドを、CORBA オペレーションとして定義しています。

アクティビティオペレーション

```
    RtmRes rtc_ready_entry();
    RtmRes rtc_ready_do();
    RtmRes rtc_ready_exit();
    RtmRes rtc_active_entry();
    RtmRes rtc_active_do();
    RtmRes rtc_active_exit();
    RtmRes rtc_error_entry();
    RtmRes rtc_error_do();
    RtmRes rtc_error_exit();
    RtmRes rtc_fatal_entry();
    RtmRes rtc_fatal_do();
    RtmRes rtc_fatal_exit();
    RtmRes rtc_init_entry();
    RtmRes rtc_starting_entry();
    RtmRes rtc_stopping_entry();
    RtmRes rtc_aborting_entry();
    RtmRes rtc_exiting_entry();
```

これらのオペレーションを定義することにより、アクティビティを実行する主体 (スレッド) をコンポーネントの外部に置くことが出来るようになります。これは、複数のコンポーネントをグループ化しアクティビティを同期的かつシーケンシャルに実行する同期複合コンポーネントにおいて使用することを目的としています。

アクティビティオスレッド制御ペレーション

```
    RtmRes rtc_stop_thread();
    RtmRes rtc_start_thread();
```

これらのオペレーションによりコンポーネント内部のアクティビティを実行するスレッドの実行を制御することが出来ます。複合コンポーネントにおいては、アクティビティ実行の主体であるスレッドを外部に置くため、コンポーネント内部のスレッドを停止させなければなりません。したがって、外部からコンポーネントアクティビティスレッドを制御する必要性から、これらのインターフェースが定義されています。

コンポーネントの親子関係を制御するオペレーション

```
RtmRes rtc_set_parent(in RTCBase comp);
RtmRes rtc_add_component(in RTCBase comp);
RtmRes rtc_delete_component(in RTCBase comp);
RtmRes rtc_replace_component(in RTCBase comp1, in RTCBase comp2);
RtmRes rtc_replace_component_by_name(in string name1, in string name2);
RTCBaseList rtc_components();
RTCBase rtc_get_component(in string name);
```

これらのオペレーションはコンポーネントの親子関係を制御するためのコンポーネントです。通常の単体のコンポーネントでは、`rtc_set_parent()` オペレーションは無効になり `RTM_ERR` を返します。複合コンポーネントでは、これらのオペレーションを使用し複数のコンポーネントを子供として所有することが出来ます。

InPort アタッチオペレーション

```
RtmRes rtc_attach_inport(in InPort in_port);
RtmRes rtc_attach_inport_by_name(in RTCBase comp, in string name);
RtmRes rtc_detatch_inport(in InPort in_port);
RtmRes rtc_detatch_inport_by_name(in string name);
```

これらのオペレーションを使用することにより複合コンポーネントが所有する子コンポーネントの InPort を親である複合コンポーネントの InPort として再定義できるようになります。実際には、親コンポーネントで子コンポーネントの InPort のオブジェクトリファレンスを保持しているだけです。

OutPort アタッチオペレーション

```
RtmRes rtc_attach_outport(in OutPort out_port);
RtmRes rtc_attach_outport_by_name(in RTCBase comp, in string name);
RtmRes rtc_detatch_outport(in OutPort out_port);
RtmRes rtc_detatch_outport_by_name(in string name);
```

これらのオペレーションを使用することにより複合コンポーネントが所有する子コンポーネントの OutPort を親である複合コンポーネントの OutPort として再定義できるようになります。実際には、親コンポーネントで子コンポーネントの OutPort のオブジェクトリファレンスを保持しているだけです。

5.4.2 RTCProfile.idl

RTCProfile.idl では RTCProfile インターフェースを定義しています。RTCProfile は RTComponent の instance_id, implementation_id 等のプロファイル定義を拡張し、構造にしたものです。RTCBase ではこのプロファイルを取得するオペレーションが拡張され、プロファイル全体を 1 度に取得できるようになっています。

RTCProfile.idl

```
module RTM {
  interface RTComponent;

  enum RTComponentType {
    STATIC,
    UNIQUE,
    COMMUTATIVE
  };

  enum RTCActivityType {
    PERIODIC,
    SPORADIC,
    EVENT_DRIVEN
  };

  enum RTCLangType {
    COMPILE,
    SCRIPT
  };

  struct RTCProfile
  {
    string name;
    string instance_id;
    string implementation_id;
    string description;
    string version;
    string maker;
    string category;
    RTComponentType component_type;
    RTCActivityType activity_type;
    long max_instance;
    string language;
    RTCLangType language_type;
    string module_profile_file;
    PortProfileList outport_profile_list;
    PortProfileList inport_profile_list;
  };
};
```

以下の instance_id, implementation_id 等については RTComponent.idl で定義されているものと全く同じです。

● 基本的なプロファイル

```
string instance_id;
string implementation_id;
string description;
string version;
string maker;
string category;
```

RTComponentType はコンポーネントのタイプを指定します。コンポーネントタイプとは、生成されるコンポーネントのインスタンスの形式を指します。コンポーネントタイプには以下の

ものがあります。

● `RTComponentType`

STATIC	コンポーネントはマネージャに登録されると同時にインスタンス化され、新たに生成することはできない。ハードウエアに密接に関係するコンポーネント等はこのタイプにするとハードウエアとコンポーネントの対応がとりやすい。
UNIQUE	コンポーネントは動的に生成・削除するができるが、component0 と component1 は異なる内部状態を持ち交換可能ではない。
COMMUTATIVE	コンポーネントは、互いに交換可能。ソフトウエアのロジックのみのコンポーネントはこのタイプになる。

`RTCActivityType` はコンポーネントのアクティビティのタイプを指定します。コンポーネントのアクティビティタイプは以下のタイプがあります。

● `RTCActivityType`

PERIODIC	コンポーネントの活動は一定周期で行われます。ただし、動作周期を守れるか否かは、OS に依存します。リアルタイム OS(ART-LINUX) を使用すれば一定周期動作を行わせることはできますが、非リアルタイム OS では厳密な周期動作をさせることは不可能です。
SPORADIC	コンポーネントの活動の周期は一定ではないが、繰り返し行われます。
EVENT_DRIVEN	外部からのオペレーションにより受動的に動作します。

その他の項目については以下のとおりです。

● **その他**

`long max_instance;`	最大のインスタンス数
`string language;`	コンポーネント記述言語
`RTCLangType language_type;`	コンポーネント記述言語型
`string module_profile_file;`	モジュールプロファイルファイル名

また、`RTCProfile` は InPort/OutPort のプロファイルもリストとして保持しています。

● **InPort/OutPort プロファイルリスト**

`PortProfileList outport_profile_list;`	InPort プロファイルのリスト
`PortProfileList inport_profile_list;`	OutPort プロファイルのリスト

`PortProfileList` は `RTCInPort.idl` で定義されている `PortProfile` のシーケンスです。

5.4.3 RTCDataType.idl

以下に、RTCDataType.idl を示します。

RTCDataType.idl (基本型)

```
#include "RTMBase.idl"

module RTM {
  struct TimedState
  {
    Time tm;
    short data;
  };
 struct TimedShort
  {
    Time tm;
    short data;
  };
 struct TimedLong
  {
    Time tm;
    long data;
  };
  struct TimedUShort
  {
    Time tm;
    unsigned short data;
  };
  struct TimedULong
  {
    Time tm;
    unsigned long data;
  };
  struct TimedFloat
  {
    Time tm;
    float data;
  };
  struct TimedDouble
  {
    Time tm;
    double data;
  };
  struct TimedChar
  {
    Time tm;
    char data;
  };
  struct TimedBoolean
  {
    Time tm;
    boolean data;
  };
  struct TimedOctet
  {
    Time tm;
    octet data;
  };
  struct TimedString
  {
    Time tm;
    string data;
  };
```

OpenRTM-aist では、InPort/OutPort でやり取りするデータ型としてタイムスタンプ付きのデータ型を、CORBA の基本型とそのシーケンス型について提供しています。

RTCDataType.idl (シーケンス型)

```
  struct TimedShortSeq
  {
    Time tm;
    sequence<short> data;
  };
  struct TimedLongSeq
```

```
  {
    Time tm;
    sequence<long> data;
  };
  struct TimedUShortSeq
  {
    Time tm;
    sequence<unsigned short> data;
  };
  struct TimedULongSeq
  {
    Time tm;
    sequence<unsigned long> data;
  };
  struct TimedFloatSeq
  {
    Time tm;
    sequence<float> data;
  };
  struct TimedDoubleSeq
  {
    Time tm;
    sequence<double> data;
  };
  struct TimedCharSeq
  {
    Time tm;
    sequence<char> data;
  };
  struct TimedBooleanSeq
  {
    Time tm;
    sequence<boolean> data;
  };
  struct TimedOctetSeq
  {
    Time tm;
    sequence<octet> data;
  };
  struct TimedStringSeq
  {
    Time tm;
    sequence<string> data;
  };
};
```

5.4.4 RTCManager.idl

RTCManager は RTComponent のモジュールのロード、インスタンス化等のライフサイクル管理を行うオブジェクトです。

RTCProfile.idl

```
#include "RTMBase.idl"
#include "RTComponent.idl"
#include "RTCBase.idl"

module RTM
{
  typedef sequence<string> ComponentFactoryList;

  interface RTCManager
  {
    RtmRes load(in string pathname, in string initfunc);
    RtmRes unload(in string pathname);
    RTCBase create_component(in string comp_name,
                             out string instance_name);
    RtmRes delete_component(in string instance_name);
    ComponentFactoryList component_factory_list();
    RTCBaseList component_list();
    RtmRes command(in string cmd, out string ret);
  };

}; // end of namespace RTM
```

ダイナミックにモジュールをロード・アンロードするための、load(), unload() オペレーションが定義されています。

● モジュールのロード・アンロードオペレーション

load(in string pathname, in string initfunc);	モジュールのロード
unload(in string pathname);	モジュールのアンロード

また、create_component(), delete_component() はコンポーネントの生成・削除を行うオペレーションです。

● コンポーネントの生成・削除を行うオペレーション

create_component(in string comp_name, out string instance_name);	コンポーネントの生成
delete_component(in string instance_name);	コンポーネントの削除

その他のオペレーションとしては、コンポーネントを生成するファクトリのリストを取得する component_factory_list() や、現在インスタンス化されているコンポーネントのリストを取得する component_list() があります。

● その他のオペレーション

component_factory_list();	コンポーネントファクトリのリスト
component_list();	コンポーネントのリスト
command(in string cmd, out string ret);	簡易コマンドインタプリタ

5.5 OpenRTM-aist の拡張オペレーションと標準化

現在、RTM Specification は標準化作業の途中であり、ここで示したインターフェース仕様は今後変わる可能性があります。標準化を行う過程で、実際に多くのアプリケーションに使用した結果からのフィードバックを得て、より実用的で使いやすい標準仕様とすることを目指しています。

OpenRTM-aist で拡張された仕様も、今後変更される可能性があります。一部のものは RTM Specification に取り込まれ、拡張され、あるいは削除されるかもしれません。

第6章　OpenRTM クラスリファレンス

6.1　クラス RTM::RtcBase

RTComponent 基底クラス.

`#include <RtcBase.h>`

Public メソッド

- **RtcBase** ()
  RtcBase(p. 99) クラスコンストラクタ.
- **RtcBase** (CORBA::ORB_ptr orb, PortableServer::POA_ptr poa)
  RtcBase(p. 99) クラスコンストラクタ.
- **RtcBase** (**RtcManager** *manager)
  RtcBase(p. 99) クラスコンストラクタ.
- virtual ~**RtcBase** ()
  RtcBase(p. 99) クラスデストラクタ.
- virtual RtmRes **rtc_start** () throw (CORBA::SystemException, RTM::RTComponent::IllegalTransition)
  [CORBA interface] コンポーネントのアクティブ化
- virtual RtmRes **rtc_stop** () throw (CORBA::SystemException, RTM::RTComponent::IllegalTransition)
  [CORBA interface] コンポーネントの非アクティブ化
- virtual RtmRes **rtc_reset** () throw (CORBA::SystemException, RTM::RTComponent::IllegalTransition)
  [CORBA interface] コンポーネントのリセット
- virtual RtmRes **rtc_exit** () throw (CORBA::SystemException, RTM::RTComponent::IllegalTransition)
  [CORBA interface] コンポーネントのリセット
- virtual RtmRes **rtc_kill** ()
  [CORBA interface] コンポーネントの強制終了

- virtual RtmRes **rtc_worker** ()
  [CORBA interface] メインアクティビティのメソッド
- virtual RtmRes **rtc_ready_entry** ()
  [CORBA interface] entry: ready() メソッド
- virtual RtmRes **rtc_ready_do** ()
  [CORBA interface] do: ready() メソッド.
- virtual RtmRes **rtc_ready_exit** ()
  [CORBA interface] exit: ready() メソッド.
- virtual RtmRes **rtc_active_entry** ()
  [CORBA interface] entry: active() メソッド.
- virtual RtmRes **rtc_active_do** ()
  [CORBA interface] do: active() メソッド.
- virtual RtmRes **rtc_active_exit** ()
  [CORBA interface] exit: active() メソッド.
- virtual RtmRes **rtc_error_entry** ()
  [CORBA interface] entry: error() メソッド
- virtual RtmRes **rtc_error_do** ()
  [CORBA interface] do: error() メソッド.
- virtual RtmRes **rtc_error_exit** ()
  [CORBA interface] exit: error() メソッド.
- virtual RtmRes **rtc_fatal_entry** ()
  [CORBA interface] entry: fatal() メソッド
- virtual RtmRes **rtc_fatal_do** ()
  [CORBA interface] do: fatal() メソッド.
- virtual RtmRes **rtc_fatal_exit** ()
  [CORBA interface] exit: fatal() メソッド.
- virtual RtmRes **rtc_init_entry** ()
  [CORBA interface] entry: init() メソッド
- virtual RtmRes **rtc_starting_entry** ()
  [CORBA interface] entry: starting() メソッド
- virtual RtmRes **rtc_stopping_entry** ()
  [CORBA interface] entry: stopping() メソッド

- virtual RtmRes **rtc_aborting_entry** ()
  [CORBA interface] entry: aborting() メソッド

- virtual RtmRes **rtc_exiting_entry** ()
  [CORBA interface] entry: exiting() メソッド

- virtual OutPort_ptr **rtc_state** ()
  [CORBA interface] アクティビティステータスの *OutPort* の取得

- virtual InPortList * **inports** ()
  [CORBA interface] InPortList の取得

- virtual InPort_ptr **get_inport** (const char *name) throw (CORBA::SystemException, RTM::RTComponent::NoSuchName)
  [CORBA interface] InPort の取得

- virtual OutPortList * **outports** ()
  [CORBA interface] OutPortList の取得

- virtual OutPort_ptr **get_outport** (const char *name) throw (CORBA::SystemException, RTM::RTComponent::NoSuchName)
  [CORBA interface] OutPort の取得

- virtual char * **instance_id** ()
  [CORBA interface] instance_id の取得

- virtual char * **implementation_id** ()
  [CORBA interface] implementation_id の取得

- virtual char * **description** ()
  [CORBA interface] description の取得

- virtual char * **version** ()
  [CORBA interface] version の取得

- virtual char * **maker** ()
  [CORBA interface] maker の取得

- virtual char * **category** ()
  [CORBA interface] category の取得

- virtual RTCProfile * **profile** ()
  [CORBA interface] profile の取得

- virtual RtmRes **rtc_start_thread** ()
  [CORBA interface] アクティビティスレッドのスタート

- virtual RtmRes **rtc_suspend_thread** ()

[CORBA interface] アクティビティスレッドのサスペンド

- virtual RtmRes **rtc_stop_thread** ()

  [CORBA interface] アクティビティスレッドのストップ

- virtual RtmRes **rtc_set_parent** (RTCBase_ptr comp)

  [CORBA interface] 親コンポーネントをセットする

- virtual RtmRes **rtc_add_component** (RTCBase_ptr comp)

  [CORBA interface] 子コンポーネントを追加する

- virtual RtmRes **rtc_delete_component** (RTCBase_ptr comp)

  [CORBA interface] 子コンポーネントを削除する

- virtual RtmRes **rtc_replace_component** (RTCBase_ptr comp1, RTCBase_ptr comp2)

  [CORBA interface] 子コンポーネントの順序を入れ替える

- virtual RtmRes **rtc_replace_component_by_name** (const char *name1, const char *name2)

  [CORBA interface] 子コンポーネントの順序を入れ替える

- virtual RTCBaseList * **rtc_components** ()

  [CORBA interface] 子コンポーネントをリストとして取得する。

- virtual RTCBase_ptr **rtc_get_component** (const char *name)

  [CORBA interface] 子コンポーネントを名前を指定して取得

- virtual RtmRes **rtc_attach_inport** (InPort_ptr in_port)

  [CORBA interface] InPort をアタッチする

- virtual RtmRes **rtc_attach_inport_by_name** (RTCBase_ptr comp, const char *name)

  [CORBA interface] InPort をアタッチする

- virtual RtmRes **rtc_detatch_inport** (InPort_ptr in_port)

  [CORBA interface] InPort をデタッチする

- virtual RtmRes **rtc_detatch_inport_by_name** (const char *name)

  [CORBA interface] InPort をデタッチする

- virtual RtmRes **rtc_attach_outport** (OutPort_ptr out_port)

  [CORBA interface] OutPort をアタッチする

- virtual RtmRes **rtc_attach_outport_by_name** (RTCBase_ptr comp, const char *name)

  [CORBA interface] OutPort をアタッチする

- virtual RtmRes **rtc_detatch_outport** (OutPort_ptr out_port)
  [CORBA interface] OutPort をデタッチする
- virtual RtmRes **rtc_detatch_outport_by_name** (const char *name)
  [CORBA interface] OutPort をデタッチする
- virtual void **init_orb** (CORBA::ORB_ptr orb, PortableServer::POA_ptr poa)
  ORB, POA のポインタを与えて初期化する.
- virtual int **open** (void *args)
  コンポーネントのアクティビティスレッドを生成する
- virtual int **svc** (void)
  コンポーネントのアクティビティスレッド関数
- virtual int **close** (unsigned long flags)
  コンポーネントのアクティビティスレッド終了関数
- virtual RTM::RTComponent::ComponentState **getState** ()
  コンポーネントステート取得
- virtual void **initModuleProfile** (RtcModuleProfile prof)
  ModuleProfile の初期化.
- virtual RtcModuleProfile & **getModuleProfile** ()
  ModuleProfile の取得.
- virtual string **setComponentName** (int num)
  コンポーネント名をセットする
- bool **registerInPort** (**InPortBase** &in_ch)
  InPort の登録.
- bool **registerPort** (**InPortBase** &in_ch)
  InPort の登録.
- bool **deleteInPort** (**InPortBase** &in_ch)
  InPort の登録解除.
- bool **deletePort** (**InPortBase** &in_ch)
  InPort の登録解除.
- bool **deleteInPortByName** (const char *ch_name)
  InPort の登録解除.
- void **readAllInPorts** ()
  全ての *InPort* のデータ取り込み.

- void **finalizeInPorts** ()
  全ての *InPort* の終了処理.
- bool **registerOutPort** (**OutPortBase** &out_ch)
  OutPort の登録.
- bool **registerPort** (**OutPortBase** &out_ch)
- bool **deleteOutPort** (**OutPortBase** &out_ch)
  OutPort の登録.
- bool **deletePort** (**OutPortBase** &out_ch)
  OutPort の登録.
- bool **deleteOutPortByName** (const char *ch_name)
  OutPort の登録解除.
- void **writeAllOutPorts** ()
  全ての *OutPort* のデータ書き出し.
- void **finalizeOutPorts** ()
  全ての *OutPort* の終了処理.
- void **appendAlias** (const char *alias)
  コンポーネント名の *alias* を登録
- void **appendAlias** (const std::string alias)
  コンポーネント名の *alias* を登録
- std::list< string > **getAliases** ()
  コンポーネント名の *alias* を取得
- void **setNamingPolicy** (**NamingPolicy** policy)
- **NamingPolicy getNamingPolicy** ()
- bool **isLongNameEnable** ()
- bool **isAliasEnable** ()
- void **forceExit** ()
- void **finalize** ()
  コンポーネントの終了処理
- bool **isThreadRunning** ()

Protected 型

- typedef RtmRes(RtcBase::* **StateFunc**)()
  アクティビティ関数ポインタ宣言

- typedef list< **InPortBase** ∗ >::iterator **InPorts_it**
  InPort リストイテレータ.

- typedef list< **OutPortBase** ∗ >::iterator **OutPorts_it**
  OutPort リストイテレータ.

- enum **ThreadStates** { **UNKNOWN**, **RUNNING**, **SUSPEND**, **STOP** }
  アクティビティスレッド状態フラグ構造体

Protected メソッド

- RtmRes **_check_error** (RtmRes result)
  アクティビティエラーチェック関数

- RtmRes **_nop** ()
  アクティビティ用ダミー関数

- RtmRes **_rtc_initializing** ()
  アクティビティ *rtc_init_entry* 実行関数

- RtmRes **_rtc_ready_entry** ()
  アクティビティ *rtc_ready_entry* 実行関数

- RtmRes **_rtc_starting** ()
  アクティビティ *rtc_starting_entry* 実行関数

- RtmRes **_rtc_active_entry** ()
  アクティビティ *rtc_active_entry* 実行関数

- RtmRes **_rtc_stopping** ()
  アクティビティ *rtc_stippoing_entry* 実行関数

- RtmRes **_rtc_aborting** ()
  アクティビティ *rtc_aborting_entry* 実行関数

- RtmRes **_rtc_error_entry** ()
  アクティビティ *rtc_error_entry* 実行関数

- RtmRes **_rtc_fatal_entry** ()
  アクティビティ *rtc_fatal_entry* 実行関数

- RtmRes **_rtc_exiting** ()
  アクティビティ *rtc_exiting_entry* 実行関数

- RtmRes **_rtc_ready** ()
  アクティビティ *rtc_ready_do* 実行関数

- RtmRes **_rtc_active** ()
  アクティビティ *rtc_active_do* 実行関数
- RtmRes **_rtc_error** ()
  アクティビティ *rtc_error_do* 実行関数
- RtmRes **_rtc_fatal** ()
  アクティビティ *rtc_fatal_do* 実行関数
- RtmRes **_rtc_ready_exit** ()
  アクティビティ *rtc_ready_exit* 実行関数
- RtmRes **_rtc_active_exit** ()
  アクティビティ *rtc_active_exit* 実行関数
- RtmRes **_rtc_error_exit** ()
  アクティビティ *rtc_error_exit* 実行関数
- RtmRes **_rtc_fatal_exit** ()
  アクティビティ *rtc_fatal_exit* 実行関数
- void **init_state_func_table** ()
  アクティビティ関数テーブルの初期化

Protected 変数

- CORBA::ORB_ptr **m_pORB**
  ORB ポインタ変数.
- PortableServer::POA_ptr **m_pPOA**
  POA ポインタ変数.
- **RtcManager** ∗ **m_pManager**
  Manager ポインタ変数.
- RTCBase_var **m_Parent**
  親コンポーネントのオブジェクトリファレンス
- **ThreadState m_ThreadState**
  アクティビティスレッド状態変数
- **ComponentStateMtx m_CurrentState**
  アクティビティの現在状態変数
- **ComponentStateMtx m_NextState**

アクティビティの次状態変数

- **StateFunc _exit_func** [11]
  アクティビティの *exit* 関数テーブル

- **StateFunc _entry_func** [11]
  アクティビティの *entry* 関数テーブル

- **StateFunc _do_func** [11]
  アクティビティの *do* 関数テーブル

- **InPorts m_InPorts**
  mutex 付き *InPort* リスト

- **OutPorts m_OutPorts**
  mutex 付き *OutPort* リスト

- TimedState **m_TimedState**
  Input port flag list アクティビティ状態変数.

- **OutPortAny**< TimedState > **m_StatePort**
  アクティビティ状態 *OutPort*

- RtcModuleProfile **m_Profile**
  コンポーネントプロファイル

- std::list< string > **m_Alias**
  コンポーネント名の *alias*

- **NamingPolicy m_NamingPolicy**
- RtcMedLogbuf **m_MedLogbuf**
  ロガー仲介バッファ

- RtcLogStream **rtcout**
  ロガーストリーム

6.1.1 説明

RTComponent 基底クラス.

RTComponent 開発者は新たに作成するコンポーネントのクラスをこの RtcBase の サブクラスとして定義しなければならない。 新たに作成するコンポーネントにおいて、必要なアクティビティの状態に対応する メソッド **rtc_active_do()**(p. 117) 等を適宜オーバーライドして、各状態にて行う処理を そのメソッドに記述する。 rtc_xxx_entry(), rtc_xxx_exit() ではその状態に入るときと出るときに一度だけ それらのメソッドが実行され、rtc_xxx_do() ではその状態にいる間中、そのメソ ッドが周期実行される。 各メソッド rtc_xxx_[entry|do|exit]() では戻り値に RTM_OK, RTM_ERR, RTM_WARNING, RTM_FATAL_ERR のいずれかを返す。 RTM_ERR

を返すと ERROR 状態に、RTM_FATAL_ERR を返すと FATAL_ERROR 状態に 移行する。その他の状態遷移についてはマニュアルを参照。

新たに作成するコンポーネントのクラスでは、 InPort , OutPort とこれに バインドされる InPort , OutPort の型に対応する変数を定義する。 コンポーネントのクラスのコンストラクタでは、これらの変数を初期化子を用いて 初期化し、コンストラクタ内で registerInPort , registerOutPort を用いて それぞれ InPort , OutPort として登録する必要がある。

6.1.2 型定義

6.1.2.1 typedef list<InPortBase*>::iterator RTM::RtcBase::InPorts_it [protected]

InPort リストイテレータ.

