システムソフトウェアと
オペレーティング・システム 92－19
（２００３．２．２８）

オブジェクト指向による組込み用JavaVMの設計

駿藤 浩之　　並木 美太郎

東京農工大学大学院工学研究科

近年，高機能化した組込みシステムに対応しようとJavaに期待が集まっている．しかし，組込みシステムにJavaを使用するためにはリアルタイム処理への対応を考えなければならない．

そこで，本研究では組込み用OSとJavaVMを一体化した組込みシステムのプラットフォームを提案する．従来は下層にOS層があり，アプリケーションとしてJavaVMがあるという構造であった．この構造ではJavaVMはOSが提供するシステムコールインタフェイスを使用し構築される．本研究で提案する一体化した組込みシステムのプラットフォームはOS層の中にJavaVMが入っている．

提案する方式の利点はリアルタイム処理への対応である．層を往来するオーバーヘッドを削減でき，スケジューラ，スレッド管理などはOSとJavaVMが管理，提供しているので，応答性に優れている．リアルタイム処理についてはJavaスレッドとOSのネイティブタスクが1対1に対応しているので，Javaスレッドは OSのネイティブタスクとしてリアルタイムスケジューリングができる．JavaVMのガーベッジコレクタは三色モデルを使用し，周期スレッドで対応する．提案する方式はオブジェクト指向の考え方が有効である．本稿ではこの方式の設計について述べる．

Design of an Embedded JavaVM with an Object-orientedApproach

Hiroyuki Sunto　　Mitaro Namiki

Graduate School of Technology ,
Tokyo University of Agriculture and Technology

In recent years, since the embedded system was having advanced features and we corresponded, Java Virtual Machines(JVMs) were expected. However, you have to consider the correspondence to a real-time processing in order to use Java with an embedded system. In this paper, we present a platform for an embedded system, an embedded-OS and JVMs are the same layers.

The present system was the structure where JVMs as application put OS layer to use. We present a platform for an enbedded system, constitutes OS with JVMs. The advantage of the system to propose is correspondence to a real-time processing. Since this system can cut down the overhead which comes and goes a layer and OS and JVMs manage and offer a scheduler, thread management, etc., it excels in the response. Since the native task of a Java thread and OS corresponds to one to one about the real-time processing, real-time scheduling can do a Java thread as a native task of OS. Since the native task of a Java thread and OS corresponds to one to one about the real-time processing, real-time scheduling can do a Java thread as a native task of OS. The system to propose has an effective object-oriented view.In this paper,we describe the design of this system.

1　はじめに

近年，高機能化している組込みシステムに Java を適用しようという試みが始まっている．しかし，Java を組込みシステムに適用する場合には次に示す問題点が挙げられている．

(1) リアルタイム処理
組込みシステムではリアルタイム処理を必要としていることが多い．しかし，Java のスケジューリングポリシはリアルタイム処理に対して考慮されていない．また，挙動が把握できないガーベッジコレクタ (以下 GC と記す) は割り込み応答性を低下させてしまう．

(2) ハードウェア資源の制限
組込みシステムではハードウェア資源の制限が厳しい．制限が厳しいので，動的コンパイラなどは適用が難しい．また，インタプリタによる実行では効率が良くない．

このような問題点を解決し，組込みシステムに Java を適用しなければならない．そこで，本研究では，組込み用 OS と JavaVM を一体化した組込みシステムのプラットフォーム『じゃいもん』を提案する．従来は下層に OS 層があり，アプリケーションとして JavaVM があるという構造であった．この構造では JavaVM は OS が提供するシステムコールインタフェイスを使用し構築される．本研究で提案する一体化した組込みシステムのプラットフォームは OS 層の中に JavaVM が入っている．この相違を図 1 に示す．一体化とは JavaVM は自

