Running HPS Java

Before You Start

Before reading these instructions, you will want to read Installing HPS Java, which explains how to build or get an HPS Java distribution jar file.

Distribution Jars

The distribution jar file contains all of the project's dependencies in a distribution that can be run standalone using the *java* command. The distribution will have "-bin" in the name.

The jar file will be found in your copy of HPS Java after you have built the project locally. You can use a simple *ls* command to check that it was built correctly.

```
ls distribution/target/hps-distribution-*-bin.jar
```

Any references to *hps-distribution-bin.jar* in command line syntax within these instructions should be replaced by the complete path to the distribution jar which was built.

Running the Jar File

The distribution jar can be run in two basic ways. You may run the default *main* method using the *-jar* switch, or the *-cp* switch can be used along with the path to a class's *main* method.

Running the Job Manager

Using the *-jar* switch from the command line will run the JobManager which processes LCIO files using a steering file configuration.

```
java -jar ./distribution/target/hps-distribution-bin.jar [args]
```

Without any arguments, it will print the command line options and then exit.

Running a Specific Class's Main Method

You can also run the main method from any class in the jar, for example:

```
java -cp ./distribution/target/hps-distribution-bin.jar org.hps.evio.EvioToLcio [args]
```

If there is a "class not found" error, then check the following: the correctness of the path to the jar, the existence and proper definition of a main method within the Java class, and the read permissions on the jar file.

A class can only be accessed from the command line if it defines a valid main method.

```
package org.example;

public class MyClass {
    public static void main(String[] args) {
        System.out.println("hello main");
    }
}
```

To be directly accessible from the command line, a class must also have the following features.

- It must be marked as *public.*
- It must have a *public static* method called *main* (as shown above).
- The main method must have an array of strings as an input variable.

Assuming that *MyClass* was bundled inside the distribution (which it isn't), then it could be run from the command line as follows:

```
java -cp ./distribution/target/hps-distribution-bin.jar org.example.MyClass arg1 arg2 [...]
```

Most classes that implement a command line interface will print out a help message when run with no additional options e.g.

```
java -cp ./distribution/target/hps-distribution-bin.jar org.hps.evio.EvioToLcio
```

By convention, the *-h* switch is also usually used to print out a help menu.

```
java -cp ./distribution/target/hps-distribution-bin.jar org.hps.evio.EvioToLcio -h
```

The specific command line syntax is unfortunately not standardized across all tools in the project and will depend on what was implemented in that class by the particular author.

Running Tests

Regular unit tests can be run from the command-line using a syntax like the following:

```
mvn test -Dtest=[TestClassName]
```

Integration tests are run from the *integration-tests* directory with the syntax:

```
mvn verify -Dit.test=[TestClassName]
```

In both cases, the class should be provided without the package name.

More information about running integration tests can be found here.

Command Line Tools

Job Manager

The JobManager runs a series of Drivers defined in an lcsim xml file on input events from one or more LCIO files. The XML steering file may contain lists of input file tags, or the input files may(more typically) be supplied by command line arguments.

Job Manager Command Line Usage

```
[1026 $] java -jar ./hps-java-trunk/distribution/target/hps-distribution-3.6-SNAPSHOT-bin.jar
Feb 18, 2016 4:02:27 PM org.lcsim.job.JobControlManager printHelp
INFO: java org.lcsim.job.JobControlManager [options] steeringFile.xml
usage:
 -b,--batch               Run in batch mode in which plots will not be
                          shown.
 -D,--define <arg>        Define a variable with form [name]=[value]
 -d,--detector <arg>      user supplied detector name (careful!)
 -e,--event-print <arg>   Event print interval
 -h,--help                Print help and exit
 -i,--input-file <arg>    Add an LCIO input file to process
 -n,--nevents <arg>       Set the max number of events to process
 -p,--properties <arg>    Load a properties file containing variable
                          definitions
 -r,--resource            Use a steering resource rather than a file
 -R,--run <arg>           user supplied run number (careful!)
 -s,--skip <arg>          Set the number of events to skip
 -w,--rewrite <arg>       Rewrite the XML file with variables resolved
 -x,--dry-run             Perform a dry run which does not process events
```