6.1.2.2 typedef list<OutPortBase*>::iterator RTM::RtcBase::OutPorts_it [protected]

OutPort リストイテレータ.

6.1.2.3 typedef RtmRes(RtcBase::* RTM::RtcBase::StateFunc)() [protected]

アクティビティ関数ポインタ宣言

6.1.3 列挙型

6.1.3.1 enum RTM::RtcBase::ThreadStates [protected]

アクティビティスレッド状態フラグ構造体

列挙型の値:

-4pt ***UNKNOWN***

RUNNING

SUSPEND

STOP

6.1.4 コンストラクタとデストラクタ

6.1.4.1 RTM::RtcBase::RtcBase ()

RtcBase(p. 99) クラスコンストラクタ.

RTComponent サーバント実装の基底クラス **RtcBase**(p. 99) のコンストラクタ。 このコンストラクタを使用する場合、ユーザは ORB へのポインタと POA のポインタ あるいは、マネー

ジャへのポインタを後でセットする必要がある。

6.1.4.2 RTM::RtcBase::RtcBase (CORBA::ORB_ptr *orb*, PortableServer::POA_ptr *poa*)

RtcBase(p. 99) クラスコンストラクタ.

RTComponent サーバント実装の基底クラス **RtcBase**(p. 99) のコンストラクタ。

ORB へのポインタと POA へのポインタがすでに得られている場合には、この コンストラクタを使用することが出来る。

引数:

-4pt *orb* ORB へのポインタ

poa POA へのポインタ

6.1.4.3 RTM::RtcBase::RtcBase (RtcManager * *manager*)

RtcBase(p. 99) クラスコンストラクタ.

通常はマネージャを通してコンポーネントクラスをインスタンス化し、 このコンストラクタを使用することを推奨する。

引数:

-4pt *manager* コンポーネントマネージャ **RtcManager**(p. 152) へのポインタ

6.1.4.4 virtual RTM::RtcBase::~RtcBase () [virtual]

RtcBase(p. 99) クラスデストラクタ.

6.1.5 関数

6.1.5.1 RtmRes RTM::RtcBase::_check_error (RtmRes *result*) [protected]

アクティビティエラーチェック関数

6.1.5.2 RtmRes RTM::RtcBase::_nop () [inline, protected]

アクティビティ用ダミー関数

6.1.5.3 RtmRes RTM::RtcBase::_rtc_aborting () [protected]

アクティビティ rtc_aborting_entry 実行関数

6.1.5.4 RtmRes RTM::RtcBase::_rtc_active () [protected]

アクティビティ rtc_active_do 実行関数

6.1.5.5 RtmRes RTM::RtcBase::_rtc_active_entry () [protected]

アクティビティ rtc_active_entry 実行関数

6.1.5.6 RtmRes RTM::RtcBase::_rtc_active_exit () [protected]

アクティビティ rtc_active_exit 実行関数

6.1.5.7 RtmRes RTM::RtcBase::_rtc_error () [protected]

アクティビティ rtc_error_do 実行関数

6.1.5.8 RtmRes RTM::RtcBase::_rtc_error_entry () [protected]

アクティビティ rtc_error_entry 実行関数

6.1.5.9 RtmRes RTM::RtcBase::_rtc_error_exit () [protected]

アクティビティ rtc_error_exit 実行関数

6.1.5.10 RtmRes RTM::RtcBase::_rtc_exiting () [protected]

アクティビティ rtc_exiting_entry 実行関数

6.1.5.11 RtmRes RTM::RtcBase::_rtc_fatal () [protected]

アクティビティ rtc_fatal_do 実行関数

6.1.5.12 RtmRes RTM::RtcBase::_rtc_fatal_entry () [protected]

アクティビティ rtc_fatal_entry 実行関数

6.1.5.13 RtmRes RTM::RtcBase::_rtc_fatal_exit () [protected]

アクティビティ rtc_fatal_exit 実行関数

6.1.5.14 RtmRes RTM::RtcBase::_rtc_initializing () [protected]

アクティビティ rtc_init_entry 実行関数

6.1.5.15 RtmRes RTM::RtcBase::_rtc_ready () [protected]

アクティビティ rtc_ready_do 実行関数

6.1.5.16 RtmRes RTM::RtcBase::_rtc_ready_entry () [protected]

アクティビティ rtc_ready_entry 実行関数

6.1.5.17 RtmRes RTM::RtcBase::_rtc_ready_exit () [protected]

アクティビティ rtc_ready_exit 実行関数

6.1.5.18 RtmRes RTM::RtcBase::_rtc_starting () [protected]

アクティビティ rtc_starting_entry 実行関数

6.1.5.19 RtmRes RTM::RtcBase::_rtc_stopping () [protected]

アクティビティ rtc_stippoing_entry 実行関数

6.1.5.20 void RTM::RtcBase::appendAlias (const std::string *alias*)

コンポーネント名の alias を登録

コンポーネント名の alias を登録する。登録された alias はネーミングサーバ に登録される。 名前の付け方は コンテキストの区切りを "/"、 id と kind の区切りを "|" とする。コンテキスト Manipulator の下に自分自身を MyManipulator0 として bind したい場合には "/Manipualtor/MyManipulator0|rtc" を文字列として 渡す。マネージャにより適切なタイミングでネーミングサーバに登録される。

引数:

-4pt ***alias*** コンポーネント名の alias

6.1.5.21 void RTM::RtcBase::appendAlias (const char * *alias*)

コンポーネント名の alias を登録

コンポーネント名の alias を登録する。登録された alias はネーミングサーバ に登録される。 名前の付け方は コンテキストの区切りを "/"、 id と kind の区切りを "|" とする。コ

ンテキスト Manipulator の下に自分自身を MyManipulator0 として bind したい場合には ”/Manipualtor/MyManipulator0|rtc” を文字列として 渡す。マネージャにより適切なタイミングでネーミングサーバに登録される。

引数:

-4pt ***alias*** コンポーネント名の alias

6.1.5.22 virtual char∗ RTM::RtcBase::category () [virtual]

[CORBA interface] category の取得

コンポーネントのカテゴリを取得する。

6.1.5.23 virtual int RTM::RtcBase::close (unsigned long *flags*) [virtual]

コンポーネントのアクティビティスレッド終了関数

コンポーネントの内部アクティビティスレッド終了時に呼ばれる。 コンポーネントオブジェクトの非アクティブ化、マネージャへの通知を行う。 これは ACE_Task サービスクラスメソッドのオーバーライド。

6.1.5.24 bool RTM::RtcBase::deleteInPort (InPortBase & *in_ch*)

InPort の登録解除.

登録されている InPort の登録を解除する。

引数:

-4pt ***in_ch*** InPort object of InPort<T>

6.1.5.25 bool RTM::RtcBase::deleteInPortByName (const char ∗ *ch_name*)

InPort の登録解除.

登録されている InPort の登録を名前を指定して解除する。

引数:

-4pt ***ch_name*** InPort 名

6.1.5.26 bool RTM::RtcBase::deleteOutPort (OutPortBase & *out_ch*)

OutPort の登録.

OutPort をコンポーネントに登録する。登録された OutPort は外部から 見えるようになる。

引数:

-4pt ***out_ch*** OutPort object of OutPort<T>

6.1.5.27 bool RTM::RtcBase::deleteOutPortByName (const char * *ch_name*)

OutPort の登録解除.

登録されている OutPort の登録を名前を指定して解除する。

引数:

-4pt ***ch_name*** OutPort 名

6.1.5.28 bool RTM::RtcBase::deletePort (OutPortBase & *out_ch*) [inline]

OutPort の登録.

OutPort をコンポーネントに登録する。登録された OutPort は外部から 見えるようになる。

引数:

-4pt ***out_ch*** OutPort object of OutPort<T>

6.1.5.29 bool RTM::RtcBase::deletePort (InPortBase & *in_ch*) [inline]

InPort の登録解除.

登録されている InPort の登録を解除する。

引数:

-4pt ***in_ch*** InPort object of InPort<T>

6.1.5.30 virtual char* RTM::RtcBase::description () [virtual]

[CORBA interface] description の取得

コンポーネントの概要説明を取得する。

6.1.5.31 void RTM::RtcBase::finalize ()

コンポーネントの終了処理

1.OutPort の終了処理+deactivate、2.InPort の終了処理+deactivate、 3. アクティビティの停止、4. コンポーネントの deactivate 5.

6.1.5.32 void RTM::RtcBase::finalizeInPorts ()

全ての InPort の終了処理.

全ての InPort に対してオブジェクトの deactivate() を行う。

6.1.5.33 void RTM::RtcBase::finalizeOutPorts ()

全ての OutPort の終了処理.

全ての OutPort に対して各サブスクライバを切断し、オブジェクトの deactivate() を行う。

6.1.5.34 void RTM::RtcBase::forceExit ()

6.1.5.35 virtual InPort_ptr RTM::RtcBase::get_inport (const char * *name*) throw (CORBA::SystemException, RTM::RTComponent::NoSuchName) `[virtual]`

[CORBA interface] InPort の取得

InPort のオブジェクトリファレンスを取得する。

引数:

-4pt *name* InPort 名

6.1.5.36 virtual OutPort_ptr RTM::RtcBase::get_outport (const char * *name*) throw (CORBA::SystemException, RTM::RTComponent::NoSuchName) `[virtual]`

[CORBA interface] OutPort の取得

OutPort のオブジェクトリファレンスを取得する。

引数:

-4pt *name* OutPort 名

6.1.5.37 std::list<string> RTM::RtcBase::getAliases ()

コンポーネント名の alias を取得

登録されているコンポーネント名の alias を取得する。

6.1.5.38 virtual RtcModuleProfile& RTM::RtcBase::getModuleProfile () `[inline, virtual]`

ModuleProfile の取得.

ModuleProfile を取得する。

6.1.5.39 NamingPolicy RTM::RtcBase::getNamingPolicy ()

6.1.5.40 virtual RTM::RTComponent::ComponentState RTM::RtcBase::getState () [virtual]

コンポーネントステート取得

コンポーネントの現在の状態を取得する。

6.1.5.41 virtual char* RTM::RtcBase::implementation_id () [virtual]

[CORBA interface] implementation_id の取得

コンポーネントのインプリメンテーション ID を取得する。

6.1.5.42 virtual void RTM::RtcBase::init_orb (CORBA::ORB_ptr *orb*, PortableServer::POA_ptr *poa*) [virtual]

ORB, POA のポインタを与えて初期化する.

コンポーネントに ORB と POA のポインタを与えてコンポーネントを初期化する。

引数:

-4pt *orb* ORB へのポインタ

poa POA へのポインタ

6.1.5.43 void RTM::RtcBase::init_state_func_table () [protected]

アクティビティ関数テーブルの初期化

6.1.5.44 virtual void RTM::RtcBase::initModuleProfile (RtcModuleProfile *prof*) [virtual]

ModuleProfile の初期化.

RtcModuleProfile クラスのインスタンスを渡して、コンポーネントの プロファイルを初期化する。

引数:

-4pt ***RtcModuleProfSpec*** モジュールプロファイル

6.1.5.45 virtual InPortList* RTM::RtcBase::inports () [virtual]

[CORBA interface] InPortList の取得

InPort のオブジェクトリファレンスのリストを取得する。

6.1.5.46 virtual char* RTM::RtcBase::instance_id () [virtual]

[CORBA interface] instance_id の取得

コンポーネントのインスタンス ID を取得する。

6.1.5.47 bool RTM::RtcBase::isAliasEnable ()

6.1.5.48 bool RTM::RtcBase::isLongNameEnable ()

6.1.5.49 bool RTM::RtcBase::isThreadRunning ()

6.1.5.50 virtual char* RTM::RtcBase::maker () [virtual]

[CORBA interface] maker の取得

コンポーネントの作成者を取得する。

6.1.5.51 virtual int RTM::RtcBase::open (void * *args*) [virtual]

コンポーネントのアクティビティスレッドを生成する

コンポーネントの内部アクティビティスレッドを生成し起動する。 これは ACE_Task サービスクラスメソッドのオーバーライド。

引数:

-4pt *args* 通常は 0

6.1.5.52 virtual OutPortList* RTM::RtcBase::outports () [virtual]

[CORBA interface] OutPortList の取得

OutPort のオブジェクトリファレンスのリストを取得する。

6.1.5.53 virtual RTCProfile* RTM::RtcBase::profile () [virtual]

[CORBA interface] profile の取得

コンポーネントのプロファイルを取得する。

6.1.5.54 void RTM::RtcBase::readAllInPorts ()

全ての InPort のデータ取り込み.

全ての InPort に対して InPort::read() を実行。予めバインドされた変数に 最新の値が代入される。変数にバインドされていない InPort では何も 起こらない。

6.1.5.55 bool RTM::RtcBase::registerInPort (InPortBase & *in_ch*)

InPort の登録.

InPort をコンポーネントに登録する。登録された InPort は外部から 見えるようになる。

引数:

-4pt ***in_ch*** InPort object of InPort<T>

6.1.5.56 bool RTM::RtcBase::registerOutPort (OutPortBase & *out_ch*)

OutPort の登録.

OutPort をコンポーネントに登録する。登録された OutPort は外部から 見えるようになる。

引数:

-4pt ***out_ch*** OutPort object of OutPort<T>

6.1.5.57 bool RTM::RtcBase::registerPort (OutPortBase & *out_ch*) `[inline]`

6.1.5.58 bool RTM::RtcBase::registerPort (InPortBase & *in_ch*) `[inline]`

InPort の登録.

InPort をコンポーネントに登録する。登録された InPort は外部から 見えるようになる。

引数:

-4pt ***in_ch*** InPort object of InPort<T>

6.1.5.59 virtual RtmRes RTM::RtcBase::rtc_aborting_entry () `[inline, virtual]`

[CORBA interface] entry: aborting() メソッド

ABORTING 状態へ進入するときに 1 度だけ呼び出されるメソッド。 エラーがなければ READY 状態へ遷移する。 戻り値が RTM_ERR で ERROR 状態へ、 RTM_FATALERR で FATAL_ERROR 状態へ 遷移する。

6.1.5.60 virtual RtmRes RTM::RtcBase::rtc_active_do () `[inline, virtual]`

[CORBA interface] do: active() メソッド.

ACTIVE 状態に居る間周期実行されるメソッド。

6.1.5.61 virtual RtmRes RTM::RtcBase::rtc_active_entry () [inline, virtual]

[CORBA interface] entry: active() メソッド.

ACTIVE 状態に進入するときに 1 度だけ実行されるメソッド。

6.1.5.62 virtual RtmRes RTM::RtcBase::rtc_active_exit () [inline, virtual]

[CORBA interface] exit: active() メソッド.

ACTIVE 状態から出るときに 1 度だけ実行されるメソッド。

6.1.5.63 virtual RtmRes RTM::RtcBase::rtc_add_component (RTCBase_ptr *comp*) [inline, virtual]

[CORBA interface] 子コンポーネントを追加する

子コンポーネントのオブジェクトリファレンスをセットする。 単体コンポーネントにおいてこのメソッドは未使用のため、RTM_ERR を返す。

6.1.5.64 virtual RtmRes RTM::RtcBase::rtc_attach_inport (InPort_ptr *in_port*) [inline, virtual]

[CORBA interface] InPort をアタッチする

子コンポーネントの InPort をこのコンポーネントの InPort にアタッチする。 単体コンポーネントにおいては自分自身オブジェクトリファレンスを返す。

6.1.5.65 virtual RtmRes RTM::RtcBase::rtc_attach_inport_by_name (RTCBase_ptr *comp*, const char * *name*) [inline, virtual]

[CORBA interface] InPort をアタッチする

子コンポーネントの InPort 名を指定してコンポーネントの InPort にアタッチ する。 単体コンポーネントにおいては自分自身オブジェクトリファレンスを返す。

6.1.5.66 virtual RtmRes RTM::RtcBase::rtc_attach_outport (OutPort_ptr *out_port*) [inline, virtual]

[CORBA interface] OutPort をアタッチする

子コンポーネントの OutPort をこのコンポーネントの OutPort にアタッチする。 単体コンポーネントにおいては自分自身オブジェクトリファレンスを返す。

6.1.5.67 virtual RtmRes RTM::RtcBase::rtc_attach_outport_by_name (RTCBase_ptr *comp*, const char * *name*) [inline, virtual]

[CORBA interface] OutPort をアタッチする

子コンポーネントの OutPort 名を指定してコンポーネントの InPort にアタッチ する。 単体コンポーネントにおいては自分自身オブジェクトリファレンスを返す。

6.1.5.68 virtual RTCBaseList* RTM::RtcBase::rtc_components () [inline, virtual]

[CORBA interface] 子コンポーネントをリストとして取得する。

子コンポーネントのリストを取得する。 単体コンポーネントにおいては自分自身オブジェクトリファレンスを返す。

6.1.5.69 virtual RtmRes RTM::RtcBase::rtc_delete_component (RTCBase_ptr *comp*) [inline, virtual]

[CORBA interface] 子コンポーネントを削除する

子コンポーネントのオブジェクトリファレンスを削除する。 単体コンポーネントにおいてこのメソッドは未使用のため、RTM_ERR を返す。

6.1.5.70 virtual RtmRes RTM::RtcBase::rtc_detatch_inport (InPort_ptr *in_port*) [inline, virtual]

[CORBA interface] InPort をデタッチする

子コンポーネントの InPort をこのコンポーネントの InPort からデタッチする。 単体コンポーネントにおいては RTM_ERR を返す。

6.1.5.71 virtual RtmRes RTM::RtcBase::rtc_detatch_inport_by_name (const char * *name*) [inline, virtual]

[CORBA interface] InPort をデタッチする

子コンポーネントの InPort を名前を指定してこのコンポーネントの InPort から デタッチする。 単体コンポーネントにおいては RTM_ERR を返す。

6.1.5.72 virtual RtmRes RTM::RtcBase::rtc_detatch_outport (OutPort_ptr *out_port*) [inline, virtual]

[CORBA interface] OutPort をデタッチする

子コンポーネントの OutPort をこのコンポーネントの OutPort からデタッチする。 単体コンポーネントにおいては RTM_ERR を返す。

6.1.5.73 virtual RtmRes RTM::RtcBase::rtc_detatch_outport_by_name (const char * *name*) [inline, virtual]

[CORBA interface] OutPort をデタッチする

子コンポーネントの OutPort を名前を指定してこのコンポーネントの OutPort から デタッチする。 単体コンポーネントにおいては RTM_ERR を返す。

6.1.5.74 virtual RtmRes RTM::RtcBase::rtc_error_do () [inline, virtual]

[CORBA interface] do: error() メソッド.

ERROR 状態にいる間周期実行されるメソッド。

6.1.5.75 virtual RtmRes RTM::RtcBase::rtc_error_entry () [inline, virtual]

[CORBA interface] entry: error() メソッド

ERROR 状態へ進入するときに 1 度だけ呼び出されるメソッド。

6.1.5.76 virtual RtmRes RTM::RtcBase::rtc_error_exit () [inline, virtual]

[CORBA interface] exit: error() メソッド.

ERROR 状態から出るときに 1 度だけ実行されるメソッド。

6.1.5.77 virtual RtmRes RTM::RtcBase::rtc_exit () throw (CORBA::SystemException, RTM::RTComponent::IllegalTransition) [virtual]

[CORBA interface] コンポーネントのリセット

コンポーネントの状態を EXITING に遷移させる。 EXITING 状態に遷移したコンポーネントは二度と復帰することなく終了する。

6.1.5.78 virtual RtmRes RTM::RtcBase::rtc_exiting_entry () [inline, virtual]

[CORBA interface] entry: exiting() メソッド

EXITING 状態へ進入するときに 1 度だけ呼び出されるメソッド。 エラーがなければ READY 状態へ遷移する。 戻り値が RTM_ERR で ERROR 状態へ、 RTM_FATALERR で FATAL_-ERROR 状態へ 遷移する。

6.1.5.79 virtual RtmRes RTM::RtcBase::rtc_fatal_do () [inline, virtual]

[CORBA interface] do: fatal() メソッド.

FATAL_ERROR 状態にいる間周期実行されるメソッド。

6.1.5.80 virtual RtmRes RTM::RtcBase::rtc_fatal_entry () [inline, virtual]

[CORBA interface] entry: fatal() メソッド

FATAL_ERROR 状態へ進入するときに 1 度だけ呼び出されるメソッド。

6.1.5.81 virtual RtmRes RTM::RtcBase::rtc_fatal_exit () [inline, virtual]

[CORBA interface] exit: fatal() メソッド.

READY 状態から出るときに 1 度だけ実行されるメソッド。

6.1.5.82 virtual RTCBase_ptr RTM::RtcBase::rtc_get_component (const char * *name*) [inline, virtual]

[CORBA interface] 子コンポーネントを名前を指定して取得

子コンポーネントを名前を指定してそのオブジェクトリファレンスを取得する。 単体コンポーネントにおいては自分自身オブジェクトリファレンスを返す。

6.1.5.83 virtual RtmRes RTM::RtcBase::rtc_init_entry () [inline, virtual]

[CORBA interface] entry: init() メソッド

INITIALIZING 状態へ進入するときに 1 度だけ呼び出されるメソッド。 エラーがなければ READY 状態へ遷移する。 戻り値が RTM_ERR で ERROR 状態へ、 RTM_FATALERR で FATAL_ERROR 状態へ 遷移する。

6.1.5.84 virtual RtmRes RTM::RtcBase::rtc_kill () [virtual]

[CORBA interface] コンポーネントの強制終了

FATAL_ERROR 状態のコンポーネントを EXITING に遷移させる。 EXITING 状態に遷移したコンポーネントは二度と復帰することなく終了する。

6.1.5.85 virtual RtmRes RTM::RtcBase::rtc_ready_do () [inline, virtual]

[CORBA interface] do: ready() メソッド.

READY 状態にいる間周期実行されるメソッド。

6.1.5.86 virtual RtmRes RTM::RtcBase::rtc_ready_entry () [inline, virtual]

[CORBA interface] entry: ready() メソッド

READY 状態へ進入するときに 1 度だけ呼び出されるメソッド。

6.1.5.87 virtual RtmRes RTM::RtcBase::rtc_ready_exit () [inline, virtual]

[CORBA interface] exit: ready() メソッド.

READY 状態から出るときに 1 度だけ実行されるメソッド。

6.1.5.88 virtual RtmRes RTM::RtcBase::rtc_replace_component (RTCBase_ptr *comp1*, RTCBase_ptr *comp2*) [inline, virtual]

[CORBA interface] 子コンポーネントの順序を入れ替える

2 つの子コンポーネントをオブジェクトリファレンスを使用して順序を入れ替え る。 単体コンポーネントにおいてこのメソッドは未使用のため、RTM_ERR を返す。

6.1.5.89 virtual RtmRes RTM::RtcBase::rtc_replace_component_by_name (const char * *name1*, const char * *name2*) [inline, virtual]

[CORBA interface] 子コンポーネントの順序を入れ替える

2 つの子コンポーネントの順序をコンポーネント名を指定して入れ替える。 単体コンポーネントにおいてこのメソッドは未使用のため、RTM_ERR を返す。

6.1.5.90 virtual RtmRes RTM::RtcBase::rtc_reset () throw (CORBA::SystemException, RTM::RTComponent::IllegalTransition) [virtual]

[CORBA interface] コンポーネントのリセット

コンポーネントの状態を ERROR から INITIALIZE に遷移させる。 INITIALIZE 後エラーがなければすぐに READY 状態に遷移する。 このオペレーションを発行するとき、コンポーネントは ERROR 状態でなければ ならない。他の状態の場合には IllegalTransition 例外が発生する。

6.1.5.91 virtual RtmRes RTM::RtcBase::rtc_set_parent (RTCBase_ptr *comp*) [virtual]

[CORBA interface] 親コンポーネントをセットする

親コンポーネントのオブジェクトリファレンスをセットする。

6.1.5.92 virtual RtmRes RTM::RtcBase::rtc_start () throw (CORBA::SystemException, RTM::RTComponent::IllegalTransition) [virtual]

[CORBA interface] コンポーネントのアクティブ化

コンポーネントの状態を READY から ACTIVE に遷移させる。 このオペレーションを発行するとき、コンポーネントは READY 状態でなければ ならない。他の状態の場合には IllegalTransition 例外が発生する。

6.1.5.93 virtual RtmRes RTM::RtcBase::rtc_start_thread () [virtual]

[CORBA interface] アクティビティスレッドのスタート

コンポーネントアクティビティの内部スレッドをスタートさせる

6.1.5.94 virtual RtmRes RTM::RtcBase::rtc_starting_entry () [inline, virtual]

[CORBA interface] entry: starting() メソッド

STARTING 状態へ進入するときに 1 度だけ呼び出されるメソッド。 エラーがなければ ACTIVE 状態へ遷移する。 戻り値が RTM_ERR で ERROR 状態へ、 RTM_FATALERR で FATAL_ERROR 状態へ 遷移する。

6.1.5.95 virtual OutPort_ptr RTM::RtcBase::rtc_state () [virtual]

[CORBA interface] アクティビティステータスの OutPort の取得

アクティビティステータスの OutPort のオブジェクトリファレンスを取得する。

6.1.5.96 virtual RtmRes RTM::RtcBase::rtc_stop () throw (CORBA::SystemException, RTM::RTComponent::IllegalTransition) [virtual]

[CORBA interface] コンポーネントの非アクティブ化

コンポーネントの状態を ACTIVE から READY に遷移させる。 このオペレーションを発行するとき、コンポーネントは ACTIVE 状態でなければ ならない。他の状態の場合には IllegalTransition 例外が発生する。

6.1.5.97 virtual RtmRes RTM::RtcBase::rtc_stop_thread () [virtual]

[CORBA interface] アクティビティスレッドのストップ

コンポーネントアクティビティの内部スレッドをストップさせる

6.1.5.98 virtual RtmRes RTM::RtcBase::rtc_stopping_entry () [inline, virtual]

[CORBA interface] entry: stopping() メソッド

STOPPING 状態へ進入するときに 1 度だけ呼び出されるメソッド。 エラーがなければ READY 状態へ遷移する。 戻り値が RTM_ERR で ERROR 状態へ、 RTM_FATALERR で FATAL_ERROR 状態へ 遷移する。

6.1.5.99 virtual RtmRes RTM::RtcBase::rtc_suspend_thread () [virtual]

[CORBA interface] アクティビティスレッドのサスペンド

コンポーネントアクティビティの内部スレッドをサスペンドさせる

6.1.5.100 virtual RtmRes RTM::RtcBase::rtc_worker () [virtual]

[CORBA interface] メインアクティビティのメソッド

コンポーネントのアクティビティの本体はこのメソッドを周期実行することに より処理される。単体のコンポーネントでは内部的なスレッドによりこのメソ ッドを周期呼出することで処理を行っている。 スレッドを停止させ、外部からこのオペレーションを呼び出すことにより、任 意のタイミングでアクティビティを実行することも出来る。

6.1.5.101 virtual string RTM::RtcBase::setComponentName (int *num*) [virtual]

コンポーネント名をセットする

コンポーネントのモジュール名に引数で与えられる数字を付加した コンポーネント名を返す。 ユーザはこのメソッドをオーバーライドして、コンポーネント名の命名 方法をカスタマイズすることが出来る。

引数:

-4pt ***num*** コンポーネントのインスタンス番号

6.1.5.102 void RTM::RtcBase::setNamingPolicy (NamingPolicy *policy*)

6.1.5.103 virtual int RTM::RtcBase::svc (void) [virtual]

コンポーネントのアクティビティスレッド関数

コンポーネントの内部アクティビティスレッドの実行関数。 これは ACE_Task サービスクラスメソッドのオーバーライド。

6.1.5.104 virtual char* RTM::RtcBase::version () [virtual]

[CORBA interface] version の取得

コンポーネントのバージョンを取得する。

6.1.5.105 void RTM::RtcBase::writeAllOutPorts ()

全ての OutPort のデータ書き出し.

