Appendix A

JASM Syntax

This chapter describes JASM syntax, and how to encode class files using this syntax. Jasm is a java assembler that accepts text in the JASM format and produces a `.class` file for use with a Java Virtual Machine. Jasm's primary use is as a tool for producing specialized tests for testing a JVM implementation

This chapter describes JASM syntax in the following sections:

General Syntax

JASM syntax can come in one of two variations: short-form or verbose-form. Short form uses Java-style names to refer to items in a constant-pool. Verbose form uses explicit constant-pool indexes to refer to items in the constant pool. The normal output from JDIS produces jasm files in the short-form. Using the -g option for JDIS (ie. jdis -g file.class) produces JASM source in the verbose-form.

The source text file can be free form (newlines are considered blanks) and may contain Java-style commenting. The first line of a JASM file represents the name of the resulting file in the destination directory. This name does not affect the content of the resulting file. This line has two forms:

```
file FILENAME
```

or

```
class CLASSNAME
```

In the latter case, extension `.class` will be added to form FILENAME. Jasm's `-d option` allows you to define the destination directory. A list of structured data items follows the class name. The length (in bytes) of each item is determined by its representation.

Description formats

TERM1|TERM2	TERM1 or TERM2 (not both)
[TERM]	TERM is optional
TERM...	TERM repeated 1 or more times
[TERM...]	TERM repeated 0 or more times

"sequence of"	all the following terms are mandatory, in the order given.
"set of"	any of following terms, or none of them, may appear in any order. However, repetitions are not allowed.
"list of"	any of following terms, or none of them, may appear in any sequence. If more than one term appear, they are separated by commas (',')

Lexical Structure

The source text file can be free form (newlines, tabs, and blank spaces are equivalent). Additionally, the source may contain standard Java and C++ comments.

`STRING`, `NUMBER`, and `IDENT` are treated the same as in the Java Language Specification. One difference is that LETTERs include also `/', `<', `>', `(', and `)' .

STRING:

" [STRING_CHARACTER...] "

NUMBER:
DIGIT...

IDENT:

LETTER [LETTER_OR_DIGIT ...]

ACCESS (depends on the context): set of

```
abstract final interface native private protected public static super synchronized
transient volatile deprecated synthetic bridge varargs
```

Not all access bits make sense for all declarations: for example, the "super" and "interface" access flags are applied to classes only.

If an access bit is used improperly, the assembler prints a warning, but places the bit in the access set.

Note that `deprecated` and `synthetic` keywords are not translated to access flags in the Java sense. For these jasm generates a corresponding `Deprecated` or `Synthetic` attributes instead of access bits. The `synthetic` access flag is used to mark compiler generated members not seen in the source (for example, a field reference to an anonymous outer class).

Local names represent labels, trap-labels and local variables. Their scope is constrained by method parenthesis.

LOCAL_NAME:

```
IDENT
```

CONSTANT_INDEX:

```
#NUMBER
```

Each CONSTANT_INDEX represents a reference into the constant pool at the specified location.

General Class Structure

INTERFACES: list of

```
CONSTANT_CELL(class)
```

TOP_LEVEL_COMPONENT: one of

```
CONSTANT_DECLARATION FIELD_DECLARATION METHOD_DECLARATION INNER_CLASS_DECARATIONS
```

CLASS: sequence of

```
ANNOTATIONS CLASS_ACCESS CONSTANT_CELL(class) [extends CONSTANT_CELL(class)]
[implements INTERFACES] [version INTEGER:INTEGER] { [TOP_LEVEL_COMPONENT...] }
```

CLASS_ACCESS: list of

```
[public][final][super][interface][abstract][synthetic][annotation][enum]
```

The `extends CONSTANT_CELL(class)` clause places the "super" element of the class file. The `implements INTERFACES` clause places the table of interfaces. Since the assembler does not distinguish interfaces and ordinary classes (the only difference is one access bit), the table of interfaces of an interface class must be declared with `implements` keyword, and not `extends`, as in Java language.

Note: The last two rules allow `TOP_LEVEL_COMPONENT` to appear in any order and number. For example, you can split constant pool table into several parts, mixing constants and method declarations.

General Source File Structure

PACKAGE_DECLARATION:

```
package IDENT;
```

Package declaration can appear only once in source file.

SOURCE_FILE: sequence of

```
PACKAGE_DECLARATION CLASS...
```

The Constant Pool and Constant Elements

A `CONSTANT_CELL` refers to an element in the constant pool. It may refer to the element either by its index or its value:

CONSTANT_CELL:

```
CONSTANT_INDEX
```

```
TAGGED_CONSTANT_VALUE
```

Tags differentiate constant entries in a constant pool:

TAG: one of

```
      int float long double Asciz String class Field Method NameAndType
 InterfaceMethod MethodType MethodHandle InvokeDynamic
```

Generic rule for TAGGED_CONSTANT_VALUE is:

TAGGED_CONSTANT_VALUE:

```
[TAG] CONSTANT_VALUE
```

A TAG may be omitted when the context only allows one kind of a tag. For example, the argument of an `anewarray` instruction should be a `CONSTANT_CELL` which represents a class, so instead of

```
anewarray class java/lang/Object
```

one may write:

```
anewarray java/lang/Object
```

It is possible to write another tag, e.g.:

```
anewarray String java/lang/Object
```

However, the resulting program will be incorrect.

