STM32Java Platform Architecture

STM32JavaF2 - Keil uVision

Reference Manual

Reference:	TLT-0595-REF-STM32JavaF2
Revision:	D
Architecture:	STM32JavaF2
Compiler:	Keil uVision
Product Version:	5.0.2

Confidentiality & Intellectual Property

Table of Contents

List of Figures

List of Tables

1 Bibliography

[JVM]	Tim Lindholm & Frank Yellin, The Java™ Virtual Machine Specification, Second Edition, 1999
[EDC]	Embedded Device Configuration: ESR 021, http://www.e-s-r.net
[B-ON]	Beyond: ESR 001, http://www.e-s-r.net
[SNIGT]	Simple Native Interface for Green Threads: ESR 012, http://www.e-s-r.net
[SP]	Shielded Plug: ESR 014, http://www.e-s-r.net

2 Introduction

2.1 Scope

STM32JavaF2 ARMCCv4 JPF is state-of-the-art embedded Java runtimes for STM32JavaF2 MCUs. They also provide simulated runtimes that execute on workstations to allow software development on "virtual hardware".

This reference manual describes the functionality of the JPFs. It is concise, but attempts to be exact and complete. Semantics of implemented standard libraries are described in their respective specifications. This reference manual includes only the specific APIs related to porting the JPFs to different real-time operating systems (RTOS).

2.2 Intended Audience

The audience for this document is software engineers who need to understand the details of the JPF components, including their APIs, error codes and options.

2.3 Configuration Variables

This document describes a number of *configuration variables*. A configuration variable is a value that can be set by the user to configure a component. The mechanism for setting the variable differs depending on the component and the environment, and is not described in this document.

Each variable is given a code of the form: `CATEGORY_nn`, where `CATEGORY` is a label that identifies the configuration category, and `nn` is a two-digit number identifying the specific variable within the category. Other documents refer to these codes. Consult the platform-specific User's Manual to see how to set or view configuration variables.

2.4 JPF Components Overview

STM32JavaF2 ARMCCv4 JPF feature a tiny and fast runtime associated with a smart RAM optimizer. They provide four built-in libraries [B-ON], [EDC], [SNIGT] and [SP]. Figure 2.1 shows the components involved.

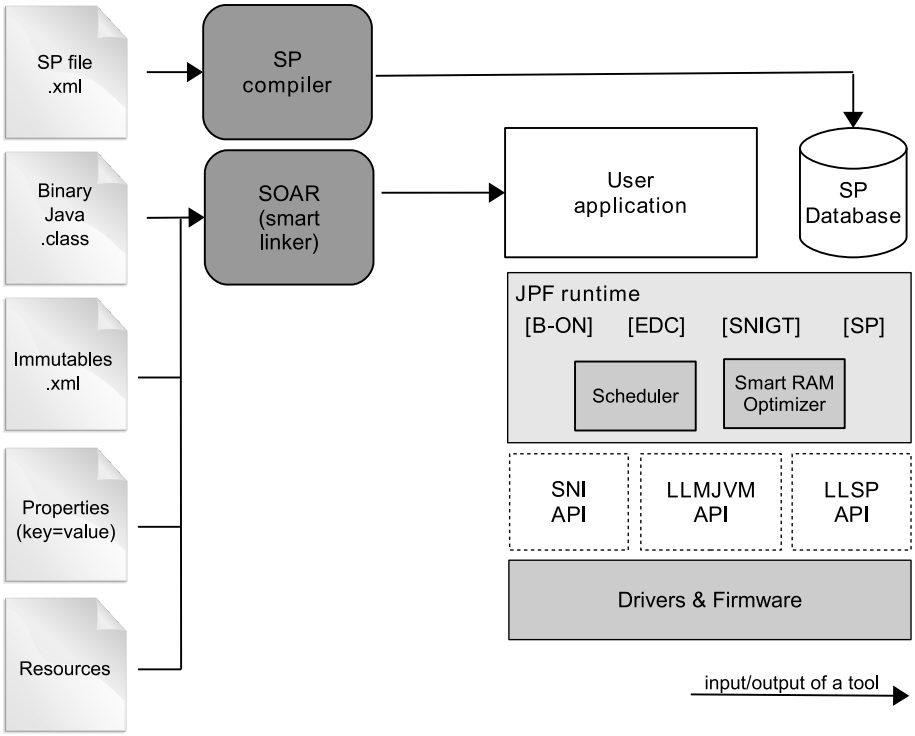

Figure 2.1. JPF Runtime Components: tools, libraries & APIs

The three APIs Simple Native Interface (SNI), Low Level MicroJvm™ virtual machine (LLMJVM), and Low Level Shielded Plug (LLSP) allow the JPF runtime to link (and port) to legacy code, such as any kind of RTOS or legacy C libraries.