全ての OutPort に対して OutPort::write() を実行。予めバインドされた変数の 値をサブスクライバに書き出すためのバッファに書き込まれる。 変数にバインドされていない InPort では何も起こらない。

6.1.6 変数

6.1.6.1 StateFunc RTM::RtcBase::_do_func[11] [protected]

アクティビティの do 関数テーブル

6.1.6.2 StateFunc RTM::RtcBase::_entry_func[11] [protected]

アクティビティの entry 関数テーブル

6.1.6.3 StateFunc RTM::RtcBase::_exit_func[11] [protected]

アクティビティの exit 関数テーブル

6.1.6.4 std::list<string> RTM::RtcBase::m_Alias [protected]

コンポーネント名の alias

6.1.6.5 ComponentStateMtx RTM::RtcBase::m_CurrentState [protected]

アクティビティの現在状態変数

6.1.6.6 InPorts RTM::RtcBase::m_InPorts [protected]

mutex 付き InPort リスト

6.1.6.7 RtcMedLogbuf RTM::RtcBase::m_MedLogbuf [protected]

ロガー仲介バッファ

6.1.6.8 NamingPolicy RTM::RtcBase::m_NamingPolicy [protected]

6.1.6.9 ComponentStateMtx RTM::RtcBase::m_NextState [protected]

アクティビティの次状態変数

6.1.6.10 OutPorts RTM::RtcBase::m_OutPorts [protected]

mutex 付き OutPort リスト

6.1.6.11 RTCBase_var RTM::RtcBase::m_Parent [protected]

親コンポーネントのオブジェクトリファレンス

6.1.6.12 RtcManager* RTM::RtcBase::m_pManager [protected]

Manager ポインタ変数.

6.1.6.13 CORBA::ORB_ptr RTM::RtcBase::m_pORB [protected]

ORB ポインタ変数.

6.1.6.14 PortableServer::POA_ptr RTM::RtcBase::m_pPOA [protected]

POA ポインタ変数.

6.1.6.15 RtcModuleProfile RTM::RtcBase::m_Profile [protected]

コンポーネントプロファイル

6.1.6.16 OutPortAny<TimedState> RTM::RtcBase::m_StatePort [protected]

アクティビティ状態 OutPort

6.1.6.17 ThreadState RTM::RtcBase::m_ThreadState [protected]

アクティビティスレッド状態変数

6.1.6.18 TimedState RTM::RtcBase::m_TimedState [protected]

Input port flag list アクティビティ状態変数.

6.1.6.19 RtcLogStream RTM::RtcBase::rtcout [protected]

ロガーストリーム

このクラスの説明は次のファイルから生成されました:

- **RtcBase.h**

6.2 構造体 RTM::RtcBase::ComponentStateMtx

アクティビティ状態変数クラス

`#include <RtcBase.h>`

Public メソッド

- **ComponentStateMtx** ()
  アクティビティ状態変数クラスコンストラクタ

Public 変数

- **ComponentState _state**
  アクティビティ状態
- ACE_Thread_Mutex **_mutex**
  アクティビティ状態変数 *Mutex*

6.2.1 説明

アクティビティ状態変数クラス

6.2.2 コンストラクタとデストラクタ

6.2.2.1 RTM::RtcBase::ComponentStateMtx::ComponentStateMtx () `[inline]`

アクティビティ状態変数クラスコンストラクタ

6.2.3 変数

6.2.3.1 ACE_Thread_Mutex RTM::RtcBase::ComponentStateMtx::_mutex

アクティビティ状態変数 Mutex

6.2.3.2 ComponentState RTM::RtcBase::ComponentStateMtx::_state

アクティビティ状態

この構造体の説明は次のファイルから生成されました:

- **RtcBase.h**

6.3 クラス RTM::RtcBase::eq_name

コンポーネント名比較 Fanctor クラス

`#include <RtcBase.h>`

Public メソッド

- **eq_name** (const char *name)
- bool **operator()** (**InPortBase** *ch)
- bool **operator()** (**OutPortBase** *ch)

Public 変数

- const string **m_name**

6.3.1 説明

コンポーネント名比較 Fanctor クラス

6.3.2 コンストラクタとデストラクタ

6.3.2.1 RTM::RtcBase::eq_name::eq_name (const char * *name*) `[inline]`

6.3.3 関数

6.3.3.1 bool RTM::RtcBase::eq_name::operator() (OutPortBase * *ch*) `[inline]`

6.3.3.2 bool RTM::RtcBase::eq_name::operator() (InPortBase * *ch*) `[inline]`

6.3.4 変数

6.3.4.1 const string RTM::RtcBase::eq_name::m_name

このクラスの説明は次のファイルから生成されました:

- **RtcBase.h**

6.4 構造体 RTM::RtcBase::InPorts

mutex 付き InPort リスト構造体

`#include <RtcBase.h>`

Public 変数

- list< **InPortBase** * > **m_List**
- ACE_Thread_Mutex **m_Mutex**

6.4.1 説明

mutex 付き InPort リスト構造体

6.4.2 変数

6.4.2.1 list<InPortBase*> RTM::RtcBase::InPorts::m_List

6.4.2.2 ACE_Thread_Mutex RTM::RtcBase::InPorts::m_Mutex

この構造体の説明は次のファイルから生成されました:

- **RtcBase.h**

6.5 構造体 RTM::RtcBase::OutPorts

mutex 付き OutPort リスト構造体

`#include <RtcBase.h>`

Public 変数

- list< **OutPortBase** * > **m_List**
- ACE_Thread_Mutex **m_Mutex**

6.5.1 説明

mutex 付き OutPort リスト構造体

6.5.2 変数

6.5.2.1 list<OutPortBase*> RTM::RtcBase::OutPorts::m_List

6.5.2.2 ACE_Thread_Mutex RTM::RtcBase::OutPorts::m_Mutex

この構造体の説明は次のファイルから生成されました:

- **RtcBase.h**

6.6 クラス RTM::RtcBase::ThreadState

```
#include <RtcBase.h>
```

Public メソッド

- **ThreadState** ()

Public 変数

- **ThreadStates m_Flag**
- ACE_Thread_Mutex **m_Mutex**

6.6.1 コンストラクタとデストラクタ

6.6.1.1 RTM::RtcBase::ThreadState::ThreadState () [inline]

6.6.2 変数

6.6.2.1 ThreadStates RTM::RtcBase::ThreadState::m_Flag

6.6.2.2 ACE_Thread_Mutex RTM::RtcBase::ThreadState::m_Mutex

このクラスの説明は次のファイルから生成されました:

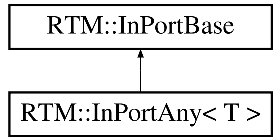
- **RtcBase.h**

6.7 クラス RTM::InPortBase

InPort 基底クラス.

`#include <RtcInPortBase.h>`

RTM::InPortBase に対する継承グラフ:



Public メソッド

- **InPortBase** ()
  InPortBase(p. 134) クラスコンストラクタ.
- virtual ~**InPortBase** ()
  InPortBase(p. 134) クラスデストラクタ.
- virtual void **put** (const CORBA::Any &value)=0 throw (CORBA::SystemException, RTM::InPort::Disconnected)
  [CORBA interface] InPort に値を渡す
- virtual PortProfile * **profile** () throw (CORBA::SystemException)
  [CORBA interface] InPort のプロファイルを取得する
- const char * **name** ()
  InPort の名前を取得する.
- virtual void **read_pm** ()=0
  バインドされた *T* 型の変数に *InPort* バッファの最新値を読み込む

Protected 変数

- PortProfile **m_Profile**
  InPort のプロファイル.

6.7.1 説明

InPort 基底クラス.

InPort の実装である InPort<T> の基底クラス。 CORBA interface への実装を提供する。

6.7.2 コンストラクタとデストラクタ

6.7.2.1 RTM::InPortBase::InPortBase () [inline]

InPortBase(p. 134) クラスコンストラクタ.

InPortBase(p. 134) のクラスコンストラクタ。

6.7.2.2 virtual RTM::InPortBase::~InPortBase () [inline, virtual]

InPortBase(p. 134) クラスデストラクタ.

InPortBase(p. 134) のクラスデストラクタ。

6.7.3 関数

6.7.3.1 const char* RTM::InPortBase::name () [inline]

InPort の名前を取得する.

6.7.3.2 virtual PortProfile* RTM::InPortBase::profile () throw (CORBA::SystemException) [virtual]

[CORBA interface] InPort のプロファイルを取得する

6.7.3.3 virtual void RTM::InPortBase::put (const CORBA::Any & *value*) throw (CORBA::SystemException, RTM::InPort::Disconnected) [pure virtual]

[CORBA interface] InPort に値を渡す

RTM::InPortAny< T > (p. 140) を実装しています.

6.7.3.4 virtual void RTM::InPortBase::read_pm () [pure virtual]

バインドされた T 型の変数に InPort バッファの最新値を読み込む

純粋仮想関数。派生クラスによりオーバーライドされポリモーフィックに 使用される事を企図したメソッド。

RTM::InPortAny< T > (p. 140) を実装しています.

6.7.4 変数

6.7.4.1 PortProfile RTM::InPortBase::m_Profile [protected]

InPort のプロファイル.

このクラスの説明は次のファイルから生成されました:

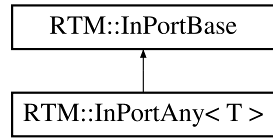
- **RtcInPortBase.h**

6.8 クラス テンプレート RTM::InPortAny< T >

InPort テンプレートクラス.

`#include <RtcInPort.h>`

RTM::InPortAny< T > に対する継承グラフ:



Public メソッド

- **InPortAny** (const char *name, T &value, int bufsize=64)
  InPortAny(p.137) クラスコンストラクタ.
- **InPortAny** (const char *name, int bufsize=64)
  InPortAny(p.137) クラスコンストラクタ.
- virtual ~**InPortAny** ()
  InPortAny(p.137) クラスデストラクタ.
- virtual void **put** (const CORBA::Any &value) throw (RTM::InPort::Disconnected, CORBA::SystemException)
  [CORBA interface] InPort に値を与える
- virtual void **initBuffer** (T &value)
  InPort 内のリングバッファの値を初期化.
- virtual void **read_pm** ()
  バインドされた *T* 型の変数に *InPort* バッファの最新値を読み込む
- virtual T **read** ()
  バインドされた *T* 型の変数に *InPort* バッファの最新値を読み込む
- virtual bool **operator>>** (T &rhs)
  T 型のデータへ *InPort* の最新値データを読み込む.
- virtual bool **isNew** ()
  最新データが未読の新しいデータかどうかを調べる
- virtual int **getNewDataLen** ()
  未読の新しいデータ数を取得する.
- virtual std::vector< T > **getNewList** ()
  未読の新しいデータを取得する.

- virtual std::vector< T > **getNewListReverse** ()
  未読の新しいデータを逆順 *(*新-> 古*)* で取得する*.*

- virtual PortProfile ∗ **profile** () throw (CORBA::SystemException)
  [CORBA interface] InPort のプロファイルを取得する

- const char ∗ **name** ()
  InPort の名前を取得する*.*

Protected 変数

- PortProfile **m_Profile**
  InPort のプロファイル*.*

6.8.1 説明

template<class T> class RTM::InPortAny< T >

InPort テンプレートクラス.

InPort の実装である InPortAny<T> のテンプレートクラス。 は RTCDataType.idl にて定義されている型で、メンバとして Time 型の tm , および T 型の data を持つ構造体でなくてはならない。 InPort は内部にリングバッファを持ち、外部から送信されたデータを順次 このリングバッファに格納する。リングバッファのサイズはデフォルトで 64 と なっているが、コンストラクタ引数によりサイズを指定することができる。 データはフラグによって未読、既読状態が管理され、isNew(), **getNewDataLen()**(p. 139) **getNewList()**(p. 139), **getNewList-Reverse()**(p. 139) 等のメソッドによりハンドリングすることが できる。

6.8.2 コンストラクタとデストラクタ

6.8.2.1 template<class T> RTM::InPortAny< T >::InPortAny (const char ∗ *name*, T & *value*, int *bufsize* = 64) `[inline]`

InPortAny(p. 137) クラスコンストラクタ.

InPortAny<T> クラスのコンストラクタ。 パラメータとして与えられる T 型の変数にバインドされる。

引数:

-4pt ***name*** InPort 名。InPortBase:**name()**(p. 140) により参照される。

value この InPort にバインドされる T 型の変数

bufsize InPort 内部のリングバッファのバッファ長

6.8.2.2 template<class T> RTM::InPortAny< T >::InPortAny (const char * *name*, int *bufsize* = 64) [inline]

InPortAny(p. 137) クラスコンストラクタ.

InPortAny<T> クラスのコンストラクタ。

引数:

-4pt *name* InPort 名。InPortBase:**name()**(p. 140) により参照される。

bufsize InPort 内部のリングバッファのバッファ長

6.8.2.3 template<class T> virtual RTM::InPortAny< T >::~InPortAny () [inline, virtual]

InPortAny(p. 137) クラスデストラクタ.

InPortAny<T> クラスのデストラクタ。

6.8.3 関数

6.8.3.1 template<class T> virtual int RTM::InPortAny< T >::getNewDataLen () [inline, virtual]

未読の新しいデータ数を取得する.

6.8.3.2 template<class T> virtual std::vector<T> RTM::InPortAny< T >::getNewList () [inline, virtual]

未読の新しいデータを取得する.

6.8.3.3 template<class T> virtual std::vector<T> RTM::InPortAny< T >::getNewListReverse () [inline, virtual]

未読の新しいデータを逆順 (新-> 古) で取得する.

6.8.3.4 template<class T> virtual void RTM::InPortAny< T >::initBuffer (T & *value*) [inline, virtual]

InPort 内のリングバッファの値を初期化.

InPort 内のリングバッファの値を初期化する。

6.8.3.5 template<class T> virtual bool RTM::InPortAny< T >::isNew () [inline, virtual]

最新データが未読の新しいデータかどうかを調べる

6.8.3.6 const char* RTM::InPortBase::name () [inline, inherited]

InPort の名前を取得する.

6.8.3.7 template<class T> virtual bool RTM::InPortAny< T >::operator>> (T & *rhs*) [inline, virtual]

T 型のデータへ InPort の最新値データを読み込む.

引数:

-4pt *rhs* InPort バッファから値を読み込む T 型変数

6.8.3.8 virtual PortProfile* RTM::InPortBase::profile () throw (CORBA::SystemException) [virtual, inherited]

[CORBA interface] InPort のプロファイルを取得する

6.8.3.9 template<class T> virtual void RTM::InPortAny< T >::put (const CORBA::Any & *value*) throw (RTM::InPort::Disconnected, CORBA::SystemException) [inline, virtual]

[CORBA interface] InPort に値を与える

InPort に値を put する。

RTM::InPortBase (p. 135) に実装されています.

6.8.3.10 template<class T> virtual T RTM::InPortAny< T >::read () [inline, virtual]

バインドされた T 型の変数に InPort バッファの最新値を読み込む

バインドされた T 型のデータに InPort の最新値を読み込む。 コンストラクタで T 型の変数と InPort がバインドされていなければならない。

6.8.3.11 template<class T> virtual void RTM::InPortAny< T >::read_pm () [inline, virtual]

バインドされた T 型の変数に InPort バッファの最新値を読み込む

バインドされた T 型のデータに InPort の最新値を読み込む。 コンストラクタで T 型の変数と InPort がバインドされていなければならない。 このメソッドはポリモーフィックに使用される事を前提としているため、 型に依存しない引数、戻り値となっている。

RTM::InPortBase (p. 135) に実装されています.

6.8.4 変数

6.8.4.1 PortProfile RTM::InPortBase::m_Profile [protected, inherited]

InPort のプロファイル.

このクラスの説明は次のファイルから生成されました:

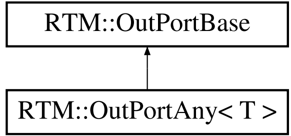
- **RtcInPort.h**

6.9 クラス RTM::OutPortBase

OutPort 基底クラス.

`#include <RtcOutPortBase.h>`

RTM::OutPortBase に対する継承グラフ:



Public メソッド

- **OutPortBase** ()
  OutPortBase(p.142) クラスコンストラクタ.
- virtual ~**OutPortBase** ()
  OutPortBase(p.142) クラスデストラクタ.
- virtual CORBA::Any ∗ **get** ()=0 throw (CORBA::SystemException)
  [CORBA interface] OutPort の現在値を取得する
- virtual RtmRes **subscribe** (InPort_ptr in_port, SubscriptionID_out id, const SubscriberProfile &profile) throw (CORBA::SystemException)
  CORBA interface Return type code of port value. OutPort をサブスクライブする.
- virtual RtmRes **unsubscribe** (const char ∗id) throw (CORBA::SystemException)
  OutPort のサブスクライブを解除する.
- virtual InPortList ∗ **inports** () throw (CORBA::SystemException)
  現在サブスクライブしている *InPort* のリストを取得する
- virtual PortProfile ∗ **profile** () throw (CORBA::SystemException)
  [CORBA interface] OutPort のプロファイルを取得する
- virtual RtmRes **push** (const InPort_ptr &inport, std::string &subsid)=0
  現在の *OutPort* の値をサブスクライバに *push* する
- virtual void **updateall** ()
  現在の *OutPort* の値をサブスクライバに対して更新
- virtual void **disconnect_all** ()
  現在のサブスクライバを全て切断
- virtual const char ∗ **name** ()
  OutPort の名前を取得する.

- virtual void **write_pm** ()=0
  バインドされた *T* 型のデータを *OutPort* の最新値として書き込む

Protected メソッド

- virtual RtmRes **unsubscribeNoLocked** (const char *id)

Protected 変数

- **Subscribers m_Subscribers**
- PortProfile **m_Profile**
  OutPort のプロファイル.

6.9.1 説明

OutPort 基底クラス.

OutPort の実装である OutPort<T> の基底クラス。CORBA interface への実装を提供する。

6.9.2 コンストラクタとデストラクタ

6.9.2.1 RTM::OutPortBase::OutPortBase () `[inline]`

OutPortBase(p. 142) クラスコンストラクタ.

OutPortBase(p. 142) のクラスコンストラクタ。

6.9.2.2 virtual RTM::OutPortBase::~OutPortBase () `[inline, virtual]`

OutPortBase(p. 142) クラスデストラクタ.

OutPortBase(p. 142) のクラスデストラクタ。

6.9.3 関数

6.9.3.1 virtual void RTM::OutPortBase::disconnect_all () `[virtual]`

現在のサブスクライバを全て切断

6.9.3.2 virtual CORBA::Any* RTM::OutPortBase::get () throw (CORBA::SystemException) [pure virtual]

[CORBA interface] OutPort の現在値を取得する

RTM::OutPortAny< T > (p. 148), **RTM::OutPortAny< TimedState >** (p. 148), と **RTM::OutPortAny< TimedString >** (p. 148) を実装しています.

6.9.3.3 virtual InPortList* RTM::OutPortBase::inports () throw (CORBA::SystemException) [virtual]

現在サブスクライブしている InPort のリストを取得する

6.9.3.4 virtual const char* RTM::OutPortBase::name () [inline, virtual]

OutPort の名前を取得する.

6.9.3.5 virtual PortProfile* RTM::OutPortBase::profile () throw (CORBA::SystemException) [virtual]

[CORBA interface] OutPort のプロファイルを取得する

6.9.3.6 virtual RtmRes RTM::OutPortBase::push (const InPort_ptr & *inport*, std::string & *subsid*) [pure virtual]

現在の OutPort の値をサブスクライバに push する

RTM::OutPortAny< T > (p. 149), **RTM::OutPortAny< TimedState >** (p. 149), と **RTM::OutPortAny< TimedString >** (p. 149) を実装しています.

6.9.3.7 virtual RtmRes RTM::OutPortBase::subscribe (InPort_ptr *in_port*, SubscriptionID_out *id*, const SubscriberProfile & *profile*) throw (CORBA::SystemException) [virtual]

CORBA interface Return type code of port value. OutPort をサブスクライブする.

6.9.3.8 virtual RtmRes RTM::OutPortBase::unsubscribe (const char * *id*) throw (CORBA::SystemException) [virtual]

OutPort のサブスクライブを解除する.

6.9.3.9 virtual RtmRes RTM::OutPortBase::unsubscribeNoLocked (const char * *id*) [protected, virtual]

6.9.3.10 virtual void RTM::OutPortBase::updateall () [virtual]

現在の OutPort の値をサブスクライバに対して更新

6.9.3.11 virtual void RTM::OutPortBase::write_pm () [pure virtual]

バインドされた T 型のデータを OutPort の最新値として書き込む

純粋仮想関数。派生クラスによりオーバーライドされポリモーフィックに 使用される事を企図したメソッド。

RTM::OutPortAny< T > (p. 150), **RTM::OutPortAny< TimedState >** (p. 150), と **RTM::OutPortAny< TimedString >** (p. 150) を実装しています.

6.9.4 変数

6.9.4.1 PortProfile RTM::OutPortBase::m_Profile [protected]

OutPort のプロファイル.

6.9.4.2 Subscribers RTM::OutPortBase::m_Subscribers [protected]

このクラスの説明は次のファイルから生成されました:

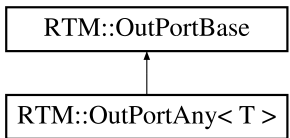
- **RtcOutPortBase.h**

6.10 クラス テンプレート RTM::OutPortAny< T >

OutPort テンプレートクラス.

`#include <RtcOutPort.h>`

RTM::OutPortAny< T > に対する継承グラフ:



Public メソッド

- **OutPortAny** (const char *name, T &value, int bufsize=DEFAULT_BUFFER_SIZE)
  OutPortAny(p.146) クラスコンストラクタ.

- **OutPortAny** (const char *name, int bufsize=DEFAULT_BUFFER_SIZE)
  OutPortAny(p.146) クラスコンストラクタ.

- virtual ~**OutPortAny** ()
  OutPortAny(p.146) クラスデストラクタ.

- CORBA::Any * **get** () throw (CORBA::SystemException)
  [CORBA interface] 現在の *OutPort* の最新の値を取得

- RtmRes **push** (const InPort_ptr &inport, std::string &subsid)
  現在の *OutPort* の値をサブスクライバに転送

- virtual void **initBuffer** (T &value)
  OutPort 内のリングバッファの値を初期化.

- virtual void **write** ()
  バインドされた *T* 型のデータを *OutPort* の最新値として書き込む

- virtual void **write_pm** ()
  バインドされた *T* 型のデータを *OutPort* の最新値として書き込む

- virtual void **write** (T value)
  T 型のデータを *OutPort* の最新値として書き込む.

