J2VMS: Exploiting OpenVMS from Java

User’s Guide & Release Notes

This manual covers installation, trouble shooting and release information for J2VMS, an interface for communicating between native OpenVMS and Java.

Revision/Update Information:	This is a new manual.
Operating System and Version:	OpenVMS Alpha Version 7.3-2
	OpenVMS I64 Version 8.2
Software Version:	J2VMS Version 1.3
	Java for OpenVMS Version 1.2 or higher

November 2008

Contents

TABLES

Preface

Manual Objectives

This manual attempts to describe, as completely as possible, the J2VMS calling interface.

Intended Audience

This manual is designed for programmers who are developing Java applications that must communicate directly with OpenVMS System Services and Run-Time Libraries (including customer libraries).

It is expected that readers will have a working knowledge of software design and development. It is encouraged that the reader also explore some of the Java documentation pointers included below.

Document Structure

This manual contains the following chapters

- Chapter 1 covers installation and removal of the software.
- Chapter 2 describes the many parts of J2VMS and how to use them
- Chapter 2 offers troubleshooting solutions to some common issues.

Associated Documents

The following documents may be useful when using J2VMS.

Java2 1.3.1 SE API Specification

HP OpenVMS RTL Library (LIB$) Manual: LIB$FIND_IMAGE_SYMBOL

J2VMS V1.3 API Specification

Optimizing Java Technology Software Performance on HP OpenVMS.

BLISS Language Reference Manual

Guide to HP Structure Definition Language

On-Line Examples

A collection of examples programs are present on line in SYS$COMMON:[SYSHLP.EXAMPLES.J2VMS]. (This device-directory specification may have been given the logical name J2VMS$EXAMPLES during installation of the J2VMS software kit.)

Reader's Comments

Kednos welcomes your comments on this manual. Please send any comments, corrections, etc. to either of the following addresses:

Email	pli-support@kednos.com
Postal Mail	Kednos Enterprises Suite 7, 220 Country Club Drive Pacific Grove, CA 93950

Release Notes

This section contains release information on J2VMS. Release information is necessary for gaining the best results from J2VMS. It is recommended that all users read this information.

J2VMS Version V1.3

Overview of Changes

J2VMS Version 1.3 contains the following enchancements and fixes:

- There is now support for passing a *java.lang.String* object by descriptor. However, it should be noted that there is no support (nor will there be in the future) for returning data via a *String* object. This is a restriction imposed by Java, not J2VMS.
- It is now possible to pass a *java.lang.StringBuffer* by descriptor. A *StringBuffer* object can be used to pass string data to a native routine as well as receive it from the called routine. Passing a *StringBuffer* by descriptor results in the native routine receiving a dynamic string descriptor. It is not currently possible to pass a *StringBuffer* by reference. However, support will be included in a future release.
- There is now support for passing read-only copies of primitive types and *java.lang.String* objects by reference.
- The class *vs.SystemServices* now includes a declaration for the System Service $PARSE.
- The STARLET module STS now includes $VMS_STATUS_* methods equivalent to the macros of the same name present in the C language environment.
- J2VMS is now distributed in a PCSI software product kit.
- The program that generates the STARLET library provided by J2VMS has been improved to generate modules that more accurately mirror those provided by native high level languages. It is possible, though unlikely, that some fields or constants may have moved into a different module. This will require that the source code be changed to match the new definitions.
- The J2VMS STARLET library (package *vs.starlet* now includes normalised names. As well as names appearing in their original case, upper and lower case names are also provided.
- This release includes a complete API specification not present in past releases. This is included in the software product kit as well as being accessible online at the Kednos website.

Overview of Restrictions

J2VMS Version 1.3 contains the following restrictions:

- There is currently no support for passing Java methods as callback or AST service routines. This is planned for a future release.
- It is not currently possible to pass a *java.lang.StringBuffer* object by reference. This will be added in a future release.
- It is not currently possible to easily regenerate the STARLET library on a target system and so the mechanism to do so is not provided. This will change in a future release.

1 Getting Started

This chapter covers the installation and removal of J2VMS including software and hardware dependencies.

1.1 Software Prerequisites

The following software is required to successfully install and use J2VMS:

- OpenVMS Alpha Version V7.3-2 or higher; or
- OpenVMS I64 Version V8.2 or higher
- Java Platform, Standard Edition, Development Kit (JDK) V1.2 or higher.

It is also recommended that all available software patches be applied to these products prior to attempting installation.

For improved performance later versions of the JDK are also recommended. Later versions include support for FastVM and other significant performance improvements.

