KVM ポーティングガイド

KVM 1.0

THE NETWORK IS THE COMPUTER™

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303 U.S.A.

2000 年 7 月 31 日
Revision A

目次

図目次

表目次

第1章

このマニュアルの内容

このマニュアルは、K Virtual Machine (KVM) の Sun Microsystems リファレンス実装を新しいプラットフォームに移植する方法について説明しています。

第2章

KVM の紹介

KVM (K Virtual Machine や KJava Virtual Machine とも呼ばれる) は、携帯電話、ポケベル、電子手帳、インターネット用の移動デバイス、POS 端末、家庭用電気製品など、リソースに制限のある小規模デバイスを対象としたコンパクトで移植可能な Java™ Virtual Machine です。

KVM の設計目標は、Java プログラミング言語の主要な機能をすべて維持し、かつ使用可能メモリがたった数 10K 〜 数 100K バイトしかないリソース制約型のデバイスでも動作する最小の「完結した」Java Virtural Machine を作成することでした (名称の K は K バイトを意味する)。具体的には、KVM は次のような仕様で設計されています。

- Virtual Machine コアの static メモリフットプリントを 40K 〜 80K バイト (ターゲットプラットフォームおよびコンパイルオプションによって異なる) に抑えた小規模システム
- クリーンで移植性が高い
- モジュール化およびカスタマイズが可能
- ほかの設計目標を犠牲にすることなく可能な限りの完全性と高速性を実現

KVM は C プログラミング言語で実装されているため、C コンパイラを使用できるさまざまなプラットフォームに簡単に移植できます。この Virtual Machine は、シンプルなバイトコードインタプリタを中心に構築されていて、移植作業やスペースを最大限に利用するために役立つさまざまなコンパイル時に使用できるフラグとオプションを備えています。

KVM は、コンシューマデバイスや組み込み型デバイスに移植可能で、動的にダウンロードでき、かつ安全性の高いアプリケーションを開発し、使用するためのスケーラブルなモジュール型アーキテクチャを提供する大規模なプログラムの一環として開発されました。このプログラムは、Java 2 Micro Edition と呼ばれています (Java 2 ME または J2ME とも呼ばれる)。

KVM および Java 2 Micro Edition の詳細は、Sun Microsystems, Inc.、Java Community Process の『The K Virtual Machine (KVM), A White Paper, KVM Technical Specification』と『Connected, Limited Device Configuration Specification』でそれぞれ参照できます。

KVM は、もともと Sun Microsystems Laboratories で開発された Spotless という研究システムから派生したものです。Spotless システムの詳細は、Sun Labs の技術レポート『The Spotless system: implementing a Java system for the Palm connected organizer』を参照してください。

第3章

コンパイラの要件

ANSI 準拠の C ファイルをコンパイルできる C コンパイラが必要です。このコンパイラでは、表 3-1 に示す C の基本型が定義されている必要があります。

表 3-1　基本型

型	説明
`char`	符号付きまたは符号なしの 8 ビット値
`signed char`	符号付き 8 ビット値
`unsigned char`	符号なし 8 ビット値
`short`	符号付き 16 ビット値
`unsigned short`	符号なし 16 ビット値
`int`	符号付き 16 ビット値または 32 ビット値
`unsigned int`	符号なし 16 ビット値または 32 ビット値
`long`	符号付き 32 ビット値
`unsigned long`	符号なし 32 ビット値
`void *`	32 ビットポインタ

J2ME 構成またはプロファイルが浮動小数点数をサポートしている場合は、コンパイラは表 3-2 に示す浮動小数点型をサポートするものでなければなりません。

表 3-2　浮動小数点型

型	説明
`float`	32 ビットの浮動小数点値
`double`	64 ビットの浮動小数点値

KVM の実装ではすべて、Java long[1] 型をサポートしているため、コンパイラが 64 ビット整数をサポートしていることが望ましいのですが、これは必須条件ではありません。Java long 型の移植については、第 9 章「64 ビットのサポート」で説明しています。

コンパイラには、次に示す書式の include を検索できるディレクトリを指定する何らかの方法が必要です。

```
#include <filename>
```

Sun のリファレンス実装のテストは、32 ビットポインタのマシンでしか行われていません。したがって、ほかのサイズのポインタを使用するプラットフォームで正常に動作するかどうかは不明です。

コードベースは、以下のコンパイラで正常にコンパイルされました。

- Metrowerks CodeWarrior Release 6 for Palm
- Solaris 上の Sun DevPro C Compiler 4.2
- Solaris 上の GNU C Compiler
- Windows 98 および Windows NT 4.0 上の Microsoft Visual C++ 6.0 Professional

ソースコードは、唯一の ANSI に準拠しない機能として 64 ビットの整数算術演算を使用します。

1. Java では、`long` 型は常に 64 ビットです。表 1 は、現在のほとんどの C 実装と同様に、long 型とは 32 ビット値を示しています。そのためこのマニュアルでは、64 ビットを意味する場合、「Java long 型」という表現を使用しています。

第4章

ディレクトリ構造

概要

KVM リリースを ZIP 解凍し、任意のディレクトリに入れてください。ZIP 解凍すると、次のサブディレクトリを持つディレクトリが作成されます。

- `api`
- `bin`
- `build`
- `docs`
- `jam`
- `kvm`
- `samples`
- `tools`

これらのサブディレクトリに含まれる内容を表 4-1 に示します。

表 4-1　KVM リリースのディレクトリ

サブディレクトリ	内容
`api`	このリリースに提供されている、Java ライブラリクラスのソースコード
`bin`	実行可能バイナリおよびコンパイル済みの Java ライブラリクラスすべて
`build`	KVM 構築用の Makefile
`doc`	マニュアルなどの文書

表 4-1　KVM リリースのディレクトリ

サブディレクトリ	内容
`jam`	KVM に付属しているオプションの Java Application Manager (JAM) コンポーネントのソースコード
`kvm`	KVM のソースコード
`samples`	多数のサンプルアプリケーションのソースコードとアイコン
`tools`	このリリースに提供されている多数のツール (JavaCodeCompact、プリベリファイア、Palm など) のソースコードとアイコン

ディレクトリ kvm/VmCommon

プラットフォームに依存しない共通の KVM ソースコードはすべて、ディレクトリ `kvm/VmCommon/src/` に置かれます。共通のインクルードファイルはすべて、ディレクトリ `kvm/VmCommon/h/` に置かれます。

ポート固有のソースファイルとインクルードファイルは、ディレクトリ `kvm/Vm`*Port*`/src/` と `kvm/Vm`*Port*`/h/` に置かれます (*Port* にはプラットフォーム名が入る。例: `kvm/VmWin`、`kvm/VmPilot`、`kvm/VmLinux`)。

ポートの中には、生成プロセスの一部 (ソースコード自体の一部ではない) であるファイルが格納される `kvm/Vm`*Port*`/build/` サブディレクトリを作成するものがあります。

表 4-2 に、`kvm/VmCommon/src/` と `kvm/VmCommon/h/` に含まれる KVM ソースコードファイルの概要を示します。

表 4-2　`VmCommon` 内のファイル

ファイル	内容
`StartJVM.c`	Virtual Machine の起動とコマンド行引数の読み取り
`cache.h` `cache.c`	メソッドルックアップの速度を上げるためのインラインキャッシングオペレーション
`class.h` `class.c`	Java クラスを表現するための実行時データ構造とオペレーション

表 4-2　VmCommon 内のファイル

ファイル	内容
events.h events.c	イベント処理のためのストリームベースのプロトコルの実装
fields.h fields.c	オブジェクトのフィールドを表現するための実行時データ構造とオペレーション
frame.h frame.c	スタックフレームと例外処理オペレーション
garbage.h garbage.c	ガベージコレクタとメモリ管理
global.h global.c	さまざまな大域変数
hashtable.h hashtable.c	Virtual Machine が内部的に使用するハッシュテーブル実装
interpret.h interpret.c	バイトコードインタプリタ
jar.h jarint.h jartables.h jar.c	JAR ファイルリーダ
loader.h loader.c	クラスローダ
log.h log.c	デバッグとプロファイリングのための記録 / 診断オペレーション
long.h	移植性を考慮した方法で 64 ビットオペレーションを処理するための特殊なマクロ
main.h	コンパイルフラグ、およびシステム全体に渡る定義
native.h native.c nativeCore.c	ネイティブ関数テーブルオペレーション、およびコアネイティブライブラリ関数
pool.h pool.c	定数プールを表現するための実行時データ構造とオペレーション
profiling.h profiling.c	Virtual Machine 実行のプロファイリングのためのデータ宣言とオペレーション
property.c	Java システムプロパティにアクセスするためのオペレーション

表 4-2 VmCommon 内のファイル

ファイル	内容
rom.h	ROM 化ツール (JavaCodeCompact) に必要なマクロ
runtime.h	一般にポートによってオーバーライドされる特定の実行時関数の定義
thread.h thread.c	Java スレッドの管理とマルチスレッド化のための実行時データ構造とオペレーション
verifier.h verifier.c	クラスファイルベリファイア (詳細は第 12 章を参照)

ディレクトリ kvm/VmExtra

ディレクトリ kvm/VmExtra/ には、多数のポートに有用な付加的なコンポーネントが含まれています。これらのファイルには、Spotless JVM (KVM の前身) から継承された Spotlet アプリケーションモデルのサポート、Windows で最も必要性の高いネットワークプロトコルの実装、新しい Generic Connection フレームワークを使用する実験的なストレージプロトコルの実装などがあります。

このディレクトリでは、非同期イベント処理用のマクロをいくつか定義しているほか、組み込み型でないターゲットプラットフォーム (Windows や Solaris など) で必要な低レベルのファイル操作と Virtual Machine の起動操作についても定義しています。

表 4-3 に、VmExtra に含まれるファイルの内容を示します。

表 4-3　VmExtra 内のファイル

ファイル	内容
async.h	非同期通知をサポートするためのマクロ (44 ページの「非同期ネイティブメソッド」と 51 ページの「非同期通知」を参照)
loaderFile.c	「実質的な」ファイルシステムを持つプラットフォームのためのファイルシステム、クラスローダおよび JAR リーダ間の低レベルバインディング
main.c	ファイルシステムを持ち、かつコマンド行からの VM 起動をサポートするプラットフォームのためのデフォルトのメインプログラム
nativeSpotlet.h nativeSpotlet.c	Spotless JVM (KVM の前身) から継承された Spotlet アプリケーションモデルのサポートに必要な低レベルのイベント処理コードとグラフィックスコード。KVM 内で必要なイベント処理オペレーションのほとんどは、現在別の機構を使用して行われている (event.c を参照)
network.c networkPrim.h networkPrim.c	Windows で最も広く使用されているネットワークプロトコルの実装
resource.c	外部リソースを読み取るためのストリームベースのプロトコルの実装
storage.h storage.c storagePrim.c	ストレージシステムに一般化された方法でアクセスするためのプロトコルの実験的な実装

第5章

各ポーティングに必要なファイルと関数

この章では、各ポーティングに定義すべきファイルと関数について説明します。

ファイル machine_md.h

各ポーティングは、Vm*Port*/h/machine_md.h という名前のファイルを提供する必要があります。このファイルの目的は、VmCommon/h/main.h に提供されているデフォルトのコンパイル時定義と宣言をオーバーライドし、個々のプラットフォームに必要な定義と宣言を追加することです。ポーティングでオーバーライドされることが多い定義と宣言の一覧は、第 6 章「コンパイルフラグ、定義、およびマクロ」を参照してください。

ポーティング固有の宣言、関数プロトタイプ、typedef 文、#include 文、および #define 文はすべて、この machine_md.h 内か、machine_md.h によって直接または間接的にインクルードされたファイル内、または開発環境[1] によって自動的にインクルードされたファイル内に必要です。あるいはコンパイラスイッチ[2] によって指定される必要があります。

ポーティング固有の関数は、任意のマシン固有ファイル内で使用できます。特に指定のないかぎり、必要となるポーティング固有の関数はマクロとしても定義できます。この場合、その実装は各引数の評価が必ず 1 度だけ行われるように注意する必要があります。

1. たとえば、Metrowerks では開発者は接頭辞ファイルを作成できます。
2. コンパイルの中には、スイッチ -Dname=value を追加できるものもあります。このスイッチは、ファイルの先頭で #define name value を指定するのと同じです。

ファイル main.c

通常は、ターゲットプラットフォームに適した新しいバージョンの main.c を提供する必要があります。ディレクトリ VmExtra/src/main.c に提供されているデフォルトの KVM 実装は、プラットフォーム固有の実装を作成する上での出発点です。詳細は、第 7 章「Virtual Machine の起動」を参照してください。

移植作業を要する実行時関数

各ポーティングは、以下の関数を定義する必要があります。これらは、マクロまたは C コードとして定義できます。通常は、C コードは Vm*Port*/src/runtime_md.c というファイルに置かれます。

- void AlertUser(const char* message)
  何か重大な問題が起きたことをユーザに警告します。この関数呼び出しは、通常、致命的エラーの前に発生します。
- cell *allocateHeap(long *sizeptr, void **heapPtrPtr)
  long の値 *sizeptr とほぼ同じサイズ (単位はバイト) のヒープを作成します。このヒープは、4 の倍数であるアドレスで始まる必要があります。ヒープのアドレスは、この関数の値として返されます。ヒープの実際のサイズ (単位はバイト) は、*sizeptr で返されます。*realresultptr に指定される値は、ヒープを解放する場合に freeHeap に対する引数として使用されます。
  ほとんどのポーティングでは、*heapPtrPtr はネイティブスペースの割り当て関数によって返される実際の値に設定されます。この値が 4 の倍数でない場合は 4 のその次の倍数に切り上げられ、*sizeptr から 4 が引かれます。
- void freeHeap(void *heapPtr)
  allocateHeap によって割り当てられたヒープスペースを解放します。heapPtr 引数の意味については、上記を参照してください。
- GetNextKVMEvent(KVMEventType *evt, bool_t forever, ulong64 waitUntil)
  この関数は、Virtual Machine のイベント処理機能とホストオペレーティングシステム間のインタフェースとしての役割を果たします。詳細は、第 11 章を参照してください。

