PFCore(RTミドルウェア)トレーニング 中級編

10:00-11:00	第1部：RTコンポーネントプログラミングの概要
	担当：安藤慶昭（産業技術総合研究所）
	概要:RTコンポーネントの作成方法，設計時の注意点などの概要について解説します。
11:00-12:00	第2部：RTミドルウェア(PFcore)開発支援ツールとRTコンポーネントの作成方法
	担当：坂本 武志（株式会社 グローバルアシスト）
	概要:RTコンポーネントを開発するために必要なツールのインストール方法，標準ツールRTCBulderを使用して、RTコンポーネントを開発する方法の概略を説明します．
12:00-13:00	休憩
13:00-17:00	第3部：RTコンポーネント開発実習
	担当：安藤慶昭（産業技術総合研究所）
	概要:OpenRTM-aistでのコンポーネントの作成方法を実際に体験して頂きます．画像処理システムを対象にRTCBuilderを使用したRTコンポーネントの設計，実装を行います．

第3部　RTコンポーネント開発実習

（独）産業技術総合研究所
知能システム研究部門
安藤慶昭

RTとは?

- RT = Robot Technology　cf. IT
  - ≠Real-time
  - 単体のロボットだけでなく、さまざまなロボット技術に基づく機能要素をも含む (センサ、アクチュエータ, 制御スキーム、アルゴリズム、etc.…)

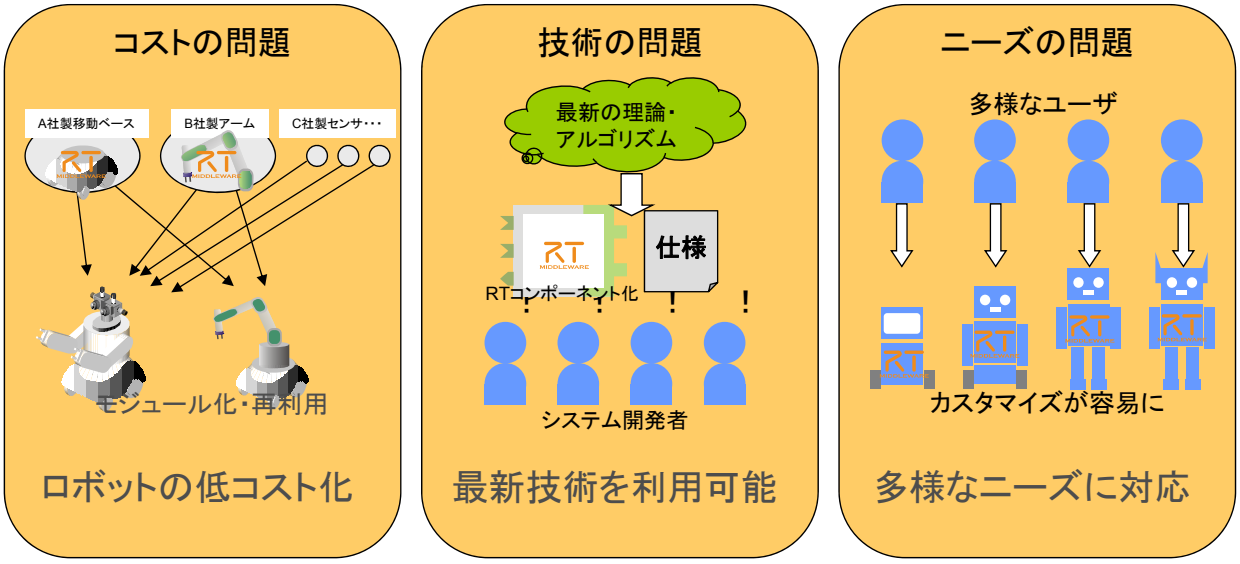
産総研版RTミドルウエア

OpenRTM-aist

- RT-Middleware（RTM)
  - RT要素のインテグレーションのためのミドルウエア
- RT-Component（RTC)
  - RT-Middlewareにおけるソフトウエアの基本単位