1.2 Hardware Requirements

The only hardware requirement is that the system be either an Alpha or Integrity system as J2VMS (like Java) is only available on OpenVMS Alpha and I64.

Be aware that Java is particullarly resource hungry so older, slower machines will often perform poorly, particularly in interactive environments. The general rule of thumb is, the more memory and the faster the CPU the better Java applications will perform. There are some documents available that offer hints and tips for performance improvements. Details of these can be found under the section Associated Documents.

1.3 Installation

The J2VMS software product kit presents a number of options to the installer to control exactly what is installed. Table 1–1 describes each of these options and their defaults. Example 1–1 shows an example of an installation of the J2VMS product.

Table 1–1 J2VMS Installation Options

Description	Default
User Guide & Release Notes in PostScript format	Yes

Table 1–1 (Cont.) J2VMS Installation Options

Description	Default
User Guide & Release Notes in PDF format	Yes
User Guide & Release Notes in HTML format	Yes
J2VMS API Specification	Yes
Example programs	Yes

Example 1–1 J2VMS Product Installation

```
$ PRODUCT INSTALL J2VMS

The following product has been selected:
    KEDNOS VMS J2VMS V1.3                  Layered Product

Do you want to continue? [YES] YES

Configuration phase starting ...

You will be asked to choose options, if any, for each selected product and for
any products that may be installed to satisfy software dependency requirements.

KEDNOS VMS J2VMS V1.3: Java Interface to Native OpenVMS

    Copyright © 2001-2002 Jim Brankin, 2008 Kednos Enterprises

    Kednos Enterprises

    This product uses the PAK: J2VMS

Do you want the defaults for all options? [YES] NO

    Install J2VMS Documentation? [YES] YES

    Do you want the defaults for all suboptions? [YES] NO

      Install the J2VMS V1.3 API Specification [YES] YES

      Install J2VMS User Guide & Release Notes in HTML format? [YES] YES

      Install J2VMS User Guide & Release Notes in PDF format? [YES] YES

      Install J2VMS User Guide & Release Notes in PostScript format? [YES] YES

    Install example programs? [YES] YES

Do you want to review the options? [NO] NO

Execution phase starting ...

The following product will be installed to destination:
    KEDNOS VMS J2VMS V1.3                  DISK$AXP082:[VMS$COMMON.]

Portion done: 0%...10%...20%...30%...70%...90%...100%

The following product has been installed:
    KEDNOS VMS J2VMS V1.3                  Layered Product

KEDNOS VMS J2VMS V1.3: Java Interface to Native OpenVMS

    Relase Notes are included in the user guide.

    J2VMS release notes are included in the User Guide & Release Notes
    manual. To install these locally ensure that at least one documentation
    option is selected. Otherwise, the manual can be found at the Kednos
    website.

    Insert the following lines in SYS$MANAGER:SYSTARTUP_VMS.COM:
        @SYS$STARTUP:J2VMS$STARTUP.COM
```

1.4 Post Installation

Once the J2VMS software product kit has been installed it is advisable to add the startup procedure, SYS$STARTUP:J2VMS$STARTUP.COM to the system startup prodcedure. Althought it is not necessary in this kit, it may become so in a later release. It als defines the J2VMS$EXAMPLES logical and installs the SYS$LIBRARY:J2VMS$SHR.EXE image.

1.5 Removal

Removal of the software product is a case of using the PCSI PRODUCT REMOVE command. There is no provision in this release for clean up of the API Specification in this kit (it is expected this will change in a future release). If the API spec was unpacked it will have to be removed manually.

Example 1–2 demonstrates removal of the J2VMS product.

Example 1–2 J2VMS Product Removal

```
$ PRODUCT REMOVE J2VMS

The following product has been selected:
    KEDNOS VMS J2VMS V1.3                 Layered Product

Do you want to continue? [YES] YES

The following product will be removed from destination:
    KEDNOS VMS J2VMS V1.3                 DISK$AXP082:[VMS$COMMON.]

Portion done: 0%...40%...50%...60%...70%...80%...100%

The following product has been removed:
    KEDNOS VMS J2VMS V1.3                 Layered Product
```

2 Developing An Application

This chapter covers the different parts of the J2VMS interface and how It has been deliberately written to draw comparisons between native coding and coding in Java using J2VMS. The intention is that it will be easier to understand for someone who has more background in native languages such as PL/I, BASIC, C and Pascal.

