Vol.2014-SE-184 No.4
Vol.2014-EMB-33 No.4
2014/5/19

アスペクト指向プログラミングに基づく分散リアルタイムＯＳ

齋藤 亘弘[1,†1]　兪 明連[1,a)]　横山 孝典[1,b)]

概要：
組込みシステムは様々な用途に使用されるため，アプリケーションによってリアルタイムＯＳに要求される機能も異なってくる．ところが組込みシステムには厳しいリソースの制約があるため，様々な機能を持つリアルタイムＯＳを使用することは困難であり，対象アプリケーションに特化したリアルタイムＯＳを提供することが望ましい．本論文では，アプリケーションによって要求されるリアルタイムＯＳの機能のひとつとして分散システムへの対応を取り上げ，アスペクト指向プログラミングを用いて分散化の機能を追加することで，既存のリアルタイムＯＳのソースコードを直接修正せずに分散リアルタイムＯＳを実現する手法について述べる．具体的には自動車制御分野向けの OSEK OS 仕様に基づく TOPPERS/OSEK カーネルを対象に，自ノード上か他ノード上かを意識せずにタスク管理やイベント制御を可能とする，位置透過性のあるシステムコールを実現するためのアスペクトを提案するとともに，実用上問題ないオーバヘッドで実装可能であることを示す．

1. はじめに

組込みシステムは様々な用途に使用されるため，要求されるＯＳの機能もアプリケーションによって異なることが多い．ところが，組込みシステムにはリソースの制約があり，様々なアプリケーションに対応可能な豊富な機能を持つ単一の組込みＯＳを使用することは困難である．このため，それぞれのアプリケーションに特化したリアルタイムＯＳを提供することが望ましい．しかし，リアルタイムＯＳの各機能は実装上複雑にからみあっていることが多く，機能単位での削除や追加は容易ではない．

そこで，横断的関心事を分離してモジュール化することのできるアスペクト指向プログラミング [1] を用いて，組込みＯＳをカスタマイズする研究がなされている．アスペクト指向プログラミングを用いることで，ソースコードを直接修正することなく，組込みＯＳのコンフィギュレーションや機能の追加・変更が可能になる．

Beuche，Spinczyk らは，アスペクト指向プログラミングを用いてアーキテクチャ非依存な組込みＯＳの実現する手法を提案している [2], [3]．その後，Lohman，Spinczyk らのグループは，彼らが開発したアスペクトプログラミング言語 AspectC++[4] による組込みＯＳの実装を行い，そのオーバヘッドが十分小さいことを示すとともに [5]，最初からアスペクトを意識した設計が必要として Aspect-Aware Design による組込み OS を提案し，実装している [6], [7]．

Afonso らは，リアルタイム組込みＯＳの同期（排他制御）やロギングにアスペクトを適用している [8]．Park らは，プログラミング言語非依存なアスペクト指向プログラミング環境 AOX を開発し，カスタマイズ可能な組込みＯＳへの適用を提案している [9]．また我々は，既存のリアルタイムＯＳのソースコードを修正せずに，固定優先度スケジューリングを EDF （Earliest Deadline First）スケジューリングに変更するアスペクトを提案した [10]．

近年，複数の組込みコンピュータをネットワーク接続した分散型の組込みシステムが増えている．このため，組込みＯＳに要求される機能のひとつとして，分散システムへの対応が挙げられる．しかし，これまでのところ，アスペクト指向プログラミングを用いて組込みＯＳに分散処理機能を追加する研究は行われていない．

そこで本研究の目的は，既存のリアルタイムＯＳのソースコードを直接修正せずに，アスペクト指向プログラミングを用いて分散処理機能を追加することで，分散リアルタイムＯＳを実現することである．

本論文では，自動車制御分野向けの標準リアルタイムＯ

[1] 東京都市大学
Tokyo City University
[†1] 現在，スタンレー電気株式会社
Presently with Stanley Electric Co., Ltd.
[a)] yoo@cs.tcu.ac.jp
[b)] yokoyama@cs.tcu.ac.jp

Sである OSEK OS[11] をベースに，アスペクト指向プログラミングにより分散リアルタイムＯＳを実現する．実現する分散リアルタイムＯＳは，自ノード上のタスクのみでなく他ノード上のタスクに対しても同一のシステムコールを用いてタスク管理やイベント制御を可能とする，位置透過なシステムコールを提供する．具体的には，オープンソースのリアルタイムＯＳである TOPPERS/OSEK カーネル [12] を対象に，そのソースコードを直接修正することなく上記機能を実現するアスペクトを提案するとともに，実用上問題ないオーバヘッドで実装可能であることを示す．

2. アスペクト指向プログラミングによるリアルタイムＯＳのカスタマイズ

我々は，OSEK OS 仕様に基づくリアルタイムＯＳである TOPPERS/OSEK カーネルを対象にカスタマイズを行う．TOPPERS/OSEK カーネルの大部分は C 言語で記述されているため，我々は C 言語ベースのアスペクト指向言語 ACC（AspeCt-oriented C）[13], [14] を用いることとした．ACC は AspectJ[15] や Aspect C++と同様な，ジョインポイントモデルに基づくアスペクト指向言語である．