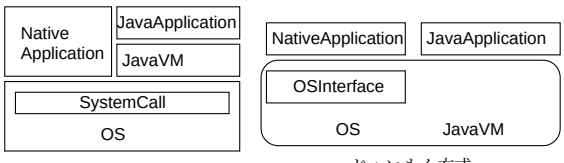

図 1 従来のシステムとの相違

由にハードウェアにアクセスすることができ，OS の機能を使用することができるので，オーバーヘッドが減少するのではないかと考えた．従来から組込みシステムでは特権モードで動作させることが多く，Java のベリファイア機能があるので，メモリ保護はできると考えている．

以下，第 2 章で関連研究について述べる．第 3 章で概要，第 4 章で OS 機能，第 5 章で JavaVM 機能，第 6 章で評価を述べる．

2　従来システムの問題点

JTRON [4] ではリアルタイム処理をネイティブタスクで実行し，グラフィックユーザインタフェイス (以下 GUI) などの他処理を Java スレッドで実行するハイブリッド構造を提案し，主にネイティブタスクと Java スレッドの同期を定義している．この方式では Java 実行環境を搭載していても Java 言語ではリアルタイム処理を記述できない．また，複雑な構造になり，ハードウェア資源を要求する．本研究では OS と JavaVM が一体化することで，Java 言語でリアルタイム処理を記述ができ，コンパクトになる．

JeRTyVM [6] では J-Consortium [2] の Core 仕様準拠である．ガーベッジコレクタとスケジューリングを改良することで対応している．この方式では Java を使用し，リアルタイム処理を実行することができる．リアルタイム処理には対応しているが，ハードウェア資源の制約に関しては考慮されていない．この方式はリアルタイム OS を使用することが前提になっているので，必ず OS のシステムコール (サービスコール) を使用するので，オーバーヘッドが大きい．本研究では OS と JavaVM を一体化することでオーバーヘッドなく構築することができる．

3　組込み用 JavaVM『じゃいもん』の概要

3.1　目標と方針

第 1 章で示した問題点に対して，組込み用 OS と JavaVM を一体化することによって解決できるのではないかと考えた．組込みシステムではアプリケーションと一体になっているシステムも多いので，組込み用 OS と JavaVM を一体化することにより，軽量かつ，高効率なシステムになり，リアルタイム性を発揮できると考えている．目標としては多様な外部割込みデバイスに対応できるように割り込み応答性を 100 μsec 以下に設定する．

目標を受けて設計方針を次のようにする．

(1) OS，JavaVM を階層化せず，一体化する
プログラムの共通化と関数コールでアクセスすることができるので，実行性能が上がり，ハードウェア資源も有効に活用できるようになる．

(2) オブジェクト指向技術を用いた設計，実装をする
OS と JavaVM を一体化するためにはオブジェクト指向技術を用いて設計，実装することが有効であると考えている．最小単位で部品化することにより，OS と JavaVM というソフトウェアを組み上げる．このとき OS と JavaVM を一体としてインタフェイスを共通化できる．

(3) 組込み用 OS『開聞』をベースとする
筆者の所属している研究室で開発が進められている組込み用 OS『開聞』はユーザがどのカーネル機能を使用するかどうかを選択することができる．本研究ではカーネル機能は実質上，割り込み管理だけの使用となるので，資源管理を JavaVM と融合するために『開聞』は適切と考えた．