2.1 Includes

The first item to cover is how to make the J2VMS interface available to a Java program. It is quite simple and is tackled in two steps. The first is configuring the Java classpath. Without this the Java compiler and interpreter will not be able to find the J2VMS classes. The example below demonstrates how to include that J2VMS library in the JAVA$CLASSPATCH logical.

```
$ DEFINE JAVA$CLASSPATH SYS$LIBRARY:J2VMS$VS.JAR
```

Another way to configure the Java classpath is to include it on the command line as shown in this example:

```
$ java -classpath "/sys$library/j2vms$vs.jar:..." ...
```

For more information on configuring the Java classpath, please consult the relevant Java for OpenVMS user guide.

Once the compiler and interpreter can find the J2VMS class library the relevant classes need to be included in the source module. It is recommended that the base *vs* package be included in it's entirety. It it not normally recommended to include external packages in this way. However, the *vs* package is not made up of many classes.

When it comes to including the different modules within STARLET it is best to only include those classes that are necessary. There are many modules in the *vs.starlet* package. This will cause unnecessary memory usage and increase compile and load time significantly.

The example below demonstrates the best way to include the *vs* package and the STARLET modules SS and STS.

```
import vs.*;
import vs.starlet.SS;
import vs.starlet.STS;
```

2.2 External Routines

In order for J2VMS to call native routines they first need to be declared. This can be likened to declaring a routine in C with the *extern* attribute or declaring an *external entry* in PL/I. The difference in Java is that there is no linker to process the external declarations and locate the relevant shared image. Therefore the caller needs to know which shareable image the routines exists in.

The external declaration is constructed using the *SystemCall*. class. This class is a simplified method for calling the LIB$ Run-Time Library routine LIB$FIND_IMAGE_SYMBOL (sometimes referred to as LIB$FIS). LIB$FIND_IMAGE_SYMBOL dynamically loads shareable images by looking up symbols.

Declaring An External Reference to LIB$PUT_OUTPUT

The following example below demonstrates how to declare the external routine LIB$PUT_OUTPUT (found in SYS$LIBRARY:LIBRTL.EXE) in Java. It also includes examples in C, PL/I and BASIC for comparison.

1
```
SystemCall lib$put_output = new SystemCall("LIB$PUT_OUTPUT", "LIBRTL");
```

The same declaration in C

2
```
extern int lib$put_output();
```

The same declaration in PL/I

3
```
declare lib$put_output entry(any, any) returns(fixed binary(31));
```

The same declaration in BASIC

4
```
external integer function lib$put_output
```

The Structure Definition Language (SDL) can be used to generate external routine declarations also. These classes can then be used in the same way as the classes *vs.SystemServices* and *vs.LibRoutines*. These classes are covered further in Section 2.4.

2.3 Argument Passing Mechanisms

Before discussing the actual call mechanism between J2VMS and native OpenVMS it is important to first cover argument passing. J2VMS provides a set of classes that allow the caller to pass arguments in the common language environment, just as they would a native language. These classes are:

- *vs.ByDesc* - by descriptor
- *vs.ByRef* - by reference
- *vs.ByVal* - by value

Passing arguments using J2VMS can be likened to constructing an argument list for the LIB$ Run-Time Library function LIB$CALLG. An array of pointers and/or values is constructed and passed to the target routine using LIB$CALLG as a catalyst (originally high level language access to the VAX CALLG instruction). J2VMS is quite similar. However, it has it’s minor differences mostly related to the object oriented architecture of Java.

J2VMS argument lists are constructed as an array of objects from the abstract class *vs.VMSparam*. Unlike an argument list destined for LIB$CALLG there is no need for the argument count in the first element as the size of the array is known, thanks to Java. This means that all elements in the argument list detail arguments. Each element consists of an argument mechanism class (shown above) that informs J2VMS as to exactly how it's own argument is to be passed. When J2VMS actually performs the call it determines the mechanism by comparing class types and then allocating internal storage as necessary. Copying between internal storage and Java class storage before and after the actual native call.

vs.VMSparam Argument List for LIB$CHAR

1

```
StringBuffer result = new StringBuffer();
Byte code = new Byte((byte) 65);
vs.VMSparam arglst = new VMSparam[] { new ByDesc(result),
                                      new ByRef(code) };
```

The LIB$ Run-Time Library routine LIB$CHAR accepts two arguments. The first is a result string, passed by descriptor, and the second is a byte containing an 8-bit ASCII character code.

