リアルタイムOSとJava実行環境の統合アーキテクチャJTRON2.1仕様の策定とその評価

中本幸一[†] 高田広章[††,☆] 八谷祥一[†††,☆☆]
朝倉義晴[†] 樫宿昌房[†]

Java言語は組込みシステムで必須なリアルタイム処理には適さないといわれている．本論文では，リアルタイム処理を行う組込みシステムでJava言語を利用するためにリアルタイムOSとJavaプログラムの実行環境を共存させたハイブリッドアーキテクチャJTRON2.1の仕様策定を行い，これに基づいた試作の評価結果と実システムでの応用事例を述べている．JTRON2.1仕様では，Javaスレッドとリアルタイムタスクは通信機能により協調動作を行う．この通信機能には，リアルタイムOSの資源をJavaスレッドから利用するアタッチクラス，Javaオブジェクトをリアルタイムタスクから利用する共有オブジェクト，Java言語のストリーム機能を利用したストリーム通信の3種類がある．この通信機能はリアルタイムOS，Java実行環境上にライブラリとして実現される．このライブラリを利用して，リアルタイム処理はリアルタイムOS上のリアルタイムタスクで処理し，非リアルタイム処理はJava実行環境上のJavaスレッドで実行させるような環境が実現される．

Specification and Evaluation of a Hybrid Architecture JTRON2.1 Integrating a Java Runtime Environment and a Real-time OS

Yukikazu Nakamoto,[†] Hiroaki Takada,[††,☆]
Shouichi Hachiya,[†††,☆☆] Yoshiharu Asakura[†]
and Norifusa Kashijuku[†]

There exist some problems in real-time processing essential in embedded systems when the Java language is applied to the embedded systems. This paper presents specification, evaluation results and applications of a hybrid architecture JTRON2.1, in which the Java runtime environment and a real-time OS coexist in order to utilize the Java language in the embedded systems requiring real-time processing. In JTRON2.1 specification, Java threads and real-time tasks cooperate through communication mechanisms. The communication mechanisms consist of attach classes, shared objects and stream communication. The first one provides methods by which Java threads can access to real-time resources. By the second one, real-time tasks can access to Java objects. The third one utilizes the Java stream to communicate with real-time tasks. The communication mechanisms are implemented by libraries on the real-time OS and the Java runtime environment. Using the libraries, the hybrid architecture is realized, where real-time processing is executed by real-time tasks on the real-time OS and non real-time processing is done by Java threads on the Java runtime environment.

1. はじめに

Java言語[☆☆☆]は多くの組込みシステムの開発においてキーテクノロジの1つとなりつつあり，広い応用領域に適用できるものと期待されている[1)]．また，携帯端末などでは普及が目覚ましい．これは，オブジェクト指向プログラミング言語であること，ハードウェ

† 日本電気株式会社
NEC Corporation
†† 豊橋技術科学大学
Toyohashi University of Technology
††† 株式会社アプリックス
Aplix Corporation
☆ 現在，名古屋大学大学院情報科学研究科
Presently with Graduate School of Information Sciences, Nagoya University
☆☆ 現在，株式会社ガイア・システム・ソリューション
Presently with Gaia System Solutions Inc.

☆☆☆ Javaは米国サンマイクロシステムズ社の登録商標である．TRONは“The Real-time Operating system Nucleus”の略称である．ITRONは“Industrial TRON”の略称である．JTRONは“Java Technology on ITRON”の略称である．

アや OS から独立したシステムを仮想マシンによって提供できること，安全な実行環境を提供することなどセットトップボックスなどの情報家電，携帯電話や工業用コンピュータなどの組込みシステムに適した特徴を有しているからである．一方で，Java 言語は，インタプリティブな実行のため実行時間が遅い，ガーベッジコレクション（GC）やプログラムの動的なリンクやロードという予測不能な実行時間が存在する，リアルタイムスケジューリングの観点からはスレッドのセマンティクスに曖昧なところがあるなどの欠点があり，組込みシステムで必須なリアルタイム処理には適さないといわれている．

Java 言語をリアルタイム環境で利用するための手段として，Java 言語のリアルタイムシステム向けへの拡張がある．これは，Java 言語の仕様，およびクラスライブラリをリアルタイム処理に適したセマンティクスに拡張するものである[2),3)]．文献 2) では Java 言語を補完する形でリアルタイムスレッド，非同期イベントなどのクラスを定義している．文献 3) ではハードリアルタイム処理専用に Java 言語のセマンティクスを変更し，専用のクラスパッケージを用意している．ただし，このアプローチはシステム全体を統一的に単一言語で記述できる長所がある一方で，リアルタイムシステム全体を記述しなおす必要があり，既存のソフトウェアの再利用ができないという短所がある．

筆者らは，Java 言語をリアルタイム機能を有する組込みシステムで利用可能とするために，リアルタイム OS と Java 実行環境を共存させたハイブリッドアーキテクチャJTRON2.1 の仕様策定[4),5)] をトロン協会 ITRON 部会 JTRON WG で行い，これに基づき試作し評価を行った．

本論文の構成は以下のとおりである．2 章では Java 実行環境・リアルタイム OS のハイブリッドアーキテクチャ仕様を策定するにあたって考慮した要求事項を述べ，関連研究との関係を述べる．3 章では本ハイブリッドアーキテクチャにおいて Java スレッドとリアルタイムタスク間の協調動作モデル，4 章ではこのモデルに基づく協調動作ライブラリの API 仕様を述べる． 5 章では試作とその評価結果を，6 章では実システムへの応用事例を，7 章でまとめと今後の課題をそれぞれ述べる．

2. 要求事項のまとめと関連研究

情報家電や携帯電話などの組込みシステムでは，次々と新しい技術が現れ，機能やサービスの展開が速い．こうした背景の下に，Java 言語をリアルタイム機能を有する組込みシステムで利用可能とするための実行環境に対して，以下のように要求事項を整理した．

RM1： Java 言語の仕様の拡張・変更だけでなく，Java 実行環境の修正を避けるようにすること．これにより，既存の Java プログラムの流用が容易となる．また，広く普及している Java 関連の技術や様々な特徴を持つ開発ツールなども利用することができる．

RM2： 既存のリアルタイムタスクを再利用できるようにすること．これは，すでに工業用システム，通信システムなど多くのリアルタイムソフトウェアが存在するからである．このためには既存のリアルタイムタスクの処理に対して影響を与えないようにする必要がある．

RM3： Java 実行環境とリアルタイム OS の複数のプログラム実行環境を共存させることによるオーバヘッドは最小限にすること．

RM4： Java クラスライブラリの仕様決定の際に，可能な限り実装に依存する仕様をプログラマが利用する API（Application Programming Interface）から分離すること．

次にこうした要求事項に対して OS 研究における関連研究の有効性を考察する．Java 言語をリアルタイム機能を有する組込みシステムで利用可能とするための実行環境は，Java プログラムの実行環境とリアルタイム OS が単一ハードウェア上で動作する異種 OS 環境ととらえることができる．これまで，単一ハードウェア上に複数の OS 機能を実現するマイクロカーネル，仮想マシン[6),7)] が研究開発されてきた．

マイクロカーネルは最小限のプロセス管理，メモリ管理，プロセス間通信を具備し，マイクロカーネル上のプロセスとして複数の OS 機能を実現するものである．マイクロカーネル上で既存の OS 機能を実現する場合，この OS 機能をマイクロカーネルが具備する機能を利用して改造する必要があるため，**RM1** の要求を満たすことは難しいと考えられる．一方，仮想マシンは下位のハードウェアの精密な複製を仮想ハードウェアとして上位の OS に提供するものであり，下位のハードウェアの複製を精密に実現するためのオーバヘッドが大きくなるといわれている[6)]．このため，仮想マシンでは，OS を含む既存ソフトウェアの流用は容易であり要求事項の **RM1** の要求は満たす．しかし，**RM2，RM3** の要求を満たすことは，PC に比べて性能が低い CPU が利用される組込みシステムでは難しいと考えられる．このような考察のもとに，筆者らは次章に述べるハイブリッドアーキテクチャを選択することにした．

3. Java 実行環境・リアルタイム OS 統合アーキテクチャ

上述した **RM1** から **RM4** を満たす統合アーキテクチャとして，以下に述べる特徴を持つハイブリッドアーキテクチャを採用した．

Java 実行環境・リアルタイム OS 統合アーキテクチャで採用したアーキテクチャでは，リアルタイム OS 上に Java 実行環境を実装する．このとき，Java 実行環境の実行単位であるスレッドはリアルタイム OS 上のタスクにより実現する．なお，Java 実行環境上の並列実行単位をスレッド，リアルタイム OS 上の並列実行単位をタスクと，各実行環境での標準的な使用に従い呼ぶことにする．CPU 以外のハードウェア資源とソフトウェア資源はリアルタイム OS と Java 実行環境で分離して実装する．これにより **RM2，RM3** を実現することができる．