ACC で扱えるジョインポイントは，関数の呼び出し（call），関数の実行（execution），変数への値の書き込み（set），変数からの値の読み出し（get）である．アスペクトはポイントカット（pointcut）とアドバイス（advice）から成る．ポイントカットはジョインポイントの集合を指定するもので，アドバイスはポイントカットに合致するジョインポイントで実行する処理を記述したものである．ACC は before，after，around の３つのアドバイスをサポートしており，対象ジョインポイントの前後，あるいはジョインポイントの代わりにアドバイスコードを実行することができる．

ACC はトランスレータとして実装されており，ACC および C のソースファイルを入力し，織り込みを行った後，C のソースファイルを出力する．出力されたファイルを C コンパイラによりコンパイルすることでオブジェクトコードを生成できる．

図 1 に，ACC を用いた TOPPERS/OSEK カーネルのカスタマイズの流れを示す．TOPPERS/OSEK カーネルのソースファイルと，カスタマイズのためのアスペクトのソースファイルおよび C ソースファイルを ACC コンパイラ（トランスレータ）に入力することで，カスタマイズされたリアルタイムＯＳの C ソースファイルが得られる．それを C コンパイラでコンパイルすることで，カスタマイズされたリアルタイムＯＳのオブジェクトファイルを得ることができる．

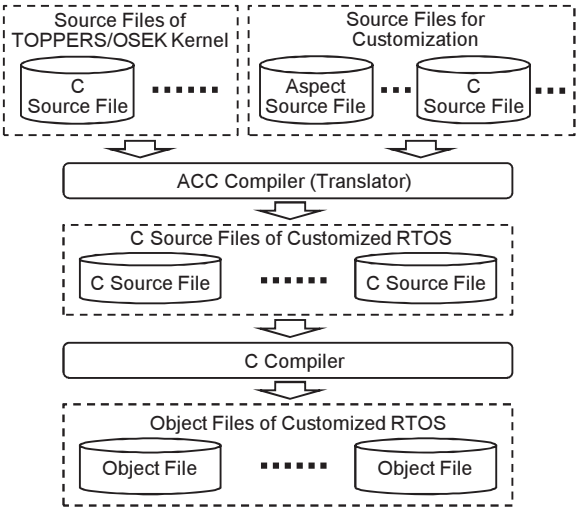

図 1 ACC を用いた TOPPERS/OSEK カーネルのカスタマイズ

3. 実現する分散リアルタイムＯＳ

3.1 分散リアルタイムＯＳの仕様

我々は既に，OSEK OS 仕様に基づく TOPPERS/OSEK カーネルを拡張し，自ノード上のタスクのみでなく他ノード上のタスクに対しても同一のシステムコールを用いたタスク管理やイベント制御が可能な分散リアルタイムＯＳを開発している [16]．本研究では，これと同一仕様の分散リアルタイムＯＳを，アスペクト指向プログラミングにより実現する．

OSEK OS 仕様では，タスク管理，イベント制御，リソース管理（排他制御），アラーム，割込み処理，ＯＳ実行制御，フックルーチンの７つのカテゴリのシステムコール（システムサービス）を規定している．我々が開発した分散リアルタイムＯＳでは，上記のうちタスク管理とイベント制御に関するシステムコールに位置透過性を持たせる拡張を行っている．位置透過性のあるシステムコールを提供することで，自ノード上のタスクに対しても他ノード上のタスクに対しても，同一のシステムコールを発行することができる．本論文では，自ノード上のタスクに対するシステムコールをローカルシステムコール，他ノード上のタスクに対するシステムコールを遠隔システムコールと呼ぶ．

OSEK OS のタスク管理とイベント制御に関するシステムコールを**表 1** に示す．引数の Task は対象タスクを指定するタスク ID，Event は設定するイベントマスク，TaskRef はタスク ID を格納する変数へのポインタ，StateRef はタスク状態を格納する変数へのポインタ，EventRef はイベントマスクを格納する変数へのポインタを表している．表 1 のタスク指定という欄には，引数で対象タスクの指定を行うかどうかを示している．タスク管理とイベント制御に関するシステムコールで引数でタスクを指定するのは，ActivateTask()，ChainTask()，GetTaskState()，SetEvent()，

表 1 タスク管理とイベント管理に関するシステムコール

分類	システムコール名	タスク指定
タスク管理	ActivateTask(Task)	○
	TerminateTask()	×
	ChainTask(Task)	○
	Schedule()	×
	GetTaskID(TaskRef)	×
	GetTaskState(Task, StateRef)	○
イベント管理	SetEvent(Task, Event)	○
	ClearEvent(Event)	×
	GetEvent(Task, EventRef)	○
	WaitEvent(Event)	×

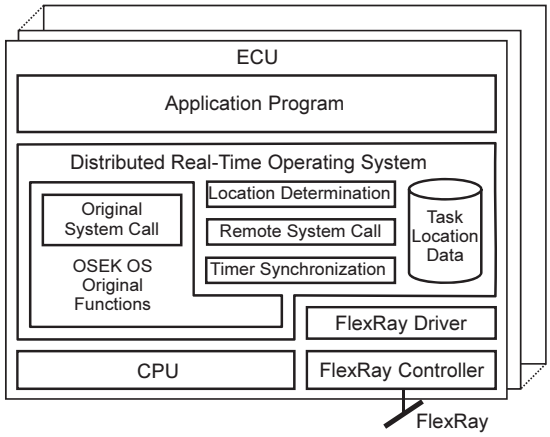

図 2 分散リアルタイムＯＳの構成

GetEvent() の 5 つである．そこで，これら 5 つを位置透過性のあるシステムコールに拡張し，他ノード上のタスクを指定した遠隔システムコールを可能とする．