2

```
declare arglst(2) pointer;
declare 1 result,
        2 dsc$w_length fixed binary(15),
        2 dsc$b_dtype fixed binary(7),
        2 dsc$b_class fixed binary(7),
        2 dsc$a_pointer pointer;
declare result_str character(dsc$w_length) based(dsc$a_pointer);
declare code fixed binary(7);

result.dsc$w_length = 0;
result.dsc$b_dtype = DSC$K_DTYPE_T;
result.dsc$b_class = DSC$K_CLASS_D;
result.dsc$a_pointer = null();

arglst(1) = addr(result);
arglst(2) = addr(code);
```

The above snippet of PL/I code attempts to demonstrate the actions of the code above in a native language. In this example the string descriptor for *result* is constructed explictly to show what J2VMS does internally when dealing with an argument passed by descriptor.

The following sections cover each of the mechanisms in close. These sections must be read carefully as there are one or two caveats that must be observed. For full documentation of what data types each passing mechanism class can deal with, please consult the J2VMS API Specification. This documentation ships with the software product and is available at the Kednos website. See Associated Documents for more details.

2.3.1 Passing Arguments By Descriptor

The class *vs.ByDesc* is used to inform J2VMS that it's argument should be passed by descriptor. Many data types can be passed by descriptor. However, probably the most used are *vs.Cmem*, *java.lang.StringBuffer* and *byte[]*. Both *vs.Cmem* and *byte[]* result in fixed length descriptors of type DSC$K_CLASS_S. *java.lang.StringBuffer* is a special case. It results in a dynamic string descriptor with a type of DSC$K_CLASS_D. Prior to a call the content of the *StringBuffer* object is copied into a dynamic descriptor. Then following the return from the call the string descriptor is copied back into the *StringBuffer* object so the result it can be used in the Java environment.

All objects and arrays passed by descriptor can be used to receive results, with one exception. The *vs.ByDesc* constructor for the *java.lang.String* object cannot be used to receive a result. This is a restriction imposed by Java, not J2VMS. There is no public method for updating the contents of a *String* object. In the case where a string object is passed by descriptor it results in a new byte array being allocated and the contents of the *String* object being copied into it. The example below demonstrates what actually happens.

Example 2–1 shows a literal *java.lang.String* object being passed to the LIB$ Run-Time Library routine LIB$PUT_OUTPUT. This example demonstrates the convenience of being able to pass string literals and have them built "in the argument list".

Example 2–1 Passing a ***java.lang.String*** **Object by Descriptor**

```
lib.lib$put_output(new VMSparam[] {
                         new ByDesc("hello, " + "world")
                   });
```

However, it is not possible to receive data into a *String* object. Example 2–2 is a complete program that can be copied and executed to demonstrate the restriction. This example is also available from the Kednos website, or online at SYS$COMMON:[SYSHLP.EXAMPLES.J2VMS]GET2.JAVA.

Table 2–1 summarises the native usage of Java object types passed by descriptor.

Example 2–2 Receiving String Data From A Called Routine

```
import java.lang.*;
import vs.*;

public class get2
{
    public static void main(String[] args)
    {
        Short result_len = new Short((short) 0);
        String result1 = new String();
        StringBuffer result2 = new StringBuffer();
        LibRoutines lib = new LibRoutines();

        System.out.println("Storage before any calls to LIB$PUT_OUTPUT");
        System.out.println("  * result_len = " + result_len);
        System.out.println("  * result1 = <" + result1 + ">");
        System.out.println("  * result2 = <" + result2 + ">");

        lib.lib$get_input(new VMSparam[] {
                              new ByDesc(result1),
                              new ByDesc("String result>> "),
                              new ByRef(result_len)
                          });

        System.out.println("Storage after first call to LIB$PUT_OUTPUT");
        System.out.println("  * result_len = " + result_len);
        System.out.println("  * result1 = <" + result1 + ">");
        System.out.println("  * result2 = <" + result2 + ">");

        lib.lib$get_input(new VMSparam[] {
                              new ByDesc(result2),
                              new ByDesc("String result>> "),
                              new ByRef(result_len)
                          });

        System.out.println("Storage after second call to LIB$PUT_OUTPUT");
        System.out.println("  * result_len = " + result_len);
        System.out.println("  * result1 = <" + result1 + ">");
        System.out.println("  * result2 = <" + result2 + ">");
    }
}
```

