情報処理学会論文誌　プログラミング　Vol. 2　No. 2　37–53 (Mar. 2009)

メモリ共有を考慮したRPCシステム

安積卓也[†1,†2]　大山博司[†3]　高田広章[†1]

本論文では，組み込みシステムの様々な環境でソフトウェアを再利用することを目的とし，メモリを共有するシステムに対しても RPC 技術を適用可能なシステムを提案する．呼び出し形態は，(1) 同タスク同プロセッサ，(2) 別タスク同プロセッサ，(3) メモリを共有する別プロセッサ，(4) メモリを共有しない別プロセッサ（ネットワーク接続など）の 4 つに分類される．特に (2) と (3) では汎用システムとは異なり，メモリの種類・用途を考慮する必要があり，想定される通信チャネルも多様である．従来 RPC システムの主眼は (4) にあるため，メモリ共有が考慮されていない．したがって (2) と (3) の呼び出し形態に対して，既存の RPC システムをそのまま適用することは容易ではない．今回我々は，ソフトウェアの再利用性をさらに高めるために，組み込みシステム向けのコンポーネントシステム上で RPC を実現する．さらに，メモリ領域の確保・解放を明記するためのインタフェース記述を提案する．従来インタフェース記述と比較することで，提案インタフェース記述の有用性を示す．

RPC System with Memory Sharing Consideration

Takuya Azumi,[†1,†2] Hiroshi Oyama[†3]
and Hiroaki Takada[†1]

In order to reuse software in a variety of environments of embedded systems, RPC technology is also applied to memory sharing systems. There are four types of procedure call as follows: (1) Same task on same processor, (2) Different tasks on same processor, (3) Different processors with a shared memory, (4) Different processors without a shared memory (network connection, etc.). Especially in (2) and (3), unlike general-purpose systems, it is necessary to consider the types and usage of a memory. Since the main purpose of existing RPC system is (4), shared memories are not considered. In addition, there are a variety of RPC channels. Therefore, it is not easy to apply existing RPC system to (2) and (3). We realize RPC on an embedded component system for enhancing reusability of software. Moreover, interface descriptions to represent memory allocation and deallocation are proposed. The effectiveness of our proposed interface descriptions is shown by comparing with that of an existing one.

1. はじめに

近年，組み込みシステムの複雑化・大規模化にともないソフトウェア生産性が問題となっている．ソフトウェアの生産性の向上を目的として，組み込み分野においても，コンポーネントベースの開発[9),11)]が注目されており，様々なプロジェクト[1),2),8),14),19),21)]が報告されている．しかし，これらは，メモリ共有を考慮できていないためマルチプロセッサへの対応が十分ではない．これまでに，我々は組み込みシステムに適したオーバヘッドの小さいコンポーネントシステムとして TOPPERS[20)] Embedded Component System（TECS）[3)–5),24)]を提案してきた．そこで，本論文では，組み込みシステムの様々な環境でソフトウェアを再利用することを目的とし，メモリを共有するシステムに対しても RPC 技術を適用可能なコンポーネントシステムを提案する．

呼び出し形態は，(1) 同タスク同プロセッサ，(2) 別タスク同プロセッサ，(3) メモリを共有する別プロセッサ，(4) メモリを共有しない別プロセッサ（ネットワーク接続など）の 4 つに分類される．特に (2) と (3) では汎用システムとは異なり，メモリの種類・用途を考慮する必要がある．しかし，従来 RPC システムの主眼は (4) にあるため，メモリ共有が考慮されていない．したがって (2) と (3) の呼び出し形態に対して，既存の RPC システムをそのまま適用することは容易ではない．さらに，組み込みシステムでは想定される通信チャネルも多様である．

本論文の主な貢献は次の 2 点である．

(A)　呼び出しをすべての形態で効率的に実現できる．

(B)　組み込みシステムにおける多種の通信チャネル・メモリアロケータに対応できる．

(A) はメモリ領域の確保・解放を明記するためのインタフェース記述によって実現される．提案するインタフェース記述によりメモリ領域の所有権を明示できる．つまり，(2) と (3) において呼び側と受け側で共通のメモリアロケータを使用でき，非同期呼び出しへの対応や並列処理も可能になる．

†1 名古屋大学大学院情報科学研究科
Graduate School of Information Science, Nagoya University
†2 日本学術振興会特別研究員
Research fellow of the Japan Society for the Promotion of Science
†3 オークマ株式会社
OKUMA Corporation

(B) は RPC チャネル・メモリアロケータをコンポーネントレベルで選択することで実現される．既存のシステムでは RPC チャネルはフレームワークの中に隠れてしまうため，自由に RPC チャネルを選択することはできない．本システムでは RPC チャネルをコンポーネントの 1 つとして扱うことにより，適切な RPC チャネルを選択できる．同様に，適切なメモリアロケータを選択できる．

本論文の構成は以下のとおりである．まず，2 章で本システムのベースとなる TECS について述べる．次に，3 章で提案するインタフェース記述，メモリアロケータの扱い方について，4 章で提案する RPC システムについて述べる．5 章で関連研究について述べ，最後に 6 章で本論文をまとめる．

2. 組み込みシステム向けコンポーネントシステム（TECS）

本章では，本システムのベースとなる TECS の概要を述べる．

2.1 TECS の特徴

TECS の最大の特徴は，コンポーネント化にともなうオーバヘッドが小さいことである．TECS では，静的なコンポーネントモデルを採用しており，インスタンス化や結合にかかる実行時オーバヘッドをなくすことができる．

コンポーネント化にともなうオーバヘッドが小さいことにより，粒度の小さなソフトウェアモジュールや資源まで，コンポーネントとして扱うことができる．コンポーネントの粒度が小さいことは，システムの細部にわたる最適化がコンポーネントのレベルで行えることを意味し，システムの目的に合わせて細部まで最適化する必要がある組み込みシステムには重要な利点となる．

コンポーネントの粒度を小さくすることにより，コンポーネントモデルが複雑になることが懸念されるが，TECS では，複合コンポーネントの機能[5] を提供することで，この問題を解決している．複合コンポーネントとは，複数のコンポーネントを組み合わせたものを，1 つのコンポーネントとして扱える仕組みのことである．複合コンポーネントの具体例は，3.3 節で紹介する．

2.2 コンポーネントモデル

TECS では，インスタンス化されたコンポーネントをセルと呼ぶ[⋆1]．セルは，自身の機能を提供するインタフェースである受け口，他のセルの機能を利用するためのインタフェースである呼び口，セルの付随情報（定数）を表す属性，セルの内部状態を表す変数で構成される．受け口は，セルの機能を提供する関数の集合であり，呼び口は，他のセルの機能を利用するための関数呼び出しの集合である．1 つのセルは，複数の呼び口や受け口を持つことができる．

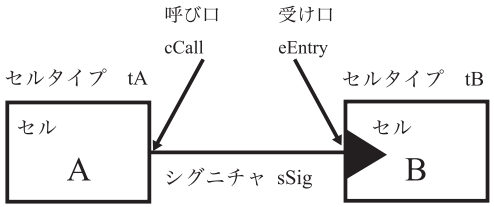


図 1　コンポーネントモデル
Fig. 1　Component model.

受け口と呼び口の型は，シグニチャと呼ばれる関数ヘッダの集合で定義される．セルの呼び口は，同一のシグニチャを持つ他のセルの受け口と結合できる．これにより，前者のセルから後者のセルの関数群を呼び出すことが可能になる．**図 1** は，セル A の呼び口 cCall とセル B の受け口 eEntry を結合したことを表している．

セルの型，すなわち，セルが持つ受け口，呼び口，属性，変数の組を，セルタイプと呼ぶ．1 つのセルタイプから，複数のセルをインスタンス化することができる．1 つのセルのみをインスタンス化できるセルタイプを，シングルトンと呼ぶ．

呼び口は，必ず 1 つの受け口と結合しなければならない．ある呼び側のセルから複数のセルを呼び出し分ける場合のように，複数の受け口と結合したい場合には，呼び口を配列にして用いる．これを呼び口配列と呼ぶ．呼び口配列の具体例は，4.4 節で紹介する．

それに対して受け口は，任意個の呼び口と結合することができる．ただし，複数の呼び口と結合した受け口は，どの呼び口から呼び出されたかを一意に判別することができない．これを判別したい場合には，受け口を配列にした受け口配列を用いる．受け口配列の用途としては，呼び出したセルを識別して，そのセルの受け口をコールバックする場合などが考えられる．

2.3 コンポーネント記述

コンポーネントの要素は，TECS の仕様で定義されたコンポーネント記述を用いて表される．コンポーネント記述は，シグニチャ記述，セルタイプ記述，組み上げ記述に分類され

⋆1 オブジェクト指向の用語との対比では，コンポーネントがオブジェクトに，後述のセルタイプがクラスに，セルがインスタンスに対応する．

```
1:  /* シグニチャsSig の定義 */
2:  signature sSig {
3:    ER func1([in] int32_t data, [out] int32_t *result);
4:    ER func2([in, size_is(size)] const char_t *buf,
               [in]            int32_t size,
               [inout]         int32_t *result);
5:  }
```

図 2　シグニチャ記述
Fig. 2　Signature description.

