

---

# **Integrated Plasma Simulator (IPS) Documentation**

***Release 2.1***

**SWIM Project**

June 22, 2011



# CONTENTS

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Where to Start? . . . . .	3
1.2	Acknowledgments . . . . .	4
1.3	Additional Information . . . . .	4
<b>2</b>	<b>Publications</b>	<b>5</b>
<b>3</b>	<b>Getting Started</b>	<b>7</b>
3.1	Obtaining, Dependencies, Platforms . . . . .	7
3.2	Building and Setting up Your Environment . . . . .	8
3.3	Running Your First IPS Simulations . . . . .	10
<b>4</b>	<b>User Guides</b>	<b>17</b>
4.1	Reference Guide for Running IPS Simulations . . . . .	18
4.2	Developing Drivers and Components for IPS Simulations . . . . .	28
4.3	The Configuration File - Explained . . . . .	41
4.4	Platforms and Platform Configuration . . . . .	47
4.5	Plasma State Guide . . . . .	56
4.6	Fundamentals of the Advanced Features of the IPS . . . . .	56
4.7	Performance Analysis . . . . .	56
4.8	Examples Listing . . . . .	56
<b>5</b>	<b>Component Guides</b>	<b>57</b>
<b>6</b>	<b>Developer Guides</b>	<b>59</b>
6.1	IPS Design Documentation . . . . .	59
6.2	The IPS Framework and Managers . . . . .	59
6.3	Plasma State for Developers . . . . .	59
6.4	Guide to Porting the IPS . . . . .	59
<b>7</b>	<b>Portal Guides</b>	<b>61</b>
<b>8</b>	<b>Code Listings</b>	<b>63</b>
8.1	IPS . . . . .	63
8.2	Framework . . . . .	64
8.3	Data Manager . . . . .	65
8.4	Task Manager . . . . .	65
8.5	Resource Manager . . . . .	67
8.6	Component . . . . .	71
8.7	Configuration Manager . . . . .	71

8.8	Services . . . . .	72
8.9	Other Utilities . . . . .	77
8.10	Framework Components . . . . .	79
<b>9</b>	<b>Indexes and tables</b>	<b>81</b>
	<b>Python Module Index</b>	<b>83</b>
	<b>Index</b>	<b>85</b>

Contents:



# INTRODUCTION

Welcome to the documentation for the Integrated Plasma Simulator (IPS). The documents contained here will provide information regarding obtaining, using and developing the IPS and some associated tools.

The IPS was originally developed for the [SWIM](#) project and is designed for coupling plasma physics codes to simulate the interactions of various heating methods on plasmas in a tokamak. The physics goal of the project is to better understand how the heating changes the properties of the plasma and how these heating methods can be used to improve the stability of plasmas for fusion energy production.

The IPS framework is thus designed to couple standalone codes flexibly and easily using python wrappers and file-based data coupling. These activities are not inherently plasma physics related and the IPS framework can be considered a general code coupling framework. The framework provides services to manage:

- the orchestration of the simulation through component invocation, task launch and asynchronous event notification mechanisms,
- configuration of complex simulations using familiar syntax,
- file communication mechanisms for shared and internal (to a component) data, as well as checkpoint and restart capabilities,

The framework performs the task, configuration, file and resource management, along with the event service, to provide these features.

## 1.1 Where to Start?

For those who have never run the IPS before, you should start with [Getting Started](#). It starts from the beginning with how to obtain the IPS code, build and run some sample simulations on two different platforms.

The [User Guides](#) section has documents on basic and advanced user topics. For those who have used the IPS before or have done the tutorial and are ready to create their own run, the [Reference Guide for Running IPS Simulations](#) document walks you through the process of using the IPS to examine a computational or physics problem, with practical hints on what to consider through out the preparation, running and analysis/debugging processes. Additional documentation for basic simulation construction include [The Configuration File - Explained](#) and [Using the Plasma State](#). [The IPS for Driver and Component Developers](#) provides component developers with basic information on the construction of a component and integrating it into the IPS, guidance on how to construct drivers and IPS services API reference. Additional documents on advanced topics such as multiple levels of parallelism, computational considerations, fault tolerance and performance analysis are located in the [User Guides](#) chapter.