2.5 Scheduler

The JPF features a green thread architecture platform that can interact with the C world [SNIGT]. The (green) thread policy is as follow:

- preemptive for different priorities,
- round-robin for same priorities,
- "priority inheritance protocol" when priority inversion occurs [1] .

Java stacks (associated with the threads) automatically adapt their sizes according to the thread requirements: once the thread has finished, its associated stack is reclaimed, freeing the corresponding RAM memory.

2.6 Smart RAM Optimizer

The JPF includes a state-of-the-art memory management system, the Garbage Collector (GC). It manages a bounded piece of RAM memory, devoted to the Java world.The GC automatically frees dead Java objects, and defragments the memory in order to optimize RAM usage. This is done transparently while the Java applications keep running.

[1]This protocol raises the priority of a thread that is holding a resource needed by a higher priority task to the priority of that task.

3 Edition – Name – Version

Editions	EVAL / DEV
Name	STM32JavaF2 ARMCCv4
Version	5.0.2
MCU	STM32JavaF2
Compiler	Keil uVision

Table 3.1. Platform references

4 Features

4.1 Resource Requirements

Resource	EmbJPF Characteristics
Flash	Less than 30 KB
RAM	Less than 1.5 KB
RTOS	Any RTOS [a]
Architecture	Green Thread Java stacks auto-sizable [b]
RTOS Stack Size	Less than 1 KB
Startup Time	Less than 2 ms at 120MHz

[a]As a special case, the JPF can be used in a system that runs only one main task: the JPF. So the JPF can be used with any RTOS or with no RTOS at all.

[b]Java stacks automatically adapt their sizes according to the Java (green) thread demand. Once the thread has finished, its associated stack is automatically reclaimed by the GC.

Table 4.1. Required resources

4.2 Standard Libraries

Library	Reference	EmbJPF Versions	SimJPF Versions	User Configurable
EDC	[EDC]	1.2	1.2	yes
B-ON	[B-ON]	1.2	1.2	yes
SNI	[SNIGT]	1.2	1.2	

Table 4.2. Standard libraries

4.3 Specific Libraries & APIs

Library	Reference	EmbJPF Versions	SimJPF Versions	User Configurable
SP	[SP]	1.0	1.0	yes
LLSP API	This document	1.0	n/a	
LLMJVM API	This document	1.0	n/a	
HIL API	This document	n/a	1.0	
ECOM Comm	This document	1.0	1.0	yes

Table 4.3. Specific API

4.4 Platform Characteristics

Name	Item	EmbJPF Characteristics	SimJPF Characteristics	User Configurable
RAM optimizer	Heap Partition	1	1	
	Immortal Space	Yes	Yes	yes
	Immutable Space	Yes (static)	Yes (static)	
Debug	Symbolic	No	JDWP (Socket)	yes

Name	Item	EmbJPF Characteristics	SimJPF Characteristics	User Configurable
Java Code	Location	In Flash (in place execution)	n/a	

Table 4.4. Platform characteristics

4.5 Configuration

The JPF has a number of application-specific configuration variables (Table 4.5).

Variable	Meaning
`JPF_01`	Configures the Java heap size (in bytes).
`JPF_02`	Configures the maximum number of Java threads that can run simultaneously.
`JPF_03`	Configures the size of Java thread stack pool, given in quantity of blocks of 512 bytes (2 blocks means 1024 (2x512)).
`JPF_04`	Configures the maximum Java thread stack size, given in quantity of blocks of 512 bytes. Must be in the range `[1..JPF_03]`.

Table 4.5. JPF configuration variables

5 SOAR: the Smart Linker

5.1 Introduction

Java source code is compiled by the Java compiler[2] into the binary format specified in [JVM]. This binary code needs to be linked before execution. The JPF comes with a linker, named the SOAR. It is in charge of analyzing `.class` files, and some other application-related files, to produce the final application that the JPF runtime can execute.