る．下記で図 1 を例として，それぞれのコンポーネント記述を説明する．

2.3.1　シグニチャ記述

シグニチャ記述は，セルのインタフェースを定義するために用いられる．signature キーワードに続けてシグニチャ名（通例，接頭辞 “s” をつける）を記述し，そのシグニチャが持つ関数ヘッダを中括弧内に列挙する．**図 2** の例では，関数 func1 と func2 を持つシグニチャsSig を定義している．

TECS では，インタフェースの定義を明確にするために，図 2 の 4 行目の func2 で見られるような，引数の指定子を用意している．指定子は引数が，in が入力，out が出力，inout が入出力，size_is(size) が大きさ size の配列であることをそれぞれ示す．

2.3.2　セルタイプ記述

セルタイプ記述は，呼び口，受け口，属性，変数を用いてセルタイプを定義するのに用いられる．celltype キーワードに続けてセルタイプ名（通例，接頭辞 “t” をつける）を記述し，セルタイプが持つ要素を列挙する．呼び口は，call キーワード，シグニチャ名，呼び口名（通例，接頭辞 “c” をつける）の順に記述する．受け口は，entry キーワード，シグニチャ名，受け口名（通例，接頭辞 “e” をつける）の順に記述する．属性は attribute，変数は var キーワードを用いてそれぞれ列挙する．属性，変数が存在しない場合は省略することができる．

図 3 の例では，シグニチャが sSig の呼び口 cCall を持つセルタイプ tA と，シグニチャが sSig の受け口 eEntry，属性，変数を持つセルタイプ tB を定義している．

2.3.3　組み上げ記述

組み上げ記述は，セルタイプをインスタンス化してセルを生成し，セルどうしの結合関係を定義してアプリケーションを構築するために用いられる．cell キーワードに続けてセルタイプ名，セル名を記述し，中括弧内には，自身の呼び口と他のセルの受け口との結合を列挙する．結合は，呼び口名，=，結合先の受け口の順に記述する．なお，呼び口のないセルには結合の記述は必要ない．**図 4** の例では，セルタイプ tB のセル B とセルタイプ tA のセル A が，それぞれインスタンス化されている．またセル A の呼び口 cCall は，セル B の受け口 eEntry と結合している．

```
 1:  /* セルタイプ tA の定義 */
 2:  celltype tA {
 3:    /* シグニチャsSig の呼び口の定義 */
 4:    call  sSig cCall;
 5:  };
 6:  /* セルタイプ tB の定義 */
 7:  celltype tB{
 8:    /* シグニチャsSig の受け口の定義 */
 9:    entry sSig eEntry;
10:    attribute{
11:       /* 属性の定義 */
12:    };
13:    var{
14:       /* 変数の定義 */
15:    };
16:  };
```

図 3　セルタイプ記述
Fig. 3　Cell type description.

```
1:  /* セル B のインスタンス化 */
2:  cell tB B{
3:  };
4:  /* セル A のインスタンス化 */
5:  cell tA A {
6:    /* セル B の受け口と結合 */
7:    cCall = B.eEntry;
8:  };
```

図 4　組み上げ記述
Fig. 4　Build description.

2.4　ファクトリ記述

ファクトリ記述は，セルをインスタンス化するときに独自の出力をファイルに行う目的で使用する．たとえば，リアルタイム OS のオブジェクト（カーネルオブジェクトと呼ぶ）で

```
 1:  /* タスクのセルタイプの定義 */
 2:  celltype tTask {
 3:    entry sTask     eTask; /* タスク受け口 */
 4:    entry siTask    eiTask;/* 非タスクコンテキスト用タスク受け口 */
 5:    call  sTaskMain cMain; /* メイン関数呼び口 */
 6:    attribute {
 7:      ID            id = C_EXP( "TASKID_$id$" );     /* タスクの ID */
 8:      [omit]ATR     task_attribute = C_EXP("TA_ACT");/* タスクの属性 */
 9:      [omit]PRI     priority = 16;                   /* タスクの優先度 */
10:      [omit]SIZE    stack_size = 4096;               /* スタックサイズ */
11:    };
12:    /* ファクトリ記述 */
13:    factory{
14:      write( "tecsgen.cfg",
                "CRE_TSK(%s,{%s,&$cb$,tTask_start_task,%s,%s,NULL});",
                 id, task_attribute, priority, stack_size );
15:    };
16:  };
```

図 5　タスクのセルタイプ記述
Fig. 5　Cell type description of task.

あるタスク，セマフォ，データキュー，メモリプールなどの静的 API[*1]を出力したいときなどに用いる．ファクトリ記述には，各セルタイプに対して 1 度だけ出力する FACTORY（すべて大文字）と各セルに対して出力する factory（すべて小文字）の 2 種類がある．ファクトリ記述を用いた例として，**図 5** にタスクのセルタイプを示す．3 行目は他のタスクへ起動や休止などの機能を提供する受け口 eTask を宣言している．4 行目は非タスクコンテキスト（割込みなど）へ提供する受け口 eiTask 宣言している．5 行目でタスクセルが呼び出す呼び口を宣言している．6–11 行目ではセルタイプの属性の宣言・デフォルト値の設定を行っている．属性の値は組み上げ時にも設定することができ，組み上げ時に設定されなかった場合は，セルタイプ記述で設定された値が使用される．組み上げ時の属性の設定の具体例は 4.4 節で紹介する．7 行目のidは，セルタイプ名とセル名が合わさった文字列に展開される．7–8 行目の C_EXP 内の記述は，マクロ定義など C 言語が解釈できる文字列であることを示す．それらの記述はセルタイプ記述内では文字列として扱われる．TA_ACT は OS の起動時にタスクが起動することを示す．13–15 行目はファクトリ記述である．ファクトリ記述内では，任意のファイルに文字列を出力する write 関数を用いることができる．write 関数の第 1 引数は，出力先のファイル名を指定する文字列である．この例の場合，コンフィギュレーションファイル（拡張子.cfg）である tecsgen.cfg を指定している．第 2 引数以降は，第 1 引数で指定されたファイルに出力するフォーマット文字列を指定する，この文字列は C 言語の printf 文のフォーマット指定と同様に扱われる．

[*1] μITRON4.0 仕様では，静的 API と呼ばれる一種のスクリプトを，コンフィギュレーションファイルに記述することで，静的にタスクなどのオブジェクトを生成することが可能になる．

```
1:  CRE_TSK(TASKID_tTask_TaskA,{TA_ACT,&tTask_CB_tab[0],
            tTask_start_task,16,4096,NULL});
2:  CRE_TSK(TASKID_tTask_TaskB,{TA_ACT,&tTask_CB_tab[1],
            tTask_start_task,16,4096,NULL});
```

図 6　出力されるコンフィギュレーションファイル
Fig. 6　Generated configuration file.

次に，セル TaskA とセル TaskB をインスタンス化した場合に出力される静的 API の例を**図 6** に示す．CRE_TSK はタスクを生成する静的 API である．パラメータは順に，タスクの ID，タスクの属性，タスクの拡張情報，タスクの起動番地，優先度，スタックサイズ，スタックの先頭番地である．タスクの ID（TASKID_tTask_TaskA または TASKID_tTask_TaskB）は，上記で述べた TASKID_idを展開したものである．tTask_CB_tab[0] は図 5 の 14 行目のcbが展開されたもので，セルのコントロールブロックを示す．このコントロールブロックを参照することで，図 6 のようにセル TaskA とセル TaskB でタスクの起動番地が tTask_start_task で同じであっても，その呼び先でそれぞれのセルの結合先（図 5 の 5 行目の呼び口の結合先）の関数を呼ぶことができる．タスクの属性，優先度，スタックサイズはセルタイプ記述で定義したデフォルトの値を使用している．スタックの先頭番地は NULL の場合は自動割付けを意味する．コンフィギュレータと呼ばれるツールで静的 API を解釈し，タスクなどの ID の自動割付け結果を C 言語のヘッダファイルの形で，カーネルオブジェクトの初期化に必要なファイルを C 言語のソースファイルの形で生成する．つまり，ファクトリ記述を使用することにより，セルをインスタンス化するだけで，カーネルオブジェクトの生成を自動で行うことができる．この仕組みを 4 章で述べる RPC チャネルでも使用する．

2.5　開発の流れ

図 7 に，TECS における開発の流れを示す．TECS の開発者は，コンポーネント設計者，コンポーネント開発者，アプリケーション開発者，プラグイン開発者の 4 つに分けられる．

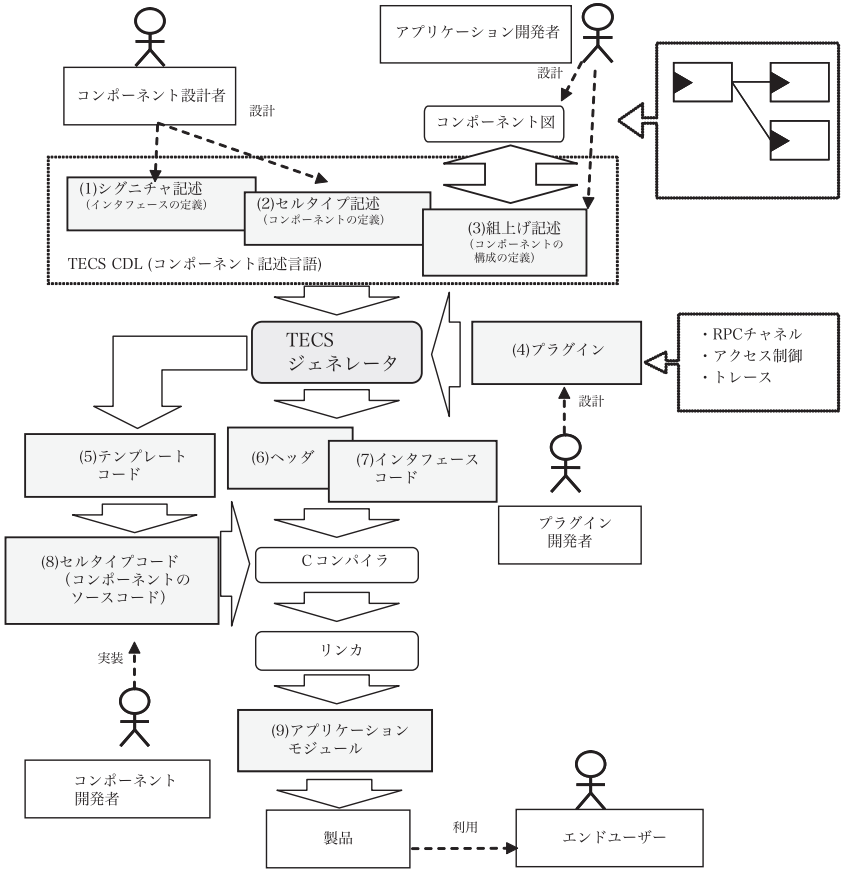


図 7　開発の流れ
Fig. 7　Development flow.