- `void InitializeVM(int *argc, char **argv)`
  Virtual Machine を必要な任意の方法で初期化します。現在のポーティングの多くでは、これは何も行わないマクロです。
- `void InitializeNativeCode(int *argc, char **argv)`
  ネイティブコードを必要な任意の方法で初期化します。ポーティングは、この関数を使用して (たとえば) ウィンドウシステムを初期化し、ネイティブコード固有のほかの初期化を実行できます。
- `void FinalizeVM()`
  Virtual Machine を停止する前に、何かクリーンアップが必要であればそれを実行します。
- `void FinalizeNativeCode()`
  ネイティブ関数のあと、何かクリーンアップが必要であればそれを実行します。多くのポートは、この関数を使用してウィンドウシステムを停止します。
- `ulong64 CurrentTime_md(void)`
  1970 年 1 月 1 日 (UTC) 以降の経過時間をミリ秒単位で返します。タイムゾーンの概念をサポートしないデバイスでは、現在のタイムゾーンの 1970 年 1 月 1 日以降の時間をミリ秒単位で返すことが認められます。

関数 `InitializeVM` と `InitializeNatives` は、大域変数の設定およびメモリ管理システムの初期化の前にこの順序で呼び出されます。これらの各関数には、もともと `StartJVM()` 関数に指定された `argc` と `argv` を指すポインタが渡されています。これらの関数は、必要に応じて引数カウントと引数ベクトルを変更できます。

関数 `FinalizeVM()` は、`FinalizeNativeCode()` の直前に呼び出されます。プロファイリングを有効にしたポーティングでは、これらの 2 つの関数の呼び出しの間にプロファイリング情報が出力されます。このため、プロファイラは必要に応じてウィンドウシステムの情報を確認し、ウィンドウシステムを使用して出力を作成できます。

非同期ネイティブ関数: ポーティングが非同期ネイティブメソッドの使用をサポートしている場合は、定義すべきポーティング固有の関数がほかにもあります。これらを次に示します。

```
yield_md()
CallAsynchronousFunction_md()
enterSystemCriticalSection()
exitSystemCriticalSection()
```

これらの関数については、44 ページの「非同期ネイティブメソッド」で説明しています。

必要な C ライブラリ関数

KVM は、以下の C ライブラリ関数を使用します。

- 文字列操作: `strcat`、`strchr`、`strcmp`、`strcpy`、`strncpy`、`strlen`
- メモリの移動: `memcpy`、`memove`、`memset`、`memcmp`
- 出力: `atoi`、`sprintf`、`fprintf`、`putchar`
- 例外処理: `setjmp`、`longjmp` (必須ではない)

開発環境でこれらの関数を定義していない場合は、これらを独自に定義するか、あるいはマクロを使用して、開発環境が認識できる同等の関数にこれらの名前を対応付ける必要があります。[1]

関数 `memmove` は、出力元と出力先が重なる状況を処理できなければなりません。関数 `memcpy` は、出力元と出力先が重ならないことが明らかな場合だけ使用されます。

関数 `fprintf` と `sprintf` は、次の書式だけを使用します。

`%s`, `%d`, `%o`, `%x`, `%ld`, `%lo`, `%lx`, `%%`

これらの書式には、オプションもフラグも指定されません。

`printf` の直接の呼び出しは存在しません。

注 – ディレクトリ `VmExtra` に含まれるコンポーネント、このリリースに提供されているマシン固有のポーティング、およびオプションの Java Application Manager (JAM) コンポーネントには、上記以外のネイティブ関数が別途必要です。

---

1. 引数の順序は、プラットフォームによって異なることに注意してください。たとえば、関数 memset は、引数 memset(location, value, count) を取ります。対応する PalmOS 関数は、MemSet(location, count, value) となります。

第6章

コンパイルフラグ、定義、およびマクロ

この章では、VmCommon/h/main.h に定義されている各種の C プリプロセッサフラグ、定義、およびマクロを示します。この章と VmCommon/h/main.h に含まれる文章を読みこれらのフラグの意味を理解すると、移植作業を進めやすくなります。

注 – VmCommon/h/main.h 内の値は変更せずに、ポーティング固有の machine_md.h ファイル内でこれらの値をオーバーライドすることをお勧めします。
Sun のリファレンス実装では、これらのフラグの多くは一般に Makefile からオーバーライドされることにも注意してください。

以下の節では、定義ごとに概要とデフォルトの設定を示します。これらのフラグとマクロについては、VmCommon/h/main.h でも説明されています。

一般的なコンパイルオプション

以下の定義は、プラットフォームに依存した一般的なコンパイラオプションを制御します。これらのオプションは、移植作業を開始する前に設定する必要があります。設定が正しく行われないと、通常、Virtual Machine の誤動作が起きます。

```
#define COMPILER_SUPPORTS_LONG 1
```

コンパイラが long (64 ビット) 整数をサポートしている場合、このフラグをオンに設定してください。

```
#define NEED_LONG_ALIGNMENT 0
```

KVM に、ホストオペレーティングシステムとコンパイラは、通常、64 ビット整数はすべて 8 バイト境界に配置されると見なされることを通知します。

```
#define NEED_DOUBLE_ALIGNMENT 0
```

KVM に、ホストオペレーティングシステムとコンパイラは、通常、double の浮動小数点数はすべて 8 バイト境界に配置されると見なされることを通知します (このフラグは浮動小数点サポートが有効になっている場合だけ意味を持つ)。

その他の注意: コンパイラは、マシンのエンディアン形式がわかるとより有用なコードを生成します。このため、マシン固有のヘッダファイル内で、以下の 2 つの変数の一方を 1 に設定することをお勧めします。

```
#define BIG_ENDIAN 0
#define LITTLE_ENDIAN 0
```

コンパイラが 64 ビットの整数算術演算をサポートしている場合で、フラグを次のように設定したときは、型 `long64` と `ulong64` を定義する必要があります。

```
#define COMPILER_SUPPORTS_LONG 1
```

コンパイラが 64 ビット整数をサポートしていない場合 (あるいはほかの何らかの理由でフラグを 0 に設定した場合) は、これらの 2 つの型の構造定義は自動的に作成されます。詳細は、第 9 章を参照してください。

一般的なシステム構成オプション

以下の定義は、ポーティングにどのコンポーネントと機能を含めるかを制御するために使用します。

```
#define INCLUDE_ALL_CLASSES 1
```

ターゲットシステムに CLDC ではないクラスを含めるかどうかを指定します。このオプションを有効に設定すると、CLDC Specification に含まれないコンポーネント (グラフィックスサポート、ネットワークプロトコル実装など) も含まれます。

```
#define IMPLEMENTS_FLOAT 0
```

KVM における浮動小数点サポートを有効または無効に設定します。CLDC 準拠の実装では、無効にする必要があります。

```
#define USES_CLASSPATH 1
```

CLASSPATH サポートを有効または無効に設定します。このオプションが有効に設定される場合、KVM はホストシステムの環境変数 CLASSPATH を使用して Java クラスファイルの位置を確認します。

```
#define PATH_SEPARATOR ':'
```

CLASSPATH で使用されている区切り文字を渡します。この定義は、USES_CLASSPATH オプションを使用している場合だけ意味を持ちます。

```
#define ROMIZING 1
```

クラスのプリリンク / プリロード (JavaCodeCompact) サポートを有効または無効に設定します。このオプションが有効に設定される場合、KVM は Virtual Machine 内ですべてのシステムクラスのプリリンクを直接行い、アプリケーションの起動速度を著しく上げます。詳細は、第 13 章を参照してください。

```
#define USE_JAM 1
```

Virtual Machine にオプションの Java Application Manager (JAM) コンポーネントを含めるかどうかを指定します。詳細は、第 14 章を参照してください。

```
#define ASYNCHRONOUS_NATIVE_FUNCTIONS 0
```

KVM に、非同期ネイティブ関数が使用されるかどうかを通知します。詳細は、44 ページの「非同期ネイティブメソッド」と第 11 章を参照してください。

Palm 固有のシステム構成オプション

以下の定義は、Palm 固有のシステム構成オプションを制御するために使用します。これらの機能はすべてもともと KVM の Palm バージョンのために設計されたものですが、特定のポーティングの出発点として便利な場合があります。

```
#define USESTATIC 0
```

KVM に、変化しない特定の実行時データ構造を palm にとって最適な状態で使用するように指示します。すなわちこのデータ構造を「動的な RAM」から「ストレージ RAM」に変えて Java ヒープスペースを節約します。KVM の Windows バージョンおよび Solaris バージョンでは、(デバッグ目的で) この機構の擬似的な実装を利用できます。

```
#define CHUNKY_HEAP 0
```

KVM に、複数のチャンクまたはセグメントに Java ヒープを最適な状態で割り当てることができるように指示します。このオプションにより Virtual Machine は、特定のプラットフォーム (Palm など) に割り当てるヒープスペースを増やすことができます。

```
#define RELOCATABLE_ROM 0
```

KVM に、リンク前処理が施されたシステムクラスを再配置 (移動) 可能な表現を使用して、最適な状態で格納するように指示します。これにより、ROM 化ツール (JavaCodeCompact) を使用したシステムクラスを Palm などのデバイスに格納できるようになります。

メモリ割り当ての設定

以下の定義は、KVM が割り当てるメモリの量に作用します。

```
#define MAXIMUMHEAPSIZE 65024 /* 0xFE00 */
```

Virtual Machine の起動時に KVM が割り当てる Java ヒープサイズ

```
#define INLINECACHESIZE 40
```

ENABLEFASTBYTECODES オプションが有効に設定されている場合に Virtual Machine の起動時に KVM が予約する特殊なインラインキャッシュ領域のサイズ。このサイズは、インラインキャッシュエントリの数として指定します (各エントリはターゲットプラットフォームに応じて 12 〜 16 バイト必要)。

```
#define STACKCHUNKSIZE 64
```

KVM 内の Java スレッドの実行スタックは、必要に応じて自動的に拡大、縮小します。この値は、より大きなスペースが必要となる場合に新しいスタックフレームチャンクのデフォルトサイズを定義します。

```
#define STRINGBUFFERSIZE 512
```

Virtual Machine が特定の文字列操作で内部的に使用する、静的に割り当てられる領域のサイズ (単位はバイト)

注 – 原則として、KVM は必要なメモリをすべて Virtual Machine の起動時に割り当てます。実行時には、事前に割り当てられた領域内で全メモリが配分されます。もちろん、Virtual Machine が Java ヒープの外で動的なメモリ割り当てを実行するホストシステム固有のネイティブ関数 (グラフィックス関数など) を呼び出す場合には状況が変わります。

ガベージコレクションのオプション

以下のオプションは、ガベージコレクタの精度を上げるために使用できます。詳細は、`VmCommon/h/garbage.h` と `VmCommon/h/main.h` 内の文章を参照してください。

```
#define SAFEGARBAGECOLLECTION 1
#define TESTNONOBJECT_DEPTH 3
```

次のオプションをゼロ以外の値に設定すると、割り当てごとにガベージコレクションが発生します。これにより、ガベージコレクションの問題を見つけやすくなります。

```
#define EXCESSIVE_GARBAGE_COLLECTION 0
```

注 – KVM の今後のバージョンでは、新しいガベージコレクタが実装される予定です。上記のオプションは、新しいガベージコレクタが提供される時点で変更される可能性があります。

インタプリタ実行のオプション

以下のマクロは、インタプリタ実行を制御する特定の機能を有効または無効に設定するために使用します。以下の定義では、製品リリース時のデフォルト値を示しています。

`#define ENABLEFASTBYTECODES 1`

実行時のバイトコード置換とメソッドインラインキャッシングを、有効または無効に設定します。このオプションは、Virtual Machine のパフォーマンスをおよそ 15 〜 20% 向上させますが、Virtual Machine のサイズを数キロバイト増やします。バイトコード置換は、バイトコードが不揮発性メモリ (ROM など) に格納されるターゲットプラットフォームでは行えません。

`#define VERIFYCONSTANTPOOLINTEGRITY 1`

Virtual Machine に、定数プールルックアップを実行する場合、実行時に定数プールエントリの型を検証するように指示します。このオプションは実行時パフォーマンスをわずかに下げますが、通常は安全性とセキュリティの維持のために設定することをお勧めします。

その他の定義とインタプリタマクロ:

`#define BASETIMESLICE 500`

この変数の値は、Virtual Machine が実行するスレッド切り替えやイベント通知のような定期的に必要となるオペレーションの基本的な周期を (実行されるバイトコードの数として) 決定します。数を小さくするとイベント処理とスレッド切り替えの待ち時間が短縮されますが、インタプリタの実行が遅くなります。

`#define DOUBLE_REMAINDER(x, y) fmod(x,y)`

`interpret.h` で定義されるコンパイラマクロであり、2 つの浮動小数点数のモジュールを見つけるために使用されます。

`#define SLEEP_UNTIL(wakeupTime)`

このマクロは、現在の時刻 (関数 `CurrentTime_md()` の戻り値によって示される) が立ち上がり時刻と等しくなるか、あるいは立ち上がり時刻を過ぎる時点まで Virtual Machine をスリープ状態にします。`SLEEP_UNTIL` のデフォルトの実装は、ビジーループです。ほとんどのポーティングは、バッテリを節約するように通常はより効率的な実装を提供することが必要です。詳細は、56 ページの「バッテリ電力の節約」を参照してください。

デバッグオプションとトレースオプション

KVM には、移植作業を促進する多数の便利なデバッグ / トレース関数とオプションが付いています。製品のリリース時には、これらのオプションはすべて無効に設定されています。

デバッグコードの組み込みと除外

`#define SLEEP_UNTIL(wakeupTime)`

Virtual Machine を新しいプラットフォームに移植する場合に便利な大量のデバッグコードとロギングコードを含めます。

`#define ENABLEPROFILING 0`

Virtual Machine の実行監視と実行統計の取得を行うための特定のプロファイリング機能を有効または無効に設定します。このオプションを有効に設定すると、Virtual Machine の実行速度がかなり遅くなります。

