The Squawk System

Preliminary Draft 2.1

16 September 2002

Sun Microsystems Laboratories

2600 Casey Avenue

Mountain View, CA 94043

Table of Contents

CHAPTER 1

Introduction

The Squawk project began as a Sun Labs research effort aimed at constructing a small, CLDC-compatible Java™ implementation written as much as possible in the Java language (following the model of the Squeak implementation of Smalltalk - see http://www.squeak.org).

As work progressed it became clear that: the CLDC preverifier step (which transforms input classfiles by removing the jsr and ret bytecodes and adding stack maps) could perform more radical transformations, making classfiles considerably smaller and more suitable for small devices; and that carefully implementing as much of the system in Java made the virtual machine small and the entire system more easily maintained.

The initial Squawk prototype was successfully demoed in April of 2002, and further, more pragmatic work based on it was considered. Next generation smart card platforms with 32-bit processors presented both significant challenges (especially with respect to heap space) and opportunities, and became the focus of further Squawk work, in collaboration with the Java Card™ team at Sun.

This document presents the Squawk system design aimed at next generation smart cards. It has been optimized to minimize various on-device and provisioning memory requirements. The key technologies embodied in the design are:

- a compact representation for a collection of classfiles called a suite;
- a compact bytecode set based on the standard Java bytecodes; and
- transformations that simplify garbage collection and bytecode verification.

The system is structured around these ideas. There is an off-device classfile transformer that transforms standard classfiles into suites, an on-device suite loader that loads suites into memory, and a small virtual machine that executes loaded suites.

The remainder of this document defines these technologies and components.

CHAPTER 2

Features and Limitations

The overall goal of the Squawk system is to provide CLDC functionality in a very resource constrained environment. It does this through a number of features.

Features

On-Device Dynamic Loading and Linking

The Squawk system dynamically loads and links program components on the target device.

On-Device Verification

The system verifies program components when they are loaded on the device, providing the same verification guarantees as the CLDC.

Exact Garbage Collection

The system performs garbage collection similar to that found on desktop systems. Specifically, it knows the location of all references and can trace and collect all objects exactly.

EEPROM-Aware Memory System

The system divides memory into three spaces, RAM, EEPROM, and ROM, and manages the three spaces accordingly.

This feature is still under development.

Extremely Compact Classfile Representation

The Squawk system transforms classfiles into a more compact representation that also enables the above features. This compact representation means that:

- off-device storage requirements are reduced;
- transmission requirements are reduced; and
- on-device loading and storage requirements are reduced.

Java Card CAP File Compatibility

This feature is still under development.

CLDC Classfile Compatibility

The system transforms and executes CLDC classfiles in a transparent manner. This specifically means that all CLDC bytecode functionality is available in the Squawk system.

CLDC Library Compatibility

The system supports the CLDC standard libraries.

Java-Based Implementation

The entire system is written in the Java language. The (small) interpreter is written in a Java subset translatable into C, which is how the system is bootstrapped.

Minimal Memory Requirements

The Squawk system runs on a device with as little as 8K bytes of RAM.

Limitations

Platform Requirements

The Squawk system requires a minimum of 8K bytes of RAM, 32K bytes of EEPROM, and 160 K bytes of ROM. It is optimized for a 32-bit processor.

CHAPTER 3

Structure of the Squawk System

In broad outline, the Squawk system has the form of other Java implementations for small devices, specifically the CLDC and Java Card.

- There is an off-device conversion program that takes J2SE classfiles as input and produces a modified representation as output. For the CLDC, the program is the preverifier. For Java Card it is the converter. For Squawk it is the translator.
- There is a carefully engineered representation produced by the off-device converter, optimized to present the appropriate information to the on-device installation and loading software. For the CLDC, that representation is the classfile augmented with stack maps (alternatively a set of such classfiles collected in a JAR file). For Java Card it is the CAP file. For Squawk it is the suite file.
- There is on-device software for setting up intermediate files to be executed. For CLDC that software is the Java Application Manager and the class loader. For Java Card it is the installer. For Squawk it is the loader.
- There is an interpreter that processes bytecodes.
- There is a memory system associated with the interpreter that deals with the particular architectural issues of the target device.

The Translator

The Squawk translator takes a set of regular classfiles and produces a suite file. In so doing it enforces various constraints that simplify on-device execution.

The Suite File Format

The Squawk suite file has been optimized to enable on-device loading, verification, linking, and execution in as little memory as possible. It does this in two ways.

- The linking information has been made quite compact, and
- A modified bytecode set is used that is substantially more compact that the standard bytecode set.

The Loader-Verifier-Linker

Due to the careful design of the suite file format, it is possible to load, verify, link, and install suite files on the device with a relatively small footprint program. This program only needs to make one linear pass over a suite file to link, verify, and install it.

The Interpreter

Because of the simplified Squawk bytecode set and semantic optimizations it introduces, the Squawk interpreter is much smaller than a standard CLDC interpreter. The core parts of the interpreter are written in a C-compatible Java subset, so that it can easily be compiled to run natively on a target platform.

The Memory System

Small devices typically have heterogeneous memories, with much more ROM than EEPROM and a very limited amount of RAM. Traditionally, heap-based languages such as Java have been designed for the desktop, where RAM is the only memory type. Java Card in contrast has been designed with ROM and EEPROM in mind. Similarly, although Squawk is suitable for a standard RAM-based memory system, it has been designed to execute out of EEPROM and ROM, and garbage collect out of RAM. In addition, EEPROM object space reclamation is supported.

Technical Highlights

- Classfile transformations that enable simple, one pass on-device verification.
- Classfile transformations that enable simple, one pass linking and installation.
- Classfile transformations that enable simple, exact garbage collection.
- A compact and simplified bytecode set.
- In-place execution of loaded bytecodes.
- Unified object memory (programmers need not be aware of distinctions between RAM, EEPROM, and ROM).
- All system components (translator, loader-verifier-linker, interpreter, class libraries, garbage collector) written in the Java language.

CHAPTER 4

The Suite File Format

This chapter describes the suite file format. Classes defined in the suite are called *suite classes*. Classes referred to by these classes are called *proxy classes*. Proxy classes are described by the linkage information necessary to resolve references to the information defining the class.

A suite file consists of a stream of 8-bit bytes. All 16-bit, 32-bit, and 64-bit quantities are constructed by reading in two, four, eight consecutive 8-bit bytes, respectively. Multibyte data items are always stored in big-endian order, where the high bytes come first.

This chapter defines its own set of data types representing suite file data: The types `u1`, `u2`, and `u4` represent an unsigned one-, two-, or four-byte quantity, respectively.

The suite file format is presented in this chapter using pseudostructures written in a C-like structure notation. To avoid confusion with the fields of the classes and class instances of the Java virtual machine (or JVM™; the terms “Java virtual machine” and “JVM” mean a virtual machine for the Java platform), the contents of the structures describing the suite file format are referred to as *items*. Unlike the fields of a C structure, items can be optional (which is denoted by being surrounded by ‘[‘ and ‘]’) and successive items are stored in the suite file sequentially, without padding or alignment.

Tables, consisting of zero or more variable-sized items, are used in several suite file structures. Although we use C-like array syntax to refer to table items, the fact that tables are streams of varying-sized structures means that it is not possible to translate a table index directly to a byte offset into the table.

Components of the Suite File

The suite file consists of a single `SuiteFile` structure:

```
SuiteFile {
    u4 magic;
    u2 minor_version;
    u2 major_version;
    u1 access_flags;
    CONSTANT_Utf8_info name;
    Type max_type;
    u2 binds_count;
    CONSTANT_Utf8_info binds[binds_count];
    u2 types_count;
    Type_info types[types_count];
    u2 methods_count;
    MethodImpl_info methods[methods_count];
}
```

The items in the `SuiteFile` structure are as follows:

`magic`

The magic item supplies the magic number identifying the suite file format; it has the value `0xCAFEFACE`.

`minor_version`, `major_version`

The values of the `minor_version` and `major_version` items are the minor and major version numbers of this suite file. Together, a major and minor version number determine the version of the suite file format. If a suite has major version number *M* and a minor version number *m*, we denote the version of its suite file format as *M.m*.

`access_flags`

The value of the `access_flags` item is a mask of flags used to denote properties of this suite. The interpretation of each flag, when set, is as shown in the following table:

Flag Name	Value	Interpretation
ACC_FINAL	0x10	The suite cannot be bound to.

`name`

The value of the name item gives the name of the suite.

`max_type`

The value of the `max_type` item gives the highest type number assigned to a class in the suite. This includes all proxy and suite classes.

`binds_count`

The `binds_count` item specifies the number of items in the `binds` array.

`binds`

Each element in the ordered `binds` array specifies the name of a suite against which this suite is bound when it is loaded and linked.

`types_count`

The value of the `types_count` items gives the number of elements in the `types` table.

`types`

Each entry in the `types` table is a variable-length `Type_info` structure giving the definition of a class or interface. All proxy classes and proxy interfaces must precede all suite classes and suite interfaces in the `types` table.

Proxy classes and interfaces are those classes and interfaces that are linked to by the classes defined in the current suite. They correspond to classes and interfaces in suites already loaded on the device. Suite classes and interfaces on the other hand are those for which there is no existing definition on the device.

`types`

The `types` item describes the types in the suite.

`methods_count`

The value of the `methods_count` item gives the number of non-abstract and non-native methods (both static and virtual) for all non-proxy classes defined in this suite.