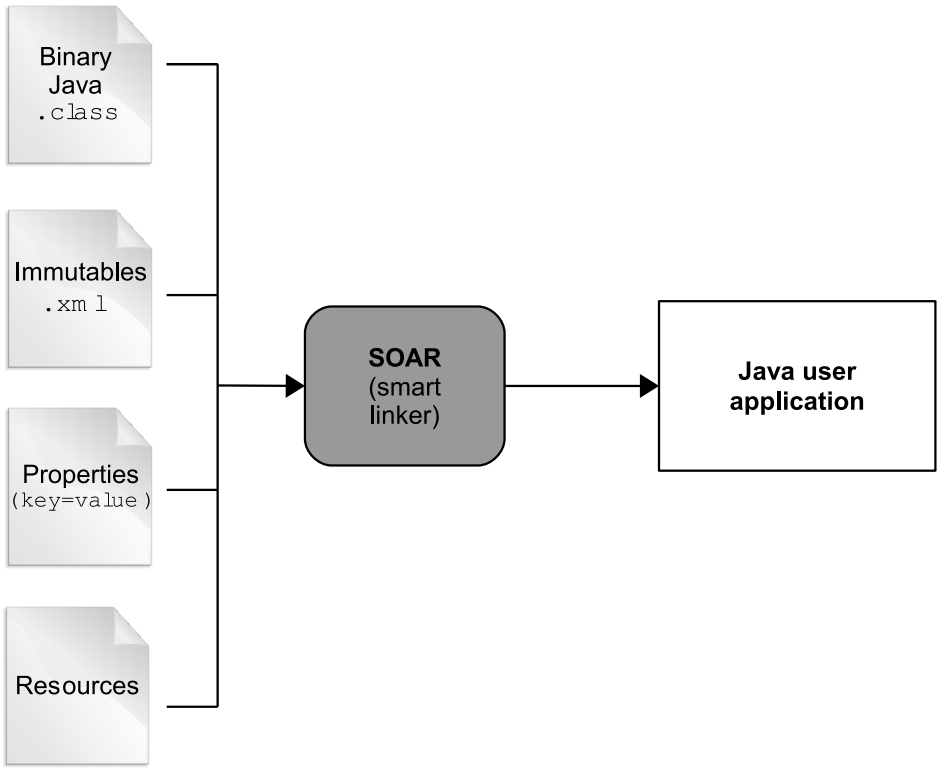

Figure 5.1. The SOAR inputs & outputs

5.1.1 Inputs

- All `.class` files necessary to run the main class.
- The Immutable files description used by the application (see [B-ON])
- Properties files
- Resources (Images, Native Language Support data, …)

5.1.2 Outputs

- The SOAR image representing the linked user Java application. The format used by the SOAR is the ELF format.
- Documentation in the form of `.xml` file, providing useful information about the generated image: the initialization phase order (`<clinit>` order [B-ON]), selected items (methods, resources, immutables) with their associated footprints. Debug information is also provided for the JDWP based Java Eclipse debugger.

[2]The JDT compiler from the Eclipse IDE.

5.2 Loading Process

SOAR only loads necessary classfiles and only embeds in output file what is needed from main class and user required types[3], plus all provided immutables, resources, and properties.

5.3 Error Messages

When a generic exception is thrown by the SOAR, the error message

SOAR ERROR [M<messageId>] <message>

is issued, where **<messageId>** and **<message>** meaning are defined in next table.

Message ID	Description
0	The SOAR process has encountered some internal limits.
1	Unknown option.
2	An option has an invalid value.
3	A mandatory option is not set.
4	A filename given in options does not exist .
5	Failed to write output file (access permissions required for `-toDir` and `-root` options).
6	The given file does not exist.
7	IO Error while reading a file.
8	An option value refers to a directory instead of a file.
9	An option value refers to a file instead of a directory or a jar file.
10	Invalid entry point class or no `main()` method.
11	Information file can not be generated entirely.
12	Limitations of the evaluation version reached.
13	IO Error while reading a jar file.
14	IO Error while writing a file.
15	IO Error while reading a jar file: unknown entry size.
16	Not enough memory to load a jar file.
17	Specified SOAR options are exclusive.
18	XML syntax error for some given files.
19	Unsupported float representation.
50	Missing code: Java code refers to a method not found in specified classes.
51	Missing code: Java code refers to a class not found in the specified classpath.
52	Wrong class: Java code refers to a field not found in specified class.
53	Wrong class: Java classfile refers to a class as an interface.
54	Wrong class: an abstract method is found in a non abstract class.
55	Wrong class: illegal access to a method, a field or a type.
56	Wrong class: hierarchy inconsistency; an interface cannot be superclass of a class.
57	Circularity detected in initializion sequence.
58	Option refers twice the same resource. The first one is used.
59	Stack inconsistency detected.