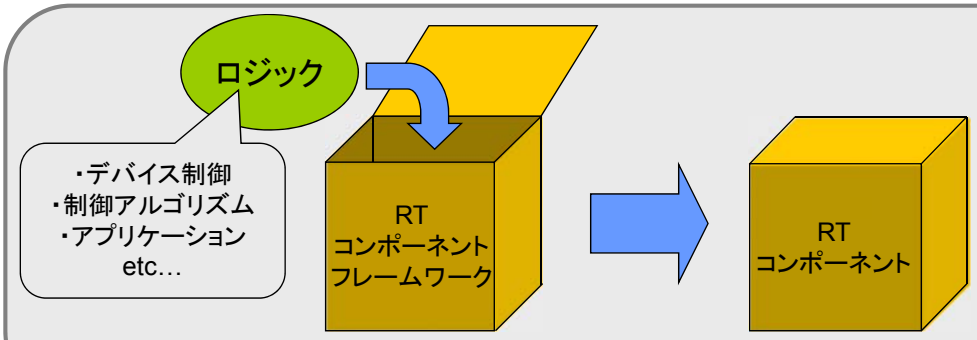
RTミドルウエアの目的

モジュール化による問題解決

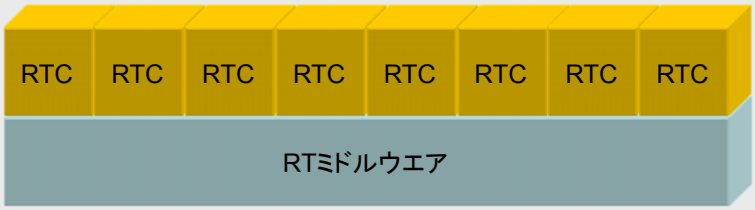


ロボットシステムインテグレーションによるイノベーション

RTミドルウエアとRTコンポーネント



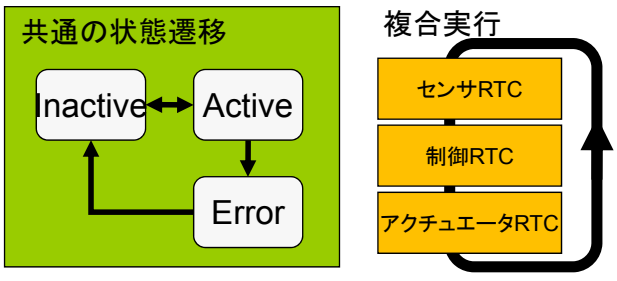
ロジックを箱（フレームワーク）に入れたもの＝RTコンポーネント（RTC）



RTCの実行環境（OSのようなもの）＝RTミドルウエア（RTM）
※RTCはネットワーク上に分散可能

RTコンポーネントの主な機能

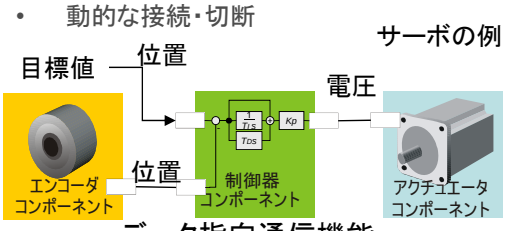
アクティビティ・実行コンテキスト



ライフサイクルの管理・コアロジックの実行

データポート

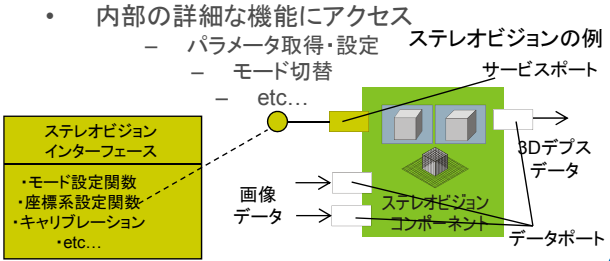
- データ指向ポート
- 連続的なデータの送受信
- 動的な接続・切断



データ指向通信機能

サービスポート

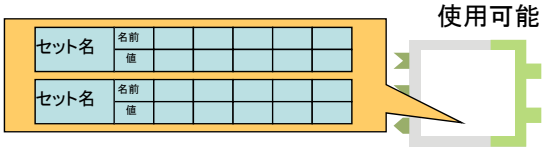
- 定義可能なインターフェースを持つ
- 内部の詳細な機能にアクセス
  - パラメータ取得・設定
  - モード切替
  - etc…



サービス指向相互作用機能

コンフィギュレーション

- パラメータを保持する仕組み
- いくつかのセットを保持可能
- 実行時に動的に変更可能



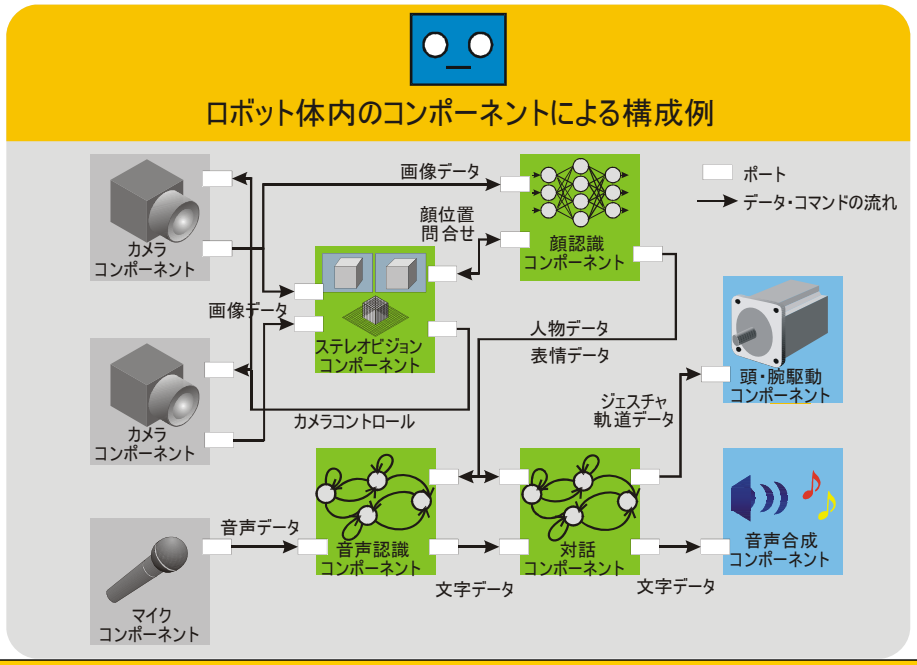
OpenRTM-aist

- コンポーネントフレームワーク + ミドルウエアライブラリ
- コンポーネントインターフェース:
  - OMG Robotic Technology Component Specification ver1.0 準拠
- OS
  - 公式:FreeBSD, Linux (Fedora, Debian, Ubuntu, Vine, Scientific), Windows
  - 非公式:Mac OS X, uITRON, T-Kernel, VxWorks
- 言語:
  - C++ (1.1.0), Python (1.0.0), Java (1.0.0)
  - .NET (implemented by SEC)
- CPU アーキテクチャ (動作実績):
  - i386, ARM9, PPC, SH4
  - PIC, dsPIC, H8 (RTC-Lite)
- ツール (Eclipse プラグイン)
  - テンプレートソースジェネレータ: rtc-template、RTCBuilder
  - システムインテグレーションツール: RTSystemEditor
  - その他
    - Pattern weaver for RT-Middleware (株式会社テクノロジックアートより発売中)