スレッドをリアルタイム OS 上で実現する場合に，次の 3 つの実現方法が考えられる[8]．

(1) Java 実行環境上の全スレッドを 1 つのタスクで実現する．

(2) Java 実行環境上の 1 スレッドをリアルタイム OS 上の 1 つのタスクに対応させる．

(3) Java 実行環境上の全スレッドを複数のタスクで実現する．

(1) は，1 つのタスクによりスレッドのスケジューラを実現し，その上で全スレッドを実行させるものであり，あるスレッドがブロックした場合にスケジューラを実装するタスクがブロックし，その結果，他のスレッドもブロックしてしまい，効率が悪くなるという欠点がある．また，(3) は他の 2 つの方法よりも複雑になるという欠点がある．このため，筆者らは (2) をスレッドとタスクの協調動作のためのインタフェース策定の前提とした． なお，(2) の方法で 1 スレッド実行に必要なリアルタイム OS のメモリは 100 バイト前後（5 章の実装で使用したリアルタイム OS では 140 バイト）であり，JTRON 仕様の対象とする組込みシステムでは大きな問題にならないと判断した．

このようにリアルタイム OS 上に汎用コンピュータ向け OS 機能を統合したハイブリッドアーキテクチャを持った OS には，これまでリアルタイム UNIX[9]，リアルタイム Linux[10),11)]，Windows NT☆のリアルタイム拡張[12),13)] がある． 従来のハイブリッド OS では，両 OS 上のプログラム間の通信機構として単純なメッセージ通信が提供されているのみで[10),11)]，あるいはリアルタイム OS のシステムコールを汎用 OS 側に提供しているのみ[9),12),13)] で，両 OS を柔軟に活用して効率的なシステムを短期間で構築することが難しい．

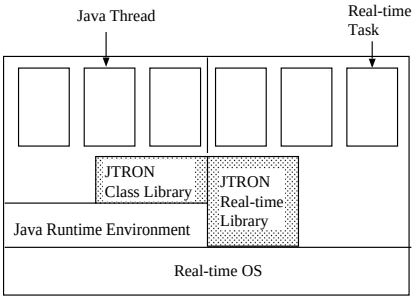

図 1 リアルタイム OS と Java 実行環境のハイブリッドアーキテクチャ
Fig. 1 Hybrid architecture integrating a real-time OS and the Java runtime environment.

これに対して，本論文では Java 実行環境上のスレッドとリアルタイム OS 上のタスクとの間で協調動作する環境を提供している点，さらに Java 実行環境の実装が多様であるためこれらの実装に最適な実装が選択できるように，OS システムコールレベルの API よりも抽象度の高い API を設定している点が異なる．

図 1 に Java 実行環境とリアルタイム OS が共存するハイブリッドアーキテクチャのソフトウェア構造を示す．本ハイブリッドアーキテクチャでは，スレッドとタスクは通信機構を介して協調動作する．典型的には，タスクは常時起動し，外界からデータを取得し外界にコマンドを送出して，リアルタイムの制御を行う下位モジュールとして働き，スレッドはタスクからデータを取り出し，HMI（Human Machine Interface）を介してそのデータに対して制御を行う上位モジュールとして働く機能分割する形態が考えられる．このモジュール間で通信機構を通じてデータとコマンドのやりとりが行われる．

この協調動作機能はリアルタイム OS 上のライブラリと Java 実行環境上のライブラリによって実現される．このライブラリをそれぞれ JTRON リアルタイムライブラリ，JTRON クラスライブラリと呼ぶことにする．これらのライブラリは協調動作のための API を提供する．

この協調動作機能を使うことによって，組込みシステムにおける Java 言語の標準的なプログラミングモデルを定義できる．これはリアルタイムプログラムのみならず Java プログラムの可搬性と再利用性を向上

☆ Windows NT は Microsoft 社の登録商標である．

させる効果も期待できると考えられる．

4. JTRON2.1 仕様

本章では，Java 実行環境とリアルタイム OS のハイブリッドアーキテクチャを構成する協調動作機能のインタフェース仕様を述べる．4.1 節で両者の間の協調動作モデルについて述べる．次に 4.2 節以降で，協調動作モデルに基づくインタフェース仕様を述べる．

4.1 Java スレッドとリアルタイムタスクの協調動作モデル

JTRON2.1 仕様（以下，JTRON 仕様と略する）で導入したスレッドとタスク間の協調動作モデルを述べる．ハイブリッドアーキテクチャでは，リアルタイム処理はリアルタイム OS で処理され，非リアルタイム処理は Java 実行環境で実行される．したがって，ハイブリッドアーキテクチャにおけるプログラムの協調動作はスレッドとタスクが機能分散型マルチプロセッサシステムを構成するものと見なしてモデル化することができる．マルチプロセッサシステムにおける通信機構に基づいて，スレッドとタスクの協調動作モデルを以下のように分類した．

タイプ 1： アタッチクラス機能
タイプ 2： 共有オブジェクト機能
タイプ 3： ストリーム通信機能

タイプ 1 ではリアルタイム OS の提供する資源やターゲットシステムのメモリにアタッチクラスと呼ばれる Java クラスライブラリを通じてスレッドがアクセスするものである．タイプ 2 ではタイプ 1 とは逆に，スレッドの提供する資源やオブジェクトにタスクがアクセスする形態であり，Java プログラムのオブジェクトをタスクが共有する形となっている．タイプ 3 ではタスクとスレッドは通信機能（Java 標準ライブラリのストリーム機能）を使って協調動作を行うものである．機能分散型マルチプロセッサと対応させると，タイプ 1 とタイプ 2 は分散された機能間で資源を共有するので密結合型マルチプロセッサシステムに相当し，タイプ 3 は分散された機能間で資源を共有せず通信により協調動作するので疎結合型マルチプロセッサシステムに相当する．

4.2 アタッチクラス機能（タイプ 1）

アタッチクラス機能仕様を策定するにあたり，リアルタイム OS の資源と API を素直に Java クラス仕様に反映させる方針をとった．これはリアルタイム OS の知識があれば容易に本クラス仕様が理解できることを目指したためである．ここではリアルタイム OS として，ITRON 仕様 OS をベースにした．

アタッチクラス機能はスレッドから ITRON 仕様 OS の機能を利用する手段を提供する[14]．OS の資源ごとにそれに対応するクラスを設け，そのクラスのインスタンスを OS の資源の実体（資源オブジェクトと呼ぶ）に対応させる． アタッチクラスで利用可能な OS 資源は，μITRON4.0 仕様で規定するタスク，セマフォ，イベントフラグ，データキュー，メールボックス，ミューテックス，メッセージバッファ，ランデブ，メモリプール，周期ハンドラ，アラームハンドラ，割込みサービスルーチンである． アタッチという名称はクラスのインスタンスが資源オブジェクトに 1 対 1 に対応するところからきている．インスタンスと資源オブジェクトを対応させる方法として，インスタンス生成時に以下のどちらかを選択することができる．

(1) 資源オブジェクトを生成し，それをインスタンスに関連づける．
(2) すでに存在する資源オブジェクトにインスタンスを関連づける．

資源オブジェクトへの操作は関連づけたインスタンスのメソッドを呼び出すことで行うので，オブジェクトの識別子を指定する必要はない．アタッチクラスの例を**図 2** に示す．`MailBox` クラスは ITRON 仕様 OS のメールボックスを利用するためのクラスである．メールボックスはメッセージを送受信するための資源オブジェクトである．メールボックスオブジェクトの識別子を指定しないコンストラクタ `MailBox` は `MailBox` インスタンスと ITRON 仕様 OS 上でメールボックスの資源オブジェクトを生成する．メールボックスオブジェクトの識別子のみを指定するコンストラクタ `MailBox` は `MailBox` インスタンスのみ生成し，メールボックスオブジェクトの識別子で指定された既存のメールボックスと関係づける．`send` メソッドはメッセージを送信するためのメソッドであり，`receive` メソッドはメッセージを受信するためのメソッドであり，各々ITRON 仕様 OS のシステムコール `snd_mbx`，`rcv_mbx` の呼び出しに相当する．

なお，資源オブジェクトの遅延生成（lazy initialization）はサポートしていない．これは，遅延生成により資源オブジェクトを実際利用する際に遅延時間が生じるためである．