[3]Types that may be dynamically loaded using `Class.forName()` method need to be declared as required types.

Message ID	Description
60	Constant pool inconsistency detected.
61	Corrupted classfile.
62	Missing native implementation of a native method.
63	Cannot read the specified resource file.
64	A same property name cannot be defined in two different property files.
65	Bad license validity.
66	Classfiles do not contains debug line table information.
67	Same as 51
150	SOAR limit reached: the specified method uses too many arguments.
151	SOAR limit reached: the specified method uses too many locals.
152	SOAR limit reached: the specified method code is too large.
153	SOAR limit reached: the specified method catches too many exceptions.
154	SOAR limit reached: the specified method defines a too large stack.
155	SOAR limit reached: the specified type defines too many methods.
156	SOAR limit reached: your application defines too many interface.
157	SOAR limit reached: the specified type defines too many fields.
158	SOAR limit reached: your application defines too many types.
159	SOAR limit reached: your application defines too many static fields.
160	SOAR limit reached: the hierarchy depth of the specified type is too high.
161	SOAR limit reached: your application defines too many bundles.
251	Error in converting IEE754 float(32) or double(64) to fixed point arithmetic number
300	Corrupted class: invalid dup_x1 instruction usage.
301	Corrupted class: invalid dup_x2 instruction usage.
302	Corrupted class:invalid dup_x2 instruction usage.
303	Corrupted class:invalid dup2_x1 instruction usage.
304	Corrupted class:invalid dup2_x1 instruction usage.
305	Corrupted class:invalid dup2_x2 instruction usage.
306	Corrupted class: invalid dup2 instruction usage.
307	Corrupted class:invalid pop2 instruction usage.
308	Corrupted class:invalid swap instruction usage.
309	Corrupted class:finally blocks must be inlined.
350	SNI incompatibility: some specified type should be an array.
351	SNI incompatibility: some type should defined some specified field.
352	SNI incompatibility: the specified field is not compatible with SNI.
353	SNI incompatibility: the specified type must be a class.
354	SNI incompatibility: the specified type must defined the specified static field.
355	SNI file error : the data must be an integer.
356	SNI file error : unexpected tag
357	SNI file error : attributes <name>, <descriptor>, <index> and <size> are expected in the specified tag.

Message ID	Description
358	SNI file error : invalid SNI tag value.
359	Error parsing SNI file.
360	XML Error on parsing SNI file.
361	SNI incompatibility : illegal call to the specified data.
362	No stack found for the specified native group
363	Invalid SNI method: The argument cannot be an object reference
364	Invalid SNI method: The array argument must only be a base type array
365	Invalid SNI method: The return type must be a base type
366	Invalid SNI method: The method must be static

Table 5.1. SOAR error messages.

6 Java Core Libraries

A Java core library is the library which provides the basic Java concepts and classes. Without this library it is not possible to write any Java application.

MicroEJ provides several Java core libraries. Only one library can be installed into the platform at any time.

6.1 Java Properties

Java properties allow the Java application to be parameterized using the `System.getProperty` API. The definition of the properties and their respective values is done using files. Each filename of a properties file must match with `*.system.properties` and must be located in `properties` package of the application classpath. These files follow the Java property list specification: key, value pairs.

```
microedition.encoding=ISO-8859-1
```

Figure 6.1. Example of content of a Java properties file

6.2 Generic Output

The `System.err` stream is connected to the `System.out` print stream. See below for how to configure the destination of these streams.

6.3 Error Messages

When an exception is thrown by the runtime, the error message

Generic:E=<messageId>

is issued, where **<messageId>** meaning is defined in the next table:

Message ID	Description
1	Negative offset.
2	Negative length.
3	Offset + length > object length.

Table 6.1. Generic error messages

6.4 Exit Codes

The RTOS task that runs the Java runtime may end, especially when the Java application calls `System.exit` method [EDC]. By convention, a negative value indicates abnormal termination.

Message ID	Meaning
0	The Java application ended normally.
-1	The SOAR and the JPF are not compatible.
-2	Incompatible link configuration (`lsc` file) with either the SOAR or the JPF.
-3	Evaluation version limitations reached: termination of the application.
-5	Not enough resources to start the very first Java thread, that executes `main` method.
-12	Maximum number of threads reached.
-13	Fail to start the JPF because the specified Java heap is too large.
-14	Invalid stack space due to a link placement error.
-15	The application has too many static (the requested static head is too large).

