組込みシステムに適した動的メモリアロケータのコンポーネント化
Componentized Dynamic Memory Allocator for Embedded Systems

山本　拓朗[1]　　大山　博司[2]　　安積　卓也[1]
Takuro Yamamoto[1]　Hiroshi Oyama[2]　Takuya Azumi[1]

1. はじめに

近年，組込みシステムは多くの機器に組み込まれており，この IoT 社会を支える重要な役割を担っている [1] [2]．多くの組込みシステムは，厳しいリソース制約を持っており，効率的なメモリ確保が要求される．さらに，リアルタイム性を要求される組込みシステムでは，メモリ確保の実行時間も重要となる．

このような要求を満たす，組込みシステムに適した動的メモリアロケータとして，TLSF (Two-Level Segregate Fit) アロケータが提案されている [3] [4]．TLSF アロケータは，メモリブロックを 2 段階で分類することでメモリ利用効率を向上させており，常に $O(1)$ で実行するため最悪実行時間を見積もることができ，リアルタイムシステムに適した動的メモリアロケータである．TLSF アロケータは，メモリ利用効率が良く，リアルタイムシステムに向いているため，多くのリアルタイム OS で採用されている．

しかし，現状では，複数のスレッドが並行動作すると，メモリの衝突が起きる場合がある．本研究では，組込み向けコンポーネントシステムである TECS (TOPPERS Embedded Component System) [5] [6] を用いて TLSF アロケータをコンポーネント化する．TECS を用いたコンポーネントベース開発は，ソフトウェアの再利用性を向上させ，システムを可視化できるため，ソフトウェア開発の生産性を向上させることができる．コンポーネント化された TLSF アロケータは，各コンポーネントが独自のヒープ領域を保持するため，スレッドセーフなメモリアロケータを実現できる．さらに，コンポーネントとして特性を利用できるため，メモリサイズの変更等が柔軟になる．本論文では，TLSF メモリアロケータコンポーネントを提案し，そのユースケースとして，利用例を述べる．

[1] 大阪大学基礎工学研究科
Graduate School of Engineering Science, Osaka University
[2] オークマ株式会社
OKUMA Corporation

本論文における貢献は以下の通りである．

- 複数のスレッドが並行動作しても排他制御なしでスレッドセーフに動作する
- 各スレッドで独自のヒープ領域を容易に設定できる
- コンポーネント化により再利用性が向上するため，様々なシステムに拡張できる
- コンポーネント図により可視化できるため，機能役割の理解が容易になる

本論文の構成は次の通りである．まず 2 章で，コンポーネントベース開発と提案するアロケータのベースとして用いる TECS について述べる．3 章では，TLSF メモリアロケータのコンポーネント設計と実装について述べる．4 章では，ユースケースとして，利用例を述べ, 最後に 5 章で本論文をまとめる．

2. コンポーネントベース開発

ソフトウェア開発の生産性を向上させる手法のひとつにコンポーネントベース開発がある [7] [8]．コンポーネントベース開発は，ソフトウェアの部品 (コンポーネント) を組み合わせて開発を行う手法である．ソフトウェアの再利用性が改善する上に，コンポーネント図によりシステム全体の構造を可視化するため，ソフトウェア開発の生産性を向上できる．さらに，システムの拡張や仕様変更にも柔軟に対応することができる．組込みシステム向けのコンポーネント技術として，TECS や AUTOSAR [9]，SaveCCM [10] がある．本研究で利用した TECS について次で述べる．

2.1 TECS

TECS (TOPPERS Embedded Component System) は，TOPPERS プロジェクト [11] で開発されている組込みシステム向けのコンポーネント技術である．TECS では，コンポーネントの生成と結合はすべて静的に行われ，最適化されるため，コンポーネント化における実行時間や消費メモリのオーバヘッドを抑えることができる．さらに，組込