Java 言語にはメモリアドレスの概念がないので，ITRON 仕様 OS の機能を利用する場合に必要なアドレスを指定することができない．またその内容を操作することもできない．アタッチクラスではアドレスを意識しないでメモリを扱うメモリ操作クラス（`ItronMemory`）を提供する．メモリ操作クラスを用

```
public class MailBox {//メールボックスのアタッチクラス
  public MailBox(int mbxid, T_CMBX pk_cmbx)
    throws ItronCauseException;
    //新たに pk_cmbx で指定されたメールボックスの
    //インスタンスと mbxid のオブジェクトを生成し，両者を関係づける．
    //T_CMBX はメールボックス生成に必要な情報を持つクラスである．
  public MailBox(int mbxid) throws JtronException;
    // 新たにメールボックスのインスタンスを生成し，
    // mbxid で指定された既存のオブジェクトに関係づける．
  public MailBox(T_CMBX pk_cmbx) throws ItronCauseException;
    //新たに pk_cmbx で指定されたメールボックスのインスタンスと
    //オブジェクトを生成し，両者を関係づける．
  public int getId();
    //関係づけられたメールボックスの ID を返す．
  public void delete() throws ItronCauseException;
    //メッセージボックスのインスタンスと対応づけられたオブジェクトを
    //削除する
  public void send(ItronMemory pk_msg) throws ItronCauseException;
    //メッセージ pk_msg をメールボックスに送信する．
  public ItronMemory receive(int length) throws JtronException;
    //長さ length のメッセージを受信する．メッセージがなければ到着を待つ．
  public ItronMemory pollReceive(int length) throws JtronException;
    //メッセージが存在すればそれを受信する．
  public ItronMemory receive(int length, int tmout) throws
    JtronException;//待ち時間 tmout を指定してメッセージを受信する．
  public T_RMBX refer(int length) throws JtronException;
    //メールボックスの状態を長さ length で参照する．
    //T_RBX はメールボックスの状態を表すクラスである．
  public static void seekNextToHeader(ItronMemory msg);
    //メッセージヘッダのアクセス位置をメッセージヘッダの直後に
    //移動させる．
  public static int readPriority(ItronMemory msg);
    //優先度付メッセージの優先度を返す．
  public static void writePriority(ItronMemory msg, int msgpri);
    //優先度付メッセージの優先度を設定する．
}
public class ItronMemory {
  public ItronMemory(int length) throws JtronException;
    //指定されたサイズのメモリを生成する．
  public void release() throws JtronException;// メモリを解放する．

  public void disableWrite() throws JtronCauseException;
    // メモリへの書き込みを禁止する．
  public void enableWrite() throws JtronCauseException;
    // メモリへの書き込みを許可する．
  public void seek(int offset) throws JtronCauseException;
    // 指定された offset 位置にカレントポインタを移動する．
  public void skipBytes(int n) throws JtronCauseException;
    // n バイトカレントポイントを移動させる．
  public byte readB(int offset) throws JtronCauseException;
    // offset 位置から 1 バイト読み出す．
              :
  public int read(byte[] b) throws JtronCauseException;
  // カレントポインタからデータを読み出し，バイト配列 b に格納する．
              :
  public void writeB(int offset, byte value) throws
    JtronCauseException;// offset 位置に value を 1 バイト書き込む．
              :
  public int write(byte[] b) throws JtronCauseException;
  // カレントポインタにバイト配列 b を書き込む．
              :
}
```

図 2 アタッチクラス
Fig. 2 Attach classes.

いるとメモリ内容の読み書きがストリームないしはランダムアクセスファイルに対する入出力と同様な方法で行うことが可能になる（図 2）．メモリ内容の読み書き時に，ITRON 仕様 OS で定義している B（バイト）や H（ハーフワード），W（ワード）型を指定する．また，配列を使って複数のデータの読み書きを行うこともできる．メモリ操作クラスではメモリ領域の

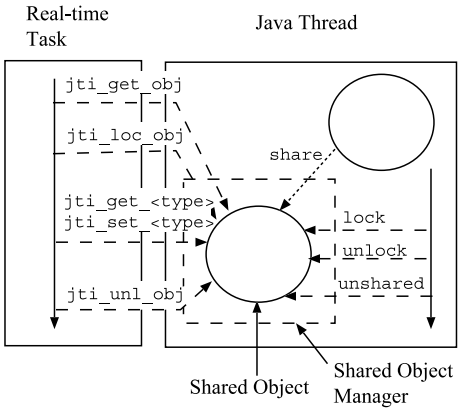

図 3 共有オブジェクトモデル
Fig. 3 Shared object model.

範囲を管理しており，この領域外のアクセスを行った場合には例外が発生する．これにより決められた範囲のメモリにしかアクセスできないので安全性が維持できる．また，より安全性を高められるようメモリ領域を書き込み禁止に設定することも可能である．

4.3 共有オブジェクト機能（タイプ 2）

共有オブジェクト機能仕様を策定するにあたって，タスクとスレッドは比較的小規模なデータを簡単な同期機構で通信するものという方針をたてた．これから大規模なデータ通信や複雑な同期機構が必要な通信はタイプ 1，もしくはタイプ 3 を利用するものとした．

共有オブジェクト機能では，スレッドとタスク間でデータの交換を行うことによる通信手段とタスクからスレッドを制御する手段を提供する（**図 3**）．スレッドでは共有対象となるオブジェクトをタスク側にエクスポートして，タスクで共有し利用できるようにする．このオブジェクトを共有オブジェクトと呼ぶ．タスクではスレッドで作成された，共有オブジェクトのデータにアクセスする．共有オブジェクト機能の利用形態としては，タスクは常時動作してデータの収集を行って取得したデータを共有オブジェクトに設定し，スレッドは必要なときに共有オブジェクトを参照するというものが考えられる．

共有オブジェクトへのアクセスに対して排他制御のための機能が必要になる．これには次のような手法が考えられる．なお，以下ではタスク，スレッドのいずれかを実行単位と呼ぶことにする．

排他制御や同期機能を実現する方法として，1 つは Java 言語の言語構成要素である `synchronized` ブロックや `wait/notify` を利用する方法，もう 1 つは排他制御の機能を持ったメソッドを用意し，これを呼び出すことにより排他制御を実現する方法が考えられる．

プログラマに学習の負担をかけないという観点からは，前者の既存の言語構成要素を利用する方が望ましい．しかし，我々は要求事項 **RM1** に基づき，Java 実行環境の修正を不要にするという要求から後者を選択した．

JTRON 仕様ではロック操作による排他制御機能のみを用意することにした．これは共有オブジェクトでは複雑な同期機能は必要でないという方針による．

次に排他制御のための共有オブジェクトに対するロック操作の仕様を述べる．ここでロック操作を実行した実行単位をロックのオーナと呼ぶことにする．

- ロック操作：共有オブジェクトをロックする．共有オブジェクトがすでにロックされている場合はロックしようとした実行単位はブロックされる．
- アンロック操作：ロックのオーナと同じ実行単位がロックを解除する．同一実行単位でロック操作，アンロック操作を行うようにしているのは，誤って他の実行単位によりロックが解除されないようにするためである．ロックを解除しようとした実行単位とロックのオーナが異なる場合は JTRON リアルタイムライブラリではエラー，JTRON クラスライブラリでは例外が発生する．
- 強制アンロック操作：ロックを解除しようとする実行単位とロックのオーナが異なる場合であってもロックを解除する．この操作はエラー発生時のリカバリに用いることを想定している．

リアルタイムライブラリには，タスクから Java スレッドを制御するスレッド操作 API がある．スレッド操作 API は，タスクから共有オブジェクトに対して操作を行う場合に，Java スレッドの一時停止，停止解除を行うためのものである．これには，スレッド，スレッドグループのオブジェクト識別子を取得する API と Java 言語☆の `Thread` と `ThreadGroup` クラスで提供されているメソッドの中で状態遷移を起こすメソッドを呼び出す API がある．

共有オブジェクトに関する JTRON クラスライブラリの概要を述べる．クラスの定義を**図 4**，さらに UML によるクラス関係図を**図 5** に示す．

インタフェースは `Sharable` は共有オブジェクトの実装に必要なメソッドを定義したものである．これらのメソッドは次に述べる共有オブジェクトマネジャ `SharedObjectManager` クラスを使って実装されるものとしている．クラス `SharedObject` はインタフェース `Sharable` を実装したものである．クラス `SharedObject` を継承した共有オブジェクトは，クラス `SharedObject` のコンストラクタを実行することによって共有オブジェクトマネジャに登録される． クラス `SharedObjectManager` は共有オブジェクトマネジャとして共有オブジェクトを管理する．このようなクラス構成にしたのは，要求事項 **RM4** に基づいている．