- virtual void **operator<<** (T &value)
  T 型のデータを *OutPort* の最新値として書き込む.

- virtual RtmRes **subscribe** (InPort_ptr in_port, SubscriptionID_out id, const SubscriberProfile &profile) throw (CORBA::SystemException)
  CORBA interface Return type code of port value. OutPort をサブスクライブする.

- virtual RtmRes **unsubscribe** (const char *id) throw (CORBA::SystemException)
  OutPort のサブスクライブを解除する.

- virtual InPortList * **inports** () throw (CORBA::SystemException)
  現在サブスクライブしている *InPort* のリストを取得する

- virtual PortProfile * **profile** () throw (CORBA::SystemException)
  [CORBA interface] OutPort のプロファイルを取得する

- virtual void **updateall** ()
  現在の *OutPort* の値をサブスクライバに対して更新

- virtual void **disconnect_all** ()
  現在のサブスクライバを全て切断

- virtual const char * **name** ()
  OutPort の名前を取得する.

Protected メソッド

- virtual RtmRes **unsubscribeNoLocked** (const char *id)

Protected 変数

- Subscribers **m_Subscribers**
- PortProfile **m_Profile**
  OutPort のプロファイル.

6.10.1 説明

template<class T> class RTM::OutPortAny< T >

OutPort テンプレートクラス.

OutPort の実装である OutPortAny<T> のテンプレートクラス。 は RTCDataType.idl にて定義されている型で、メンバとして Time 型の tm , および T 型の data を持つ構造体でなくてはならない。

6.10.2 コンストラクタとデストラクタ

6.10.2.1 template<class T> RTM::OutPortAny< T >::OutPortAny (const char * *name*, T & *value*, int *bufsize* = DEFAULT_BUFFER_SIZE) [inline]

OutPortAny(p. 146) クラスコンストラクタ.

OutPortAny<T> クラスのコンストラクタ。 パラメータとして与えられる T 型の変数にバインドされる。

引数:

-4pt *name* OutPort 名。OutPortBase:**name()**(p. 149) により参照される。

value この OutPort にバインドされる T 型の変数

bufsize OutPort 内部のリングバッファのバッファ長

6.10.2.2 template<class T> RTM::OutPortAny< T >::OutPortAny (const char * *name*, int *bufsize* = DEFAULT_BUFFER_SIZE) [inline]

OutPortAny(p. 146) クラスコンストラクタ.

OutPortAny<T> クラスのコンストラクタ。

引数:

-4pt *name* OutPort 名。OutPortBase:**name()**(p. 149) により参照される。

bufsize OutPort 内部のリングバッファのバッファ長

6.10.2.3 template<class T> virtual RTM::OutPortAny< T >::~OutPortAny () [inline, virtual]

OutPortAny(p. 146) クラスデストラクタ.

OutPortAny<T> クラスのデストラクタ。

6.10.3 関数

6.10.3.1 virtual void RTM::OutPortBase::disconnect_all () [virtual, inherited]

現在のサブスクライバを全て切断

6.10.3.2 template<class T> CORBA::Any* RTM::OutPortAny< T >::get () throw (CORBA::SystemException) [inline, virtual]

[CORBA interface] 現在の OutPort の最新の値を取得

現在の OutPort の最新の値を取得する

RTM::OutPortBase (p. 144) に実装されています.

6.10.3.3 template<class T> virtual void RTM::OutPortAny< T >::initBuffer (T & *value*) [inline, virtual]

OutPort 内のリングバッファの値を初期化.

OutPort 内のリングバッファの値を初期化する。

6.10.3.4 virtual InPortList* RTM::OutPortBase::inports () throw (CORBA::SystemException) [virtual, inherited]

現在サブスクライブしている InPort のリストを取得する

6.10.3.5 virtual const char* RTM::OutPortBase::name () [inline, virtual, inherited]

OutPort の名前を取得する.

6.10.3.6 template<class T> virtual void RTM::OutPortAny< T >::operator<< (T & *value*) [inline, virtual]

T 型のデータを OutPort の最新値として書き込む.

引数として与えられた T 型のデータを OutPort の最新値として書き込む。

引数:

-4pt *value* OutPort バッファに書き込む T 型の値

6.10.3.7 virtual PortProfile* RTM::OutPortBase::profile () throw (CORBA::SystemException) [virtual, inherited]

[CORBA interface] OutPort のプロファイルを取得する

6.10.3.8 template<class T> RtmRes RTM::OutPortAny< T >::push (const InPort_ptr & *inport*, std::string & *subsid*) [inline, virtual]

現在の OutPort の値をサブスクライバに転送

現在の OutPort の値をサブスクライバに転送する

RTM::OutPortBase (p. 144) に実装されています.

6.10.3.9 virtual RtmRes RTM::OutPortBase::subscribe (InPort_ptr *in_port*, SubscriptionID_out *id*, const SubscriberProfile & *profile*) throw (CORBA::SystemException) [virtual, inherited]

CORBA interface Return type code of port value. OutPort をサブスクライブする.

6.10.3.10 virtual RtmRes RTM::OutPortBase::unsubscribe (const char * *id*) throw (CORBA::SystemException) [virtual, inherited]

OutPort のサブスクライブを解除する.

6.10.3.11 virtual RtmRes RTM::OutPortBase::unsubscribeNoLocked (const char * *id*) [protected, virtual, inherited]

6.10.3.12 virtual void RTM::OutPortBase::updateall () [virtual, inherited]

現在の OutPort の値をサブスクライバに対して更新

6.10.3.13 template<class T> virtual void RTM::OutPortAny< T >::write (T *value*) [inline, virtual]

T 型のデータを OutPort の最新値として書き込む.

引数として与えられた T 型のデータを OutPort の最新値として書き込む。

引数:

-4pt *value* OutPort バッファに書き込む T 型の値

6.10.3.14 template<class T> virtual void RTM::OutPortAny< T >::write () [inline, virtual]

バインドされた T 型のデータを OutPort の最新値として書き込む

バインドされた T 型のデータを OutPort の最新値として書き込む。 コンストラクタで T 型の変数と OutPort がバインドされている必要がある。

6.10.3.15 template<class T> virtual void RTM::OutPortAny< T >::write_pm () [inline, virtual]

バインドされた T 型のデータを OutPort の最新値として書き込む

バインドされた T 型のデータを OutPort の最新値として書き込む。 コンストラクタで T 型の変数と OutPort がバインドされている必要がある。 このメソッドはポリモーフィックに使用される事を前提としているため、 型に依存しない引数、戻り値となっている。

RTM::OutPortBase (p. 145) に実装されています.

6.10.4 変数

6.10.4.1 PortProfile RTM::OutPortBase::m_Profile [protected, inherited]

OutPort のプロファイル.

6.10.4.2 Subscribers RTM::OutPortBase::m_Subscribers [protected, inherited]

このクラスの説明は次のファイルから生成されました:

- **RtcOutPort.h**

6.11 クラス RTM::RtcManager

RTComponent マネージャクラス.

`#include <RtcManager.h>`

Public 型

- typedef bool(∗ **RtcComponentInit**)(**RtcManager** ∗manager)
  コンポーネントモジュール初期化関数
- typedef **OutPortAny**< TimedString > **LogOutPort**

Public メソッド

- **RtcManager** (int argc, char ∗∗argv)
  RtcManager(p. 152) クラスコンストラクタ.
- void **shutdown** ()
- virtual ∼**RtcManager** ()
  RtcManager(p. 152) クラスデストラクタ.
- int **open** (void ∗args)
  マネージャタスクをスタートさせる
- int **svc** (void)
  サービスのスレッド関数
- int **close** (unsigned long flags)
- virtual RtmRes **load** (const char ∗pathname, const char ∗initfunc)
  [CORBA interface] モジュールのロード
- virtual RtmRes **unload** (const char ∗pathname)
  [CORBA interface] モジュールのアンロード
- virtual RTCBase_ptr **create_component** (const char ∗module_name, const char ∗category_name, CORBA::String_out instance_name)
  [CORBA interface] コンポーネントの生成
- virtual RtmRes **delete_component** (const char ∗instance_name, const char ∗category_name)
  [CORBA interface] コンポーネントの削除
- virtual RTCFactoryList ∗ **factory_list** ()
  [CORBA interface] コンポーネント *Factory* リストの取得

- virtual RTCBaseList * **component_list** ()
  [CORBA interface] コンポーネントリストの取得

- virtual RtmRes **command** (const char *cmd, CORBA::String_out ret)
  [CORBA interface] 簡易インタプリタ

- void **initManager** ()
  マネージャの初期化

- void **runManager** ()
  マネージャの実行

- void **runManagerNoBlocking** ()
  マネージャの実行 *(*非ブロックモード*)*

- bool **activateManager** ()
  マネージャサーバントのアクティブ化

- void **initModuleProc** (**RtcModuleInitProc** proc)
  モジュール初期化ルーチンの実行

- bool **createCommand** (string cmd, boost::function2< bool, vector< string > &, vector< string > & > func)
  簡易インタプリタコマンドの登録

- bool **registerComponent** (RtcModuleProfile &profile, **RtcNewFunc** new_func, **RtcDeleteFunc** delete_func)
  コンポーネントファクトリの登録

- bool **registerComponent** (RtcFactoryBase *factory)
  コンポーネントファクトリの登録

- **RtcBase** * **createComponent** (const string &module_name, const string &category_name, string &comp_name)
  コンポーネント生成

- **RtcBase** * **createComponent** (const string &module_name, const string &category_name)
  コンポーネント生成

- void **cleanupComponent** (const string &instance_name, const string &category_name)
  コンポーネント削除のための処理

- std::vector< RTCBase_ptr > **findComponents** (const string &comp_name)
  コンポーネント検索

- std::string **bindInOut** (RTCBase_ptr comp_in, const std::string &inp_name, RTCBase_ptr comp_out, const std::string &outp_name, RTM::SubscriptionType sub_type=RTM::OPS_NEW)

  InPort と *OutPort* を接続.

- std::string **bindInOutByName** (const std::string &comp_name_in, const std::string &inp_name, const std::string &comp_name_out, const std::string &outp_name, RTM::SubscriptionType sub_type=RTM::OPS_NEW)

  InPort と *OutPort* を名前により接続.

- CORBA::ORB_var **getORB** ()

  ORB のポインタを取得.

- PortableServer::POA_var **getPOA** ()

  POA のポインタを取得.

- RtcLogbuf & **getLogbuf** ()
- **RtcConfig** & **getConfig** ()
- bool **loadCmd** (const vector< string > &cmd, vector< string > &retval)
- bool **unloadCmd** (const vector< string > &cmd, vector< string > &retval)
- bool **createComponentCmd** (const vector< string > &cmd, vector< string > &retval)
- bool **listComponent** (const vector< string > &cmd, vector< string > &retval)
- bool **listModule** (const vector< string > &cmd, vector< string > &retval)
- bool **commandListCmd** (const vector< string > &cmd, vector< string > &retval)

Public 変数

- **ComponentMap m_Components**

  コンポーネントインスタンスデータベースマップ

- string **m_ManagerName**

  マネージャ名

- RtcLogbuf **m_Logbuf**

  ロガーバッファ

- RtcMedLogbuf **m_MedLogbuf**

  ロガー仲介バッファ

- RtcLogStream **rtcout**

  ロガーストリーム

- TimedString **m_LoggerOut**
- **LogOutPort** * **m_pLoggerOutPort**
- **LogEmitter** * **m_pLogEmitter**
- RTCBase_var **m_pMasterLogger**

6.11.1 説明

RTComponent マネージャクラス.

RtcManager(p. 152) はコンポーネントのロード、生成、破棄などのライフサイクルを管理する。またコンポーネントに対して各種サービスを提供する。

6.11.2 型定義

6.11.2.1 typedef OutPortAny<TimedString> RTM::RtcManager::LogOutPort

6.11.2.2 typedef bool(* RTM::RtcManager::RtcComponentInit)(RtcManager* manager)

コンポーネントモジュール初期化関数

6.11.3 コンストラクタとデストラクタ

6.11.3.1 RTM::RtcManager::RtcManager (int *argc*, char ** *argv*)

RtcManager(p. 152) クラスコンストラクタ.

通常コマンドライン引数を引数としてとる。

6.11.3.2 virtual RTM::RtcManager::~RtcManager () [virtual]

RtcManager(p. 152) クラスデストラクタ.

6.11.4 関数

6.11.4.1 bool RTM::RtcManager::activateManager ()

マネージャサーバントのアクティブ化

6.11.4.2 std::string RTM::RtcManager::bindInOut (RTCBase_ptr *comp_in*, const std::string & *inp_name*, RTCBase_ptr *comp_out*, const std::string & *outp_name*, RTM::SubscriptionType *sub_type* = RTM::OPS_NEW)

InPort と OutPort を接続.

6.11.4.3 std::string RTM::RtcManager::bindInOutByName (const std::string & *comp_name_in*, const std::string & *inp_name*, const std::string & *comp_name_out*, const std::string & *outp_name*, RTM::SubscriptionType *sub_type* = RTM::OPS_NEW)

InPort と OutPort を名前により接続.

6.11.4.4 void RTM::RtcManager::cleanupComponent (const string & *instance_name*, const string & *category_name*)

コンポーネント削除のための処理

6.11.4.5 int RTM::RtcManager::close (unsigned long *flags*)

6.11.4.6 virtual RtmRes RTM::RtcManager::command (const char ∗ *cmd*, CORBA::String_out *ret*) [virtual]

[CORBA interface] 簡易インタプリタ

マネージャの簡易インタプリタコマンドの実行

6.11.4.7 bool RTM::RtcManager::commandListCmd (const vector< string > & *cmd*, vector< string > & *retval*)

6.11.4.8 virtual RTCBaseList∗ RTM::RtcManager::component_list () [virtual]

[CORBA interface] コンポーネントリストの取得

コンポーネントのリストを取得する

6.11.4.9 virtual RTCBase_ptr RTM::RtcManager::create_component (const char ∗ *module_name*, const char ∗ *category_name*, CORBA::String_out *instance_name*) [virtual]

[CORBA interface] コンポーネントの生成

コンポーネントのインスタンスを生成する

引数:

-4pt *comp_name* コンポーネントモジュール名

comp_name インスタンス名 (戻値)

6.11.4.10 bool RTM::RtcManager::createCommand (string *cmd*, boost::function2< bool, vector< string > &, vector< string > & > *func*)

簡易インタプリタコマンドの登録

6.11.4.11 RtcBase* RTM::RtcManager::createComponent (const string & *module_name*, const string & *category_name*)

コンポーネント生成

6.11.4.12 RtcBase* RTM::RtcManager::createComponent (const string & *module_name*, const string & *category_name*, string & *comp_name*)

コンポーネント生成

6.11.4.13 bool RTM::RtcManager::createComponentCmd (const vector< string > & *cmd*, vector< string > & *retval*)

6.11.4.14 virtual RtmRes RTM::RtcManager::delete_component (const char * *instance_name*, const char * *category_name*) [virtual]

[CORBA interface] コンポーネントの削除

コンポーネントのインスタンスを削除する

引数:

-4pt *instance_name* インスタンス名

category_name カテゴリ名

6.11.4.15 virtual RTCFactoryList* RTM::RtcManager::factory_list () [virtual]

[CORBA interface] コンポーネント Factory リストの取得

コンポーネント Factory のリストを取得する

6.11.4.16 std::vector<RTCBase_ptr> RTM::RtcManager::findComponents (const string & *comp_name*)

コンポーネント検索

6.11.4.17 RtcConfig& RTM::RtcManager::getConfig () [inline]

6.11.4.18 RtcLogbuf& RTM::RtcManager::getLogbuf () [inline]

6.11.4.19 CORBA::ORB_var RTM::RtcManager::getORB ()

ORB のポインタを取得.

6.11.4.20 PortableServer::POA_var RTM::RtcManager::getPOA ()

POA のポインタを取得.

6.11.4.21 void RTM::RtcManager::initManager ()

マネージャの初期化

6.11.4.22 void RTM::RtcManager::initModuleProc (RtcModuleInitProc *proc*)

モジュール初期化ルーチンの実行

6.11.4.23 bool RTM::RtcManager::listComponent (const vector< string > & *cmd*, vector< string > & *retval*)

6.11.4.24 bool RTM::RtcManager::listModule (const vector< string > & *cmd*, vector< string > & *retval*)

6.11.4.25 virtual RtmRes RTM::RtcManager::load (const char * *pathname*, const char * *initfunc*) [virtual]

[CORBA interface] モジュールのロード

コンポーネントのモジュールをロードして初期化関数を実行する。

引数:

-4pt *pathname* コンポーネントモジュールファイル名

initfunc 初期化関数名

6.11.4.26 bool RTM::RtcManager::loadCmd (const vector< string > & *cmd*, vector< string > & *retval*)

6.11.4.27 int RTM::RtcManager::open (void * *args*)

マネージャタスクをスタートさせる

6.11.4.28 bool RTM::RtcManager::registerComponent (RtcFactoryBase * *factory*)

コンポーネントファクトリの登録

6.11.4.29 bool RTM::RtcManager::registerComponent (RtcModuleProfile & *profile*, RtcNewFunc *new_func*, RtcDeleteFunc *delete_func*)

コンポーネントファクトリの登録

6.11.4.30 void RTM::RtcManager::runManager ()

マネージャの実行

6.11.4.31 void RTM::RtcManager::runManagerNoBlocking ()

マネージャの実行 (非ブロックモード)

6.11.4.32 void RTM::RtcManager::shutdown ()

6.11.4.33 int RTM::RtcManager::svc (void)

サービスのスレッド関数

6.11.4.34 virtual RtmRes RTM::RtcManager::unload (const char * *pathname*) [virtual]

[CORBA interface] モジュールのアンロード

コンポーネントのモジュールをアンロードする

引数:

-4pt *pathnae* コンポーネントモジュールのファイル名

6.11.4.35 bool RTM::RtcManager::unloadCmd (const vector< string > & *cmd*, vector< string > & *retval*)

6.11.5 変数

6.11.5.1 ComponentMap RTM::RtcManager::m_Components

コンポーネントインスタンスデータベースマップ

m_Components._map[_category_name_][_instance_name_]

6.11.5.2 RtcLogbuf RTM::RtcManager::m_Logbuf

ロガーバッファ

6.11.5.3 TimedString RTM::RtcManager::m_LoggerOut

6.11.5.4 string RTM::RtcManager::m_ManagerName

マネージャ名

6.11.5.5 RtcMedLogbuf RTM::RtcManager::m_MedLogbuf

ロガー仲介バッファ

6.11.5.6 LogEmitter* RTM::RtcManager::m_pLogEmitter

6.11.5.7 LogOutPort* RTM::RtcManager::m_pLoggerOutPort

6.11.5.8 RTCBase_var RTM::RtcManager::m_pMasterLogger

6.11.5.9 RtcLogStream RTM::RtcManager::rtcout

ロガーストリーム

このクラスの説明は次のファイルから生成されました:

- **RtcManager.h**

6.12 クラス RTM::RtcConfig

RtcManager(p. 152) コンフィギュレーションクラス.

`#include <RtcConfig.h>`

Public メソッド

- **RtcConfig** ()
  RtcConfig(p. 161) クラスコンストラクタ.
- **RtcConfig** (int argc, char **argv)
  RtcConfig(p. 161) クラスコンストラクタ.
- virtual ~**RtcConfig** ()
  RtcConfig(p. 161) クラスデストラクタ.
- bool **initConfig** (int argc, char **argv)
  RtcConfig(p. 161) クラスの初期化.
- char ** **getOrbInitArgv** () const
  ORB_init() に渡す引数を取得する.
- int **getOrbInitArgc** () const
  ORB_init() に渡す引数の数を取得する.
- string **getNameServer** () const
  ネームサーバ名を取得
- list< string > & **getComponentLoadPath** ()
  コンポーネントロードパスを取得
- string **getBinName** () const
  現在の実行ファイル名を取得
- string **getOSname** () const
  現在の *OS* 名を取得
- string **getHostname** () const
  現在の *host* 名を取得
- string **getOSrelease** () const
  現在の *OS release level* を取得
- string **getOSversion** () const
  現在の *OS version* を取得
- string **getArch** () const

現在の *machien architecture* を取得

- string **getPid** () const
  現在のプロセス *ID* を取得
- string **getLogFileName** ()
- string **getErrorLogFileName** ()
- int **getLogLevel** ()
- int **getLogLock** ()
- std::string **getLogTimeFormat** ()

Protected メソッド

- bool **parseCommandArgs** (int argc, char **argv)
  コマンドライン引数をパースする
- bool **findConfigFile** ()
  コンフィギュレーションファイルをデフォルトパスから探す
- bool **parseConfigFile** ()
  コンフィギュレーションファイルをパースする
- bool **collectSysInfo** ()
  システム情報を取得する
- void **printUsage** (char *arg)
  ヘルプを表示する
- void **argsToArgv** ()
  引数形式を変換する
- bool **fileExist** (const char *filename)
  ファイル存在チェック
- bool **split** (const string &input, const string &delimiter, list< string > &results)
  文字列の分割.

6.12.1 説明

RtcManager(p. 152) コンフィギュレーションクラス.

コンフィギュレーションファイルを読み込み **RtcManager**(p. 152) のコンフィギュレーションを行う。

6.12.2 コンストラクタとデストラクタ

6.12.2.1 RTM::RtcConfig::RtcConfig () [inline]

RtcConfig(p. 161) クラスコンストラクタ.

RtcConfig(p. 161) クラスのコンストラクタ。

6.12.2.2 RTM::RtcConfig::RtcConfig (int *argc*, char ** *argv*)

RtcConfig(p. 161) クラスコンストラクタ.

RtcConfig(p. 161) クラスのコンストラクタ。

引数:

-4pt ***argc*** コマンドライン引数の数

argv コマンドライン引数の配列

6.12.2.3 virtual RTM::RtcConfig::~RtcConfig () [virtual]

RtcConfig(p. 161) クラスデストラクタ.

6.12.3 関数

6.12.3.1 void RTM::RtcConfig::argsToArgv () [protected]

引数形式を変換する

6.12.3.2 bool RTM::RtcConfig::collectSysInfo () [protected]

システム情報を取得する

6.12.3.3 bool RTM::RtcConfig::fileExist (const char * *filename*) [protected]

ファイル存在チェック

6.12.3.4 bool RTM::RtcConfig::findConfigFile () [protected]

コンフィギュレーションファイルをデフォルトパスから探す

6.12.3.5 string RTM::RtcConfig::getArch () const [inline]

現在の machien architecture を取得

6.12.3.6 string RTM::RtcConfig::getBinName () const [inline]

現在の実行ファイル名を取得

6.12.3.7 list<string>& RTM::RtcConfig::getComponentLoadPath () [inline]

コンポーネントロードパスを取得

コンフィギュレーションファイルから得たコンポーネントロードパスを 取得する。

6.12.3.8 string RTM::RtcConfig::getErrorLogFileName ()

6.12.3.9 string RTM::RtcConfig::getHostname () const [inline]

現在の host 名を取得

6.12.3.10 string RTM::RtcConfig::getLogFileName ()

6.12.3.11 int RTM::RtcConfig::getLogLevel ()

6.12.3.12 int RTM::RtcConfig::getLogLock ()

6.12.3.13 std::string RTM::RtcConfig::getLogTimeFormat ()

6.12.3.14 string RTM::RtcConfig::getNameServer () const [inline]

ネームサーバ名を取得

コンフィギュレーションファイルから得たネームサーバ名を取得する。

6.12.3.15 int RTM::RtcConfig::getOrbInitArgc () const [inline]

ORB_init() に渡す引数の数を取得する.

ORB_init() に渡す引数の数を取得する。

6.12.3.16 char** RTM::RtcConfig::getOrbInitArgv () const [inline]

ORB_init() に渡す引数を取得する.

コンフィギュレーションファイルから得たコンフィギュレーション情報の内 ORB の初期化に必要な情報を ORB_init() に渡す引数として取得する。

6.12.3.17 string RTM::RtcConfig::getOSname () const [inline]

現在の OS 名を取得

6.12.3.18 string RTM::RtcConfig::getOSrelease () const [inline]

現在の OS release level を取得

6.12.3.19 string RTM::RtcConfig::getOSversion () const [inline]

現在の OS version を取得

6.12.3.20 string RTM::RtcConfig::getPid () const [inline]

現在のプロセス ID を取得

6.12.3.21 bool RTM::RtcConfig::initConfig (int *argc*, char ** *argv*)

RtcConfig(p. 161) クラスの初期化.

RtcConfig(p. 161) クラスをコマンドライン引数で初期化する

引数:

-4pt ***argc*** コマンドライン引数の数

argv コマンドライン引数の配列

6.12.3.22 bool RTM::RtcConfig::parseCommandArgs (int *argc*, char ** *argv*) [protected]

コマンドライン引数をパースする

6.12.3.23 bool RTM::RtcConfig::parseConfigFile () [protected]

コンフィギュレーションファイルをパースする

6.12.3.24 void RTM::RtcConfig::printUsage (char * *arg*) [protected]

ヘルプを表示する

6.12.3.25 bool RTM::RtcConfig::split (const string & *input*, const string & *delimiter*, list< string > & *results*) [protected]

文字列の分割.