`methods`

Each entry in the `methods` table is a variable-length `MethodImpl_info` structure defining the implementation of a method. The static method implementations for a class precede the virtual method implementations for the class in this table. The method implementations for one class may not be interspersed with the method implementations for another class. The method implementations for a class must preceed the method implementations for any of its sub-classes.

String Structure

The `CONSTANT_Utf8_info` structure is used to represent constant string values. The definition of this structure is almost identical to that given in the Second Edition of the Java Virtual Machine Specification, except that the `tag` field is omitted:

```
CONSTANT_Utf8_info{
    u2 length;
    u1 bytes[length];
}
```

Class Type

The type of a class or interface in Squawk is given a unique integer value. Other classes or interfaces can use this integer value to denote the class or interface instead of a fully qualified name. The type space is the set of all classes and interfaces within a suite. A type is defined as

```
typedef u2 Type;
```

The `Type` value 0 has a special meaning; it denotes the absence of a type. This is used as the value for the `super_class` item of `Object`, for example.

Primitive types

The primitive types supported by the Squawk system are:

- `byte`, whose values are 8-bit signed two's-complement integers.
- `short`, whose values are 16-bit signed two's-complement integers.
- `int`, whose values are 32-bit signed two's-complement integers.
- `long`, whose values are the most significant 32-bits of the language level long as defined by the Java Virtual Machine Specification Version Two.
- `long2`, whose values are the least significants 32-bits of the language level long as defined by the Java Virtual Machine Specification Version Two.
- `char`, whose values are 16-bit unsigned integers representing Unicode characters.

The floating-point types are:

- `float` – whose values are elements of the float value set as defined by Java Virtual Machine Specification Version Two.
- `double`, whose values are the most significant 32 bits of the language level double as defined in the Java Virtual Machine Specification Version Two.
- `double2` – whose values are the least significant 32 bits of the language level double as defined in the Java Virtual Machine Specification Version Two.

Normalization of types

The verification process needs to be able to identify primitive, reference, and array data types uniformly. The verifier does this by using regular Java classes with special names to represent primitive and array data types. The class types corresponding to primitive types are as follows:

Primitive type	Class
boolean	java.lang._boolean_
byte	java.lang._byte_
short	java.lang._short_
char	java.lang._char_
int	java.lang._int_
long	java.lang._long_
long2	java.lang._long2_
float	java.lang._float_
double	java.lang._double_
double2	java.lang._double2_
void	java.lang._void_

Thus, for example, where a standard JVM would describe an integer as being type "I", Squawk uses the name "java.lang._int_" for method and field signatures.

All array types in the Squawk system are represented by special array classes. Internally these classes have names such as "[java.lang._int_", "[java.lang.Object", and "[[java.lang.String". Again, these names differ from those in a standard JVM, where, for example, java.lang.String would be "[[Ljava/lang/String;".

A key concept embodied in the suite file data structures is that each type is described using a normal Java class. There are no cases where a class and a number of array dimensions are used together to denote a type.

Note on long and double types

The 64 bit Java long and double types are represented in a suite file as pairs of 32 bit values. The following table shows the equivalence between the language specification and the suite file specification.

Java language	Suite file
long (64 bit)	long (32 bit) and long2 (32 bit)
double (64 bit)	double (32 bit) and double2 (32 bit)

In the Java language the term `long` means a 64 bit data type. In the Squawk system the term means a 32 bit data type that contains half of the Java `long` value. This `long` must be paired with a `long2`, which is another 32 bit data type that carries with it the other half of the Java `long` value.

Thus whenever a Java `long` is used as a parameter, local variable, instance variable, or static variable, the suite file must always contain a `long` followed by a `long2` (and exactly the same is true for `double` and `double2`). The bytecodes that access these data types (*load*, *store*, *getfield*, *putfield*, *getstatic* and *putstatic*) do so always by referencing the first of the two integer data types. For instance, the following method will be defined in the suite as having six local variables:

```
void foo() {
    long a = 1;
    long b = 2;
    long c = a + b
}
```

The variable `a` will be represented as two integers at offsets 0 and 1, the variable `b` will be represented as two integers at offsets 2 and 3, and the variable `c` will be represented as two integers at offsets 4 and 5. The *load* and *store* bytecodes will only refer to the first of these pairs, but both values will be used. The resulting bytecodes for this example are:

```
load 0    // load "a" from locals 0 and 1
load 2    // load "b" from locals 2 and 3
longOp    // long...
ladd      // ...add
store 4   // store into "c" at locals 4 and 5
```

However, there are two cases where a `long` or a `double` refers to a 64-bit data type (as they would following the Java standard). The first is when the type functions as the return type of a method. For example, a method returning a Java `long` has the return type `long`, even though the actual suite file data types returned will be a `long` followed by a `long2`. The other case occurs in the identification of arrays. An array of longs will have a signature in the suite file of "`[java.lang._long_`". However, as with method return types, accessing an element of this array will actually access a `long` and a `long2`.

Virtual Methods and Vtables

All the unique virtual methods defined by a set of classes in a hierarchy with a common super class are put into a single table commonly called a *vtable*. A virtual method is unique if it does not override a method in a super class. For example, there are three unique methods in the following class hierarchy:

```
class Base {
   void f() {}
   void g() {}
}

class Sub extends Base {
   void f() {}
   void h() {}
}
```

The vtable identifier for a virtual method is its index in the vtable for the class hierarchy its defining class is part of. The identifier for an overriding method is the same as the identifier for the overridden method. All the unique methods of a super class have lower valued identifiers than the identifiers of all unique methods in all its subclasses. Assuming a zero-based vtable indexing scheme (and `Object` has no methods), the identifiers for the methods in the above example would be as follows:

Method	Identifier
`Base.f()`	`0`
`Base.g()`	`1`
`Sub.f()`	`0`
`Sub.h()`	`2`

There is one vtable per class that maps method identifiers to method definitions. The identifiers are unique within a single vtable. The vtables for the classes in the example above are shown below:

Method identifier	Method definition
`0`	`Base.f()`
`1`	`Base.g()`

Method identifier	Method definition
`0`	`Sub.f()`
`1`	`Base.g()`
`2`	`Sub.h()`

Encoding With and Without Names

As an optimization, a suite file need not contain the names of fields and methods. This reduces the suite file size but means that the suite cannot be referenced by subsequently loaded suites. Methods or fields that have no symbolic information are denoted with the ACC_SYMBOL flag unset in the `access_flags` item. Typically a suite with names will be used as a library, while a suite without names will be an application.

Attributes

Attributes are used in the `Type_info` and `MethodImpl_info` structures of the suite file format. All attributes have the following general format:

```
attribute_info {
    CONSTANT_Utf8_info attribute_name;
    u4 attribute_length;
    u1 info[attribute_length];
}
```

The items of the `attribute_info` structure are as follows:

`attribute_name`

The value of the `attribute_name` item gives the name of this attribute.

`attribute_length`

The value of the `attributes_length` item indicates the length of the subsequent information in bytes. It does not include the initial six bytes that contain the `attribute_name` and `attribute_length` items.

`info`

The contents of the `info` array are the contents of the attribute.

Type_info structure

A single class or interface is defined by the following structure:

```
Type_info {
   CONSTANT_Utf8_info name;
   u2 access_flags;
   Type this_type;
   Type super_class;
   u2 interfaces_count;
   Interface_info interfaces[interfaces_count];
   u2 static_fields_count;
   Field_info static_fields[static_field_count];
   u2 instance_fields_count;
   Field_info instance_fields[instance_fields_count];
   u2 static_methods_count;
   Method_info static_methods[static_methods_count];
   u2 virtual_methods_count;
   Method_info virtual_methods[virtual_methods_count];
   u2 overriding_count;
   Overriding_info overriding[overriding_count];
   u2 class_refs_count;
   Type class_refs[class_refs_count];
   u2 objects_count;
   Object_info objects[objects_count];
   u2 attributes_count;
   attributes_info attributes[attributes_count];
}
```

The items in the `Type_info` structure are as follows:

`name`

The value of the `name` item gives the fully qualified name of the class or interface represented by the `Type_info` structure.

`access_flags`

The value of the `access_flags` item is a mask of flags used to denote access permissions to and properties of this class or interface. The interpretation of each flag, when set, is as shown in the following table:

Flag Name	Value	Interpretation
ACC_PUBLIC	0x0001	Declared `public`; may be accessed from outside its package.
ACC_FINAL	0x0010	Declared `final`; no subclasses allowed.
ACC_INTERFACE	0x0200	Is an interface, not a class.
ACC_ABSTRACT	0x0400	Declared `abstract`; may not be instantiated.
ACC_PROXY	0x0800	Is a proxy class; must exist on the target system.

An interface is distinguished by its ACC_INTERFACE flag being set. If the ACC_INTERFACE flag is not set, this `Class` defines a class, not an interface.

If the ACC_INTERFACE flag is set, the ACC_ABSTRACT must also be set and the ACC_PUBLIC flag may be set. No other flags should be set.

If the ACC_INTERFACE flag is not set, then any of the other flags may be set. However, the `Type_info` cannot have both the ACC_FINAL and ACC_ABSTRACT flags set.

`this_type`

The value of the `this_type` item is the `Type` value assigned to this class or interface. The value must be between one and `types_count` inclusive.

`super_class`

For a suite class, the value of the `super_class` item either must be zero or must be a value between one and `types_count` inclusive. If the value of the `super_class` item is nonzero, then it denotes the direct super class of this class or interface. Neither the direct super class or any of its super classes may be a final class.