ここでは共有オブジェクトのクラスを作る方法とし

☆ JTRON 仕様では，Java 言語仕様 1.1 以降をもとにしている．

```
public interface Sharable {
  public void lock();//ロック操作を行う.
  public void lock(int timeout) throws ShmTimeoutException;
    //タイムアウト時間を指定してロックする.
  public void unlock()  throws ShmIllegalStateException;
    //アンロック操作を行う.
  public void forceUnlock();
    //強制的アンロック操作を行う.
  public void unshare() throws ShmIllegalStateException;
    //オブジェクトの共有を終了させる．本メソッドを実行した時,
    //対象となるオブジェクトがロックされている場合はロックが
    //解除されるまでブロックされる.
  public void unshare(int timeout)
    throws ShmTimeoutException, ShmIllegalStateException;
    //タイムアウト時間を指定して，オブジェクトの共有を終了させる.
  public Object getContent();
    //共有されたオブジェクトを返す.
};

public class SharedObject extends Object implements Sharable{
  protected Sharable shm;
  public SharedObject(String name) throws ShmIllegalStateException;
    //本オブジェクトを name という名前で共有オブジェクトとして登録する.
  public SharedObject(Object obj, String name)
    throws ShmIllegalStateException;
    //オブジェクト obj を名前 name で共有オブジェクトとして登録する.
    //オブジェクトがロックされている場合はロックが解除されるまで
    //ブロックする.
  public void lock();
  public void lock(int timeout) throws ShmTimeoutException;
  public vpid unlock() throws ShmIllegalStateException;
  public void forceUnlock();
  public void unshare() throws ShmIllegalStateException;
  public void unshare(int timeout)
    throws ShmTimeoutException, ShmIllegalStateException;
  public Object getContent();
}
// 以下は実装依存クラスである.
public class SharedObjectManager {
  public static ShareaObjectManager
    getSharedObjectManager() throws ShmIllegalStateException;
    //SharedObjectManager を返す.
  public void share(Sharable obj, String name)
    throws ShmIllegalStateException;//SharedObjectManager に
    //obj を name という名前で共有オブジェクトとして登録する.
  public void  unshare(String name)
    throws ShmIllegalStateException;
    //name の共有オブジェクトの共有を終了する.
  public void  unshare(String name, int timeout)
    throws ShmIllegalStateException;//タイムアウト時間を
    //指定して name の共有オブジェクトの共有を終了する.
  public void lock(Sharable obj);
    //obj に対してロック操作を行う.
  public void lock(Sharable obj, int timeout)
    throws ShmTimeoutException;
    //タイムアウト時間を指定して obj に対してロック操作を行う.
  public void unlock(Sharable obj)
    throws ShmIllegalStateException;
    //obj に対してアンロック操作を行う.
  public void forceunlock(Sharable obj);
    //obj に対して強制アンロック操作を行う.
}
```

図 4 共有オブジェクトクラス
Fig. 4 Shared object classes.

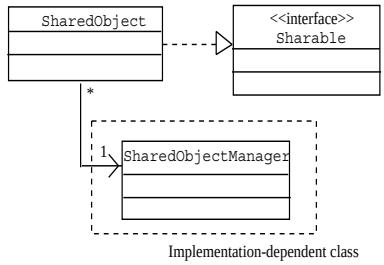

図 5 共有オブジェクトクラスの関係
Fig. 5 Relationship of shared object classes.

て以下の手段を提供している．

(1) 共有対象のクラスを SharedObject のインスタンスとしてもたせる方法

(2) 共有対象のクラスで共有オブジェクトクラス SharedObject を継承する方法

(3) 共有対象のクラスで共有オブジェクト用インタフェース Sharable を実装する方法

(1), (3) は一般のクラスを共有オブジェクト化するために，(2) はスーパクラスのないクラスを簡単に共有オブジェクト化するために用意している．また，(1), (2) では共有オブジェクトの実装方法がクラスライブラリで固定されてしまっているのに対して，(3) ではプログラマが必要に応じて実装方法を変更することができる．

共有オブジェクト機能を実現する JTRON リアルタイムライブラリで提供する API の一覧を**図 6** に示す．これらは共有オブジェクトアクセス用 API とスレッド制御 API に分類される．この中で jti_get_<type>, jti_set_<type> メソッドを説明する．これらのメソッドは，タスクから共有オブジェクトにアクセスする場合に，共有オブジェクトのクラス名，フィールド名とフィールドのデータ型を指定して，タスクとの間でのデータの受け渡しを行うものである．<type> には，Java 言語の boolean, byte などの型名が入る．クラス名の指定には，Java Native Interface (JNI)[16] の FindClass のメソッドの引数と同じ方式としている（たとえば，java.lang.String の場合，"java/lang/String" となる）．フィールド名の指定には Java プログラム中のクラス定義のフィールド名がそのまま使用される．このようなアクセスを行うことができるように，共有可能な Java オブジェクトは，単純型のフィールドから構成されるものとし，配列やインナークラスは利用できないものとしている．これは共有オブジェクト機能は比較的小規模なデータの共有化を行うものとしているからである．

共有オブジェクト機能の利用手順を述べる．まず，ス

```
typedef integer INT;    /* integer の型 */
typedef long ID;        /* 識別子の型 */
typedef long TMO;       /* タイムアウト値の型 */
typedef void *VP;       /* 可変長データ領域へのポインタ型 */
typedef unsigned long ATR;/* ストリームの属性のための型 */
typedef integer JNO;   /* Java スレッドの識別子の型 */

/* 共有オブジェクトアクセス API */
jti_get_obj(char *objnm, JNO *p_objno);
/* objnm に対応する共有オブジェクトの識別番号を p_objno に返す*/
jti_get_<type>(JNO objno, const char *classname,
        const char *fieldname, <c_type> *p_retval);
/* objno で指定された共有オブジェクトにおいて，クラス名 clsnm とフィールド名 fldnm で指定されたフィールドで，<type>に対応する<c_type>型のデータを返す．<type>には booelean, byte 等 Java 言語の型名が入り，clsnm と fldnm で指定されたクラスフィールドの型が，<type>に一致していなければならない．*/
jti_set_<type>(JNO objno, const char *classname,
   const char *fieldname, <c_type> val);
/* objno で指定された共有オブジェクトのクラス名 clsnm とフィールド名
   fldnm で指定された<type>に対応する<c_type>型のデータを設定する．*/
jti_loc_obj(JNO objno, TMO tmout);
/* objno で指定された共有オブジェクトをロックする．*/
jti_unl_obj(JNO objno);
/* objno で指定された共有オブジェクトをアンロックする*/
jti_funl_obj(JNO objno);
/* objno で指定された共有オブジェクトを強制的にアンロックする．*/
/* スレッド操作 API */
jti_get_thr(char *thrnm, JNO *p_thrno);
/* Java スレッドの名前 thrnm に対応する Java スレッド識別番号
   thrno を返す．*/
jti_sus_thr(JNO thrno);
/* thrno に対応するスレッドに対して suspend メソッドを実行する．*/
        :
jti_get_tgr(char *tgrnm, JNO *p_tgrno);
/* Java グループの名前 tgrnm に対応する Java スレッドグループ識別番号
   tgrno を返す．*/
jti_rsm_tgr(JNO tgrno);
/* tgrno に対応するスレッドグループに対して resume メソッドを実行する．*/
        :
```

図 6 共有オブジェクトのためのリアルタイムライブラリ API
Fig. 6 Real-time APIs for shared object.

レッド側から共有対象オブジェクトと SharedObject メソッドなどを利用して共有オブジェクトとして名前を付与して登録する．タスクではその名前を指定して共有オブジェクトの識別子を取得する．共有オブジェクトアクセス時は，ロック，アンロック（スレッドの場合 SharedObject の lock メソッド，unlock メソッド，タスクの場合は，jti_loc_obj，jti_unl_obj API）を行い排他制御する．タスクからは共有オブジェクトをロックしたあと，jti_get_<type>，jti_set_<type> API を使ってアクセスし，アクセス完了後アンロックする．Java スレッドから unshare メソッドを実行することにより，共有オブジェクトの共有を終了させることができる．

jti_loc_obj 実行から jti_unl_obj メソッド実行まで予測不可能な時間が発生するのは jti_loc_obj 呼び出し時である．jti_loc_obj 呼び出し時にタイムアウト時間を指定することが可能であり，タスク側で一定時間以上たってもロックができない場合はエラー処理を行うことになる．典型的な処理として，周期的にデータ送信を行っている場合に，この周期のデータ送

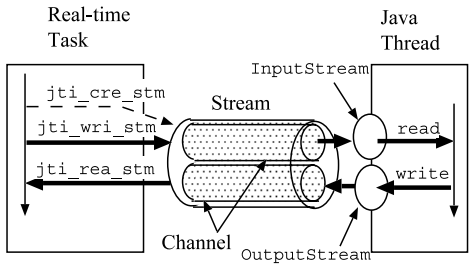

図 7 ストリーム通信モデル
Fig. 7 Stream communication model.