コンポーネント設計者は，シグニチャ記述（図 7 (1)）によりセル間のインタフェースを定義し，定義されたインタフェースを利用し，セルの呼び口，受け口，属性，変数をセルタイプ記述（図 7 (2)）に記述する．コンポーネント開発者は，シグニチャとセルタイプの情報から自動生成されたテンプレートコード（図 7 (5)）を利用し，コンポーネントの振舞いを C 言語で実装する（図 7 (8)）．テンプレートコードは，セルタイプ記述で定義された受け

```
 1:  /* セルタイプ tB の受け口 eEntry の関数 func2 のテンプレートコード */
 2:  ER eEntry_func2(CELLIDX idx, const char_t *buf,
                     int32_t size, int32_t *result)
 3:  {
 4:      ER      ercd = E_OK;
 5:      CELLCB  *p_cellcb;
 6:      if (VALID_IDX(idx)) {
 7:              p_cellcb = GET_CELLCB(idx);
 8:      }
 9:      else {
10:              return(E_ID);
11:      }
12:      /* ここに処理本体を記述 */
13:      return(ercd);
14:  }
```

図 8　テンプレートコードの例
Fig. 8　An example of template code.

口の関数（受け口関数と呼ぶ）に対してそれぞれ自動生成される．図 3 で定義したセルタイプ tB の受け口 eEntry の関数 func2 のテンプレートコードの例を**図 8** に示す．4–11 行目はエラー処理が記述されている．コンポーネント開発者は 12 行目以降にプログラムを記述する．受け口関数の実装では，呼び口の関数（呼び口関数と呼ぶ）の利用や属性・変数の参照など TECS の提供する機能を使うことができる．アプリケーション開発者（コンポーネント利用者）は定義されたセルタイプを利用し，組み上げ記述（図 7 (3)）により組み上げを行う．なお，GUI ツール[4] を利用してコンポーネント図から組み上げを行うこともできる．TECS では，コンポーネントどうしの結合部分にコンポーネントを挟み込む仕組みがある．たとえば，本論文における RPC チャネルの選択，トレース機能，アクセス制御[3] は，この仕組みを利用している．どのようなコンポーネントを挟み込むかをプラグイン形式で決めることができ，プラグイン開発者は，そのプラグイン（図 7 (4)）を定義する．この仕組みの詳細は，4.3 節で紹介する．

TECS ジェネレータの持つ機能は，上記で説明したテンプレートコードの生成，ファクトリ記述で出力されるファイルの生成，プラグイン読み込み以外に，組み上げ記述の情報から，セル間を結合するためのグルーコードを生成することがあげられる．グルーコードは，セルの呼び口関数のマクロなどを含むヘッダ（図 7 (6)），受け口側の関数テーブルなど含むインタフェースコード（図 7 (7)）から構成されている．TECS では静的なコンポーネント

モデルを採用しているため，構成によっては結合を最適化することもできる．ヘッダ，インタフェースコード，セルタイプコードをコンパイル・リンクして目的のソフトウェアであるアプリケーションモジュール（図 7 (9)）を生成する．

3. 提案するインタフェース記述

本章では，まず，1 章で説明した呼び出し形態 (2)，(3) に対応するためのインタフェース記述と，本システムにおけるメモリアロケータについて述べる．CORBA などのインタフェース記述で利用されている in（入力），out（出力），inout（入出力）指定子に加えて，呼び側セルから受け側セルに引数のデータとメモリ領域の所有権をともに渡す send，および，受け側セルから呼び側セルに引数のデータとメモリ領域の所有権をともに渡す receive を導入する．

3.1 in と send の違い

in は引数を入力として扱うこと示す指定子である．受け側の関数の実行中に，in で指定された引数のメモリ領域は受け側が利用する．呼び側に処理が戻ってきたら，呼び側は in で指定したメモリ領域を再利用でき，不要になればメモリ領域の解放を行う．

send は in と同様に呼び側から受け側にデータを渡す場合に使用される．**図 9** に in と send の違いを示す．in と send の違いは渡すデータを呼び側と受け側のどちら側でメモリ領域を解放するかという点である．in はメモリ領域の確保と解放のいずれも呼び側で行う．それに対し，send では確保は呼び側で行うが，解放は受け側で行う．

send は非同期呼び出しや並列処理を行いたいときに有効である．たとえば，呼び側が受け側の処理の終了を待つ必要がない場合が考えられる．このような呼び出しを oneway [⋆1] と呼ぶ．呼び出し形態 (2)，(3) で oneway 呼び出しを行った場合，呼び側で確保したメモリ領域を in により受け側に渡すには，RPC チャネル内でメモリ領域をコピーする必要がある．なぜなら，受け側の処理の終了を待たずに，処理が呼び側に戻ってくるため，in で渡したメモリ領域を受け側がまだ使用している間に，呼び側がその領域の内容を書き換えたり，その領域を解放したりする可能性があるからである．たとえば，in の引数のメモリ領域を C 言語のローカル変数として確保していた場合，呼び側の関数の終了時にメモリ領域が自動的に解放される．受け側がまだ使用していたとしても，次の関数が呼び出されたときに，解放されたメモリ領域は別目的で確保・利用される．send では，呼び側が渡したメモリ領域の書き換え・解放を行わないことを明示できる．send を使うことで，コピーをせずに効率的な呼び出しを実現する．

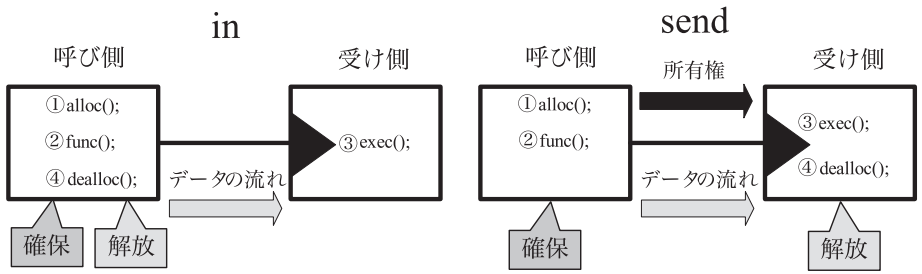


図 9　in と send の違い
Fig. 9　Difference between in and send.

また，書き換え・解放がないことをコンパイラなどで静的に検出することは容易ではない．なぜなら，上記で述べたとおり，呼び側の関数のローカル変数を利用した場合は，その領域は自動的に解放されるからである．さらに，グローバル変数を利用した場合にも，呼び側の関数内で書き込みがないことを検出できたとしても，他の関数で書き換えるかどうかまで検出することは難しい．コンパイラなどで自動的に書き換え・解放がないことを検出し，コピーをしない最適化を行うことは容易ではないため，提案記述である send を利用することで，開発者が明示的にコピーが必要ないことを示し効率的な呼び出しが実現できる．

そのほか，呼び側の書き換えや解放を禁止することも考えられるが，その実現には，呼び側は受け側の処理の終了を待つ必要がある．その場合は，非同期呼び出しや並列処理にはならない．

3.2 out と receive の違い

out は引数を出力として扱うこと示す指定子である．out が指定された場合，呼び側で確保したメモリ領域に対して，受け側がそのメモリ領域に書き込みを行い，呼び側がデータを受け取る．

receive は out と同様に呼び側が受け側からデータを受け取るときに使用される．**図 10** に out と receive の違いを示す．out と receive の違いは受け取るデータのメモリ領域をどちら側で確保するかという点である．out では呼び側で確保したメモリ領域に対して受け側がデータを書き込む．それに対して，receive は受け側でメモリ領域を確保する．メモリ領

⋆1 oneway という用語は CORBA から導入した．

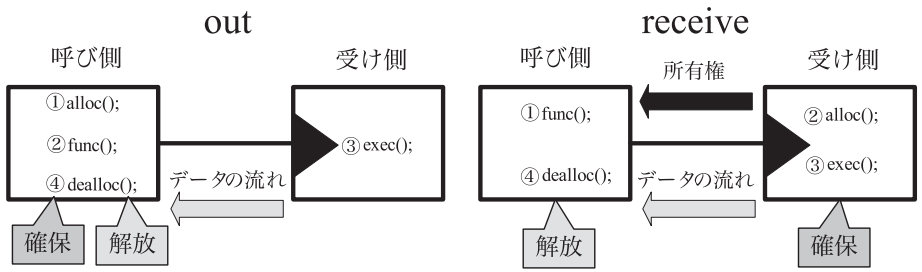


図 10　out と receive の違い
Fig. 10　Difference between out and receive.