If the value of the `super_class` item is zero, then this `Type_info` structure must represent the class `Object`.

For a suite interface, the value of the `super_class` item must always be the `Type` value for the class `Object`.

For a proxy class or proxy interface, the value of the `super_class` item must be zero.

`interfaces_count`

For a suite class or interface, the value of the `interfaces_count` item gives the number of unique interfaces (direct or indirect) of this class or interface type that are not in the `interfaces` item of the super class.

For a proxy class or proxy interface, the value of the `interfaces_count` item must be zero.

`interfaces`

Each entry is the `interfaces` table is a variable-length `Interface_info` structure describing a direct or indirect interface of this class or interface that is not described by an entry in the `interfaces` table of the super class.

The `Interface_info` structure is defined as:

```
Interface_info {
    Type type;
    u2 implementation_methods_count;
    u2 implementation_methods[implementation_methods_count];
}
```

The items of the `Interface_info` structure are as follows:

`type`

The value of the type item identifies the `interface` represented by this `Interface_info` structure.

`implementation_methods_count`

The value of the `implementation_method_count` item gives the number of entries in the `implementation_methods` array.

If the enclosing `Type_info` structure represents an interface or abstract class, the value of the `implementation_methods_count` item must be zero.

`implementation_methods`

The `implementation_methods` array gives a mapping from the method of the interface to the virtual method of the enclosing class that implements the interface method. That is, the value *m* at index *i* in

`implementation_methods` array indicates that the method at index *m* in the vtable of the enclosing class implements the interface method at index *i* in the `virtual_methods` table of the interface class.

`static_fields_count`

The `static_fields_count` gives the number of static fields in this class or interface.

`static_fields`

Each entry in the `static_fields` table defines a static field in the class or interface and has the following structure:

```
Field_info {
    u2 access_flags;
    Type type;
    [CONSTANT_Utf8_info name;]
}
```

The items in the `Field_info` structure are the following:

`access_flags`

The value of the `access_flags` item is a mask of flags used to denote access permissions to and properties of this field. The interpretation of each flag, when set, is as shown in the following table:

Flag Name	Value	Interpretation
ACC_PUBLIC	0x0001	Declared `public`; may be accessed from outside its package.
ACC_PRIVATE	0x0002	Declared `private`; usable only within the defining class
ACC_PROTECTED	0x0004	Declared `protected`; may be access with subclass
ACC_FINAL	0x0010	Declared `final`; no further assignment after initialization.
ACC_VOLATILE	0x0040	Declared `volatile`; cannot be cached.
ACC_TRANSIENT	0x0080	Declared `transient`; not written or read by a persistent object manager
ACC_SYMBOLIC	0x1000	Has non-empty `name` value.

A static field may have at most one of its ACC_PRIVATE, ACC_PROTECTED, and ACC_PUBLIC flags set and may not have both its ACC_FINAL and ACC_VOLATILE flags set.

All static fields of interfaces must have their ACC_PUBLIC, ACC_STATIC, and ACC_FINAL flags set.

`type`

The value of the `type` item gives the unique type number of the declared class of the field.

`name`

The value of the `name` item gives the `String` identifier of the field. The `name` item only exists if the ACC_SYMBOLIC flag is set in `access_flags`.

`instance_fields_count`

The `instance_fields_count` gives the number of instance fields in this class.

The value of the `instance_fields_count` must be zero for an interface.

`instance_fields`

Each entry of the `instance_fields` table defines an instance field in the class. Each item in the table has the format of the `Field_info` structure defined above.

`static_methods_count`

The `static_methods_count` gives the number of static methods in this class or interface.

An interface may have at the most one static method – the `<clinit>` method.

`static_methods`

Each entry in the `static_methods` table defines a static method in the class or interface and has the following structure:

```
Method_info {
    u2 access_flags;
    Type type;
    u2 parameters_count;
    Type parameters[parameters_count];
    [CONSTANT_Utf8_info name;]
}
```

The items in the Method_info structure are the following:

access_flags

The value of the access_flags item is a mask of flags used to denote access permission to and properties of this method. modifiers that apply to this method. The flags modifiers are shown in the following table:

Flag Name	Value	Interpretation
ACC_PUBLIC	0x0001	Declared public; may be accessed from outside its package.
ACC_PRIVATE	0x0002	Declared private; accessible only within the defining class.
ACC_PROTECTED	0x0004	Declared protected; may be accessed within subclasses.
ACC_FINAL	0x0010	Declared final; may not be overridden.
ACC_NATIVE	0x0100	Declared native; implemented in a language other than Java.
ACC_ABSTRACT	0x0400	Declared abstract; no implementation is provided.
ACC_SYMBOLIC	0x1000	Has non-empty name value.
ACC_INIT	0x2000	Is an instance initializer method.

type

The value of the type item gives the return type for the method.

parameters_count

The value of the parameters_count item gives the number of parameters for this method.

parameters

Each entry in the parameters array denotes the type of a parameter to this method. Note, the parameters are also included in the locals.

name

The value of the name item gives the String identifier of the method. The name item only exists if the ACC_SYMBOLIC flag is set in access_flags.

The `name` of a method special to the virtual machine must exist for the virtual machine to be able to find it. Examples of such methods include the `main` method and `<clinit>` method.

`virtual_methods_count`

The value of the `virtual_methods_count` item gives the number of virtual methods in this class or interface that do not override a method in a super class.

`virtual_methods`

Each entry in the `virtual_methods` table defines a virtual method in the class or interface that does not override a method in a super class. Each entry has the format of the `Method_info` structure defined above.

If this is a class (not an interface), its vtable is the concatenation of all the `virtual_methods` tables in the class hierarchy of this class, starting with the `virtual_methods` table for `Object`. As such, the vtable identifier for a non-interface virtual method is equal to its index in the `virtual_methods` table plus the total size of all the `virtual_methods` tables in all this class's super classes.

`overriding_count`

The value of the `overriding_count` item gives the number of virtual methods in this class that override a method in a super class where the overriding method has a different `access_flags` value than the overridden method.

`overriding`

Each entry in the `overriding` array denotes a virtual method of this class that overrides a method of a superclass with a different `access_flags` value.

Each entry has the following structure:

```
Overridden_method_info {
    u2 vindex;
    u2 access_flags;
}
```

The items in the `Overriding_info` structure are the following:

`vindex`

The value of the `vindex` item gives the vtable index of the virtual method being overridden.

`access_flags`

The value of the `access_flags` items gives the different access flags for the overriding method.

`class_refs_count`

The value of the `class_refs_count` item gives the number of class types that methods of this class may refer to using the *class* and *class_n* bytecodes.

The value of the `class_refs_count` item must be zero if the ACC_PROXY flag is set in `access_flags`.

`class_refs`

Each entry in the `class_refs` array denotes a class type that one or more methods of this class refer to using the *class* and *class_n* bytecodes.

`objects_count`

The value of the `objects_count` item gives the number of elements in the `objects` table.

The value of the `objects_count` item must be zero if the ACC_PROXY flag is set in `access_flags`.

`objects`

Each entry in the `objects` table represents a pre-constructed immutable object that one or more methods of this class refer to using the *object* and *object_n* bytecodes.

The general format of each entry is defined by the following structure:

```
Object_info {
    u1 tag;
    u1 info[];
}
```

The items in the `Object_info` structure are the following:

`tag`

The value of the tag item denotes the type of the immutable object represented by the `Object_info` entry. The contents of the `info` array vary with the value of tag. The valid tags and their values are listed in the table below.

Tag Name	Value
CONSTANT_String	1
CONSTANT_Int_array	2
CONSTANT_Short_array	3
CONSTANT_Char_array	4
CONSTANT_Byte_array	5
CONSTANT_Boolean_array	6
CONSTANT_Float_array	7
CONSTANT_Long_array	8
CONSTANT_Double_array	9

`info`

The contents of the `info` array give the value of the object. Detailed information about the possible formats for `info` are given in the next section.

`attributes_count`

The value of the `attributes_count` item gives the number of attributes in the `attributes` table.

The value of the `attributes_count` item must be zero if the ACC_PROXY flag is set in `access_flags`.

`attributes`

Each entry in the `attributes` table must be an `attribute_info` structure. There are no attributes defined in this specification that appear in the `attributes` table of a `Type_info` structure.

Constant Objects

The format of an `Object_info` structure is determined by its tag.

`CONSTANT_String_info` is defined as follows:

```
CONSTANT_String_info {
    u1 tag;
    CONSTANT_Utf8_info value;
}
```

The items of the `CONSTANT_String_info` structure are:

`tag`

The `tag` item has the value `CONSTANT_String` (1).

`value`

The value of the `value` item is a `CONSTANT_Utf8_info` structure giving the contents of the `String`.

`CONSTANT_Int_array_info` is defined as follows:

```
CONSTANT_Int_array_info {
    u1 tag;
    u2 length;
    int elements[length];
}
```

The items of the `CONSTANT_Int_array_info` structure are:

`tag`

The `tag` item has the value `CONSTANT_Int_array` (2).

`length`

The value of the `length` item gives the length of the `elements` array.

`elements`

The array of 4-byte integers.

CONSTANT_Short_array_info is defined as follows:

```
CONSTANT_Short_array_info {
    u1 tag;
    u2 length;
    short elements[length];
}
```

The items of the CONSTANT_Short_array structure are:

tag

The tag item has the value CONSTANT_Short_array (3).

length

The value of the length item gives the length of the elements array.

elements

The array of 2-byte shorts.