Developers of the IPS framework and services, or brave souls who wish to understand how these pieces work, should look at the [Developer Guides](#) and code listings. The code listings here will include internal and external APIs. The developer guides include information about the design of the IPS at a high level and the framework and managers at a lower level to acquaint developers with the structures and mechanisms that are used in the IPS framework source code.

Component developers should also contribute and update the documentation associated with each component in the *Component Guides* section.

Documentation related to using and developing for the web portal is contained in *Portal Guides*. Basic usage of the portal is covered in the appropriate documents in the User Guides section.

## 1.2 Acknowledgments

This documentation has been primarily written or adapted from other sources by [Samantha Foley](#), as part of the *SWIM* team. Don Batchelor provided examples and documentation that provided the basis for the *Getting Started* and *Basic IPS Usage* sections. Wael Elwasif provided much of the code documentation and initial documents on the directory structure and build process.

## 1.3 Additional Information

For more information on the design and implementation of the IPS and physics results generated using the IPS, see *Publications*.



# PUBLICATIONS

## Coming Soon

A listing of publications about the IPS or research presented that used the IPS.



# GETTING STARTED

This document will guide you through the process of running an IPS simulation and describe the overall structure of the IPS. It is designed to help you build and run your first IPS simulation. It will serve as a tutorial on how to get, build, and run your first IPS simulation, but not serve as a general reference for constructing and running IPS simulations. See the [Basic User Guides](#) for a handy reference on running and constructing simulations in general, and for more in-depth explanations of how and why the IPS works.

## 3.1 Obtaining, Dependencies, Platforms

The IPS code is currently located in the SWIM project's Subversion (SVN) repository. In order to checkout a copy, you must have SVN installed on the machine you will be using and be given permission to check out the IPS from the SWIM project. (SVN is installed on the machines used in the examples below.) See the [SWIM](#) project's website for details and instructions on how to sign up.

Once you have permission to access the repository, replace <csxim\_user> with your user name and check out the IPS trunk thusly:

```
svn co https://<csxim_user>@csxim.org/svn/csxim/ips/trunk ips
```

### 3.1.1 Dependencies

#### IPS Proper

The IPS framework is written in [Python](#), and requires Python 2.5+ <sup>1</sup>. There are a few other packages that may be needed for certain components or utilities. For Python packages listed below, we recommend using [easy\\_install](#) (it is what the developers use and like). The framework does use the Python package [ConfigObj](#), however the source is already included and no package installation is necessary (likewise for Python 2.5 and the processing module).

#### Portal

The portal is a web interface for monitoring IPS runs and requires only a connection to the internet and a web browser. Advanced features on the portal require an OpenID account backed by ORNL's XCAMS. Information on getting an XCAMS backed OpenID can be found on the [SWIM](#) website. There are also visualization utilities that can be accessed that require [Elvis](#) or [PCMF](#) (see below).

