# Getting Started with the DØ Framework

Dhiman Chakraborty, Herbert Greenlee, Jim Kowalkowski,
Qizhong Li, Gordon Watts

May 15, 1998

## Contents



This document assumes that the reader is familiar with basic framework concepts, such as hooks and packages. Refer to the Framework User Guide for additional information.

## 1 Making a package

Here are instructions for creating a package class:

1. Inherit from base class `Package`.

2. Inherit from one or more hook classes (e.g. `Process`).

3. Supply a constructor taking single argument of type `Context*`. Pass the context argument to any base class constructors.

4. Include a static method `const std::string version()`. Conventionally, the version method should return the cvs version string `"$Id:"`.

5. Include a static method `const std::string package_name()` that returns the name of the package.

6. Override method `std::string packageName()` of base class `Package`. This function should return the same string as the static method `package_name()`. Method `packageName()` can call `package_name()`, but not vice versa.

7. Supply the required overrides of the hook base classes (e.g. `Result Process::processEvent(Event&)`).

8. In an implementation file, define a global registration variable as follows:

   ```
   static Registry<MyPackage> dummy;
   ```

   where the template parameter is the package class.

## 1.1 Using gmake to compile a package in the SRT environment

The package implementation file containing the global registry variable should be compiled to an object file in the SRT lib area (use gmake variable `OBJCPPFILES`). Any other package implementation files can be placed into libraries as usual. Use gmake function `$(filter-out ...)` to remove the global registry file from `LIBCPPFILES`. For example:

```
# Top level package file
OBJCPPFILES := mypackage.cpp

# Other source files
LIBCPPFILES := $(filter-out $(OBJCPPFILES), $(wildcard *.cpp))

include SoftRelTools/standard.mk
```

# 2 Linking the framework executable

In order to be selectable at run time, packages must be linked into a framework executable (it may eventually be possible to use a prelinked framework executable and dynamically linked packages). Modules containing package global registry variables should be linked as object files, not from libraries. There is also an object file associated with the framework main program (`framework.o`) that must be linked in as an object file. Include the libraries required by the framework (`-lframework`, `-lidentifiers`, and `-lrcp`), as well as libraries required by the packages.

## 2.1 Using gmake to link the framework executable in the SRT environment

The instructions in this section are for how to write a linking rule in your own GNUmakefile, rather than using any of SRT's rules in standard.mk, or other make include files. Here is an example:

```
# Tell gmake where to find link objects
vpath %.o $(libdir) $(BFDIST)/releases/$(BFCURRENT)/lib/$(BFARCH)

# Object files
objects := framework.o mypackage.o ReadEvent.o LoadDSPACK.o

# Libraries needed by framework and packages
libs := -lio_packages -lframework -ledm -lrcp -lidentifiers -lstream_ds \
        -ld0om_ds -ldspack -lstream -ld0om -lcint-lite -ld0_util -lNameTrans \
        -lExceptions -lZMutility -lCLHEP -lftn -lm

# Rule to build executable
exe := $(bindir)myprogram
$(exe): $(objects)
        @echo "Linking $(exe)"
        -rm -f $(exe) 2> /dev/null
        $(CXX) -o $@ $(CXXFLAGS) $(CXXCFLAGS) $(CPPFLAGS) $(LDFLAGS) $^ $(libs)

# Standard SRT target
bin: $(exe)
```

Here are some comments on the above makefile fragment.

1. The `vpath` directive sets up a search path for gmake to search for the object files. Gmake invokes the linker with full pathnames of the object files. For this to work, the object files must be specified as dependents of the executable target.

2. The object file `LoadDSPACK.o` must be force-loaded to make the DSPACK I/O mechanism available through DØOM.

3. The above libraries are typical for an edm/DØOM program.

4. The executable should be specified as a dependent of one of the standard SRT targets (`bin` in the above example). Reasonable targets to use are `bin` and `tbin`.

5. The executable can be built in any directory (`$(bindir)` in the above example). Reasonable places to build executables are `$(bindir)` and `$(workdir)`. If no directory is specified, the executable will be built in the same directory as the GNUmakefile (that is usually not a good idea).

# 3 Running the framework executable

The framework executable must be invoked with command line parameters to specify the top level framework rcp file. There are two ways of doing this:

```
myprogram myrcp
```

or

```
myprogram <name> -rcp myrcp.rcp
```

In either case, the framework looks for a top level rcp file called myrcp.rcp.