CONSTANT_Char_array_info is defined as follows:

```
CONSTANT_Char_array_info {
    u1 tag;
    u2 length;
    char elements[length];
}
```

The items of the CONSTANT_Char_array structure are:

tag

The tag item has the value CONSTANT_char_array (4).

length

The value of the length item gives the length of the elements array.

elements

The array of 2-byte chars.

CONSTANT_Byte_array_info is defined as follows:

```
CONSTANT_Byte_array_info {
    u1 tag;
    u2 length;
    byte elements[length];
}
```

The items of the CONSTANT_Byte_array structure are:

tag

The tag item has the value CONSTANT_Byte_array (5).

length

The value of the length item gives the length of the elements array.

elements

The array of bytes.

`CONSTANT_Boolean_array_info` is defined as follows:

```
CONSTANT_Boolan_array_info {
    u1 tag;
    u2 length;
    byte elements[length];
}
```

The items of the `CONSTANT_Boolean_array` structure are:

`tag`

The `tag` item has the value `CONSTANT_Boolean_array` (6).

`length`

The value of the `length` item gives the length of the `elements` array.

`elements`

The array of one or zero valued bytes representing the booleans.

`CONSTANT_Float_array_info` is defined as follows:

```
CONSTANT_Float_array_info {
    u1 tag;
    u2 length;
    float elements[length];
}
```

The items of the `CONSTANT_Float_array` structure are:

`tag`

The `tag` item has the value `CONSTANT_Float_array` (7).

`length`

The value of the `length` item gives the length of the `elements` array.

`elements`

The array of floats.

CONSTANT_Long_array_info is defined as follows:

```
CONSTANT_Long_array_info {
    u1 tag;
    u2 length;
    u4 elements[length*2];
}
```

The items of the CONSTANT_Long_array structure are:

tag

The tag item has the value CONSTANT_Long_array (8).

length

The value of the length item gives the length of the elements array. The value returned by the *arraylength* bytecode for this array will be length/2 given that language level longs are 64-bit values.

elements

The array of long and long2 pairs.

CONSTANT_Double_array_info is defined as follows:

```
CONSTANT_Double_array_info {
    u1 tag;
    u2 length;
    u4 elements[length];
}
```

The items of the CONSTANT_Double_array structure are:

tag

The tag item has the value CONSTANT_Double_array (9).

length

The value of the length item gives the length of the elements array. The value returned by the *arraylength* bytecode for this array will be length/2 given that language level doubles are 64-bit values.

elements

The array of double and double2 pairs.

MethodImpl_info structure

The `MethodImpl_info` structure is defined as follows:

```
MethodImpl_info {
    Type ofClass;
    u2 index;
    u2 access_flags;
    [ Code_info code_info; ]
    [ u2 attributes_count;
      attribute_info attributes[attributes_count]; ]
}
```

The items in the `MethodImpl_info` structure are as follows:

`ofClass`

The value of the `ofClass` item denotes the class that contains this method. The class must not be an interface.

`access_flags`

The value of the `access_flags` is a mask of flags used to specify properties relevant to the implementation of a method. The flags modifiers are shown in the following table:

Flag Name	Value	Interpretation
ACC_STATIC	0x0008	Declared `static`; this is a static method.
ACC_NATIVE	0x0100	Declared `native`; implemented in a language other than Java.
ACC_ABSTRACT	0x0400	Declared `abstract`; no implementation is provided.
ACC_ATTRIBUTES	0x1000	Has one or more attributes.

`index`

If the ACC_STATIC flag is set in `access_flags`, then the value of `index` denotes the entry in the `static_methods` table of this class corresponding to this method implementation. If the ACC_STATIC flag is not set, then the value of `index` denotes the vtable entry corresponding to this method implementation.

`code_info`

The `code_info` item provides the details of the code implementing the method. This item exists if and only if the ACC_NATIVE and ACC_ABSTRACT flags are not set in `access_flags`. The `Code_info` structure is defined as follows:

```
Code_info {
    u2 locals_count;
    Type locals[locals_count];
    u2 stack_size;
    u1 exception_table_length;
    {
        u4 start_pc;
        u4 end_pc;
        u4 handler_pc;
        Type catch_type;
    } exception_table[exception_table_length];
    u4 code_length;
    u1 code[code_length];
}
```

The items in the `Code_info` structure are as follows:

`locals_count`

The value of the `locals_count` item gives the number of local variables used by the method.

`locals`

Each element in the `locals` array gives the type of the local variable addressed by its index in this array.

`max_stack`

The value of the `max_stack` item gives the maximum depth of the operand stack of this method at any point during execution of the method.

`exception_table_length`

The value of the `exception_table_length` item gives the number of entries in the `exception_table` table.

`exception_table`

Each entry in the `exception_table` array describes one exception handler in the `code` array.

Each entry contains the following four items:

`start_pc, end_pc`

The values of the two items `start_pc` and `end_pc` indicate the ranges of the `code` array at which the exception handler is active. The value of `start_pc` must be a valid index into the `code` array of the opcode of an instruction. The value of `end_pc` either must be a valid index into the `code` array of the opcode of an instruction or must be equal to `code_length`, the length of the `code` array. The value of `start_pc` must be less than the value of `end_pc`.

The `start_pc` is inclusive and the `end_pc` is exclusive; that is, the exception handler must be active while the program counter is within the interval `[start_pc, end_pc)`.

`handler_pc`

The value of the `handler_pc` item indicates the start of the exception handler. The value of the item must be a valid index into the `code` array and must be the index of the opcode of an instruction.

`catch_type`

The value of the `catch_type` denotes the class of exceptions that this exception handler is designated to catch. This class must be a subclass of `Throwable`.

`code_length`

The value of the `code_length` item gives the number of bytes in the `code` array for this method implementation. The value of `code_length` must be greater than zero.

`code`

The `code` array gives the actual bytes of the virtual machine code that implement the method.

`attributes_count`

The value of the `attributes_count` item indicates the number of attributes of the method implementation. This item exists if and only if the ACC_ATTRIBUTES flags is set in `access_flags`.

`attributes`

Each entry of the `attributes` table must be an `attribute_info` structure. This item exists if and only if the ACC_ATTRIBUTES flags is set in `access_flags`.

Use of the Java evaluation stack

The Squawk verification rules add certain restrictions to the use of the Java evaluation stack. There are two reasons for this. Firstly, the restrictions mean that a single type map can be used for an entire method. This greatly simplifies the work that the garbage collector needs to do in order to identify the live object references in activation records. Secondly, the restrictions greatly simplify the verification process because there is never any data present on the stack when code paths merge. The bytecode definitions all contain a note when stack usage restrictions apply.

Verifier errors

The following are verification errors that explain the rules applied by the verifier.

- VE_BAD_CATCH_VARIABLE

  The parameter variable of a catch clause as specified in an exception handler was not a subclass of `Throwable`.

- VE_BAD_BYTECODE_BEFORE_HANDLER

  The bytecode just before the start of an exception handler was not a *goto, return, throw, tableswitch,* or *stableswitch*.

- VE_UNKNOWN_BYTECODE

  A bytecode was found that was not a part of the specification.

- VE_STACK_OVERFLOW

  Due to an incorrect bytecode stream a condition was found where an element was popped from an empty stack.

- VE_STACK_UNDERFLOW

  More elements were pushed onto the evaluation stack than the maximum specified in a method body header.

- VE_TYPE_MISMATCH

  A type verification error was detected.

- VE_TOO_WIDE

  One of the following uses of *wide* was detected. An *object, class, goto, if,* or *ifcmp* instruction was prefixed by a *wide_full*. A *load, store, getfield, putfield, getstatic,* or *putstatic* instruction was prefixed by a *wide_half* or a *wide_full*. None of these combinations are currently supported.

- VE_BAD_SWITCH

  The high value of a *tableswitch,* or *stableswitch* was lower than the low value.

- VE_BAD_ALOAD

  An *aload* instruction was applied to a non-array type.

- VE_BAD_STORE

  An *astore* instruction was applied to a non-array type, or the values being stored were incompatible with the array type.

- VE_BAD_INC

  An *inc* instruction was applied to a type other than `int`, or the local variable was greater than 255 which is a current restriction.

- VE_BAD_DEC

  A *dec* instruction was applied to a type other than `int`, or the local variable was greater than 255 which is a current restriction.

- VE_INVALID_TYPE

Either a *load, store, getfield, putfield, getstatic*, or *putstatic* instruction was illegally applied to a variable of type `long2` or `double2`, or a `long/double` was not followed by a `long2/double2` in the local variable array.

- VE_MISSING_INIT

  A constructor was found without a call to initialize its super class.

- VE_SP_NOT_ZERO

  The stack was not found to be zero at one of the points where it is defined to be so in this specification.

- VE_TWO_INIT_CALLS

  A second called to a super class initializer was found in a constructor.

- VE_BAD_WIDE_OP

  A *load* or *store* instruction was prefixed with a *wide_half* or a *wide_full*. This is a current restriction.

- VE_BAD_INVOKEINIT

  An *invokeinit* instruction was found that was not targeted to a constructor.

- VE_BAD_LOCAL

  A local variable was defined as being of type `void`, `boolean`, `byte`, `char`, or `short`.

- VE_BAD_PARM_COUNT

  The parameters of an invoke instruction were incompatible with the target method.

- VE_BRANCH_OVER_INIT

  A branch was detected that crossed the call to initialize the super class of a constructor. Constructors must not do this in order to preserve the type safety of the receiver variable.