トレースオプション

以下のオプションは、個々の実行トレースオプションと出力オプションを有効または無効に設定するために使用します。出力はすべて標準出力に送られます。これらのオプションの多くは、`INCLUDEDEBUGCODE` または `ENABLEPROFILING` オプション (あるいはこの両方) が有効に設定されることを必要とします。

```
#define TRACEMEMORYALLOCATION 0
#define TRACEGARBAGECOLLECTION 0
#define TRACEGARBAGECOLLECTIONVERBOSE 0
#define TRACECLASSLOADING 0
#define TRACECLASSLOADINGVERBOSE 0
#define TRACETHREADING 0
#define TRACEBYTECODES 0
#define TRACEMETHODCALLS 0
#define TRACEVERIFIER 0
#define TRACEEXCEPTIONS 0
#define TRACEEVENTS 0
#define TRACEMONITORS 0
#define TRACE_STACK_CHUNKS 0
#define TRACE_FRAMES 0
```

さらに、次のオプションを変更し、出力されるトレースメッセージの形式 (簡潔または詳細) を制御できます。

```
#define TERSE_MESSAGES 0
```

ネットワークオプションとストレージオプション (Generic Connection)

CLDC Specification は、ネットワーキング、ストレージ、リソース管理、およびその他の関連事項を効率の良い拡張可能な方法でサポートするための新しい Generic Connection フレームワークを定義しています。KVM は、以下のマクロを使用してこれらの機構の一部を内部的に利用できます。

```
#define GENERICEVENTS 1
```

汎用的な接続機構を使用して、KVM の内部にイベント処理を実装します。これはデフォルトの機構です (古いイベント処理コードは Virtual Machine から除外された)。

```
#define GENERICNETWORK 0
#define GENERICSTORAGE 0
```

汎用的な接続機構を使用して実装された特定のネットワーク機能とストレージ機能を Virtual Machine で利用できるようにします。

エラー処理マクロ

インタプリタは、図 6-1 に示す書式のコードを使用します。

「通常のコード」のどこかでマクロ `ERROR_THROW(int)` の呼び出しがある場合、VM は直ちにエラー処理コードにジャンプします。このマクロを使用すると、静的にあるいは動的に入れ子にすることができます。`ERROR_THROW` は、最も内側の `ERROR_CATCH` エラー処理コードにジャンプします。

```
ERROR_TRY {
  normal code
} ERROR_CATCH (error) {
  error handling code
} ERROR_END_CATCH
  always continue here
```

図 6-1　エラー処理

デフォルトでは、この動作は `setjmp` と `longjmp` を使用してエミュレートされます。しかし、類似した機構を既に提供しているプラットフォーム (PalmOS など) ではネイティブ機構を使用する必要があります。

各種のマクロとオプション

`#define UNUSEDPARAMETER(var)`

リファレンス実装の関数の中には、それらの関数自体が使用しない引数を取るものがあります。このような指定に対して警告を出力するコンパイラもあれば、警告を出力しないコンパイラもあります。警告を出力しないコンパイラが「変数を使用しないのは意図的であり警告を受けることを希望しない」ということを指定する方法はそれぞれ異なります。このマクロは、コンパイラの出力を中止させるために必要などんなことでも実施します。

第7章

Virtual Machine の起動

Virtual Machine の起動方法は、KVM ポーティングによって大幅に異なります。デフォルトでは KVM は Java Virtual Machine のコマンド行ベースの起動方法をサポートしていますが、コマンド行ベースの起動が適切でない環境ではその環境に適するように Virtual Machine を簡単に変更できます。

コマンド行からの起動

この節では、KVM をコマンド行から起動する場合の Virtual Machine の起動上の規則について説明します。

ファイル `VmExtra/src/main.c` には、`main()` のデフォルト実装が提供されています。Virtual Machine は、コマンド行から次のように呼び出されます。

```
kvm [option]* className [arg]*
```

各オプションは次のどれかです。

```
-debug
-verbose
-classpath <ディレクトリのリスト>
```

必須引数 `className` には、メソッド `static main(String argv[])` が呼び出されるクラスを指定します。このクラス名のあとに指定される引数はすべて解釈されていない文字列であり、単一の `String[]` オブジェクトとしてまとめられ、単一の引数として `main` メソッドに渡されます。

上記の <ディレクトリのリスト> は、ディレクトリを `PATH_SEPARATOR` 文字で区切った単一の文字列です。

`main(int argc, char **argv)` のデフォルト実装は、オプションがすべて除かれた `argv` と適切に減算された `argc` を使用して、関数 `StartJVM()` を呼び出します。

その他の起動方法

実装が Virtual Machine をコマンド行から起動しない場合は (グラフィカル環境を使用してアプリケーションを起動する場合など)、適切な引数を使用してコードが `StartJVM()` を呼び出すように調整する必要があります。

JAM (Java Application Manager) の使用

多くの KVM ポーティングは、デスクトップのオペレーティングシステムで一般に利用できる多くの機能 (コマンド行言語、グラフィカルファイルマネージャ、ファイルシステムなど) が欠如したリソース制約型のデバイス上で動作します。このようなプラットフォームへの KVM の移植を簡単にするため、KVM は JAM (Java Application Manager) という機能のリファレンス実装を提供しています。

JAM は、次のフラグを使用してコンパイルレベルで有効または無効に設定できます。

```
#define USE_JAM 0
```

JAM リファレンス実装では、Generic Connection フレームワークを使用して実装されたネットワークプロトコルまたはストレージプロトコルを使用すればアプリケーションを JAR ファイルとしてダウンロードできることになっています (詳細は CLDC Specification を参照)。JAM は、JAR ファイルの内容および関連する記述子ファイルを読み取り、メインクラスと共に KVM をパラメータとして起動します。

JAM は、ホストオペレーティングシステムと Virtual Machine 間のインタフェースとしての役割を果たします。このため、JAM はデバイス固有のグラフィカル Java アプリケーションの管理と起動を行う環境 (マイクロブラウザ) の出発点として使用できるほか、Virtual Machine をテストするテストハーネスとしても使用できます。JAM リファレンス実装は、JAM が Virtual Machine を毎回再起動することなく多数の Java アプリケーション (テストケースなど) を実行できる特殊な `-repeat` モードを提供します。

JAM の詳細は、第 14 章「Java Application Manager (JAM)」を参照してください。

第8章

クラスのロード

KVM コードには、クラスファイルをディレクトリ内のファイルや圧縮された JAR ファイル内のエントリとして読み取る実装が含まれています。

クラスファイルをロードするための別の方法を提供する必要がある場合は、独自のクラスローディング機構を定義してください。`VmExtra/src/loaderFile.c` 内のデフォルトの実装は、プラットフォーム固有の実装を作成する上での出発点として使用できます。

汎用インタフェース

開発者は、C 構造体 `filePointerStruct` を定義する必要があります。汎用コードは、以下の定義を使用します。

```
struct filePointerStruct;
typedef struct filePointerStruct *FILEPOINTER;
```

この場合、この構造体のフィールドについて何も認識しません。

開発者は、次の関数も定義する必要があります。

- `void initializeClassPath()`
  コードは、ファイルローディングに必要な `ClassPathTable` などの変数を初期化する必要があります。この関数は、前処理定数 `USES_CLASSPATH` がゼロ以外の値

(これがデフォルト) を持つ場合にだけ定義すればすみます。ClassPathTable 内の値はガベージコレクションのルートであり、NULL、またはヒープから割り当てられるオブジェクトにする必要があります。[1]

C プリプロセッサ定数 PATH_SEPARATOR は、クラスパス内のディレクトリを区切る文字を示します。この定数のデフォルト値は「:」です。Windows および Windows に類似したベースを持つ実装では、この値を「;」に変更する必要があります。

- FILEPOINTER openClassfile(const char *className)
  className という名前のクラスが入ったクラスファイルを開きます。変数 className は、パッケージの区切り文字としてスラッシュ (/) を使用した絶対パスによるクラス名です。

- unsigned char loadByte(FILEPOINTER ClassFile)
  unsigned short loadShort(FILEPOINTER ClassFile)
  unsigned long loadCell(FILEPOINTER ClassFile)
  クラスファイルから次の 1 バイトか 2 バイト、または 4 バイトを読み取り、結果を符号なしの 8 ビット値、16 ビット値、または 32 ビット値として返します。Java クラスファイルにおける 16 ビット値と 32 ビット値は、常にビッグエンディアン形式です。

- void loadBytes(FILEPOINTER ClassFile, char *buffer, int len)
  クラスファイルから次の len バイトをロードし、指定されたバッファに書き込みます。

- void skipBytes(FILEPOINTER ClassFile, unsigned int len)
  クラスファイル内の次の len バイトをスキップします。

- void closeClassfile(FILEPOINTER ClassFile)
  指定されたクラスファイルを閉じるとともに、そのクラスファイルに関連付けられたシステムリソース (ファイルハンドルやデータベースレコードなど) をすべて閉じます。

openClassFile によって返されるクラスファイル構造は、Java ヒープから割り当てられるオブジェクトである必要があります。

1. KVM の今後のリリースでは、この関数はより汎用的な initializeFileLoading() に置き換えられる可能性があります。その場合、シンボル USES_CLASSPATH は不要になります。そして、変数 ClassPathTable は汎用コードでだけ使用されるようになります。

JAR ファイルリーダ

KVM 実装は、圧縮された JAR ファイルからクラスファイルを読み取れることが必要です。JAR ファイルの位置は、実装に応じた方法で指定されます。

JAR ファイルからクラスファイルエントリとリソースの圧縮解除を行う関数は、`loaderFile.c` と `jar.c` に提供されています。

次の関数は、

```
bool_t findJARDirectories(FILE *jarFile,
                          unsigned long *localDirectory,
                          unsigned long *centralDirectory)
```

既に開かれている JAR ファイル内でローカルディレクトリとセントラルディレクトリを見つけようとしています。これらのディレクトリが見つかる場合には TRUE を返し、変数 `localDirectory` と `centralDirectory` を適切な値に設定します。見つからない場合には、FALSE を返します。この場合、開かれているファイルが JAR ファイルである可能性はほとんどありません。

次の関数は、

```
JAR_DataStreamPtr
loadJARfile(const char *jarFileName,
            unsigned long localDirectory,
            unsigned long centralDirectory,
                     const char *fileName)
```

`fileName` に指定されたファイルの圧縮解除コンテンツを含むデータ構造を返します。JAR ファイルにそのファイル名が見つからないか、そのファイルを検出することで何か問題が発生した場合には NULL を返します。

`loadJARfile` は、関数 `inflate()` を使用します。この関数を使用すると、「圧縮」アルゴリズムを使用して圧縮されたすべて (Zip ファイルエントリなど) を圧縮解除できます。

通常、`inflate()` は以下の引数を取ります。

```
bool_t inflate(FILE *compressedData,
               int   compressedLength,
               unsigned char *decompressedData
               int   decompressedLength)
```

- compressedData:
  圧縮されたデータの最初のバイトを読み取るために位置付けられるファイル
- compressedLength:
  圧縮されたデータの長さ
- decompressedData:
  結果を書き込むためのバッファ
- decompressedLength:
  圧縮解除後のデータの長さ

compressedLength と decompressedLength は、どちらも正確でなければなりません。圧縮されたデータが compressedLength に指定されたバイト長と同じでない、あるいは圧縮解除されたデータが decompressedLength に指定されたバイト長と同じでない場合、解凍アルゴリズムはエラーと見なします。

inflate() は、データの解凍に成功する場合は TRUE を返し、エラーを見つけた場合は FALSE を返します。inflate() が FALSE を返す場合、実際に使用するファイル内の compressedData の位置、および decompressedData の内容は不明です。

inflate() アルゴリズムは、ヒープからメモリを割り当てる場合があります。decompressedData 配列が Java ヒープから割り当てられている場合、呼び出し側はガベージコレクションから decompressedData 配列を保護するように処置を施す必要があります。フラグを次のように設定すると、

```
#define JAR_INFLATER_USES_STDIO 0
```

inflate() の最初の引数は次のようになります。

```
bool_t inflate(unsigned char *compressedData, . . . .)
```

この場合、圧縮解除されるバイトを含む配列を最初の引数として渡す必要があります。

このオプションは、入出力をサポートしないポーティングやダウンロードされたアプリケーションをメモリに格納するポーティングに有用です。この場合、呼び出し側はガベージコレクションから compressedData 配列を保護するように処置を施す必要があります。

第9章

64 ビットのサポート

プラットフォームが浮動小数点演算規格をサポートしている場合、コンパイラは適切なサポートを提供する必要があります。

コンパイラが 64 ビット演算規格をサポートすることは必須条件ではありませんが、64 ビット対応のコンパイラを使用すると移植がはるかに容易になります。

設定

コンパイラが 64 ビット整数をサポートする場合: プラットフォームに応じたインクルードファイルの 1 つで型 `long64` と `ulong64` を定義する必要があります。これらの型の意味を次の表 9-1 に示します。

表 9-1　64 ビットの型

型	説明
`long64`	符号付き 64 ビット整数
`ulong64`	符号なし 64 ビット整数

コンパイラ定数 `BIG_ENDIAN` または `LITTLE_ENDIAN` の一方をゼロ以外の値に設定することをお勧めします。この設定を必ず行わなければならないのは Java Code Compactor を使用する場合だけですが、マシンのエンディアン形式がわかると KVM はより有用なコードを生成します。

次に、Gnu C コンパイラまたは Solaris C コンパイラを使用する場合の記述方法を示します。

```
typedef long long long64;
typedef unsigned long long ulong64;
```

Microsoft Visual C を使用する場合は、次のように記述します。

```
typedef __int64 long64;
typedef unsigned __int64 ulong64;
```

コンパイラが 64 **ビット整数をサポートしない場合**[1]: プリプロセッサ定数 `COMPILER_SUPPORTS_LONG` をゼロに設定する必要があります。さらに、`BIG_ENDIAN` または `LITTLE_ENDIAN` [2] の一方だけをゼロ以外の値に定義する必要があります。

型 `long64` と `ulong64` は、符号なし `long` ワードである `high` と `low` という 2 つのフィールドから構成される構造体として定義されます。マシンがビッグエンディアンの場合は `high` フィールドが先で、マシンがリトルエンディアンの場合は `low` フィールドが先です。