なお，OSEK OS 仕様ではタスク ID のデータ長は 1Byte と規定されているが，分散システム全体でタスクを一意に指定するには 1Byte では不十分である．そこで本分散リアルタイムＯＳでは，従来のタスク ID を拡張し，システム全体で一意にタスクを指定できる 2Byte 長のグローバルタスク ID を定義することとした．そして，上位 1Byte をノードを識別するノード ID，下位 1Byte をノード内でタスクを識別するローカルタスク ID として使用する．

3.2 分散リアルタイムＯＳの構成と動作

分散リアルタイムＯＳの構成を**図 2** に示す．本分散リアルタイムＯＳは，TDMA（Time Division Multiple Access）方式に基づくリアルタイムネットワークである FlexRay[17] により接続された分散システムを対象としている．FlexRay は，一定の通信サイクル（Communication Cycle）に従って，周期的に通信を行う．各ノードの FlexRay コントローラは，全 FlexRay コントローラ間で同期したネットワーク時刻に基づいて，同期した通信を行う．

図 2 に示すように，本分散リアルタイムＯＳは，OSEK OS 本来の機能（OSEK OS Original Functions）に，時刻同期（Timer Synchronization）機能，タスク位置判定（Location Determination）機能，遠隔システムコール（Remote Procedure Call）機能を追加することで実現する．また，OSEK 仕様におけるコンフィギュレーション情報を記述する OIL（OSEK Implementation Language）[18] を，複数ノードの情報を記述できるように拡張するとともに，その OIL 記述からタスク位置情報（Task Location Data）を生成できるように SG（System Generator）を拡張する．そして，生成したタスク位置情報を分散リアルタイムＯＳ内に記憶する．

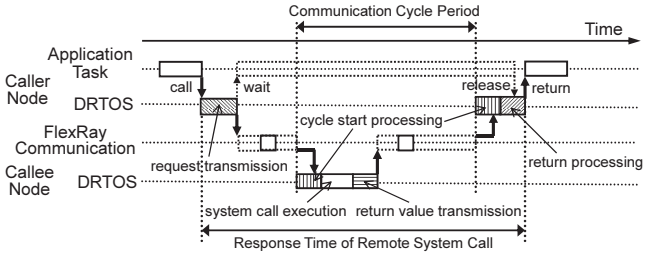

図 3 遠隔システムコールのタイムチャート

本分散リアルタイムＯＳでは，FlexRay のネットワーク時刻を分散リアルタイムＯＳに供給することで，全ノード上で同一のグローバル時刻に基づいたタスク管理を可能とする．時刻同期機能は，ネットワーク時刻を元にグローバル時刻を実現するための機能であり，FlexRay の毎通信サイクル開始時に，時刻同期の処理を行う．

タスク位置判定機能と遠隔システムコール機能は，位置透過性のあるシステムコールを実現するために必要な機能である．タスク位置判定機能は，タスク位置情報を参照して，システムコールの対象タスクがどのノードに存在するかを判定する．対象タスクが自ノード上に存在する場合は，OSEK OS 本来のシステムコール（Original System Call）を実行する．対象タスクが他ノード上に存在する場合は，遠隔システムコール機能を実行する．

遠隔システムコール機能は，対象タスクが存在するノードに処理要求を送信するとともに，当該ノードからの返信を受信し，戻り値やパラメータを要求元のタスクに渡す．また，他ノードから処理要求を受信した場合は，要求されたシステムコールを実行し，返信として戻り値やパラメータを遠隔システムコールの要求元ノードに送信する．

遠隔システムコールのタイムチャートを**図 3** に示す．図 3 は，呼び出し元ノード（Caller Node）上のアプリケーションタスク（Application Task）が，呼び出し先ノード（Callee Node）上のタスクに対する遠隔システムコールを発行してから，戻り値を受け取って処理を再開するまでを表している．

前述のように，FlexRay による通信（FlexRay Communication）は一定の通信サイクルに従って周期的に実行される．本分散リアルタイムＯＳでは，FlexRay のダイナミッ

クセグメントを用いて，遠隔システムコールの要求メッセージおよび返信メッセージの転送を行う．また，要求メッセージおよび返信メッセージの読み出しは，FlexRayの毎通信サイクル開始時に行う．毎通信サイクル開始時に行う時刻同期処理と受信メッセージ読み出し処理をまとめて，サイクル開始処理（cycle start processing）と呼ぶことにする．サイクル開始処理は，通信サイクル開始時に起動される割り込み処理（ISR: Interrupt Service Routine）により実行される．この割り込み処理は，OSEK OS 仕様で定義されているカテゴリ 2 の ISR である．

図 3 において，呼び出し元ノード上のアプリケーションタスクがシステムコールを発行し，分散リアルタイムＯＳ（DRTOS）のタスク位置判定機能により対象タスクが他ノード上にあると判定されると，遠隔システムコール機能は要求メッセージを生成して呼び出し先ノードに送信し，アプリケーションタスクを待ち状態に遷移させる（request transmission）．