OpenRTMの利点

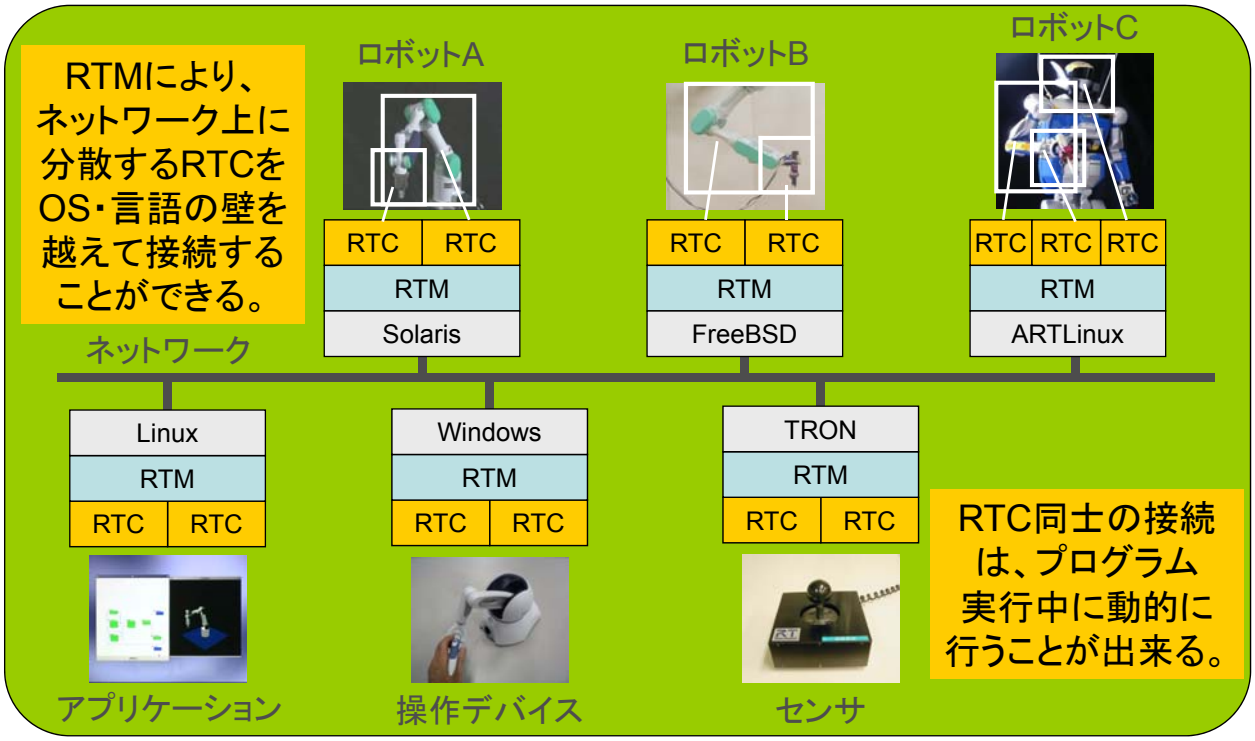
- 共通コンポーネントフレームワークを提供
  - OMG標準
  - コールバックベースの枠組み、共通状態マシン、複合化に対応
  - 大部分のコード生成を自動化
- 多言語対応
  - C++, Java, Python, .NET (by SEC)
- 多様なOSへのネイティブ対応
  - FreeBSD, Linux, Mac OS X, Windows
  - 試験的:TOPPERS, T-Kernel, VxWorks
- ツールの提供
  - Eclipseベースのツール群 (RTCB, RTSE)
  - コマンドラインツール群 (rtchell)
- デュアルライセンス (LGPL・EPLと個別ライセンス)
  - RTCにはラインセンスが及ばない(RTCのバイナリ供給が可能に)
  - 商用化、事業化、組込み用途には個別ライセンスで対応