このクラスの説明は次のファイルから生成されました:

- **RtcConfig.h**

6.13 クラス RTM::RtcNaming

CORBA Naming Service アクセスヘルパークラス.

```
#include <RtcNaming.h>
```

Public メソッド

- **RtcNaming** ()
  RtcNaming(p. 166) クラスコンストラクタ.
- **RtcNaming** (CORBA::ORB_ptr orb)
  RtcNaming(p. 166) クラスコンストラクタ.
- ~**RtcNaming** ()
  RtcNaming(p. 166) クラスデストラクタ.
- bool **initNaming** (const CORBA::ORB_ptr orb)
  RtcNaming(p. 166) クラスデストラクタ.
- CosNaming::NamingContextExt_var **createContext** (CosNaming::NamingContext-Ext_var context, const std::string &id, const std::string &kind)
  ネーミングコンテキストの生成
- bool **createHostContext** (const std::string &id)
  ホストコンテキストの生成
- bool **createManagerContext** (const std::string &id)
  マネージャコンテキストの生成
- bool **createCategoryContext** (const std::string &category)
  カテゴリコンテキストの生成
- bool **createModuleContext** (const std::string &module, const std::string &category)
  モジュールコンテキストの生成
- bool **bindObject** (CosNaming::NamingContextExt_var context, const std::string &id, const std::string &kind, CORBA::Object_ptr obj)
  オブジェクトのバインド
- bool **bindObjectByFullPath** (const std::string &path, CORBA::Object_ptr obj)
  オブジェクトをフルパス指定でバインド
- bool **bindComponent** (const std::string &component, const std::string &module, const std::string &category, CORBA::Object_ptr obj)
  コンポーネントのバインド

- bool **bindManager** (const std::string &id, CORBA::Object_ptr obj)
  マネージャのバインド

- bool **destroyHostContext** ()
  自分のホストコンテキストの削除

- bool **destroyManagerContext** ()
  自分のマネージャコンテキストの削除

- bool **destroyCategoryContext** (const std::string &id)
  カテゴリコンテキストの削除

- bool **destroyModuleContext** (const std::string &module, const std::string &category)
  モジュールコンテキストの削除

- bool **unbindObject** (CosNaming::NamingContextExt_var context, const std::string &id, const std::string &kind)
  オブジェクトのアンバインド

- bool **unbindObjectByFullPath** (const std::string &path)
  オブジェクトをフルパス指定でアンバインド

- bool **unbindLocalComponent** (const std::string category, const std::string module, const std::string instance)
  コンポーネントオブジェクトをアンバインド

- bool **findHostContext** (const std::string &id_seq, **ContextList** &context)
  ホストコンテキストの検索

- bool **findCategoryContext** (const std::string &id_seq, **ContextList** &context)
  カテゴリコンテキストの検索

- bool **findModuleContext** (const std::string &id_seq, **ContextList** &context)
  モジュールコンテキストの検索

- bool **findManagerContext** (const std::string &id_seq, **ContextList** &context)
  マネージャコンテキストの検索

- bool **findManager** (std::string &id, **ObjectList** &objects)
  マネージャオブジェクトリファレンスの検索・取得

- bool **findComponents** (const std::string &id_seq, **ObjectList** &objects)
  コンポーネントオブジェクトリファレンスの検索・取得

Protected メソッド

- bool **destroyRecursive** (CosNaming::NamingContextExt_var context)
  ネーミングコンテキストの再帰的削除
- void **findObjectsRecursive** (CosNaming::NamingContextExt_ptr context, const std::string &path, **ObjectList** &obj)
  オブジェクトリファレンスを再帰的に取得
- void **findContextRecursive** (CosNaming::NamingContextExt_var context, const std::string &path, **ContextList** &context_list)
  ネーミングコンテキストを再帰的に取得
- void **bindObjectRecursive** (CosNaming::NamingContextExt_var context, const std::string &path, CORBA::Object_ptr obj)
  コンテキスト・オブジェクトを再帰的にバインド

6.13.1 説明

CORBA Naming Service アクセスヘルパークラス.

6.13.2 コンストラクタとデストラクタ

6.13.2.1 RTM::RtcNaming::RtcNaming () `[inline]`

RtcNaming(p. 166) クラスコンストラクタ.

6.13.2.2 RTM::RtcNaming::RtcNaming (CORBA::ORB_ptr *orb*) `[inline]`

RtcNaming(p. 166) クラスコンストラクタ.

6.13.2.3 RTM::RtcNaming::~RtcNaming ()

RtcNaming(p. 166) クラスデストラクタ.

6.13.3 関数

6.13.3.1 bool RTM::RtcNaming::bindComponent (const std::string & *component*, const std::string & *module*, const std::string & *category*, CORBA::Object_ptr *obj*)

コンポーネントのバインド

オブジェクトを指定したコンテキスト下にバインド

引数:

-4pt *id* コンポーネントの ID

category コンポーネントのカテゴリ

obj コンポーネントのオブジェクトリファレンス

6.13.3.2 bool RTM::RtcNaming::bindManager (const std::string & *id*, CORBA::Object_ptr *obj*)

マネージャのバインド

マネージャを指定したコンテキスト下にバインド

引数:

-4pt *id* マネージャ名

obj マネージャのオブジェクトリファレンス

6.13.3.3 bool RTM::RtcNaming::bindObject (CosNaming::NamingContextExt_var *context*, const std::string & *id*, const std::string & *kind*, CORBA::Object_ptr *obj*)

オブジェクトのバインド

オブジェクトを指定したコンテキスト下にバインド

引数:

-4pt *context* オブジェクトをバインドするコンテキスト

id オブジェクト ID

kind オブジェクト KIND

obj バインドするオブジェクトのオブジェクトリファレンス

6.13.3.4 bool RTM::RtcNaming::bindObjectByFullPath (const std::string & *path*, CORBA::Object_ptr *obj*)

オブジェクトをフルパス指定でバインド

オブジェクトを指定フルパスに従ってバインドする。

引数:

-4pt *path* オブジェクトをバインドするネーミングツリーのフルパス

obj バインドするオブジェクトのオブジェクトリファレンス

6.13.3.5 void RTM::RtcNaming::bindObjectRecursive (CosNaming::NamingContextExt_var *context*, const std::string & *path*, CORBA::Object_ptr *obj*) [protected]

コンテキスト・オブジェクトを再帰的にバインド

引数:

-4pt *context* バインドを開始するコンテキスト

path 検索パス

obj オブジェクトリファレンス

6.13.3.6 bool RTM::RtcNaming::createCategoryContext (const std::string & *category*)

カテゴリコンテキストの生成

カテゴリコンテキストレベルにマネージャ名のコンテキストを生成する

引数:

-4pt *id* コンテキスト名として与えるカテゴリ名

6.13.3.7 CosNaming::NamingContextExt_var RTM::RtcNaming::createContext (CosNaming::NamingContextExt_var *context*, const std::string & *id*, const std::string & *kind*)

ネーミングコンテキストの生成

与えられたネーミングコンテキスト上に id,kind のコンテキストを生成

引数:

-4pt *context* コンテキストを生成する親ネーミングコンテキスト

id コンテキスト ID

kind コンテキスト KIND

6.13.3.8 bool RTM::RtcNaming::createHostContext (const std::string & *id*)

ホストコンテキストの生成

ホストコンテキストレベルにホスト名のコンテキストを生成する

引数:

-4pt *id* コンテキスト名として与えるホスト名

6.13.3.9 bool RTM::RtcNaming::createManagerContext (const std::string & *id*)

マネージャコンテキストの生成

マネージャコンテキストレベルにマネージャ名のコンテキストを生成する

引数:

-4pt *id* コンテキスト名として与えるマネージャ名

6.13.3.10 bool RTM::RtcNaming::createModuleContext (const std::string & *module*, const std::string & *category*)

モジュールコンテキストの生成

モジュールコンテキストレベルにマネージャ名のコンテキストを生成する

引数:

-4pt *module* コンテキスト名として与えるモジュール名

category モジュールコンテキストを作成するカテゴリ名

6.13.3.11 bool RTM::RtcNaming::destroyCategoryContext (const std::string & *id*)

カテゴリコンテキストの削除

引数:

-4pt *id* カテゴリ名

6.13.3.12 bool RTM::RtcNaming::destroyHostContext ()

自分のホストコンテキストの削除

6.13.3.13 bool RTM::RtcNaming::destroyManagerContext ()

自分のマネージャコンテキストの削除

6.13.3.14 bool RTM::RtcNaming::destroyModuleContext (const std::string & *module*, const std::string & *category*)

モジュールコンテキストの削除

引数:

-4pt *module* モジュール名

category カテゴリ名

6.13.3.15 bool RTM::RtcNaming::destroyRecursive (CosNaming::NamingContextExt_var *context*) [protected]

ネーミングコンテキストの再帰的削除

引数:

-4pt *context* 削除を開始するコンテキスト

6.13.3.16 bool RTM::RtcNaming::findCategoryContext (const std::string & *id_seq*, ContextList & *context*)

カテゴリコンテキストの検索

引数:

-4pt *id_seq* 検索する ID 列

context コンテキスト列 (戻り値)

6.13.3.17 bool RTM::RtcNaming::findComponents (const std::string & *id_seq*, ObjectList & *objects*)

コンポーネントオブジェクトリファレンスの検索・取得

引数:

-4pt *id_seq* 検索する ID 列

object オブジェクトリファレンス列 (戻り値)

6.13.3.18 void RTM::RtcNaming::findContextRecursive (CosNaming::NamingContextExt_var *context*, const std::string & *path*, ContextList & *context_list*) [protected]

ネーミングコンテキストを再帰的に取得

引数:

-4pt *context* 検索を開始するコンテキスト

path 検索パス

context_list コンテキスト列 (戻り値)

6.13.3.19 bool RTM::RtcNaming::findHostContext (const std::string & *id_seq*, ContextList & *context*)

ホストコンテキストの検索

引数:

-4pt *id_seq* 検索する ID 列

context コンテキスト列 (戻り値)

6.13.3.20 bool RTM::RtcNaming::findManager (std::string & *id*, ObjectList & *objects*)

マネージャオブジェクトリファレンスの検索・取得

引数:

-4pt *id_seq* 検索する ID 列

object オブジェクトリファレンス列 (戻り値)

6.13.3.21 bool RTM::RtcNaming::findManagerContext (const std::string & *id_seq*, ContextList & *context*)

マネージャコンテキストの検索

引数:

-4pt *id_seq* 検索する ID 列

context コンテキスト列 (戻り値)

6.13.3.22 bool RTM::RtcNaming::findModuleContext (const std::string & *id_seq*, ContextList & *context*)

モジュールコンテキストの検索

引数:

-4pt *id_seq* 検索する ID 列

context コンテキスト列 (戻り値)

6.13.3.23 void RTM::RtcNaming::findObjectsRecursive (CosNaming::NamingContextExt_ptr *context*, const std::string & *path*, ObjectList & *obj*) [protected]

オブジェクトリファレンスを再帰的に取得

引数:

-4pt *context* 検索を開始するコンテキスト

path 検索パス

obj オブジェクトリファレンス列 (戻り値)

6.13.3.24 bool RTM::RtcNaming::initNaming (const CORBA::ORB_ptr *orb*)

RtcNaming(p. 166) クラスデストラクタ.

引数:

-4pt *orb* ORB へのポインタ

6.13.3.25 bool RTM::RtcNaming::unbindLocalComponent (const std::string *category*, const std::string *module*, const std::string *instance*)

コンポーネントオブジェクトをアンバインド

当該 RtcNaming クラスでバインドされたロングネームのオブジェクトを アンバインドする。

引数:

-4pt *category* アンバインドするオブジェクトのカテゴリ名

module アンバインドするオブジェクトのモジュール名

instance アンバインドするオブジェクトのインスタンス名

6.13.3.26 bool RTM::RtcNaming::unbindObject (CosNaming::NamingContextExt_var *context*, const std::string & *id*, const std::string & *kind*)

オブジェクトのアンバインド

指定したコンテキスト下のオブジェクトをアンバインド

引数:

-4pt *context* オブジェクトをバインドするコンテキスト

id オブジェクト ID

kind オブジェクト KIND

6.13.3.27 bool RTM::RtcNaming::unbindObjectByFullPath (const std::string & *path*)

オブジェクトをフルパス指定でアンバインド

オブジェクトを指定フルパスに従ってアンバインドする。

引数:

-4pt *path* アンバインドするオブジェクトのフルパス

このクラスの説明は次のファイルから生成されました:

- **RtcNaming.h**

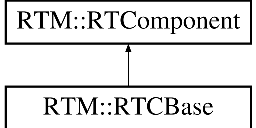
第7章　OpenRTM IDLリファレンス

7.1　インタフェース RTM::RTComponent

RTComponent(p. 175) インターフェース.

```
import "RTComponent.idl";
```

RTM::RTComponent に対する継承グラフ:



Public 型

- typedef short **ComponentState**
  コンポーネントのアクティビティ状態

Public メソッド

- **RtmRes rtc_start** () raises (IllegalTransition)
  コンポーネントのアクティブ化
- **RtmRes rtc_stop** () raises (IllegalTransition)
  コンポーネントの非アクティブ化
- **RtmRes rtc_reset** () raises (IllegalTransition)
  コンポーネントのリセット
- **RtmRes rtc_exit** () raises (IllegalTransition)
  コンポーネントのリセット
- **RtmRes rtc_kill** ()
  コンポーネントの強制終了
- **RtmRes rtc_worker** ()
  メインアクティビティのメソッド

- **InPort get_inport** (in string name) raises (NoSuchName)
  InPort(p. **??**) の取得.

- **OutPort get_outport** (in string name) raises (NoSuchName)
  OutPort(p. 186) の取得.

Public 変数

- readonly attribute string **instance_id**
  コンポーネントのインスタンス *ID*

- readonly attribute string **implementation_id**
  コンポーネントのインプリメンテーション *ID*

- readonly attribute string **description**
  コンポーネントの概要

- readonly attribute string **version**
  コンポーネントのバージョン

- readonly attribute string **maker**
  コンポーネントの作成者

- readonly attribute string **category**
  コンポーネントのカテゴリ

- const **ComponentState RTC_UNKNOWN** = 0
  UNKNOWN state.

- const **ComponentState RTC_BORN** = 1
  BORN state.

- const **ComponentState RTC_INITIALIZING** = 2
  INITIALIZING state.

- const **ComponentState RTC_READY** = 3
  READY state.

- const **ComponentState RTC_STARTING** = 4
  STARTING state.

- const **ComponentState RTC_ACTIVE** = 5
  ACTIVE state.

- const **ComponentState RTC_STOPPING** = 6
  STOPPING state.

- const **ComponentState RTC_ABORTING** = 7
  ABORTING state.

- const **ComponentState RTC_ERROR** = 8
  ERROR state.

- const **ComponentState RTC_FATAL_ERROR** = 9
  FATAL_ERROR state.

- const **ComponentState RTC_EXITING** = 10
  EXITING state.

- readonly attribute **OutPort rtc_state**
  アクティビティステータスの *OutPort* の取得

- readonly attribute **InPortList inports**
  InPortList の取得.

- readonly attribute **OutPortList outports**
  OutPortList の取得.

7.1.1 説明

RTComponent(p. 175) インターフェース.

7.1.2 型定義

7.1.2.1 typedef short RTM::RTComponent::ComponentState

コンポーネントのアクティビティ状態

7.1.3 関数

7.1.3.1 InPort RTM::RTComponent::get_inport (in string *name*) raises (NoSuchName)

InPort(p. ??) の取得.

InPort(p. ??) のオブジェクトリファレンスを取得する。

引数:

-4pt *name* **InPort**(p. ??) 名

7.1.3.2 OutPort RTM::RTComponent::get_outport (in string *name*) raises (NoSuchName)

OutPort(p. 186) の取得.

OutPort(p. 186) のオブジェクトリファレンスを取得する。

引数:

-4pt *name* **OutPort**(p. 186) 名

7.1.3.3 RtmRes RTM::RTComponent::rtc_exit () raises (IllegalTransition)

コンポーネントのリセット

コンポーネントの状態を EXITING に遷移させる。 EXITING 状態に遷移したコンポーネントは二度と復帰することなく終了する。

7.1.3.4 RtmRes RTM::RTComponent::rtc_kill ()

コンポーネントの強制終了

FATAL_ERROR 状態のコンポーネントを EXITING に遷移させる。 EXITING 状態に遷移したコンポーネントは二度と復帰することなく終了する。

7.1.3.5 RtmRes RTM::RTComponent::rtc_reset () raises (IllegalTransition)

コンポーネントのリセット

コンポーネントの状態を ERROR から INITIALIZE に遷移させる。 INITIALIZE 後エラーがなければすぐに READY 状態に遷移する。 このオペレーションを発行するとき、コンポーネントは ERROR 状態でなければ ならない。他の状態の場合には **IllegalTransition**(p. 182) 例外が発生する。

7.1.3.6 RtmRes RTM::RTComponent::rtc_start () raises (IllegalTransition)

コンポーネントのアクティブ化

コンポーネントの状態を READY から ACTIVE に遷移させる。 このオペレーションを発行するとき、コンポーネントは READY 状態でなければ ならない。他の状態の場合には **IllegalTransition**(p. 182) 例外が発生する。

7.1.3.7 RtmRes RTM::RTComponent::rtc_stop () raises (IllegalTransition)

コンポーネントの非アクティブ化

コンポーネントの状態を ACTIVE から READY に遷移させる。 このオペレーションを発行するとき、コンポーネントは ACTIVE 状態でなければ ならない。他の状態の場合には

IllegalTransition(p. 182) 例外が発生する。

7.1.3.8 RtmRes RTM::RTComponent::rtc_worker ()

メインアクティビティのメソッド

コンポーネントのアクティビティの本体はこのメソッドを周期実行することに より処理される。単体のコンポーネントでは内部的なスレッドによりこのメソ ッドを周期呼出することで処理を行っている。 スレッドを停止させ、外部からこのオペレーションを呼び出すことにより、任 意のタイミングでアクティビティを実行することも出来る。

7.1.4 変数

7.1.4.1 readonly attribute string RTM::RTComponent::category

コンポーネントのカテゴリ

7.1.4.2 readonly attribute string RTM::RTComponent::description

コンポーネントの概要

7.1.4.3 readonly attribute string RTM::RTComponent::implementation_id

コンポーネントのインプリメンテーション ID

7.1.4.4 readonly attribute InPortList RTM::RTComponent::inports

InPortList の取得.

InPort(p. ??) のオブジェクトリファレンスのリストを取得する。

7.1.4.5 readonly attribute string RTM::RTComponent::instance_id

コンポーネントのインスタンス ID

7.1.4.6 readonly attribute string RTM::RTComponent::maker

コンポーネントの作成者

7.1.4.7 readonly attribute OutPortList RTM::RTComponent::outports

OutPortList の取得.

OutPort(p. 186) のオブジェクトリファレンスのリストを取得する。

7.1.4.8 const ComponentState RTM::RTComponent::RTC_ABORTING = 7

ABORTING state.

7.1.4.9 const ComponentState RTM::RTComponent::RTC_ACTIVE = 5

ACTIVE state.

7.1.4.10 const ComponentState RTM::RTComponent::RTC_BORN = 1

BORN state.

7.1.4.11 const ComponentState RTM::RTComponent::RTC_ERROR = 8

ERROR state.

7.1.4.12 const ComponentState RTM::RTComponent::RTC_EXITING = 10

EXITING state.

7.1.4.13 const ComponentState RTM::RTComponent::RTC_FATAL_ERROR = 9

FATAL_ERROR state.

7.1.4.14 const ComponentState RTM::RTComponent::RTC_INITIALIZING = 2

INITIALIZING state.

7.1.4.15 const ComponentState RTM::RTComponent::RTC_READY = 3

READY state.

7.1.4.16 const ComponentState RTM::RTComponent::RTC_STARTING = 4

STARTING state.

7.1.4.17 readonly attribute OutPort RTM::RTComponent::rtc_state

アクティビティステータスの OutPort の取得

アクティビティステータスの OutPort のオブジェクトリファレンスを取得する。

7.1.4.18 const ComponentState RTM::RTComponent::RTC_STOPPING = 6

STOPPING state.

7.1.4.19 const ComponentState RTM::RTComponent::RTC_UNKNOWN = 0

UNKNOWN state.

7.1.4.20 readonly attribute string RTM::RTComponent::version

コンポーネントのバージョン

このインタフェースの説明は次のファイルから生成されました:

- **RTComponent.idl**

7.2 例外 RTM::RTComponent::IllegalTransition

不正状態遷移例外.

```
import "RTComponent.idl";
```

7.2.1 説明

不正状態遷移例外.

この例外の説明は次のファイルから生成されました:

- **RTComponent.idl**

7.3 例外 RTM::RTComponent::NoSuchName

不正な名前指定例外.

```
import "RTComponent.idl";
```

Public 変数

- string **name**

7.3.1 説明

不正な名前指定例外.

名前指定で InPort/OutPort 等を取得するとき、該当する名前の オブジェクトがなかった。

7.3.2 変数

7.3.2.1 string RTM::RTComponent::NoSuchName::name

この例外の説明は次のファイルから生成されました:

- **RTComponent.idl**

7.4 例外 RTM::InPort::Disconnected

切断例外.

```
import "RTCInPort.idl";
```

7.4.1 説明

切断例外.

すでに切断しているポートから入力を受け付けたとき発生

この例外の説明は次のファイルから生成されました:

- **RTCInPort.idl**

7.5 構造体 RTM::NamedValue

名前付き変数値.

```
import "RTMBase.idl";
```

Public 変数

- string **name**
  Name of calue.

- any **value**
  Any value.

- short **flag**

7.5.1 説明

名前付き変数値.

任意の型の値を名前付きで格納する構造体

7.5.2 変数

7.5.2.1 short RTM::NamedValue::flag

7.5.2.2 string RTM::NamedValue::name

Name of calue.

7.5.2.3 any RTM::NamedValue::value

Any value.

この構造体の説明は次のファイルから生成されました:

- **RTMBase.idl**

7.6 インタフェース RTM::OutPort

OutPort(p. 186) インターフェース.

```
import "RTCOutPort.idl";
```

Public メソッド

- any **get** ()
  OutPort(p. 186) の値を *Any* 型で取得.

- **RtmRes subscribe** (in **InPort** in_port, out **SubscriptionID** id, in **SubscriberProfile profile**)
  OutPort(p. 186) をサブスクライブする.

- **RtmRes unsubscribe** (in **SubscriptionID** id)
  OutPort(p. 186) をアンサブスクライブする.

Public 変数

- readonly attribute **InPortList inports**
  OutPort(p. 186) をサブスクライブしている *InPort* のリスト.

- readonly attribute **PortProfile profile**
  OutPort(p. 186) のプロファイル.

7.6.1 説明

OutPort(p. 186) インターフェース.

OutPort(p. 186) のインターフェースを定義。

7.6.2 関数

7.6.2.1 any RTM::OutPort::get ()

OutPort(p. 186) の値を Any 型で取得.

7.6.2.2 RtmRes RTM::OutPort::subscribe (in InPort *in_port*, out SubscriptionID *id*, in SubscriberProfile *profile*)

OutPort(p. 186) をサブスクライブする.

7.6.2.3 RtmRes RTM::OutPort::unsubscribe (in SubscriptionID *id*)

OutPort(p. 186) をアンサブスクライブする.

7.6.3 変数

7.6.3.1 readonly attribute InPortList RTM::OutPort::inports

OutPort(p. 186) をサブスクライブしている InPort のリスト.

7.6.3.2 readonly attribute PortProfile RTM::OutPort::profile

OutPort(p. 186) のプロファイル.

このインタフェースの説明は次のファイルから生成されました:

- **RTCOutPort.idl**

7.7 構造体 RTM::PortProfile

InPort/OutPort のプロファイル構造体.

```
import "RTCInPort.idl";
```

Public 変数

- string **name**
  InPort/OutPort の名前.
- CORBA::TypeCode **port_type**
  InPort/OutPort のデータ型.
- **NVList properties**
  InPort/OutPort のプロパティリスト.

7.7.1 説明

InPort/OutPort のプロファイル構造体.

7.7.2 変数

7.7.2.1 string RTM::PortProfile::name

InPort/OutPort の名前.

7.7.2.2 CORBA::TypeCode RTM::PortProfile::port_type

InPort/OutPort のデータ型.

InPort/OutPort のデータ型の CORBA TypeCode を格納する。

7.7.2.3 NVList RTM::PortProfile::properties

InPort/OutPort のプロパティリスト.

この構造体の説明は次のファイルから生成されました:

- **RTCInPort.idl**

7.8 構造体 RTM::SubscriberProfile

SubscriberProfile(p. 189).

```
import "RTCOutPort.idl";
```

Public 変数

- **SubscriptionType subscription_type**
  サブスクライブ型
- boolean **event_base**
  イベント型サブスクライブフラグ
- **NVList properties**
  サブスクライブプロパティ

7.8.1 説明

SubscriberProfile(p. 189).

サブスクライブに関するサブスクライバのプロファイルを定義。

7.8.2 変数

7.8.2.1 boolean RTM::SubscriberProfile::event_base

イベント型サブスクライブフラグ

7.8.2.2 NVList RTM::SubscriberProfile::properties

サブスクライブプロパティ

7.8.2.3 SubscriptionType RTM::SubscriberProfile::subscription_type

サブスクライブ型

サブスクライブに関するサブスクライバのプロファイルを定義。 OPS_ONCE OPS_PERIODIC OPS_NEW OPS_TRIGGERED OSP_PERIODIC_NEW OPS_NEW_PERIODIC OPS_PERIODIC_TRIGGERED OPS_TRIGGERRED_PERIODIC のいずれかを指定。

この構造体の説明は次のファイルから生成されました:

- **RTCOutPort.idl**

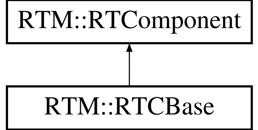
第8章　OpenRTM 拡張IDLリファレンス

8.1　インタフェース RTM::RTCBase

RTCBase(p. 191) インターフェース.