Job Manager Command Line Options

Switch	Description	Notes
-b	Activates batch mode plotting so plots will not be shown when running job.	This switch can be used to suppress the display of interactive plots for batch usage.
-D	Add a variable definition that applies to the input steering file.	All variables in the XML input file must be resolved using this switch or a fatal exception will occur.
-d	Set the name of the detector model.	This should be a valid detector model defined in *detector-data/detectors* of HPS Java. Usually this does not need to be set by the user.
-e	Print out an informational message every N events.	When this is left out no event-by-event print outs will display during the job.
-h	Print help and exit.	Using this with other arguments will still cause the help message to print and job will exit afterwards.
-i	Add an LCIO input file.	This switch can be used multiple times to specify more than one input file.
-n	Maximum number of events to run in job.	To run through all events in the input file(s), do not set this switch.
-p	Load a properties file containing steering file variable definitions.	The input properties file should define *name: value pairs* for each variable that needs to be defined in the steering file.
-r	Treat the supplied steering as a classpath resource rather than a file.	These are typically resources from *org/hps/steering* in the steering-files module of HPS Java.
-R	Set the run number to be used when initializing the conditions system.	This has the side effect of "freezing" the conditions system, as the run numbers from the input events will be ignored.
-s	Skip N events at the beginning of the job.	This will not skip N events in each file but the first N events read by the job.
-w	Rewrite the XML steering file with the variables resolved.	This can be used as a simple template engine to generate steering files without variables in them.
-x	Execute in dry run mode which means actual job will not execute.	This switch is typically used to check for initialization errors in the job.

The steering file is a mandtaory argument, and it is supplied as an extra argument rather than as a command switch.

If a detector name and run number are both supplied as arguments from the command line, then the conditions system will be initialized and frozen, meaning that subsequent event numbers from data will be ignored.

Here is an example command using most of these switches together.

```
java -jar ./hps-java-trunk/distribution/target/hps-distribution-bin.jar -b -DoutputFile=output -d HPS-
EngRun2015-Nominal-v3 -e 100 -i input.slcio \
-n 1000 -p myvars.prop -r -R 5772 -s 10 -w myjob.xml -x /org/hps/steering/dummy.lcsim
```

The above command is actually nonsense and will not work! 🙂

EvioToLcio

The EvioToLcio tool converts EVIO files to LCIO format and optionally may run a steering file job on the in-memory events. This scheme allows the conversion and reconstruction to run within the same job for efficiency.

EvioToLcio Command Line Usage

```
[1037 $] java -cp ./distribution/target/hps-distribution-3.6-SNAPSHOT-bin.jar org.hps.evio.EvioToLcio
EvioToLcio [options] [evioFiles]
usage:
 -b         enable headless mode in which plots will not show
 -d <arg>   detector name (required)
 -D <arg>   define a steering file variable with format -Dname=value
 -f <arg>   text file containing a list of EVIO files
 -h         print help and exit
 -L <arg>   log level (INFO, FINE, etc.)
 -l <arg>   path of output LCIO file
 -m <arg>   set the max event buffer size
 -M         use memory mapping instead of sequential reading
 -n <arg>   maximum number of events to process in the job
 -r         interpret steering from -x argument as a resource instead of a
            file
 -R <arg>   fixed run number which will override run numbers of input
            files
 -t <arg>   specify a conditions tag to use
 -v         print EVIO XML for each event
 -x <arg>   LCSim steeering file for processing the LCIO events
```

EvioToLcio Command Line Options Table

Switch	Description	Notes
-b	Activates batch mode plotting so plots will not be shown when running job.	
-d	Set the name of the detector model.	This should be a valid detector model from the *detector-data/detectors* directory.
-D	Add a variable definition that applies to the input steering file.	
-f	Text file containing a list of input EVIO files, one per line.	
-L	Set the output logging level.	DEPRECATED Use logging config file or class instead.
-l	Path of the output LCIO file.	
-m	Set the max event buffer size.	Experts only.
-M	Use memory mapping in EVIO reader instead of sequential access.	Experts only.
-n	Maximum number of events to process in the job.	
-R	Set the run number to be used when initializing the conditions system.	
-t	Specify a conditions tag for filtering conditions records.	
-v	Print out EVIO converted to XML for every event.	For verbose debugging of events.
-r	Interpret steering file from -x as a classpath resource rather than a file.	
-x	Steering file	Could be resource or file depending on if -r switch is used.