3.2 全体構成

図 2 に全体構成を示す．この図のなかで，各機

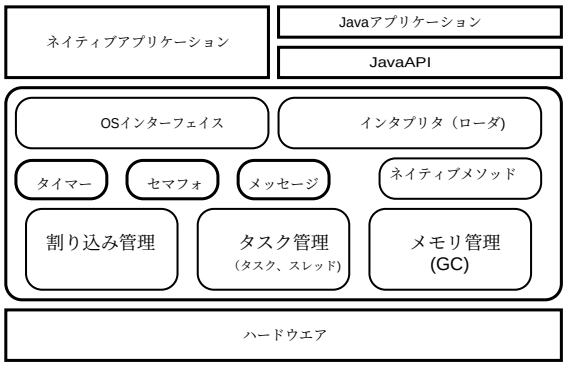

図 2 『じゃいもん』の全体構成

能は 1 個から複数のオブジェクト (インスタンス) で構成されている．図 3 に jyaimonClass を中心にクラス関係図を示す．

3.3 OS インタフェイス

オブジェクト指向で設計し，構築するとシステムが小規模ならばよいが，大規模になると複雑になり，保守性が低下してしまう．そこで，OS インタフェイス部設計した．OS インタフェイス部は各オブジェクトを集約している．ネイティブ (C 言語など) アプリケーションを構築する場合にはこの OS インタフェイス部を使用することにより，容易にアプリケーションを構築することができる．OS インタフェイスは従来のシステムコールではないので，オーバーヘッドは小さい．この OS インタフェイス部はソフトウェア資産を有効に活用するために組込み用 OS『開聞』のインタフェイスと統一する．

3.4 タスク管理部

タスク管理部は CPU を仮想化する．『じゃいもん』の実行単位であるタスクはタスク単位で管理するために一つのタスクで一つのオブジェクトである．タスク管理部はタスクオブジェクト，スレッドオブジェクト, スケジュールオブジェクトが複数ある．リアルタイムタスクはタスクから派生する作り方をする方法もあるが，共通項が多く，逆に保守が難しいと判断してリアルタイムタスクと通常のタスクは同じタスクオブジェクトの状態により区別している．スケジュールオブジェクトのインタフェイスは同じであるが，アルゴリズムが違う．デッドラインスケジューリングと優先度スケジューリングがあるが，他のスケジューリングを増やすことも可能である．どのスケジューリングを選択するかはタスク，スレッドの状態により判断される．スケジューラはタスクもスレッドも区別せずにスケジューリングされる．

3.5 メモリ管理部

メモリ管理部はメモリの割り当てと開放を基本的に受け持つ．仮想記憶ではないので，決まっているヒープ領域から増やすことはできない．Java では GC を必要としている．GC はマークスイープ方式を採用している．途中で停止できるように三色モデルを基本としている．ネイティブアプリケーションがメモリ管理部を使用するときには開放しなければならない．

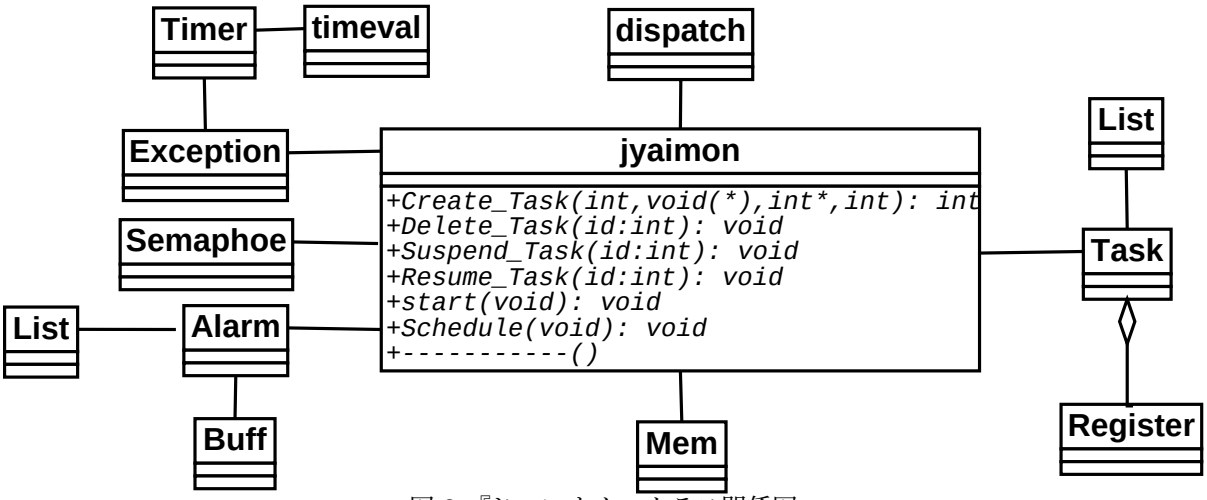

図 3 『じゃいもん』クラス関係図

3.6 インタプリタ部

インタプリタ部は Java クラスファイルをロードし，解釈実行する部分である．本システムではオブジェクト数が一番多い．インタプリタ部は他のタスク管理部，メモリ管理部などに処理を依頼することで進めていく，JavaVM のコアである．