`import "RTCBase.idl";`

RTM::RTCBase に対する継承グラフ:



Public 型

- typedef short **ComponentState**
  コンポーネントのアクティビティ状態

Public メソッド

- **RtmRes rtc_ready_entry** ()
  entry: ready() メソッド
- **RtmRes rtc_ready_do** ()
  do: ready() メソッド.
- **RtmRes rtc_ready_exit** ()
  exit: ready() メソッド.
- **RtmRes rtc_active_entry** ()
  entry: active() メソッド.
- **RtmRes rtc_active_do** ()
  do: active() メソッド.
- **RtmRes rtc_active_exit** ()
  exit: active() メソッド.

- **RtmRes rtc_error_entry** ()
  entry: error() メソッド
- **RtmRes rtc_error_do** ()
  do: error() メソッド.
- **RtmRes rtc_error_exit** ()
  exit: error() メソッド.
- **RtmRes rtc_fatal_entry** ()
  entry: fatal() メソッド
- **RtmRes rtc_fatal_do** ()
  do: fatal() メソッド.
- **RtmRes rtc_fatal_exit** ()
  exit: fatal() メソッド.
- **RtmRes rtc_init_entry** ()
  entry: init() メソッド
- **RtmRes rtc_starting_entry** ()
  entry: starting() メソッド
- **RtmRes rtc_stopping_entry** ()
  entry: stopping() メソッド
- **RtmRes rtc_aborting_entry** ()
  entry: aborting() メソッド
- **RtmRes rtc_exiting_entry** ()
  entry: exiting() メソッド
- **RtmRes rtc_stop_thread** ()
  アクティビティスレッドのスタート
- **RtmRes rtc_start_thread** ()
  アクティビティスレッドのストップ
- **RtmRes rtc_set_parent** (in **RTCBase** comp)
  親コンポーネントをセットする
- **RtmRes rtc_add_component** (in **RTCBase** comp)
  子コンポーネントを追加する
- **RtmRes rtc_delete_component** (in **RTCBase** comp)
  子コンポーネントを削除する

- **RtmRes rtc_replace_component** (in **RTCBase** comp1, in **RTCBase** comp2)
  子コンポーネントの順序を入れ替える

- **RtmRes rtc_replace_component_by_name** (in string name1, in string name2)
  子コンポーネントの順序を入れ替える

- **RTCBaseList rtc_components** ()
  子コンポーネントをリストとして取得する。

- **RTCBase rtc_get_component** (in string name)
  子コンポーネントを名前を指定して取得

- **RtmRes rtc_attach_inport** (in **InPort** in_port)
  InPort をアタッチする.

- **RtmRes rtc_attach_inport_by_name** (in **RTCBase** comp, in string name)
  InPort をアタッチする.

- **RtmRes rtc_detatch_inport** (in **InPort** in_port)
  InPort をデタッチする.

- **RtmRes rtc_detatch_inport_by_name** (in string name)
  InPort をデタッチする.

- **RtmRes rtc_attach_outport** (in **OutPort** out_port)
  OutPort をアタッチする.

- **RtmRes rtc_attach_outport_by_name** (in **RTCBase** comp, in string name)
  OutPort をアタッチする.

- **RtmRes rtc_detatch_outport** (in **OutPort** out_port)
  OutPort をデタッチする.

- **RtmRes rtc_detatch_outport_by_name** (in string name)
  OutPort をデタッチする.

- **RtmRes rtc_start** () raises (IllegalTransition)
  コンポーネントのアクティブ化

- **RtmRes rtc_stop** () raises (IllegalTransition)
  コンポーネントの非アクティブ化

- **RtmRes rtc_reset** () raises (IllegalTransition)
  コンポーネントのリセット

- **RtmRes rtc_exit** () raises (IllegalTransition)
  コンポーネントのリセット

- **RtmRes rtc_kill** ()
  コンポーネントの強制終了

- **RtmRes rtc_worker** ()
  メインアクティビティのメソッド

- **InPort get_inport** (in string name) raises (NoSuchName)
  InPort(p. ??) の取得.

- **OutPort get_outport** (in string name) raises (NoSuchName)
  OutPort(p. 186) の取得.

Public 変数

- readonly attribute RTCProfile **profile**
  コンポーネントのプロファイル構造体

- readonly attribute string **instance_id**
  コンポーネントのインスタンス *ID*

- readonly attribute string **implementation_id**
  コンポーネントのインプリメンテーション *ID*

- readonly attribute string **description**
  コンポーネントの概要

- readonly attribute string **version**
  コンポーネントのバージョン

- readonly attribute string **maker**
  コンポーネントの作成者

- readonly attribute string **category**
  コンポーネントのカテゴリ

- const **ComponentState RTC_UNKNOWN** = 0
  UNKNOWN state.

- const **ComponentState RTC_BORN** = 1
  BORN state.

- const **ComponentState RTC_INITIALIZING** = 2
  INITIALIZING state.

- const **ComponentState RTC_READY** = 3
  READY state.

- const **ComponentState RTC_STARTING** = 4
  STARTING state.

- const **ComponentState RTC_ACTIVE** = 5
  ACTIVE state.

- const **ComponentState RTC_STOPPING** = 6
  STOPPING state.

- const **ComponentState RTC_ABORTING** = 7
  ABORTING state.

- const **ComponentState RTC_ERROR** = 8
  ERROR state.

- const **ComponentState RTC_FATAL_ERROR** = 9
  FATAL_ERROR state.

- const **ComponentState RTC_EXITING** = 10
  EXITING state.

- readonly attribute **OutPort rtc_state**
  アクティビティステータスの *OutPort* の取得

- readonly attribute **InPortList inports**
  InPortList の取得.

- readonly attribute **OutPortList outports**
  OutPortList の取得.

8.1.1 説明

RTCBase(p. 191) インターフェース.

RTComponent(p. 175) インターフェースを継承し、主に複合コンポーネント化に必要な メソッドを追加したインターフェース。 コンポーネントアクティビティ状態に対応するメソッド、アクティビティスレッド の制御、子コンポーネントの追加、削除、置換を行うオペレーションが追加され ている.

8.1.2 型定義

8.1.2.1 typedef short RTM::RTComponent::ComponentState [inherited]

コンポーネントのアクティビティ状態

8.1.3 関数

8.1.3.1 InPort RTM::RTComponent::get_inport (in string *name*) raises (NoSuchName) [inherited]

InPort(p. **??**) の取得.

InPort(p. **??**) のオブジェクトリファレンスを取得する。

引数:

-4pt *name* **InPort**(p. **??**) 名

8.1.3.2 OutPort RTM::RTComponent::get_outport (in string *name*) raises (NoSuchName) [inherited]

OutPort(p. 186) の取得.

OutPort(p. 186) のオブジェクトリファレンスを取得する。

引数:

-4pt *name* **OutPort**(p. 186) 名

8.1.3.3 RtmRes RTM::RTCBase::rtc_aborting_entry ()

entry: aborting() メソッド

ABORTING 状態へ進入するときに 1 度だけ呼び出されるメソッド。 エラーがなければ READY 状態へ遷移する。 戻り値が RTM_ERR で ERROR 状態へ、 RTM_FATALERR で FATAL_ERROR 状態へ 遷移する。

8.1.3.4 RtmRes RTM::RTCBase::rtc_active_do ()

do: active() メソッド.

ACTIVE 状態に居る間周期実行されるメソッド。

8.1.3.5 RtmRes RTM::RTCBase::rtc_active_entry ()

entry: active() メソッド.

ACTIVE 状態に進入するときに 1 度だけ実行されるメソッド。

8.1.3.6 RtmRes RTM::RTCBase::rtc_active_exit ()

exit: active() メソッド.

ACTIVE 状態から出るときに 1 度だけ実行されるメソッド。

8.1.3.7 RtmRes RTM::RTCBase::rtc_add_component (in RTCBase *comp*)

子コンポーネントを追加する

子コンポーネントのオブジェクトリファレンスをセットする。 単体コンポーネントにおいてこのメソッドは未使用のため、RTM_ERR を返す。

8.1.3.8 RtmRes RTM::RTCBase::rtc_attach_inport (in InPort *in_port*)

InPort をアタッチする.

子コンポーネントの InPort をこのコンポーネントの InPort にアタッチする。 単体コンポーネントにおいては自分自身オブジェクトリファレンスを返す。

8.1.3.9 RtmRes RTM::RTCBase::rtc_attach_inport_by_name (in RTCBase *comp*, in string *name*)

InPort をアタッチする.

子コンポーネントの InPort 名を指定してコンポーネントの InPort にアタッチ する。 単体コンポーネントにおいては自分自身オブジェクトリファレンスを返す。

8.1.3.10 RtmRes RTM::RTCBase::rtc_attach_outport (in OutPort *out_port*)

OutPort をアタッチする.

子コンポーネントの OutPort をこのコンポーネントの OutPort にアタッチする。 単体コンポーネントにおいては自分自身オブジェクトリファレンスを返す。

8.1.3.11 RtmRes RTM::RTCBase::rtc_attach_outport_by_name (in RTCBase *comp*, in string *name*)

OutPort をアタッチする.

子コンポーネントの OutPort 名を指定してコンポーネントの InPort にアタッチ する。 単体コンポーネントにおいては自分自身オブジェクトリファレンスを返す。

8.1.3.12 RTCBaseList RTM::RTCBase::rtc_components ()

子コンポーネントをリストとして取得する。

子コンポーネントのリストを取得する。 単体コンポーネントにおいては自分自身オブジェクトリファレンスを返す。

8.1.3.13 RtmRes RTM::RTCBase::rtc_delete_component (in RTCBase *comp*)

子コンポーネントを削除する

子コンポーネントのオブジェクトリファレンスを削除する。 単体コンポーネントにおいてこのメソッドは未使用のため、RTM_ERR を返す。

8.1.3.14 RtmRes RTM::RTCBase::rtc_detatch_inport (in InPort *in_port*)

InPort をデタッチする.

子コンポーネントの InPort をこのコンポーネントの InPort からデタッチする。 単体コンポーネントにおいては RTM_ERR を返す。

8.1.3.15 RtmRes RTM::RTCBase::rtc_detatch_inport_by_name (in string *name*)

InPort をデタッチする.

子コンポーネントの InPort を名前を指定してこのコンポーネントの InPort から デタッチする。 単体コンポーネントにおいては RTM_ERR を返す。

8.1.3.16 RtmRes RTM::RTCBase::rtc_detatch_outport (in OutPort *out_port*)

OutPort をデタッチする.

子コンポーネントの OutPort をこのコンポーネントの OutPort からデタッチする。 単体コンポーネントにおいては RTM_ERR を返す。

8.1.3.17 RtmRes RTM::RTCBase::rtc_detatch_outport_by_name (in string *name*)

OutPort をデタッチする.

子コンポーネントの OutPort を名前を指定してこのコンポーネントの OutPort から デタッチする。 単体コンポーネントにおいては RTM_ERR を返す。

8.1.3.18 RtmRes RTM::RTCBase::rtc_error_do ()

do: error() メソッド.

ERROR 状態にいる間周期実行されるメソッド。

8.1.3.19 RtmRes RTM::RTCBase::rtc_error_entry ()

entry: error() メソッド

ERROR 状態へ進入するときに 1 度だけ呼び出されるメソッド。

8.1.3.20 RtmRes RTM::RTCBase::rtc_error_exit ()

exit: error() メソッド.

ERROR 状態から出るときに 1 度だけ実行されるメソッド。

8.1.3.21 RtmRes RTM::RTComponent::rtc_exit () raises (IllegalTransition) [inherited]

コンポーネントのリセット

コンポーネントの状態を EXITING に遷移させる。 EXITING 状態に遷移したコンポーネントは二度と復帰することなく終了する。

8.1.3.22 RtmRes RTM::RTCBase::rtc_exiting_entry ()

entry: exiting() メソッド

EXITING 状態へ進入するときに 1 度だけ呼び出されるメソッド。エラーがなければ READY 状態へ遷移する。 戻り値が RTM_ERR で ERROR 状態へ、 RTM_FATALERR で FATAL_ERROR 状態へ 遷移する。

8.1.3.23 RtmRes RTM::RTCBase::rtc_fatal_do ()

do: fatal() メソッド.

FATAL_ERROR 状態にいる間周期実行されるメソッド。

8.1.3.24 RtmRes RTM::RTCBase::rtc_fatal_entry ()

entry: fatal() メソッド

FATAL_ERROR 状態へ進入するときに 1 度だけ呼び出されるメソッド。

8.1.3.25 RtmRes RTM::RTCBase::rtc_fatal_exit ()

exit: fatal() メソッド.

READY 状態から出るときに 1 度だけ実行されるメソッド。

8.1.3.26 RTCBase RTM::RTCBase::rtc_get_component (in string *name*)

子コンポーネントを名前を指定して取得

子コンポーネントを名前を指定してそのオブジェクトリファレンスを取得する。 単体コンポーネントにおいては自分自身オブジェクトリファレンスを返す。

8.1.3.27 RtmRes RTM::RTCBase::rtc_init_entry ()

entry: init() メソッド

INITIALIZING 状態へ進入するときに 1 度だけ呼び出されるメソッド。 エラーがなければ READY 状態へ遷移する。 戻り値が RTM_ERR で ERROR 状態へ、 RTM_FATALERR で FATAL_ERROR 状態へ 遷移する。

8.1.3.28 RtmRes RTM::RTComponent::rtc_kill () `[inherited]`

コンポーネントの強制終了

FATAL_ERROR 状態のコンポーネントを EXITING に遷移させる。 EXITING 状態に遷移したコンポーネントは二度と復帰することなく終了する。

8.1.3.29 RtmRes RTM::RTCBase::rtc_ready_do ()

do: ready() メソッド.

READY 状態にいる間周期実行されるメソッド。

8.1.3.30 RtmRes RTM::RTCBase::rtc_ready_entry ()

entry: ready() メソッド

READY 状態へ進入するときに 1 度だけ呼び出されるメソッド。

8.1.3.31 RtmRes RTM::RTCBase::rtc_ready_exit ()

exit: ready() メソッド.

READY 状態から出るときに 1 度だけ実行されるメソッド。

8.1.3.32 RtmRes RTM::RTCBase::rtc_replace_component (in RTCBase *comp1*, in RTCBase *comp2*)

子コンポーネントの順序を入れ替える

2 つの子コンポーネントをオブジェクトリファレンスを使用して順序を入れ替え る。 単体コンポーネントにおいてこのメソッドは未使用のため、RTM_ERR を返す。

8.1.3.33 RtmRes RTM::RTCBase::rtc_replace_component_by_name (in string *name1*, in string *name2*)

子コンポーネントの順序を入れ替える

2つの子コンポーネントの順序をコンポーネント名を指定して入れ替える。 単体コンポーネントにおいてこのメソッドは未使用のため、RTM_ERR を返す。

8.1.3.34 RtmRes RTM::RTComponent::rtc_reset () raises (IllegalTransition) `[inherited]`

コンポーネントのリセット

コンポーネントの状態を ERROR から INITIALIZE に遷移させる。 INITIALIZE 後エラーがなければすぐに READY 状態に遷移する。 このオペレーションを発行するとき、コンポーネントは ERROR 状態でなければ ならない。他の状態の場合には **IllegalTransition**(p. 182) 例外が発生する。

8.1.3.35 RtmRes RTM::RTCBase::rtc_set_parent (in RTCBase *comp*)

親コンポーネントをセットする

親コンポーネントのオブジェクトリファレンスをセットする。

8.1.3.36 RtmRes RTM::RTComponent::rtc_start () raises (IllegalTransition) `[inherited]`

コンポーネントのアクティブ化

コンポーネントの状態を READY から ACTIVE に遷移させる。 このオペレーションを発行するとき、コンポーネントは READY 状態でなければ ならない。他の状態の場合には **IllegalTransition**(p. 182) 例外が発生する。

8.1.3.37 RtmRes RTM::RTCBase::rtc_start_thread ()

アクティビティスレッドのストップ

コンポーネントアクティビティの内部スレッドをストップさせる

8.1.3.38 RtmRes RTM::RTCBase::rtc_starting_entry ()

entry: starting() メソッド

STARTING 状態へ進入するときに1度だけ呼び出されるメソッド。 エラーがなければ ACTIVE 状態へ遷移する。 戻り値が RTM_ERR で ERROR 状態へ、 RTM_FATALERR で FATAL_ERROR 状態へ 遷移する。

8.1.3.39 RtmRes RTM::RTComponent::rtc_stop () raises (IllegalTransition) [inherited]

コンポーネントの非アクティブ化

コンポーネントの状態を ACTIVE から READY に遷移させる。 このオペレーションを発行するとき、コンポーネントは ACTIVE 状態でなければ ならない。他の状態の場合には **IllegalTransition**(p. 182) 例外が発生する。

8.1.3.40 RtmRes RTM::RTCBase::rtc_stop_thread ()

アクティビティスレッドのスタート

コンポーネントアクティビティの内部スレッドをスタートさせる

8.1.3.41 RtmRes RTM::RTCBase::rtc_stopping_entry ()

entry: stopping() メソッド

STOPPING 状態へ進入するときに 1 度だけ呼び出されるメソッド。 エラーがなければ READY 状態へ遷移する。 戻り値が RTM_ERR で ERROR 状態へ、 RTM_FATALERR で FATAL_ERROR 状態へ 遷移する。

8.1.3.42 RtmRes RTM::RTComponent::rtc_worker () [inherited]

メインアクティビティのメソッド

コンポーネントのアクティビティの本体はこのメソッドを周期実行することに より処理される。単体のコンポーネントでは内部的なスレッドによりこのメソ ッドを周期呼出することで処理を行っている。 スレッドを停止させ、外部からこのオペレーションを呼び出すことにより、任 意のタイミングでアクティビティを実行することも出来る。

8.1.4 変数

8.1.4.1 readonly attribute string RTM::RTComponent::category [inherited]

コンポーネントのカテゴリ

8.1.4.2 readonly attribute string RTM::RTComponent::description [inherited]

コンポーネントの概要

8.1.4.3 readonly attribute string RTM::RTComponent::implementation_id [inherited]

コンポーネントのインプリメンテーション ID

8.1.4.4 readonly attribute InPortList RTM::RTComponent::inports [inherited]

InPortList の取得.

InPort(p. ??) のオブジェクトリファレンスのリストを取得する。

8.1.4.5 readonly attribute string RTM::RTComponent::instance_id [inherited]

コンポーネントのインスタンス ID

8.1.4.6 readonly attribute string RTM::RTComponent::maker [inherited]

コンポーネントの作成者

8.1.4.7 readonly attribute OutPortList RTM::RTComponent::outports [inherited]

OutPortList の取得.

OutPort(p. 186) のオブジェクトリファレンスのリストを取得する。

8.1.4.8 readonly attribute RTCProfile RTM::RTCBase::profile

コンポーネントのプロファイル構造体

8.1.4.9 const ComponentState RTM::RTComponent::RTC_ABORTING = 7 [inherited]

ABORTING state.

8.1.4.10 const ComponentState RTM::RTComponent::RTC_ACTIVE = 5 [inherited]

ACTIVE state.

8.1.4.11 const ComponentState RTM::RTComponent::RTC_BORN = 1 [inherited]

BORN state.

8.1.4.12 const ComponentState RTM::RTComponent::RTC_ERROR = 8 [inherited]

ERROR state.

8.1.4.13 const ComponentState RTM::RTComponent::RTC_EXITING = 10 [inherited]

EXITING state.

8.1.4.14 const ComponentState RTM::RTComponent::RTC_FATAL_ERROR = 9 [inherited]

FATAL_ERROR state.

8.1.4.15 const ComponentState RTM::RTComponent::RTC_INITIALIZING = 2 [inherited]

INITIALIZING state.

8.1.4.16 const ComponentState RTM::RTComponent::RTC_READY = 3 [inherited]

READY state.

8.1.4.17 const ComponentState RTM::RTComponent::RTC_STARTING = 4 [inherited]

STARTING state.

8.1.4.18 readonly attribute OutPort RTM::RTComponent::rtc_state [inherited]

アクティビティステータスの OutPort の取得

アクティビティステータスの OutPort のオブジェクトリファレンスを取得する。

8.1.4.19 const ComponentState RTM::RTComponent::RTC_STOPPING = 6 [inherited]

STOPPING state.

8.1.4.20 const ComponentState RTM::RTComponent::RTC_UNKNOWN = 0 [inherited]

UNKNOWN state.

8.1.4.21 readonly attribute string RTM::RTComponent::version [inherited]

コンポーネントのバージョン

このインタフェースの説明は次のファイルから生成されました:

- **RTCBase.idl**

8.2 インタフェース RTM::RTCManager

RTComponent(p. 175) インターフェース.

```
import "RTCManager.idl";
```

Public メソッド

- **RtmRes load** (in string pathname, in string initfunc)
  モジュールのロード
- **RtmRes unload** (in string pathname)
  モジュールのアンロード
- **RTCBase create_component** (in string module_name, in string category_name, out string instance_name)
  コンポーネントの生成
- **RtmRes delete_component** (in string instance_name, in string category_name)
  コンポーネントの削除
- **RTCFactoryList factory_list** ()
  コンポーネント *Factory* リストの取得
- **RTCBaseList component_list** ()
  コンポーネントリストの取得
- **RtmRes command** (in string cmd, out string ret)
  簡易インタプリタ

8.2.1 説明

RTComponent(p. 175) インターフェース.