Message ID	Meaning
-16	The JPF virtual machine cannot be restarted.

Table 6.2. JPF exit codes

7 EDC

EDC is a Java core library (see “Java Core Libraries”).

7.1 Error Messages

When an exception is thrown by the implementation of the EDC API, the error message

EDC-1.2:E=<messageId>

is issued, where **<messageId>** meaning is defined in the next table:

Message ID	Description
-4	No native stack found to execute the Java native method.
-3	Maximum stack size for a thread has been reached. Increase the maximum size of thread stack parameter.
-2	No Java stack block could be allocated with the given size. Increase the Java stack block size.
-1	The Java stack space if full. Increase the Java stack size or the number of Java stack blocks.
1	A closed stream is being written/read.
2	The operation `Reader.mark()` is not supported.
3	`lock` is `null` in `Reader(Object lock)`.
4	String index is out of range.
5	Argument must be a positive number.
6	Invalid radix used. Must be from `Character.MIN_RADIX` to `Character.MAX_RADIX`.

Table 7.1. EDC error messages

7.2 Configuration

The library implementation has application specific configuration variables:

Variable	Meaning
CORE_01	Configures the redirection of standard output to a user defined `OutputStream`. By default, standard output is internally connected to the platform's C `printf` implementation.
CORE_02	Configures string encoding(s) that are embedded at runtime. The JPF provides the following encoding(s): `UTF-8`.

Table 7.2. EDC configuration variables

8 B-ON

8.1 Immutable Files Related Error Messages

The following error messages are issued at SOAR time (link phase) and not at runtime.

Message ID	Description
0	Duplicated ID in immutable files. Each immutable object should have an unique ID in SOAR image.
1	Immutable file refers an unknown field of an object.
2	Tried to assign twice the same object field.
3	All immutable object fields should be defined in the immutable file description.
4	The assigned value does not match the expected Java type.
5	An immutable object refers to an unknown ID.
6	The length of the immutable object does not match the length of the assigned object.
7	The type defined in the file doesn't match the Java expected type.
8	Generic error while parsing an Immutable file.
9	Cycle detected in alias definition.
10	An immutable object is an instance of an abstract class or an interface.
11	Unknown XML attribute in an immutable file.
12	A mandatory XML attribute is missing.
13	The value is not a valid Java literal.
14	Alias already exists.

Table 8.1. Errors when parsing immutable files at link time.

8.2 Configuration

The library implementation has application specific configuration variables:

Message ID	Meaning
`BON_02`	Configures the immortal heap size in bytes.

Table 8.2. B-ON configuration variables

9 SNI

For details about SNI see the specification [SNIGT]. Note that [SNIGT] defines the MicroJvm life-cycle API (create, start, stop, destroy) – the usage of this API is described in Section 13, “LLMJVM: Low Level JPF API ”, of this document.

9.1 SNI Link Time Error Messages

The following error messages are issued at SOAR time and not at runtime.

Message ID	Description
363	Argument cannot be a reference.
364	Argument can only be from a base type array.
365	Return type must be a base type.
366	Method must be a static method.

Table 9.1. SNI error messages.

10 SP

10.1 XML File Description

The Shielded Plug [SP] provides data segregation with a clear publish-subscribe API. The data sharing between modules uses the concept of shared memory blocks, with introspection. The database is made of blocks: chunks of RAM.

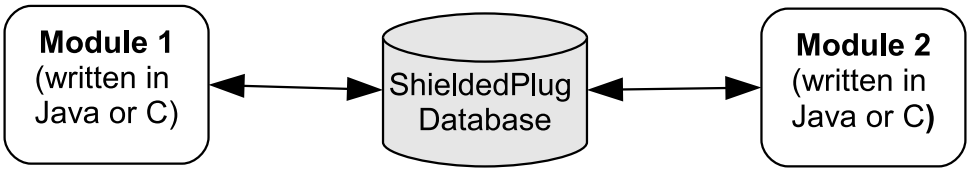

Figure 10.1. A Shielded Plug between two application (Java/C) modules.

The implementation of the SP for the JPF uses a XML file description to describe the database; the syntax follows the one proposed by [SP].