4 『じゃいもん』OS

OS 機能として必ず必要なのがハードウェアの仮想化である．CPU を仮想化するタスク管理機能，割り込みを仮想化する割り込み管理機能は必ず必要になってくる．また，リアルタイムシステムを想定しているのでタイマを仮想化するタイマ機能は必須である．JavaVM で GC 機能が必要なので，低水準のメモリ管理機能が必要になってくる．オブジェクト指向設計なので，各ハードウェアを仮想化しているものはオブジェクトである．しかし，システムが大きくなってくるとこのオブジェクト間の依存関係が複雑になり，管理が難しくなる．そこで，OS インタフェイス部を作成する．

4.1 タスク管理

タスク管理機能は CPU を仮想化する．基本的な状態管理は組込み用 OS『開聞』と同様である [7]．しかし，タスク管理機能は JavaVM からも操作できるようにすることでオーバーヘッドを削減できるようにしたいので，オブジェクト指向設計を取り入れて，一つのタスクは一つのオブジェクトとして管理する．オブジェクト化することで各タスクに対する操作は各オブジェクトを操作することでタスク機能を果たすことができる．Java スレッドを構築するためにはネイティブタスクも Java スレッドも同様に扱えるほうが良い．そこで，Java スレッドはタスクを継承して構築する．このようにすることで，Java スレッドはネイティブタスクとしての機能を持つことになる．OS としてみると Java スレッドとネイティブタスクの違いはなく，リアルタイムスケジューリングなどもネイティブタスクとしてスケジューリングされる．Java スレッドもネイティブタスクの機能を持っているので問題なくスケジューリングされる．

4.2 割り込み管理

割り込み管理は割り込みを仮想化する．具体的な割り込み管理とはコンテキスト保存，復元，割り込み要因の特定である．組込み用 OS『開聞』ではこの処理をユーザに開放し，割り込み管理機能を使用しない選択も与えていた．組込み用 JavaVM『じゃいもん』もネイティブプログラムならば可能である．割り込み管理は外部割込みを仮想化しているが，内部割込みはコンテキスト保存，復元処理を必要なときに提供する．例えば，タスク生成処理は必要ないので，コンテキスト保存，復元処理は実行しない．このようにすることで，オーバーヘッドを削減する．

4.3 メモリ管理

メモリ管理機能はメモリを仮想化している．組込みシステムということから仮想記憶機能は持っていない．また，ネイティブタスクを使用するシステムでは基本的に静的にメモリを確保しているので，ネイティブタスクが使用することはあまりない．GC にメモリ機能を提供している．このメモリ管理機能はファーストフィットを基本的な戦略に使用している．基本的にメモリ割り当てには O(n) の時間がかかる．しかし，割り当て予定メモリを用意することで，極端に大きい領域でなければ一定時間の割り当てが可能である．メモリ開放はメモリリストを用意することで O(1) のオーダで可能である．割り当てと開放時間，また，メモリブロックのサイズにはトレードオフになっているが，GC を考慮した場合には開放時間を短縮することが組込みシステムではよいのではないかと考えている．GC が必要になるときにはメモリが不足しているときである．このとき，メモリを開放しなければならないと考えると開放時間を短縮するほうが全体的にプラスになると考えている．