## 3.1 The framework rcp file

Here is an example of a top level framework RCP file.

```
string ProcessorPackage = "ProcessGroup"
string ProcessorHooks = "builder filter process dump"

// List of packages

string Packages = "p1 p2"

// RCP files for the packages.

RCP p1 = <ReadEvent.rcp>
RCP p2 = <mypackage.rcp>

// only call dump event hooks every this many events

int DumpPeriod = 1
```

The first two lines of the framework rcp file (the `ProcessorPackage` and `ProcessorHooks` variables do not normally need to be changed. The variable `Packages` specifies a list of packages to be instantiated. The names specified in the `Packages` variable are instance names, which are arbitrary except that they must match the names of the RCP objects immediately following. Packages specified in the `Packages` variable are instantiated using the corresponding RCP files (the packages must have been linked into the executable).

Currently all required RCP files (the framework RCP file, and the package RCP files) should be copied to the directory from which the framework executable is run. Eventually, an RCP search path may be defined.

### 3.2 Package rcp files

Here is the RCP file from the `ReadEvent` package.

```
string PackageName = "ReadEvent"
string InputFile = "test.ds"
string InputFormat = "DSPACK"
int NumEvents = 0              // Maximum number of events to read (0 = all).
string ApplicationName = "ReadEvent"  // String passed to d0om_init.
```

The main thing to note about this RCP file is that it contains a string variable called `PackageName`. All package RCP files must contain this variable. The `PackageName` variable is how the framework finds out what package is to be instantiated. The string specified as the value of `PackageName` must match the string returned by the package static method `package_name()`.

# 4 Available Hooks

Here is a list of the currently available hooks.

1. `Generator` – Generate (read, create) an event.
2. `Destroyer` – Destroy an event.
3. `Process` – Process an event.
4. `Filter` – Say whether further processing should be done for this event.
5. `Builder` – Add data to the event (runs before `Process` hook).
6. `JobSummary` – Runs at end of job.
7. `RunInit` – Runs at beginning of each run.
8. `RunEnd` – Runs at end of each run.
9. `Group` – Used for nesting packages.

# 5 Standard Packages

Here is a list of currently available standard packages.

1. `ReadEvent` – Read an edm event.
2. `NewEvent` – Generate an empty edm event.
3. `WriteEvent` – Write an edm event.
4. `DumpEvent` – Dump an edm event.

# 6 Hints and advice

1. Framework code lives in SRT package `framework`.
2. Standard packages live in SRT package `io_packages`.
3. The framework classes and standard packages live in namespace `fwk`.
4. If any required package components are missing, except for the global registry variable, you will get a compilation or linking error.
5. If the global registry variable is missing, your package will compile, but you will get a run time error from the framework executable.
6. When writing a package, make sure that the global registry variable is defined exactly once. Put the global registry varialbe in an implementation (`.cpp`) file, not in a header file.
7. It does not matter which implementation file contains the global registry variable, as it is never accessed. The only thing that matters is that it is defined (so that its constructor will run), and that it gets linked into the executable.
8. Avoid unnecessary template instantiations in the file containing the global registry variable (KCC might try to recompile the package object file at an inopportune moment). It is perfectly acceptable to have the global registry variable in a file all by itself.
9. The global registry variable should have local scope (define it to be `static` or put it in the unnamed namespace).
10. Compile the file containing the global registry variable to an object file, but do not put this file into a library.
11. If the global registry variable gets defined multiple times (for example, because it accidentally got into a library), the framework program will get a run time error.
12. When linking the framework executable, you must include at least one package that implements the `Generator` hook (among the standard packages `ReadEvent` and `NewEvent`), and you must select one such package in the framework rcp file. If you forget to do this, any event processing hooks will never be called.
13. A package class does not need access to the context variable. Treat the context pointer argument of the package constructor as an opaque pointer.
14. A package class should access its defining RCP object using method `packageRCP()` of base class `Package`.

15. Do not send ascii output directly to `cout`. Instead, use streams returned by `Package` class methods `out()` and `log()`. This will give the framework user a chance to redirect output at run time via framework rcp parameters.

16. Use the `RunInit` hook for run dependent initialization. The `RunInit` hook is guaranteed to be called, regardless of whether the input data contains begin run records.