10.2 SP Compiler

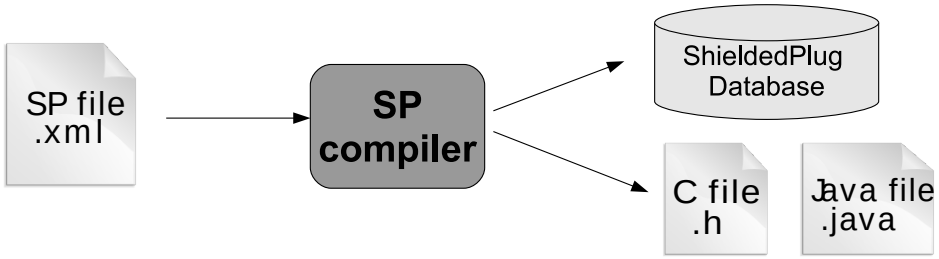

Figure 10.2. Shielded Plug compiler flow.

The Shielded Plug compiler takes as input a shielded plug description (XML). It outputs:

- A description of the requested resources of the database as a binary file (`.o`) that will be linked to the overall application by the linker. It is an ELF format description that reserves both the necessary RAM and the necessary Flash memory for the database of the shielded plug.
- Two descriptions, one in Java and one in C, of the block ID constants to be used by either Java or C application modules.

Option name	Description
`-verbose[e...e]`	Extra messages are printed out to the console according to the number of 'e'.
`-descriptionFile file`	XML Shielded Plug description file. Multiple files allowed.
`-waitingTaskLimit value`	Max number of task/threads that can wait on a block: number between 0 and 7. -1 is for no limit, 8 for unspecified.
`-immutable`	When specified, only immutable Shielded Plugs can be compiled.
`-output dir`	Output directory. Default is the current directory.
`-outputName name`	Output name for the Shielded Plug layout description. Default is "shielded_plug".
`-endianness name`	Either "little" or "big". Default is "little".
`-outputArchitecture value`	Output ELF architecture. Only "ELF" architecture available.
`-rwBlockHeaderSize value`	Read/Write header file value.

Option name	Description
`-genIdsC`	When specified, generate C header file with block ID constants.
`-cOutputDir dir`	Output directory of C header files. Default is the current directory.
`-cConstantsPrefix prefix`	C constants name prefix for block IDs
`-genIdsJava`	When specified, generate Java interfaces file with block ID constants.
`-jOutputDir dir`	Output directory of Java interfaces files. Default is current directory.
`-jPackage name`	The name of the package for Java interfaces.

Table 10.1. Shielded Plug compiler options.

10.3 Error Messages

Message ID	Description
0	Internal limits reached.
1	Invalid endianness.
2	Invalid output architecture.
3	Error while reading / writing files.
4	Missing mandatory option.

Table 10.2. Shielded Plug compiler error messages.

11 ECOM

ECOM is the communication core library.

11.1 Error Messages

When an exception is thrown by the implementation of the ECOM API, the error message

ECOM-1.0:E=<messageId>

is issued, where **<messageId>** meaning is defined in the next table:

Message ID	Description
1	The connection has been closed. No more action can be done on this connection.
2	The connection has been already closed.
3	The connection description is invalid. The connection cannot be opened.
4	The connection stream has been already opened. Only one stream per kind of stream (input or output stream) can be opened at the same time.
5	Too many connections have been opened at the same time. The platform is not able to open a new one. Try to close an useless connection before trying to open the new connection.

Table 11.1. ECOM error messages

12 ECOM Comm

12.1 Configuration

The library implementation has application specific configuration variables:

Variable	Meaning
ECOM_COMM_01	Configures whether the ECOM Comm connection factory is enabled. If it is not enabled it is not possible to create ECOM Comm connections.
ECOM_COMM_02	Configures the mappings between logical port ids and the defined physical ids.

Table 12.1. ECOM Comm configuration variables

13 LLMJVM: Low Level JPF API

13.1 Principle

13.2 Naming Convention

The Low Level MicroJvm API, the `LLMJVM` API, relies on functions that need to be implemented. The naming convention for such functions is that their names match the `LLMJVM_IMPL_*` pattern.

13.3 Porting the JPF

Here is a summary of the functions that need to be implemented to port the JPF:

- `LLMJVM_IMPL_initialize`
- `LLMJVM_IMPL_vmTaskStarted`
- `LLMJVM_IMPL_scheduleRequest`
- `LLMJVM_IMPL_idleVM`
- `LLMJVM_IMPL_wakeupVM`
- `LLMJVM_IMPL_ackWakeup`
- `LLMJVM_IMPL_getCurrentTaskID`
- `LLMJVM_IMPL_setApplicationTime`
- `LLMJVM_IMPL_getCurrentTime`
- `LLMJVM_IMPL_getTimeNanos`
- `LLMJVM_IMPL_isInReadOnlyMemory`

14 LLCOMM: Low Level ECOM Comm API

14.1 Naming Convention

The Low Level Comm API (LLCOMM), relies on functions that need to be implemented by engineers in a driver. The names of these functions match the `LLCOM_BUFFERED_CONNECTION_IMPL_*` or the `LLCOM_CUSTOM_CONNECTION_IMPL_*`pattern.