域の解放は out，receive ともに呼び側で行う．

receive は呼び側が事前に大きさの分かっていないデータを受け取るときに有効である．out では呼び側が確保したメモリ領域に対して受け側がデータを書き込むため，呼び側が確保したメモリ領域以上のデータを書き込むことはできない．データサイズの最大値が分かっていれば，その大きさのメモリを確保すればよいが，使用されなかった領域は無駄になる．一方，receive では，受け側がメモリを確保するため，呼び側が大きさを知らなくても受け側が適切な大きさのメモリ領域を確保することができる．そのため，事前に大きさの分からないデータも受け取ることができる．

3.3　メモリアロケータ

本節では，本システムにおけるメモリアロケータについて述べる．1 章で述べたとおり，組み込みシステムでは多種のメモリの種類・メモリアロケータを考慮する必要がある．たとえば，デュアルポートメモリなど特別なメモリを利用することが考えられる．それに加えて，メモリを確保・解放するためのメモリアロケータの種類が多い．たとえば，タスクの優先度ごとやメモリサイズごとでメモリアロケータを使い分けることが考えられる．前者は，低優先度タスクが多くのメモリを確保することにより，高優先度タスクがメモリを確保できなくなることを防ぐためである．後者は，小さなメモリ確保・解放を繰り返すことによるメモリのフラグメントを避けるためである．そのうえ，malloc，リアルタイム OS の機能である固定長メモリプール・可変長メモリプールなど，メモリアロケータの実装も様々である．メモリアロケータをコンポーネントとして扱うことで，変更を容易にすることができる．

本システムにおけるメモリアロケータの扱い方をメモリアロケータに 3 つの固定長メモリプールを利用した例を通して説明する．**図 11** にメモリアロケータのセルタイプ記述を，

```
 1:  /* メモリアロケータの複合コンポーネントの定義 */
 2:  composite tAlloc{
 3:    entry sAlloc eAlloc;   /* メモリプールの受け口の定義 */
 4:    attribute{ /* 属性定義：サイズ，回数 */
 5:      uint32_t small_size = 8,   middle_size = 80,  large_size = 160;
 6:      uint32_t small_count = 10, middle_count = 10, large_count = 10;
 7:    };
 8:    /* 3 つの固定長メモリプールのインスタンス化 */
 9:    cell tMemoryPoolFixedSize MpfSmall{
10:      block_count = composite.small_count;
11:      block_size  = composite.small_size + 4;
12:    };
13:    cell tMemoryPoolFixedSize MpfMiddle{
14:      block_count = composite.middle_count;
15:      block_size  = composite.middle_size + 4;
16:    };
17:    cell tMemoryPoolFixedSize MpfLarge{
18:      block_count = composite.large_count;
19:      block_size  = composite.large_size + 4;
20:    };
21:    /* メモリアロケータの制御部分のインスタンス化 */
22:    cell tAllocMpf AllocMpf{
23:      cMpf[ SMALL ] = MpfSmall.eMemoryPoolFixedSize;
24:      cMpf[ MIDDLE] = MpfMiddle.eMemoryPoolFixedSize;
25:      cMpf[ LARGE ] = MpfLarge.eMemoryPoolFixedSize;
26:      small_size  = composite.small_size;
27:      middle_size = composite.middle_size;
28:      large_size  = composite.large_size;
29:    };
30:    composite.eAlloc => AllocMpf.eAlloc;   /* 受け口の委譲 */
31:  };
```

図 11　メモリアロケータの複合コンポーネント
Fig. 11　Composite component for memory allocator.

図 12 にそのコンポーネント図を示す．このメモリアロケータは，3 つの固定長メモリプールセルとそれらを制御するセルで構成されている複合コンポーネントである．複合コンポーネントは composite キーワード（図 11 の 2 行目）に続けてセルタイプ名を記述し，複合コンポーネントの内部にセルのインスタンス化や結合の記述をすることで複合コンポーネントの構成を定義する．メモリプールを制御するセルは，確保するメモリのサイズによって，

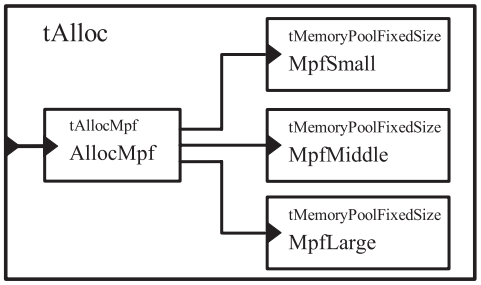


図 12　メモリアロケータのコンポーネント図
Fig. 12　An example of compoent model for memory allocator.

メモリ確保先のメモリプールを決める．そうすることで，サイズの小さなメモリ確保を繰り返すことで発生するメモリのフラグメントを避けることができる．

図 11 の 5 行目で各固定長メモリプールのサイズを，6 行目で確保できる回数を指定している．それぞれの値はデフォルト値であり，セルタイプ tAlloc をインスタンス化するときにサイズや回数を変更できる．

図 13 にメモリアロケータのシグニチャとメモリアロケータの制御部分のセルタイプを示す．メモリアロケータのシグニチャにはメモリ確保用の関数 alloc（3 行目）と解放用の関数 dealloc（4 行目）がある．7–18 行目にメモリアロケータ制御部分のセルタイプ定義を示す．**図 14** にメモリアロケータのメモリ確保用の実装コードを示す．まず，一番大きなメモリプールより大きなサイズが要求された場合には，メモリ不足エラー（E_NOMEM）を返す（8–10 行目）．C 言語の実装コードで，セルタイプ記述で宣言した属性を参照するには，8 行目のように属性マクロ（ATTR_属性名）を使用する．次に，要求されたサイズに合う添字を取得する（11–19 行目）．固定長メモリプールに結合した呼び口関数を呼び，メモリを確保する（20 行目）．確保したメモリの先頭番地にどの固定長メモリプールで確保したか（添字）を保存する（21 行目）．添字分のメモリ領域を確保するために，図 11 の 11，15，19 行目で固定長メモリアロケータのサイズを 4 バイト多く指定している．最後に，アプリケーションが使用するために 4 バイトずらした番地を返す（22 行目）．

図 15 にメモリアロケータのメモリ解放用の実装コードを示す．メモリ確保時に保存した添字のデータを取り出し，メモリの確保に使用した固定長メモリプールに結合した呼び口関数を呼び，メモリを解放する（5 行目）．可変長メモリプールを利用してメモリアロケータ

```
 1:  /* メモリアロケータ用のシグニチャsAlloc の定義 */
 2:  signature sAlloc{
 3:    ER alloc( [out] void **p, [in] int32_t size); /* メモリ確保 */
 4:    ER dealloc( [in] const void *p); /* メモリ解放 */
 5:  };
 6:  /* メモリアロケータ制御部分のセルタイプの定義 */
 7:  celltype tAllocMpf{
 8:    /* 受け口 */
 9:    entry sAlloc eAlloc;
10:    /* 固定長メモリプール呼び口配列 */
11:    call sMemoryPoolFixedSize cMpf[ 3 ];
12:    /* 属性定義：固定長メモリプールのそれぞれのサイズ */
13:    attribute{
14:      uint32_t small_size;
15:      uint32_t middle_size;
16:      uint32_t large_size;
17:    };
18:  };
```

図 13　メモリアロケータのセルタイプ記述
Fig. 13　Cell type description for memory allocator.

の実装を行うと効率良く実装することが可能であるが，スタンダードプロファイル[⋆1]では可変長メモリプールが提供されてないため，固定長メモリプールを使用している．

ここからは，send を使用したインタフェース定義とメモリの確保・解放の方法について述べる．まず，図 2 で定義したシグニチャsSig に send を使用した func3 と receive を使用した func4 を追加する（**図 16**）．func3 の第 1 引数（buf）が send で，func4 の第 1 引数（buf）が receive 宣言されており，シグニチャsAlloc を持つメモリアロケータを利用してメモリ領域を確保されることを示す．

次に，メモリアロケータを使用した組み上げの仕方について説明する．メモリアロケータと結合するための組み上げ記述を**図 17** に，そのコンポーネントモデルを**図 18** に示す．図 17 の 2–5 行目では，図 11 で定義したメモリアロケータのインスタンス化・属性の設定をしている．また，3–4 行目のように，インスタンス化の時点で各メモリプールのサイズ

⋆1 スタンダードプロファイルとは，μITRON4.0 仕様におけるプロファイル規定の 1 つで，互換性を保つための標準的な機能のセットのことである．μITRON4.0 仕様準拠の OS はスタンダードプロファイルで実装されていることが多い．

```
 1:  /* セルタイプ tAllocMpf の受け口 eAlloc の関数 alloc の実装コード */
 2:  ER eAlloc_alloc(CELLIDX idx, void** p, int32_t size)
 3:  {
 4:    ER er;
 5:    int32_t index;
 6:     /* 省略：エラー処理 */
 7:    /* 要求されたサイズによって使用するメモリプールを変える */
 8:    if( size >  ATTR_large_size){
 9:      return E_NOMEM;
10:    }
11:    else if(size <= ATTR_small_size ){
12:      index = SMALL;
13:    }
14:    else if(size <= ATTR_middle_size){
15:     index = MIDDLE;
16:    }
17:    else{
18:      index = LARGE;
19:    }
20:    er = cMpf_get( index, p ); /* メモリ確保 */
21:    *(int32_t *)(*p)= index;   /* 添字の保存 */
22:    *p = *p + 4;               /* 4 バイトずらす */
23:    return er;
24:  }
```

図 14　メモリアロケータの実装（メモリ確保）
Fig. 14　Implementation memory allocator for allocation.

```
1:  /* セルタイプ tAllocMpf の受け口 eAlloc の関数 dealloc の実装コード */
2:  ER eAlloc_dealloc(CELLIDX idx, const void* p)
3:  {
4:     /* 省略：エラー処理 */
5:    return  cMpf_release( *(int32_t *)(p - 4), (p - 4) ); /* メモリの解放 */
6:  }
```

図 15　メモリアロケータの実装（メモリ解放）
Fig. 15　Implementation memory allocator for deallocation.