信をとりやめ，次の周期にデータ送信を行うなどが考えられる．

4.4 ストリーム通信機能（タイプ 3）

タスクとスレッド間の通信機能は，Java 言語のストリーム機能を利用する方針を立てた．これは既存の仕様を利用することにより，新たな機能を学習するなどプログラマに負担をかけないためである．

ストリーム通信機能は，Java 言語の標準的な入出力インタフェースである InputStream，OutputStream クラスを利用して，タスクとスレッド間の通信手段を提供するものである．**図 7** にこのモデルを示す．

ストリーム通信機能は，タスクからスレッドへデータを送るためのチャネルと，スレッドからタスクへデータを送るためのチャネルの，2 つのチャネルから構成される．また，1 方向のチャネルのみ生成することもできる．スレッドは 2 つのチャネルに対して，InputStream，OutputStream クラスを実装したクラスでそれぞれアクセスする．タスクは，JTRON リアルタイムライブラリで提供されるストリーム通信機能の API を呼び出す．

ストリーム通信機能に関連する JTRON クラスライブラリを**図 8** に示し，UML によるクラス関係を**図 9** に示す．JtronStream クラスはタスクと通信するストリーム機能を提供し，JtronStreamImpl クラスはストリーム通信の実装を持つクラスを定義するための抽象クラスである．JtronStream を実装するにあたって，ブリッジ[15] と呼ばれるフレームワークを使って，JtronStream とその実装である JtronStreamImpl を分離した．JtronStream クラスの各メソッドは抽象クラス JtronStreamImpl のメソッドを呼び出すように記述し，実際のメソッドの実装は JtronStreamImpl クラスを継承し実装したクラスで行う．ブリッジパターンを用いることにより JtronStream を実装したクラスを交換するだけで様々な機器に対応することができる．タスクと通信するストリームは，InputStream，OutputStream の抽象クラスの実装として提供される．

```
public class JtronStream {
  public JtronStream(int stmid) throws
    JtronStreamIllegalStateException;
    //stmid を識別子としてストリームを生成する.
  public JtronStream(int stmid, int timeout) throws
    JtronStreamIllegalStateException,JtronStreamTimeoutException;
    //タイムアウト時間を指定してストリームを生成する.
  public synchronized InputStream getInputStream()
    throws JtronStreamIllegalStateException; //受信ストリームを取得
                                             //する.
  public synchronized OutputStream getOutputStream()
    throws JtronStreamIllegalStateException; //送信ストリームを取得
                                             //する.
  public synchronized void setTimeOut(int timeout)
    throws JtronStreamIllegalStateException;
    //InputStream への read メソッド実行時にタイムアウト時間を設定
  public synchronized int getTimeOut() throws
    JtronStreamIllegalStateException;//上述のタイムアウト時間を取得
  public synchronized void close() throws
    JtronStreamIllegalStateException;//ストリームをクローズする.
}

public abstract class JtronStreamImpl {
  public abstract void setStreamId(int stmid);
  public abstract int getStreamId();
    //ストリーム識別子の設定，取得を行う.
  public abstract InputStream getInputStream() throws
    JtronStreamIllegalStateException;//送信ストリームを取得する.
  public abstract OutputStream getOutputStream(); throws
    JtronStreamIllegalStateException;//受信ストリームを取得する.
  public abstract void setTimeout(int timeout) throws
    JtronStreamIllegalStateException;//InputStream に対して
    //read メソッドを発行した場合のタイムアウト時間を設定する.
  public abstract int getTimeout() throws
    JtronStreamIllegalStateException;
    //上述したタイムアウト時間を取得する.
  public abstract void close() throws
    JtronStreamIllegalStateException;
}
// 以下は実装依存クラスである.
public class PlainJtronStreamImpl extends JtronStreamImpl {
  public PlainJtronStream(int stmid) throws
    JtronStreamIllegalStateException;
    //指定された識別子でストリームをオープンする.
  public PlainJtronStream(int stmid, int timeout)  throws
    JtronStreamIllegalStateException;
    //タイムアウト時間を指定してストリームをオープンする.
    //以下は JtronStremImpl のメソッドを実装したものである.
  public void setStreamId(int stmid);
  public int getStreamId();
  public InputStream getInputStream();
  public OutputStream getOutputStream();
  public void setTimeout(int timeout);
  public int getTimeout();
  public void close();
}
```

図 8 ストリーム通信クラス
Fig. 8 Stream communication classes.

なお，図 9 の破線部分は，API 仕様上規定する必要がないので実装依存としており，5 章で述べる．

ストリーム通信機能を提供する JTRON リアルタイムライブラリ API を**図 10** に示す．ストリーム通信のための資源の生成・削除はタスクから行うことにした．これはタスクが外部データを取得し，ストリームデータとして送信することが多く，ストリーム通信の制御はタスク側で行うべきと考えたためである．ストリーム通信にはブロック型と非ブロック型の通信を用意した．したがって，ブロック型の read/write 操

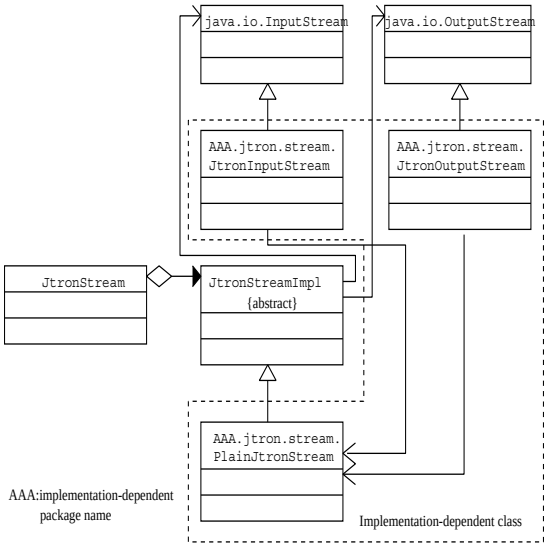

図 9 ストリーム通信クラスの関係
Fig. 9 Relationship of stream communication classes.

```
typedef struct t_jti_cstm{
    VP  exinf;/*実装依存データの格納領域*/
    ATR stmatr; /* ストリームの属性を示す.
        TA_WRITE:送信する，TA_READ:受信する */
    VP *wbuf;   /* 送信バッファの先頭アドレス */
    INT wbufsz; /* 送信バッファの大きさ */
    VP *rbuf;   /* 受信バッファの先頭アドレス */
    INT rbufsz; /* 受信バッファの大きさ */
} T_JTI_CSTM; /* ストリーム生成時のパラメタを指定する型 */
typdef struct t_jti_rstm {
    VP  exinf;  /* 実装に依存したデータ*/
    INT wrisz;  /* 直ちに送信可能なデータ長 (バイト)*/
    INT reasz;  /* 直ちに受信可能なデータ長 (バイト)*/
} T_JTI_RSTM; /* ストリームの状態を示す型 */
jti_cre_stm(ID stmid, T_JTI_CSTM *pk_cstm);
/* pk_cstm で指定されたパラメタで識別子 stmid のストリームを生成する．*/
jti_del_stm(ID stmid);/* 指定されたストリームを削除する．*/
jti_wri_stm(ID stmid, VP data, INT len, TMO tmout);
/* data で指定された長さ len のデータを送信する．タイムアウト時間
   tmout が正の場合は指定された時間（ミリ秒）の待ちを意味し，
   0 の場合はポーリング，-1 の場合は無限待ちを意味する．*/
jti_rea_stm(ID stmid, VP data, INT len, TMO tmout);
/* data で指定された長さ len のデータを受信する．タイムアウト時間
   tmout が正の場合は指定された時間（ミリ秒）の待ちを意味し，
   0 の場合はポーリング，-1 の場合は無限待ちを意味する．*/
jti_sht_stm(ID stmid);
/* ID で指定されたストリームのデータ送信を終了する．*/
jti_ref_stm(ID stmid, T_JTI_RSTM *pk_rstm);
/* ストリームの状態参照を行う．*/
```

図 10 ストリーム通信のためのリアルタイムライブラリ API
Fig. 10 Real-time APIs for stream communication.