Another example of an implicit tag (eg. a context which implies tag) is the header of a class declaration. You may write:

```
aClass {
}
```

which is equivalent to:

```
class aClass {
}
```

Below, the tag implied by context will be included in the rules, e.g.:

```
CONSTANT_VALUE(int).
```

The exact notation of `CONSTANT_VALUE` depends on the (explicit or implicit) `TAG`.

TAGGED_CONSTANT_VALUE:

int	`INTEGER`					
long	`[INTEGER|LONG]`					
float	[FLOAT|INTEGER]					
float	bits INTEGER					
double	[FLOAT|DOUBLE|INTEGER|LONG]					
double	[bits INTEGER | bits LONG]					
Asciz	`EXTERNAL_NAME`					
class	CONSTANT_NAME					
String	`CONSTANT_NAME`					
NameAndType	`NAME_AND_TYPE`					
Field	`CONSTANT_FIELD`					
Method	`CONSTANT_FIELD`					
MethodHandle	`INVOKESUBTAG`	:	`CONSTANT_FIELD [FIELDREF|METHODREF|INTERFACEMETHODREF]`			
`MethodType`	`CONSTANT_NAME`					
InvokeDynamic	`INVOKESUBTAG`	:	`CONSTANT_FIELD`	:	`NAME_AND_TYPE`	`[INVOKEDYNAMIC_STATIC_ARGS]`

Note

When the JASM parser encounters an InvokeDynamic constant, it creates an entry in the *BootstrapMethods* attribute (the *BootstrapMethods* attribute is produced if it has not already been created). The entry contains a reference to the *MethodHandle* item in the constant pool, and, optionally, a sequence of references to additional static arguments (*ldc*-type constants) to the *bootstrap method.*

INVOKESUBTAGs for MethodHandle and (const) InvokeDynamic are defined as follows:

INVOKESUBTAG:

REF_GETFIELD	[1]
REF_GETSTATIC	[2]
REF_PUTFIELD	[3]
REF_PUTSTATIC	[4]
REF_INVOKEVIRTUAL	[5]
REF_INVOKESTATIC	[6]
REF_INVOKESPECIAL	[7]
REF_NEWINVOKESPECIAL	[8]
REF_INVOKEINTERFACE	[9]

Static arguments for an *InvokeDynamic* constant are defined as follows:

INVOKEDYNAMIC_STATIC_ARGUMENTS:

```
INVOKEDYNAMIC_STATIC_ARG ',' ...
```

INVOKEDYNAMIC_STATIC_ARG: (one of)

```
INVOKEDYNAMIC_STATIC_ARG_CONSTANT_VALUE
```

INVOKEDYNAMIC_STATIC_ARG_CONSTANT_VALUE:

int	INTEGER
long	[INTEGER|LONG]
float	[FLOAT|INTEGER]
double	[FLOAT|DOUBLE|INTEGER|LONG]
class	CONSTANT_NAME
String	CONSTANT_NAME
MethodHandle	INVOKESUBTAG: CONSTANT_FIELD
MethodType	CONSTANT_NAME

`INTEGER`, `LONG`, `FLOAT`, and `DOUBLE` correspond to `IntegerLiteral` and `FloatingPointLiteral` as described in The Java Language Specification. If a double-word constant (`LONG` or `DOUBLE`) is represented with a single-word value (`INTEGER` or `FLOAT`, respectively), single-word value is simply promoted to double-word, as described in The Java Language Specification. If floating-point constant (`FLOAT` or `DOUBLE`) is represented with an integral value (`INTEGER` or `LONG`, respectively), the result depends on whether the integral number is preceded with the keyword "bits". If "bits" is not used, the result is a floating-point number closest in value to the decimal number. If the keyword "bits" is used, the floating-point constant takes bits of the integral value without conversion.

Thus,

```
float 2;
```

means the same as

```
float 2.0f;
```

and the same as

```
float bits 0x40000000;
```

while

```
float bits 2;
```

actually means the same as

```
float bits 0x00000002;
```

and the same as

```
float 2.8026e-45f
```

CONSTANT_NAME:

```
CONSTANT_INDEX
```

```
EXTERNAL_NAME
```

EXTERNAL_NAME:

```
IDENT STRING
```

External names are names of class, method, field, or type, which stay in resulting .class file, and may be represented both by `IDENT` or by `STRING` (which is useful when name contains non-letter characters).

NAME_AND_TYPE:

```
CONSTANT_INDEX
```

```
CONSTANT_NAME:CONSTANT_NAME
```

In this second example, the first `CONSTANT_NAME` denotes the name of a field and second denotes its type.

CONSTANT_FIELD:

```
CONSTANT_INDEX
```

```
[CONSTANT_NAME.]NAME_AND_TYPE
```

In this third example, `CONSTANT_NAME` denotes to the class of a field. If `CONSTANT_NAME` is omitted, the current class is assumed.