RTCの分割と連携

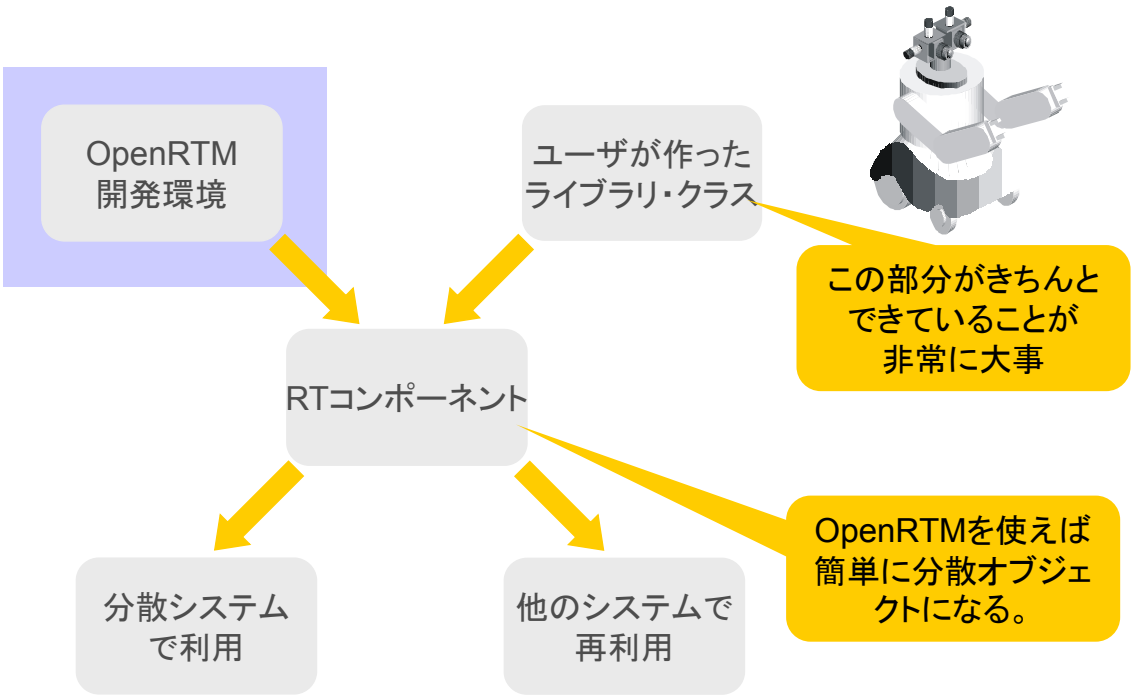


（モジュール）情報の隠蔽と公開のルールが重要

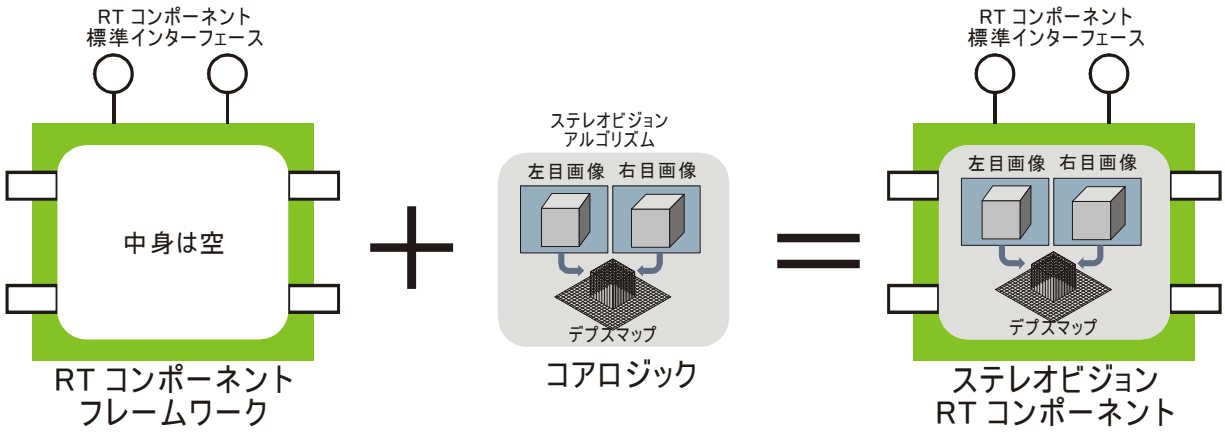
RTミドルウエアによる分散システム



OpenRTMを使った開発の流れ

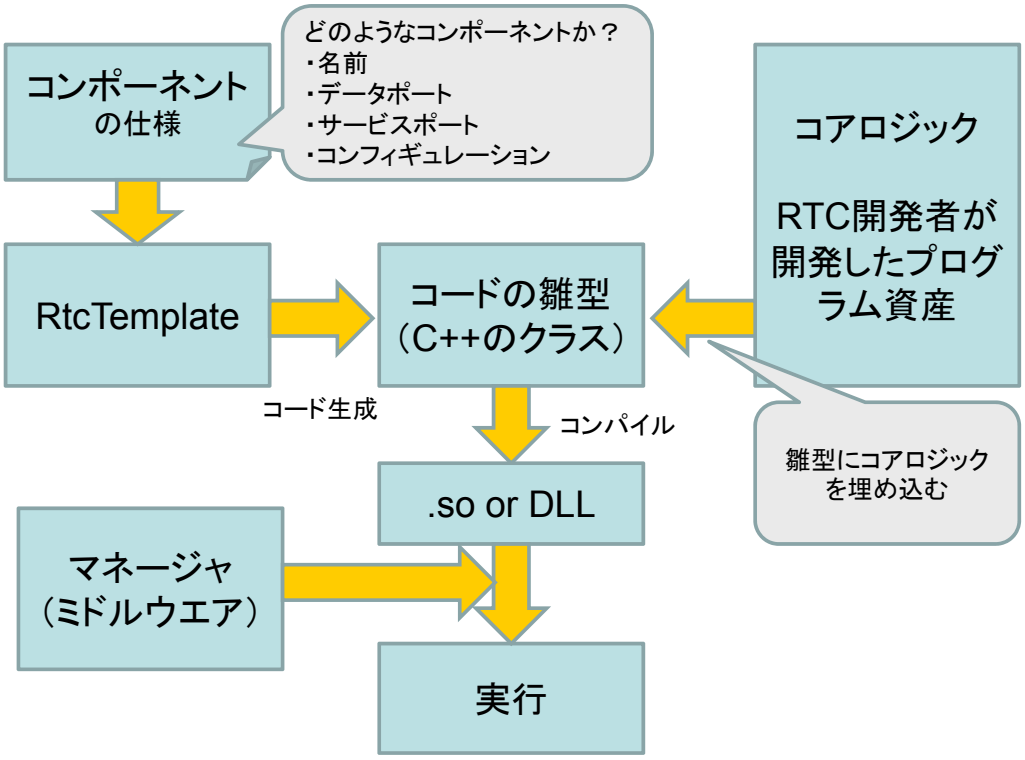


フレームワークとコアロジック



RTCフレームワーク＋コアロジック＝RTコンポーネント

OpenRTMを使った開発の流れ



コンポーネントの作成
（Windowsの場合）

コンポーネントの
仕様の入力

VCのプロジェクト
ファイルの生成

実装および
VCでコンパイル
実行ファイルの生成

テンプレートコード
の生成

コード例

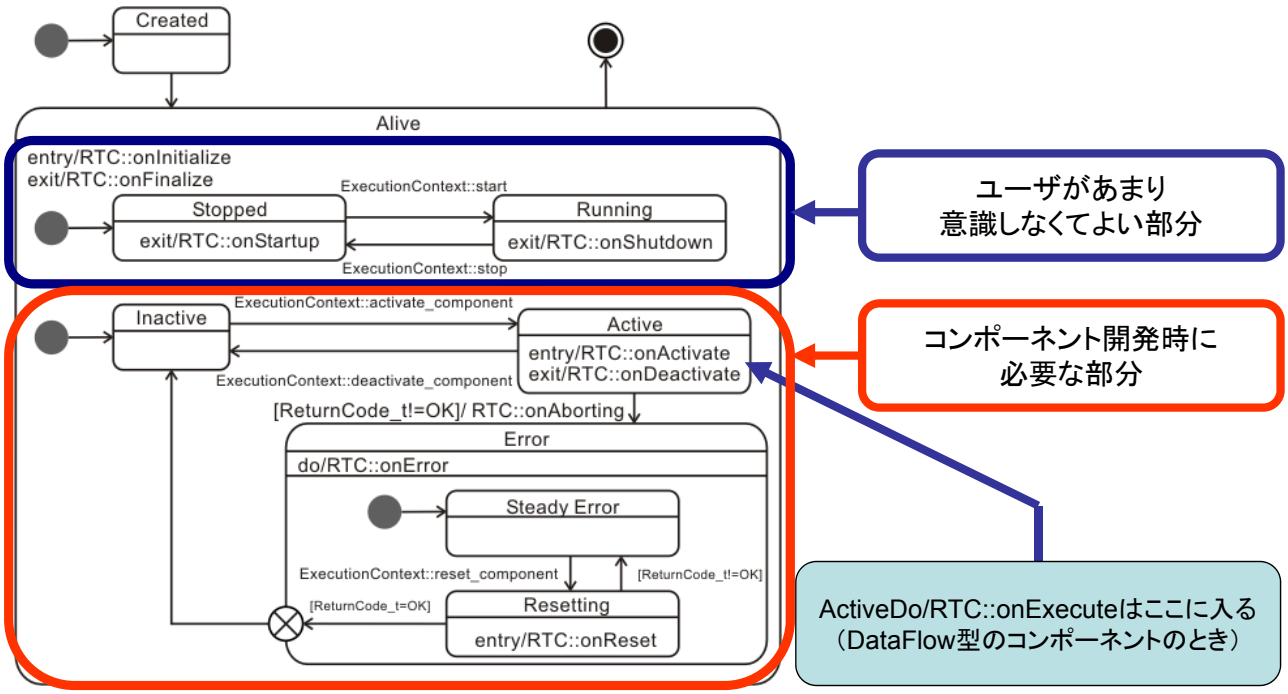
- 生成されたクラスのメンバー関数に必要な処理を記述
- 主要な関数
  - onExecute (周期実行)
- 処理
  - InPortから読む
  - OutPortへ書く
  - サービスを呼ぶ
  - コンフィギュレーションを読む

```
class MyComponent
  : public DataflowComponentBase
{
public:
  // 初期化時に実行したい処理
  virtual ReturnCode_t onInitialize()
  {
    if (mylogic.init())
      return RTC::RTC_OK;
    return RTC::RTC_ERROR;
  }

  // 周期的に実行したい処理
  virtual ReturnCode_t onExecute(RTC::UniqueId ec_id)