を決めることができる．7–8 行目のようにメモリアロケータと結合の記述は，send または receive を含む受け口関数を持つセルに対して行われる．ここで注目したいのは，メモリアロケータの結合先が buf に対して行われていることである．そうすることで，呼び側でも受け側でも同じメモリアロケータを使用できるようになる．

```
1:  ER func3([send(sAlloc), size_is(size)] int32_t *buf, [in]  int32_t size);
2:  ER func4([receive(sAlloc), size_is(*size)] int32_t **buf,
             [out] int32_t *size);
```

図 16　send と receive を使用したシグニチャ記述例
Fig. 16　An example of signature description for send and receive.

```
 1:  /* メモリアロケータのインスタンス化 */
 2:  cell tAlloc Alloc{
 3:    middle_size = 40;
 4:    large_size = 80;
 5:  };
 6:  /* メモリアロケータと結合 */
 7:  [allocator(eEntry.func3.buf = Alloc.eAlloc,
 8:             eEntry.func4.buf = Alloc.eAlloc)]
 9:  cell tB B{
10:  };
11:  cell tA A {
12:    cCall = B.eEntry;
13:  };
```

図 17　メモリアロケータセルの結合の方法
Fig. 17　A way to connect memory allocator cell.

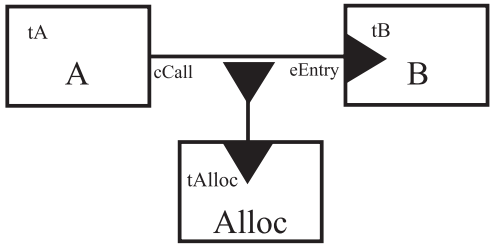


図 18　メモリアロケータを使用したコンポーネントモデル
Fig. 18　Component model for memory allocator.

最後に，実装コードでどのようにメモリを確保し解放するかを func3 を例として解説する．**図 19** に呼び側の実装コードを，**図 20** に func3 の受け口関数の実装コードを示す．図 19 の 3 行目でメモリの確保を行う cCall_func3_buf_alloc と，図 20 の 7 行目でメモリ

```
1:    int32_t *buf;
2:    /* メモリアロケータを使用してメモリ確保する */
3:    cCall_func3_buf_alloc( &buf, SIZE * sizeof(int32_t) );
4:    /* 省略：データの初期化 */
5:    /* 呼び口 cCall の関数 func3 を呼び出す */
6:    cCall_func3( buf, SIZE );
```

図 19　呼び側の実装コード（メモリ確保）
Fig. 19　Implementation code for caller to allocate memory.

```
1:  /* 受け口 eEntry の関数 func3 の実装コード */
2:  ER eEntry_func3(CELLIDX idx, int32_t *buf, int32_t size)
3:  {
4:     /* 省略：エラー処理 */
5:     /* 省略：処理 */
6:     /* メモリアロケータを使用してメモリの開放する */
7:    eEntry_func3_buf_dealloc( buf );
8:  }
```

図 20　受け側の実装コード（メモリ解放）
Fig. 20　Implementation code for callee to deallocate memory.

の解放を行う eEntry_func3_buf_dealloc は，send で宣言された引数に対して自動生成されるもので呼び口関数の一種であり，その呼び先は図 17 の 7–8 行目で指定されたメモリアロケータ（Alloc）になる．まず，呼び側で cCall_func3_buf_alloc を利用してメモリを確保し（図 19 の 3 行目），確保したメモリの初期化を行う（図 19 の 4 行目）．その後，呼び口関数（cCall_func3）を呼び出す（図 19 の 6 行目）．一方，受け側では，処理（図 20 の 5 行目）の終了後，eEntry_func3_buf_dealloc を利用しメモリの解放（図 20 の 7 行目）を行う．

前述のとおり，send や receive を使用することで，メモリ領域の所有権が明記できる．また，メモリアロケータをコンポーネント化することで，多種のメモリアロケータに対応できる．そのため，実装コードは，メモリアロケータに依存しないコードが記述でき再利用性が高まる．この仕組みは，メモリを共有する別プロセッサ（呼び出し形態 (3)）にも適用でき，呼び出しを効率的に実現できる．RPC チャネルについては次章で詳しく述べる．

4. RPC システム

本章では，まず既存の RPC システムの概要を述べた後，提案する RPC システムについて説明する．

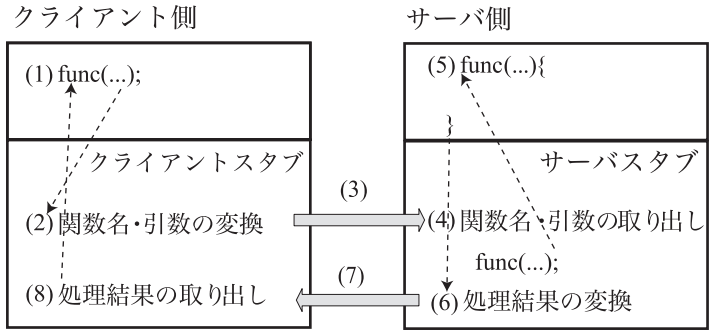


図 21　RPC の処理の流れ
Fig. 21　RPC flow.

4.1 既存の RPC システムの概要

一般に RPC とはローカルの関数呼び出し（手続き）と同じ形式で異なるマシン間で通信を行う技術である．RPC の概要を RFC 1831 [17] で標準化されている SunRPC（ONC RPC）を例として説明する．RPC の処理の流れを**図 21** に示す．この例では，RPC にクライアント・サーバモデルを適用している．呼び側のプログラムをクライアントと呼び，手続きを実行するプログラムをサーバと呼ぶ．RPC は下記の手順で実行される．

(1)　クライアントはローカルの関数と同じように呼び出す．
(2)　クライアントのスタブでは，関数名・引数の値を XDR（external data representation）[18] などの標準的なデータ形式に変換する．
(3)　TCP/IP や UDP/IP といった通信プロトコルなどを利用してサーバへデータを転送する．
(4)　サーバのスタブ（スケルトン）は，呼び出された関数名・引数の値を取り出し，該当する関数を呼び出す．
(5)　要求されたプログラムを実行する．
(6)　処理結果をサーバのスタブに戻し，結果を標準データ形式に変換する．
(7)　結果をクライアントへ転送する．
(8)　クライアントのスタブが結果を取り出して，処理を依頼したプログラムに戻す．

以上の手順により，ローカルの関数の呼び出しを行っているかのようにして，サーバ側の

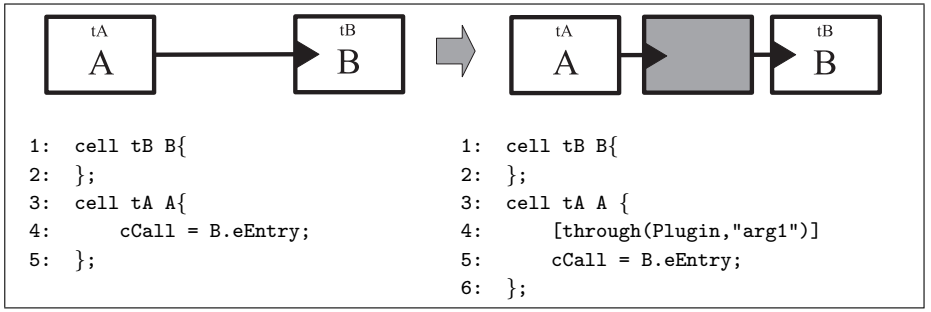


図 22　through 記述の例
Fig. 22　An example of through description.

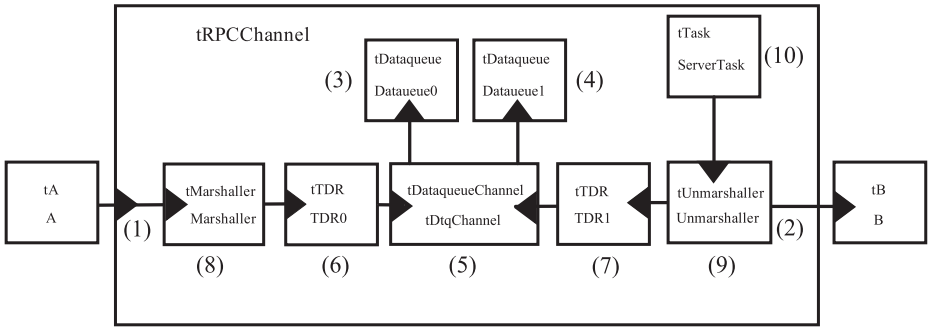


図 23　データキューを用いた RPC チャネルの例
Fig. 23　An example of RPC channel of date queue.