Table 2–1 Summary of Native Usage of Java Objects Passed by Descriptor

Java Class/Primitive Type	Descriptor Class	Writable
byte[ ]	Static (DSC$K_CLASS_S)	Yes
int[ ]	Static (DSC$K_CLASS_S)	Yes
long[ ]	Static (DSC$K_CLASS_S)	Yes
short[ ]	Static (DSC$K_CLASS_S)	Yes
java.lang.Byte	Static (DSC$K_CLASS_S)	Yes
java.lang.Integer	Static (DSC$K_CLASS_S)	Yes
java.lang.Long	Static (DSC$K_CLASS_S)	Yes
java.lang.Short	Static (DSC$K_CLASS_S)	Yes
java.lang.String	Static (DSC$K_CLASS_S)	No
java.lang.StringBuffer	Dynamic (DSC$K_CLASS_D)	Yes
vs.Cmem	Static (DSC$K_CLASS_S)	Yes
vs.VmsStruct	Static (DSC$K_CLASS_S)	Yes

Note: There is currently no support for passing arrays by descriptor in the same way as other native high level languages. Currently all arrays of primitive types are treated as *byte[ ]* arrays. However, support is planned in a future release.

2.3.2 Passing Arguments By Reference

The J2VMS class *vs.ByRef* is used to pass arguments by reference (by address). It can be used to pass many data types. Table 2–2 summarises each of the supported types and their usage by native routines.

Table 2–2 Summary of Native Usage of Java Objects Passed by Descriptor

Java Class/Primitive Type	Writeable
byte	No
int	No
long	No
short	No
byte[ ]	Yes
int[ ]	Yes
long[ ]	Yes
short[ ]	Yes
java.lang.Byte	Yes
java.lang.Integer	Yes
java.lang.Long	Yes

Table 2–2 (Cont.) Summary of Native Usage of Java Objects Passed by Descriptor

Java Class/Primitive Type	Writeable
java.lang.Short	Yes
java.lang.String	No
vs.Cmem	Yes
vs.VmsStruct	Yes

Note: There is currently no support for passing a *java.lang.StringBuffer* object by reference. However, this will appear in a future release of J2VMS.

All object and arrays passed by reference can be used to receive results. There is support for passing primitive types by reference. However, these are convenience constructors and cannot be used to receive a result. This is much the same problem as passing a *java.lang.String* object by descriptor (seen in the Section 2.3.1). Example 2–3 demonstrates the usefulness of being able to pass primitive types and *java.lang.String* objects as routine inputs.

Example 2–3 Copying A *java.lang.String* Object to Descriptor

```
import java.lang.*;
import vs.*;
import vs.starlet.DSC;

public class desc
{

    public static void main(String args[])
    {
        LibRoutines lib = new LibRoutines();
        VmsStruct descriptor = new VmsStruct(DSC.DSC$C_D_BLN);
        String buffer = "hello, world";

        descriptor.put(DSC.DSC$W_MAXSTRLEN, 0);
        descriptor.put(DSC.DSC$B_DTYPE, DSC.DSC$K_DTYPE_T);
        descriptor.put(DSC.DSC$B_CLASS, DSC.DSC$K_CLASS_D);
        descriptor.put(DSC.DSC$A_POINTER, 0);

        lib.lib$scopy_r_dx(new VMSparam[] {
                               new ByRef(buffer.length()),
                               new ByRef(buffer),
                               new ByRef(descriptor)
                           });

        lib.lib$put_output(new VMSparam[] {
                               new ByRef(descriptor)
                           });
    }

}
```

2.3.3 Passing Arguments By Value

The J2VMS class *vs.ByVal* is used to pass arguments by value. Table 2–3 lists the supported argument types passed by value.

Table 2–3 Java Classes And Primitive Types That Can Be Passed By Value

Java Class/Primitive Type
byte
int
long
short
java.lang.Byte
java.lang.Integer
java.lang.Long
java.lang.Short

Arguments passed by value can be altered by called routines. However, as with compiled languages the results are lost on return.

2.4 Calling a Native Routine

Calling the native routine is done by passing an array of *vs.VMSparam* objects to the *call* method of the *vs.SystemCall*.

Example 2–4 is a complete example that builds on the previous sections to declare an external routine reference, construct an argument list and finally call the native routine.

There are times when it is not convenient to declare many routines individually within each method that uses them. Libraries of external routine references can be gather in a class. The most popular examples of this are probably the *vs.SystemServices* and *vs,LibRoutines* classes. Similar libraries can also be constructed using the Java language backend for the Structure Definition Language (SDL) compiler.

Example 2–5 shows the code in Example 2–4 after it has been rearranged to take advantage of the *vs.LibRoutines* class.

As can be seen in Example 2–5 there is no longer any need to call the *call* method. The name of the routine is now used to make the call. The advantage of using the *vs.SystemServices* and *vs.LibRoutines* classes or classes generated from SDL is that it is now possible to call a range of routines present in the class.