14.2 Header Files

Four C header files are provided:

- LLCOMM_BUFFERED_CONNECTION_impl.h

  Defines the set of functions that the driver must implement to provide a Buffered connection

- LLCOMM_BUFFERED_CONNECTION.h

  Defines the set of functions provided by ECOM Comm that can be called by the driver (or other C code) when using a Buffered connection

- LLCOMM_CUSTOM_CONNECTION_impl.h

  Defines the set of functions that the driver must implement to provide a Custom connection

- LLCOMM_CUSTOM_CONNECTION.h

  Defines the set of functions provided by ECOM Comm that can be called by the driver (or other C code) when using a Custom connection

15 Simulation

15.1 HIL Engine Options

Below are the HIL Engine options:

Option name	Description
`-verbose[e....e]`	Extra messages are printed out to the console (add extra e to get more messages)
`-ip <address>`	SimJPF connection IP address (A.B.C.D). By default set to local-host.
`-port <port>`	SimJPF connection port. By default set to 8001.
`-connectTimeout <timeout>`	timeout in s for SimJPF connections. By default set to 10 seconds.
`-excludes <name[sep]name>`	Types that will be excluded from the HIL Engine class resolution provided mocks. By default, no types are excluded.
`-mocks <name[sep]name>`	Mocks are either `.jar` file or `.class` files.

Table 15.1. HIL Engine options

15.2 Heap Dumping

15.2.1 XML Schema

Below is the XML schema for heap dumps.

```
<?xml version='1.0' encoding='UTF-8'?>
<!--
    Schema

    Copyright 2012 IS2T. All rights reserved.
    IS2T PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
-->

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <!-- root element : heap -->
    <xs:element name="heap">
        <xs:complexType>
            <xs:choice minOccurs="0" maxOccurs="unbounded">
                <xs:element ref="class"/>
                <xs:element ref="object"/>
                <xs:element ref="array"/>
                <xs:element ref="stringLiteral"/>
            </xs:choice>
        </xs:complexType>
    </xs:element>
```

```
    <!-- class element -->
    <xs:element name="class">
        <xs:complexType>
            <xs:choice minOccurs="0" maxOccurs="unbounded">
                <xs:element ref="field"/>
            </xs:choice>
            <xs:attribute name="name" type="xs:string" use = "required"/>
            <xs:attribute name="id" type="xs:string" use = "required"/>
            <xs:attribute name="superclass" type="xs:string"/>
        </xs:complexType>
    </xs:element>
```

```
    <!-- object element-->
    <xs:element name="object">
        <xs:complexType>
            <xs:choice minOccurs="0" maxOccurs="unbounded">
                <xs:element ref="field"/>
            </xs:choice>
            <xs:attribute name="id" type="xs:string" use = "required"/>
            <xs:attribute name="class" type="xs:string" use = "required"/>
            <xs:attribute name="createdAt" type="xs:string" use = "optional"/>
            <xs:attribute name="createdInThread" type="xs:string" use = "optional"/>
            <xs:attribute name="createdInMethod" type="xs:string"/>
               <xs:attribute name="tag" type="xs:string" use = "required"/>
        </xs:complexType>
    </xs:element>
```

```
    <!-- array element-->
    <xs:element name="array" type = "arrayTypeWithAttribute"/>
    <!-- stringLiteral element-->
    <xs:element name="stringLiteral">
        <xs:complexType>
            <xs:sequence>
                <xs:element minOccurs ="4" maxOccurs="4" ref="field "/>
            </xs:sequence>
            <xs:attribute name="id" type="xs:string" use = "required"/>
            <xs:attribute name="class" type="xs:string" use = "required"/>
        </xs:complexType>
    </xs:element>
```

```
    <!-- field element :  child of class, object and stringLiteral-->
    <xs:element name="field">
        <xs:complexType>
            <xs:attribute name="name" type="xs:string" use = "required"/>
            <xs:attribute name="id" type="xs:string" use = "optional"/>
            <xs:attribute name="value" type="xs:string" use = "optional"/>
            <xs:attribute name="type" type="xs:string" use = "optional"/>
        </xs:complexType>
    </xs:element>

    <xs:simpleType name = "arrayType">
        <xs:list itemType="xs:integer"/>
    </xs:simpleType>


    <!-- complex type "arrayTypeWithAttribute". type of array element-->
    <xs:complexType name = "arrayTypeWithAttribute">
        <xs:simpleContent>
            <xs:extension base="arrayType">
                <xs:attribute name="id" type="xs:string" use = "required"/>
                <xs:attribute name="class" type="xs:string" use = "required"/>
                <xs:attribute name="createdAt" type="xs:string" use = "optional"/>
                <xs:attribute name="createdInThread" type="xs:string" use =
 "optional"/>
                <xs:attribute name="createdInMethod" type="xs:string" use =
 "optional"/>
                <xs:attribute name="length" type="xs:string" use = "required"/>
                <xs:attribute name="elementsType" type="xs:string" use = "optional"/
>
                <xs:attribute name="type" type="xs:string" use = "optional"/>
            </xs:extension>
        </xs:simpleContent>
    </xs:complexType>

</xs:schema>
```