Constant Declarations

Constant declarations are demonstrated in the examples below:

```
const #1=int 1234
    , #2=String "a string"
    , #3=Method get:I
;
```

CONSTANT_DECLARATION:

```
const CONSTANT_DECLARATORS ;
```

CONSTANT_DECLARATORS: list of

```
CONSTANT_DECLARATOR
```

CONSTANT_DECLARATOR:

```
CONSTANT_INDEX = TAGGED_CONSTANT_VALUE
```

Field Variables

FIELD_DECLARATION:

```
ANNOTATIONS FIELD_ACCESS Field FIELD_DECLARATORS ;
```

FIELD_DECLARATORS: list of

```
FIELD_DECLARATOR
```

FIELD_DECLARATOR:

```
EXTERNAL_NAME:CONSTANT_NAME [ = TAGGED_CONSTANT_VALUE ]
```

FIELD_ACCESS: list of

```
[public|private|protected][final][static][volatile][transient][synthetic][enum]
```

Example:

```
    public static Field
        field1:I = int 1234,
        field2:S
    ;
```

Access bits (public and static) are applied both to field1 and field2. The `EXTERNAL_NAME` denotes the name of the field, `CONSTANT_NAME` denotes its type, `TAGGED_CONSTANT_VALUE`denotes initial value.

Method Declarations

METHOD_DECLARATION: sequence of

```
ANNOTATIONS METHOD_ACCESS Method

EXTERNAL_NAME:CONSTANT_NAME

[THROWS]

STACK_SIZE

[LOCAL_VAR_SIZE]

{ INSTRUCTION_STATEMENT... }
```

The `EXTERNAL_NAME` denotes the name of the method, `CONSTANT_NAME` denotes its type.

METHOD_ACCESS: list of

```
[public|private|protected][static][final][synthetic][bridge][varargs]
[native][abstract][strict][synthetic]
```

THROWS:

```
throws EXCEPTIONS
```

EXCEPTIONS: list of

```
CONSTANT_CELL(class)
```

The meaning of the `THROWS` clause is the same as in Java Language Specification - it forms Exceptions attribute of a method. Jasm itself does not use this attribute in any way.

STACK_SIZE:

```
stack NUMBER
```

The `NUMBER` denotes maximum operand stack size of the method.

LOCAL_VAR_SIZE:

```
locals NUMBER
```

The `NUMBER` denotes number of local variables of the method. If omitted, it is calculated by assembler according to the signature of the method and local variable declarations.

Instructions

VM Instructions

INSTRUCTION_STATEMENT:

```
[NUMBER] [LABEL:] INSTRUCTION|PSEUDO_INSTRUCTION ;
```

Jasm allows for a `NUMBER` (which is ignored) at the beginning of each line. This is allowed in order to remain consistent with the jdis disassembler. Jdis puts line numbers in disassembled code that may be reassembled using Jasm without any additional modifications.

INSTRUCTION:

```
OPCODE [ARGUMENTS]
```

ARGUMENTS: list of

```
ARGUMENT
```

ARGUMENT:

```
NUMBER LABEL LOCAL_VARIABLE TRAP_IDENT CONSTANT_CELL SWITCHTABLE TYPE
```

LABEL:

```
NUMBER IDENT
```

LOCAL_VARIABLE:

```
NUMBER IDENT
```

TRAP_IDENT:

```
IDENT
```

TYPE:

```
NUMBER boolean byte char int float long double class
```

SWITCHTABLE:

```
{ [NUMBER:LABEL...] [default:LABEL] }
```

SWITCHTABLE example: Java_text

```
 switch (x) {
        case 11:
                    x=1;
                    break;
        case 12:
                    x=2;
                    break;
        default:
                    x=3;
 }
```

will be coded in assembler as follows:

```
tableswitch  {
         11: L24;
         12: L29;
         default: L34
     }
L24: iconst_1;
     istore_1;
     goto  L36;
L29: iconst_2 ;
     istore_1;
     goto  L36;
L34: iconst_3;
     istore_1;
L36:    ....
```

OPCODE is any mnemocode from the instruction set. If mnemocode needs an ARGUMENT, it cannot be omitted. Moreover, the kind (and number) of the argument(s) must match the kind (and number) required by the mnemocode:

aload, astore, fload, fstore, iload, istore, lload, lstore, dload, dstore, ver, endvar:	LOCAL_VARIABLE
iinc:	LOCAL_VARIABLE,NUMBER
sipush, bipush, bytecode:	NUMBER
tableswitch, lookupswitch:	SWITCHTABLE
newarray:	TYPE

jsr, goto, ifeq, ifge, ifgt, ifle, iflt, ifne, if_icmpeq, if_icmpne, if_icmpge, if_icmpgt, if_icmple, if_icmplt, if_acmpeq, if_acmpne, ifnull, ifnonnull, try, endtry:	LABEL
jsr_w, goto_w:	LABEL
ldc_w, ldc2_w, ldc:	CONSTANT_CELL
new, anewarray, instanceof, checkcast,	CONSTANT_CELL(class)
multianewarray	NUMBER,CONSTANT_CELL(class)