Here is an example showing how to use most of these command line options.

```
java -jar ./hps-java-trunk/distribution/target/hps-distribution-bin.jar org.hps.evio.EvioToLcio -b -
DoutputFile=output -l lcio_file_output -m 50 \
-v -M -n 1000 -d HPS-EngRun2015-Nominal-v3 -R 5772 -t pass1 -r -x /org/hps/steering/dummy.lcsim file1.evio
file2.evio file3.evio [etc.]
```

Some of these arguments are similar to the job manager, but the steering file is supplied in a different way. Evio2Lcio also uses a command switch to specifiy the steering file, whereas the job manager expects this as an extra argument without a command switch.

Run Scripts

Run scripts that wrap a number of HPS Java command line utilities are generated when building the distribution. These can be used to easily run tools in the project without typing the full java command.

After the build completes, they should be found in this HPS Java directory.

```
distribution/target/appassembler/bin/
```

For instance, the EvioToLcio utility can be run using this script.

```
distribution/target/appassembler/bin/evio2lcio.sh [...]
```

These scripts have several advantages over running the *java* commands yourself.

- A full classpath is created in the script so it is not necessary to use the distribution jar.
- Reasonable JVM options are set such as the min heap size.
- Logging is configured automatically by a default logging properties file.
- You do not need to type all the Java boilerplate commands like "java -jar".
- You do not need to know the corresponding class's full name in order to run its command line utility.

Using symlinks to these scripts should work fine e.g.

```
ln -s distribution/target/appassembler/bin/evio2lcio.sh
./evio2lcio.sh
```

When using these scripts, you cannot directly supply Java system properties, so the JAVA_OPTS variable should be used instead.

```
export JAVA_OPTS="-Dorg.hps.conditions.enableSvtAlignmentConstants"
```

The full list of Java system properties to be used should be included in this variable. You should not set the options *-Xmx* or *-Djava.util.logging.config.class,* as these are set by each script.

Steering Files

Steering File Locations

The standard location for steering files is *steering-files/src/main/resources/org/hps/steering/* in HPS Java.

This folder is organized into the following sub-directories which contain sets of related steering files.

Directory	Description
analysis	includes analysis template
broken	broken steering files (which may be removed at any time!)
calibration	not really used currently
monitoring	monitoring application steering files
production	various production steering files including event filtering and DQM
readout	readout simulation drivers to be run on MC data from SLIC
recon	production reconstruction steering files

users	user files (organized into sub-directories by user name)

These steering files can be accessed using their path on disk, or they may be referenced as classpath resources. The exact syntax will depend on the command line tool.

For example, a steering file resource can be accessed like this using the job manager:

```
java -jar ./hps-java/distribution/target/hps-distribution-bin.jar -x /org/hps/steering/recon
/EngineeringRun2015FullRecon.lcsim [...]
```

The same steering file could also be accessed as a file from the local copy of HPS Java.

```
java -jar ./hps-java/distribution/target/hps-distribution-bin.jar hps-java-trunk/steering-files/src/main
/resources/org/hps/steering/recon/EngineeringRun2015FullRecon.lcsim [...]
```

You can use a file rather than a resource if you running a steering file which is not checked into HPS Java or you are using a modified steering file that has not been packaged into the distribution jar.