Table 15.2. XML schema for heap dumps

15.2.2 File Specification

Types referenced in heap dumps are represented in the internal classfile format (Figure 15.1). Fully qualified names are names separated by / separator (For example: `a/b/C`).

```
Type = <BaseType> | <ClassType> | <ArrayType>
BaseType: B(byte), C(char), D(double), F(float), I(int), J(long), S(short),
 Z(boolean),
ClassType: L<ClassName>;
ArrayType: [<Type>
```

Figure 15.1. Internal classfile format for types

Tags used in the heap dumps are described in the table below.

Tag	Attributes	Description
`heap`		The root element
`class`		Element that references a Java class
	`name`	Class type (`<ClassType>`).
	`id`	Unique identifier of the class
	`superclass`	Identifier of the superclass of this class
`object`		Element that references a Java object
	`id`	Unique identifier of this object
	`class`	Fully qualified name of the class of this object
`array`		Element that references a Java array
	`id`	Unique identifier of this array
	`class`	Fully qualified name of the class of this array
	`elementsType`	Type of the elements of this array
	`length`	Array length
`stringLiteral`		Element that references a `java.lang.String` literal
	`id`	Unique identifier of this object
	`class`	Id of `java.lang.String` class
`field`		Element that references the field of an object or a class
	`name`	Name of this field
	`id`	Object or Array identifier, if it holds a reference
	`type`	Type of this field, if it holds a base type
	`value`	Value of this field, if it holds a base type

Table 15.3. Tag descriptions

15.3 Configuration

The SimJPF has the following application-specific configuration variables:

Variable	Meaning
`SIMJPF_01`	Configures symbolic debugger (JDWP).
`SIMJPF_02`	Configures a Java heap dump when the `System.gc()` method is called.
`SIMJPF_03`	Configures SimJPF runtime to stick to both the JPF threads policy and memory sizes.
`SIMJPF_04`	Configures a slowing factor in order to provide the engineers a simulation which computation speed is similar to the one of the EmbJPF on their PC.
`SIMJPF_05`	Configures code coverage analysis.

Table 15.4. SimJPF specific configuration variables

16 Limitations

16.1 EmbJPF Limitations

Item		EVAL	DEV
Number of classes		2500	4000
Number of methods per class		1500	65000
Total number of methods		1500	unlimited
Class / Interface hierarchy depth		127 max	127 max
Number of monitors[a] per thread		8 max	8 max
Number of fields	Base type	65000	65000
	References	65000	65000
Number of statics	boolean + byte	limited	65000
	short + char	limited	65000
	int + float	limited	65000
	long + double	limited	65000
	References	limited	65000
Method size		65000	65000
Time limit		60 minutes	unlimited

[a]No more than n different monitors can be held by one thread at any time.

Table 16.1. EmbJPF Limitations

16.2 SimJPF Limitations

Item	EVAL version	DEV version
Number of calls	limited	unlimited
Time limit	60 minutes	unlimited

Table 16.2. SimJPF Limitations

17 Document History

Date	Revision	Description
June 4th 2012	A	First release
October 24th 2012	B	Introduction of Shielded Plug ([SP] & LL_SP API)
November 20th 2013	C	Global documentation review
June 20th 2014	D	Update for STM32Java 3.0.0