開発者は、表 9-2 に示す関数を定義する必要があります。プラットフォームが浮動小数点をサポートしている場合は、表 9-3 に示す関数も定義してください。

これらの関数はどれも、マクロとしても実装できます。

表 9-2　long の実装

関数または定数	**対応する** Java **文字列**
`long64 ll_mul(long64 a, long64 b);`	`a * b`
`long64 ll_div(long64 a, long64 b);`	`a / b`
`long64 ll_rem(long64 a, long64 b);`	`a % b`
`long64 ll_shl(long64 a, int b);`	`a << b`
`long64 ll_shr(long64 a, int b);`	`a >> b`
`long64 ll_ushr(long64 a, int b);`	`a >>> b`

1. あるいは、コードが厳密に ANSI 標準規格に準拠していなければならない場合。
2. ビッグエンディアンとリトルエンディアンの詳細は、Jonathan Swift 著『Gulliver's Travels, Part I: A Voyage to Lilliput』を参照してください。

表 9-3　long と float 双方の実装

関数または定数		対応する Java 文字列
`long64`	`float2ll(float f);`	`(long)f`
`long64`	`double2ll(double d);`	`(long)d`
`float`	`ll2float(long64 a);`	`(float)a`
`double`	`ll2double(long64 a);`	`(double)a`

配置に関する事項

Java `long` 型または `double` 型のオブジェクトが Java スタック上または定数プール内に存在する場合、そのアドレスは 4 の倍数です。

ハードウェアプラットフォームの中には、64 ビットの型に対し、それらのアドレスが 8 の倍数となるように配置することが必要なものもあります。

64 ビット整数が 8 バイト境界に配置されることがプラットフォームで必要な場合は、次のように設定してください。

```
#define NEED_LONG_ALIGNMENT 1
```

倍精度の浮動小数点数が 8 バイト境界に配置されることがプラットフォームで必要な場合は、次のように設定してください。

```
#define NEED_DOUBLE_ALIGNMENT 1
```

これらの値が 0 であると、コンパイラはより有用なコードを生成できます。

第10章

ネイティブコード

KVM は、Java Native Interface (JNI) をサポートしていません。Virtual Machine から呼び出されるネイティブコードは Virtual Machine に直接リンクされていなければならず、この呼び出しはこの章で説明している機構を使用している必要があります。

KVM 用の独自のネイティブ関数を記述する方法は、38 ページの「ネイティブメソッドの実装」の節で説明しています。

ネイティブコードのルックアップテーブル

構築プロセスの一環として、メソッドを対応するネイティブ実装に割り当てるルックアップテーブルを構築する必要があります。

JavaCodeCompact は、これらのテーブルを自動的に生成します[1]。JavaCodeCompact のほかの機能を使用するかどうかにかかわらず、ルックアップテーブルの生成にはこのユーティリティを使用する必要があります。

JavaCodeCompact の詳細は、第 13 章を参照してください。ルックアップテーブルが入ったファイルの作成方法は、65 ページで詳しく説明しています。

ネイティブメソッドを実装する C 関数の名前は、JNI[2] がネイティブメソッドに割り当てる名前と同じでなければなりません。

1. 以前のバージョンでは、移植者はこれらのテーブルを手動で構築しなければなりませんでした。現在は、手動での構築の必要はありません。しかし、C 関数には必ず JNI で指定された名前と同じ名前にする必要があります。

2. JNI の命名規則の詳細は、Sheng Liang 著『The Java Native Interface: Programmer's Guide and Specification (Java Series)』(Addison Wesley、1999 年) を参照してください。この情報は、`http://java.sun.com/docs/books/jni/index.html` でオンラインでも提供されています。

ネイティブメソッドの実装

警告: ネイティブメソッドを記述する場合には、実装全体によく目を通し、その構造を理解した上で行なってください。この移植ガイド内の説明は全体的にわかりやすくまとめられていますが、この節は例外です。

KVM リファレンス実装は、ネイティブメソッドの呼び出しに JNI を使用しません。ネイティブメソッドの記述には、十分な注意を払う必要があります。細部への注意を怠ると、Virtual Machine に致命的なエラーを引き起こします。

インクルードファイル

ネイティブ関数を含むコードは、次の行から始めます。

```
#include <global.h>
```

この行は、KVM の一部であるすべてのインクルードファイルを組み込みます。必要に応じ、ほかのファイルも `#include` で組み込むことができます。

ネイティブメソッドからの引数のアクセス

ネイティブメソッドが呼び出される場合、その引数は Java スタックの先頭に位置付けられます。スタックから static メソッドの引数をポップするには、それらがプッシュされた順序とは逆の順序で行う必要があります。図 10-1 は、このコード形式の例を示しています。

Java code:

```
static native void
drawRectangle(int x, int y, int width, int height);

Native implementation:
static void Java_com_sun_kjava_Graphics_drawRectangle() {
    int height = popStack();
    int width =  popStack();
    int y =      popStack();
    int x =      popStack();
    windowSystemDrawRectangle(x, y, width, height);
}
```

図 10-1 ネイティブメソッド

インスタンスメソッド (非 static メソッド) は、this 引数以外の引数をポップしたあとで this 引数をスタックからポップする必要があります。ネイティブインスタンスメソッドで this 引数をポップしないと、Virtual Machine 内でほとんど確実に致命的なエラーが発生します。

表 10-1 は、スタックから引数をポップするために使用するマクロを示しています。

表 10-1 スタックから引数をポップするマクロ

C の型	ポップ用のマクロ
char, byte, int, long	popStack()
float	popStackAsType(float)
long64, ulong64	popLong()
double	popDouble()
pointerType	popStackAsType(*pointerType*)

KVM の以前のバージョンでは、スタックからポインタ型を除去する場合、コードは popStack() を呼び出し、続いて結果を適切な型にキャストしました。次に例を示します。

```
STRING_INSTANCE string = (STRING_INSTANCE)popStack()
```

このコーディング形式は、現バージョンでは使用しないでください。代わりに、次のように記述できます。

```
STRING_INSTANCE string = popStackAsType(STRING_INSTANCE)
```

ネイティブ関数から結果を戻す

ネイティブメソッドが結果を戻す場合、ネイティブメソッドはその結果をスタックにプッシュする必要があります。ネイティブコードは、表 10-2 に示す適切なマクロを使用して結果をスタックにプッシュバックする必要があります。

表 10-2 引数をスタックにプッシュするマクロ

C の型	プッシュ用のマクロ
char, byte, int, long	pushStack()
float	pushStackAsType(float)

表 10-2 引数をスタックにプッシュするマクロ

C **の型**	**プッシュ用のマクロ**
`long64, ulong64`	`pushLong()`
`double`	`pushDouble()`
`pointerType`	`pushStackAsType(`*`pointerType`*`)`

KVM の以前のバージョンでは、ポインタ型をスタックにプッシュする場合、スタックを `long` に変換し、続いてその値をスタックにプッシュしました。この方法はお勧めできません。

ショートカット

ネイティブコードの中には、スタックから最後の引数をポップする代わりにマクロ `topStack` を使用するものがあります。このようなネイティブコードは、続いて、戻したい値に `topStack` を設定します。

この方法はお勧めできません。この方法は、引数にアクセスし、値を単一の文で返す「ワンライナ (one-liner)」にだけ使用してください。この場合、`pushStack` と `popStack` は使用できません。これは、C がそれらの評価順序を保証しないためです。

一般には、値のポップ、計算の実行、およびスタックへの値のプッシュバックを個別の 3 つのステップとして行う方がより安全です。

また、`topStack` の値をポインタ型に変換することや、ポインタを long に変換して `topStack` に割り当てることは、現在はお勧めできません。代わりに、マクロ `topStackAsType(`*`pointerType`*`)` を使用することをお勧めします。このマクロは、値としても割り当ての対象 (左辺値) としても使用できます。このマクロは、スタックの先頭で float 値のアクセスと設定を行うためにも使用できます。

コールバック

ネイティブコードは、Java にコールバックできません。KVM には、ネイティブコードがインタプリタ状態を変更して新しいコードを実行し始めるための機構があります。そのコードの実行を終了した時点で、この機構は呼び出されるべき新しい C 関数を示すことができます。

ネイティブコードにおける例外処理

エラーまたは例外をスローする必要がある場合、ネイティブコードは次の関数を呼び出す必要があります。

```
raiseException(string)
```

`string` 引数には、`exception` クラスまたは `error` クラスの (パッケージ区切り文字として「/」を使用した) 絶対パス名が含まれます。

ネイティブコード内で便利な関数

次に、ネイティブメソッドが必要に応じて呼び出せるその他の便利な関数を示します。

- `void fatalError(string)`
  コードは、重大なエラーが発生したことを示すためにこのメソッドを呼び出します。`string` 引数には問題の概略説明が入ります。このメソッドは結果を返しません。
- `CLASS getClass(const char *name)`
  このメソッドは、指定された引数を名前として持つクラスを返します。必要に応じ、`INSTANCE_CLASS` または `ARRAY_CLASS` になるように戻り値を変換できます。
- `INSTANCE instantiateString(const char* string)`
  このメソッドは、指定された C 文字列を Java 文字列に変換します。
- `char *getStringContents(Instance string)`
  instance 引数は、Java 文字列でなければなりません。この引数は、NULL で終わる C 文字列に変換され、結果として返されます。
  文字列は大域バッファに置かれます。コードがこの文字列を一定時間保持する必要がある場合は、バッファをスタックに割り当てられたストレージにコピーするか、あるいは Java ヒープから領域を割り当てる必要があります。
- `INSTANCE instantiate(CLASS class)`
  指定されたクラスの新しい Java インスタンスを作成します。
- `ARRAY instantiateArray(ARRAY_CLASS arrayType, long length)`
  指定された型と長さを持つ Java 配列を作成します。
- `ARRAY createCharArray(const char* string)`
  引数として渡された C 文字列から Java の文字配列を作成します。

- `char* mallocBytes(long sizeInBytes)`
  ガベージコレクトされたヒープ内に、`sizeInBytes` に指定されたバイト数を保持できるだけの十分な大きさのメモリブロックを割り当てます。メモリブロックがガベージコレクトされる対象とならないように、一時的なルート (42 ページの「ガベージコレクションに関する問題」を参照) を作成できます。

ガベージコレクションに関する問題

KVM がガベージコレクションを実施する場合、C スタックは走査されません。ネイティブコードが新しい Java オブジェクトを割り当てる場合は、その新しい Java オブジェクトが誤ってガベージコレクションの対象とならないように特別な予防策を施す必要があります。一般的な指針としては、ネイティブコード内に新しい Java オブジェクトを作成する場合は常に、ガベージコレクタを呼び出すためのオペレーションを行う前に、オブジェクトのポインタを Virtual Machine が認識できる状態にしておく必要があります。このための方法はいくつかあります。

- オペレーションを実行する前に、(`pushStack` を使用して) オブジェクトを Java スタックにプッシュできます。この場合、あとでそのオブジェクトをスタックから必ずポップする必要があります。
- 図 10-2 または図 10-3 に示す形式のコードを作成できます。
- 起動時に Java オブジェクトを指すようにコードが C 変数を初期化する場合で、その変数の内容がガベージコレクトされる対象となるべきではないときは、図 10-4 に示すコードを使用できます。しかし、一連のグローバルルートから変数を削除する関数は現在存在しません。

```
START_TEMPORARY_ROOT(var)
    ;; var will not be garbage collected
    code that might garbage collect.
END_TEMPORARY_ROOT
```

図 10-2 一時的なルートを 1 つ作成する場合

```
START_TEMPORARY_ROOTS
    MAKE_TEMPORARY_ROOT(var1);
    ;; var1 will not be garbage collected
    code that might garbage collect
    MAKE_TEMPORARY_ROOT(var2);
    ;; neither var1 nor var2 will be garbage collected
    more code
END_TEMPORARY_ROOTS
```

図 10-3 一時的なルートを複数作成する場合

```
variable = <value>
MakeGlobalRoot((cell **)value);
```

図 10-4 グローバルルートを 1 つ作成する場合

重要: コードが連続して 2 つのオブジェクトを割り当てる必要がある場合、2 つめのオブジェクトを割り当てる前に最初のオブジェクトを保護する必要があります。これを怠ると、通常、致命的なエラーが発生します。

ネイティブ関数をデバッグする上での便利なヒント: ガベージコレクタには、メモリ割り当てオペレーションの前にガベージコレクションを引き起こす特殊なモード / フラグ `EXCESSIVE_GARBAGE_COLLECTION` があります。このモードをオンに設定すると、値の保護の設定をし忘れた変数を効果的に見つけることができます。パフォーマンス上の理由から、製品リリースではこのモードを常にオフに設定してください。

大域変数の初期化と再初期化

一般に C 言語は、すべての大域変数および static 変数が 0 (ゼロ) に初期化されることを保証します。

現在の実装は、組み込み環境で動作するように設計されています。たとえば PalmOS では、ユーザは Virtual Machine を起動し、1 つのプログラムを終了し、続いて異なる引数セットを使用して Virtual Machine を再起動できます。これらの 2 つの実行の間に、大域変数または static 変数が初期化し直されることはありません。

一般に、コードはどの変数の初期値も想定できません。1 度だけの初期化を行うべきタイミングを決定する方法はいくつかあります。

- 関数 `initializeNativeMethods()` を使用すると、変数を初期化することも、初期化を行う必要があることを示すフラグを設定することもできます。

- クラスの静的な初期化の一環としてプライベートネイティブメソッドが呼び出される場合、そのクラスが初めて使用される時にそのメソッドのネイティブ実装が呼び出されます。ネイティブ実装は、そのクラスに必要などのような初期化でも行えます。
- 変数がグローバルルートセット (上記の `MakeGlobalRoot()` を参照) の一部である場合、次に Virtual Machine が動作する時点でその値は 0 であることが保証されます。