呼び出し先ノードでは，FlexRay 通信により要求メッセージが転送された次のサイクル開始処理で，受信した要求メッセージの内容を解読し，対応するシステムコールの処理を実行する（system call execution）．そして，システムコールの実行後，戻り値やパラメータを含む返信メッセージを生成し，呼び出し元ノードに送信する（return value transmission）．

呼び出し元ノードでは，FlexRay 通信により返信メッセージが転送された次のサイクル開始処理で，返信メッセージから戻り値やパラメータを取り出した後，アプリケーションの待ち状態を解除する（return processing）．待ち状態を解除されたアプリケーションタスクは，戻り値を受け取って処理を再開する．

4. アスペクト指向プログラミングによる分散リアルタイムＯＳの実現

4.1 遠隔システムコール実現に必要な処理

3 章で述べた位置透過性のあるシステムコールを実現するための処理の流れを**図 4** に示す．呼び出し元ノード上のアプリケーションタスクが API を呼び出すと，タスク位置情報を参照してタスク位置判定処理（Location Determination）を行う．タスク位置判定処理により，対象タスクが自ノード上にあると判定された場合は，オリジナルのシステムコール処理（Original System Call）を呼び出す．

タスク位置判定処理で対象タスクが他ノード上にあると判定された場合は，要求送信処理（Request Transmission）を実行する．要求送信処理は要求メッセージを生成して，FlexRay ドライバを呼び出してメッセージ送信を要求するとともに，アプリケーションタスクを待ち状態に遷移させる．

送信要求を依頼された FlexRay ドライバは，送信するメッセージ本体にヘッダ情報等を付加したフレームを生成し，FlexRay コントローラのメッセージ RAM に書き込む．メッセージの転送は FlexRay コントローラによりなされる．メッセージ転送が完了すると，受信ノード（呼び出し先ノード）の FlexRay コントローラのメッセージ RAM に要求メッセージが格納される．

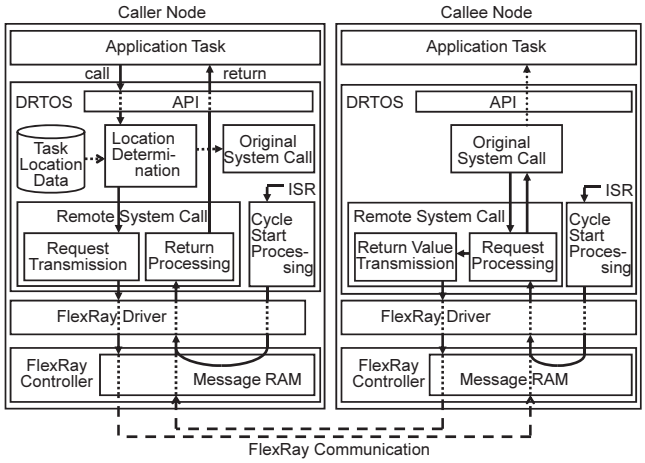

図 4 位置透過性のあるシステムコールを実現するための処理

呼び出し先ノードでは，FlexRay の通信サイクル開始時に起動される ISR によりサイクル開始処理（Cycle Start Processing）を実行する．サイクル開始処理では，時刻同期の処理とともに，メッセージ RAM に格納されたメッセージのチェックを行う．要求メッセージを受信した場合は，それを読み出して要求解読処理（Request Processing）を行い，要求されたシステムコールを実行する．

システムコールの実行後，戻り値送信処理（Return Value Transmission）を実行する．戻り値送信処理は返信メッセージを生成し，FlexRay ドライバを呼び出してメッセージ送信を要求する．FlexRay ドライバは，メッセージ本体にヘッダ情報等を付加したフレームを生成し，FlexRay コントローラのメッセージ RAM に書き込む．

返信メッセージを受信した呼び出し元ノードでは，ISR により実行されるサイクル開始処理がメッセージ RAM に格納された返信メッセージを読み出した後，リターン処理（Return Processing）を行う．リターン処理は，返信メッセージから戻り値やパラメータを取り出して返信データ用バッファに格納し，アプリケーションタスクの待ち状態を解除する．待ち状態を解除されたタスクは，返信用バッファの値を読み出して処理を再開する．

以上の処理において，リアルタイムＯＳに新規に追加する必要があるものは，タスク位置判定処理，要求送信処理，サイクル開始処理，要求解読処理，戻り値送信処理，リターン処理である．これらのうち，ISR で実行されるサイクル開始処理，要求解読処理，戻り値送信処理，リターン処理についてはアスペクトで織り込む必要はなく，新規 ISR を追加することで実現できる．

```
/* Original source code of ActivateTask */
StatusType ActivateTask(TaskType tskid) {
  StatusType ercd = E_OK;
  LOG_ACTTSK_ENTER(ercd);  /* macro for logging */
  CHECK_CALLLEVEL(TCL_TASK | TCL__ISR2);
                    /* macro to check call level*/
  CHECK_TSKID(tskid);  /* macro to check task ID */

  lock_cpu();  /* disable interrupts */
  . . . . .
  shift the state of the task to ready and
              enqueue the task to the ready queue
  call the dispatcher
              if the task has the highest priority
  . . . . .
  if an error occurs
    goto d_error_exit;

exit:
  unlock_cpu();  /* enable interrupts */
  LOG_ACTTSK_LEAVE(ercd);  /* macro for logging */
  return(ercd);

error_exit:
  lock_cpu();  /* disable interrupts */
d_error_exit:
  _errorhook_par1.tskid = tskid;
  call_errorhook(ercd, OSServiceId_ActivateTask);
                            /* error hook routine */
  goto exit;
}
```