Steering File Variables

Suppose a steering file contains the following variable definition:

```
<driver name="MyDriver" type="org.example.MyDriver">
    <someNumber>${numVar}</someNumber>
</driver>
```

The variable would then need to be resolved with a command such as:

```
 java -cp ./distribution/target/hps-distribution-bin.jar org.hps.evio.EvioToLcio -DnumVar=1234 [...]
```

All variables defined in the XML steering files must be resolved from the command line or an error will occur and the job will terminate.

System Properties

The command line tools may be affected by some Java system properties.

Name	Description	Values
java.util.logging.config.file	Java logging config file	described in logging package doc
java.util.logging.config.class	Java logging config initialization class	described in logging package doc
hep.aida.IAnalysisFactory	AIDA backend factory class	• hep.aida.ref.AnalysisFactory - default • hep.aida.jfree.AnalysisFactory - JFree backend • hep.aida.ref.BatchAnalysisFactory - batch mode (no plot display)
org.hps.conditions. enableSvtAlignmentConstants	Enables application of SVT alignment constants from conditions db to the detector model	Actual value is not checked.
org.hps.conditions.connection.file	Properties file with connection settings for conditions database	should point to a valid connection.prop file
org.hps.conditions.connection. resource	Resource that points to properties file with connection settings for conditions database	• /org/hps/conditions/config/jlab_connection. prop - default connection

These values are set as system properties in Java itself rather than the command line tool.

```
java -Dorg.hps.conditions.enableSvtAlignmentConstants [...]
```

When set in this way, these values are accessible to the framework as Java system properties.

Java Arguments

The JVM accepts a number of command line arguments that alter its behavior.

In particular, when running *java* you will likely want to increase the default heap space, as the default is generally too low for running full reconstruction jobs.

This command will change the heap space to 2 gigabytes:

```
java -Xmx2g [...]
```

Also, you may want to run the JVM in server mode.

```
java -server [...]
```

The server JVM has been optimized for performance speed rather than responsiveness, and it should run faster than the default client JVM.

Logging and Debugging

Logging Config

HPS Java uses the built-in logging facilities of Java which are described in the logging package documentation.

Each package in HPS Java has a default level which is set in the following config file:

```
logging/src/main/resources/org/hps/logging/config/logging.properties
```

This config, or any other custom logging config, can be activated from the command line by setting the config file property to its path as a Java argument.

```
java -Djava.util.logging.config.file=logging/src/main/resources/org/hps/logging/config/logging.properties [...]
```

Alternately, you may also activate a Java class which will load this configuration.

```
java -Djava.util.logging.config.class=org.hps.logging.config.DefaultLoggingConfig [...]
```

Log Levels

These are the available log levels, in descending order.

Log Levels Table

Level	Description	Use
SEVERE	severe error message usually meaning program should halt	unrecoverable errors that halt the program
WARNING	warning message indicating a non-fatal error or problem	warning messages
INFO	informational messages	informational messages that should usually print when the program runs
CONFIG	configuration messages	printing out config information for a class or tool
FINE	debug print outs	high level debugging messages that should not typically be active
FINER	more verbose debug print outs	more verbose debugging messages
FINEST	most verbose debug messages	the most verbose debugging messages
ALL	print all error messages	when logger should always print all messages
OFF	disable all messages	when logger should be completely disabled from printing

Each logger inherits by default from the global log setting.

This is defined in the config file as follows.

```
# default global level
.level = WARNING
```

If a package's log level is not explicitly configured, then it will inherit the default *WARNING* log level from the global logger.

Defining Loggers

In the config file, loggers are defined and configured by their package rather than class.

```
# evio
org.hps.evio.level = CONFIG
```

In the above config, any logger within the *org.hps.evio* package will have a log level of *CONFIG*.

A class's logger is typically defined within the codebase using the following template.

```
package org.example;

import java.util.logging.Logger;

class MyClass {

    static private final Logger LOGGER = Logger.getLogger(MyClass.class.getPackage().getName());

    void someMethod() {
        LOGGER.info("some method was called");
    }
}
```

The handler and the level should *not* be assigned within the code, because changing the level would require recompilation. Instead, the level should be defined using the config file as described above.