- VE_HEADER_TOO_LARGE

  The header portion of a method was greater than 255 bytes. This is a current restriction.

- VE_INTERNAL_ERROR

  An internal error was detected in the verifier.

- VE_NOT_CLASS

  A bytecode that must be prefixed by a *class* or *class_n* instruction was not.

- VE_BAD_JUMPIP

  A branching instruction was not targeted at a valid instruction.

- VE_BAD_JUMPSP

  A branching instruction was not targeted at an instruction where the evaluation stack was empty.

CHAPTER 5

Loading, Linking and Verifying

Suite file processing

A suite is a collection of classes much like a JAR file. A suite differs from a JAR file, however, in that all the classes in the suite are internally linked together. All the classes in a suite are numbered, and when a class contains a reference to another class this is represented in terms of the class number of that class within the suite. The numbering system for all internal classes is purely internal to a particular suite and has no meaning outside it. If a class is built into two different suites the class will very probably have a different number in each suite. The class number is only a way to link classes together inside a suite.

In order for classes in a suite to link to system classes or classes provided in another suite, a special type of class called a proxy class is placed in the suite. All suites (other than the initial system suite) contain proxy classes (to reference system libraries, for example). These proxy classes always come before the real classes in the suite. The proxy classes contain only enough information to correctly identify the corresponding classes previously loaded on the device and the methods and fields needed by the code in the real classes. Typically a suite will contain proxies for classes such as java.lang.Object, java.lang.String, and so on.

After all the proxy classes have been read by the loader a mapping is constructed that translates the definition in the proxy classes to the definition of the real classes. When this is done all the symbolic information that was presented in the proxy classes is discarded. The real classes are then read by the loader followed by the method code. The loader will verify that the method code is correct and will perform the final relocation of the bytecodes that access object fields and call virtual methods based upon the proxy class mappings and the definition of the real classes.

Loading and unloading suites

When a suite is loaded a search is made for the corresponding classes for each of the proxy classes. When these classes are found the suite is loaded and a dependency mapping is created between the suites containing the classes referred to by the proxy classes and the suite being loaded. This mapping is maintained so the system can prevent a suite from being removed that another suite references. Thus if a system contains the initial system suite and, for example, a cryptography library suite, and an application is loaded that binds to the cryptography suite, this will prevent the cryptography suite from being unloaded until the application is unloaded.

Bytecode rewriting

The Squawk bytecodes specified in this document are further converted during loading. They are partially transformed as they are written into non-volatile memory at load time.

The loader is at liberty to transform any bytecodes to facilitate execution by an optimized runtime execution engine. For instance, the *aload* instruction may be applied to an array of any type. The size of the elements are not encoded in the instruction because the verifier must be able to derive this from the type of the top most element on the verification stack. At runtime, however, the element size (and sometimes the type) is needed to perform the correct operation, so the verifier transforms the *aload* into one of the following internal bytecodes: *aload_i*, *aload_b*, *aload_c*, *aload_s*, or *aload_i2*. The first of these is used when the array is of type int, float, or reference (on a 32 bit system). There are *aload* variants for byte, short, and char arrays. There is an *aload* for double integer data types of long and double. The elimination of several different *aload* instructions makes the on-device verifier easier to implement and makes the Squawk specification simpler and more logical. However, at runtime a single *aload* instruction is not convenient. By having the verifier rewrite this instruction on-device customized and efficient code can be produced. Several other bytecodes are candidates for treatment in a similar manner: *astore*, *getfield*, *putfield*, *getstatic*, *putstatic*, *load*, *store*, and *return*.

In-place execution

After all the load-time transformations are applied to the bytecodes they do not need to be changed and can so be written into read-only memory.

Unified persistent object memory

Although it is not mandatory, a core element of the Squawk design is for the on-device persistent memory (EEPROM, for example) to be a garbage collectable heap image and for all the data structures used by the VM to be regular Java objects. This has many advantages. The same mechanism that is used for general persistent object memory can also be used for storing application code. The loader and verifier code can be written in the Java language and run as an application on the VM rather then have them be written in C and be a part of the core VM code.

Load time bytecode resolution

Bytecodes that access class members, such as *getfield* and *putfield*, must be converted, before they are executed, from referring to logical offsets in a field table to being absolute offsets into an object. This conversion can be best accomplished in conjunction with verification during the loading process. All bytecodes are verified and transformed at the same time. After loading all bytecodes can be directly executed out of read-only memory. No runtime resolution is necessary.

CHAPTER 6

The Bytecode Set

The Squawk bytecode set is an optimized version of the standard JVM bytecode set. It has been designed to make the bytecodes

- more compact, so they will require less memory, and
- simpler, so verification and interpretation will be faster and require less code to implement.

Bytecode simplification

Several bytecodes have been redesigned in order to simplify the VM. For instance, the *invokespecial* bytecode has been replaced with *invokeinit* and *invokesuper* bytecodes to simplify verification. The *new* bytecode has been removed because of the considerable complications it causes for verification. The bytecode itself has been replaced with the *clinit* bytecode, and some of its functionality has been moved into the *invokeinit* bytecode, which creates the new object. The *lookupswitch* bytecode has been replaced by the *lookup* bytecode that precedes a *tableswitch* or *stableswitch* bytecode. This allows more compact lookup switch tables to be implemented and also simplifies the VM by only having one kind of multi-branch opcode.

Typed variables

All variables and parameters are typed in the new bytecode format. This has two important consequences. The number of bytecodes is reduced because the type of a load or store is no longer encoded in an instruction, and verification of the bytecodes is greatly simplified because a local variable cannot be used to hold references to different types.

Aids to verification and garbage collection

Certain rules must be followed in the formation of the bytecodes in order to simplify verification and garbage collection. Principally these are that nothing must be on the evaluation stack when a branch takes place, and when an *invoke* bytecode is executed. Only the values of the *invoke* may be present on the stack. A few other bytecodes carry a similar restriction so that the VM may be internally implemented in regular Java code rather than native machine code.

Class references

All references to other classes are internally encoded in the new bytecodes using the *class* bytecode. This bytecode refers to an entry in the class reference table of the class in which the method using the bytecode is defined. The *class* bytecode must precede the following instructions: *checkcast*, *instanceof*, *clinit*, *getstatic*, *putstatic*, *invokestatic*, *invokeinterface*, *invokeinit*, *invokesuper*, *newarray*, and *newdimension*.

Object references

In places where the *ldc* bytecode was used to refer to strings, the *object* bytecode is now used to push references to strings. The *object* bytecode refers to an entry in the object reference table of the class in which the method using the bytecode is defined.

Constants

All numeric constants are expressed directly in the bytecode stream using one of the following bytecodes: *const_null*, *const_m1*, *const_0 - const_15*, c*onst_byte*, *const_short*, *const_char*, *const_int*, *const_long*, *const_float*, and *const_double*.

Access to fields and methods

Instead of a constant pool, each class contains tables of class and object references used by its methods, and an access of a class member (method or field) is done by a logical member offset rather then a reference to a symbol in a constant pool.The invoke and field access bytecodes thus use these offsets into the tables for the target class. There are four such tables in each class. Two are for methods and two for fields. Each group is divided into virtual members and static members. The verifier converts these logical member offsets into absolute offsets in a process similar to “quickening”. The VM can execute these bytecodes quite quickly.

Small bytecodes

Certain frequently occurring bytecodes are given alternative compact encodings. In all cases the compact encoding includes a parameter from 0 to 15. There are bytecodes in this format to load and store local variables, access class and object reference tables, and push integer constants.

Hybrid bytecodes

Some instructions have hybrid combinations. These include *this_getfield* and *this_putfield*, which are the equivalent of a *getfield* or *putfield* to the receiver of a method. There are also *class_getstatic* and *class_putstatic*, which reference static fields of a method's class.

Bytecode widening

A system of bytecode widening has been implemented that is more extensive than that in the regular VM. Widening can be applied to all bytecodes that carry a succeeding bytecode. These bytecodes are: *object*, *class*, *load*, *store*, *inc*, *dec*, *invokevirtual*, *invokesuper*, *invokestatic*, *invokeinit*, *invokeinterface*, *getstatic*, *putstatic*, *class_getstatic*, *class_putstatic*, *getfield*, *putfield*, *this_getfield*, *this_putfield*, *ifeq*, *ifne*, *iflt*, *ifle*, *ifgt*, *ifge*, *if_icmpeq*, *if_icmpne*, *if_icmplt*, *if_icmple*, *if_icmpgt*, *if_icmpge*, and *goto*. By default all these bytecodes are followed by a succeeding 8 bit parameter. A widening sequence can be used that adds either 4 bits, 8 bits, 16 bits, or 24 bits to this value.

Numeric data types

In order to simplify VM construction the only numeric data type supported directly by the instruction set is the 32-bit integer (although load and store instructions exist to access smaller and larger data types). The *long* and *double* data types are always split into pairs of 32 bit integers. For the purposes of verification these are called *long*, *long2*, *double*, and *double2*.

List of Bytecodes

The remainder of this chapter consists of a list of the Squawk bytecodes. The format used is based on that found in the Java Virtual Machine Specification. The information presented includes

- the bytecode mnemonic;
- a short summary description of the bytecode;
- the format of the bytecode;
- optionally, the various forms of the bytecode;
- optionally, the effect the bytecode has on the evaluation stack;
- a longer description of the bytecode;
- optionally, the exceptions thrown by the bytecode; and
- optionally, additional comments on the bytecode.