Example 2–4 Calling LIB$CHAR from Java

```
import java.lang.*;
import vs.*;

public class chr
{
    public static void main(String[] args)
    {
        /* Declare result storage, essentailly a dynamic string
         * string descriptor.
         */
        StringBuffer result = new StringBuffer();

        /* Declare external reference to routine LIB$CHAR.
         */
        SystemCall lib$char = new SystemCall("LIB$CHAR", "LIBRTL");

        /* Call the native routine. Here the argument list is constructed
         * inline so that it looks more like a regular call.
         */
        lib$char.call(new VMSparam[] {
                          new ByDesc(result),
                          new ByRef(65)
                      });

        /* Output the result received from LIB$CHAR.
         */
        System.out.println("ASCII 65 = " + result);
    }
}
```

Example 2–5 Calling LIB$CHAR from Java Using *vs.LibRoutines*

```
import java.lang.*;
import vs.*;

public class chr2
{
    public static void main(String[] args)
    {
        /* Declare result storage, essentailly a dynamic string
         * string descriptor.
         */
        StringBuffer result = new StringBuffer();

        /* "Include" vs.LibRoutines. This is similar to C where the
         * following might be used to include all routines from
         * SYS$LIBRARY:LIBRTL.EXE.
         */
        LibRoutines lib = new LibRoutines();

        /* Call the native routine. Here the argument list is constructed
         * inline so that it looks more like a regular call.
         */
        lib.lib$char(new VMSparam[] {
                         new ByDesc(result),
                         new ByRef(65)
                     });

        /* Output the result received from LIB$CHAR.
         */
        System.out.println("ASCII 65 = " + result);
    }
}
```

2.5 Structure Declaration

The Java language has support for classes. These can be likened to OpenVMS native structures. However, as far as physical storage and arrangement there is very little that is similar. It is not possible to easily and uniformly map a native OpenVMS data structure to a Java class. Some element of manual human intervention is required. This is why J2VMS provides the *vs.VmsStruct* and *vs.FieldDescriptor* classes. With the use of these two classes access to OpenVMS data structures becomes simple uniform and easy to understand.

Both these classes are based on the mechanism used by the BLISS system programming language for accessing structures. To read further on the BLISS language please see the Associated Documents section.

Structures are allocated internally as an array of *byte*s and accessed through the class *vs.VmsStruct*. The target *byte* array can be allocated by the class constructor or supplied by the caller. From this point forward it is now possible to manipulate the byte array using the *put* and *get* methods of the *VmsStruct* class.

Example 2–6 demonstrates how to allocate storage for an Record Management System (RMS) File Access Block (FAB) structure in Java.

Example 2–6 Allocating a FAB in Java

```
import vs.*;
import vs.starlet.FAB;
   .
   .
   .
VmsStruct fab = new VmsStruct(FAB.FAB$S_FABDEF);
```

Example 2–7 demonstrates how to allocate the same structure in the C language.

Example 2–7 Allocating a FAB in C

```
#include <fabdef.h>
   .
   .
   .
struct FAB fab;
```

Lastly, Example 2–8 demonstrates how to declare a FAB in the BLISS language. The BLOCK attribute can be thought of as the equivalent of the *Vs.VmsStruct* constructor.

Example 2–8 Allocating a FAB in BLISS

```
library ’sys$library:starlet’;
   .
   .
   .
local
    fab : block[FAB$S_FABDEF, byte];
```

2.6 Structure Manipulation

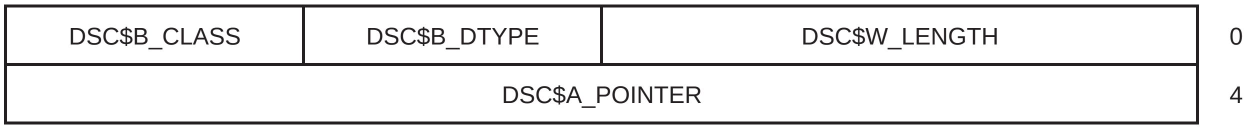
Structures allocated with the *vs.VmsStruct* class can be manipulated using the *get* and *put* methods. These methods are used, combined with *vs.FieldDescriptor* objects, to alter specific regions of the *byte* array target of the *VmsStruct* object. Each object declares a storage area described in terms of bit and byte offsets from the beginning of a byte array.