  {
    if (mylogic.do_someting())
      return RTC::RTC_OK;
    RTC::RTC_ERROR;
  }

private:
  MyLogic mylogic;
  // ポート等の宣言
  //   :
};
```


コンポーネント内の状態遷移



コールバック関数

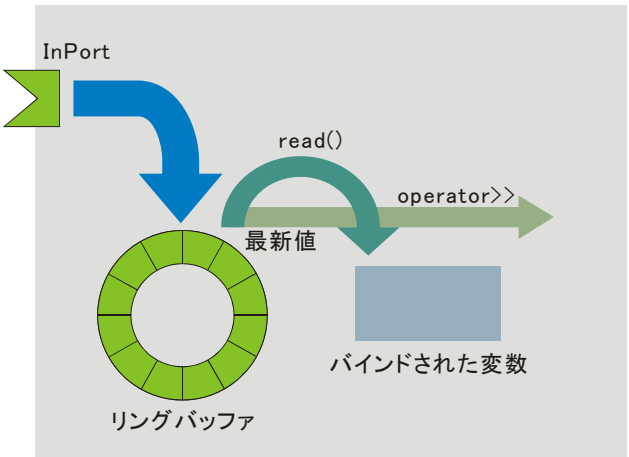
RTCの作成＝コールバック関数に処理を埋め込む

コールバック関数	処理
onInitialize	初期化処理
onActivated	アクティブ化されるとき1度だけ呼ばれる
onExecute	アクティブ状態時に周期的に呼ばれる
onDeactivated	非アクティブ化されるとき1度だけ呼ばれる
onAborting	ERROR状態に入る前に1度だけ呼ばれる
onReset	resetされる時に1度だけ呼ばれる
onError	ERROR状態のときに周期的に呼ばれる
onFinalize	終了時に1度だけ呼ばれる
onStateUpdate	onExecuteの後毎回呼ばれる
onRateChanged	ExecutionContextのrateが変更されたとき1度だけ呼ばれる
onStartup	ExecutionContextが実行を開始するとき1度だけ呼ばれる
onShutdown	ExecutionContextが実行を停止するとき1度だけ呼ばれる

とりあえずは
この5つの関数
を押さえて
おけばOK

InPort

- InPortのテンプレート第2引数：バッファ
  - ユーザ定義のバッファが利用可能
- InPortのメソッド
  - read(): InPort バッファからバインドされた変数へ最新値を読み込む
  - >> : ある変数へ最新値を読み込む