図 5 オリジナルの ActivateTask() のソースコード

アスペクトにより織り込む必要があるのは，タスク位置確認処理と要求送信処理の２つである．これらの処理の実行をアスペクトで記述し，システムコールの呼び出し処理に織り込む．なお，オリジナルのシステムコールを呼び出す際には，2Byte 長のグローバルタスク ID を，OSEK 本来の 1Byte 長のタスク ID（ローカルタスク ID）に変換して引数として渡す必要がある．

4.2 処理の織り込み

アスペクト指向プログラミングを用いて位置透過性のあるシステムコールを実現するには，ActivateTask()，ChainTask()，GetTaskState()，SetEvent()，GetEvent() の５つのシステムコールに対して，タスク位置確認処理と要求送信処理を織り込む必要がある．

オリジナルのシステムコールのソースコードを**図 5** に示す．図 5 はシステムコール ActivateTask() の場合のソースコードで，一部は省略または簡略化している．

まず最初に，マクロ関数によるエラーチェックを行う（ロギングのためのマクロはデバッグ用で，通常は何も行わない）．図 5 のコードでは，コールレベルのチェック（呼び出し元がタスクあるいはカテゴリー２の ISR かをチェック）と，タスク ID のチェック（存在しないタスク ID を指定していないかのチェック）を行っている．エラーチェック用のマクロ関数の形式を**図 6** に示す．エラーを検出した場合は，goto 文により，図 5 中のラベル error_exit にジャンプする．

その後，lock_cpu() により割り込みを禁止し，システム

```
/* Original source code for error checking */
#define CHECK_*****(arg) do {                ¥
  if ( an error detected ) {                 ¥
    ercd = error code;                       ¥
    goto error_exit;                         ¥
  }                                          ¥
} while (0)
```

図 6 エラーチェックマクロのソースコード

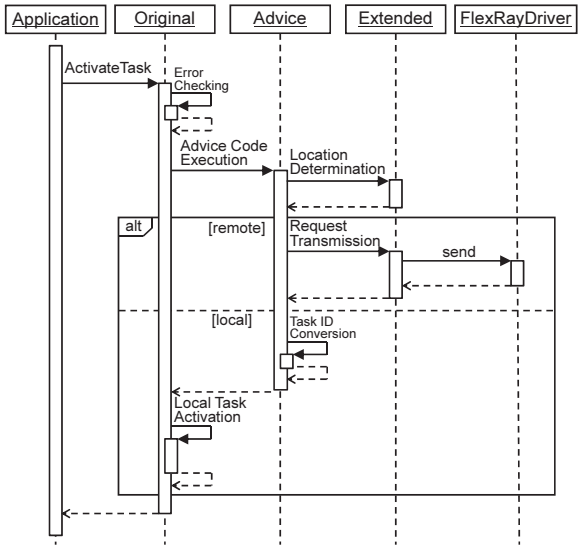

図 7 システムコール内部への織り込み

コール本来の処理を行う．ActivateTask() の場合は，タスクを実行可能状態に遷移させるとともにレディキューに挿入し，起動したタスクが最高優先順位の場合はディスパッチャを呼び出す．この時点でエラーが発生した場合は，ラベル d_error_exit にジャンプし，エラーフックルーチンが定義されていれば，それを実行する．最後に unlock_cpu() により割り込み禁止を解除し，リターンする．

他の４つのシステムコールも，処理の内容は異なるものの，ほぼ同様の形式のソースコードとなっている．

位置透過性のあるシステムコールをアスペクト指向プログラミングにより実現する場合，できるだけ効率的な実装が望まれる．そこでまず，システムコール内部に織り込みを行うことを考える．

ActivateTask() の場合を例に，システムコール内部にタスク位置確認処理と要求送信処理を織り込む場合の処理の流れを**図 7** のシーケンス図に示す．図では，オリジナルのシステムコールの処理を Original，アスペクトのアドバイスに記述する処理を Advice，位置透過性のあるシステムコールを実装するために追加する処理を Extended で表している．また，エラーチェック処理（Error Checking）は記載しているが，エラー処理自体は省略している．

システムコールの処理を開始すると，エラーチェック後にアドバイスの処理を実行し，タスク位置判定処理（Location Determination）を呼び出す．対象タスクが他ノードに存在する（remote）と判定された場合は，要求送信処

理（Request Transmission）を呼び出し，要求送信処理はFlexRay ドライバに要求メッセージの送信を依頼する．対象タスクが自ノードに存在する（local）と判定された場合は，グローバルタスク ID をローカルタスク ID に変換して（Task ID Conversion），オリジナルの処理であるタスクの起動を行う（Local Task Activation）．

以上の処理を実現するには，エラーチェックを行っているマクロの実行後，あるいは割り込み禁止を行っている関数 lock_cpu() の呼び出し前に，タスク位置判定処理を行うとともに，対象タスクが他ノード上にあると判定された場合は，要求送信処理を行った後，オリジナルの処理をスキップしてラベル exit にジャンプする処理を含む if 文を織り込む必要がある．ところが，アスペクト指向言語 ACC を使用した場合，このような織り込みを行うアスペクトを記述することはできない．