4.4 OS インタフェイス部

オブジェクト指向で OS 機能を設計するとオブジェクト間のメッセージなどが複雑になり，管理が難しい．通常，各オブジェクトへメッセージを送信することで処理を進めるが，一つのオブジェクトで処理を完了させると各オブジェクト間に依存関係ができてしまうので，ユーザは一つの処理を進めるために複数のオブジェクトへ処理を依頼しなければならないという問題を解決するために各オブジェクトを管理するための OS インタフェイス部を設計した．次にインタフェイスを示す．初期化などはすべて OS インタフェイス部が行う．図 4 に OS の初期化処理例を示す．

```
jyaimon os;          /*インタフェイス部を宣言*/
void main(){
  os.init();         /*初期化処理*/
  os.start();        /*アイドルタスクの起動*/
  /*ここにくることはない*/
}
void jyaimon_start(){ /*アイドルタスクのエントリ*/
  /*処理するプログラムを記述*/
}
```

図 4 組込み用 OS のプログラム例

5 『じゃいもん』JavaVM

Java クラスファイルを読み込み解釈実行することからインタプリタ機能は必ず必要になってくる．このインタプリタ機能を動作させるために，Java クラスを管理するためのクラス管理機能，オブジェクト管理機能，GC 機能など多くの機能が必要である．このような機能は JavaVM 独自の仕様によるものなので，Java を動作させるためには構築しなければならない．従来は OS が機能を提供し，構築していた Java スレッド管理機能，タイマ機能，イベント管理機能なども必要と考えられる．しかし，OS 層のなかにあり，オブジェクト単位で管理されているので，JavaVM は直接管理することができる．

5.1 クラス管理機能

Java クラスをローダから読み込み，そのクラスを管理する機能が必要になってくる．通常の JavaVM ではシステムに複数のローダが存在することが可能である．しかし，組込みシステムでは複数のローダはハードウェアの制限もあり，必要ないのではないかと考えた．そこで本システムではローダはシステムに一つ存在だけである．このようにすることで，各ローダ単位のクラス管理は必要がなくなる．クラスを管理する必要はあるので，ハッシュ法を使用したクラステーブルを定義している．ハッシュ法にすることで探索時間を一定にすることができる．オブジェクト指向なので，ハッシュ法を使用したクラステーブルのクラスを定義している．ハッシュ法では衝突を起こすことがある．テーブルの大きさを小さくできるほうが組込みシステムでは有効と考え，衝突した場合にはチェーン法で回避する．先に探索時間を一定にすると述べたが厳密にいうと衝突した場合には線形時間かかる．

5.2 オブジェクト管理機能

Java で実際に扱われるのはオブジェクトである．JavaVM ではこのオブジェクトを内部で保持しなければならない．この Object クラスは Class クラスと Instance クラスへの参照を保持している．Class クラスは Java クラス情報を保持している構

造体である．Instance クラスはインスタンス変数を保持している構造体である．クラス情報とインスタンス変数情報は必ず必要である．また，メソッドテーブルを保持する例もよくあるが，メソッド情報はクラス情報から取得する．このため線形時間はかかる．しかし，クラス情報のほかにメソッドテーブルを持つのは大きな負担であり，メソッドテーブルを生成する負担も大きい．static 変数は Instance クラスが参照値を持っている．

5.3 Java スレッド管理機構

Java スレッドはネイティブタスクと同じように管理されている．Java スレッドもネイティブタスクと同じことから生成時にはサスペンド状態になっている．start() メソッドを起動すると起床する．この時点では通常のタスクである．リアルタイムタスクにするにはリアルタイム状態をセットしなければならない．

```
[クラス]
public class RealTimeThread extends Thread
[インタフェイス]
public RealTimeThread()
引数なし，戻り値なし．
public RealTimeThread()
[引数]int priority:優先度, 戻り値なし
[機能]
スレッドを生成する
[インタフェイス]
public void SetDeadLine(int deadlinetime)
[引数]  int  deadlinetime：  デッドライン時間
[機能]
デッドラインを設定する
```