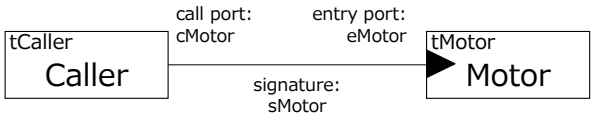

図 1　TECS コンポーネント図の例

```
1  signature sMotor {
2      void initializePort( [in]int32_t type );
3      int32_t getCounts( void );
4      ER resetCounts( void );
5      ER setPower( [in]int power );
6      ER stop( [in] bool_t brake );
7      ER rotate( [in] int degrees,
8                 [in] uint32_t speed_abs,
9                 [in] bool_t blocking );
10 };
```

図 2　シグニチャ記述

みシステム開発に広く用いられている C 言語による開発を採用しており，組込みシステムに特化したコンポーネントシステムである．

2.1.1　コンポーネントモデル

図 1 に TECS コンポーネント図の例を示す．TECS では，インスタンス化されたコンポーネントはセル (*cell*) と呼ばれ，受け口 (*entry*)，呼び口 (*call*)，属性，変数を持つ．受け口は自身の機能を提供するインタフェースで，呼び口は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．セルは複数の受け口や呼び口を持つことができる．セルの提供する関数は，C 言語で実装される．

受け口と呼び口の型は，セルの機能を使うためのインタフェースであるシグニチャによって定義される．セルの呼び口は，同じシグニチャを持つ他のセルの受け口と結合できる．セルの型は，セルタイプと呼ばれ，受け口，呼び口，属性，変数の組を定義している．

2.1.2　コンポーネント記述

TECS のコンポーネント記述は，シグニチャ記述，セルタイプ記述，組上げ記述に分類され，CDL (Component Description Language) ファイルに記述する．図 1 のコンポーネント記述について次に述べる．

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．図 2 に示す通り，*signature* キーワードに続けて，シグニチャ名 (sMotor) を記述する．TECS では，インタフェースの定義を明確にするために，入力と出力にはそれぞれ，[in] と [out] という指定子が付けられる．

セルタイプ記述

セルタイプ記述は，受け口，呼び口，属性，変数を用いてセルタイプを定義する．*celltype* キーワードに続けて，セルタイプ名 (tCaller) を記述する．図 3 に示す通り，受け口は，*entry* キーワードに続けて，シグニチャ名，受け口名を記述する．同様にして，呼び口も定義できる．属性と変数は，それぞれ *attr*，*var* キーワードを用いて列挙する．

```
1  celltype tCaller {
2      call sMotor cMotor;
3  };
4  celltype tMotor {
5      entry sMotor eMotor;
6      attr {
7          int32_t port;
8      };
9      var {
10         int32_t currentSpeed = 0;
11     };
12 };
```

図 3　セルタイプ記述

```
1  cell tMotor Motor {
2  };
3  cell tCaller Caller {
4      cMotor = Motor.eMotor;
5  };
```

図 4　組上げ記述

組上げ記述

組上げ記述は，セルをインスタンス化し，セルを結合する．*cell* キーワードに続けて，セルタイプ名，セル名を記述する．呼び口名，“=”，結合先の受け口名の順に記述し，セルを結合する．図 4 では，セル Caller の呼び口 cMotor と，セル Motor の受け口 eMotor が結合されている．

2.1.3　開発フロー

図**??**に TECS を利用した開発フローを示す．TECS ジェネレータは，CDL ファイルから，C 言語のインターフェースコード (.h や.c) とリアルタイム OS のシステム設定ファイル (.cfg) を生成する．

TECS を用いたソフトウェア開発者は，コンポーネント設計者とアプリケーション開発者に分けられる．コンポーネント設計者は，セル間のインターフェースであるシグニチャやセルの型であるセルタイプを定義する．これらが定義された CDL ファイルから生成されたテンプレートコードを利用し，コンポーネントの提供する機能や振る舞いを C 言語で実装する．コンポーネントの機能を実装したソースコードは，セルタイプコードと呼ばれる．アプリケーション開発者は，コンポーネント図や定義済みのセルタイプを利用して，組上げ記述によりセル同士を結合させることでアプリケーションを開発する．最終的にヘッダ，インターフェースコード，セルタイプコードをコンパイル・リ