基本的にOutPortと対になる

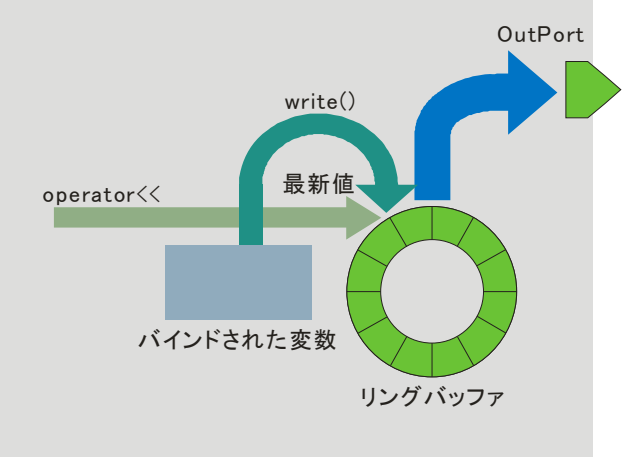
データポートの型を
同じにする必要あり

例

Sensor Data → Robot Component

OutPort

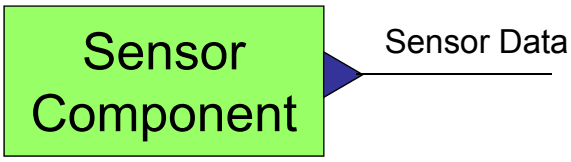
- OutPortのテンプレート第2引数：バッファ
  - ユーザ定義のバッファが利用可能
- OutPortのメソッド
  - write(): OutPort バッファへバインドされた変数の最新値として書き込む
  - >> : ある変数の内容を最新値としてリングバッファに書き込む



基本的にInPortと対になる

データポートの型を
同じにする必要あり

例



データ変数

```
struct TimedShort
 {
      Time tm;
      short data;
 };
```

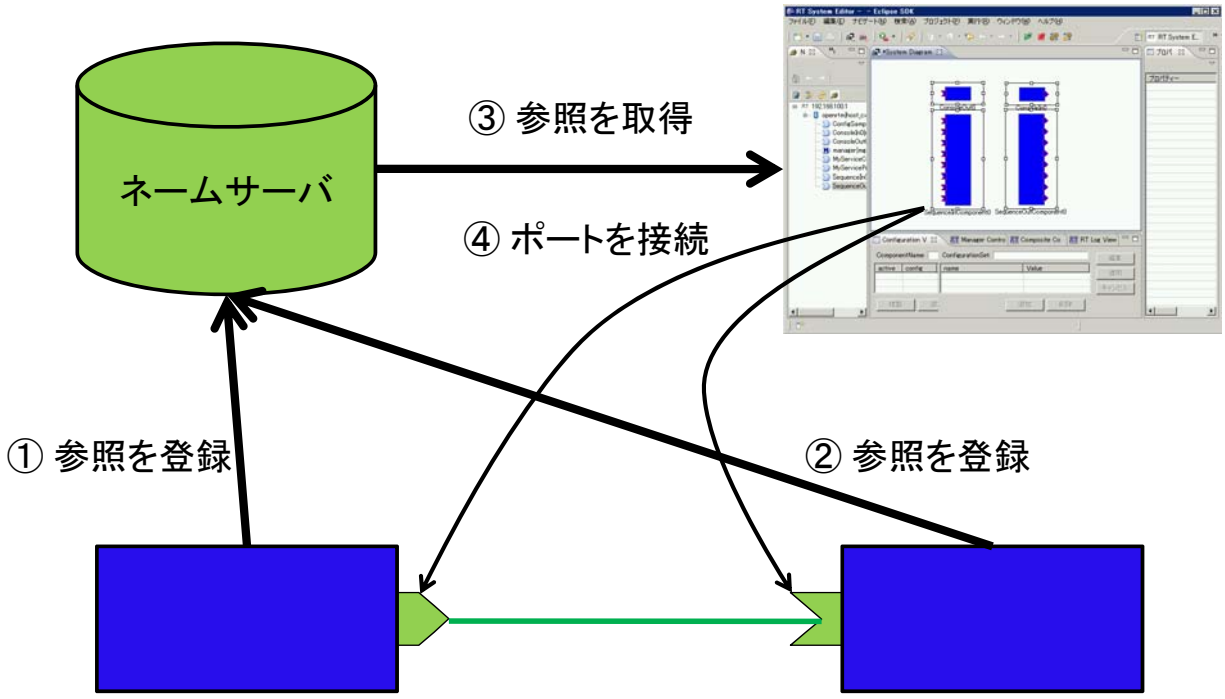
```
struct TimedShortSeq
 {
      Time tm;
      sequence<short> data;
 };
```

- 基本型
  - tm：時刻
  - data: データそのもの

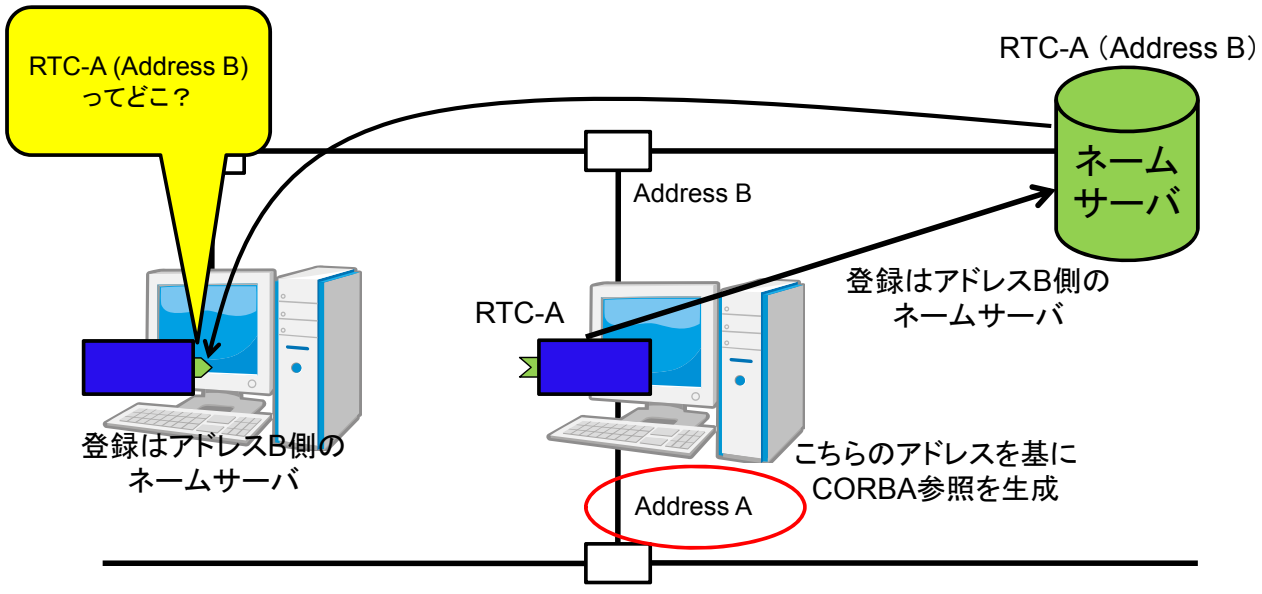
- シーケンス型
  - data[i]: 添え字によるアクセス
  - data.length(i): 長さiを確保
  - data.length(): 長さを取得