putstatic, getstatic, putfield, getfield:	CONSTANT_CELL(Field)
invokevirtual, invokenonvirtual, invokestatic:	CONSTANT_CELL(Method)
invokeinterface:	NUMBER,CONSTANT_CELL(Method)

invokedynamic:	CONSTANT_CELL(InvokeDynamic)
aaload, aastore, aconst_null, aload_0, aload_1, aload_2, aload_3, aload_w , areturn, arraylength, astore_0, astore_1, astore_2, astore_3, astore_w, athrow, baload, bastore, caload, castore, d2f, d2i, d2l, dadd, daload, dastore, dcmpg, dcmpl, dconst_0, dconst_1, ddiv, dead, dload_0, dload_1, dload_2, dload_3, dload_w , dmul, dneg, drem, dreturn, dstore_0, dstore_1, dstore_2, dstore_3, dstore_w, dsub, dup, dup2, dup2_x1, dup2_x2, dup_x1, dup_x2, f2d, f2i, f2l, fadd, faload, fastore, fcmpg, fcmpl, fconst_0, fconst_1, fconst_2, fdiv, fload_0, fload_1, fload_2, fload_3, fload_w, fmul, fneg, frem, freturn , fstore_0, fstore_1, fstore_2, fstore_3, fstore_w, fsub , i2b, i2c, i2d, i2f, i2l, i2s, iadd, iaload, iand, iastore, iconst_0, iconst_1, iconst_2, iconst_3, iconst_4, iconst_5, iconst_m1, idiv, iinc_w, iload_0, iload_1, iload_2, iload_3, iload_w, imul, ineg, int2byte, int2char, int2short, ior, irem, ireturn, ishl, ishr, istore_0, istore_1, istore_2, istore_3, istore_w, isub, iushr, ixor, l2d, l2f, l2i, label, ladd, laload, land, lastore, lcmp, lconst_0, lconst_1, ldiv, lload_0, lload_1, lload_2, lload_3, lload_w, lmul, lneg, lor, lrem, lreturn, lshl, lshr, lstore_0, lstore_1, lstore_2, lstore_3, lstore_w, lsub, lushr, lxor, monitorenter, monitorexit, nonpriv, nop, pop, pop2, priv, ret, return, ret_w, saload, sastore, swap, wide	<NoArguments>

InvokeDynamic Instructions

InvokeDynamic instructions are instructions that allow dynamic binding of methods to a call site. These instructions in JASM form are rather complex, and the JASM assembler does some of the necessary work to create a *BootstrapMethods* attribute for entries of binding methods.

```
class Test
        version 51:0
{


   Method m:"()V"
     stack 0 locals 1
   {
       invokedynamic REF_invokeSpecial:bsmName:"()V"
          // information about bootstrap method
                                    :methName:"(I)I"
          // dynamic call-site name ("methName")
          // plus the argument and return types of the call ("(I)I")
                                    int 1, long 2l;
          // optional sequence of additional static arguments to the
          // bootstrap method (ldc-type constants)
   }

} // end Class Test
```

his JASM code has an *invokedynamic* instruction of the form:

invokedynamic (CONSTANT_CELL(INVOKEDYNAMIC))

where the INVOKEDYNAMIC constant is represented as specified

invokedynamic INVOKESUBTAG : CONSTANT_FIELD (bootstrapmethod signature) : NAME_AND_TYPE (CallSite) [Arguments (Optional)]

The JASM assembler creates the appropriate constant entries and entries into the BootstrapMethods attribute in a resulting class file.

You can also create InvokeDynamic constants and BootstrapMethods explicitly:

```
        #22; //class Test3
        version 51:0
{

const #1 = InvokeDynamic        0:#11;
     //  REF_invokeSpecial:Test3.bsmName:"()V":name:"(I)I" int 1, long 2l
const #2 = Asciz        "Test3";
const #3 = long        2l;
const #5 = class        #6;        //  java/lang/Object
const #6 = Asciz        "java/lang/Object";
const #7 = Asciz        "name";
const #8 = int        1;
const #9 = Asciz        "SourceFile";
const #10 = Asciz        "Test3.jasm";
const #11 = NameAndType        #7:#21;        //  name:"(I)I"
const #12 = Asciz        "()V";
const #13 = Method        #22.#17;        //  Test3.bsmName:"()V"
const #14 = Asciz        "Code";
const #15 = Asciz        "m";
const #16 = Asciz        "BootstrapMethods";
const #17 = NameAndType        #20:#12;        //  bsmName:"()V"
const #18 = Asciz        "LineNumberTable";
const #19 = MethodHandle        7:#13;        //  REF_invokeSpecial:Test3.bsmName:"()V"
const #20 = Asciz        "bsmName";
const #21 = Asciz        "(I)I";
const #22 = class        #2;        //  Test3
const #23 = class        #6;        //  java/lang/Object


Method #15:#12
        stack 0 locals 1
{
        0:        invokedynamic        #1;
   //        InvokeDynamic REF_invokeSpecial:Test3.bsmName:"()V":name:"(I)I" int 1, long 2l;
}

BootstrapMethod #19 #8 #3;

} // end Class Test3
```

In this example, `const #1 = InvokeDynamic 0:#11;` is the InvokeDynamic constant that refers to BootstrapMethod at index '0' in the BootstrapMethods Attribute (`BootstrapMethod #19 #8 #3;` which refers to the *MethodHandle* at const #19, plus 2 other static args (at const #8 and const #3).