Although cumbersome and a little involved, *vs.FieldDescriptor* objects do allow complete access to any part of the *byte* array managed by a *VmsStruct* class. It is recommended that the Structure Definition Language (SDL) be used to declare data structures that can then be translated into *FieldDescriptor* declarations which can be used with J2VMS.

For further detail on using the SDL language and compiler as well as details of the J2VMS extension to the SDL compiler, see the Associated Documents section at the beginning of this manual.

Figure 2–1 describes that layout of a string descriptor in memory. Example 2–9 demonstrates how this data structure is declared using *vs.FieldDescriptor* objects.

Figure 2–1 String Descriptor Layout



Example 2–9 String Descriptor Declaration in Java

```
import vs.FieldDescriptor;

public class DSCDEF
{
    public static final FieldDescriptor dsc$w_length =
                                        new FieldDescriptor(0, 0, 16, 0);
    public static final FieldDescriptor dsc$b_dtype =
                                        new FieldDescriptor(2, 0, 8, 0);
    public static final FieldDescriptor dsc$b_class =
     new FieldDescriptor(3, 0, 8, 0);
    public static final FieldDescriptor dsc$a_pointer =
     new FieldDescriptor(4, 0, 32, 0);
}
```

Example 2–10 demonstrates the same declaration in BLISS. It is easy when comparing these examples to see the heritage of the J2VMS method.

Example 2–10 String Descriptor Declaration in Java

```
field dscdef =
set
    dsc$w_length  = [  0,  0, 16,  0 ],
    dsc$b_dtype   = [  2,  0,  8,  0 ],
    dsc$b_class   = [  3,  0,  8,  0 ],
    dsc$b_pointer = [  4,  0, 32,  0 ]
tes;
```

Lastly, Example 2–11 shows the same declaration again. However, this time it is done using PL/I.

Example 2–11 String Descriptor Declaration in PL/I

```
declare 1 dscdef based,
        2 dsc$w_length fixed binary(15),
        2 dsc$b_dtype fixed binary(7),
        2 dsc$b_class fixed binary(7),
        2 dsc$a_pointer pointer;
```

2.6.1 Cmem

The J2VMS class *vs.Cmem* is an interface to the C Run-Time Library memory allocation and deallocation routines *malloc* and *free*. This class is particularly useful when it comes to allocate storage that must evade the Java garbage collector. An instance of this might be to allocate storage that is referenced by a structure.

Example 2–12 demonstrates a common use of the Cmem class. In this example Cmem is used to allocate storage for a NAML block that can then be referenced by a FAB block. The output storage for the resultant filename is also a Cmem class.

Example 2–12 *vs.Cmem* **Usage Demonstration**

```
import java.lang.*;
import java.lang.reflect.Array;
import vs.*;
import vs.starlet.FAB;
import vs.starlet.NAM;
import vs.starlet.SS;
import vs.starlet.STS;

public class parse
{
    public static void main(String[] args)
    {
        VmsStruct fab = new VmsStruct(FAB.FAB$S_FABDEF);
        VmsStruct naml = new VmsStruct(NAM.NAML$S_NAMLDEF);
        Cmem cfab = new Cmem(FAB.FAB$S_FABDEF);
        Cmem cnaml = new Cmem(NAM.NAML$S_NAMLDEF);
        Cmem long_filename;
        Cmem long_result = new Cmem(NAM.NAML$C_MAXRSS);
        int sstatus = SS.SS$_NORMAL;
        SystemServices sys = new SystemServices();

        if ((Array.getLength(args) <= 0)
            || args[0].toUpperCase().startsWith("-H"))
        {
            System.out.println("Usage:");
            System.out.println("  [java-cmd] parse -h | <filename>");
            return;
        }

        /* Allocate memory region for input filename and copy it in.
         */
        long_filename = new Cmem(args[0].length());
        long_filename.copyin(args[0]);

        /* Initialize tha NAML...
         */
        naml.put(NAM.NAML$B_BID, NAM.NAML$C_BID);
        naml.put(NAM.NAML$B_BLN, NAM.NAML$K_BLN);
        naml.put(NAM.NAML$L_LONG_FILENAME, long_filename.getPeer());
        naml.put(NAM.NAML$L_LONG_FILENAME_SIZE, long_filename.length());
        naml.put(NAM.NAML$L_LONG_RESULT, long_result.getPeer());
        naml.put(NAM.NAML$L_LONG_RESULT_ALLOC, long_result.length());
```