- データを入れるときにはあらかじめ長さをセットしなければならない。
- CORBAのシーケンス型そのもの
- 今後変更される可能性あり

動作シーケンス



ネットワークインターフェースが2つある場合の注意



Rtc.confについて

RT Component起動時の登録先NamingServiceや、登録情報などについて記述するファイル

記述例：

corba.nameservers: localhost:9876

naming.formats: SimpleComponent/%n.rtc

（詳細な記述方法は etc/rtc.conf.sample を参照）

以下のようにすると、コンポーネント起動時に読み込まれる

./ConsoleInComp –f rtc.conf

ネーミングサービス設定

corba.nameservers	host_name:port_numberで指定、デフォルトポートは2809(omniORBのデフォルト)、複数指定可能
naming.formats	%h.host_cxt/%n.rtc →host.host_cxt/MyComp.rtc 複数指定可能、0.2.0互換にしたければ、 %h.host_cxt/%M.mgr_cxt/%c.cat_cxt/%m.mod_cxt/%n.rtc
naming.update.enable	“YES” or “NO”: ネーミングサービスへの登録の自動アップデート。コンポーネント起動後にネームサービスが起動したときに、再度名前を登録する。
naming.update.interval	アップデートの周期[s]。デフォルトは10秒。
timer.enable	“YES” or “NO”: マネージャタイマ有効・無効。naming.updateを使用するには有効でなければならない
timer.tick	タイマの分解能[s]。デフォルトは100ms。

必須の項目　　必須でないOption設定

ログ設定

logger.enable	“YES” or “NO”: ログ出力を有効・無効
logger.file_name	ログファイル名。 %h:ホスト名、%M:マネージャ名,%p:プロセスID 使用可
logger.date_format	日付フォーマット。strftime(3)の表記法に準拠。 デフォルト:%b %d %H:%M:%S → Apr 24 01:02:04
logger.log_level	ログレベル: SILENT, ERROR, WARN, NORMAL, INFO, DEBUG, TRACE, VERBOSE, PARANOID SILENT:何も出力しない PARANOID:全て出力する ※以前はRTC内で使えましたが、現在はまだ使えません。

必須の項目　　必須でないOption設定

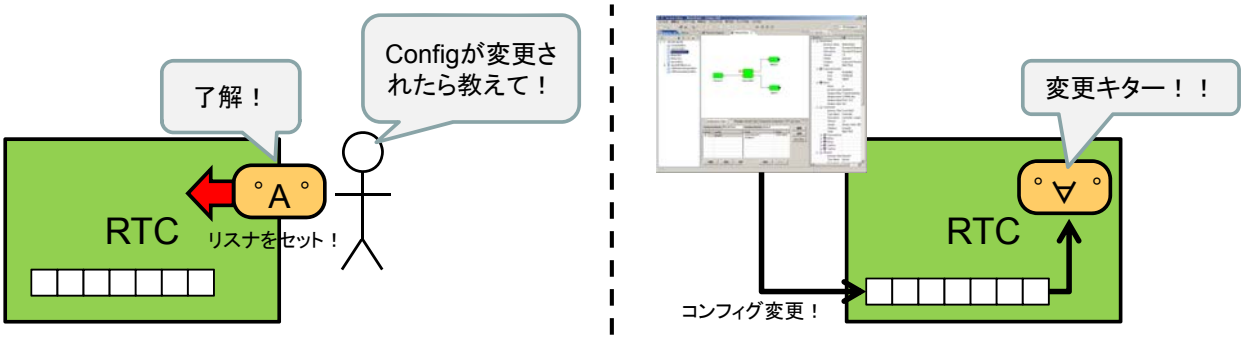
その他

corba.endpoints	IP_Addr:Port で指定:NICが複数あるとき、ORBをどちらでlistenさせるかを指定。Portを指定しない場合でも”:”が必要。 例 “corba.endpoints: 192.168.0.12:” NICが2つある場合必ず指定。 (指定しなくても偶然正常に動作することもあるが念のため。) 使いたいNICに割り当てられているIPアドレス
corba.args	CORBAに対する引数。詳細はomniORBのマニュアル参照。
[カテゴリ名]. [コンポーネント名]. config_file または [カテゴリ名]. [インスタンス名]. config_file	コンポーネントの設定ファイル •カテゴリ名:manipulator, •コンポーネント名:myarm, •インスタンス名myarm0,1,2,… の場合 manipulator.myarm.config_file: arm.conf manipulator.myarm0.config.file: arm0.conf のように指定可能

必須の項目　　必須でないOption設定

コールバック機能

- コンポーネント内で発生する様々なイベントに対してあるアクションを行う機能
  - 例:コンフィグパラメータが変更されたら、画面の表示を更新する
  - 例:ポートが接続されたら、実際に計算を行ってデータを出力、等
- 開発者が予めセットしておいたリスナオブジェクトの特定の関数がイベント発生時に呼ばれる。



イベントタイプ(1)