Pseudo Instructions

Pseudo instructions are 'assembler directives', and not really instructions (in the VM sense) They typically come in two forms: Code-generating Pseudo-Instructions, and Attribute-Generating Pseudo-Instructions.

Code-Generating Pseudo-Instructions

The *bytecode* directive instructs the assembler to put a collection of raw bytes into the code attribute of a method:

bytecode NUMBERS

NUMBERS is list of NUMBERs (divided by COMMA).

Inserts bytes in place of the instruction. May have any number of numeric arguments, each of them to be converted into a byte and inserted in method's code.

Attribute-Generating Pseudo-Instructions

The rest of pseudo_instructions do not produce any bytecodes, and are used to form tables: local variable table, exception table, Stack Maps, and Stack Map Frames. Line Number Tables can not be specified, but they are constructed by the assembler itself.

Local Variable Table Attribute Generation

var LOCAL_VARIABLE

```
    Starts local variable range
```

endvar LOCAL_VARIABLE

```
    Ends local variable range. LOCAL_VARIABLE means name or index of local variable table entry.
```

Example:

```
static void main (String[] args) {
        Tester inst = new Tester();
    inst.callSub();
}
```

will be coded in assembler as follows:

```
static Method #8:#9         // main:"([Ljava/lang/String;)V"
        stack 2 locals 2
{
4 var 0; // args:"[Ljava/lang/String;".
        0:        new       #1; //        class Tester;
        3:        dup;
        4:        invokespecial       #2; //        Method "<init>":"()V";
        7:        astore_1;
6 var 1; // inst:"LTester
        8:        aload_1;
        9:        invokevirtual       #3; //        Method callSub:"()V";
7        12:        return;
 endvar 0, 1;
}
```

Exception Table Attribute Generation

To generate exception table, three pseudo-instructions are used.

try TRAP_IDENT

```
      Starts trap range
```

endtry TRAP_IDENT

```
      Ends trap range
```

catch TRAP_IDENT CONSTANT_CELL(class)

```
      Starts exception handler.
```

`TRAP_IDENT` represents the name or number of an exception table entry. `CONSTANT_CELL` in "catch" pseudo_instruction means catch type. Each exception table entry contains 4 values:start-pc, end-pc, catch-pc, catch-type. In jasm, each entry is denoted with some (local) identifier, as an example: `TRAP_IDENT`.

To set start-pc, place "try TRAP_IDENT" before the instruction with the desirable program counter. Similarly, use "endtry TRAP_IDENT" for end-pc and "catch TRAP_IDENT, catch-type" for catch-pc and catch-type (which is usually a constant pool reference). Try, endtry, and catch pseudoinstructions may be placed in any order. The order of entries in exception table is significant (see JVM specification). However, the only way to control this order is to place catch-clauses in appropriate textual order: assembler adds an entry in the exception table each time it encounters a catch-clause.

Example:

```
    try {
            try {
                        throw new Exception("EXC");
            } catch (NullPointerException e){
                        throw e;
            } catch (Exception e){
                        throw e;
            }
         } catch (Throwable e){
                throw e;
         }
```

will be coded in assembler as follows:

```
try R1, R2; // single "try" or "endtry" can start several regions
                new        class java/lang/Exception;
                dup;
                ldc        String "EXC";
                invokespecial java/lang/Exception.<init>:"(Ljava/lang/String;)V";
                athrow;
 endtry R1;
 catch R1 java/lang/NullPointerException; // only one "catch" per entry allowed
                astore_1;
                aload_1;
                athrow;
 catch R1 java/lang/Exception; // same region (R1) can appear in different catches
                astore_1;
                aload_1;
                athrow;
 endtry R2;
 catch R2 java/lang/Throwable;
                astore_1;
                aload_1;
                athrow;
```

StackMap Table Attribute Generation

Stack Maps are denoted by the pseudo-op opcode stack_map, and they can be identified by three basic items:

```
stackMap_Item_MapType = (bogus | int | float | double | long | null | this | CP)

 stackMap_Item_Object = CONSTANT_CELL_CLASS
 stackMap_Item_NewObject = at LABEL
```

All stack_map directives are collected by the assembler, and are used to create a StackMap Table attribute.