非同期ネイティブメソッド

オペレーティングシステムの観点から見ると、KVM はたった 1 つの実行スレッドを持つ 1 つのプロセス (C プログラム) にすぎません。KVM のマルチスレッド機能は、実際に使用するオペレーティングシステムのマルチタスク機能を利用することなくソフトウェアとして完全に実装されました。この手法は Virtual Machine をオペレーティングシステムから独立した移植性の高いものにするだけでなく、Virtual Machine 設計を大いに単純化し、コードベースの可読性を向上させます。このため、Virtual Machine の設計者は、マルチスレッド対応のソフトウェアにありがちな相互排除などの問題を気にかける必要がありません。

しかし、この手法には、デフォルトでは KVM 内のすべてのネイティブメソッドが「ブロック」しているという問題が残ります。これは、ネイティブ関数が Virtual Machine から呼び出される場合、VM 内のスレッドはすべてネイティブメソッドが実行を終えるまで実行を停止するということを意味しています。

一般的な指針としては、KVM から呼び出されるネイティブ関数はすべての実行ができるかぎり早く終了するように記述する必要があります。しかし、多くの環境ではこのことは望ましくありません。このため、KVM には以下で説明している「非同期ネイティブメソッド」の実装が含まれています。

非同期メソッドの設計

オペレーティングシステムの観点から見ると、KVM の標準実装は 1 つの「タスク」として動作します。ネイティブメソッドがブロック可能なオペレーションを実行する場合、KVM 全体がブロックされます。

非同期ネイティブメソッドは、この問題を解決することを目的にしています。非同期ネイティブメソッドが呼び出される場合、オペレーションは実装に依存した方法でオフラインで実行されます。ほかの Java スレッドは、通常どおり実行を継続できます。最初にネイティブメソッドを呼び出した Java スレッドは、ネイティブ呼び出しが終了した時点で実行を継続します。

非同期ネイティブメソッドを使用するには、次の行をマシンに応じたインクルードファイルに組み込む必要があります。

```
#define ASYNCHRONOUS_NATIVE_METHODS 1
```

非同期ネイティブメソッドは、通常のネイティブメソッドと同じファイル内には定義できません。通常のインクルードファイルに加えて、インクルードファイル `async.h` も加える必要があります。

非同期メソッドは、必ず次の書式に従う必要があります。

```
ASYNC_FUNCTION_START(functionname)
  code
ASYNC_FUNCTION_END
```

コードには、スタックポインタ、フレームポインタ、または現在のスレッドを参照する `pushStack()`、`popStack()`、`topStack` などのマクロまたは関数を決して使用しないでください。代わりに、表 10-3 に示すマクロを使用してください。

表 10-3 非同期メソッドで使用するマクロ

ネイティブ関数マクロ	非同期ネイティブ関数マクロ
`popStack`	`ASYNC_popStack`
`pushStack`	`ASYNC_pushStack`
`popLong`	`ASYNC_popLong`
`pushLong`	`ASYNC_pushLong`
`popStackAsType`	`ASYNC_popStackAsType`
`pushStackAsType`	`ASYNC_pushStackAsType`
`raiseException`	`ASYNC_raiseException`
`topStack`	このマクロは使用しない

コードでは、「リターン」も実行しないでください。`ASYNC_FUNCTION_END` は必要に応じてクリーンアップコードを生成するため、コードは最後まで継続される必要があります。

表 10-3 内のマクロはすべて、シンボル ASYNCHRONOUS_NATIVE_METHODS が 0 の場合、非同期メソッドが通常のネイティブメソッドとしてコンパイルされるように設計されています。

非同期ネイティブメソッドを使用する場合、以下のマシンに固有な関数を定義する必要があります。

- `void yield_md()`
  このオペレーティングシステムタスクを一時的に中断し、ほかのタスクの実行を許可します。
- `void CallAsynchronousFunction_md(THREAD, void(f*)(THREAD))`
  関数 `f` を呼び出し、この関数にその単一の引数としてスレッド引数を渡します。関数 `f` は、非同期に呼び出す必要があります (別のタスク内で呼び出すなど)。
- `enterSystemCriticalSection`
  `exitSystemCriticalSection`
  クリティカルセクションを開始または終了します。オペレーティングシステムは、クリティカルセクション内では一度に 1 つのオペレーティングシステムタスクしか許可されないようにする必要があります。

非同期メソッドの実装

KVM では、2 つの非同期メソッド実装が想定されています。

現在のリファレンス実装では、関数 `CallAsynchronousFunction_md` は指定された関数を実行する別のオペレーティングシステムタスクを生成します。たとえば、Posix 実装では `pthread_create` を使用できます。

次の図 10-5 は、次のメソッド
`int readBytes(byte[] dst, int offset, int length)`を、この形式の非同期ネイティブメソッドを使用して実装する方法を示しています。

```
ASYNC_FUNCTION_START(ReadBytes)
    long   length = ASYNC_popStack();
    long   offset = ASYNC_popStack()
    BYTEARRAY dst = ASYNC_popStackAsType(BYTEARRAY)
    INSTANCE instance = ASYHNC_popStackAsType(INSTANCE)/* this*/
    long fd = getFD(instance);
    length = read(fd, dst->bdata + offset, length);
    ASYNC_pushStack((length == 0) ? -1 : length);
ASYNC_FUNCTION_END
```

図 10-5 `ReadBytes` の非同期実装（例１）

次に示す別の実装例では、`CallAsynchronousFunction_md` は関数 `f` を直接呼び出しています。関数 `f` がオペレーションを開始しますが、その完了を待たないことが仮定されています。オペレーティングシステムは、オペレーションの完了時点を示す何らかの割り込みかコールバックを提供する必要があります。

```
static void ReadBytes(THREAD thisThread)
{
    long   length = ASYNC_popStack();
    long   offset = ASYNC_popStack();
    BYTEARRAY dst = ASYNC_popStackAsType(BYTEARRAY);
    INSTANCE instance = ASYNC_popStackAsType(INSTANCE);
    long fd = getFD(instance);
    THREAD thisThread = CurrentThread;
    /* Call OS to perform I/O. Perform callback when done. */
    AsyncRead(fd, p + offset, length, ReadBytesDone,thisThread);
}

/* Callback function when I/O is finished */
static void ReadBytesDone(void *parm, int length)
{
     THREAD thisThread = (THREAD)parm;
     ASYNC_pushStack((length == 0) ? -1 : length);
     ASYNC_RESUME_THREAD();
}
```

図 10-6 `ReadBytes` の非同期実装 (例２)

この 2 つめの実装は、オペレーティングシステムに依存しています。フラグの値次第では、同期と非同期の両方で動作できるネイティブメソッドを記述できない場合もあります。

非同期コードの記述の詳細は、51 ページの「非同期通知」を参照してください。

第 11 章

イベント処理

概要

KVM には、イベントの通知と処理を行う方法が 4 つあります。

1. 同期通知 (ブロック化)
2. Java コード内でのポーリング
3. バイトコードインタプリタ内でのポーリング
4. 非同期通知

同期通知 (ブロック化)

同期通知は、KVM が Virtual Machine から直接ネイティブ関数を呼び出してイベント処理を実行する状況を示すために使用します。KVM は Virtual Machine 内に物理制御スレッドを 1 つしか持たないため、ネイティブ関数が処理されている間はほかの Java スレッドの処理は不可能であり、ガベージコレクションのような VM システム機能も発生することがありません。これはイベント通知のための最もシンプルな方法ですが、設計者が十分な注意を払ってネイティブ関数をできるかぎり短くかつ効率良く使用できれば、このソリューションはさまざまな状況で便利に使えます。

たとえば、ネットワークへのデータグラムの書き込みは、一般にこの手法によって効率良く行えます。これは、データグラムが、通常、バッファを含んだネットワークスタックに送信され、イベントの完了までの待機時間はわずかであるためです。しかし、データグラムの読み出しはまったく別の話であり、以下に説明しているほかのソ

リューションを使用する方が良い処理結果が得られることも少なくありません。データグラム全体が受信されるまでネイティブ関数を使用して待機すると、読み取りオペレーションが進行している間 KVM 全体がブロックされます。

Java コード内でのポーリング

ネイティブコードと Java コードの組み合わせを使用すると、イベント処理を効率良く実装できることがよくあります。これは、1 つのイベントの完了を待つ間ほかの Java スレッドを実行できるようにするシンプルな方法です。この手法を使用する場合、Java ループのポーリングは、通常、Java 実行時ライブラリのどこかに置かれます。これは、ループをアプリケーションから隠すためです。実行時ライブラリは、短いネイティブ入出力オペレーションを開始し、続いて入出力オペレーションが完了するまで繰り返しその状態を問い合わせるという方法を一般にとります。ポーリングする Java コードループには、ほかの Java スレッドが効率良く実行されるように、必ず `Thread.yield()` の呼び出しが含まれています。

イベント通知を待つためのこの方法は簡単に実装でき、徹底した非同期スレッドにありがちな複雑さ (クリティカルセクション、セマフォ、モニターなどを必要とするなど) がありません。しかし、この設計には難点が 2 つあります。その 1 つは、通常はアプリケーションコードの実行に使用できる CPU サイクルを Java レベルのポーリングを実施するために使用しなければならないことです (しかしオーバーヘッドは通常ごくわずか)。2 つめは、解釈のオーバーヘッドのために、イベント通知に関連した余分な待ち時間が生じる可能性があることです (特にポーリングする Java コードループ内で `Thread.yield()` を呼び出すことを忘れた場合)。このオーバーヘッドも、速度が重視されるアプリケーションを除きあらゆるアプリケーションで通常は無視できます。

バイトコードインタプリタでのポーリング

イベント処理を実装する 3 つめの手法として、バイトコードインタプリタを周期的に使用してネイティブイベント処理オペレーションを呼び出すことができます。この手法は、前述の同期通知による方法に変化を持たせたものです。この手法は、Palm プラットフォーム用の GUI イベント処理を実装するなど、KVM で広範囲に渡って使用されています。

この手法では、インタプリタループからネイティブイベント処理関数が周期的に呼び出されます。パフォーマンス上の理由から、この手法は各バイトコードの前に実行するのではなく、約数百のバイトコードごとに実行します。そうすることで、イベント

処理にかかる負担は適切に使われます。Virtual Machine の設計者は、イベント処理コードを呼び出す前に実行されるバイトコードの数を変更することにより、イベント配布の待ち時間と、新しいイベントを探すために費やされる CPU 時間を制御できます。この数を小さくすると、待ち時間が短くなり、CPU オーバーヘッドが大きくなります。数を大きくすると、CPU オーバーヘッドが減少しますが、イベント処理の待ち時間が長くなります。

この手法の利点は、パフォーマンスの損失が Java でポーリングを実行するよりも少なくてすみ、イベント通知の待ち時間の予測と制御が容易であることです。この手法は、次の節で説明している非同期通知と密接な関係があります。

非同期通知

当初の KVM 実装でサポートされたのは、前述した 3 つのイベント処理実装だけでした。しかし、非同期イベント処理をサポートするために、最近新しい機構がいくつか取り入れられました。

非同期通知は、Virtual Machine が実行を継続している間並行してイベント処理が行われる状況を示すために使用します。これは一般に最も効率の良いイベント処理手法であり、通知の待ち時間は通常ごくわずかですみます。しかし、この手法には一般に、実際に使用するオペレーティングシステムが非同期イベント処理を実装するための適切な機能が必要となります。すべてのオペレーティングシステムがこのような機能を提供しているとはかぎりません。また、この手法は実装がかなり複雑であり、Virtual Machine の設計者はロックおよび相互排除の問題を認識している必要があります。リファレンス実装には、各デバイスにより適したイベント処理オペレーションを実装する場合の手段として使用できるサンプルがいくつか提供されています。

非同期通知の一般的な手順は次のようになります。スレッドが、入出力オペレーションを開始するためにネイティブ関数を呼び出します。続いてネイティブコードがそのスレッドの実行を中断し、ただちにインタプリタループに戻ります。これにより、ほかのスレッドが実行を継続できるようになります。続いてインタプリタが、実行する新しいスレッドを選択します。その後しばらくして非同期イベントが発生し、その結果、中断されているスレッドを再開するネイティブコードのどれかが実行されます。続いてインタプリタが、イベントの発生に待機していたスレッドの実行を再開します。

実装レベルでこのような非同期通知を実装する方法は 2 つあります。その 1 つはネイティブ (オペレーティングシステム) スレッドを使用するもので、もう 1 つはある種のソフトウェア割り込みか、コールバックルーチン、またはポーリングルーチンを使用するものです。

最初の方法では、ネイティブ関数が呼び出されて Java スレッドが中断される前に新しいオペレーティングシステムスレッドが作成され (または呼び戻され)、このスレッドがネイティブ関数を指定します。この時点で、Virtual Machine の内部で動作するネイティブ制御スレッドが 1 つ増えます。ネイティブ入出力スレッドが開始されたあと、Virtual Machine 内における実行の順序は確定的ではなく、外部イベントの発生によって決まります。一般には、当初のスレッドがインタプリタループ内の別の Java スレッドの実行を開始し、その新しいスレッドが通常オペレーティングシステムに対する入出力オペレーションをブロックして入出力オペレーションを開始します。

ネイティブ入出力関数はコンテキストの範囲外で実行される (つまり Virtual Machine のコンテキストは別のスレッドである) ということに注意してください。この事実をほぼ覆い隠す特殊な C マクロセットが作成されていますが、このルーチンでコンテキストポインタが使用されないように特別な注意を払う必要があります。ブロック呼び出しが完了した時点で、ネイティブ入出力スレッドは実行を再開し、Java スレッドのブロックを解除します。続いて Java スレッドが再スケジュールされ、ネイティブ入出力スレッドは破棄されるか、あるいは再び必要になるまで休止状態に置かれます。KVM リファレンス実装の Win32 ポートは、入出力が行われる時に再使用される入出力スレッドのプールを作成してこの処理を行います。

非同期イベント処理の 2 つめの実装は、入出力要求に関連付けられたコールバック機能を利用して行えます。この場合、通常のインタプリタスレッドを使用してネイティブコードが入力され、入出力が開始され、続いて入出力オペレーションが完了した時点でオペレーティングシステムによってコールバックルーチンが呼び出されて、Java スレッドの中断が解除されます。このケースでは、ネイティブコードは 2 つのルーチン、入出力オペレーションを開始するルーチンと入出力が完了する時点のルーチンに分割されます。最初のルーチンは呼び出す Java スレッドのコンテキスト内で動作しますが、2 つめはそうではありません。