ンクすることで，アプリケーションモジュールが生成される．

3. 設計と実装

3.1 TLSF

TLSF (Two-Level Segregate Fit) メモリアロケータは，M. Masmano らによって提案されたリアルタイムシステムに適した動的メモリアロケータである．TLSF メモリアロケータは以下のような特徴がある．

リアルタイム性

メモリの確保や解放にかかる最悪実行時間はデータサイズに依存しないため，常に $O(1)$ で実行される．応答時間を見積もることができるため，リアルタイムシステムに適したメモリアロケータである．

高速

最悪実行時間が常に見積もれることに加え，TLSF は高速に実行される．x86 アーキテクチャで最大 168 のプロセッサ命令を実行することができる．

効率的なメモリ消費

TLSF では，メモリの断片化 (フラグメンテーション) を抑えることで，メモリ効率化を実現している．様々なテストで，平均フラグメンテーション 15 %未満，最大フラグメンテーション 25 %未満を得ている．

3.1.1 TLSF アルゴリズム

TLSF アルゴリズムは，メモリブロックを 2 段階に分類して，要求されたメモリサイズに最適なメモリブロックを検索する．図 5 に TLSF アルゴリズムの概要を示す．例として，*malloc*(100) というメモリ確保の要求がされた場合を考える．まず第一段階では，要求されたメモリサイズの最上位ビットで分類する．この場合，100 を 2 進数で表すと 1100100 であるため，最上位ビットから 64 から 128 の範囲であることが分かる．次に第二段階では，さらに細かい分類を行う．今回の場合，64 から 128 を 4 分割しており，100 は 96-111 のブロックに入る．この範囲にあるフリーブロック [*1] を使用するという流れである．

単純な固定サイズブロック確保では最大 50%の無駄を生じてしまうが，TLSF では 2 段階で細かく分類するため，メモリ効率が良いアルゴリズムとなっている．また，検索にかかる時間も高速で常に同じ速度で実行される．

3.2 コンポーネント設計

TLSF メモリアロケータのコンポーネント設計について述べる．本研究では，TECS を用いて TLSF をコンポーネント化を行っている．使用した TLSF のバージョンは，2.4.6 である [*2]．

図 6 は，アロケータが使用するメモリ管理用のシグニチャ記述である．メモリプール初期化関数 *initializeMemoryPool*，メモリ確保用の関数 *calloc*, *malloc*, *realloc*，そしてメモリ解放用の関数 *free* をシグニチャとして定義している．

TLSF メモリアロケータコンポーネントのセルタイプ記述を図 7 に示す．受け口 *eMalloc* は，メモリ管理 (確保や解放) を行うすべてのコンポーネントと結合される．ここで，*[inline]* は受け口関数をインライン関数として提供するための指定子である．メモリプールサイズをコンポーネントの属性として，メモリプールへのポインタをコンポーネントの内部変数として定義している．各コンポーネントが独自のヒープ領域を保持しているため，異なるスレッドで同時にメモリ管理用の関数を呼び出した場合でも，排他

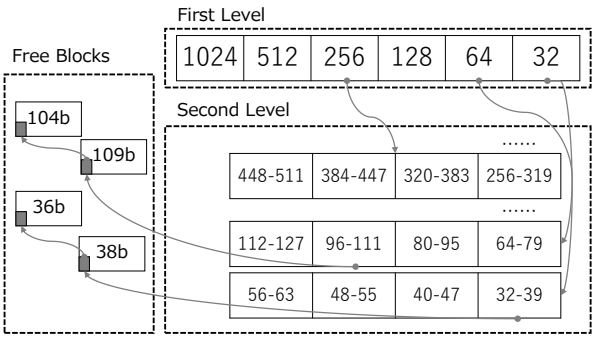

図 5 TLSF アルゴリズム

```
1  [deviate]
2  signature sMalloc {
3      int initializeMemoryPool(void);
4      void *calloc( [in]size_t nelem,
5                    [in]size_t elem_size );
6      void *malloc( [in]size_t size );
7      void *realloc( [in]const void *ptr,
8                     [in]size_t new_size );
9      void free( [in]const void *ptr );
10 };
```