Example 1 (MapType):

```
public Method "<init>":"()V"
    stack 1 locals 1
{
        aload_0;
        invokespecial    Method java/lang/Object."<init>":"()V";
        return;
        stack_frame_type full;

        stack_map bogus;
        ...
}
```

Example 2 (Object):

```
public Method "<init>":"()V"
   stack 2 locals 1
{
     ...
     stack_map class java/lang/Object;
     nop;
     return;
}
```

Example 3 (NewObject):

```
public Method "<init>":"()V"
   stack 2 locals 1
{
     ...
     stack_map at L5;
     nop;
     return;
}
```

StackFrameType Table Attribute Generation

StackFrameTypes are similar assembler directives as StackMap. These directives can appear anywhere in the code, and the assembler will collect them to produce a StackFrameType attribute.

```
frame_type = ( same | stack1 | stack1_ex | chop1 | chop2 | chop3 |
               same_ex | append | full )
```

Example 1 (full *stack frame type*):

```
public Method "<init>":"()V"
    stack 1 locals 1
{
        aload_0;
        invokespecial    Method java/lang/Object."<init>":"()V";
        return;
        stack_frame_type full;
        stack_map bogus;
        ...
}
```

Example 2 (append, chop2, and same *stack frame types*):

```
public Method foo:"(Z)V"
        stack 2 locals 5
{
        ...
                iload_2;
                iconst_2;
                if_icmpge        L30;

 L27: stack_frame_type append;
                locals_map int, int;
                iconst_2;
                istore        4;

 L30: stack_frame_type chop2;
                goto        L9;

 L33: stack_frame_type same;
                getstatic        Field java/lang/System.out:"Ljava/io/PrintStream;";
                ldc        String "Chop2 attribute test";
                invokevirtual        Method java/io/PrintStream.println:"(Ljava/lang/String;)V";
                return;
        ...
}
```

LocalsMap Table

Locals Maps are typically associated with a *stack_frame_type*, and are accumulated per stack frame. They typically follow a *stack_frame_type* directive.

```
locals_type = stackMap_Item_MapType | CONSTANT_CELL_CLASS
```

Example (a *locals map* specifying 2 ints):

```
public Method foo:"(Z)V"
        stack 2 locals 5
{
        ...
                iload_2;
                iconst_2;
                if_icmpge        L30;
        L27:        stack_frame_type append;

 locals_map int, int;
                iconst_2;
                istore        4;
        L30:        stack_frame_type chop2;
                goto        L9;
        L33:        stack_frame_type same;
                getstatic        Field java/lang/System.out:"Ljava/io/PrintStream;";
                ldc        String "Chop2 attribute test";
                invokevirtual        Method java/io/PrintStream.println:"(Ljava/lang/String;)V";
                return;
        ...
}
```

Inner-Class Declarations

INNER_CLASS_DECLARATIONS: list of

```
    INNER_CLASS_DECLARATION
```

INNER_CLASS_DECLARATION:

```
    INNER_CLASS_ACCESS InnerClass [INNER_CLASS_NAME=]? INNER_CLASS_INFO [of OUTER_CLASS_INFO]? ;
```

INNER_CLASS_NAME:

```
    IDENT | CPX_name
```

INNER_CLASS_INFO:

```
    CONSTANT_CELL(class)
```

OUTER_CLASS_INFO:

```
    CONSTANT_CELL(class)
```

INNER_CLASS_ACCESS: list of

```
    [public|protected|private][static][final][interface][abstract]
[synthetic][annotation][enum]
```

Example:

```
    InnerClass InCl=class test$InCl of class test;
```

Annotation Declarations

Member Annotations

Member annotations are a subset of the basic annotations support provided in JDK 5.0 (1.5). These are annotations that ornament Packages, Classes, and Members either visibly (accessible at runtime) or invisibly (not accessible at runtime). In JASM, visible annotations are denoted by the token **@**, while invisible annotations are denoted by the token **@-**.

Synopsis

ANNOTATIONS:

```
[ANNOTATION_DECLARATION]+;
```

ANNOTATION_DECLARATION:

```
@+|@- ANNOTATION_NAME [ANNOTATION_VALUE_DECLARATIONS]
```

The '@+' token identifies a Runtime Visible Annotation, where the '@-' token identifies a Runtime Invisible Annotation.

ANNOTATION_NAME:

```
IDENT
```

ANNOTATION_VALUE_DECLARATIONS: list of (comma separated)

```
ANNOTATION_VALUE_DECLARATION
```

ANNOTATION_VALUE_DECLARATION:

```
[ANNOTATION_VALUE_IDENT=] [ANNOTATION_VALUE]
```

ANNOTATION_VALUE_IDENT:

```
IDENT
```

ANNOTATION_VALUE:

```
ANNOTATION_VALUE_PRIMITIVE | Array of ANNOTATION_VALUE_PRIMITIVE
```

ANNOTATION_VALUE_PRIMITIVE:

```
PRIMITIVE_TYPE | STRING | CLASS | ENUM | ANNOTATION_DECLARATION
```

CLASS:

```
class CONSTANT_CELL(class)
```

ENUM:

```
enum CONSTANT_CELL(class) CONSTANT_CELL(string) (where string is Enum type name)
```

PRIMITIVE_TYPE:

```
BOOLEAN | BYTE | CHAR | SHORT | INTEGER | LONG | FLOAT | DOUBLE
```

The '**@+**' token identifies a Runtime Visible Annotation, where the '@-' token identifies a Runtime Invisible Annotation.