最後に、効率がやや落ちる非同期イベント処理として、インタプリタループによる完了のために入出力ルーチンがポーリングされる場合が挙げられます。これは、入出力が完了したかをチェックするために 2 つめのルーチンがインタプリタによって繰り返し呼び出される点を除きコールバックによる方法に非常に似ています。最終的に、入出力オペレーションが完了した時点で、待機している Java スレッドのブロックをルー

チンが解除します。インタプリタによるこのネイティブコード呼び出しは、保留中のイベントが存在しない場合でも常に行われます。ネイティブコードは、どの Java スレッドを再開すべきかを決定する必要があります。

同期に関する注意事項: 別のネイティブイベント処理スレッドまたはコールバックルーチンが使用される場合、イベント処理用のコードはいつでも Virtual Machine に割り込む可能性があることに注意してください。したがって、Virtual Machine の設計者はプログラムが共通のデータ構造を操作する可能性がある位置や相互排除問題が発生し得るすべての位置に必ずクリティカルセクション、モニター、またはセマフォを追加する必要があります。明白な共有データ構造として、中断された Java スレッドとアクティブな Java スレッドの待ち行列が挙げられます。これらは常に、あらかじめ適切に同期がとられた、Virtual Machine 内の特殊なルーチンを使用して操作します。ほかにも共有データ構造が存在する場合、それらはネイティブコード内で同期をとる必要があります。この作業を正しく行わないと、デバッグが非常に困難な不可解なバグが発生します。

パラメータの引き渡しとガベージコレクションに関する事項

ネイティブイベント処理コードが呼び出される場合、そのパラメータは呼び出し側の Java スレッドのスタック上に存在します。これらはネイティブコードによってスタックからポップされ、返される結果の値が存在する場合はスレッドの実行を再開する直前に Java スタック上にプッシュされます。ネイティブパラメータの引き渡しについては、第 10 章で説明しています。

ネイティブイベント処理コードはオブジェクトメモリにアクセスできるため、ガベージコレクションの問題が発生する可能性があります (特に長い非同期入出力オペレーションを実行する場合)。通常、ガベージコレクタは、実行中のネイティブコードが存在する場合には行われません。このことは、ある種の長い入出力オペレーションが行われる場合に問題となります。明らかな例として、着信するネットワーク要求に待機している場合が挙げられます。この問題を解決するため、2 つの関数 `startLongIOActivity()` と `endLongIOActivity()` が提供されています。この最初の関数はガベージコレクタを開始しますが、2 つめの関数はコレクタの開始を防止し、コレクタが動作していた場合は停止するのを待ちます。

Java コード内でオブジェクト参照がネイティブメソッドに渡されているが、`startLongIOActivity()` の呼び出し以降はネイティブメソッドに対する参照は存在しないという場合、オブジェクトがガベージコレクタによって誤って再生される可能性があることに注意してください。このような状況が発生することは実際の場面ではほとんどないのですが、可能性もないとはいえません。このようなコードの例を次に示します。

```
native read(byte[]);
void skipBytes(int n) {
    read(new byte[n]);
}
```

このコードでは、バイト配列オブジェクトの唯一の参照がネイティブ関数のパラメータスタック上に存在します。スタックからパラメータをポップしたあとにネイティブコードが `startLongIOActivity()` を呼び出すと、この配列はガベージコレクションの対象となり得ます。

KVM における実装

KVM におけるイベント処理実装は、新しいハードウェアプラットフォームに KVM を移植する場合に考慮すべき 2 つの主要な層から構成されます。

インタプリタループの先頭には次のコードが存在します。

```
if  (isTimeToReschedule()) {
     reschedule();
```

}
標準の再スケジューリングコードは、以下のオペレーションを実行します。

1. アクティブ Java スレッドが存在するかチェックし、存在しない場合は VM を停止します。
2. 一定時間待機していたスレッドが再開できるように十分な時間が経過したかをチェックします。このようなスレッドが存在する場合は、そのスレッドが自動的に再開します。

3. 入出力イベントが何か発生したかをチェックし、必要に応じて関連スレッド間にCPU 時間を割り当てます。

4. 別のスレッドへの切り替えを試みます。

パフォーマンス上の理由から、上記のオペレーションはデフォルトで`VmCommon/h/events.h` に定義されるマクロとして実装されます。デバイス固有のイベント処理コードを配置できるのは、このファイル内です。デフォルトでは、`isTimeToReschedule()` マクロがグローバルカウンタを減分し、このカウンタがゼロになるかどうかをテストします。ゼロになると、2 つめのマクロが実行されます。この場合の目的は、多数のバイトコードの実行に対して `reschedule()` が 1 度だけ実行されるようにすることです。名前が示しているように、`reschedule()` は必要に応じてスレッドコンテキストの切り替えが行われる位置です。

イベント処理実装内の 2 つめの層は次の関数です。

```
GetNextKVMEvent(KVMEventType *evt, bool_t forever,
                ulong64 waitUntil)
```

イベントがまったく発生しない場合、この関数は FALSE を返す必要があります。イベントが発生する場合、`evt` 引数にはそのイベントの詳細が入り、この関数は TRUE を返します。

ほかの引数は次のようになります。

- `forever` 引数が `TRUE` の場合、この関数はイベントが発生するのを必要なかぎり待ちます (以下に説明しているようにバッテリ節約のために使用される)。
- `forever` 引数が `FALSE` の場合、この関数は長くてもイベントの `waitUntil` が発生するまでしか待機しません。

これらの関数のリファレンス実装には、バッテリ節約機能が含まれています。これは、イベントチェック機能に対し forever フラグか最大の待機時間を渡すためです。保留中のイベントが存在しない場合、`GetNextKVMEvent` ルーチンのネイティブ実装は次のイベントが発生するまでデバイスをスリープ状態にすることができます。バッテリ節約の詳細は、次の節で説明しています。

バッテリ電力の節約

KVM のほとんどのターゲットデバイスはバッテリによって稼動するため、これらのデバイスのメーカーは一般に過度のバッテリ電力消費に対する対策をとっています。バッテリの使用を最小限に抑えるため、KVM は Virtual Machine 内にアクティブ Java スレッドが存在しない時と Virtual Machine が外部イベントの発生に待機している時は KVM インタプリタループの実行を停止するように設計されています。しかし、この機能は実際に使用するオペレーティングシステムのサポートを必要とします。

電力節約機能を十分に利用するためには、次に示す低レベルのイベント読み取り関数を移植する必要があります。

```
GetNextKVMEvent(KVMEventType *evt, bool_t forever,
                ulong64 waitUntil)
```

Virtual Machine が `forever` 引数を TRUE に設定してこの関数を呼び出す場合、この関数はホストシステムに固有のスリープ機能を呼び出します。KVM は、Virtual Machine がその時点でほかに何も行うべきことがない場合 `forever` 引数を `TRUE` に設定したこの関数を自動的に呼び出すように設計されています。

このため、イベント読み取り関数のネイティブ実装は、次のネイティブイベントが発生するまでデバイスに固有の適切なスリープ機能を呼び出すことができます。

さらに、マクロ `SLEEP_UNTIL(wakeupTime)` も、`wakeupTime` ミリ秒が渡されるまでターゲットデバイスがスリープ状態に入るように定義する必要があります。

第12章

クラスファイルの検証

既存の JDK クラスファイルベリファイアは、リソース制約型の小さなデバイスには適しません。JDK ベリファイアは、実行時に最小 50K のバイナリコード領域と最小 30K 〜 100K のダイナミック RAM を必要とします。また、標準の JDK ベリファイアの繰り返し型のデータフローアルゴリズムを実行するには相当な CPU パワーが必要です。

Sun は、既存の JDK ベリファイアよりもかなり小さい新しいクラスファイルベリファイアを設計し、実装しました。この新しいベリファイアは、一般的なクラスファイルの場合、実行時に約 10K バイトの Intel x86 バイナリコードと 100 バイト未満のダイナミック RAM を使用します。このベリファイアはバイトコードの線形スキャンを実行するだけであり、負担の大きい繰り返し型のデータフローアルゴリズムは不要です。この新しいベリファイアは、KVM (リソース制約型デバイスを対象とした小規模の Java Virtual Machine) にとりわけ適しています。

この新しいバイトコードベリファイアは、(`jsr` 命令および `ret` 命令が存在しないように) すべてのサブルーチンがインライン化され、かつクラスファイルに `StackMap` 属性が含まれることを要求します。KVM には、javac のあとでこの属性を通常のクラスファイルに挿入するプリベリファイアというクラスファイル変換ツールが付いています。変換されたクラスファイルには実行時に検証を効率良く行うための属性が追加されますが、 J2SE クラスファイルとして使用できます。

KVM リリースに付属のこのプリベリファイアは、JDK 1.1.8 Virtual Machine 実装から得られたコード、およびサブルーチンのインライン化と `StackMap` 属性の追加のために記述されたコードを含む C プログラムです。このプログラムはコンパイルと実行を Windows 上と Solaris 上で行いますが、ほかの開発プラットフォームに簡単に移植できます。

新しいベリファイアの使用

プリベリファイアの起動

事前検証は、通常、開発用のワークステーション上でアプリケーションの開発時に行われます。プリベリファイアは次のように使用します。ここでは、既に javac を使用して次のように Foo.java をコンパイルしてあります。

```
javac -classpath kvm/classes Foo.java
```

次は、別のディレクトリに javac の出力を書き込み、続いて生成されたクラスファイルをプリベリファイアを使用して変換する必要があります。

```
javac -classpath kvm/classes -d tmpdir Foo.java
preverify -classpath kvm/classes -d . tmpdir
```

上記のプリベリファイアコマンドは、tmpdir/ の下のクラスファイルをすべて変換し、変換後のクラスファイルを (-d オプションで指定されているとおり) 現在のディレクトリに書き込みます。

KVM リリース内の Makefile は、プリベリファイアを自動的に起動します。

プリベリファイアのオプション

プリベリファイアは、多くの引数とオプションを取ります。

`-classpath <directories>`

- クラスがロードされるディレクトリ。ディレクトリの区切り文字は、プラットフォームによって異なります。Solaris ではコロンが使用されます。Win32 ではセミコロンが使用されます。

`-d <directory>`

- 出力クラスが書き込まれるディレクトリ。デフォルトの出力ディレクトリは、./output です。

`@<filename>`

- コマンド行引数が読み取られるテキストファイルの名前

コマンド行オプションのあとには、クラス名とディレクトリ名のリストが指定されます。プリベリファイアは、クラス名ごとに classpath に指定されたディレクトリで対応するクラスファイルを検索し、そのクラスファイルを変換します。さらに、ディレクトリ名ごとにそのディレクトリ内の各クラスファイルを再帰的に変換します。

将来、`javac` はサブルーチンを生成せずに適切な属性を生成するように変更される可能性があります。その場合、プリベリファイアツールは不要になります。

新しいベリファイアの移植

実行時部分: 新しいベリファイアの実行時部分は、通常、移植作業を必要としません。これは、この部分が Virtual Machine の残りの部分と密接に統合されており、移植可能な C コードで実装されているためです。

プリベリファイアの部分: プリベリファイアも C で記述されています。プリベリファイアはデフォルトでは Windows と Solaris 用として提供されていますが、ほかのオペレーティングシステムでも動作するようにコンパイルする作業は比較的容易です。

プリベリファイアのコンパイル

プリベリファイア用のソースは、ディレクトリ `tools/preverifier/src` に入っています。

Solaris では、`tools/preverifier/build/solaris` ディレクトリで `gnumake` コマンドを入力してプリベリファイアを構築できます。これにより、`tools/preverifier/src` ディレクトリ内のすべての `.c` ファイルがコンパイルされ、リンクされて、生成された実行可能ファイルが `tools/preverifier/build/solaris` ディレクトリに書き込まれます。

Win32 では、`tools/preverifier/build/win32` ディレクトリのワークスペースファイルをロードすることによりプリベリファイアを構築できます。これにより、`tools/preverifier/src` サブディレクトリ内のすべての `.c` ファイルがコンパイルされ、リンクされて、生成された実行可能ファイルが `tools/preverifier/build/win32/{Debug,Release}` ディレクトリに書き込まれます。

第13章

JavaCodeCompact

KVM は、JavaCodeCompact (JCC) ユーティリティ (クラスプリリンカ、クラスプリローダなどとも呼ばれる) をサポートしています。このユーティリティを使用すると、Java クラスを Virtual Machine 内で直接リンクし、VM の起動時間を大幅に短縮できます。

JavaCodeCompact ユーティリティは、実装レベルで Java クラスファイルを結合し、コンパイルして Java Virtual Machine とリンクできる C ファイルを生成します。

一般的なクラスローディングでは、javac を使用して Java ソースファイルをコンパイルして、Java クラスファイルを生成します。これらのクラスファイルは、個々に、あるいは JAR アーカイブファイルの一部として Java システムにロードされます。クラスローディング機構は、必要に応じてほかのクラス定義への参照も解決します。

JavaCodeCompact は、プログラムリンクとシンボル解決を行う代替手法として利用できます。JavaCodeCompact はプログラム構築モデルとしては柔軟性にやや欠けますが、VM の帯域幅とメモリの要件を減らすのに効果的です。

JavaCodeCompact は次の作業に使用できます。

- 複数の入力ファイルを結合する
- オブジェクトインスタンスのレイアウトとサイズを決定する
- 指定されたクラスメンバだけをロードし、ほかのクラスメンバを破棄する

JavaCodeCompact のオプション

JavaCodeCompact は、多数の引数とオプションを取ります。以下に示しているのは、現在 KVM でサポートされているオプションだけです。

- *filename*

  入力として使用されるファイルの名前を指定します。このファイルの内容は、出力されます。接尾辞 `.class` を持つファイル名は、単一クラスファイルとして読み取られます。

  接尾辞 `.jar` または `.zip` を持つファイル名は、Zip ファイルとして読み取られます。これらのファイルの要素であるクラスファイルは読み取られますが、ほかの要素は何も処理されずに無視されます。