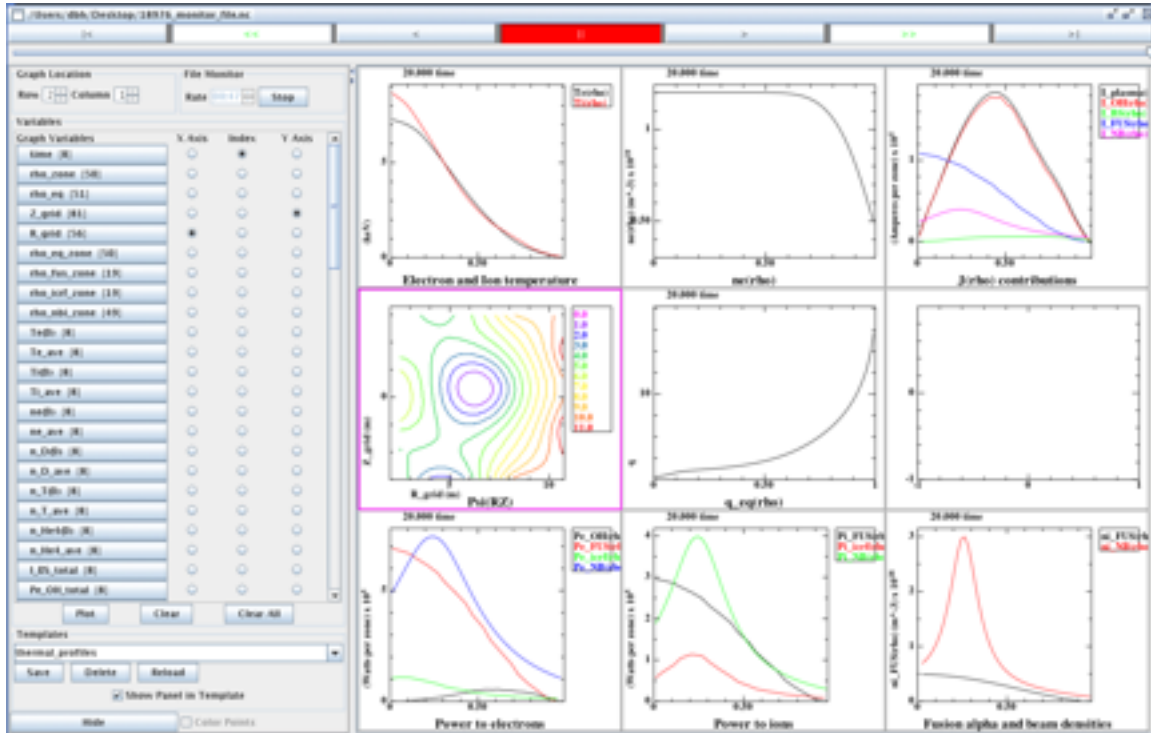
---

<sup>1</sup> For Python 2.5, an external package - [processing](#) - is used by the framework, however it was incorporated into Python 2.6 and higher as the [multiprocessing](#) module. Each package allows Python to spawn, manage and communicate between multiple processes, a key capability that allows the IPS to achieve multiple levels of parallelism.

## Other Utilities

**PCMF** This utility generates plots from monitor component files for visual analysis of runs. It can be run locally on your machine and generates plots like this one of the thermal profiles of an ITER run:

Requires: [Matplotlib](#) (which requires [Numpy/Scipy](#))



**Resource Usage Simulator (RUS)** This is a utility for simulation the execution of tasks in the IPS for research purposes.

Requires: [Matplotlib](#) (which requires [Numpy/Scipy](#))

**Documentation** The documentation you are reading now was created by a Python-based tool called Sphinx.

Requires: [Sphinx](#) (which requires [docutils](#))

**\*Plus\*** anything that the components or underlying codes that you are using need (e.g., MPI, math libraries, compilers). For the example in this tutorial, all packages that are needed are already available on the target machines and the shell configuration script sets up your environment to use them.

## 3.2 Building and Setting up Your Environment

The IPS has been ported to and is in regular use on the following platforms. See [Platforms and Platform Configuration](#) for more information:

- [Franklin](#) - Cray XT4
- [Hopper](#) - Cray XE6
- [Stix](#) - SMP
- [Pacman](#) - Linux Cluster

It is not hard to port the IPS to another platform, but it may not be trivial. *Please contact the framework developers before attempting it on your own.* For this tutorial, we will be using Franklin and Stix as our example platforms. It is **strongly** recommended that you use one of these two platforms as the example scripts and data files are located on these machines.

To build the IPS on one of the locations it has been ported to:

1. Checkout or copy the ips trunk to the machine of your choice:

```
head_node: ~ > svn co https://cswim_user@cswim.org/svn/cswim/ips/trunk ips
```

2. Configure your shell (assumes you are using bash). If you use a different shell, type `bash` at the command line before sourcing the `swim.bashrc.*` file:

```
head_node: ~/ips > cd ips
head_node: ~/ips > source swim.bashrc.<machine_name>
```

3. Setup the make configuration:

```
head_node: ~/ips > cp config/makeconfig.<machine_name> config/makeconfig.local
```

4. Build the IPS:

```
head_node: ~/ips > make
head_node: ~/ips > make install
```

Now you are ready to set up your configuration files, and run simulations.

### 3.2.1 IPS Directory Structure

Before running your first simulation, we should go over the contents of these selected `ips` subdirectories.