Notational Notes

- The word size of the stack is 32 bits. In the description of the bytecodes, each item shown on the stack is a 32-bit word.
- The symbol $\perp$ is used to denote an empty stack.;

Arithmetic Operations

Operation Various integer arithmetic operations

Format

opcode

Opcodes *iadd, isub, imul, idiv, irem, iand, ior, ixor, ishl, ishr, iushr*

Stack *..., value1, value2* ⇒ *..., result*

Description

The two source operands are popped from the stack the operation is performed and the result is pushed onto the stack.

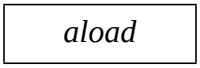
Exceptions

The *idiv* or *irem* bytecodes may result in a `DivideByZeroException` being thrown.

Array Load

Operation Load an element from an array

Format



Stack *..., arrayref, index* ⇒ *..., value*

Description

The *arrayref* must be of type reference and will refer to a type established by the verifier. The *index* must be of type int. Both *arrayref* and *index* are popped from the operand stack. The value in the array at *index* is retrieved and pushed onto the operand stack.

Exceptions

If the *arrayref* is null, the *aload* instruction throws a `NullPointerException`.

Otherwise, if *index* is not within the bounds of the array referenced by *arrayref*, the *aload* instruction throws an `ArrayIndexOutOfBoundsException`.

Array Store

Operation Load an element from an array

Format

astore

Stack *..., arrayref, index, value* ⇒ *...*

Description

The *arrayref* must be of type reference and will refer to a type established by the verifier. The *index* must be of type int. The *arrayref*, *index*, and *value* elements are popped from the operand stack. The *value* is placed into the array at *index*.

Exceptions

If the *arrayref* is null, the *astore* instruction throws a `NullPointerException`.

Otherwise, if *index* is not within the bounds of the array referenced by *arrayref*, the *astore* instruction throws an `ArrayIndexOutOfBoundsException`.

Otherwise, if *arrayref* is not null and the actual type of value is not assignment compatible with the actual type of the components of the array, *astore* throws an `ArrayStoreException`.

Note

If the type of the components of the array is a reference type, then the stack must only contain the parameters for these bytecodes when they are executed.

Array Length

Operation Get length of array

Format

arraylength

Stack *..., arrayref* ⇒ *..., length*

Description

The *arrayref* must be of type reference and must refer to an array. It is popped from the operand stack. The length of the array it references is determined. That length is pushed onto the operand stack as an int.

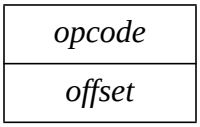
Exceptions

If the *arrayref* is null, the *arraylength* instruction throws a `NullPointerException`.

Branch Operations

Operation Compare two values and conditionally branch.

Format



Forms *ifeq ifne iflt ifle ifgt ifge*

Stack *value* ⇒ ⊥

Description

Compare *value* and zero and conditionally branch to *offset* if the comparison returns true.

Forms *if_icmpeq if_icmpne if_icmplt if_icmple if_icmpgt if_icmpge*

Stack *value1, value2* ⇒ ⊥

Description

Compare *value1* and *value2* and conditionally branch to *offset* if the comparison returns true.

Form *goto*

Stack ⊥ ⇒ ⊥

Description

The branch is unconditionally taken.

Note

The stack must only contain the parameters for these bytecodes when they are executed.

Check Cast

Operation Check to see if a reference type can be cast.

Format

checkcast

Stack *objectref, classref* ⇒ *objectref*

Description

Check to see if *objectref* can be cast to *class.*

Exceptions

If *objectref* is not null and the cast is not valid then a `ClassCastException` is thrown.

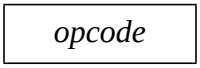
Note

The stack must only contain the parameters for this bytecode when it is executed.

Conversion Operations

Operation Convert a value from one type to another.

Format



Forms *i2b i2c i2s neg*

Stack ..., *value* ⇒ ..., *value*

Description

Convert the value on the stack from one form to another.

Class initialize

Operation Initialize a class

Format

clinit

Stack *classref* $\Rightarrow \perp$

Description

Initialize a class. The *classref* parameter must refer to a non-array class. If the class has not been initialized prior to this operation it is done by calling the `<clinit>` routine is called.

This bytecode is used where the *new* bytecode would be used in the standard Java bytecodes. The *new* bytecode has the side effect of initializing the class of the type being allocated. In this system object allocation is always done by the *invokeinit* bytecode, but in order to preserve the normal Java execution semantics it is necessary to make sure that the class is initialized at the correct time.

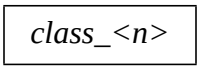
Note

The stack must only contain the parameters for this bytecode when it is executed.

Class references

Operation Push a class reference onto the stack

Format

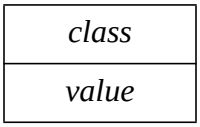


Stack ... ⇒ ..., *classRef*

Description

The low four bits of the instruction is used to index into the class reference table in the class in which the method was defined. The class reference is pushed onto the stack.

Format



Stack ... ⇒ ..., *classRef*

Description

The value of the operand in the instruction is used to index into the class reference table in the class for which the method was defined. The class reference is pushed onto the stack.

Constant values

Operation Push a constant onto the stack.

Format

const_null

Stack ... ⇒ ..., *value*

Description

A null is pushed onto the stack.

Format

const_m1

Stack ... ⇒ ..., *value*

Description

The value –1 is pushed onto the stack.

Format

const_<n>

Stack ... ⇒ ..., *value*

Description

A value between 0 and 15 is pushed onto the stack.

Format

const_byte
byte

Stack ... ⇒ ..., *value*

Description

The sign-extended byte is pushed onto the stack.

Format

const_short
byte1
byte2

Stack ... ⇒ ..., *value*

Description

The sign-extended short is pushed onto the stack.

Format

const_char
byte1
byte2

Stack ... ⇒ ..., *value*

Description

The unsigned char is pushed onto the stack.

Format

const_int
byte1
byte2
byte3
byte4

Stack ... ⇒ ..., *value*

Description

The int is pushed onto the stack.

Format

const_long
byte1
byte2
byte3
byte4
byte5
byte6
byte7
byte8

Stack ... ⇒ ..., *value1, value2*

Description

The long is pushed onto the stack.

Format

const_float
byte1
byte2
byte3
byte4

Stack ... ⇒ ..., *value*

Description

The float is pushed onto the stack.

Format

const_double
byte1
byte2
byte3
byte4
byte5
byte6
byte7
byte8

Stack ... ⇒ ..., *value1*, *value2*

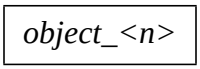
Description

The double is pushed onto the stack.

Constant object references

Operation Load a constant onto the stack

Format

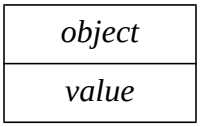


Stack ... ⇒ ..., *objectref*

Description

The low four bites of the instruction is used to index into the object reference table in the class in which the method was defined. The object reference is pushed onto the stack.

Format



Stack ... ⇒ ..., *objectref*

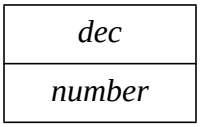
Description

The value of the operand in the instruction is used to index into the object reference table in the class for which the method was defined. The object reference is pushed onto the stack.

Decrement

Operation Subtract 1 from a local variable

Format



Stack … ⇒ …

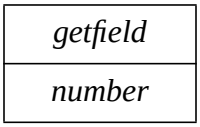
Description

The local variable is decremented. The *number* operand is the ordinal number of the variable in the method. The variable must be of type `int`.

Get Field

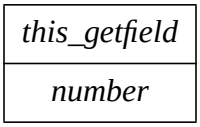
Operation Get the field of an object

Format



Stack … *objectref* ⇒ …, *value*

Format



Stack … ⇒ …, *value*

Description

The *number* operand is used to lookup the field in the class definition of *objectref* or the method receiver in the case of *this_getfield.* This parameter describes the ordinal number of the field as defined in the object after accounting for all the fields in the object's super class hierarchy. After the field is identified the relevant offset is used to access the data in the object.

A useful optimization can be made during the loading and verification of a method where the offset is calculated and the bytecode changed to be an internal type specific bytecode followed by the offset.

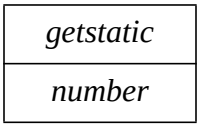
Exceptions

If the *objectref* is null, the *getfield* instruction throws a `NullPointerException`.

Get Static

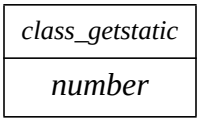
Operation Get the field of a class

Format



Stack *classref* ⇒ *value*

Format



Stack ⊥ ⇒ *value*

Description

The *number* operand is used to lookup the field in the class definition of *classref* or the current class in the *class_getstatic* format. This parameter describes the ordinal number of the field as defined in the object. After the field is identified the relevant offset is used to access the data in the object.

A useful optimization can be made during the loading and verification of a method where the offset is calculated and the bytecode changed to be an internal type specific bytecode followed by the offset.

If the class has not been initialized prior to this operation it is done by calling the `<clinit>` routine is called before the field is read.

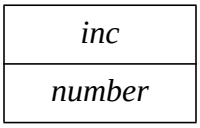
Note

The stack must only contain the parameters for this bytecode when it is executed.

Increment

Operation Add 1 to a local variable

Format



Stack … ⇒ …

Description

The local variable is incremented. The *number* operand is the ordinal number of the variable in the method. The variable must be of type `int`.

Instance Of

Operation Check to see if a reference is an instance of another

Format

instanceof

Stack *objectref, classref* ⇒ *result*

Description

Check to see if *objectref* is not null and is an instance of *class*. If is then *result* is set to 1 otherwise it is set to 0.