- `-o` *outputfilename*

  生成される出力ファイルの名前を指定します。このオプションが指定されない場合、`ROMjava.c` という名前でファイルが生成されます。

- `-nq`

  JavaCodeCompact でバイトコードがそれらの「quickened」形式に変換されないようにします。`-nq` は、現在、KVM の必須オプションです。

- `-classpath` *path*

  JavaCodeCompact がクラスの検索に使用するパスを指定します。ディレクトリと Zip ファイルは、Java 定数 `java.io.File.pathSeparatorChar` で定義された区切り文字で区切られます。通常、Unix プラットフォームではコロン、Windows プラットフォームではセミコロンが区切り文字として使用されます。

  複数のクラスパスオプションを指定でき、左から順に検索されます。このオプションは、複数指定可能なリンクオプション `-v`、および選択式のリンクオプション `-memberlist` と共に使用されます。

- `-memberlist` *filename*

  指定されたファイルに示されたとおりに、選択的なロードを行います。このファイルは JavaFilter によって生成された時点で ASCII ファイルであり、クラスとクラスメンバの名前が記述されています。

- `-v`

  リンク処理で詳細な指定ができます。このオプションは複数指定可能です。現在、最高 3 レベルまで認識できます。このオプションは、デバッグ補助機能としてのみ使用されます。

- `-arch` *Architecture*

  ROM イメージを生成するアーキテクチャを指定します。PalmOS 用の JavaCodeCompact を使用している場合は、アーキテクチャとして `PALM` を指定する必要があります。それ以外では、アーキテクチャとして KVM を指定する必要があります。

JavaCodeCompact の移植

1 つの例外を除き、JavaCodeCompact はプラットフォームにまったく依存しない C コードを出力します。

`final static long` または `final static double` である変数を初期化する場合、JavaCodeCompact は次の 2 つのマクロを使用して適切な初期化を行います。

```
ROM_STATIC_LONG(high-32-bits, low-32-bits)
ROM_STATIC_DOUBLE(high-32-bits, low-32-bits)
```

コンパイラ `BIG_ENDIAN` または `LITTLE_ENDIAN` をゼロ以外の値に初期化した場合、ファイル `src/VmCommon/h/rom.h` はこれらのマクロに対してデフォルト値を生成します。

`BIG_ENDIAN`、`LITTLE_ENDIAN` のどちらもまだ定義してない場合、あるいは何らかの理由で `rom.h` に定義されているマクロがプラットフォームに適さない場合、プラットフォームに応じた位置に `ROM_STATIC_LONG` または `ROM_STATIC_DOUBLE` (あるいはこの両方) に適した定義を作成する必要があります。

プラットフォームまたはポートについて判明している依存性はほかにはありません。

JavaCodeCompact のコンパイル

`JavaCodeCompact` のソースは、ディレクトリ `tools/jcc/src` に入っています。

Unix と Windows マシンでは、`tools/jcc/` ディレクトリでコマンド `gnumake` を入力して JavaCodeCompact をコンパイルします。これにより、`tools/jcc/src` サブディレクトリ内のすべての `.java` ファイルがコンパイルされ、生成されたコンパイル済みファイルが `tools/jcc/classes` ディレクトリに書き込まれます。

必要に応じてこのファイルを変更し、使用する javac コンパイラの位置を指定してください。

JavaCodeCompact ファイル

ディレクトリ tools/jcc には、JavaCodeCompact の実行に必要なすべての手順を示した Makefile が入っています。この Makefile は、現在次の 3 つのプラットフォームを対象にしています。

```
unix
windows
palm
```

これらはそれぞれ、そのプラットフォームに必要なすべてのファイルを作成するために使用できます。

unix と windows プラットフォームでは、次の 2 つのファイルが作成されます。

```
ROMjava<i>Platform</i>.c
nativeFunctionTable<i>Platform</i>.c
```

最初のファイルには、Zip ファイル内のクラスに対応する C データ構造が含まれます。2 つめのファイルには、ネイティブ関数を使用するために必要なテーブル (37 ページの「ネイティブコードのルックアップテーブル」を参照) が含まれます。JavaCodeCompact ユーティリティのほかの機能を使用する予定があるかどうかにかかわらず、この 2 つめのファイルをコンパイルして KVM にリンクする必要があります。

Palm では、次に示す数個のファイルが作成されます。

```
nativeFunctionTablePalm.c
nativeRelocationPalm.c
kvm/VmPilot/build/bin/PalmROM1001.bin
...
kvm/VmPilot/build/bin/PalmROM1010.bin
```

この場合も、ファイル nativeFunctionTable.c には、ネイティブ関数を使用するために必要なテーブルが含まれます。JavaCodeCompact ユーティリティのほかの機能を使用する予定があるかどうかにかかわらず、このファイルをコンパイルして KVM にリンクする必要があります。ファイル nativeRelocationPalm.c には、ネイティブメソッドの実行に必要な再配置情報が含まれます。ディレクトリ kvm/VmPilot/build/bin には、kvm.prc ファイルに含める必要がある一連の Palm リソースファイルが含まれます。

JavaCodeCompact の実行

JavaCodeCompact ユーティリティは、ネイティブメソッドの呼び出しに必要なテーブルが入った、プラットフォーム固有のファイル `nativeFunctionTablePlatform.c` を構築するために使用されます。

このファイルは、JavaCodeCompact の機能を使用してクラスをプリロードする予定がなくても構築する必要があります。

JavaCodeCompact を使用しない場合、以下の手順 4 は省略できます。

JavaCodeCompact ユーティリティを使用するには、提供されている `Makefile` を使用するか、あるいはプラットフォームに合わせてこの `Makefile` を変更するのが簡単です。次に、`Makefile` が実行する手順を示します。

1. `api/src` ディレクトリ内のすべての `.java` ファイルをコンパイルします。生成されるクラスファイルが検証されて、単一の Zip ファイル `classes.zip` としてマージされます。この Zip ファイルが、`tools/jcc` ディレクトリにコピーされます。
2. 前述の 63 ページの「JavaCodeCompact のコンパイル」で説明しているように、JCC のソースをコンパイルします。
3. `classes.zip` を `classes`*`Platform`*`.zip` にコピーします。このプラットフォーム固有の Zip ファイルから、使用しているプラットフォームで使用すべきでないクラスやパッケージを削除します。
4. 使用しているシステムに合わせて、コマンドを実行します。

   a. [Palm 以外] `jcc` ディレクトリで、次のコマンドを実行します。

   ```
   env CLASSPATH=classes \
       JavaCodeCompact -nq -arch KVM \
       -o ROMjava*Platform*.c classes*Platform*.zip
   ```

   `env CLASSPATH-classes` は、JavaCodeCompact を実行するコードが `classes` というサブディレクトリに存在することを示す環境変数を設定します。その次のコマンド行では、メインメソッドが実行されるクラスの名前 (`JavaCodeCompact`) と、そのメソッドの引数が指定されています。

   b. [Palm] 以下の 2 つのコマンドを実行します。

```
env CLASSPATH=classes \
    JavaCodeCompact -nq -arch Palm \
    -o ROMjavaPalm.c classesPalm.zip
env CLASSPATH=classes \
    JavaCodeCompact -nq -arch Palm \
    -imageAttribute relocating
    -o nativeRelocationPalm.c classesPalm.zip
```

ファイル `nativeRelocationPalm.c` は、構築プロセスにおいてソースファイルとしてインクルードされます。このファイルを、次のようにコンパイルして、実行します。

```
cc -I../kvm/VmPilot/h -I../../ikvmvm/VmCommon/h \
       -DRELOCATABLE_ROM -DROMIZING ROMjavaPalm.c \
       -o ROMjavaPalm
```

生成された実行可能ファイル `ROMjavaPalm` を、次のように実行します。

```
ROMjavaPalm ../../kvm/VmPilot/build/bin/PalmROM
```

リソースファイルを指定されたディレクトリ内に作成します。

5. `jcc` ディレクトリで、使用しているシステムに合わせて次のコマンドに相当するコマンドを実行します。

```
env CLASSPATH=classes \
    JavaCodeCompact -nq -arch KVM_Native
    -o nativeFunctionTable_Platform_.c classes_Platform_.zip
```

このコマンドは、ネイティブメソッドを対応する C コードにリンクするために必要なネイティブ関数テーブルを含むファイルを生成します。

6. KVM のソースをすべて再コンパイルします。この場合、プリプロセッサマクロ `USING_ROMIZER` がゼロ以外の整数値に設定されていることを確認する必要があります。また、ファイル `ROMjavaPlatform.c` (Palm 以外) または `nativeRelocationPalm.c` (Palm) がソースファイルの 1 つとして存在していることも確認する必要があります。

生成される kvm イメージは、オリジナルの `classesPlatform.zip` ファイルに存在していた、ロード済みのすべてのクラスファイルを含みます。

制限事項

JavaCodeCompact の現在の実装では、圧縮されるクラスファイルが「他動的な閉鎖」を構成していることが必要です。クラス A が圧縮される場合でクラス A の定数プールがクラス B を参照するとき、その圧縮の一部としてクラス B も含まれる必要があります。

以下の状況のどれかに該当する場合、クラス A はその定数プールにクラス B を含みます。

- クラス A がクラス B の直接のサブクラスであるか、クラス A がクラス B を直接実装している
- クラス A がクラス B のインスタンス、またはクラス B の配列を作成する
- クラス A がクラス B で定義されているメソッドを呼び出す
- クラス A が、あるオブジェクトがタイプ B のインスタンスであるかをチェックするか、あるいはタイプ B にオブジェクトをキャストする

以下の状況は、クラス B がクラス A の定数プールに含まれる原因とはなりません。条件次第では、B を圧縮することなく A を圧縮できる場合もあります。

- クラス A がタイプ B のインスタンス変数を持つ
- クラス A が、引数または戻り値の型がそのシグニチャーにタイプ B を含むメソッドを持つ
- クラス A が、`ClassforName()` メソッドを使用してクラス B のインスタンスを作成する

JavaCodeCompact に必要なクラスファイルを組み込まないと、JavaCodeCompact は失敗し、エラーメッセージを表示します。

第14章

Java Application Manager (JAM)

ほとんどのターゲットデバイスで KVM に必要なことは、ネットワークから動的にダウンロードされたアプリケーションを実行できることです。いったんダウンロードすると、一般にユーザはアプリケーションを数回使用してから削除しようとします。Java アプリケーションのダウンロード、インストール、検査、起動、およびインストール解除のプロセスは、一般にアプリケーション管理と呼ばれます。典型的なデスクトップコンピューティング環境では、これらの作業はホストオペレーティングシステムの機能を利用して行えます。しかし、組み込みファイルシステムのような基本機能すら欠如しがちな多くのリソース制約型の小規模デバイスでは、状況はまったく異なります。

リソース制約型の小規模プラットフォームへの KVM の移植を簡単にするために、KVM 実装にはマシン固有の実装を開発する出発点として使用できるコンポーネントである Java Application Manager (JAM) が含まれています。

JAM は、次のフラグを使用してコンパイルレベルで有効または無効に設定できます。

```
#define USE_JAM 1
```

この節では、KVM に付属の JAM リファレンス実装の概要を示します。以下の説明は、アプリケーションのダウンロードの開始に使用できるある種の「マイクロブラウザ」がターゲットデバイスに存在していることを想定しています。このマイクロブラウザは一般にはネイティブコンピューティング環境の一部として提供されていますが、一部の実装では JAM の一部として存在します。

注 – JAM 実装は、現在、KVM の Windows バージョンと Solaris バージョンにだけ提供されています。

JAM を使用してアプリケーションをインストールする

Java Application Manager は、Java アプリケーションのダウンロード、インストール、検査、起動、およびインストール解除を行うネイティブ C アプリケーションです。

ユーザは一般に、JAM を次のように使用します。

1. コンテンツプロバイダの Web ページに載せられたアプリケーションを見つけます。
2. そのアプリケーションをインストールするためのタグを選択します。
3. Java アプリケーションがダウンロードされ、インストールされます。
4. このアプリケーションを実行します。

次に、この手順の詳細を示します。

1. ユーザがネイティブマイクロブラウザを使用してコンテンツプロバイダの Web ページをブラウズすると、そのページの文中に Java アプリケーションについての記述とそのアプリケーションをインストールするかどうかを問う強調表示されたタグ (またはボタン) を見つけることができます。タグには、アプリケーションの記述子ファイルへの参照が含まれています。記述子ファイル (一般にファイル拡張子 `.jam` を持つ) は、名前と値の組み合わせから成るテキストファイルです。このファイルにより JAM は、選択した Java アプリケーションのダウンロードをユーザが試みる前に、そのアプリケーションがデバイスに正常にインストールできるかどうかを確認できます。このため、その Java アプリケーションをインストールできない場合、ユーザはデバイスにアプリケーションをインストールしようとする手間を省くことができます。Java アプリケーションには 10 〜 20K バイトが必要です。このファイルのサイズはわずか (数百バイト) です。このため、Java アプリケーション一式よりもダウンロードの方がはるかに安上がりです。
2. インストール処理を開始するタグを選択します。ブラウザが、Web サイトから記述子ファイルを検出します。
3. 記述子ファイルの内容とブラウズされているページの URL を渡すことにより、ブラウザがプログラムの制御を JAM に引き渡します。

4. JAM がデバイス上にそのアプリケーションが既に存在するかどうかと、そのバージョン番号を確認します (後述のアプリケーションの更新の説明を参照)。続いて JAM は、保存に必要な十分なスペースがあるかどうかを確認するため、Java アプリケーションの JAR-File-Size タグを読み取ります。

5. アプリケーションのインストールに必要な十分なスペースがある場合、JAM は記述子ファイル内の JAR-File-URL タグを使用して JAR ファイルの URL を取得し (JAR-File-URL タグが相対 URL の場合、JAM は記述子ファイルのベース URL を使用する場合がある)、HTTP を使用してダウンロード処理を開始します。続いて JAM は、デバイス上に JAR ファイルを格納します。

   ダウンロード処理に割り込みが発生する場合、JAM はそのアプリケーションがまったくダウンロードされなかったように部分的にダウンロードされたアプリケーションを破棄します。