RTCManager(p. 206) はコンポーネントのロード、生成、破棄などのライフサイクルを管理 する。またコンポーネントに対して各種サービスを提供する。

8.2.2 関数

8.2.2.1 RtmRes RTM::RTCManager::command (in string *cmd*, out string *ret*)

簡易インタプリタ

マネージャの簡易インタプリタコマンドの実行

8.2.2.2 RTCBaseList RTM::RTCManager::component_list ()

コンポーネントリストの取得

コンポーネントのリストを取得する

8.2.2.3 RTCBase RTM::RTCManager::create_component (in string *module_name*, in string *category_name*, out string *instance_name*)

コンポーネントの生成

コンポーネントのインスタンスを生成する

引数:

-4pt *module_name* コンポーネントモジュール名

category_name コンポーネントカテゴリ

instance_name インスタンス名 (戻値)

8.2.2.4 RtmRes RTM::RTCManager::delete_component (in string *instance_name*, in string *category_name*)

コンポーネントの削除

コンポーネントのインスタンスを削除する

引数:

-4pt *comp_name* インスタンス名

8.2.2.5 RTCFactoryList RTM::RTCManager::factory_list ()

コンポーネント Factory リストの取得

コンポーネント Factory のリストを取得する

8.2.2.6 RtmRes RTM::RTCManager::load (in string *pathname*, in string *initfunc*)

モジュールのロード

コンポーネントのモジュールをロードして初期化関数を実行する。

引数:

-4pt *pathname* コンポーネントモジュールファイル名

initfunc 初期化関数名

8.2.2.7 RtmRes RTM::RTCManager::unload (in string *pathname*)

モジュールのアンロード

コンポーネントのモジュールをアンロードする

引数:

-4pt ***pathnae*** コンポーネントモジュールのファイル名

このインタフェースの説明は次のファイルから生成されました:

- **RTCManager.idl**

8.3 構造体 RTM::Time

時刻構造体

```
import "RTMBase.idl";
```

Public 変数

- unsigned long **sec**
- unsigned long **nsec**

8.3.1 説明

時刻構造体

時刻を格納する構造体。データのタイムスタンプなどに使用。

8.3.2 変数

8.3.2.1 unsigned long RTM::Time::nsec

8.3.2.2 unsigned long RTM::Time::sec

この構造体の説明は次のファイルから生成されました:

- **RTMBase.idl**

8.4 構造体 RTM::TimedBoolean

```
import "RTCDataType.idl";
```

Public 変数

- **Time tm**
- boolean **data**

8.4.1 変数

8.4.1.1 boolean RTM::TimedBoolean::data

8.4.1.2 Time RTM::TimedBoolean::tm

この構造体の説明は次のファイルから生成されました:

- **RTCDataType.idl**

8.5 構造体 RTM::TimedBooleanSeq

```
import "RTCDataType.idl";
```

Public 変数

- **Time tm**
- sequence< boolean > **data**

8.5.1 変数

8.5.1.1 sequence<boolean> RTM::TimedBooleanSeq::data

8.5.1.2 Time RTM::TimedBooleanSeq::tm

この構造体の説明は次のファイルから生成されました:

- **RTCDataType.idl**

8.6 構造体 RTM::TimedChar

```
import "RTCDataType.idl";
```

Public 変数

- **Time tm**
- char **data**

8.6.1 変数

8.6.1.1 char RTM::TimedChar::data

8.6.1.2 Time RTM::TimedChar::tm

この構造体の説明は次のファイルから生成されました:

- **RTCDataType.idl**

8.7 構造体 RTM::TimedCharSeq

```
import "RTCDataType.idl";
```

Public 変数

- **Time tm**
- sequence< char > **data**

8.7.1 変数

8.7.1.1 sequence<char> RTM::TimedCharSeq::data

8.7.1.2 Time RTM::TimedCharSeq::tm

この構造体の説明は次のファイルから生成されました:

- **RTCDataType.idl**

8.8 構造体 RTM::TimedDouble

```
import "RTCDataType.idl";
```

Public 変数

- **Time tm**
- double **data**

8.8.1 変数

8.8.1.1 double RTM::TimedDouble::data

8.8.1.2 Time RTM::TimedDouble::tm

この構造体の説明は次のファイルから生成されました:

- **RTCDataType.idl**

8.9 構造体 RTM::TimedDoubleSeq

```
import "RTCDataType.idl";
```

Public 変数

- **Time tm**
- sequence< double > **data**

8.9.1 変数

8.9.1.1 sequence<double> RTM::TimedDoubleSeq::data

8.9.1.2 Time RTM::TimedDoubleSeq::tm

この構造体の説明は次のファイルから生成されました:

- **RTCDataType.idl**

8.10 構造体 RTM::TimedFloat

```
import "RTCDataType.idl";
```

Public 変数

- **Time tm**
- float **data**

8.10.1 変数

8.10.1.1 float RTM::TimedFloat::data

8.10.1.2 Time RTM::TimedFloat::tm

この構造体の説明は次のファイルから生成されました:

- **RTCDataType.idl**

8.11 構造体 RTM::TimedFloatSeq

```
import "RTCDataType.idl";
```

Public 変数

- **Time tm**
- sequence< float > **data**

8.11.1 変数

8.11.1.1 sequence<float> RTM::TimedFloatSeq::data

8.11.1.2 Time RTM::TimedFloatSeq::tm

この構造体の説明は次のファイルから生成されました:

- **RTCDataType.idl**

8.12 構造体 RTM::TimedLong

```
import "RTCDataType.idl";
```

Public 変数

- **Time tm**
- long **data**

8.12.1 変数

8.12.1.1 long RTM::TimedLong::data

8.12.1.2 Time RTM::TimedLong::tm

この構造体の説明は次のファイルから生成されました:

- **RTCDataType.idl**

8.13 構造体 RTM::TimedLongSeq

```
import "RTCDataType.idl";
```

Public 変数

- **Time tm**
- sequence< long > **data**

8.13.1 変数

8.13.1.1 sequence<long> RTM::TimedLongSeq::data

8.13.1.2 Time RTM::TimedLongSeq::tm

この構造体の説明は次のファイルから生成されました:

- **RTCDataType.idl**

8.14 構造体 RTM::TimedOctet

```
import "RTCDataType.idl";
```

Public 変数

- **Time tm**
- octet **data**

8.14.1 変数

8.14.1.1 octet RTM::TimedOctet::data

8.14.1.2 Time RTM::TimedOctet::tm

この構造体の説明は次のファイルから生成されました:

- **RTCDataType.idl**

8.15 構造体 RTM::TimedOctetSeq

```
import "RTCDataType.idl";
```

Public 変数

- **Time tm**
- sequence< octet > **data**

8.15.1 変数

8.15.1.1 sequence<octet> RTM::TimedOctetSeq::data

8.15.1.2 Time RTM::TimedOctetSeq::tm

この構造体の説明は次のファイルから生成されました:

- **RTCDataType.idl**

8.16 構造体 RTM::TimedShort

```
import "RTCDataType.idl";
```

Public 変数

- **Time tm**
- short **data**

8.16.1 変数

8.16.1.1 short RTM::TimedShort::data

8.16.1.2 Time RTM::TimedShort::tm

この構造体の説明は次のファイルから生成されました:

- **RTCDataType.idl**

8.17 構造体 RTM::TimedShortSeq

```
import "RTCDataType.idl";
```

Public 変数

- **Time tm**
- sequence< short > **data**

8.17.1 説明

Sequence data type

8.17.2 変数

8.17.2.1 sequence<short> RTM::TimedShortSeq::data

8.17.2.2 Time RTM::TimedShortSeq::tm

この構造体の説明は次のファイルから生成されました:

- **RTCDataType.idl**

8.18 構造体 RTM::TimedState

```
import "RTCDataType.idl";
```

Public 変数

- **Time tm**
- short **data**

8.18.1 変数

8.18.1.1 short RTM::TimedState::data

8.18.1.2 Time RTM::TimedState::tm

この構造体の説明は次のファイルから生成されました:

- **RTCDataType.idl**

8.19 構造体 RTM::TimedString

```
import "RTCDataType.idl";
```

Public 変数

- **Time tm**
- string **data**

8.19.1 変数

8.19.1.1 string RTM::TimedString::data

8.19.1.2 Time RTM::TimedString::tm

この構造体の説明は次のファイルから生成されました:

- **RTCDataType.idl**

8.20 構造体 RTM::TimedStringSeq

```
import "RTCDataType.idl";
```

Public 変数

- **Time tm**
- sequence< string > **data**

8.20.1 変数

8.20.1.1 sequence<string> RTM::TimedStringSeq::data

8.20.1.2 Time RTM::TimedStringSeq::tm

この構造体の説明は次のファイルから生成されました:

- **RTCDataType.idl**

8.21 構造体 RTM::TimedULong

```
import "RTCDataType.idl";
```

Public 変数

- **Time tm**
- unsigned long **data**

8.21.1 変数

8.21.1.1 unsigned long RTM::TimedULong::data

8.21.1.2 Time RTM::TimedULong::tm

この構造体の説明は次のファイルから生成されました:

- **RTCDataType.idl**

8.22 構造体 RTM::TimedULongSeq

```
import "RTCDataType.idl";
```

Public 変数

- **Time tm**
- sequence< unsigned long > **data**

8.22.1 変数

8.22.1.1 sequence<unsigned long> RTM::TimedULongSeq::data

8.22.1.2 Time RTM::TimedULongSeq::tm

この構造体の説明は次のファイルから生成されました:

- **RTCDataType.idl**

8.23 構造体 RTM::TimedUShort

```
import "RTCDataType.idl";
```

Public 変数

- **Time tm**
- unsigned short **data**

8.23.1 変数

8.23.1.1 unsigned short RTM::TimedUShort::data

8.23.1.2 Time RTM::TimedUShort::tm

この構造体の説明は次のファイルから生成されました:

- **RTCDataType.idl**

8.24 構造体 RTM::TimedUShortSeq

```
import "RTCDataType.idl";
```

Public 変数

- **Time tm**
- sequence< unsigned short > **data**

8.24.1 変数

8.24.1.1 sequence<unsigned short> RTM::TimedUShortSeq::data

8.24.1.2 Time RTM::TimedUShortSeq::tm

この構造体の説明は次のファイルから生成されました:

- **RTCDataType.idl**

付 録 A　依存パッケージ類のビルドについて

A.1　ACE のビルド

A.1.1　ACE (The ADAPTIVE Communication Environment)

The ADAPTIVE Communication Environment (ACE) はオープンソースのフリーなオブジェクト指向フレームワークです。オブジェクト指向を有効に利用するための数多くのデザインパターンが適用されたており、ネットワーク通信のための共通の基盤を提供します。ACE の主な特長は、

- 移植性の向上
- デザインパターンの適用による高品質なソフトウエア開発
- QoS のサポート
- より高レベルなミドルウエア (たとえば CORBA) のための再利用可能なコンポーネントやパターンの提供

などが挙げられます。

ACE は非常に多くのプラットフォームをサポートしており、フラットフォーム間で異なる各種の関数やスレッドモデルを隠蔽し、共通のインターフェースを提供しています。AIST-RTM は、ACE のスレッドモデル、動的ライブラリの読み込み、時刻取得・計測に関する機能、各種デザインパターンを利用して実装されています。ACE がサポートしているプラットフォームは以下のとおりです。

● ACE が動作可能なプラットフォーム一覧

Windows (WinNT 3.5.x, 4.x, 2000, Embedded NT, XP, Win95/98, WinCE) および各種開発環境 (MSVC++, Borland C++ Builder, IBM Visual Age (32-, 64-bit Intel/Alpha) Mac OS X, Solaris 1.x, 2.x (SPARC / Intel), SGI IRIX 5.x, 6.x, DG/UX, HP-UX 9.x, 10.x, 11.x, Tru64UNIX 3.x, 4.x, AIX 3.x, 4.x, 5.x, UnixWare, SCO, Debian Linux 2.x, RedHat Linux 5.2, 6.x, 7.x, 8x, and 9.x, Timesys Linux, FreeBSD, NetBSD, LynxOS, VxWorks, ChorusOS, QnX Neutrino, RTEMS, OS9, PSoS, OpenVMS, MVS OpenEdition, CRAY UNICOS

A.1.2　ACE のダウンロード

ACE および ACE を用いた ORB(TAO) は以下の URL で配布されています。

ACE http://www.cs.wustl.edu/ schmidt/ACE.html

TAO http://www.cs.wustl.edu/ schmidt/TAO.html

以下の URL からソースコードがダウンロードできます。

http://deuce.doc.wustl.edu/Download.html

ソースコードは tar.gz, tar.bz, zip 形式で圧縮されています。それぞれ、適切な方法で展開します。

A.1.3 UNIX 系 OS でのビルド

ビルドは基本的に展開したディレクトリ ACE_wrapper にある INSTALL ファイルの記述に従って行ってください。手順は以下のとおりです。

- 環境変数 ACE_ROOT の設定
- platform.macros.GNU の編集
- config.h の編集
- ビルド

A.1.4 環境変数 ACE_ROOT の設定

まずはじめに、環境変数 ACE_ROOT を設定します。

```
「csh 系の場合」

$ setenv ACE_ROOT [展開したディレクトリ]

「bash 系の場合」

$ ACE_ROOT=[展開したディレクトリ]

$ export ACE_ROOT
```

A.1.4.1 platform.macros.GNU の設定

以後、ACE を展開したディレクトリ ACE_wrapper を $ACE_ROOT として話を進めます。

platform.macros.GNU の設定を行います。ディレクトリ$ACE_ROOT/include/makeinclude/内に、対応プラットフォーム毎の Makefile 用マクロが用意されています。ここでは、Linux および gcc(g++) を用いてビルドするものと仮定します。Linux および gcc(g++) のマクロは

platform_linux.GNU ですので、これを platform.macros.GNU としてシンボリックリンクを張ります。

```
> cd $ACE_ROOT/include/makeinclude/
> ln -s platform_linux.GNU platform.macros.GNU
```

A.1.4.2 config.h の設定

次に、config.h の設定を行います。ディレクトリ$ACE_ROOT/ace/ 内に、対応プラットフォーム毎の config.h が用意されています。プラットフォームは Linux と仮定してますので、config.h を config-linux.h へのシンボリックリンクとして作成します。

```
> cd $ACE_ROOT/ace/
> ln -s config-linux.h config.h
```

A.1.5 ビルド

これで準備が整いました。$ACE_ROOT に移動して make します。

```
> cd $ACE_ROOT
> make
```

ACE はかなり大きなシステムなので、ビルドには若干時間がかかります。

A.2 boost のビルド

A.2.1 The boost C++ Library

boost は C++ のテンプレートライブラリ集です。C++ の標準的なテンプレートライブラリとしてはSTL がありますが、boost はさらに強力な標準ライブラリを求めて、 標準化委員会のメンバー達が立ち上げたプロジェクトによって開発されているテンプレートライブラリ集です。したがって、boost は次の C++の標準ライブラリとなるかもしれません。

現在のところは標準ではないので、ほとんどの開発環境には付属していませんので、入っていなければ自分でビルド・インストールしなければなりません。

ビルドといっても、boost はほとんどがテンプレートライブラリなので、ヘッダをインストールすするだけでビルドする必要のないものがほとんどです。しかし、OpenRTM では boost の正規表現ライブラリを使用しており、これはコンパイルが必要な boost ライブラリのひとつですので、boost をビルドする必要があります。以下では、boost のビルドの仕方を順を追って説明します。

なお、使用しているシステムのパッケージに boost が含まれている場合にはそれを利用してもよいでしょう。

A.2.2 ダウンロード

`http://www.boost.org/` の "Download" メニューから boost をダウンロードしてください。

ダウンロードしたソースコードを展開します。

```
> tar vxzf boost_1_32_0.tar.gz
```

A.2.3 bjam のビルド

まず、boost のビルドツール `bjam` をビルドします。bjam のビルドは付属の `bild.sh` により行います。たいていの場合は、適切に環境を判断して自動的にビルドを行います。

```
> cd boost_1_32_0
> cd tools/build/jam_src
> ./build.sh
   : (ビルドされる)
> ls bin.(system_name)
> ls bin.linuxx86 : 例
> ls
bjam*           jam*            mkjambase*      yyacc*
> cd - (boost ソースのトップディレクトリに戻る)
```

ビルドされたバイナリは bin.(system_name) のディレクトリ内におかれます。

A.2.4 boost のビルド

まず、boost の Python 拡張ライブラリをビルドするためには tools/build/v1/python.jam で定義されている、PYTHON_VERSION を適切なバージョンに設定する必要があります。上記のバージョンではデフォルトは Python2.2 を使用するようになっています。

その他の設定については、付属のドキュメント等を参照してください。

bjam による boost のビルドは以下のように行います。

```
> tools/build/jam_src/bin.(システム名)/bjam -sTOOLS=gcc \
  --prefix=/usr/local -sPYTHON_ROOT=/usr -sPYTHON_VERSION=2.3
       :
ビルドが始まる
       :
```

ここでは、boost のインストールディレクトリを /usr/local/[include|lib] とし、Python のインストールディレクトリを /usr、Python のバージョンを 2.3 としてビルドしています。細かいビルドオプションについては、付属のドキュメントを参照してください。

ビルドは CPU の速度にも依りますが、Pen4 数 GHz 程度のマシンで数十分ほどかかるでしょう。

A.2.5 boost のインストール

最後に root になり、インストールを行います。

```
> su
# tools/build/jam_src/bin.(システム名)/bjam -sTOOLS=gcc \
  --prefix=/usr/local -sPYTHON_ROOT=/usr -sPYTHON_VERSION=2.3 install
         :
インストールされる
         :
```

インストールされた boost の共有ライブラリは、libboost_regex-gcc-1_32.so.1.32.0 や libboost_regex-gcc-1_32.so などのバージョン名、ビルドツール名を含んだ名前でインストールされる場合があります。そういう場合には、

```
# cd /usr/local/lib
# ln -s libboost_regex-gcc-1_32.so.1.32.0 libboost_regex.so
```

のように、シンボリックリンクを張ってやる必要があるかもしれません。とくに OpenRTM をコンパイルする際には少なくとも `libboost_regex.so` が必要となりますので、`/usr/local/lib` や `/usr/lib` の下に `libboost_regex.so` が存在する必要があります。

A.3 omniORB のビルド

A.3.1 omniORB

omniORB は、ORB (object request broker) に準拠した CORBA (Common Object Request Broker Architecture) 2.6 のフリーな実装です。もともとは AT&T Lab. で開発されていましたが、AT&T 研究所の閉鎖にともない、sourceforge にホストされるようになりました。フリーな CORBA 実装の中でも omniORB は TAO や MICO などに比べると CORBA サービス類が少ないといったデメリットがありますが、もっともパフォーマンスが高く、スピードが速い実装といわれています。

A.3.2 omniORB のダウンロード

omniORB の開発は sourceforge で行われていますので、ソースコードは omniORB の sourceforge サイト上の `http://omniorb.sourceforge.net/` からダウンロードできます。google などで omniORB を検索するとたまに昔の AT&T Lab. のページにたどり着くことがありますので間違えないようにしてください。

ダウンロードしたソースコードを任意のディレクトリに展開します。

```
> tar vxzf omniORB-4.0.5.tar.gz
> cd omniORB-4.0.5
```

A.3.3 omniORB のビルド

原則として付属のドキュメントにしたがってビルド・インストールを行います。

以下に、簡単な手順を示します。omniORB のソースディレクトリ下において、以下のような手順でビルドします。基本的には autoconf, automake を利用しているので `./configure; make` のみでビルドできるはずです。

`build` ディレクトリは、`configure` コマンドによって生成されるインストール用のソースファイル、オブジェクトファイルを保存するために作成します。これはマニュアルでも推奨されている方法です。

```
> tar vxzf omniORB-4.0.5.tar.gz
> cd omniORB-4.0.5
> mkdir build
> ../configure [options]
```

`configure` コマンドの主なオプションを以下に示します。

configure オプション

`--prefix=dir`	インストール先のデイレクトリを設定します。
`--disable-static`	静的なライブラリによるビルドを禁止します。
`--enable-thread-tracing`	omniORB におけるスレッドのバグを発見しやすくします。(デフォルト) ただし実行パフォーマンスを多少低下させるので、そうしたくない場合には、`--disable-thread-tracing` オプションを使用してください。
`--with-openssl=dir`	openSSL を使用する場合に、openSSL のインストールされたデイレクトリを指定します。
`--with-omniORB-config=dir/file`	omniORB 設定ファイルの場所とファイル名を指定します。デフォルトは `/etc/omniORB.cfg` です。
`--with-omniNames-logdir=dir/file`	omniORB ログファイルの場所とファイル名を指定します。デフォルトは `/var/omninames` です。

A.3.4 omniORB のインストール

`build` ディレクトリ下で root になり `make install` でインストールします。

```
> su
# make install
```

索 引

_check_error
RTM::RtcBase, 109
_do_func
RTM::RtcBase, 125
_entry_func
RTM::RtcBase, 125
_exit_func
RTM::RtcBase, 125
_mutex
RTM::RtcBase::ComponentStateMtx, 128
_nop
RTM::RtcBase, 109
_rtc_aborting
RTM::RtcBase, 109
_rtc_active
RTM::RtcBase, 109
_rtc_active_entry
RTM::RtcBase, 110
_rtc_active_exit
RTM::RtcBase, 110
_rtc_error
RTM::RtcBase, 110
_rtc_error_entry
RTM::RtcBase, 110
_rtc_error_exit
RTM::RtcBase, 110
_rtc_exiting
RTM::RtcBase, 110
_rtc_fatal
RTM::RtcBase, 110
_rtc_fatal_entry
RTM::RtcBase, 110
_rtc_fatal_exit
RTM::RtcBase, 110
_rtc_initializing
RTM::RtcBase, 110
_rtc_ready
RTM::RtcBase, 111
_rtc_ready_entry
RTM::RtcBase, 111
_rtc_ready_exit
RTM::RtcBase, 111
_rtc_starting
RTM::RtcBase, 111
_rtc_stopping
RTM::RtcBase, 111
_state
RTM::RtcBase::ComponentStateMtx, 128
~InPortAny
RTM::InPortAny, 139
~InPortBase
RTM::InPortBase, 135
~OutPortAny
RTM::OutPortAny, 148
~OutPortBase
RTM::OutPortBase, 143
~RtcBase
RTM::RtcBase, 109
~RtcConfig
RTM::RtcConfig, 163
~RtcManager
RTM::RtcManager, 155
~RtcNaming
RTM::RtcNaming, 168

ACE, 231
activateManager
RTM::RtcManager, 155
appendAlias
RTM::RtcBase, 111
argsToArgv
RTM::RtcConfig, 163

bindComponent
RTM::RtcNaming, 168
bindInOut
RTM::RtcManager, 155
bindInOutByName
RTM::RtcManager, 155

bindManager
RTM::RtcNaming, 169
bindObject
RTM::RtcNaming, 169
bindObjectByFullPath
RTM::RtcNaming, 169
bindObjectRecursive
RTM::RtcNaming, 169

category
RTM::RTCBase, 202
RTM::RtcBase, 112
RTM::RTComponent, 179
cleanupComponent
RTM::RtcManager, 156
close
RTM::RtcBase, 112
RTM::RtcManager, 156
collectSysInfo
RTM::RtcConfig, 163
command
RTM::RTCManager, 206
RTM::RtcManager, 156
commandListCmd
RTM::RtcManager, 156
component_list
RTM::RTCManager, 206
RTM::RtcManager, 156
ComponentState
RTM::RTCBase, 195
RTM::RTComponent, 177
ComponentStateMtx
RTM::RtcBase::ComponentStateMtx, 128
create_component
RTM::RTCManager, 207
RTM::RtcManager, 156
createCategoryContext
RTM::RtcNaming, 170
createCommand
RTM::RtcManager, 156
createComponent
RTM::RtcManager, 157
createComponentCmd
RTM::RtcManager, 157
createContext
RTM::RtcNaming, 170
createHostContext
RTM::RtcNaming, 170
createManagerContext
RTM::RtcNaming, 170
createModuleContext
RTM::RtcNaming, 171

data
RTM::TimedBoolean, 210
RTM::TimedBooleanSeq, 211
RTM::TimedChar, 212
RTM::TimedCharSeq, 213
RTM::TimedDouble, 214
RTM::TimedDoubleSeq, 215
RTM::TimedFloat, 216
RTM::TimedFloatSeq, 217
RTM::TimedLong, 218
RTM::TimedLongSeq, 219
RTM::TimedOctet, 220
RTM::TimedOctetSeq, 221
RTM::TimedShort, 222
RTM::TimedShortSeq, 223
RTM::TimedState, 224
RTM::TimedString, 225
RTM::TimedStringSeq, 226
RTM::TimedULong, 227
RTM::TimedULongSeq, 228
RTM::TimedUShort, 229
RTM::TimedUShortSeq, 230
delete_component
RTM::RTCManager, 207
RTM::RtcManager, 157
deleteInPort
RTM::RtcBase, 112
deleteInPortByName
RTM::RtcBase, 112
deleteOutPort
RTM::RtcBase, 112
deleteOutPortByName
RTM::RtcBase, 112
deletePort
RTM::RtcBase, 113
description
RTM::RTCBase, 202
RTM::RtcBase, 113
RTM::RTComponent, 179
destroyCategoryContext

RTM::RtcNaming, 171
destroyHostContext
RTM::RtcNaming, 171
destroyManagerContext
RTM::RtcNaming, 171
destroyModuleContext
RTM::RtcNaming, 171
destroyRecursive
RTM::RtcNaming, 171
disconnect_all
RTM::OutPortAny, 148
RTM::OutPortBase, 143

eq_name
RTM::RtcBase::eq_name, 130
event_base
RTM::SubscriberProfile, 189

factory_list
RTM::RTCManager, 207
RTM::RtcManager, 157
fileExist
RTM::RtcConfig, 163
finalize
RTM::RtcBase, 113
finalizeInPorts
RTM::RtcBase, 113
finalizeOutPorts
RTM::RtcBase, 113
findCategoryContext
RTM::RtcNaming, 172
findComponents
RTM::RtcManager, 157
RTM::RtcNaming, 172
findConfigFile
RTM::RtcConfig, 163
findContextRecursive
RTM::RtcNaming, 172
findHostContext
RTM::RtcNaming, 172
findManager
RTM::RtcNaming, 173
findManagerContext
RTM::RtcNaming, 173
findModuleContext
RTM::RtcNaming, 173
findObjectsRecursive
RTM::RtcNaming, 173
flag
RTM::NamedValue, 185
forceExit
RTM::RtcBase, 114

get
RTM::OutPort, 186
RTM::OutPortAny, 148
RTM::OutPortBase, 143
get_inport
RTM::RTCBase, 196
RTM::RtcBase, 114
RTM::RTComponent, 177
get_outport
RTM::RTCBase, 196
RTM::RtcBase, 114
RTM::RTComponent, 177
getAliases
RTM::RtcBase, 114
getArch
RTM::RtcConfig, 163
getBinName
RTM::RtcConfig, 163
getComponentLoadPath
RTM::RtcConfig, 164
getConfig
RTM::RtcManager, 157
getErrorLogFileName
RTM::RtcConfig, 164
getHostname
RTM::RtcConfig, 164
getLogbuf
RTM::RtcManager, 158
getLogFileName
RTM::RtcConfig, 164
getLogLevel
RTM::RtcConfig, 164
getLogLock
RTM::RtcConfig, 164
getLogTimeFormat
RTM::RtcConfig, 164
getModuleProfile
RTM::RtcBase, 114
getNameServer
RTM::RtcConfig, 164
getNamingPolicy