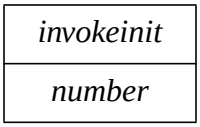
Note

The stack must only contain the parameters for this bytecode when it is executed.

Invoke Init

Operation Call a constructor method

Format



Stack *objectref_or_null, [arg1, [arg2 …]], classRef ⇒ objectref*

Description

Invoke a constructor. This bytecode must always be preceded by the *class* or *class_n* bytecode. This class reference along with the specified *number* is used to determine the method to be called. The *number* refers to the entry in the static method table of the referenced class.

The first parameter must either be a null or a reference to a receiver object. If the first parameter is null then invokeinit will allocate an object of the type described by the class reference and replace the null with this object reference prior to entering the target method. This value is always returned by the constructor method.

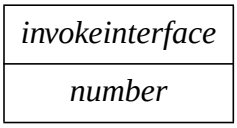
Note

The stack must only contain the parameters for this bytecode when it is executed.

Invoke Interface

Operation Call an interface.

Format



Stack *objectref, [arg1, [arg2 …]], classRef ⇒ possible_result*

Description

Invoke an interface method. This bytecode must always be preceded by the *class* or *class_n* bytecode. This class reference along with the specified *number* is used to determine the method to be called. The slot refers to the interface method table of the referenced class.

The *objectref* must be the receiver of the target method, and the class and number will specify the interface method to be called.

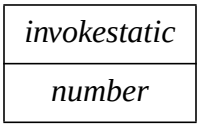
Note

The stack must only contain the parameters for this bytecode when it is executed.

Invoke Static

Operation Call a static method

Format



Stack *[arg1, [arg2 …]], classRef ⇒ possible_result*

Description

Invoke a static method. This bytecode must always be preceded by the *class* or *class_n* bytecode. This class reference along with the specified *number* is used to determine the method to be called. The *number* refers to the static method table of the referenced class.

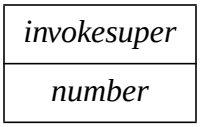
Note

The stack must only contain the parameters for this bytecode when it is executed.

Invoke Super

Operation Call a super class method

Format



Stack *objectref, [arg1, [arg2 …]], classRef* ⇒ *possible_result*

Description

Invoke a virtual method of a super class. This bytecode must always be preceded by the *class* or *class_n* bytecode. This class reference along with the specified *number* is used to determine the method to be called. The *number* refers to the virtual method table of the referenced class' super class.

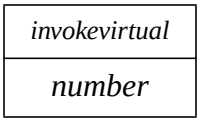
Note

The stack must only contain the parameters for this bytecode when it is executed.

Invoke Virtual

Operation Call a virtual method

Format



Stack *objectref, [arg1, [arg2 …]]* ⇒ *possible_result*

Description

The class of the *objectref* is resolved and its the method table is indexed using the *number* parameter. The corresponding method is called.

Note

The stack must only contain the parameters for this bytecode when it is executed.

Load Local

Operation Load a local variable.

Format

load_<n>

Stack ... ⇒ ..., *value*

Description

Load a local variable between 0 and 15.

Format

load
number

Stack ... ⇒ ..., *value*

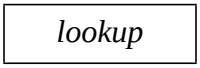
Description

Load a local variable. The *number* operand is the ordinal number of the variable in the method.

Lookup

Operation Lookup a value in an array

Format



Stack *..., key, arrayref* ⇒ *... index*

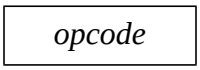
Description

Lookup *key* in a sorted array of numbers. The array must be an array of ints, shorts, chars or bytes. The result is the index, or –1 if not present.

Monitor Operations

Operation Lock / unlock objects

Format



Forms *monitorenter monitorexit*

Stack *objectref* $\Rightarrow \perp$

Description

Claim or release the lock on an object.

Forms class_*monitorenter class_monitorexit*

Stack $\perp \Rightarrow \perp$

Description

Claim or release the lock on an object.

Note

The stack must only contain the parameters for this bytecode when it is executed.

New Array

Operation Create an array object

Format

newarray

Stack *size, classref* ⇒ *arrayref*

Description

Create a new array object. The *classref* parameter must refer to an array class. The *size* parameter is the length of the array to be created.

Note

The stack must only contain the parameters for this bytecode when it is executed.

New Dimension

Operation Add a dimension to an array

Format

newdimension

Stack *arrayref, size* ⇒ *arrayref*

Description

This bytecode is used to populate multi-dimensional arrays. The references array is searched for the first dimensional level with null references. This level is then populated with new array references of the appropriate type.

Note

The stack must only contain the parameters for this bytecode when it is executed.

Pop

Operation Pop a stack word

Format

pop

Stack ..., *value* ⇒ ...

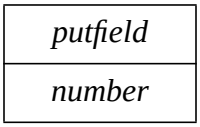
Description

Pop a stack word.

Put Field

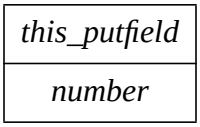
Operation Set the field of an object

Format



Stack … *objectref, value* ⇒ …

Format



Stack … *value* ⇒ …

Description

The *number* parameter is used to lookup the field in the class definition of *objectref* or the method receiver in the case of *this_putfield*. This parameter describes the ordinal number of the field as defined in the object after accounting for all the fields in the object's super class hierarchy. When the field is identified then the relevant offset is used to access the data in the object and the *value* parameter is written there.

A useful optimization can be made during the loading and verification of a method where the offset is calculated and the bytecode changed to be an internal type specific bytecode followed by the offset.

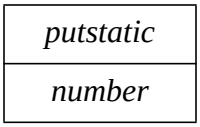
Exceptions

If the *objectref* is null, the *putfield* instruction throws a `NullPointerException`.

Put Static

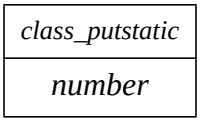
Operation Set the field of a class

Format



Stack *value, classref* $\Rightarrow \perp$

Format



Stack *value* $\Rightarrow \perp$

Description

The *number* parameter is used to lookup the field in the class definition of *classref* or the current class in the *class_putstatic* format. This parameter describes the ordinal number of the field as defined in the object. When the field is identified then the relevant offset is used to access the data in the object and the *value* parameter is written there.

A useful optimization can be made during the loading and verification of a method where the offset is calculated and the bytecode changed to be an internal type specific bytecode followed by the offset.

If the class has not been initialized prior to this operation it is done by calling the `<clinit>` routine is called before the field is read.

Note

The stack must only contain the parameters for this bytecode when it is executed.

Return

Operation Return from a method

Format

return

Stack ..., $\Rightarrow \perp$

Stack ..., *value* $\Rightarrow \perp$

Stack ..., *value, value* $\Rightarrow \perp$

Description

Return from a method. The method signature determines the type of return. There are basically three types of return. One that returns void, one that returns one word from the stack, and one that returns two words from the stack.

Store Local

Operation Store a local variable.

Format

store_<n>

Description

Store a local variable between 0 and 15.

Stack ..., *value* ⇒ ...

Format

store
number

Stack ..., *value* ⇒ ...

Description

Store a local variable. The *number* operand is the ordinal number of the variable in the method.

Table Switch

Operation Table Switch

Format

tableswitch
0-3 byte pad
default byte 1
default byte 2
default byte 3
default byte 4
low byte 1
low byte 2
low byte 3
low byte 4
high byte 1
high byte 2
high byte 3
high byte 4
jump offsets…

Stack $key \Rightarrow \perp$

Description

tableswitch is the same as the regular Java bytecode.

Note

The stack must only contain the parameters for this bytecode when it is executed.

Format

stableswitch
0-1 byte pad
default byte 1
default byte 1
low byte 1
low byte 2
low byte 3
low byte 4
high byte 1
high byte 2
high byte 3
high byte 4
jump offsets…

Stack *key* $\Rightarrow \perp$

Description

s*tableswitch* is the same as *tableswitch* except all fields are 16 bits only.

Note

The stack must only contain the parameters for this bytecode when it is executed.

Throw

Operation Throw an exception.

Format

throw

Stack ..., *objref* $\Rightarrow \perp$

Description

Throw the exception *value*.

Wide

Operation Modify operand format of the next bytecode as defined below.

Format

wide_<n>
opcode
byte

Format

wide_half
opcode
byte1
byte2

Format

wide_full
opcode
byte1
byte2
byte3
byte4

Stack *same as the modified instruction*

Description

The wide bytecode is can be used as a prefix to all bytecodes that have a single extra byte. These are: *class, object, load, store, if, goto, invoke, invokevirtual, getstatic, putstatic, getfield, putfield*.

The *wide_n* format expands into 16 different opcodes: *wide_0* … *wide_15*. Each of these adds four bits to the instruction such that the operand to *opcode* is calculated to be *(n << 8) | (byte & 0xFF)*.

For instance the sequence:

```
wide_1
load
7
```

Will cause local variable number 263 to be loaded. If the *opcode* is *if* or *goto*, the operand is sign extended. For example, a conditional branch to offset –700 is expressed as:

```
wide_13
ifeq
68
```

The *wide_half* and *wide_full* forms simply extend the size of a one-byte parameter to a 16-bit and 32-bit size operand respectively. For instance the sequence:

```
wide_half
load
1
7
```

Will also cause local variable number 263 to be loaded.

Long integer operations

Note – Note on longs. The Java language type long is represented in the VM spec as a pair of 32 bit data types called long and long2.