関数が呼ばれる．汎用システムでは，様々な RPC システム[13),22),23)] が存在する．たとえば，XML-RPC[23)] や SOAP[22)] では，データ形式に XML，転送に HTTP を採用しているが基本構成は同じである．スタブは手書きで記述されることもあるが，通常はスタブ生成器やコンパイラと呼ばれるプログラムにインタフェースの定義を与えて自動生成させる．たとえば，SunRPC では rpcgen と呼ばれるスタブ生成器を提供している．本論文では，上記のような RPC 技術をメモリを共有するシステムに対しても適用する．

4.2　提案する RPC システム

提案する RPC システムは既存のものと基本構造は似ているが，メモリ共有を考慮している点，RPC チャネルを選択可能にしている点が異なる．もちろん，汎用システムで RPC 技術をプロセス間通信で適用し，共有メモリを利用しているシステムもあるが，それは使用しているメモリアロケータが 1 種類であるため実現できる．複数のメモリアロケータが存在する組み込みシステムにおける環境では，既存の RPC システムをそのまま適用できない．なお，メモリの扱い方の詳細は 3 章で述べたとおりである．

1 章で述べたとおり，組み込みシステムでは様々な通信チャネルが考えられる．たとえば，TCP/IP，UDP/IP，CAN[10)]，FlexRay[12)]，共有メモリ，リアルタイム OS の機能（データキュー，メールボックス）などがあげられる．本システムでは RPC チャネルをコンポーネントとして扱うことにより，適切な RPC チャネルを選択することができる．

4.3　RPC チャネルの選択方法

RPC チャネルの選択は through 記述を用いて行われる．through 記述とはコンポーネントどうしの結合部分に影響を与えずにコンポーネントを追加する仕組みのことである．すなわち，既存のコンポーネントの変更は必要ない．through 記述は RPC チャネル以外にトレース目的やアクセス制御用のコンポーネントを挟み込む[3)] などの用途で用いられる．**図 22** の左側は元の組み上げ記述とコンポーネント図を示す．図 22 の右側は through を用いた組み上げ記述の例を示す．組み上げ記述の呼び口と受け口を結合する記述に対して（図 22 左 4 行目）through 記述（図 22 右 4 行目）を追加する．through 記述の第 1 引数には，追加されるセルを生成するためのプラグイン名，それ以降はプラグインに渡す引数を記述する．なお，through により追加されるセルの受け口・呼び口のシグニチャは，追加される前の元々の結合のシグニチャと同じである．

4.4　RPC チャネルの例

本節では RPC チャネルの構成をデータキューを用いた RPC チャネルを例として説明する．**図 23** のコンポーネントモデルは，図 1 のセル A とセル B の結合部分に，RPC プラグイン用いた through 記述によって RPC チャネルが追加されたものである．**図 24** に図 23 のセルタイプ記述に示す．図 23 中の番号は，図 24 の記述のコメント内の番号に対応する．RPC チャネルは複合コンポーネントで構成されており，内部に (3)，(4) 2 つのデータキュー，(5) 通信チャネル，(6)，(7) 2 つの TDR（TECS Data Representation），(8) マーシャラ，(9) アンマーシャラ，(10) サーバタスクのセルを持つ．データキューは通信路として使われる．TDR セルは通信チャネルを制御するセルである．マーシャラセルは関数名や引数を通信チャネルに適したデータ形式に変換し受け側に送信する．アンマーシャラセルは受け取った関数や引数の情報をもとに目的の関数を呼び出す．サーバタスクセルはアンマーシャラセルを起動させるセルである．なお，マーシャラがクライアントスタブ，アン

```
 1:  /* RPC チャネルの複合コンポーネントの定義 */
 2:  composite tRPCChannel{
 3:    entry sSig eEntry; /* (1):呼び側セルと結合する受け口 */
 4:    call sSig cCall;   /* (2):受け側セルと結合する呼び口 */
 5:    cell tDataqueue Dataqueue0{}; /* (3):データキュー */
 6:    cell tDataqueue Dataqueue1{}; /* (4):データキュー */
 7:    cell tDataqueueChannel DtqChannel{ /* (5):通信チャネル */
 8:      cDataqueue[ 0 ] = Dataqueue0.eDataqueue;
 9:      cDataqueue[ 1 ] = Dataqueue1.eDataqueue;
10:    };
11:    cell tTDR TDR0{ /* (6):マーシャラ側の TDR */
12:      cChannel = DtqChannel.eChannel0;
13:    };
14:    cell tTDR TDR1{ /* (7):アンマーシャラ側の TDR  */
15:      cChannel = DtqChannel.eChannel1;
16:    };
17:    cell tMarshaller Marshaller{ /* (8):マーシャラ */
18:      cTDR = TDR0.eTDR;
19:    };
20:    cell tUnmarshaller Unmarshaller{ /* (9):アンマーシャラ */
21:      cCall => composite.cCall;      /* (2):呼び口の委譲 */
22:      cTDR = TDR1.eTDR;
23:    };
24:    cell tTask ServerTask{ /* (10):サーバタスク  */
25:      cMain = Unmarshaller.eMain;
26:      priority = 8;     /* タスクの初期優先度 */
27:      stack_size = 1024;/* タスクのスタックサイズ */
28:    };
29:    composite.eEntry =>Marshaller.eEntry; /* (1):受け口の委譲 */
30:  };
```

図 24　自動生成される RPC チャネルコンポーネント記述
Fig. 24　Automatically generated component description of RPC channel.

```
 1:  /* データキューのセルタイプの定義 */
 2:  celltype tDataqueue {
 3:    entry  sDataqueue   eDataqueue;/* データキュー受け口 */
 4:    entry  siDataqueue  eiDataqueue;/* 非タスクコンテキスト用の受け口 */
 5:    attribute {
 6:      ID            id = C_EXP( "DTQID_$id$" ); /* データキューの ID */
 7:      [omit] ATR    dataqueue_attribute = C_EXP( "TA_NULL" );/* 属性 */
 8:      [omit] uint_t count = 4;                  /* データキューの個数 */
 9:      [omit] void   *pdqmb = C_EXP( "NULL" );   /* データキュー領域の先頭番地 */
10:    };
11:    factory { /* ファクトリ記述 */
12:      write( "tecsgen.cfg", "CRE_DTQ( %s, { %s, %s, %s } );",
                 id, dataqueue_attribute, count, pdqmb);
13:    };
14:  };
15:  /* データキューチャネルのセルタイプの定義*/
16:  celltype tDataqueueChannel{
17:    call  sDataqueue   cDataqueue[ 2 ];/* データキューに結合する呼び口配列 */
18:    entry sChannel     eChannel0; /* 呼び側用の受け口 */
19:    entry sChannel     eChannel1; /* 受け側用の受け口 */
20:  };
21:  /* TDR のセルタイプの定義 */
22:  celltype tTDR {
23:    call  sChannel cChannel; /* データキューチャネルに結合する呼び口 */
24:    entry sTDR     eTDR;     /* マーシャラ・アンマーシャラに提供する受け口 */
25:    attribute {
26:      TMO   tmo = C_EXP( "TMO_FEVR" ); /* タイムアウト値 */
27:    };
28:  };
```

図 25　RPC チャネルで利用するセルタイプ記述
Fig. 25　Cell type description for RPC channel.

マーシャラがサーバスタブに相当する．

次に，RPC チャネルの詳細について説明する．図 24 のセルタイプ記述は RPC プラグインによって自動生成されたものである．3 行目において RPC チャネルの受け口，4 行目において呼び口が宣言される．この受け口・呼び口のシグニチャは RPC チャネルが追加される前のシグニチャと同じである．**図 25** は RPC チャネルで使用するセルタイプ，**図 26** は自動生成されるセルタイプをそれぞれ示す．図 24 の 5–6 行目でデータキューを 2 つインスタンス化している．データキューのセルタイプは図 25 の 2–14 行目で定義されている．図 25 の 3 行目は他のタスクに機能を提供する受け口 eDataqueue，4 行目では非タスクコンテキストへ機能を提供する受け口 eiDataqueue をそれぞれ宣言している．また，データキューでも，2.4 節で述べたファクトリ記述（11–13 行目）を使用している．ファクトリ記述を使用することで，RPC チャネルがインスタンス化されたときにデータキューの静的 API が

```
 1:  /* 自動生成されるマーシャラのセルタイプ定義 */
 2:  celltype tMarshaller{
 3:    entry sSig eEntry; /* 呼び側セルと結合する受け口 */
 4:    call  sTDR cTDR;   /* TDR を利用する呼び口 */
 5:  };
 6:  /* 自動生成されるアンマーシャラのセルタイプ定義 */
 7:  celltype tUnmarshaller{
 8:    entry sTaskMain eMain; /* サーバタスクに呼ばれる受け口 */
 9:    call sSig cCall;       /* 受け側セルと結合する呼び口 */
10:    call sTDR cTDR;        /* TDR を利用する呼び口 */
11:  };
```

図 26　自動生成されるセルタイプ
Fig. 26　Automatically generated cell type.