ACC では，マクロ関数をジョインポイントとして扱うことができないため，エラーチェック用のマクロ関数をポイントカットで指定することはできない．また，関数 lock_cpu() の呼び出しが複数存在するが，ACC にはそれらを区別する手段がないため，エラーチェック直後の lock_cpu() の呼び出しのみをポイントカットで指定することができない．さらに，アドバイスに記述した処理から，ベースとなるソースコードのラベルに goto 文でジャンプすることもできない．

そこで本研究では，システムコールの呼び出しあるいは実行をジョインポイントとすることで，位置透過性のあるシステムコールを実現することとする．また，対象とした分散リアルタイム O S では 2Byte に拡張したグローバルタスク ID が定義され，オリジナルのシステムコールとは引数の型が異なることから，システムコールの呼び出しをジョインポイントとする．

ActivateTask() の場合を例に，システムコールの呼び出しに織り込む場合の処理の流れを**図 8** に示す．アプリケーションがシステムコールを呼び出すと，アドバイスで記述された処理を実行する．グローバルタスク ID のエラーチェックを行った後，タスク位置判定処理を呼び出す．対象タスクが他ノードにあると判定された場合は，オリジナルと同様のエラーチェックを行った後，要求送信処理を呼び出す．シーケンス図には明記していないが，エラーを検出した場合はエラー処理にジャンプする．対象タスクが自ノード上にあると判定された場合は，グローバルタスク ID をローカルタスク ID に変換して，オリジナルのシステムコール ActivateTask() を呼び出す．

4.3 アスペクトの記述

図 8 に示した処理の流れを実現するアスペクトを ACC により記述する．具体的には，対象とするシステムコールを call ポイントカットで指定し，around アドバイスにより前節で述べた処理を記述する．

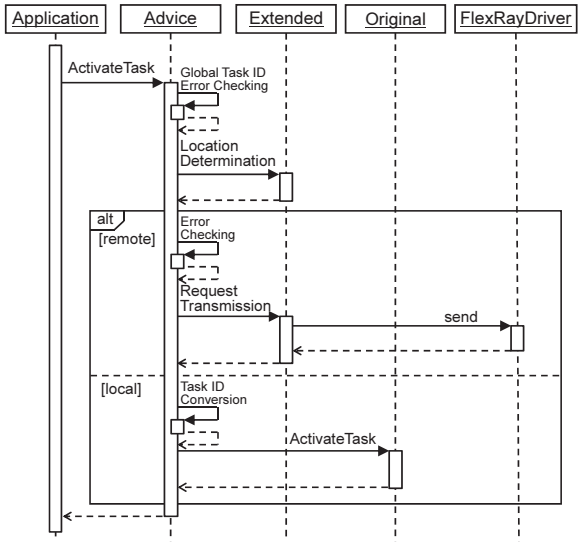

図 8 システムコールの呼び出しへの織り込み

```
/* Aspect for ActivateTask */
StatusType around(GlbTaskType glbtskid):
    call(StatusType ActivateTask(GlbTaskType))
                              && args(glbtskid) {
  StatusType ercd = E_OK;
  TaskType tskid = 0;
  CHECK_TSKID(glbtskid);
  if (Check_target_location(glbtskid)) {
                          /* Location Determination */
    CHECK_CALLLEVEL(TCL_TASK | TCL__ISR2);
    lock_cpu();
    global_syscal_flag = TRUE;
    reqest_sending_service(glbtskid,
         OSServiceId_ActivateTask, 0xFFFFFFFF);
                           /* Request Transmission */
    ercd = (UINT8)(get_return_value);
    if (ercd == E_OK) {
      unlock_cpu();
      goto exit;
    } else {
      goto d_error_exit;
    }
  }
  tskid = (taskType)(glbtskid & 0x00FF);
                             /* Task ID Conversion */
  ercd = ActivateTask(tskid);
                      /* call original system call */
exit:
  LOG_ACTTSK_LEAVE(ercd);
  return(ercd);

error_exit:
  lock_cpu();
d_error_exit:
  tskid = (TaskType)(glbtskid & 0x00FF);
  _errorhook_par1.tskid = tskid;
  call_errorhook(ercd, OSServiceId_ActivateTask);
  goto exit;
}
```

図 9 ActivateTask() のためのアスペクト

ActivateTask() の場合のアスペクトの記述を**図 9** に示す．call ポイントカットにより ActivateTask() を指定するとともに，args ポイントカットで引数 glbtskid（グローバルタスク ID）を取り出す．そして around アドバイスにより，ActivateTask() の呼び出しを，アドバイス内に記述した一連の処理に置き換える．

アドバイスに記述した処理では，マクロ関数

表 2 実験環境

項目	型式・仕様
CPU	V850E/PHO3 クロック:128MHz
内蔵メモリ	ROM:992kB RAM:92kB
FlexRay コントローラ	オンチップ クロック:80MHz
FlexRay 通信	伝送速度:10MHz 通信サイクル:2ms
ベース OS	TOPPERS/OSEK kernel ver.1.1
アスペクト指向言語	ACC ver.0.9