RealTimeThread() コンストラクタでスレッドは生成される．このときの状態は Thread クラスで生成したときと同じである．リアルタイムシステムでは優先度順スケジューラで十分に性能を発揮できる場合があるので，コンストラクタには優先度を持たせた方が良いと考えた．また，RealTimeThread クラスを作成して，Thread クラスを継承するほうが従来のクラスとの整合性がとれると考えた．また，最終的にはネイティブメソッドを呼び出すのでネイティブタスクと同様になる．図**??**プログラミング例を示す．

```
public class RealTimeThread extends Thread{
    int deadlinetime;
    public RealTimeThread(){
      Create_Task();
   }
       public RealTimeThread(int priority){
      Create_Task(priority);
   }
public void SetDeadLine(int deadlinetime){
    this.deadlinetime = deadlinetime;
    Set_Dead_Line(deadlinetime);
  }
  native private  void  Create_Task();
  native private  void  Create_Task(int priority);
  native private  void  Set_Dead_Line(int time);
}
public class RTThread extends RealTimeThread{
  public static void main(String args[]){
   RealTimeThread rt = new RealTimeThread();
    //各処理
    rt.start();
       //各処理
  }
  public void run(){
    rt.SetDeadLine(100);
    //スレッドの処理
  }
}
```

図 5 プログラミング例

6 実現と評価

6.1 『じゃいもん』OS の実現

『じゃいもん』OS は組込み用 OS『開聞』をベースとして実装を行った．組込み用 OS と JavaVM を一体化した組込みシステムのプラットフォームを設計する場合にオブジェクト指向の考え方が有効である．オブジェクト指向ではデータや操作のカプセル化やクラスの継承など手続き指向にはない効果を得ることが可能である．各ハードウェアをオブジェクトとして仮想化し，各オブジェクトは操作，状態を保持しているので，その状態に対応して操作することができる．一例としては，Java スレッドやネイティブタスクへの操作は共通しているものも多く，同じインタフェイスを提供する．同じインタフェイスを提供していても各オブジェクト (Java スレッド，ネイティブタスク) に対して操作をすることができる．また，機能を追加する場合でもオブジェクト単位で分離しているので，容易に実行することができる．このように考え,『開

聞』をベースとしながらもオブジェクト指向で設計を行った.『開聞』の機能をオブジェクト指向 OS で実現した．システムが大きくなるとイベントの種類や数が複雑かつ増加し，適切な操作を選択することが困難になるという問題があったため,『開聞』ではなかった，OS インタフェイス部を設計し，実装した．OS インタフェイス部はイベントや操作を仮想化することで適切な操作をし，各オブジェクトを管理する．このようにすることで，各オブジェクトは他のオブジェクトに依存することなしに基本的な機能を提供することでシステムを構築することができる．

6.2 『じゃいもん』JavaVM の実現

組込み用 OS と一体化するための JavaVM を設計した．組込み用 OS に合わせるため，GC 以外はフルスクラッチで実装した．各部の実現状況を次に示す.

(1) インタプリタ部
実際に解釈実行しているところであり，実現した.

(2) クラス管理機能
クラス情報を保持しているところであり，実現した.

(3) オブジェクト管理機能
オブジェクトを保持管理しているところである．継承，インタフェイスなどをデバック中である.

(4) スレッド管理機能
スレッドを管理しているところである．マルチスレッド対応をデバック中である.

(5) ガーベッジコレクタ機能
Kaffe [1]から移植し，デバック中である.

(6) クラスローダ機能
クラスファイルを読み込むところであり，実現した.

本システムの性能を評価するため，次の実験を行った．実験環境を表 1 に示す.

[1]http://www.kaffe.org/

(1) スレッド生成，破棄，ディスパッチの時間計測
組込み用 OS と JavaVM を一体化している『じゃいもん』と通常の OS 層とユーザ層が分かれている場合を比較する．本システムは組込み用 OS と JavaVM を一体化しているシステムで，ネイティブタスクと Java スレッドは 1 対 1 に対応している．このシステムを評価するために OS 層とアプリケーション層が分かれているシステムと比較する．比較するシステムは SH3(60MHz) を使用したシステムなので，正規化して比較する．また，二通りのスレッド構築方式を実験した．一つはネイティブタスクと Java スレッドが 1 対 1 に対応しているネイティブスレッド方式で，もう一つは一つのタスクと複数の Java スレッドを対応させているグリーンスレッド方式である．この本システムを含めた 3 通りの方法を比較評価した．スレッド生成の結果を表 2 に示す．表 2 の結果より，本システム方式のオーバーヘッドが小さいことがわかる.