作を行うことにより，同期を実現することができる．

JTRON2.1でのストリーム通信機能の概要を述べる．(1) タスクからスレッドへストリームを送信する場合：スレッドでストリーム識別子を指定して JtronStream クラスのコンストラクタを呼び出すことにより，ストリームを生成しオープンすることができる．ストリームがオープンされると，ストリームのチャネルが接続状態になる．接続状態になるとタスクから jti_wri_stm を使ってデータを送信し，スレッドは InputStream.read メソッドを使ってデータを受信する．ストリーム通信を終了するために，タスク側から先に jti_sht_stm によってチャネルをクローズするとチャネルは送信終了状態になる．この状態で，受信するデータがなくなるとスレッドにおいて InputStream.read は −1 を返し，チャネルは切断状態になる．一方，スレッド側から先に Inputstream.close を呼び出すと，チャネルは強制クローズされたものとし，強制切断状態になる．この状態で，タスクで jti_wri_stm，または jti_sht_stm を実行すると E_CLS が返され，チャネルの強制切断状態が検出される．この後チャネルは切断状態になる．(2) スレッドからタスクへストリームを送信する場合：(1) と同様にして，ストリームを生成しオープンし，チャネルを接続状態にする．接続状態になるとスレッドから OutputStream.write メソッドを使ってデータを送信し，タスクは jti_rea_stm を使って受信する．送信を中止するには，スレッドから OutputStream.close メソッドを使ってチャネルをクローズし，送信終了状態にする．この後，タスクにおいてチャネルが送信終了状態になったことを意味する jti_rea_stm の返却値が 0 を返すと，チャネルは切断状態になる．この場合，タスクからチャネルを強制クローズする手段は提供していない．これはタスクは常時動作してデータの取得やコマンド送出を行っているため，タスクからチャネルを強制クローズすることはないと考えたからである．

4.5 方式の比較

本節では，上述した 3 つの協調動作方式について仕様の比較を行う．**図 11**，**図 12** にアタッチクラス，共有オブジェクトの例を示す．これは，6 章の工業用コンピュータでの応用例を簡単化したもので，sensor_task タスクで温度データを取得，main スレッドに送信し，main スレッドで受信，ユーザに表示するものである．

アタッチクラスの長所は，リアルタイム OS の機能を Java プログラムから直接利用できる点にある．特に，リアルタイム OS 上で動作するデバイスドライバがある場合に，Java プログラムからデバイスドライバを制御する場合に有効である．デバイスドライバ中のタスクの生成や割込みハンドラ制御をアタッチクラスを使って行い，デバイスドライバとのデータやコマンドの受け渡しは，同じくアタッチクラスのタスク間通信機能を利用するというものである．たとえば，図 11 では，Task，Mbx，ItronMemory のアタッチクラスを使って，sensor_task を制御，温度データを受信して

```
public class TempDisplay {
  public static void main(String argv[]) {
    Task  tsk; MailBox  mbx; ItronMemory  memory;
    long  time; long  temperature;
    tsk = new Task(tid);   //生成済のセンサタスクにアタッチする
    mbx = new MailBox(mid);//生成済のメールボックスにアタッチする
    while(true) {
      memory = mbx.receive(8);  //メールボックスからデータを受信する
      temperature = memory.readUW();//温度データを読込む
      time = memory.readUW();       //時刻データを読込む
      display.showTemperature(time, temperature);//表示する.
    }
  }
}
```

(a) Java プログラム

```
struct temperature {
  long temp, time;  /* 温度データとその取得時刻 */
};
sensor_task()
{
  struct temperature *tp;
  cre_mbx(mid,..);
  tp = malloc(sizeof(struct temperature));
   while(1){
    tp->time = get_time();       /*時刻データを取得パケットに格納*/
    tp->temp = get_temperature();/*温度データを取得パケットに格納*/
    snd_mbx(mid,tp);/*メールボックスを通して Java プログラムに送信*/
    slp_tsk(100);    /* 100 ミリ秒停止*/
  }
}
```

(b) リアルタイムタスクプログラム

図 11 アタッチクラスによるスレッド・タスク間通信の例
Fig. 11 A usage example of attach class.

いる（図 11 (a)）．このとき，Java プログラムではリアルタイム OS の資源を直接利用するため，プログラミングにあたってリアルタイム OS の知識を必要とする．また，アタッチクラスではリアルタイム OS の機能をそのまま利用しているため，リアルタイム OS の仕様が異なるとアタッチクラスのライブラリの仕様も異なる．このため，異なる仕様のリアルタイム OS を下位 OS とするアタッチクラスを利用した Java プログラムでは変更が必要になる．一方，タスク側からはスレッドと協調動作を行うにあたって特別な改造は必要ない．たとえば，図 11 (b) に示すように表示機能に取得したデータをメッセージボックス機能を使って送信すればよい．すなわち，既存のリアルタイム OS 上のソフトウェア資産は変更なく再利用できる．

共有オブジェクト機能，ストリーム通信機能ともに，Java プログラムから見て，アタッチクラスのように下位 OS に依存するということがなく，これらの機能を利用した Java プログラムの可搬性は高い． 図 12 (a) の例に示すように Java プログラムではリアルタイム OS に依存するような機能は隠蔽されている．また，ストリーム通信機能の長所としては，タスク・スレッド間で可変長データの送受信を Java ライブラリのストリーム通信を利用して行う点にあり，プログラマは

```
class Temperature extends SharedObject {
  int time; int temperature;
  public Temperature(String name) { ... }
  public Temperature(Sharable shm, String name) { ... }
};
public class TempDisplay {
  public static void main(String argv[]) {
    Temperature         temp;             //温度オブジェクトを
    temp = new Temperature("temperature");//temperature という名前
                                          //で作成
    while(true) {
      temp.lock();                    //温度オブジェクトのロック
      display.showTemperature(temp); //温度データの取得，表示
      temp.unlock();                  //温度オブジェクトのアンロック
      Thread.sleep(500);              //500 ミリ秒寝る
    }
  }
}
```

(a) Java プログラム

```
sensor_task()
{
  JNO shmid; /*共有オブジェクトの識別子 */
  jti_get_obj("temperature",&shmid);
   while(1){
    jti_loc_obj(shmid,0);
    jti_set_int(shmid,"Temperature","time",get_time());
    jti_set_int(shmid,"Temperature","temperature",get_temperature());
    /* Temperature という名の共有オブジェクトに時間と温度を設定 */
    jti_unl_obj(shmid);
    slp_tsk(100);    /* 100 ミリ秒停止*/
  }
}
```

(b) リアルタイムタスクプログラム

図 12 共有オブジェクトによるスレッド・タスク間通信の例
Fig. 12 A usage example of shared object.

新たなプログラミングモデルの知識を必要としない点があげられる．ただし，こうした機能を利用する場合にリアルタイムタスク側にはスレッドとの通信のための改造が必要となる．しかし，スレッドと通信する仲介タスクを置くことによりこの改造は局所化できると考えられる． 共有オブジェクト機能が有効であるのは，スレッドとタスクで非同期にデータを送受信を行う場合である．たとえば，図 12 にあるように，タスクは周期的に温度データを取得，共有オブジェクト内に設定し，スレッド側で共有オブジェクトから適宜温度データを取得し，表示するというものが考えられる．一方，ストリーム通信機能が有効となるのは，スレッドとタスクでやりとりするデータが同期をとる必要がある場合であり，たとえば，アプリケーションスレッドからデバイスドライバタスクに対してコマンドを送信するなどが考えられる．

5. 実装と評価

5.1 実　　装

第 1 著者のグループでは，JTRON 仕様に基づいた協調動作ライブラリを NEC 製の産業用ネットワークコンピュータ NETJACS [17),18)] 上に試作し評価を行っ

表 1 実装プラットフォーム仕様
Table 1 Implementation platform specification.

分類	仕様
CPU	Vr4300（128 MHz）
DRAM/SRAM/FROM	32 MB/256 KB/4 MB
通信 IO	10BASE/T,PC104 × 2
OS	μITRON3.0 仕様
ミドルウェア	ウィンドウシステム，TCP/IP，ファイルシステム，JDK1.1.7b

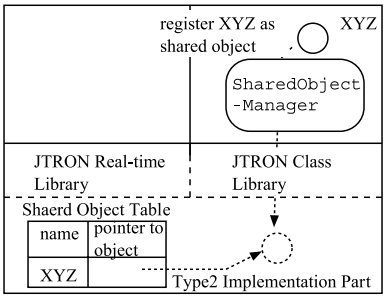

図 13 共有オブジェクト機能の実装
Fig. 13 Implementation of shared object implementation.