RTM::RtcBase, 114
getNewDataLen
RTM::InPortAny, 139
getNewList
RTM::InPortAny, 139
getNewListReverse
RTM::InPortAny, 139
getORB
RTM::RtcManager, 158
getOrbInitArgc
RTM::RtcConfig, 164
getOrbInitArgv
RTM::RtcConfig, 164
getOSname
RTM::RtcConfig, 164
getOSrelease
RTM::RtcConfig, 164
getOSversion
RTM::RtcConfig, 165
getPid
RTM::RtcConfig, 165
getPOA
RTM::RtcManager, 158
getState
RTM::RtcBase, 115

implementation_id
RTM::RTCBase, 202
RTM::RtcBase, 115
RTM::RTComponent, 179
init_orb
RTM::RtcBase, 115
init_state_func_table
RTM::RtcBase, 115
initBuffer
RTM::InPortAny, 139
RTM::OutPortAny, 149
initConfig
RTM::RtcConfig, 165
initManager
RTM::RtcManager, 158
initModuleProc
RTM::RtcManager, 158
initModuleProfile
RTM::RtcBase, 115
initNaming
RTM::RtcNaming, 173
InPort, 7
InPortAny
RTM::InPortAny, 138
InPortBase
RTM::InPortBase, 135
inports
RTM::OutPort, 187
RTM::OutPortAny, 149
RTM::OutPortBase, 144
RTM::RTCBase, 203
RTM::RtcBase, 115
RTM::RTComponent, 179
InPorts_it
RTM::RtcBase, 108
instance_id
RTM::RTCBase, 203
RTM::RtcBase, 115
RTM::RTComponent, 179
isAliasEnable
RTM::RtcBase, 116
isLongNameEnable
RTM::RtcBase, 116
isNew
RTM::InPortAny, 139
isThreadRunning
RTM::RtcBase, 116

listComponent
RTM::RtcManager, 158
listModule
RTM::RtcManager, 158
load
RTM::RTCManager, 207
RTM::RtcManager, 158
loadCmd
RTM::RtcManager, 158
LogOutPort
RTM::RtcManager, 155

m_Alias
RTM::RtcBase, 125
m_Components
RTM::RtcManager, 159
m_CurrentState
RTM::RtcBase, 125
m_Flag
RTM::RtcBase::ThreadState, 133

m_InPorts
RTM::RtcBase, 125
m_List
RTM::RtcBase::InPorts, 131
RTM::RtcBase::OutPorts, 132
m_Logbuf
RTM::RtcManager, 159
m_LoggerOut
RTM::RtcManager, 160
m_ManagerName
RTM::RtcManager, 160
m_MedLogbuf
RTM::RtcBase, 125
RTM::RtcManager, 160
m_Mutex
RTM::RtcBase::InPorts, 131
RTM::RtcBase::OutPorts, 132
RTM::RtcBase::ThreadState, 133
m_name
RTM::RtcBase::eq_name, 130
m_NamingPolicy
RTM::RtcBase, 126
m_NextState
RTM::RtcBase, 126
m_OutPorts
RTM::RtcBase, 126
m_Parent
RTM::RtcBase, 126
m_pLogEmitter
RTM::RtcManager, 160
m_pLoggerOutPort
RTM::RtcManager, 160
m_pManager
RTM::RtcBase, 126
m_pMasterLogger
RTM::RtcManager, 160
m_pORB
RTM::RtcBase, 126
m_pPOA
RTM::RtcBase, 126
m_Profile
RTM::InPortAny, 141
RTM::InPortBase, 136
RTM::OutPortAny, 151
RTM::OutPortBase, 145
RTM::RtcBase, 126
m_StatePort
RTM::RtcBase, 126
m_Subscribers
RTM::OutPortAny, 151
RTM::OutPortBase, 145
m_ThreadState
RTM::RtcBase, 126
m_TimedState
RTM::RtcBase, 127
maker
RTM::RTCBase, 203
RTM::RtcBase, 116
RTM::RTComponent, 179
MDA, 5

name
RTM::InPortAny, 140
RTM::InPortBase, 135
RTM::NamedValue, 185
RTM::OutPortAny, 149
RTM::OutPortBase, 144
RTM::PortProfile, 188
RTM::RTComponent::NoSuchName, 183
nsec
RTM::Time, 209

OMG, 5
open
RTM::RtcBase, 116
RTM::RtcManager, 158
OpenRTM, 8
operator()
RTM::RtcBase::eq_name, 130
operator<<
RTM::OutPortAny, 149
operator>>
RTM::InPortAny, 140
OutPort, 7
OutPortAny
RTM::OutPortAny, 148
OutPortBase
RTM::OutPortBase, 143
outports
RTM::RTCBase, 203
RTM::RtcBase, 116
RTM::RTComponent, 179
OutPorts_it

RTM::RtcBase, 108

parseCommandArgs
RTM::RtcConfig, 165
parseConfigFile
RTM::RtcConfig, 165
port_type
RTM::PortProfile, 188
printUsage
RTM::RtcConfig, 165
profile
RTM::InPortAny, 140
RTM::InPortBase, 135
RTM::OutPort, 187
RTM::OutPortAny, 149
RTM::OutPortBase, 144
RTM::RTCBase, 203
RTM::RtcBase, 116
properties
RTM::PortProfile, 188
RTM::SubscriberProfile, 189
push
RTM::OutPortAny, 149
RTM::OutPortBase, 144
put
RTM::InPortAny, 140
RTM::InPortBase, 135

read
RTM::InPortAny, 140
read_pm
RTM::InPortAny, 140
RTM::InPortBase, 135
readAllInPorts
RTM::RtcBase, 116
registerComponent
RTM::RtcManager, 158, 159
registerInPort
RTM::RtcBase, 116
registerOutPort
RTM::RtcBase, 117
registerPort
RTM::RtcBase, 117
RT, 3
RTC_ABORTING
RTM::RTCBase, 203
RTM::RTComponent, 180
rtc_aborting_entry
RTM::RTCBase, 196
RTM::RtcBase, 117
RTC_ACTIVE
RTM::RTCBase, 203
RTM::RTComponent, 180
rtc_active_do
RTM::RTCBase, 196
RTM::RtcBase, 117
rtc_active_entry
RTM::RTCBase, 196
RTM::RtcBase, 117
rtc_active_exit
RTM::RTCBase, 196
RTM::RtcBase, 118
rtc_add_component
RTM::RTCBase, 196
RTM::RtcBase, 118
rtc_attach_inport
RTM::RTCBase, 197
RTM::RtcBase, 118
rtc_attach_inport_by_name
RTM::RTCBase, 197
RTM::RtcBase, 118
rtc_attach_outport
RTM::RTCBase, 197
RTM::RtcBase, 118
rtc_attach_outport_by_name
RTM::RTCBase, 197
RTM::RtcBase, 118
RTC_BORN
RTM::RTCBase, 203
RTM::RTComponent, 180
rtc_components
RTM::RTCBase, 197
RTM::RtcBase, 119
rtc_delete_component
RTM::RTCBase, 197
RTM::RtcBase, 119
rtc_detatch_inport
RTM::RTCBase, 198
RTM::RtcBase, 119
rtc_detatch_inport_by_name
RTM::RTCBase, 198
RTM::RtcBase, 119
rtc_detatch_outport
RTM::RTCBase, 198

RTM::RtcBase, 119
rtc_detatch_outport_by_name
RTM::RTCBase, 198
RTM::RtcBase, 119
RTC_ERROR
RTM::RTCBase, 204
RTM::RTComponent, 180
rtc_error_do
RTM::RTCBase, 198
RTM::RtcBase, 120
rtc_error_entry
RTM::RTCBase, 198
RTM::RtcBase, 120
rtc_error_exit
RTM::RTCBase, 199
RTM::RtcBase, 120
rtc_exit
RTM::RTCBase, 199
RTM::RtcBase, 120
RTM::RTComponent, 178
RTC_EXITING
RTM::RTCBase, 204
RTM::RTComponent, 180
rtc_exiting_entry
RTM::RTCBase, 199
RTM::RtcBase, 120
rtc_fatal_do
RTM::RTCBase, 199
RTM::RtcBase, 120
rtc_fatal_entry
RTM::RTCBase, 199
RTM::RtcBase, 121
RTC_FATAL_ERROR
RTM::RTCBase, 204
RTM::RTComponent, 180
rtc_fatal_exit
RTM::RTCBase, 199
RTM::RtcBase, 121
rtc_get_component
RTM::RTCBase, 199
RTM::RtcBase, 121
rtc_init_entry
RTM::RTCBase, 200
RTM::RtcBase, 121
RTC_INITIALIZING
RTM::RTCBase, 204
RTM::RTComponent, 180
rtc_kill
RTM::RTCBase, 200
RTM::RtcBase, 121
RTM::RTComponent, 178
RTC_READY
RTM::RTCBase, 204
RTM::RTComponent, 180
rtc_ready_do
RTM::RTCBase, 200
RTM::RtcBase, 121
rtc_ready_entry
RTM::RTCBase, 200
RTM::RtcBase, 121
rtc_ready_exit
RTM::RTCBase, 200
RTM::RtcBase, 122
rtc_replace_component
RTM::RTCBase, 200
RTM::RtcBase, 122
rtc_replace_component_by_name
RTM::RTCBase, 200
RTM::RtcBase, 122
rtc_reset
RTM::RTCBase, 201
RTM::RtcBase, 122
RTM::RTComponent, 178
rtc_set_parent
RTM::RTCBase, 201
RTM::RtcBase, 122
rtc_start
RTM::RTCBase, 201
RTM::RtcBase, 122
RTM::RTComponent, 178
rtc_start_thread
RTM::RTCBase, 201
RTM::RtcBase, 123
RTC_STARTING
RTM::RTCBase, 204
RTM::RTComponent, 180
rtc_starting_entry
RTM::RTCBase, 201
RTM::RtcBase, 123
rtc_state
RTM::RTCBase, 204
RTM::RtcBase, 123
RTM::RTComponent, 180
rtc_stop

- RTM::RTCBase, 201
- RTM::RtcBase, 123
- RTM::RTComponent, 178

rtc_stop_thread
- RTM::RTCBase, 202
- RTM::RtcBase, 123

RTC_STOPPING
- RTM::RTCBase, 204
- RTM::RTComponent, 181

rtc_stopping_entry
- RTM::RTCBase, 202
- RTM::RtcBase, 123

rtc_suspend_thread
- RTM::RtcBase, 124

RTC_UNKNOWN
- RTM::RTCBase, 205
- RTM::RTComponent, 181

rtc_worker
- RTM::RTCBase, 202
- RTM::RtcBase, 124
- RTM::RTComponent, 179

RtcBase
- RTM::RtcBase, 108, 109

RtcComponentInit
- RTM::RtcManager, 155

RtcConfig
- RTM::RtcConfig, 163

RtcManager
- RTM::RtcManager, 155

RtcNaming
- RTM::RtcNaming, 168

RTComponent, 7

rtcout
- RTM::RtcBase, 127
- RTM::RtcManager, 160

RTM::InPort::Disconnected, 184

RTM::InPortAny, 137

RTM::InPortAny
- ~InPortAny, 139
- getNewDataLen, 139
- getNewList, 139
- getNewListReverse, 139
- initBuffer, 139
- InPortAny, 138
- isNew, 139
- m_Profile, 141
- name, 140
- operator>>, 140
- profile, 140
- put, 140
- read, 140
- read_pm, 140

RTM::InPortBase, 134

RTM::InPortBase
- ~InPortBase, 135
- InPortBase, 135
- m_Profile, 136
- name, 135
- profile, 135
- put, 135
- read_pm, 135

RTM::NamedValue, 185

RTM::NamedValue
- flag, 185
- name, 185
- value, 185

RTM::OutPort, 186

RTM::OutPort
- get, 186
- inports, 187
- profile, 187
- subscribe, 186
- unsubscribe, 186

RTM::OutPortAny, 146

RTM::OutPortAny
- ~OutPortAny, 148
- disconnect_all, 148
- get, 148
- initBuffer, 149
- inports, 149
- m_Profile, 151
- m_Subscribers, 151
- name, 149
- operator<<, 149
- OutPortAny, 148
- profile, 149
- push, 149
- subscribe, 149
- unsubscribe, 150
- unsubscribeNoLocked, 150
- updateall, 150
- write, 150
- write_pm, 150

RTM::OutPortBase, 142

RTM::OutPortBase
~OutPortBase, 143
disconnect_all, 143
get, 143
inports, 144
m_Profile, 145
m_Subscribers, 145
name, 144
OutPortBase, 143
profile, 144
push, 144
subscribe, 144
unsubscribe, 144
unsubscribeNoLocked, 144
updateall, 145
write_pm, 145
RTM::PortProfile, 188
RTM::PortProfile
name, 188
port_type, 188
properties, 188
RTM::RTCBase, 191
category, 202
ComponentState, 195
description, 202
get_inport, 196
get_outport, 196
implementation_id, 202
inports, 203
instance_id, 203
maker, 203
outports, 203
profile, 203
RTC_ABORTING, 203
rtc_aborting_entry, 196
RTC_ACTIVE, 203
rtc_active_do, 196
rtc_active_entry, 196
rtc_active_exit, 196
rtc_add_component, 196
rtc_attach_inport, 197
rtc_attach_inport_by_name, 197
rtc_attach_outport, 197
rtc_attach_outport_by_name, 197
RTC_BORN, 203
rtc_components, 197
rtc_delete_component, 197
rtc_detatch_inport, 198
rtc_detatch_inport_by_name, 198
rtc_detatch_outport, 198
rtc_detatch_outport_by_name, 198
RTC_ERROR, 204
rtc_error_do, 198
rtc_error_entry, 198
rtc_error_exit, 199
rtc_exit, 199
RTC_EXITING, 204
rtc_exiting_entry, 199
rtc_fatal_do, 199
rtc_fatal_entry, 199
RTC_FATAL_ERROR, 204
rtc_fatal_exit, 199
rtc_get_component, 199
rtc_init_entry, 200
RTC_INITIALIZING, 204
rtc_kill, 200
RTC_READY, 204
rtc_ready_do, 200
rtc_ready_entry, 200
rtc_ready_exit, 200
rtc_replace_component, 200
rtc_replace_component_by_name, 200
rtc_reset, 201
rtc_set_parent, 201
rtc_start, 201
rtc_start_thread, 201
RTC_STARTING, 204
rtc_starting_entry, 201
rtc_state, 204
rtc_stop, 201
rtc_stop_thread, 202
RTC_STOPPING, 204
rtc_stopping_entry, 202
RTC_UNKNOWN, 205
rtc_worker, 202
version, 205
RTM::RtcBase, 99
RUNNING, 108
STOP, 108
SUSPEND, 108
UNKNOWN, 108
RTM::RtcBase
_check_error, 109
_do_func, 125

_entry_func, 125
_exit_func, 125
_nop, 109
_rtc_aborting, 109
_rtc_active, 109
_rtc_active_entry, 110
_rtc_active_exit, 110
_rtc_error, 110
_rtc_error_entry, 110
_rtc_error_exit, 110
_rtc_exiting, 110
_rtc_fatal, 110
_rtc_fatal_entry, 110
_rtc_fatal_exit, 110
_rtc_initializing, 110
_rtc_ready, 111
_rtc_ready_entry, 111
_rtc_ready_exit, 111
_rtc_starting, 111
_rtc_stopping, 111
~RtcBase, 109
appendAlias, 111
category, 112
close, 112
deleteInPort, 112
deleteInPortByName, 112
deleteOutPort, 112
deleteOutPortByName, 112
deletePort, 113
description, 113
finalize, 113
finalizeInPorts, 113
finalizeOutPorts, 113
forceExit, 114
get_inport, 114
get_outport, 114
getAliases, 114
getModuleProfile, 114
getNamingPolicy, 114
getState, 115
implementation_id, 115
init_orb, 115
init_state_func_table, 115
initModuleProfile, 115
inports, 115
InPorts_it, 108
instance_id, 115
isAliasEnable, 116
isLongNameEnable, 116
isThreadRunning, 116
m_Alias, 125
m_CurrentState, 125
m_InPorts, 125
m_MedLogbuf, 125
m_NamingPolicy, 126
m_NextState, 126
m_OutPorts, 126
m_Parent, 126
m_pManager, 126
m_pORB, 126
m_pPOA, 126
m_Profile, 126
m_StatePort, 126
m_ThreadState, 126
m_TimedState, 127
maker, 116
open, 116
outports, 116
OutPorts_it, 108
profile, 116
readAllInPorts, 116
registerInPort, 116
registerOutPort, 117
registerPort, 117
rtc_aborting_entry, 117
rtc_active_do, 117
rtc_active_entry, 117
rtc_active_exit, 118
rtc_add_component, 118
rtc_attach_inport, 118
rtc_attach_inport_by_name, 118
rtc_attach_outport, 118
rtc_attach_outport_by_name, 118
rtc_components, 119
rtc_delete_component, 119
rtc_detatch_inport, 119
rtc_detatch_inport_by_name, 119
rtc_detatch_outport, 119
rtc_detatch_outport_by_name, 119
rtc_error_do, 120
rtc_error_entry, 120
rtc_error_exit, 120
rtc_exit, 120
rtc_exiting_entry, 120

rtc_fatal_do, 120
rtc_fatal_entry, 121
rtc_fatal_exit, 121
rtc_get_component, 121
rtc_init_entry, 121
rtc_kill, 121
rtc_ready_do, 121
rtc_ready_entry, 121
rtc_ready_exit, 122
rtc_replace_component, 122
rtc_replace_component_by_name, 122
rtc_reset, 122
rtc_set_parent, 122
rtc_start, 122
rtc_start_thread, 123
rtc_starting_entry, 123
rtc_state, 123
rtc_stop, 123
rtc_stop_thread, 123
rtc_stopping_entry, 123
rtc_suspend_thread, 124
rtc_worker, 124
RtcBase, 108, 109
rtcout, 127
setComponentName, 124
setNamingPolicy, 124
StateFunc, 108
svc, 124
ThreadStates, 108
version, 124
writeAllOutPorts, 125
RTM::RtcBase::ComponentStateMtx, 128
RTM::RtcBase::ComponentStateMtx
_mutex, 128
_state, 128
ComponentStateMtx, 128
RTM::RtcBase::eq_name, 130
RTM::RtcBase::eq_name
eq_name, 130
m_name, 130
operator(), 130
RTM::RtcBase::InPorts, 131
RTM::RtcBase::InPorts
m_List, 131
m_Mutex, 131
RTM::RtcBase::OutPorts, 132
RTM::RtcBase::OutPorts
m_List, 132
m_Mutex, 132
RTM::RtcBase::ThreadState, 133
RTM::RtcBase::ThreadState
m_Flag, 133
m_Mutex, 133
ThreadState, 133
RTM::RtcConfig, 161
RTM::RtcConfig
~RtcConfig, 163
argsToArgv, 163
collectSysInfo, 163
fileExist, 163
findConfigFile, 163
getArch, 163
getBinName, 163
getComponentLoadPath, 164
getErrorLogFileName, 164
getHostname, 164
getLogFileName, 164
getLogLevel, 164
getLogLock, 164
getLogTimeFormat, 164
getNameServer, 164
getOrbInitArgc, 164
getOrbInitArgv, 164
getOSname, 164
getOSrelease, 164
getOSversion, 165
getPid, 165
initConfig, 165
parseCommandArgs, 165
parseConfigFile, 165
printUsage, 165
RtcConfig, 163
split, 165
RTM::RTCManager, 206
command, 206
component_list, 206
create_component, 207
delete_component, 207
factory_list, 207
load, 207
unload, 207
RTM::RtcManager, 152
RTM::RtcManager
~RtcManager, 155

activateManager, 155
bindInOut, 155
bindInOutByName, 155
cleanupComponent, 156
close, 156
command, 156
commandListCmd, 156
component_list, 156
create_component, 156
createCommand, 156
createComponent, 157
createComponentCmd, 157
delete_component, 157
factory_list, 157
findComponents, 157
getConfig, 157
getLogbuf, 158
getORB, 158
getPOA, 158
initManager, 158
initModuleProc, 158
listComponent, 158
listModule, 158
load, 158
loadCmd, 158
LogOutPort, 155
m_Components, 159
m_Logbuf, 159
m_LoggerOut, 160
m_ManagerName, 160
m_MedLogbuf, 160
m_pLogEmitter, 160
m_pLoggerOutPort, 160
m_pMasterLogger, 160
open, 158
registerComponent, 158, 159
RtcComponentInit, 155
RtcManager, 155
rtcout, 160
runManager, 159
runManagerNoBlocking, 159
shutdown, 159
svc, 159
unload, 159
unloadCmd, 159

RTM::RtcNaming, 166

RTM::RtcNaming
~RtcNaming, 168
bindComponent, 168
bindManager, 169
bindObject, 169
bindObjectByFullPath, 169
bindObjectRecursive, 169
createCategoryContext, 170
createContext, 170
createHostContext, 170
createManagerContext, 170
createModuleContext, 171
destroyCategoryContext, 171
destroyHostContext, 171
destroyManagerContext, 171
destroyModuleContext, 171
destroyRecursive, 171
findCategoryContext, 172
findComponents, 172
findContextRecursive, 172
findHostContext, 172
findManager, 173
findManagerContext, 173
findModuleContext, 173
findObjectsRecursive, 173
initNaming, 173
RtcNaming, 168
unbindLocalComponent, 174
unbindObject, 174
unbindObjectByFullPath, 174

RTM::RTComponent, 175
category, 179
ComponentState, 177
description, 179
get_inport, 177
get_outport, 177
implementation_id, 179
inports, 179
instance_id, 179
maker, 179
outports, 179
RTC_ABORTING, 180
RTC_ACTIVE, 180
RTC_BORN, 180
RTC_ERROR, 180
rtc_exit, 178
RTC_EXITING, 180
RTC_FATAL_ERROR, 180

RTC_INITIALIZING, 180
rtc_kill, 178
RTC_READY, 180
rtc_reset, 178
rtc_start, 178
RTC_STARTING, 180
rtc_state, 180
rtc_stop, 178
RTC_STOPPING, 181
RTC_UNKNOWN, 181
rtc_worker, 179
version, 181
RTM::RTComponent::IllegalTransition, 182
RTM::RTComponent::NoSuchName, 183
RTM::RTComponent::NoSuchName
name, 183
RTM::SubscriberProfile, 189
RTM::SubscriberProfile
event_base, 189
properties, 189
subscription_type, 189
RTM::Time, 209
nsec, 209
sec, 209
RTM::TimedBoolean, 210
RTM::TimedBoolean
data, 210
tm, 210
RTM::TimedBooleanSeq, 211
RTM::TimedBooleanSeq
data, 211
tm, 211
RTM::TimedChar, 212
RTM::TimedChar
data, 212
tm, 212
RTM::TimedCharSeq, 213
RTM::TimedCharSeq
data, 213
tm, 213
RTM::TimedDouble, 214
RTM::TimedDouble
data, 214
tm, 214
RTM::TimedDoubleSeq, 215
RTM::TimedDoubleSeq
data, 215
tm, 215
RTM::TimedFloat, 216
RTM::TimedFloat
data, 216
tm, 216
RTM::TimedFloatSeq, 217
RTM::TimedFloatSeq
data, 217
tm, 217
RTM::TimedLong, 218
RTM::TimedLong
data, 218
tm, 218
RTM::TimedLongSeq, 219
RTM::TimedLongSeq
data, 219
tm, 219
RTM::TimedOctet, 220
RTM::TimedOctet
data, 220
tm, 220
RTM::TimedOctetSeq, 221
RTM::TimedOctetSeq
data, 221
tm, 221
RTM::TimedShort, 222
RTM::TimedShort
data, 222
tm, 222
RTM::TimedShortSeq, 223
RTM::TimedShortSeq
data, 223
tm, 223
RTM::TimedState, 224
RTM::TimedState
data, 224
tm, 224
RTM::TimedString, 225
RTM::TimedString
data, 225
tm, 225
RTM::TimedStringSeq, 226
RTM::TimedStringSeq
data, 226
tm, 226
RTM::TimedULong, 227

RTM::TimedULong
data, 227
tm, 227
RTM::TimedULongSeq, 228
RTM::TimedULongSeq
data, 228
tm, 228
RTM::TimedUShort, 229
RTM::TimedUShort
data, 229
tm, 229
RTM::TimedUShortSeq, 230
RTM::TimedUShortSeq
data, 230
tm, 230
RT コンポーネント, 3, 6
RT コンポーネントフレームワーク, 6
RT ミドルウエア, 3, 6
runManager
RTM::RtcManager, 159
runManagerNoBlocking
RTM::RtcManager, 159
RUNNING
RTM::RtcBase, 108

sec
RTM::Time, 209
setComponentName
RTM::RtcBase, 124
setNamingPolicy
RTM::RtcBase, 124
shutdown
RTM::RtcManager, 159
split
RTM::RtcConfig, 165
StateFunc
RTM::RtcBase, 108
STOP
RTM::RtcBase, 108
subscribe
RTM::OutPort, 186
RTM::OutPortAny, 149
RTM::OutPortBase, 144
subscription_type
RTM::SubscriberProfile, 189
SUSPEND
RTM::RtcBase, 108
svc
RTM::RtcBase, 124
RTM::RtcManager, 159

ThreadState
RTM::RtcBase::ThreadState, 133
ThreadStates
RTM::RtcBase, 108
tm
RTM::TimedBoolean, 210
RTM::TimedBooleanSeq, 211
RTM::TimedChar, 212
RTM::TimedCharSeq, 213
RTM::TimedDouble, 214
RTM::TimedDoubleSeq, 215
RTM::TimedFloat, 216
RTM::TimedFloatSeq, 217
RTM::TimedLong, 218
RTM::TimedLongSeq, 219
RTM::TimedOctet, 220
RTM::TimedOctetSeq, 221
RTM::TimedShort, 222
RTM::TimedShortSeq, 223
RTM::TimedState, 224
RTM::TimedString, 225
RTM::TimedStringSeq, 226
RTM::TimedULong, 227
RTM::TimedULongSeq, 228
RTM::TimedUShort, 229
RTM::TimedUShortSeq, 230

unbindLocalComponent
RTM::RtcNaming, 174
unbindObject
RTM::RtcNaming, 174
unbindObjectByFullPath
RTM::RtcNaming, 174
UNKNOWN
RTM::RtcBase, 108
unload
RTM::RTCManager, 207
RTM::RtcManager, 159
unloadCmd
RTM::RtcManager, 159
unsubscribe
RTM::OutPort, 186
RTM::OutPortAny, 150

RTM::OutPortBase, 144
unsubscribeNoLocked
RTM::OutPortAny, 150
RTM::OutPortBase, 144
updateall
RTM::OutPortAny, 150
RTM::OutPortBase, 145

value
RTM::NamedValue, 185
version
RTM::RTCBase, 205
RTM::RtcBase, 124
RTM::RTComponent, 181

write
RTM::OutPortAny, 150
write_pm
RTM::OutPortAny, 150
RTM::OutPortBase, 145
writeAllOutPorts
RTM::RtcBase, 125

フレームワーク, 5