6. JAM が、インストール済み Java アプリケーションのリストにそのアプリケーションを追加し、必要に応じてほかのネイティブツールにもそのアプリケーションを登録します。JAM は、JAR ファイルと共に次の情報を保存します。

   - JAR ファイルの名前
   - JAR ファイルがダウンロードされた絶対 URL
   - Java アプリケーションのメインクラス
   - アプリケーションの名前
   - アプリケーションのバージョン番号

   絶対 URL とバージョン番号は、アプリケーションの更新時にアプリケーションの一意性を確認するために使用されます (次の節を参照)。

   JAM のリファレンス実装では、そのデバイスに既にインストールされている Java アプリケーションのリストがユーザに示されます。このリストでは、インストールされたばかりのアプリケーションがすぐに実行できるように選択されています。

   しかし、Use-Once タグが yes に設定されている場合は、JAM はこのリストにそのアプリケーションを追加せず、そのアプリケーションをただちに起動します。

7. プロセスが進行している間に発生するエラーはすべて、JAM によって処理される必要があります。コンテンツプロバイダ用のヘルプページの URL は、記述子ファイルに含まれています。続いて JAM は、ネイティブブラウザを使用してユーザにこの URL を示すことができます。

アプリケーションの起動

この節では、Java アプリケーションの一般的な起動方法を説明します。

1. ユーザに、Java アプリケーションのリストが示されます (ユーザインタフェースの設計はメーカに任されている)。リファレンス実装では、デバイス上にインストールされている Java アプリケーションのリストがユーザに示されます。このリストでは、インストールされたばかりのアプリケーションがすぐに実行できるように選択または強調表示されています。
2. ユーザが、起動したい Java アプリケーションを選択します (ユーザインタフェースの設計はメーカに任されている)。
3. JAM が、アプリケーションのメインクラスを含むパラメータを使用して KVM を実行します。KVM がそのメインクラスを初期化し、その実行を開始します。アプリケーションの実行のためにクラスがさらに必要な場合、KVM はメーカー定義の API を使用して、格納されている JAR ファイルからクラスファイルのアンパックとロードを行います。
4. Java アプリケーションがユーザの画面に表示されます。
5. アプリケーションが終了した時点で記述子ファイル内の `Use-Once` タグが `YES` に設定されている場合、JAM は JAR ファイルを削除します。

アプリケーションの更新

コンテンツプロバイダが (バグの修正や新機能の追加などの目的で) アプリケーションを更新する場合、コンテンツプロバイダは次の処理を行う必要があります。

1. アプリケーションに新しいバージョン番号を割り当てます。
2. 新しいバージョン番号を使用するようにアプリケーションの記述子ファイルを変更します。
3. アプリケーションの以前のバージョンと同じ JAR-File-URL タグを使用して、更新された JAR ファイルをコンテンツプロバイダの Web サイトに載せます。

ユーザがアプリケーションのインストールを要求する場合、JAM はそのアプリケーションの JAR-File-URL がインストール済みアプリケーションのものと同じであるかをチェックします。同じ場合、要求されたバージョンの Application-Version がインストール済みバージョンよりも新しいときは、JAM は新バージョンのアプリケーションのダウンロードとインストールを開始する前にユーザに確認を求めます。

リファレンス実装には、次の書式でバージョン番号を指定する文字列を使用します。

`Major.Minor[.Micro] (X.X[.X])`
`Micro` の部分はオプションです (デフォルトは 0)。バージョン番号のそれぞれの部分には、最大 2 桁の十進数を使用できます (つまり範囲は 0 〜 99)。

たとえば、アプリケーションの最初のバージョンを示すには 1.0.0 を使用できます。バージョン番号の各部とも、先行するゼロは有効な数字ではありません。たとえば、08 は 8 と指定するのと同じです。また、1.0 は 1.0.0 と指定するのと同じです。しかし、1.1 は 1.1.0 に相当し、1.0.1 ではありません。

リファレンス実装では、Application-Version タグが欠如していると 0.0.0 と想定されます。つまり、ゼロ以外のバージョン番号は、そのアプリケーションの新しいバージョンと見なされます。

アプリケーションの更新が何らかの理由で失敗する場合、JAM はそのデバイス上に古いバージョンが元のまま残っているか確認する必要があります。更新が成功した場合、古いバージョンは削除されます。

JAM コンポーネント

セキュリティ要件

JAM、そのデータ、および関連ライブラリは、デバイス上に安全に格納されなければなりません。デバイスメーカーは、これらのコンポーネントが Java アプリケーションなどのダウンロード可能なコンテンツによって変更されないように手段を講ずる必要があります。

JAR ファイル

JAR ファイルは、クラスファイルとアプリケーションリソースデータを圧縮形式で保持するように設計された標準の Java 機能です。JAM 準拠の JAR ファイルには、Java アプリケーション 1 つとその関連リソースが入ります。圧縮された JAR ファイルは、アプリケーションサイズをおよそ 40% 〜 50% 減らします。これにより、デバイスのストレージ要件とアプリケーションのダウンロード時間の両方が軽減されます。JAR ファイル内のアイテムは、必要に応じ JAM によってアンパックされます。

アプリケーション記述子ファイル

アプリケーション記述子ファイルは、読み取り可能なテキストファイルです。このファイルは、関連付けられた Java アプリケーションの重要な情報を説明した名前と値のペアから構成されます。このファイルは、コンテンツプロバイダの Web ページでタグから参照されます。このファイルの作成と管理は Java アプリケーションの開発者によって行われ、同じ Web サイト上にそのアプリケーション JAR ファイルと共に格納されます。開発者は、このファイルを任意のテキストエディタを使用して作成できます。

記述子ファイルには、以下のエントリが含まれます (タグ名は大文字と小文字の区別が必要)。

`Application-Name`

表示可能なテキストであり、デバイスの画面の幅に収まる長さにします。

`Application-Version`

`Major.Minor[.Micro] (X.X[.X]`
X は 1 桁または 2 桁の十進数で、`.Micro` の部分は必要に応じて入力します。

`KVM-Version`

CLDC microedition.configuration システムプロパティの定義に従ってコンマで区切られた KVM バージョンの文字列のリストです (『CLDC Specification』を参照)。この KVM バージョン文字列は、たとえば「CLDC-1.0」と指定できます。リスト内の項目はデバイス上の KVM バージョン文字列と一致し、このアプリケーションを実行するためには正確に一致している必要があります。デバイス上の KVM バージョン文字列に一致する項目はすべて、この条件を満たします。たとえば、「CLDC-1.0, CLDC-1.1」はデバイス上のこのどちらの KVM バージョンでも動作します。

`Main-Class`

アプリケーションの Main クラスの名前を標準の Java 形式で示したものです。

`JAR-File-Size`

バイト単位で示した整数

`JAR-File-URL`

ソース URL を指定する標準の URL テキスト形式。これが相対 URL の場合、記述子ファイルの URL がベース URL です。

`Use-Once`

yes または no

`Help-Page-URL`

ヘルプページにアクセスするためにブラウザが使用する標準の URL テキスト形式

その他の要件と制限:

- 記述子ファイルの MIME タイプは `application/x-jam`、拡張子は `.jam` です。
- すべての URL は、Web ページのロード元である同じサーバを指す必要があります。
- JAM は、あとで使用される場合に備えて、記述子ファイルのコンテンツをメーカ固有の形式で格納する必要があります (アプリケーションの更新については、70 ページの「JAM を使用してアプリケーションをインストールする」の手順 6 と72 ページの「アプリケーションの更新」を参照)。

アプリケーション開発者は、記述子ファイルにアプリケーション固有の、任意の名前と値のペアを追加できます。このため、記述子ファイル内の値を変更することにより、配布時にアプリケーションを構成できます。したがって、複数の記述子ファイルが同じアプリケーション JAR ファイルを使用し、異なるアプリケーションパラメータを指定できます。

タグの形式は文字列ですが、上記の表に示されたタグと類似した形式にすることをお勧めします。値の形式は、アプリケーション固有の文字列です。

JAM を介して値を検出するために使用できる単純な API を次に示します。

```
public String GetApplicationParameter(String name)
```

ネットワーク通信

Java アプリケーションが HTTP 接続を試みる場合は常に、ネットワーク実装は JAM をチェックしてアプリケーションがダウンロードされたサーバの名前を見つける必要があります。これは、アプリケーションのダウンロード元である同じサーバに接続を行うためです。両方の URL のホスト名間で文字列比較が行われます。

アプリケーションのライフサイクル管理

Java アプリケーションのライフサイクルは、次のように定義されます。

- KVM タスクが起動され、(記述子ファイルの Main-Class エントリで指定されているように) Java アプリケーションのメインクラスを実行するように指示される
- Java アプリケーションが KVM タスクのコンテキスト内で動作し、ユーザイベントに応答する
- KVM タスクは自動的に終了するかまたは、強制終了されて Java アプリケーションを終了する

ここでは、論理的に識別できる実行単位として KVM を説明するために、タスクという用語を使用しています。実際のデバイスにおいては、KVM タスクは実際に使用するオペレーティングシステムのタスク、プロセス、またはスレッドとして実装できます。

プラットフォームによって機構は著しく異なるため、KVM のライフサイクルを制御する API 関数は指定していません。代わりに、すべての JAM 実装が以下の機能をサポートしていることが必要です。

- JAM 実装は、KVM タスクを起動し、Java アプリケーションのメインクラスの実行を開始できる
- JAM 実装は、KVM タスクを強制的に終了でき、必要に応じて KVM タスクの中断と再開ができる)
- KVM の中断、再開、および終了は、以下に示した手続きによって行える

KVM タスクの終了

KVM タスクは、それ自体で終了させることも、強制的に終了させることもできます。

アプリケーションは、Java メソッド `System.exit()` を呼び出してそれ自体で随意に終了できます。状況によっては、JAM が KVM を強制終了することを決定する場合があります。強制終了させる方法は、プラットフォームによって異なります。たとえば、JAM は一定時間後にアクティブになるウォッチドッグスレッドを生成できます。KVM が随意に終了したのではないことを検出した場合、ウォッチドッグスレッドは KVM を強制的に終了します。

強制終了では、JAM は KVM によって割り当てられているリソースをすべてアクティブに解放し、KVM タスクを終了します。終了手続きの詳細は、プラットフォームによって異なります。`exit()` または `kill()` を呼び出すだけですむプラットフォームもあれば、入念なクリーンアップが必要なプラットフォームもあります。

エラー処理

JAM は、Java アプリケーションのインストールと起動で検出されるすべてのエラーを処理する必要があります。エラー処理の方法は実装によって異なりますが、できればJAM がユーザと対話してエラーを解決できることが望まれます。このため記述子ファイルには、コンテンツプロバイダによって設定される Help-Page-URL というタグが入っています。JAM は、状況次第ではブラウザを起動し、ユーザにヘルプページを表示することができます。ヘルプページの情報により、ユーザはコンテンツプロバイダに問い合わせを行いアドバイスを受けることもできます。

エラー条件

以下に、起こり得るエラー条件と、ユーザにエラーについて説明するために表示するメッセージの例 (英語) を示します。メーカーは、自社のデバイスユーザインタフェースに適したメッセージを設計する必要があります。

- デバイスで使用できるストレージスペースの合計を超えるサイズのアプリケーションをユーザがインストールしようとしている

  NAMEOFAPP が大きすぎてこのデバイスでは実行できないため、インストールは行えません。

- デバイスのストレージの空き領域を超える (しかしストレージスペースの合計よりは小さい) サイズのアプリケーションをユーザがインストールしようとしている

  インストールするだけの十分な空き領域がありません。アプリケーションをどれか削除して、もう一度試してください。

- デバイスに既にインストールされているアプリケーションをユーザがインストールしようとしている

  NAMEOFAPP は既にインストールされています (ソフトウェアのボタンは、[OK] や [Launch] などの名前にするとよい。[Launch] は、デバイス上の既存のアプリケーションを実行する)。

- 使用しているデバイスで動作しないアプリケーションをユーザがインストールしようとしている

  NAMEOFAPP は、このデバイスでは動作しません。ほかのアプリケーションを選択してください (ソフトボタンラベル: [Back]、[Done])。

- ユーザがアプリケーションのインストールを試みているが、その Java アプリケーションの説明タグに構文エラーがあるか、あるいはインストールを失敗させる無効な書式を使用している

  インストールは失敗しました。ご利用の ISP に問い合わせてください。

- ユーザがアプリケーションのインストールを試みているが、アプリケーションの URL が不正であるかあるいはアクセス不能であるため、アプリケーションのインストールが失敗する

  NAMEOFAPP の URL が無効です。ご利用の ISP に問い合わせてください。

- ユーザがアプリケーションのインストールを試みているが、アプリケーションのサイズが記述子ファイルに示されたサイズと異なる。このため、アプリケーションを破棄する必要がある

  NAMEOFAPP は、その記述と一致していません。このアプリケーションは、無効である可能性があります。ご利用の ISP に問い合わせてください。

- ユーザがアプリケーションのインストールを試みているが、アプリケーションのダウンロード中に接続が切れ、アプリケーションはデバイスに正常にロードされない

  接続が切れ、インストールは完了しませんでした。もう一度インストールを行なってください (ソフトボタンラベル: [Install]、[Back])。

- URL 全体がデバイスに既に存在する URL に完全に一致するアプリケーションをユーザがインストールしようとしている

  JAM は両方のバージョンの番号をチェックし、ユーザに結果を示す必要があります。

- ユーザがアプリケーションの実行を試みているが、何らかの理由でアプリケーションが起動できない (JAM が KVM を実行するための新しい OS タスクの作成に失敗した場合など)

  NAMEOFAPP を起動できません。ご利用の ISP に問い合わせてください。

- ユーザがあるアプリケーションを実行し、アプリケーションがスクラッチパッドの保存を試みたが、失敗する

  データを保存できません。ご利用の ISP に問い合わせてください。

- ユーザがあるアプリケーションを実行していて、その実行中にアプリケーションがクラッシュまたはハングアップする。注: これは一般的なエラーです。

  予期しないエラーのために NAMEOFAPP は終了しました。