自動的に生成される．セル Dataqueue0（図 24 の 5 行目）は呼び側から受け側への，セル Dataqueue1（図 24 の 6 行目）は逆の向きの通信路として使われる．

次にデータキューチャネルについて説明する．データキューチャネルの定義は図 25 の 16–20 行目でされており，データキューを使用する呼び口配列（17 行目），TDR に提供する受け口（18–19 行目）を持つ．図 24 の 7–10 行目でデータキューチャネルをインスタンス化し，その呼び口配列をデータキューと結合している．

TDR は通信チャネルの送受信の制御を行う．TDR が提供する機能（シグニチャ）を**図 27** に示す．4–10 行目で初期化や制御関数を定義する．12–15 行目で送信関数（put_int）・受信関数（get_int）をそれぞれ定義する．プロセッサ間でエンディアンが異なる場合はここで調整を行う．図 24 の 11–16 行目で TDR を 2 つインスタンス化し，それぞれデータキューチャネルに結合している．

マーシャラは呼び側セル（セル A）から直接呼び出されるセルで，アンマーシャラは受け側セル（セル B）を呼び出すセルである．マーシャラとアンマーシャラは RPC プラグインにより自動生成される（図 26）．さらに，マーシャラとアンマーシャラはセルタイプだけでなく実装コードも自動生成される．自動生成されるマーシャラ（**図 28**）とアンマーシャラのコード（**図 29**）をシグニチャsSig の関数 func1 を例として解説する．マーシャラ・アンマーシャラでは本来送受信のエラーの確認を行っているが，図 28，図 29 では簡略化のため省略する．

下記でマーシャラとアンマーシャラ処理の流れを解説する．まず，マーシャラが送信開始要求を行い（図 28 の 7 行目），アンマーシャラがそれを受け取る（図 29 の 7 行目）．次に，

```
 1:  /* TDR を利用するためのシグニチャsTDR の定義 */
 2:  signature sTDR{
 3:    /* 制御関数 */
 4:    ER  open( [in,size_is(size)]const int8_t *arg,
                 [in]int16_t size, [in]TMO tmo );
 5:    ER  close( void );        /* 通信チャンネルを閉じる */
 6:    ER  reset( void );        /* 通信チャンネルのリセット */
 7:    ER  sop( [in]int no );    /* 送信開始を通知 */
 8:    ER  is_sop( [in]int no );/* 受信開始の確認 */
 9:    ER  eop( void );          /* 送信終了を通知 */
10:    ER  is_eop( void );       /* 受信終了の確認 */
11:    /* データ送受信関数 */
12:     /* 省略：put_int8, put_int16, put_int64 */
13:    ER  put_int32( [in]int32_t in );
14:     /* 省略：get_int8, get_int16, get_int64 */
15:    ER  get_int32( [out]int32_t *out );
16:  }
```

図 27　TDR のシグニチャ
Fig. 27　Signature of TDR.

```
 1:  /* 自動生成されるマーシャラのコード */
 2:  ER eEntry_func1(CELLIDX idx, int32_t data, int32_t* result)
 3:  {
 4:    ER  retval_,ercd_;
 5:    int32_t  func_id = FUNC1;
 6:     /* 省略：エラー処理 */
 7:    cTDR_sop( 1 );               /* 送信開始を通知 */
 8:    cTDR_put_int32( func_id );  /* 関数 ID の送信 */
 9:    cTDR_put_int32( data );      /* 入力引数の送信 */
10:    cTDR_eop();                  /* 送信終了を通知 */
11:    cTDR_is_sop( 2 );            /* 受信開始の確認 */
12:    cTDR_get_int32( result);     /* 出力値の受け取り */
13:    cTDR_get_int( &(retval_) ); /* 戻り値（エラーコード）の受け取り */
14:    cTDR_is_eop();               /* 受信終了の確認 */
15:    return retval_;
16:  }
```

図 28　自動生成されるマーシャラのコード
Fig. 28　Automatically generated marshaller code.

```
 1:  /* 自動生成されるアンマーシャラのコード */
 2:  ER doService_main(CELLIDX idx, intptr_t exinf)
 3:  {
 4:    int32_t   func_id;
 5:     /* 省略： エラー処理 */
 6:    while(1){
 7:      cTDR_is_sop( 1 );            /* 受信開始の確認 */
 8:      cTDR_get_int32( &func_id )); /* 関数 ID の取得 */
 9:      switch( func_id ){
10:        case FUNC1:
11:        {
12:          int32_t  data, result;
13:          ER       retval_;
14:          cTDR_get_int32( &(data) ) );/* 入力引数の受信 */
15:          /* 対象関数の呼び出し */
16:          retval_ = cServerCall_func1( data, &result );
17:          cTDR_is_eop();              /* 受信終了の確認 */
18:          cTDR_sop( 2 );              /* 送信開始を通知 */
19:          cTDR_put_int32( result );   /* 出力値の送信 */
20:          cTDR_put_int32( retval_ );  /* 戻り値（エラーコード）の送信 */
21:          cTDR_eop();                 /* 送信の終了を通知 */
22:        }
23:        break;
24:        case FUNC2:
25:          /* func2 の処理 */
26:        break;
27:      }
28:    }
29:  }
```

図 29　自動生成されるアンマーシャラのコード
Fig. 29　Automatically generated unmarshaller code.

マーシャラは関数の ID を送信し（図 28 の 8 行目），アンマーシャラが関数 ID を受け取り（図 29 の 8 行目），関数の ID ごとの処理を行う（図 29 の 9 行目以降）．関数の ID は TECS ジェネレータによってシステム全体で一意に決まるように自動的に生成される．その後，マーシャラでは関数の引数を送信する（図 28 の 9 行目）．引数が 2 つ以上ある場合は，この処理を繰り返す．引数の送信後，マーシャラは送信終了を通知する（図 28 の 10 行目）．なお，通信チャネルに CAN や TCP/IP などを使用した場合，通信チャネルでは送信可能な最大のデータサイズまでデータをまとめてから送信することが考えられる．しかし，送信終了の通知を受けたときには，通信チャネルは送信可能な最大のデータサイズに達していなくても送信処理を行う．一方，アンマーシャラでは引数を受信し（図 29 の 14 行目），その情報をもとに該当する関数を呼び出す（図 29 の 16 行目）．その後，処理結果と戻り値（エラーコード）を呼び側へ送信する（図 29 の 19–20 行目）．なお，非同期呼び出しの場合は，受け側セルの処理の終了を待たなくてよく，呼び側の受信処理（図 28 の 11–14 行目）は必要ない．

次に，図 24 の 24–28 行目で，図 5 で定義したタスクをインスタンス化し，マーシャラに結合，優先度・スタックサイズの属性の設定を行っている．最後に，図 24 の 21 行目と 29 行目で，複合コンポーネントの受け口をマーシャラに，呼び口をアンマーシャラに処理を委譲している．

上記の RPC チャネルは，一例にすぎない．たとえば，通信チャネルにメールボックスを使用した場合は，上記のマーシャラと TDR のようにデータを 1 つずつを送信するのでなく，データを 1 つの構造体に組み立て，それを送信する方法が考えられる．リアルタイム OS を使用できる環境の場合，提供される機能を使用することで排他制御やタスク間の同期が容易に行える．さらに，ヘテロジニアスな環境であれば，通信チャネルとして共有メモリや DMA を使用することが考えられる．また，既存の RPC システムと同じように TCP/IP や UDP/IP などを使用することも考えられる．それぞれの通信チャネル用の RPC プラグインが提供されていれば，使用する RPC チャネルを変更するには組み上げ記述の through 記述を 1 行変更するだけでよい．

4.5　比 較 実 験

提案するインタフェース記述の有用性を確認するために，別タスク同プロセッサ（呼び出し形態 (2)），メモリを共有する別プロセッサ（呼び出し形態 (3)）で send と in の比較実験を行った[*1]．3.1 節で述べたとおり，send が有用である状況は，非同期呼び出しや並列処理を行う場合である．つまり，呼び出し形態 (2)，(3) において，呼び側セルが受け側セルの処理の終了を待つ必要がない場合（oneway）である．実験では oneway の状況において，send と in を使用した場合のそれぞれの実行サイクル数を比較した．RPC チャネルは 4.4 節で説明したデータキューを用いたものを使用した．測定区間は呼び側セルが関数を呼び出して

*1 out と receive の場合は，処理内容の違いがないため，send と in のみ比較実験を行う．ただし，3.2 節の receive の利点で述べたとおり，receive の場合は適切なメモリサイズを確保でき，メモリ使用量を削減できる．

表 1　実行サイクル数の比較
Table 1　Comparison of the number of clock cycle.

	呼び出し形態 (2)	呼び出し形態 (3)
in (oneway)	22,124	20,063
send (oneway)	20,355	18,371

から，受け側セルの処理が終了するまでである．引数のデータとして 400 バイト（100 個の int32 の配列）を使用した．なお，受け側セルの処理内容は，in と send で同じである．実行サイクル数を測定は ARM アーキテクチャの命令セット・シミュレータである SkyEye[16] を使用した．シミュレータの動作環境は，プロセッサが Centrino Duo 1.83 GHz，OS が Windows XP である．SkyEye 上で動作させるリアルタイム OS として，シングルプロセッサ用に TOPPERS/ASP カーネルを，マルチプロセッサ用に TOPPERS/FMP カーネル[20] をそれぞれ使用した．**表 1** 中の数値は，測定を 100 回行った実行サイクル数の平均値を示す．結果が示すとおり，呼び出し形態 (2) と (3) のどちらの場合でも，send の方が in を使用した場合よりも実行サイクル数が少なく，性能が向上している．

この性能差が生じるのは，send を in で代用したときに RPC チャネル内でメモリ領域のコピーが行われるためである．コピーが必要な理由は，次のとおりである．3.1 節で述べたとおり，oneway の場合では，呼び側セルと受け側セルが並列に動作しているため，呼び側セルと受け側セルがメモリ共有している場合は，in で渡した引数の置かれたメモリ領域を，受け側セルがまだ使用している間に，呼び側セルが内容を書き換えたり，領域を解放したりする可能性がある．そのため，RPC チャネル内で元のメモリ領域をコピーしてから，それを受け側セルに渡す必要がある．

上記の結果から，非同期呼び出しや並列処理を行いたい場合，send は in より有効である．

5. 関 連 研 究

5.1　組み込みシステム向けコンポーネントシステム

組み込みシステムの特定の分野向けのコンポーネントシステムが研究開発されてきている．下記で，それぞれのコンポーネントシステムの特徴と TECS との違いについて述べる．

Koala[21] は，電化製品向けのコンポーネントシステムである．システムの一部を取り替えることにより，製品ファミリを作成できるため，プロダクトライン開発に適している．しかし，Koala は呼び出し形態 (3)，(4) には対応していない．