Note

Types (Boolean, Byte, Char, and Short) are normalized into Integer's within the constant pool.
Annotation values with these types may be identified with a keyword in front of an integer value.

eg. **boolean** true (or: boolean 1)
byte 20
char 97
short 2130

Other primitive types are parsed according to normal prefix and suffix conventions
(eg. Double = xxx.x**d**, Float = xxx.x**f**, Long = xxx**L**).
Strings are identified and delimited by "" (quotation marks).

Keywords '**class**' and '**enum**' identify those annotation types explicitly. Values within classes and enums may either be identifiers (strings) or Constant Pool IDs.

Annotations specified as the value of an Annotation field are identified by the JASM annotation keywords '**@+**' and '**@-**'.

Arrays are delimited by '{' and '}' marks, with individual elements delimited by ',' (comma).

Examples

Example 1 (Class Annotation, Visible)

```
@+ClassPreamble {
     author = "John Doe",
     date = "3/17/2002",
     currentRevision = 6,
     lastModified = "4/12/2004",
     lastModifiedBy = "Jane Doe",
     reviewers = {
        "Alice",
        "Bob",
        "Cindy"}
}

super public class MyClass
        version 50:0
{
 ...
```

Example 2 (Field Annotation, Invisible)

```
@-FieldPreamble {
     author = "Mustafa",
     date = "3/17/2009",
     currentRevision = 4
}
Field foo:I;

...
```

Example 3 (Field Annotation, All subtypes)

```
@+FieldPreamble {
        boolAnnot      = boolean 1,                                      // Boolean
        charBear       = char 97,                                      // Char
        sharkByte      = byte 17,                                      // Byte
        shortCircuit   = short 4386,                                     // Short
        integerHead    = 42,                                             // Int
        longJohnSilver = 55l,                                              // Long
        floatBoat      = 1.0f,                                               // Float
        doubleDip      = 10.0d,                                                // Double
        stringBeans    = "foo",                                                 // String
        severity       = enum FieldPreamble$Severity IMPORTANT,          // Enum
        classAnnot     = class FieldPreamble$FooBall,                      // Class
        tm = @+Trademark { description = "embedded", owner = "ktl"}       // Annotation
}
Field foo:I;

...
```

Note:
JASM does not enforce the annotation value declarations like a compiler would. It only checks to see that an annotation structure is well-formed.

Type Annotations

Member annotations are a subset of the basic annotations support provided in JDK 7.0 (1.7). These are annotations that ornament Packages, Classes, and Members either visibly (accessible at runtime) or invisibly (not accessible at runtime). In JASM, visible annotations are denoted by the token **@T+**, while invisible annotations are denoted by the token **@T-**.

Synopsis

***TYPE_ANNOTATION_DECLARATION*:@T+|@T-** ANNOTATION_NAME [TYPE_ANNOTATION_VALUE_DECLARATIONS]

TYPE_ANNOTATION_VALUE_DECLARATIONS: list of (comma separated)
TYPE_ANNOTATION_VALUE_DECLARATION

***TYPE_ANNOTATION_VALUE_DECLARATION*:{ {** ANNOTATION_VALUE_DECLARATION^{+} **}** TARGET PATH **}**

***TARGET*:{** TARGET_TYPE TARGET_INFO **}**

TARGET_TYPE:

***TARGET_TYPE*:**	***TARGET_INFO_TYPE*:**
CLASS_TYPE_PARAMETER	**TYPEPARAM**
METHOD_TYPE_PARAMETER	**TYPEPARAM**
CLASS_EXTENDS	**SUPERTYPE**
CLASS_TYPE_PARAMETER_BOUND	**TYPEPARAM_BOUND**
METHOD_TYPE_PARAMETER_BOUND	**TYPEPARAM_BOUND**
FIELD	**EMPTY**
METHOD_RETURN	**EMPTY**
METHOD_RECEIVER	**EMPTY**
METHOD_FORMAL_PARAMETER	**METHODPARAM**
THROWS	**EXCEPTION**
LOCAL_VARIABLE	***LOCALVAR***
RESOURCE_VARIABLE	***LOCALVAR***
EXCEPTION_PARAM	**CATCH**
INSTANCEOF	***OFFSET***
NEW	***OFFSET***
CONSTRUCTOR_REFERE CE_RECEIVER	***OFFSET***
METHOD_REFERENCE_RECEIVER	***OFFSET***
CAST	**TYPEARG**
CONSTRUCTOR_INVOCATION_TYPE_ARGUMENT	**TYPEARG**
METHOD_INVOCATION_TYPE_ARGUMENT	**TYPEARG**
CONSTRUCTOR_REFERENCE_TYPE_ARGUMENT	**TYPEARG**
METHOD_REFERENCE_TYPE_ARGUMENT	**TYPEARG**

TARGET_INFO_TYPE: TYPEPARAM | SUPERTYPE | TYPEPARAM_BOUND | EMPTY | METHODPARAM | EXCEPTION | LOCALVAR | CATCH | OFFSET | TYPEARG
TYPEPARAM:
paramIndex(*INTEGER*)