図 6 メモリ管理用のシグニチャ記述

```
1  celltype tTLSFMalloc {
2      [inline]
3          entry sMalloc eMalloc;
4      attr {
5          /* memory pool size in bytes */
6          size_t memoryPoolSize;
7      };
8      var {
9          [size_is( memoryPoolSize / 8 )]
10             uint64_t *pool;
11     };
12 };
```

図 7 TLSF メモリアロケータのセルタイプ記述

[*1] フリーブロックは使用可能なメモリブロックのことである．
[*2] http://www.gii.upv.es/tlsf/main/repo

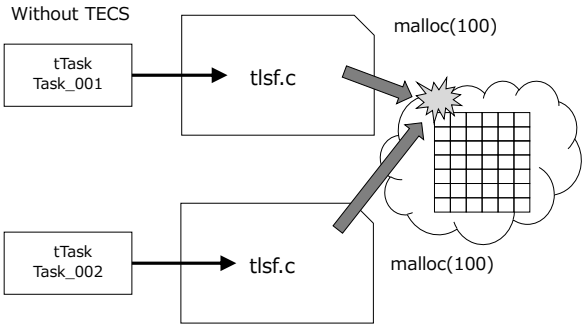

図 8　コンポーネント化される前の TLSF

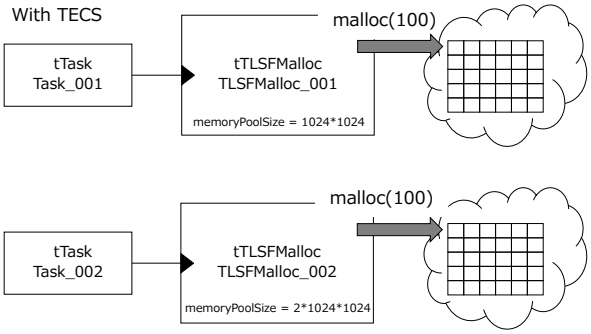

図 9　コンポーネント化された TLSF

```
1  cell tTask Task_001 {
2      cMalloc = TLSFMalloc_001.eMalloc;
3  };
4  cell tTLSFMalloc TLSFMalloc_001 {
5      memoryPoolSize = 1024*1024; /* 1MB */
6  };
7  cell tTask Task_002 {
8      cMalloc = TLSFMalloc_002.eMalloc;
9  };
10 cell tTLSFMalloc TLSFMalloc_002 {
11     memoryPoolSize = 2*1024*1024; /* 2MB */
12 };
```

図 10　TLSF メモリアロケータコンポーネントの組上げ記述

制御を行うことなく，メモリ競合を起こさない．

図 8 に示すように，コンポーネント化を行う前の TLSF では，ヒープ領域を複数のスレッドで共有しているため，複数のスレッドからメモリの確保や解放を同時に行うとメモリ競合が生じる場合があった．図 9 のように，TECS を用いて TLSF のコンポーネント化を行うと，各コンポーネントが独自にヒープ領域を保持し，その中でメモリ管理を行うため，排他制御なしでスレッドセーフに動作することが可能になる．

図 10 に，図 9 のコンポーネント図で表される TLSF メモリアロケータコンポーネントの組上げ記述を示す [*3]．2 組のタスクコンポーネントと TLSF コンポーネントが結合

[*3] 他にも呼び口や受け口，属性，内部変数を記述するが，ここでは簡略化のため省略している．

```
1  void*
2  mrb_TECS_allocf(mrb_state *mrb, void *p,
3                  size_t size, void *ud)
4  {
5      CELLCB *p_cellcb = (CELLCB *)ud;
6      if (size == 0) {
7          //tlsf_free(p);
8          cMalloc_free(p);
9          return NULL;
10     }
11     else if (p) {
12         //return tlsf_realloc(p, size);
13         return cMalloc_realloc(p, size);
14     }
15     else {
16         //return tlsf_malloc(size);
17         return cMalloc_malloc(size);
18     }
19 }
```