Example 2–12 Cont'd on next page

Example 2–12 (Cont.) *vs.Cmem* **Usage Demonstration**

```
        /* Initialize the FAB...
         */
        fab.put(FAB.FAB$B_BID, FAB.FAB$C_BID);
        fab.put(FAB.FAB$B_BLN, FAB.FAB$K_BLN);
        fab.put(FAB.FAB$W_IFI, 0);
        fab.put(FAB.FAB$L_FNA, -1);
        fab.put(FAB.FAB$B_FNS, 0);
        fab.put(FAB.FAB$L_NAM, cnaml.getPeer());

        /* Copy the NAML block into the memory storage.
         */
        cnaml.copyin(naml);

        /* Now we do the parse...
         */
        sstatus = sys.sys$parse(new VMSparam[] {
                                    new ByRef(fab),
                                    new ByVal(0),
                                    new ByVal(0)
                                });
        if (!STS.$VMS_STATUS_SUCCESS(sstatus))
        {
            sys.sys$exit(new VMSparam[] {
                            new ByVal(sstatus)
                        });
        }

        /* Copy the NAML from the Cmem object back to the VmsStruct
         *.object.
        cnaml.copyout(naml);

        System.out.println("Original specification = "
                          + long_filename.copyout());

        System.out.println("Resultant specification = "
            + long_result.copyout((int)naml.get(NAM.NAML$L_LONG_RESULT_SIZE))
    }
}
```

2.7 OpenVMS Status Codes

J2VMS provides a set of methods in the STARLET class STS for accessing the different fields of a status code within a longword. These methods are based on the C macros provided the STSDEF module of SYS$LIBRARY:SYS$STARLET_C.TLB. For further documentation on the available methods please consult the J2VMS API Specification. A pointer to this documentation can be found under Associated Documents.

Example 2–13 demonstrates probably the most common use of the *$VMS_STATUS_SUCCESS* method.

Note: All methods that return a field of only one bit convert their results to *boolean*. **This makes it easier to use these methods as an** *if* **expression. All other methods return** *int* **results.**

Example 2–13   Using $VMS_STATUS_SUCCESS

```
import java.io.*;
import java.lang.*;
import vs.*;
import vs.starlet.SS;
import vs.starlet.STS;

public class status
{
    public static void main(String[] args)
    {
        int sstatus = SS.SS$_ACCVIO;

        if (STS.$VMS_STATUS_SUCCESS(sstatus))
            System.out.println("Status indicates success!");
        else
            System.out.println("Status indicates failure!");
    }
}
```

3 Troubleshooting

The following section covers common issues that are encountered by usesrs of J2VMS.

3.1 java.lang.UnsatisfiedLinkError

This error indicates a failure in the Java method *java.lang.System.loadLibrary*. This method is used for loading native libraries in Java. It is not the method used by *vs.SystemCall* to load libraries. See Section 3.2 for more information on that topic.

If the error looks similar to:

```
java.lang.UnsatisfiedLinkError: no J2VMS$SHR in java.library.path
        at java.lang.ClassLoader.loadLibrary(ClassLoader.java:1578)
        at java.lang.Runtime.loadLibrary0(Runtime.java:788)
        at java.lang.System.loadLibrary(System.java:834)
        at vs.VmsStruct.<clinit>(VmsStruct.java:81)
        at parse.main(parse.java:13)
```

This is normally an indication that the J2VMS startup procedure, SYS$STARTUP:J2VMS$STARTUP.COM, has not been run. When Java loads native libraries using *java.lang.System.loadLibrary* it does not search in SYS$LIBRARY like LIB$FIND_IMAGE_SYMBOL. It uses logical names only.

3.2 java.lang.NoClassDefFoundError

This error most commonly indicates a mistake in the classpath. There may be a mispelt filename, or a Java archive missing entirely. However, it can also indicate a failure in *vs.SystemCall* when loading a routine from a shareable image. The best action to take is to check that the shareable image is either in SYS$SHARE or has an appropriately defined logical.

3.3 Cannot Resolve Symbol

The Java program compiled under a previous version of J2VMS. However, since upgrading to version V1.3 or higher the Java compiler reports errors similar to the following:

```
parse.java:5: cannot resolve symbol
symbol  : class NAML
location: package starlet
                  ^
parse.java:14: cannot resolve symbol
symbol  : variable NAML
location: class parse
```

It is likely this is caused by changes in the STARLET library provided by J2VMS. Version V1.3 and higher includes changes to the STARLET library generator so that is is more closely inline with those provided with native languages. To correct this, simply adjust the name of the class to the one that includes the definition. In the case above, the definitions for the NAML block are included in the class *vs.starlet.NAM*.