SUPERTYPE:
typeIndex(*INTEGER*)typeIndex(INTEGER)

TYPEPARAM_BOUND:
paramIndex(*INTEGER*) boundIndex(*INTEGER*)

EMPTY:

METHODPARAM:
index(paramIndex(*INTEGER*)
EXCEPTION:
typeIndex(*INTEGER*)

LOCALVAR:
{ **LVENTRY** }**+numEntries**

LVENTRY:
startpc(*INTEGER)* length(*INTEGER)* index(*INTEGER)*

offset(*INTEGER)*
TYPEARG:
offset(*INTEGER*) typeIndex(*INTEGER*)

PATH: list of (space separated)
{ *PATH_ENTRY*+ **}**
{ PATH_KIND PATH_INDEX }

PATH_KIND: ARRAY | INNER_TYPE | WILDCARD | TYPE_ARGUMENT

PATH_INDEX:
INTEGER

Parameter Names and Parameter Annotations

Parameter annotations are another subset of the basic annotations support provided in JDK 5.0 (1.5). These are annotations that ornament Parameters to methods either visibly (accessible at runtime) or invisibly (not accessible at runtime). In JASM, visible parameter annotations are denoted by the token **@+**, while invisible parameter annotations are denoted by the token **@-**.

Parameter names come from an attribute introduced in JDK 8.0 (1.8). These are fixed parameter names that are used to ornament parameters on methods. In Jasm, parameter names are identified by the token # followed by { } braclets

Synopsis

METHOD DECLARATION:

```
MODIFIERS Method METHOD_NAME:"METHOD_SIGNATURE" [STACK_DECL] [LOCALS_DECL] [PARAMETERS_DECL] {[CODE]}
```

PARAMETERS_DECL:

```
[PARAMETER_DECL]N (where N < number of params in method, each N is a unique param number)
```

PARAMETER_DECL:

```
PARAM_NUM : [PARAM_NAME_DECL] [ANNOTATION_DECLARATIONS]
```

PARAM_NAME_DECL:

```
#{ name PARAM_ACCESS}
```

PARAM_ACCESS: list of

```
[final][synthetic][mandated]
```

Examples

Example 1 (Parameter Annotation)
Java Code

```
public class MyClass2 {

 ...

    public int doSomething(
        @VisParamPreamble ( author = "gummy" )  @InVisParamPreamble ( author = "bears" )  int barber,
        boolean of,
        @VisParamPreamble ( author = "sour" )  @InVisParamPreamble ( author = "worms" )    int seville,
        @InVisParamPreamble1 ( reviewers = {"Dilbert", "Garfield"} ) boolean pastrami) {
         ...
    }
 ...
}
```

JASM Code

Note: The first two parameters are named ('P0'- 'P3'). Since this is a compiler controlled option, there is no way to specify parameter naming in Java source.

```
super public class MyClass2
        version 50:0
{
 ...

  public Method doSomething:"(IZIZ)I"
        stack 2 locals 5

        0: #{P0 mandated} @+VisParamPreamble { author = "gummy" } @-InVisParamPreamble { author = "bears" }
        1: #{P1 final synthetic mandated}
        2: #{P2 mandated} @+VisParamPreamble { author = "sour" } @-InVisParamPreamble { author = "worms" }
        3: #{P3 mandated} @-InVisParamPreamble1 { reviewers = {  "Dilbert",  "Garfield"} }
   {
        ...
   }

} // end Class MyClass2
```

Default Annotations

Default annotations are another subset of the basic annotations support provided in JDK 5.0 (1.5). These are annotations that ornament Annotations either visibly (accessible at runtime) or invisibly (not accessible at runtime). Default annotations specify a default value for a given annotation field.

Synopsis

ANNOTATION INTERFACE DECLARATION:

```
@interface ANNOTATION_NAME { ANNOTATION_FIELD_DECL+ }
```

ANNOTATION_FIELD_DECL:

```
ANNOT_FIELD_TYPE ANNOTATION_NAME [ANNOTATION_DEFAULT_VALUE_DECL];
```

ANNOTATION_DEFAULT_VALUE_DECL:

```
default ANNOTATION_VALUE (where value must be of the type ANNOT_FIELD_TYPE)
```

Examples

Example 1 (Default Annotation)

Java Code

```
import java.lang.annotation.*;

@Retention(RetentionPolicy.RUNTIME)

@interface Meth2Preamble {
   String author() default "John Steinbeck";
}
```

JASM Code

```
interface  Meth2Preamble
    implements java/lang/annotation/Annotation
    version 50:0

{
   public abstract Method author:"()Ljava/lang/String;" default { "John Steinbeck" } ;
} // end Class Meth2Preamble
```

PicoJava Instructions

These instructions takes 2 bytes: prefix (254 for non-privileged variant and 255 for privileged) and the opcode itself. These instructions can be coded in assembler in 2 ways: as single mnemocode identical to the description or using "priv" and "nonpriv" instructions followed with an integer representing the opcode.

Appendix B