図 11　TLSF メモリアロケータコンポーネントの利用部分

されている．それぞれのメモリプールサイズは 5 行目及び 11 行目のように内部変数として設定可能である．図 11 は，TLSF メモリアロケータコンポーネントの受け口関数を実際に呼び出しているコード部分である．利用部分は，4.1 章で紹介する mruby on TECS フレームワーク [12] で，VM がメモリ管理を行う関数である．8 行目は，TLSF メモリアロケータコンポーネントの *free* 関数を呼び出している．*cMalloc_*は，呼び口名を表わしている (図 10 の 2 行目) 同様に，13 行目，17 行目はメモリ確保を行っている部分である．図 11 のコードが実行されるセルが，*Task_001* の場合は *TLSFMalloc_001* で，*Task_002* の場合は *TLSFMalloc_002* でメモリ管理が行われる．このように，TECS を用いたコンポーネントベース開発では，それぞれ結合しているセルは異なるものの，同じコードで動作させることが出来る．

4. ユースケース

本章では，提案するコンポーネント化された TLSF メモリアロケータのユースケースとして，利用されるプラットフォームとその効果について述べる．

4.1 mruby on TECS

mruby on TECS は，mruby (軽量 Ruby) [13] [14] を用いたコンポーネントベース開発が可能な組込みソフトウェア開発フレームワークである [12]．スクリプト言語を用いることで，組込みソフトウェア開発の生産性向上を目指している．一般に，スクリプト言語は実行速度が遅いため，組込みソフトウェアに向いていないが，mruby on TECS では，mruby-TECS ブリッジの機能によって，mruby プログラムから C 言語の関数を呼び出すことができるため，

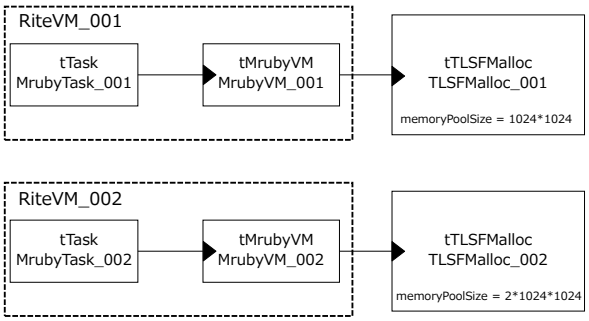

図 12 RiteVM と TLSF のコンポーネント図

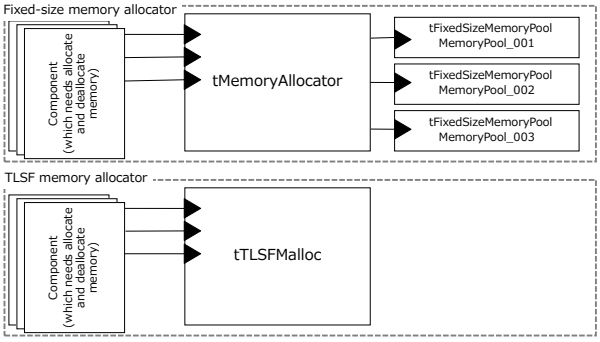

図 13 固定長メモリアロケータと TLSF メモリアロケータ

C 言語と変わらない速度でプログラムを実行することができる．mruby プログラムは，mruby コンパイラによってバイトコードに変換され，RiteVM と呼ばれる VM 上で実行される．本フレームワークでは，RiteVM や RTOS (Real-Time OS) の機能もすべて TECS によってコンポーネント化されている．

4.1.1 マルチ VM 対応

本フレームワークでは，VM が行うメモリ管理に TLSF メモリアロケータを採用している．しかし，既存の TLSF メモリアロケータではスレッドセーフではないため，複数のスレッドからメモリの確保や解放を行うと，メモリ衝突が起きてしまう．VM は高い頻度でメモリの確保と解放を繰り返すため，マルチ VM として複数の VM を起動すると，すぐに衝突を起こしてしまう．