表 2 スレッド生成の計測結果

方式	最悪値 / 最善値
本システム	1.5 μ秒/1.8 μ秒
ネイティブスレッド方式	3.0 μ秒/4.2 μ秒
グリーンスレッド方式	2.7 μ秒/3.0 μ秒

表 3 にスレッド破棄の計測結果を示す．表 3 の結果より，本システムのオーバーヘッドは小さいことがわかる．また，本システムのディスパッチ時間は 1.2μsec であった.

表 3 スレッド破棄の計測結果

方式	最悪値 / 最善値
本システム	0.6 μ秒/0.6 μ秒
ネイティブスレッド方式	1.5 μ秒/1.5 μ秒
グリーンスレッド方式	0.6 μ秒/0.6 μ秒

(2) Java プログラムの時間計測
JavaVM 単体の性能を評価するため，スレッド機能を使用しないプログラムの性能を評価

表 1 実験環境

ターゲットボード	CQ RISC 評価用キット SH4
クロスコンパイラ	GCC 3.2
開発機 OS	linux 2.4.18
ターゲットプロセッサ	SH7750 200MHz
ターゲットボード搭載メモリ	4M バイト

表 4 比較実験環境

ターゲット	PC/AT 互換機
OS	Linux 2.4.18
プロセッサ	Pentium MMX200 MHz
Java 実行環境	JDK1.4(-Djava.compiler=NONE オプション)

する．JavaVM 単体の性能を比較するために JDK1.4 を使用した．JDK1.4 の環境を表 4 に示す．実験プログラムを図 6 に示す．

```
import java.util.*;
public class timetest{
  public static void main(String[] args){
    int count = 10000000;
    int j = 0;
    long time2 = System.currentTimeMillis();
    for(int i=0; i<count;i++){
        if(i % count == 0) j++;
    }
    long time1 = System.currentTimeMillis()-time2 ;
    System.out.println("Time="+time1+"usec");
  }
}
```

図 6 実験プログラム

この実験プログラムを各 10 回動作させ，平均値を計測した．この結果を表 5 に示す．表 5

表 5 プログラム実行結果

方式	平均時間
本システム	7192m 秒
JDK1.4	7031m 秒

の結果から JDK1.4 のほうが性能が良いと言える．しかし，本システムのサイズは 128KB に対して JDK1.4 は数 MB ということを考慮すると十分であると考えている．

7 おわりに

本稿では，組込み用 OS と JavaVM を一体化した組込みシステムプラットフォームの設計について述べた．OS 層とアプリケーション層の往来のオーバーヘッドを削減することで，効率を上げ，リアルタイムシステムに対応できる．メモリ保護については Java のベリファイアで対応することで問題はないと考えている．今後の課題としてデバイスドライバなどに対応できるクラスの作成が挙げられる．

参考文献

[1] J-Consortium,http://www.j-consortium.org/

[2] 奥山, 瀧本, 芝, 大久保，リアルタイムオペレーティングシステム Easel における JavaVM の設計と実装, 情報処理学会研究報告, 2002-OS-89, pp.87-93, 2002.

[3] トロン協会，http://www.tron.org

[4] The Real-Time Specification for JavaTM, http://www.rtj.org

[5] 乾,Java のリアルタイム化，共立出版，bit, Vol30，No.5，pp.61-67,1998.

[6] 萱嶋, 並木, 組込み用 OS『開聞』の割込み管理機構, 情報処理学会研究報告, 2000-OS-84, pp.47-54, 2000.