CHECK_TSKID() でグローバルタスク ID のエラーチェックを行う．エラーを検出した場合は，ラベル error_exit にジャンプする．エラーでない場合は，タスク位置判定処理を行う関数 check_target_location() を呼び出す．check_target_location() は，対象タスクが他ノード上にある場合は true，自ノード上にある場合は false を返す．

対象タスクが他ノード上にあると判定された場合は，CHECK_CALLLEVEL() でコールレベルのチェックを行い，エラーでなければ，lock_cpu() により割り込みを禁止する．そして，遠隔システムコール発行中を表すフラグを立て，関数 request_sending_service() を呼び出して要求送信処理を行う．その後 get_return_value により遠隔システムコールの戻り値を受け取り，戻り値が E_OK（正常）であれば，unlock_cpu() で割り込み禁止を解除し，ラベル exit にジャンプし，戻り値をリターンする．戻り値がエラーを示している場合は，ラベル d_error_exit にジャンプする．

対象タスクが自ノード上にあると判定された場合は，グローバルタスク ID の下位 1Byte を取り出してローカルタスク ID に変換し，オリジナルのシステムコール ActivateTask() を呼び出し，その戻り値をリターンする．

他の 4 つのシステムコール，ChainTask()，GetTaskState()，SetEvent()，GetEvent() のためのアスペクトも，ActivateTask() の場合と同様の形で記述できる．

5. 実装および評価

5.1 実験環境

4 章で提案したアスペクトを用いて，位置透過性を実現するためのタスク位置判定処理と要求送信処理を，TOPPERS/OSEK カーネルに織り込むとともに，ISR で実行するサイクル開始処理，要求解読処理，戻り値送信処理，リターン処理を追加することで，分散リアルタイムＯＳを実装した．

実装に使用した実験環境を**表 2** に示す．ハードウェアとして，FlexRay コントローラを内蔵したマイクロコントローラ V850E/PHO3 を搭載した評価ボードを使用した．ベースとして使用したリアルタイムＯＳは，TOPPERS/OSEK カーネル Version 1.1 を V850E/PHO3 に移植したものを用いた．アスペクト指向言語には ACC Version 0.9 を用いた．

表 3 要求送信処理実行時間（単位:μsec）

システムコール	アスペクト指向による分散 RTOS	直接書き換えによる分散 RTOS
ActivateTask	10.4	10.3
ChainTask	10.5	10.5
GetTaskState	10.5	10.5
SetEvent	10.5	10.4
GetEvent	10.5	10.5

5.2 評価

アスペクト指向プログラミングによるオーバヘッドを評価するため，アスペクト指向により拡張した 5 つのシステムコールについて実行時間を測定し，ソースコードを直接書き換えて実装した同一仕様の分散リアルタイムＯＳ [16] の実行時間と比較する．実行時間の計測にはクロック 32MHz のハードウェアタイマを用いた．

まず，遠隔システムコールのオーバヘッドを評価するため，要求送信処理時間を計測した．ここで要求送信処理時間と呼んでいるのは，アプリケーションプログラムがシステムコールを発行してから，タスク位置判定を行い，要求メッセージを生成し，FlexRay コントローラのメッセージ RAM に格納するまでの時間である．

アスペクト指向プログラミングにより実装した分散リアルタイムＯＳと，ソースコードを直接書き換えて実装した分散リアルタイムＯＳについて，要求送信処理時間の測定結果を**表 3** に示す．表に示した値は 50 回計測した平均値である．アスペクト指向プログラミングにより実装した場合の要求送信処理の時間は，直接書き換えて実装した場合とほぼ同じであり，無視できる程度のオーバヘッドになっている．

次に，ローカルシステムコールのオーバヘッドを評価するため，システムコールを発行してからリターンするまでの時間を計測した．アスペクト指向プログラミングにより実装した分散リアルタイムＯＳとソースコードを直接書き換えて実装した分散リアルタイムＯＳの他，オリジナルの TOPPERS/OSEK カーネルの実行時間も測定した．

ローカルシステムコールの実行時間の測定結果を**表 4** に示す．この場合も 50 回計測した平均値を示している．アスペクト指向プログラミングにより実装した場合の実行時間は，直接書き換えて実装した場合に比べ，最大で 30%程度増大している．ただ，その増加量は絶対値で 1μ 秒以下であり，実用上はそれほど大きな問題にはならないものと考えている．

アスペクト指向プログラミングにより実装すると，アドバイスコードを呼び出すための関数呼び出しが発生するため，直接書き換えて実装した場合と比較して，関数呼び出し 1 回分のオーバヘッドが発生する．ただし，遠隔システムコールの要求送信処理については，FlexRay コントローラのメッセージ RAM への格納処理の呼び出しをアドバイ

表 4 ローカルシステムコール実行時間（単位:μsec）

システムコール	アスペクト指向による分散 RTOS	直接書き換えによる分散 RTOS	TOPPERS/OSEK カーネル
ActivateTask	5.6	4.7	4.5
ChainTask	6.2	5.7	5.3
GetTaskState	3.0	2.3	1.9
SetEvent	8.2	7.4	7.0
GetEvent	8.3	7.4	7.1