図 12 のように，VM に TLSF メモリアロケータコンポーネントを結合し，各 VM で独自のヒープ領域を持つように設計する．各 VM が結合している TLSF コンポーネントがそれぞれメモリプールを保持しているため，複数の VM がメモリ衝突を起こすことなく，実行できる．図 12 では，1 つ目の VM が 1MB (1024*1024) のヒープ領域，2 つ目の VM が 2MB (2*1024*1024) のヒープ領域を持っていることを示している．コンポーネント化により，各 VM で異なるヒープ領域を設定することが容易になっている．さらに，RiteVM はインクリメンタル GC (Garbage Collection) を行うが，独自のメモリプールを保持しているため，GC を始めた VM が GC の実行によって，他の VM の実行を妨げることもない．

4.2 TINET+TECS

TINET+TECS は，組込み向けの TCP/IP プロトコルスタックである TINET (Tomakomai InterNETworking) [15] [16] を TECS によってコンポーネント化した TCP/IP プロトコルスタックである [17]．複雑で大規模なソースコードやマクロで構成されている TINET を TECS によってコンポーネント化することで，実行時間やメモリ消費量のオーバヘッドを抑えつつ，TCP/UDP や IPv4/IPv6 といった通信プロトコルの変更等のコンフィグラビリティが向上している．

4.2.1 データ送受信用のメモリ管理

データの送受信処理を行う際，各プロトコルでメモリ領域の確保と解放が行われる．TINET+TECS や TINET は，プロトコル間のデータコピー回数を最少とするなど，組込みシステムの厳しいメモリ制約に対応している．しかし，現状では TOPPERS/ASP[*4] でサポートされている固定長メモリアロケータを利用している．TCP/IP プロトコルスタックでは，TCP 層や IP 層，Ethernet 層といった各層でメモリの確保や解放が繰り返されるので，アロケータの役割は重要になっている．図 13 のように，TINET+TECS ではメモリ管理を行うすべてのコンポーネントがアロケータと結合されている．TLSF メモリアロケータは，固定長メモリアロケータと比べて，同じ速度でメモリ効率を向上できるため，TINET+TECS に TLSF メモリアロケータを組み込むことで，より良いメモリ効率を期待できる．

さらに，図 13 のように，固定長メモリアロケータは異なるサイズのメモリプールを用意し，必要に応じてメモリプールを選択する必要がある．TLSF メモリアロケータコンポーネントを利用することで，メモリプールを選択する必要もなく，効率的にメモリ管理を行うことが可能になる．TINET+TECS は，TECS によってコンポーネントベースで開発されているため，提案する TLSF メモリアロケータコンポーネントの拡張性も良い．

5. おわりに

本論文では，リアルタイムシステムに適した動的メモリアロケータである TLSF メモリアロケータのコンポーネント化を提案した．提案するコンポーネント化された TLSF メモリアロケータは独自のヒープ領域を保持できるため，複数のスレッドから同時にメモリの確保や解放を行っても，メモリ競合が起こすことがない．ヒープ領域のサイズはコンポーネントの内部変数として設定するため，サイズ変更も容易になる．

提案する TLSF メモリアロケータは，コンポーネント化

[*4] TOPPERS/ASP は，TOPPERS プロジェクトで開発されている μITRON [18] をベースにしたリアルタイム OS である．

によって再利用性も向上している．ユースケースとして，mruby on TECS フレームワークや，TINET+TECS で利用できることを述べたが，その他にも TECS を採用したリアルタイム OS である TOPPERS/ASP3 [19] に適用することも可能である．利用したいシステムのメモリアロケータ部分を TLSF コンポーネントに入れ替えるだけで，利用可能となる．さらに，コンポーネント図によってシステムを可視化でき，複雑で大規模なシステムを把握することに役立つ．

今後は，提案する TLSF コンポーネントの拡張として，統計情報を取得できるような機能を提供する．メモリ確保の頻度や残りのヒープ領域サイズ等の情報を取得することで，メモリ制約の厳しい組込みシステムのソフトウェア開発の一助になると考えている．

参考文献

[1] Xu, L. D., He, W. and Li, S.: Internet of Things in Industries: A Survey, *IEEE Transactions on Industrial Informatics*, Vol. 10, No. 4, pp. 2233–2243 (online), DOI: 10.1109/TII.2014.2300753 (2014).