た[19]．その実装プラットフォーム仕様を**表 1** に示す．共有オブジェクト，ストリーム通信ともに JTRON リアルタイムライブラリの API は，OS の一部として実装するのではなく，μITRON3 仕様のリアルタイム OS 上に実装した．

(1) アタッチクラス（タイプ 1）
アタッチクラスのメソッドをネイティブメソッドで実装し，その中で ITRON 仕様 OS の API を呼び出すことにより実現できる．

(2) 共有オブジェクト機能（タイプ 2）
共有オブジェクトを実装する Java クラスを図 5 の破線内に示す．クラス `SharedObjectManager` は共有オブジェクトマネジャを実装し，タスクと通信を行い，共有オブジェクトを管理する（**図 13** 参照）．共有オブジェクトマネジャに，共有されるオブジェクトが登録されると，このオブジェクトを Type2 Implementation Part に渡す．Type2 Implementation Part はタスクからのアクセスの手段やロック機構を実装する．ここでは 2 つの実装手段で実現した．

実装 1： JNI では，C プログラムから Java プログラム内部の実装を隠蔽し，API によりアクセスする手段を提供している[16]．共有対象のオブジェクトが共有オブジェクトマネジャから登録されると，そのオブジェクトを Type2 Implementation Part に渡す．このときこのオブジェクトが他のスレッドからもアクセスできるように，JNI の `NewGlobalRef` を使ってそのオブジェクトのグローバル参照を生成する．Type2 Implementation Part はオブジェクト名とそのオブジェクトへのグローバル参照を管理する表（Shared Object Table と呼ぶ）を作成する．タスクからロック操作後，アクセスし，その後アンロック操作を行う．このため，ロック操作（`jti_loc_obj`）実行時に，実際のロック操作と同時に，JNI の `AttachCurrentThread` を実行して，タスクを Java 実行環境中の VM に結び付け，以降のインスタンス変数アクセスを可能とするための情報を JNI 環境変数として取得する．`jti_get_<type>` や `jti_set_<type>` の API によってインスタンス変数にアクセスするとき，先に取得した JNI 環境変数と対象オブジェクトへのポインタを使って，JNI の `GetFieldID`，`Get<type>Field`，あるいは `Set<type>Field` を順次呼び出して，インスタンス変数へのデータの取得・設定を行う．アンロック操作（`jti_unl_obj`）実行時に，JNI の `DetachCurrentThread` により，タスクと VM の結び付きを解放する．

実装 2： 何らかの手段で Java プログラム内部の実装方法を利用することができるとこの情報を使った共有オブジェクト機能の実装が可能である．Java Developers Kit ではクラス定義から C ヘッダを生成する `javah` コマンドが提供されている．このコマンドを利用して，共有対象となるオブジェクトのクラスのインスタンス変数名とそのクラス内のオフセット値の組を求めることができる．この組を Type2 Implementation Part が読み込み，クラスごとにインスタンス変数名とオフセット値の表（クラス表）を作成する．次に共有対象のオブジェクトが共有オブジェクトマネジャから登録されると，共有オブジェクトのポインタを求める．Type2 Implementation Part はオブジェクト名とそのオブジェクトへのポインタを管理する Shared Object Table を作成する．タスクから `jti_get_<type>` や `jti_set_<type>` の API によってアクセスするとき，Shared Object Table からアクセス対象へのポインタを求め，次にクラス表から対象インスタンス変数のオフセット値を求め，値の設定・参照を行う．タスクのアクセス中に共有オブジェクトは GC の対象となってはならない．このために，VM が提供している JNI の `GetPrimitiveArrayCritical` API 相当の機能を共有オブジェクト登録時に呼び出す必要がある[16]．
なお，両実装ともに共有オブジェクトに対するロック操作はセマフォを使って実現している．

リアルタイムライブラリのスレッド操作 API は，リアルタイムライブラリ中から JNI の `Call<type>Method` を使って，`Thread` クラス，`ThreadGroup` クラスのメ

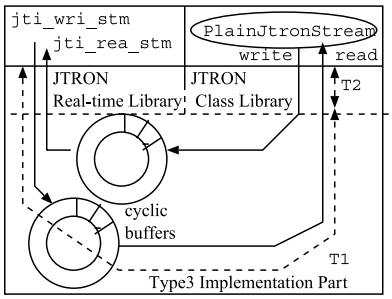

図 14 ストリーム通信機能の実装
Fig. 14 Implementation of stream communication.

ソッドを呼び出すことで実現している．

(3) ストリーム通信機能（タイプ 3）

ストリーム通信を実装する Java クラスを図 9 の破線内に示す．クラス `PlainJtronStream` は `JtronSreamImpl` を継承し実装したクラスであり，次に述べるタイプ 3 実装とのデータの受け渡しを行う（**図 14**）．`JtronInputStream` と `JtronOutputStream` はそれぞれ `InputStream` と `OutputStream` を継承したクラスであり，`PlainJtronStream` 中のメソッドを呼び出し，Type3 Implementation Part からデータの送受信を行う．チャネルはサイクリックバッファを使って実現している（図 14）．送信タスクあるいはスレッドから送信データをサイクリックバッファに書き込み，受信タスクあるいはスレッドがサイクリックバッファからデータを読み込み受信する．サイクリックバッファの一貫性を保つために，これへの書き込みと読み出しを排他制御する必要がある．さらに，Type3 Implementation Part はチャネルの読み/書きを待つタスクを指定するためにタスクの識別子を保持しており，これも排他制御の対象となる．これらの排他制御は ITRON 仕様 OS の 2 つのセマフォにより実現した．

NETJACS 上への JTRON 仕様は，JDK1.1.7 を用いて μITRON 仕様 OS 上に実装した．この実装に際して，Java 実行環境を改造することなく実現することができ，要求事項の **RM1** を達成したことを確かめることができた．

5.2 性能評価

共有オブジェクト機能の性能評価結果を示す．実装 1 と実装 2 でタスクからの実行時間が大きく異なることが予測されることから，タスクから共有オブジェクトへデータの設定時に要する際の API の実行時間を計測した．ここでの共有オブジェクトは `int` 型データ 1 つである．その結果を**表 2** に示す．JNI を用いた実装よりも `javah` コマンドを用いた実装の方が 1〜2 けた高速である．実装 1 のロック処理，アンロック処理に時間を要し

表 2 共有オブジェクトの性能比較
Table 2 Comparison of shared object performance.

API	実装 1	実装 2
ロック操作（`jti_loc_obj`）	3.91 ms	0.012 ms
データ設定（`jti_set_obj`）	0.29 ms	0.056 ms
アンロック操作（`jti_unl_obj`）	2.83 ms	0.027 ms

表 3 ストリームの性能比較
Table 3 Comparison of stream performance.

実装		性能値
(ST1) タスク → スレッド		0.62 ms
	T_1	0.46 ms
	T_2	0.16 ms
(ST2) タスク → タスク（コピー 2 回）		0.12 ms

ているのは，それぞれ JNI の `AttachCurrentThread`, `DetachCurrentThread` 呼び出しが含まれているためであり，JNI の利用時の実装のオーバヘッドが大きいことが分かる．これから，共有オブジェクト機能インタフェースは，Java オブジェクトの実装情報などが利用できるような場合に実現環境に適した効率良い実装を可能するインタフェースであると考えられる．

次にストリーム通信の性能評価を行うために，タスク・スレッド間のストリーム通信速度を測定した．図 14 に測定点を示す．T_1 は送信元のタスクから送信先のスレッドに対応したタスクまでサイクリックバッファを介したデータ送信にかかる時間，T_2 はスレッドに対応したタスクがデータを受信した後，スレッドレベルまでに要する時間である．さらに比較のために，タスク間で送信側タスクからバッファ，バッファから受信タスクの単純に 2 回のコピーを行った場合に要する時間（ST2）を測定した．ストリームで送受信されるデータ長は 1,000 バイトであり，100 回送受信を行い，平均値を求めた．その結果を**表 3** に示す．T_2 は Java プログラムのネイティブメソッド実装時に必要とされる時間である．T_1 と ST2 の差は，Type3 Implementation Part で排他制御が必要な 2 つのセマフォを送信側，受信側で各 2 回ずつ，計 4 回の操作を行ったために発生したオーバヘッドである．

6. 応用事例

本章では，実システムでの応用例を述べ，これから得られた知見を述べる．

JTRON 仕様は，アプリックス社の JBlend [20]，パーソナルメディア社の J-right/V [21]，富士通（株）の J-REALOS [22] で商用化されている．また，情報処理振興事業協会が公募した “次世代デジタル応用基盤技術開発事業” における “情報家電のための分散ソフトウェ

アプラットフォームの構築” プロジェクトの一部として開発された[23]．

JBlend は JTRON 仕様のアタッチクラス機能を実装しており，情報家電向けのアプリケーションプラットフォームとして数多くの製品で利用されている[20]．デジタル TV など情報家電向けのアプリケーションプラットフォームとしては，描画インタフェース，ビットマップフォント組込み用インタフェース，ポインティングデバイスドライバインタフェース，キーボードドライバインタフェース，ネットワークインタフェース，ファイルシステムインタフェースなどリアルタイムタスクプログラムによるデバイスドライバを利用するインタフェースを JTRON 仕様のアタッチクラス機能を利用して提供している．このようなデバイスドライバやターゲットマシンに依存した処理を JTRON 仕様で統一することによって，これらを使った Java 言語による情報家電のアプリケーションプログラムの生産性が高まり，製品の開発期間短縮に寄与した．