ス内で行うことで，関数呼び出しの回数を相殺し，関数呼び出し回数を直接書き換えた場合と同一にするができ，オーバヘッドを削減できた．

一方ローカルシステムコールの場合は，関数呼び出し 1 回分のオーバヘッドはそのまま表れる．また，オリジナルの TOPPERS/OSEK カーネルのシステムコールと比較すると，グローバルタスク ID のエラーチェック処理，タスク位置判定処理，タスク ID 変換処理のオーバヘッドが発生する．なお ChainTask() については，オリジナルのシステムコール内でディスパッチが行われ，リターン処理を行わないため，他のシステムコールに比較してオーバヘッドはやや小さくなる．

6. おわりに

本論文では，アスペクト指向プログラミングにより分散化の機能を追加することで，既存のリアルタイムＯＳのソースコードを直接修正することなく，分散リアルタイムＯＳを実現する手法について述べた．具体的には，自動車制御分野向けの標準リアルタイム OS である OSEK OS を対象に，位置透過性のあるタスク管理やイベント制御が可能なシステムコールを実現するためのアスペクトを提案した．そして，提案したアスペクトを OSEK OS 仕様に基づく TOPPERS/OSEK カーネルに適用し，実用上問題ないオーバヘッドで分散リアルタイムＯＳを実現可能であることを示した．

謝辞

本研究で使用した TOPPERS/OSEK カーネルの開発者と ACC の開発者に感謝する．

参考文献

[1] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J. M. and Irwin, J.: Aspect-Oriented Programming, *Proceedings of the 11th European Conference on Object-Oriented Programming*, pp.220–242 (1997).

[2] Beuche, D., Fröhlich, A. A., Reinhard, M., Papajewski, H., Schön F., Schröder-Preikschat, W., Spinczyk, O. and Spinczyk, U.: On Architecture Transparency in Operating Systems, *Proceedings of the 9th Workshop on ACM SIGOPS European Workshop: Beyond the PC: New Challenges for the Operating system,* pp.147–152 (2000).

[3] Spinczyk, O. and Lohmann, D.: Using AOP to Develop Architecture-Neutral Operating System Components, *Proceedings of the 11th workshop on ACM SIGOPS European workshop Article,* No.34 (2004).

[4] Spinczyk, O., Gal, A. and Schröder-Preikschat, W.: Aspect C++: An Aspect-Oriented Extension to the C++ Programming Language, *Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems*, pp.53–59 (2002).

[5] Lohmann, D., Scheler, F., Tartler, R., Spinczyk, O. and Schröder-Preikschat, W.: A Quantitative Analysis of Aspects in the eCos Kernel, *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems,* pp.191–204 (2006).

[6] Lohmann, D., Hofer, W., Schröder-Preikschat, W. and Spinczyk, O.: Aspect-Aware Operating System Development, *Proceedings of the 10th International Conference on Aspect-Oriented Software Development 2011,* pp.69–80 (2011).

[7] Lohmann, D., Spinczyk, O., Hofer, W. and Schröder-Preikschat, W: The Aspect-Aware Design and Implementation of the CiAO Operating-System Family, *Transactions on Aspect-Oriented Software Development IX,* Lecture Notes in Computer Science Vol.7271, pp 168–215 (2012).

[8] Afonso, F., Silva, C., Montenegro, S. and Tavares, A.: Applying Aspects to a Real-Time Embedded Operating System, *Proceedings of the 6th Workshop on Aspects, Components, and Patterns for Infrastructure Software,* Article No.1 (2007).

[9] Park J. and Hong, S.: Building a Customizable Embedded Operating System with Fine-Grained Joinpoints Using the AOX Programming Environment, *Proceedings of the 2009 ACM symposium on Applied Computing,* pp.1952-1956 (2009).

[10] Abe, K., Yoo, M. and Yokoyama, T.: Aspect-Oriented Customization of the Scheduling Algorithm and the Resource Access Protocol of a Real-Time Operating System, *Proceedings of the IEEE 16th International Conference on Computational Science and Engineering,* pp.627–634 (2013).

[11] OSEK/VDX: *Operating System, Version 2.2.3* (2005).

[12] TOPPERS Project, http://www.toppers.jp/

[13] Gong, M., Zhang, C. and Jacobsen, H.-A.: Systems Development with AspeCt-oriented C (ACC), *Connections 2007 (ECE Graduate Symposium, University of Toronto),* Talk 5.6 (2007).

[14] AspeCt-oriented C, https://sites.google.com/a/gapp.msrg.utoronto.ca/aspectc/

[15] Kiczales, G., Hilsdale, E., Hugonin, J., Kersten, M. Palm J. and Griswold, W. G.:An Overview of AspectJ, *Proceedings of the 15th European Conference on Object-Oriented Programming*, pp.327–353 (2001).

[16] 知場貴洋，齊藤政典，伊丹悠一，兪明連，横山孝典：位置等価性のあるシステムコールを有する組み込み制御システム向け分散リアルタイムＯＳ，情報処理学会論文誌，Vol.53，No.12，pp.2702–2714 (2012).

[17] Makowitz, R. and Temple, C.: FlexRay - A Communication Network for Automotive Control Systems, *Proceedings of 2006 IEEE International Workshop on Factory Communication Systems*, pp.207–212 (2006).

[18] OSEK VDX, *System Generation OIL: OSEK Implementation Language Version 2.5* (2004).