[2] Perera, C., Zaslavsky, A., Christen, P. and Georgakopoulos, D.: Context Aware Computing for The Internet of Things: A Survey, *IEEE Communications Surveys Tutorials*, Vol. 16, No. 1, pp. 414–454 (online), DOI: 10.1109/SURV.2013.042313.00197 (2014).

[3] Masmano, M., Ripoll, I., Crespo, A. and Real, J.: TLSF: a new dynamic memory allocator for real-time systems, *Proceedings. 16th Euromicro Conference on Real-Time Systems, 2004. ECRTS 2004.*, pp. 79–88 (online), DOI: 10.1109/EMRTS.2004.1311009 (2004).

[4] : TLSF, `http://www.gii.upv.es/tlsf/`.

[5] : TECS, `http://www.toppers.jp/tecs.html`.

[6] Azumi, T., Yamamoto, M., Kominami, Y., Takagi, N., Oyama, H. and Takada, H.: A New Specification of Software Components for Embedded Systems, *Proceedings of the 10th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC)*, pp. 46–50 (2007).

[7] Crnkovic, I.: Component-based Software Engineering for Embedded Systems, *Proceedings of the 27th International Conference on Software Engineering (ICSE)*, pp. 712–713 (2005).

[8] Cai, X., Lyu, M. R., Wong, K.-F. and Ko, R.: Component-based software engineering: technologies, development frameworks, and quality assurance schemes, *Processings of 7th Asia-Pacific Software Engineering Conference (APSEC)*, pp. 372–379 (2000).

[9] : AUTOSAR, `http://www.autosar.org/`.

[10] Hansson, H., Åkerholm, M., Crnkovic, I. and Torngren, M.: SaveCCM - a component model for safety-critical real-time systems, *Euromicro Conference, 2004. Proceedings. 30th*, pp. 627–635 (2004).

[11] : TOPPERS Project, `http://www.toppers.jp/en/index.html`.

[12] Azumi, T., Nagahara, Y., Oyama, H. and Nishio, N.: mruby on TECS: Component-Based Framework for Running Script Program, *Proceedings of the 18th IEEE International Symposium on Real-Time Computing (ISORC)*, pp. 252–259 (2015).

[13] Tanaka, K., Nagumanthri, A. D. and Matsumoto, Y.: mruby – Rapid Software Development for Embedded Systems, *Proceedings of the 15th International Conference on Computational Science and Its Applications (ICCSA)*, pp. 27–32 (2015).

[14] : mruby, `https://github.com/mruby/mruby`.

[15] : TINET, `https://www.toppers.jp/en/tinet.html`.

[16] 阿部司, 吉村斎 and 久保洋: 組込みシステム用 TCP/IP プロトコルスタックの実装と評価, **情報処理学会論文誌**, Vol. 44, No. 6, pp. 1583–1592 (2003).

[17] 原拓, 石川拓也, 大山博司 and 高田広章: 組込み向け TCP/IP プロトコルスタックのコンポーネント設計, **第 *26* 回組込みシステム研究発表会**, No. 1 (2012).

[18] Takada, H. and Sakamura, K.: μITRON for Small-Scale Embedded Systems, *IEEE Micro*, Vol. 15, No. 6, pp. 46–54 (1995).

[19] Kawada, T., Azumi, T., Oyama, H. and Takada, H.: Componentizing an Operating System Feature Using a TECS Plugin, *Proceedings of the 4th IEEE International Conference on Cyber-Physical Systems, Networks, and Applications (CPSNA)*, pp. 95–99 (2016).