次に，第 5 著者のグループで JTRON 仕様を搭載した工業用コンピュータ（NETJACS）での利用例を述べる．NETJACS では，図 1 に示したソフトウェア構成において，アタッチクラス中のメールボックス機能とメモリ操作クラスを実装した．タスクとスレッド間で利用したメールボックス機能は，送信するデータをポインタで渡すものである．NETJACS の JTRON 実装部は，電力監視システムやレーザ加工機の制御用 HMI 装置を監視，制御するソフトウェア基盤として使用された．電力監視システムでは，NETJACS 1 台に対して数百台の電力メータから上がってくるパルスをカウントする．このパルスカウントのために，30 ms ごとに接点をポーリングするという処理をタスク側で行い，カウントの結果をサーバに FTP で転送という，ネットワークの処理を Java アプリケーションで構築した．本例では，ネットワーク処理を含むサーバとの連携の部分で Java プログラムにより構築できたのが大きなメリットであった．レーザ加工機では，2，3 秒間の加工時間におけるレーザ強度をグラフ表示する処理を行った．具体的には，10 ms 程度の周期で強度データをポーリングし，これをタスク側で保持して加工終了後にスレッド側に転送し，グラフとして表示した．データ収集時に正確な時間の刻みが必須の処理であるので，この処理をタスク側で行った．以上からタスクによるリアルタイム処理と Java プログラムによる GUI やインターネットアクセスなどの処理を共存させた実用的なシステムの構築を行うことができた．これから JTRON 仕様に基づくハイブリッドアーキテクチャのシステムは十分実用に供することができるということができる．

7. おわりに

本論文では，ハイブリッドアーキテクチャに Java 実行環境を改造せずにタスクとスレッドの共存を可能にした実行環境 JTRON2.1 の仕様検討を行い，スレッドとタスク間で協調動作をさせるための API 仕様を規定し，その試作とその評価結果，応用事例を述べ，JTRON 仕様の有効性を示した．

本アーキテクチャの長所としては，タスクとスレッドの協調動作機能によりタスクによるリアルタイム処理とスレッドによる GUI 機能や動的にロードした機能との結合が容易になるという点があげられる．

本論文で提案したハイブリッドアーキテクチャは ITRON 仕様以外のリアルタイム OS や Java 実行環境以外の OS にも適用可能である．他のリアルタイム OS へ適用には，アタッチクラスにおいてクラスを OS 資源に，クラスのメソッドを OS 資源に対するシステムコールに対応させることになる．アタッチクラスのクラス名，メソッド名と共有オブジェクト，ストリーム通信のリアルタイム OS ライブラリの API 名は，利用されるリアルタイム OS の名前規則に従って変更になる． また，Java 実行環境以外の他の OS へは，たとえば，ITRON 仕様 OS と Linux を統合させた “Linux on ITRON” [24] に適用されている．

一方，JTRON 仕様では，ハイブリッドアーキテクチャで実現する，タスクとスレッドに対する実行優先度の割当てやメモリ量の割当てなど，資源割当て手法については言及していない．今後の課題として静的にハイブリッドアーキテクチャ間で共有する資源の割当て手法，あるいは動的に資源割当てを変更する手法の研究[25] があげられる．

謝辞 JTRON2.1 仕様を検討していただいたトロン協会 ITRON 部会プロジェクト JTRON WG のメンバ，特に斉藤栄治氏（富士通デバイス（株）），丹崎剛介氏（（株）アプリックス），臼井和敏氏（NEC）に感謝します．

参考文献

1) Sun Microsystems: Java 2 Platform Micro Edition (J2ME). http://java.sun.com/j2me
2) Bollella, G., Brosgol, B., Dibble, P., Furr, S., Gosling, J., Hardin, D. and Turnbull, M.: *The Real Time Specification for Java*, Addison-Wesley (2000).
3) Consortium, J.: Core Real-Time Extensions

for the Java Platform: Draft Straw Man Position (1999). http://www.j-consortium.org/
4) トロン協会 ITRON 部会：JTRON2.1 仕様 (2000). http://www.assoc.tron.org/spec/jtron/jtron-201j.pdf
5) Nakamoto, Y. and Hachiya, S.: JTRON: An Object-oriented Real-time System based on a Hybrid Architecture, *Proc. 4th International Symposium on Object-oriented Real-time distributed Computing*, pp.243–250 (2001).
6) Silberschatz, A., Galvin, P. and Gagne, G.: *Operating System Concepts*, Addison-Wesley (2002).
7) 岡崎世雄，全先 実：VM，共立出版 (1989).
8) Butenhof, D.: *Programming with POSIX Threads*, pp.189–195, Addison-Wesley (1997).
9) Grzelakowski, M., Cambell, J and Dubnan, M.: DMERT Operating System, *Bell System Technical Journal*, Vol.62, No.1, pp.303–323 (1983).
10) FSMLabs: RTLinux. http://www.fsmlabs.com
11) Bianchi, E., Dozio, L. and Mantegazza, P.: A Hard Real Time Support for Linux, DIAPM RTAI. http://www.aero.polimi.it/~rtai/
12) http://www.tenasys.com/
13) http://www.vci.com/
14) Hachiya, S.: Java Use in Mobile Information Devices: Introducing JTRON, *IEEE Micro*, Vol.21, No.4, pp.16–21 (2001).
15) Gamma, E., Helm, R. Johnson, R., Vlissides, J. and Booch, G.: *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley (1995).
16) Liang, S.: *The Java Native Interface Programmer's Guide and Specification*, Addison-Wesley (1999).
17) 中野 徹，山口昭彦，江藤康孝，清水直良，工藤毅史：産業用ネットワークコンピュータ NETJACS，NEC 技報，Vol.51, No.11 (1998).
18) 富森博幸，臼井和敏，則本英保，千嶋 博，高木淳司，菊地康之：組込み向け Java 実行環境 JEANS，NEC 技報，Vol.51, No.11 (1998).
19) 朝倉義晴，臼井和敏，中本幸一：ITRON 仕様 OS と Java の協調機構 JTRON2 の実装について，電子情報通信学会技術報告，CPSY98-169 (1999).
20) http://www.jblend.com/
21) http://www.personal-media.co.jp/
22) http://edevice.fujitsu.com/jp/
23) http://www.ertl.jp/ITRON/JCG/
24) Takada, H., Iiyama, S., Kindaichi, T. and Hachiya, S.: Linux on ITRON: A Hybird Operating System Architecture for Embedded Systems, *Proc. 2002 Symposium on Application and the Internet*, pp.4–7 (2002).
25) Mok, A., Feng, X. and Deji C.: Resource partition for real-time systems, *Proc. 7th Real-Time Technology and Applications Symposium*, pp.75–84 (2001).

(平成 15 年 7 月 25 日受付)
(平成 15 年 11 月 15 日採録)

中本 幸一（正会員）

1982 年大阪大学大学院基礎工学研究科博士前期課程修了．同年 NEC 入社．現在システムプラットフォーム研究所研究部長．組込みソフトウェア，モバイルシステムソフトウェアの研究開発に従事．1990 年～1991 年 Cornell 大学計算機科学科客員研究員．1997 年大阪大学大学院情報数理系専攻博士課程入学．2000 年単位取得退学．博士（工学）．2003 年より電気通信大学客員教授．電子情報通信学会，日本ソフトウェア科学会，IEEE Computer Society 各会員．

高田 広章（正会員）

名古屋大学大学院情報科学研究科情報システム学専攻教授．1988 年東京大学大学院理学系研究科情報科学専攻修士課程修了．同学科の助手，豊橋技術科学大学情報工学系助教授等を経て，2003 年 4 月より現職．リアルタイム OS，リアルタイムスケジューリング理論，組込みシステム開発技術等の研究に従事．ITRON 仕様の標準化活動に，中心的メンバとして参加．博士（理学）．IEEE，ACM，電子情報通信学会，日本ソフトウェア科学会各会員．

八谷 祥一

株式会社ガイア・システム・ソリューション技術企画担当副社長．1986 年法政大学経営学部経営学科中退，1987 年株式会社アスキー，1992 年株式会社アプリックスを経て，2003 年より現職．光ディスク論理フォーマット，組込みシステムの研究開発および標準化活動に従事．1999 年 ISO/IEC JTC1/SC23 DIS 9660 プロジェクトエディタ．IEEE Computer Society 会員．

朝倉　義晴

1998 年大阪大学大学院基礎工学研究科情報数理系専攻前期課程修了．同年 NEC 入社．現在システムプラットフォーム研究所所属．組込みソフトウェアの研究開発に従事．電子情報通信学会会員．

樫宿　昌房

1987 年東京理科大学理工学部情報科学科卒業．同年 NEC 入社．現在社会情報ソリューション事業部第三システム部エキスパート．ファクトリ・コンピュータのソフトウェア開発に従事．