AUTOSAR[2] は車載向けのソフトウェアコンポーネントである．RTE（Run-Time Environment）上でコンポーネントを配置することで，同一 ECU（Electronic Control Unit：電子制御ユニット）にあるか，異なった ECU にあるかを意識せずに，コンポーネントを配置でき，コンポーネントの再利用性を高めることができるが，呼び出し形態 (3) 対応が十分でない[⋆1]．さらに，通信チャネルは車載用のネットワーク（CAN，FlexRay）に依存しているため，他の組み込みシステムの分野にそのまま適用することは容易ではない．

SaveCCT[1] は車載向けのソフトウェアコンポーネントである．AUTOSAR と同様に通信チャネルは車載用ネットワークに依存してるため，AUTOSAR と同様の問題がある．

Lightweight CCM[14] は，通信ミドルウェアに CORBA[13] を使用したソフトウェアコンポーネントである．CORBA を使用することで，呼び出し形態 (4) に対応できるが，呼び出し形態 (3) のおけるメモリ共有が考慮されていない．さらに，通信ミドルウェアが CORBA に依存しているため，通信チャネルの変更が容易ではない．

CREAM[8] は，呼び出し形態 (4) における通信ミドルウェアを変更可能なソフトウェアコンポーネントである．呼び出し形態 (4) における通信チャネルを選択することができるが，呼び出し形態 (3) におけるメモリ共有は考慮されていない．

5.2　共有メモリを考慮したメッセージ通信

Mach OS[6),15] では，共有メモリを考慮したメッセージ通信を提供している．Mach OS では，呼び側または受け側のどちらかが書き込みを行ったときに，メッセージ通信で送られたメモリ領域のコピーを行う仕組み（コピー・オン・ライト[7]）を利用し効率的にメッセージ通信を行う．コピー・オン・ライトの実現には，MMU（Memory Management Unit）が必要である．しかし，本論文で対象としている組み込みシステムでは，MMU を提供していないプロセッサも数多く存在する．そのため，通信をすべてのプロセッサで利用するには，MMU が提供されていない環境でも利用できる仕組みが必要である．また，MMU が提供されているシステムにおいても，数バイトのメモリ領域をコピーするために，ページ単位（数 K バイト以上）のコピーが必要である．コピー・オン・ライトは，平均実行時間の向上には有効であるが，実行時間の予測可能性が重要なリアルタイムシステムには適さない．本 RPC システムでは MMU が提供されていない環境でも利用でき，提案インタフェース記述を用いることで，効率的に RPC 通信を行うことができる．

⋆1 今後，呼び出し形態 (3) にも対応することが考えられる．

6. おわりに

本論文では，組み込みシステムの様々な環境でソフトウェアを再利用することを目的とし，メモリを共有するシステムに対しても RPC 技術を適用可能なシステムを提案した．提案したインタフェース記述である send と receive によりメモリアロケータとの結合，メモリ領域の所有権を明示できる．メモリアロケータと結合することで，呼び側と受け側で共通のメモリアロケータを使用できる．また，メモリ領域の所有権を明示することで，非同期呼び出しや並列処理を効率的に行うことができる．この記述を用いることで，メモリを考慮することができ，呼び出しをすべての形態で効率的に実現できる．なお，本論文では，組み込み向けコンポーネントシステムである TECS を使用して RPC の実現を行ったが，提案した記述を用いることにより，TECS を用いないシステムにも適用可能だと考えられる．ただし，TECS を用いることで下記のような利点を得られる．

RPC チャネルとメモリアロケータをコンポーネントとして扱うことで，組み込みシステムにおける多種の RPC チャネルやメモリアロケータの中から，適切なものを選択することができる．そうすることで，RPC チャネル・メモリアロケータの実装をプログラムから独立させることができるため，再利用性の高いソフトウェアが記述できる．

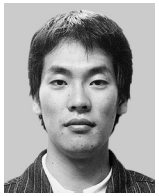
謝辞　本研究の一部は（独）情報処理推進機構（IPA）が実施した 2006 年度下期未踏ソフトウェア創造事業の支援を受けたものである．

参 考 文 献

1) Åkerholm, M., Carlson, J., Fredriksson, J., Hansson, H., Håkansson, J., Möller, A., Pettersson, P. and Tivoli, M.: The SAVE approach to component-based development of vehicular systems, *Journal of Systems and Software*, Vol.80, No.5, pp.655–667 (2007).
2) AUTOSAR. http://www.autosar.org/
3) Azumi, T., Yamada, S., Oyama, H., Nakamoto, Y. and Takada, H.: A New Security Framework for Embedded Component Systems, *The 11th IASTED International Conference on Software Engineering and Applications*, Cambridge, Massachusetts, USA, pp.584–589 (2007).
4) Azumi, T., Yamada, S., Oyama, H., Nakamoto, Y. and Takada, H.: Visual Modeling Environment for Embedded Component Systems, *Proc. IEEE 7th International Conference on Computer and Information Technology*, Fukushima, Japan (2007).
5) Azumi, T., Yamamoto, M., Kominami, Y., Takagi, N., Oyama, H. and Takada, H.: A New Specification of Software Components for Embedded Systems, *Proc. 10th IEEE International Symposium on Object/component/service-Oriented Real-Time Distributed Computing*, Santorini Island, Greece, pp.46–50 (2007).
6) Black, D.: Scheduling Support for Concurrency and Parallelism in the Mach Operating System, *IEEE Computer*, Vol.23, No.5, pp.35–43 (1990).
7) Bobrow, D.G., Burchfiel, J.D., Murphy, D.L. and Tomlinson, R.S.: TENEX, a paged time sharing system for the PDP — 10, *Comm. ACM*, Vol.15, No.3, pp.135–143 (1972).
8) Chetan, R., Jiyong, P., Jungkeun, P. and Seongso, H.: CREAM: A Generic Build-Time Component Framework for Distributed Embedded Systems, *14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, Kaohsiung, Taiwan, pp.318–323 (2008).
9) Crnkovic, I.: Component-based software engineering for embedded systems, *Proc. 27th International Conference on Software Engineering*, Missouri, USA, pp.712–713 (2005).
10) Kiencke, U.: Controller Area Network — from Concept to Reality, *Proc. 1st International CAN Conference*, Mainz, Germany, pp.0-11–0-20 (1994).
11) Lau, K.-K. and Wang, Z.: Software Component Models, *IEEE Trans. Softw. Eng.*, Vol.33, No.10, pp.709–724 (2007).
12) Makowitz, R. and Temple, C.: Flexray — A communication network for automotive control systems, *Proc. 2006 IEEE International Workshop on Factory Communication Systems*, Torino, Italy, pp.207–212 (2006).
13) OMG: CORBA. http://www.omg.org/corba/
14) OMG: Lightweight Corba Component Model (CCM). http://www.omg.org/docs/ptc/03-11-03.pdf
15) Rashid, R., Tevanian, A., Young, M., Golub, D., Baron, R., Black, D., Bolosky, W.J. and Chew, J.: Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures, *IEEE Trans. Comput.*, Vol.37, No.8, pp.896–908 (1988).
16) SkyEye. http://www.skyeye.org/index.shtml
17) Srinivasan, R.: RFC 1831. ftp://ftp.rfc-editor.org/in-notes/rfc1831.txt
18) Srinivasan, R.: RFC 1832. ftp://ftp.rfc-editor.org/in-notes/rfc1832.txt
19) Stewart, D.B., Volpe, R.A. and Khosla, P.K.: Design of dynamically reconfigurable real-time software usingport-based objects, *Software Engineering*, Vol.23, No.12, pp.759–776 (1997).
20) TOPPERS プロジェクト．http://www.toppers.jp/
21) van Ommering, R., van der Linden, F., Kramer, J. and Magee, J.: The Koala Component Model for Consumer Electronics Software, *IEEE Computer*, Vol.33, No.3,

pp.78–85 (2000).
22) W3C: SOAP. http://www.w3.org/TR/soap12-part0/
23) Winer, D.: XML-RPC. http://www.xmlrpc.com/spec
24) Yamada, S., Nakamoto, Y., Azumi, T., Oyama, H. and Takada, H.: Generic Memory Protection Mechanism for Embedded System and Its Application to Embedded Component Systems, *Proc. IEEE 8th International Conference on Computer and Information Technology*, Sydney, Australia, pp.557–562 (2008).

(平成 20 年 9 月 28 日受付)
(平成 20 年 12 月 22 日採録)



安積　卓也（学生会員）

日本学術振興会特別研究員．2006 年より名古屋大学大学院情報科学研究科情報システム学専攻博士後期課程．2006 年会津大学大学院コンピュータ理工学研究科情報システム学専攻博士前期課程修了．組み込みシステム向けのコンポーネントシステムの研究に従事．IEEE，電子情報通信学会，日本ソフトウェア科学会各会員．

大山　博司

オークマ株式会社・主管技師．1984 年岐阜大学工学部電気工学科卒業．同年株式会社大隈鐵工所（現オークマ株式会社）に入社．2002 年岐阜大学大学院工学研究化電子情報システム工学専攻後期課程修了．2004 年より TOPPERS プロジェクト・コンポーネント仕様 WG 主査を兼務．数値制御装置の開発に従事し，リアルタイム制御，組み込みシステムのプログラミング言語に関する研究等を行う．博士（工学）．電子情報通信学会，計測自動制御学会各会員．

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．同専攻助手，豊橋技術科学大学情報工学系助教授等を経て，2003 年より現職．2006 年より大学院情報科学研究科附属組込みシステム研究センター長を兼務．リアルタイム OS，リアルタイムスケジューリング理論，組み込みシステム開発技術等の研究に従事．オープンソースの ITRON 仕様 OS 等を開発する TOPPERS プロジェクトを主宰．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．