タイプ	概要	ヘッダ
コンポーネント本体に関連するイベント		
PreComponentActionListenerType	ComponentAction (onInitialize()等)の実行直前に発生するイベント	ComponentActionListener.h
PostComponentActionListenerType	ComponentAction (onInitialize()等)の実行直後に発生するイベント	ComponentActionListener.h
PortActionListenerType	ポートの追加、削除イベント	ComponentActionListener.h
ExecutionContextActionListenerType	実行コンテキストのアタッチ、でタッチなどのイベント	ComponentActionListener.h
コンフィギュレーションパラメータに関連するイベント		
ConfigurationParamListenerType	コンフィグパラメータの更新操作時のイベント	ConfigurationListener.h
ConfigurationSetListenerType	コンフィグパラメータセットの操作時のイベント	ConfigurationListener.h
ConfigurationSetNameListenerType	コンフィグパラメータセットの操作時にイベント	ConfigurationListener.h

イベントごとに、イベントタイプやリスナクラスの型が異なる

イベントタイプ（2）

タイプ	概要	ヘッダ
ポート内部の振る舞いに関連するイベント		
InPort read系 (旧式)	InPortに対してreadを行う際に発生するイベント	PortCallback.h
OutPort write系 (旧式)	OutPortに対してwriteを行うに発生するイベント	PortCallback.h
PortConnectListenerType	ポートの接続時の各種処理に関係するイベント (notify_(dis)connect(), unsubscribeinterfaces()等)	PortConnectListener.h
PortConnectRetListenerType	ポートの接続時の各種処理に関係するイベント (connect_next(), subscribeinterfaces(), 接続切断完了通知)	PortConnectListener.h
ConnectorDataListenerType	データポートのコネクタ内のイベント (bufferフル, send/received などの完了やエラー通知)	ConnectorListener.h
ConnectorListenerType	データポートのコネクタ内のイベント (buffer空, 接続・切断などの完了やエラー通知)	ConnectorListener.h

どのようなイベントがあるかは、クラスリファレンスか
イベントごとのヘッダファイル内のドキュメントを参照

使い方

1. 利用するイベントを決める
   - 例：Configurationの更新
2. イベントタイプを調べる
   - 例：ConfigurationParamListenerTypeのON_UPDATE_CONFIG_PARAMを利用
3. リスナの基底を継承してファンクタを実装
   - 例： ConfigurationParamListenerを継承してMyConfigUpdateParamを実装
4. onInitialize()などでリスナを登録
   - 例： addConfigurationParamListener()
5. イベント発生時にリスナがコールされる

コンポーネントのヘッダファイル

```
#include <rtm/ConfigurationListener.h>

class MyConfigUpdateParam
 : public ConfigurationParamListener
{
  virtual void operator()(const char* config_set_name,
                          const char* config_param_name)
  {
    std::cout << config_set_name << “の”
              << config_param_name
              << “が更新されました” << std::endl;
  }
}
```

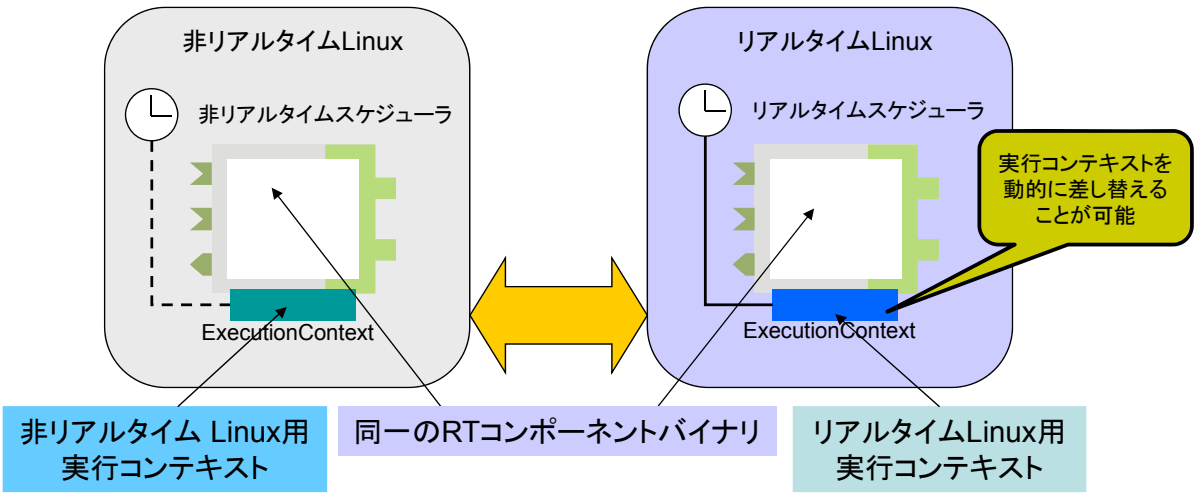
コンポーネントの実装ファイル

```
RTC::ReturnCode_t ConsoleIn::onInitialize()
{
  : 中略
  addConfigurationParamListener(
              ON_UPDATE_CONFIG_PARAM,
              new MyConfigUpdateParam ());
}
```

リアルタイムECの提供

- 2つのリアルタイム実行コンテキスト（EC）
- ArtLinuxEC
  - ARTLinux用の実行コンテキスト
  - 1ms（orそれ以上）の精度でリアルタイム実行が可能
  - Ubuntu用のkernel debパッケージが利用可能なので、インストールし、rtc.confで利用するECをArtLinuxECに指定すれば利用可能
- PreemptEC
  - LinuxのPreemption Pathcedなkernelのリアルタイム機能を利用したEC
  - 1ms程度の精度でリアルタイム実行が可能
  - Ubuntu等では標準でrt-kernelとして提供されている

リアルタイム実行コンテキスト



非リアルタイムLinux環境で作られたRTコンポーネントを
再コンパイルせずにリアルタイムLinux上でリアルタイム実行可能