Operation Perform long integer operation.

Format

longOp
opcode

Forms *ladd, lsub, lmul, ldiv, lrem, land, lor, lxor*

Stack *..., long, long2, long, long2* ⇒ *..., long, long2*

Description

Pop two longs from the stack perform operation and then push a single long result.

Exceptions

An *idiv* or *irem* may result in a `DivideByZeroException` being thrown.

Forms *lneg*

Stack *..., long, long2,* ⇒ *..., long, long2*

Description

Negate the long.

Forms *lshl, lshr, lushr*

Stack *..., long, long2, int* ⇒ *..., long, long2*

Description

Shift operation

Forms *lcmp*

Stack *..., long, long2, long, long2, ⇒ ..., int*

Description

Compare the longs

Forms *l2i*

Stack *..., long, long2, ⇒ ..., int*

Description

Long to int.

Forms *i2l*

Stack *..., int ⇒ ..., long, long2,*

Description

Int to long.

Floating point operations

Note – Note on doubles. The Java language type double is represented in the VM spec as a pair of 32 bit data types called double and double2.

Operation Perform long integer operation.

Format

floatOp
opcode

Forms *fadd, fsub, fmul, fdiv, frem*

Stack *..., float, float ⇒ ..., float*

Description

Pop two floats from the stack perform operation and then push a single float result.

Forms *fneg*

Stack *..., float ⇒ ..., float*

Description

Negate the float

Forms *fcmpg, fcmpl*

Stack *..., float, float ⇒ ..., int*

Description

Compare two floats and push an integer result.

Forms *dadd, dsub, dmul, ddiv, drem*

Stack *..., double, double2, double, double2 ⇒ ..., double, double2*

Description

Pop two doubles from the stack perform operation and then push a single double result.

Forms *dneg*

Stack *..., double, double2* ⇒ *..., double, double2*

Description

Negate the double

Forms *dcmpg, dcmpl*

Stack *..., double, double2, double, double2* ⇒ *..., int*

Description

Compare two doubles and push an integer result.

Forms *i2f, l2f, f2i, f2l, i2d, l2d, f2d, d2i, d2l, d2f*

Stack *..., something* ⇒ *..., somethingelse*

Description

Convert data type.

CHAPTER 7

The Translator

The translator converts a set of standard classfiles into a suite. It performs the tasks of format conversion and bytecode translation. In addition it applies two transformations that are important to the Squawk design.

Transformations for Verification

The verification process detailed in the CLDC specification is an evolution of the standard J2SE™ verification process. It improves the static and dynamic resource requirements for verification without compromising type safety guarantees.

Continuing that evolution, the Squawk system makes verification yet smaller and simpler. As with the CLDC, verification is performed via a linear scan over the bytecodes, but without the need for stack maps.This is achieved by placing extra constraints on how bytecodes use local variables and the operand stack. In particular, the following constraints are imposed:

- At the end of all basic blocks in the control flow graph of a method, there must be nothing on the operand stack.
- Each local variable does not change its type for the duration of a method.

The Squawk system implements these constraints by transforming existing Java bytecodes to meet them. This involves creating extra local variables for existing local variables that are reused and creating extra local variables to spill stack values into at the end of a basic block. Also, additional load and store instructions are inserted to perform the spilling and filling.

The impact of the transformations is quite small with respect to the increase in code and frame sizes. Code size is typically reduced 15% compared to the original bytecodes, even though there are typically more instructions in the transformed code (for spilling and filling). This is due to the greater density of the Squawk bytecodes. Local variables usage (frame size) is only slightly greater in the transformed bytecodes. This is due to the fact that most Java source compilers do not do aggressive register allocation, typically equating a variable's liveness in the bytecodes with its scope in the source code. Including a better register allocation

algorithm in the translator can therefore reduce frame sizes even though more local variables are typically being generated (for the purpose of stack spilling and filling).

Note that code generated by javac only violates these constraints for two Java source level constructs:

- The ternary operator; for example:

```
String s = (if a % 2 ? “even” : “odd”);
```

- Using the result of a comparison as anything but an operand to a boolean expression; for example:

```
return (this == obj);
```

Transformations for Exact Garbage Collection

The hard part of exact garbage collection in Java is identifying the pointers in activation frames and on the evaluation stack. The translator applies two transformations that make this significantly easier:

- Local variables are partitioned into pointer variables and non-pointer variables. This means that a single pointer map per method can describe which slots in an activation record are pointers. Note that this transformation comes for free as a result of the transformation applied to enable on-card verification.
- The evaluation stack is made to be empty (“trimmed”) before any instruction that may potentially result in an allocation (and therefore a GC). That is, all values on the stack except for the parameters and operands of the current instruction are spilled before the potentially allocating instruction is executed. The table below shows the list of bytecodes that require stack trimming as well as the reason they may cause an allocation.

Instruction	Reason
checkcast	object allocation
instanceof	object allocation
getstatic	invocation of <clinit> / initialization
putstatic	invocation of <clinit> / initialization
invoke*	invocation of <clinit> / initialization / object allocation
monitorenter / exit	invocation of <clinit> / initialization, context switch
newdimension	object allocation
newarray	object allocation
newobject	object allocation

Instruction	Reason
clinit	invocation of <clinit> / initialization

Note that instructions whose only potential allocation related side effect is the exception object for a VM raised exception are not included above. This is because the current state of the evaluation stack is ignored during exception handling and so can be treated as empty should a GC occur when allocating an exception object.

Note also that these transformations only simplify exact GC in VM implementations where thread scheduling is under the explicit control of the VM and the VM ensures that context switching can only occur at one of the instructions listed in the above table.

APPENDIX **A**

The Minimal Virtual Machine

The Squawk system supports full CLDC functionality. It will execute any CLDC-compliant classfile. It executes all the bytecodes and is compliant with all the size parameters of the classfile.

However, a Squawk subset has been defined for memory constrained platforms (specifically next generation smart cards). In this subset certain fields have been reduced in size. The subset still support reasonably sized Java programs for such platforms. The reductions impose the following limitations:

- Each method can use no more than 256 local variables.
- Arrays can be no larger than 65536 elements long and may have no more than 31 dimensions.
- Methods can be no larger than 65536 bytes long.
- There may be no more than 256 classes in a suite.
- There may be no more than 256 suites loaded on a card.
- There may be no more than 256 virtual methods in a single class's hierarchy.
- There may be no more than 256 static and <init> methods in a single class.
- There may be no more than 256 non-static fields in a single class's hierarchy.
- There may be no more than 256 static fields in a single class.
- There may be no more than 256 methods in an interface's hierarchy.

In addition, floating point data types (float and double) are not supported.

Notes

There are aspects of the Squawk system that depart from standard Java semantics in the "letter of the law". In practice these deviations are inconsequential.

Access to uninitialized local variables

Normal Java verification checks that local variables are initialized before they are used. This is easy for the KVM to check because the information is present in the stackmaps. There is no way to do this in Squawk, so all local variables are initialized to zero at the start of a method. This means that access to pointer variables that have not been initialized results in a null pointer exception at runtime rather than a verification exception at load time. Accesses to uninitialized non-pointer variables simply get the default value of the appropriate type (null or 0). This means that local variables have the same default initialization semantics as object fields. Note that these semantic differences are only observable at the bytecode level, as javac checks use of uninitialized variables.

Class instance construction

The Java constructor idiom "`new Foo()`" results in a *new* bytecode followed by a call to a constructor method when compiled with javac. This causes several major problems in the verifier, as the type of the reference returned by the *new* bytecode has to be changed by the verifier after the constructor is called, and any location to which it was copied also has to be changed. Given that it cannot be guaranteed that the code will be contained in a basic block (for example, "`new Foo(a?b:c)`" will generate two forward branches), the correct verification of these initialized reference types is more complex than is desirable for Squawk.

The simplest solution to this problem is for the constructor to allocate the object if the first parameter is zero and for it to return the initialized object as its result. In order for class initialization to happen in the right order the *new* is replaced with *clinit*. This is done because code such as "`new Foo(new Bar))`" results in the Foo class being initialized before the Bar class.

For example, the expression "`Foo x = new Foo(1,2,3)`" would be changed from:

```
new Foo
dup
iconst_1
iconst_2
iconst_3
invokespecial <init>
astore x
```

to:

```
clinit Foo
aconst_null
iconst_1
iconst_2
iconst_3
invokespecial <init>
astore x
```

It is hard for these alternative semantics to be detected at the Java language level. The edge case where this difference would be noticed is where the memory allocation caused by *new* would fail. In the transformed code, the resulting OutOfMemoryError exception could be masked out if the evaluation of the parameters to the constructor caused an exception.

Constructor transformation issues

The above example is transformed by the translator by searching for certain bytecode sequences that are known to be output by javac. There are possible sequences that are legal that the transformer will not recognize, and in such circumstances the translator will respond by aborting the transformation operation.

Constant field definitions

The Java idiom for defining constants (for example, "`final static int foo = 1;`") causes the 1.3 and 1.4 javac compilers to inline references to the constant. However, the 1.4 javac compiler also generates code to initialize the field as well (probably in order to support the full reflection model). This will, however, permit another class to bind to the value at link time. The Squawk translator will not do this and so all constants must be resolved at compile time. It appears that this is precisely the way that javac works, so it should not affect the semantics of programs built in the normal way.

APPENDIX **B**

Tables of Bytecodes

Alphabetic Table Of Bytecodes

Insert table here

Numeric Table Of Bytecodes

Insert table here