*ips/*

*bin/*

Transient. Installation directory for all executable objects (binaries, scripts) which are generally expected to be invoked by users. Also expected installation location for executables from external packages which IPS needs to operate.

*components/*

*class1/*

*class2/*

...

Subversion. Each class of component wrapper gets its own directory tree. Underneath each class may be multiple implementations targeting specific packages. Various component wrappers of a given class will share some source code, and require some individual code.

*doc/*

Subversion. Documentation. Hierarchy is not specifically designed, but would generally be expected to relate to the various components and packages involved in IPS.

*framework/*

Subversion. Framework source code and utilities reside here. Generally used by framework developers. Relevant Python scripts are placed in `ips/bin/` during make install for execution.

### Explanation and Rationale

The IPS directory hierarchy is designed to provide a (mostly) self-contained work space for IPS developers and users. Multiple instances of the IPS tree (with different names, of course), can coexist in the same parent directory without interference.

The caveat “mostly”, above, arises from the fact that not all required packages will be under version control by the SWIM project. The expectation is that such packages will be built separately, but installed into directories within the ips/ tree, and that ips/bin, ips/lib, etc. will be the only directories users will have to add to their paths to use their IPS installation.

Subdirectories in the tree are either transient or under Subversion control. Transient directories are created and populated as part of the installation process of either IPS code or external code. They should never appear within the Subversion repository. In fact, the Subversion repository is configured to ignore directories marked below as transient.

## 3.3 Running Your First IPS Simulations

This section will take you step-by-step through running a “hello world” example and a “model physics” example. These examples contain all of the configuration, batch script, component, executables and input files needed to run them. To run IPS simulations in general, these will need to be borrowed, modified or created. See the [Basic User Guides](#) for more information.

Before getting started, you will want to make sure you have a copy of the ips checked out and built on either Franklin or Stix.

On **Franklin** you will want to work in your \$SCRATCH directory and move to having the output from more important runs placed in the /project/projectdirs/m876/\* directory.

On **Stix** you will want to work in a directory within /p/swim1/ that you own. You can keep important runs there or in /p/swim1/data/.

### 3.3.1 Hello World Example

This example simply uses the IPS to print “Hello World,” using a single driver component and worker component. The driver component (hello\_driver.py) invokes the worker component (hello\_worker.py) that then prints a message. The implementations of these components reside in ips/components/drivers/hello/, if you would like to examine them. In this example, the *call()* and *launch\_task()* interfaces are demonstrated. In this tutorial, we are focusing on running simulations and will cover the internals of components and constructing simulation scenarios in the various User Guides (see [Index](#)).

1. Copy the following files to your working directory:

- Configuration file:

```
/ips/doc/examples/hello_world.config
```

- Batch script:

```
/ips/doc/examples/<machine>/sample_batchscript.<machine>
```

2. Edit the configuration file:

- Set the location of your web-enabled directory for the portal to watch and for you to access your data via the portal. If you do not have a web-enabled directory, you will have to create one using the following convention: on Franklin: /project/projectdirs/m876/www/<username>; on Stix: /p/swim/w3\_html/<username>.

Franklin:

```
USER_W3_DIR = /project/projectdirs/m876/www/<username>
USER_W3_BASEURL = http://portal.nersc.gov/project/m876/<username>
```

Stix:

```
USER_W3_DIR = /p/swim/w3_html/<username>
USER_W3_BASEURL = http://w2.pppl.gov/swim/<username>
```

This step allows the framework to talk to the portal, and for the portal to access the data generated by this run.

- Edit the *IPS\_ROOT* to be the absolute path to the IPS checkout that you built. This tells the framework where the IPS scripts are:

```
IPS_ROOT = /path/to/ips
```

- Edit the *SIM\_ROOT* to be the absolute path to the output tree that will be generated by this simulation. Within that tree, there will be work directories for each of the components to execute for each time step, along with other logging files. For this example you will likely want to place the *SIM\_ROOT* as the directory where you are launching your simulations from, and name it using the *SIM\_NAME*:

```
SIM_ROOT = /current/path/${SIM_NAME}
```

- Edit the *USER* entry that is used by the portal, identifying you as the owner of the run:

```
USER = <username>
```

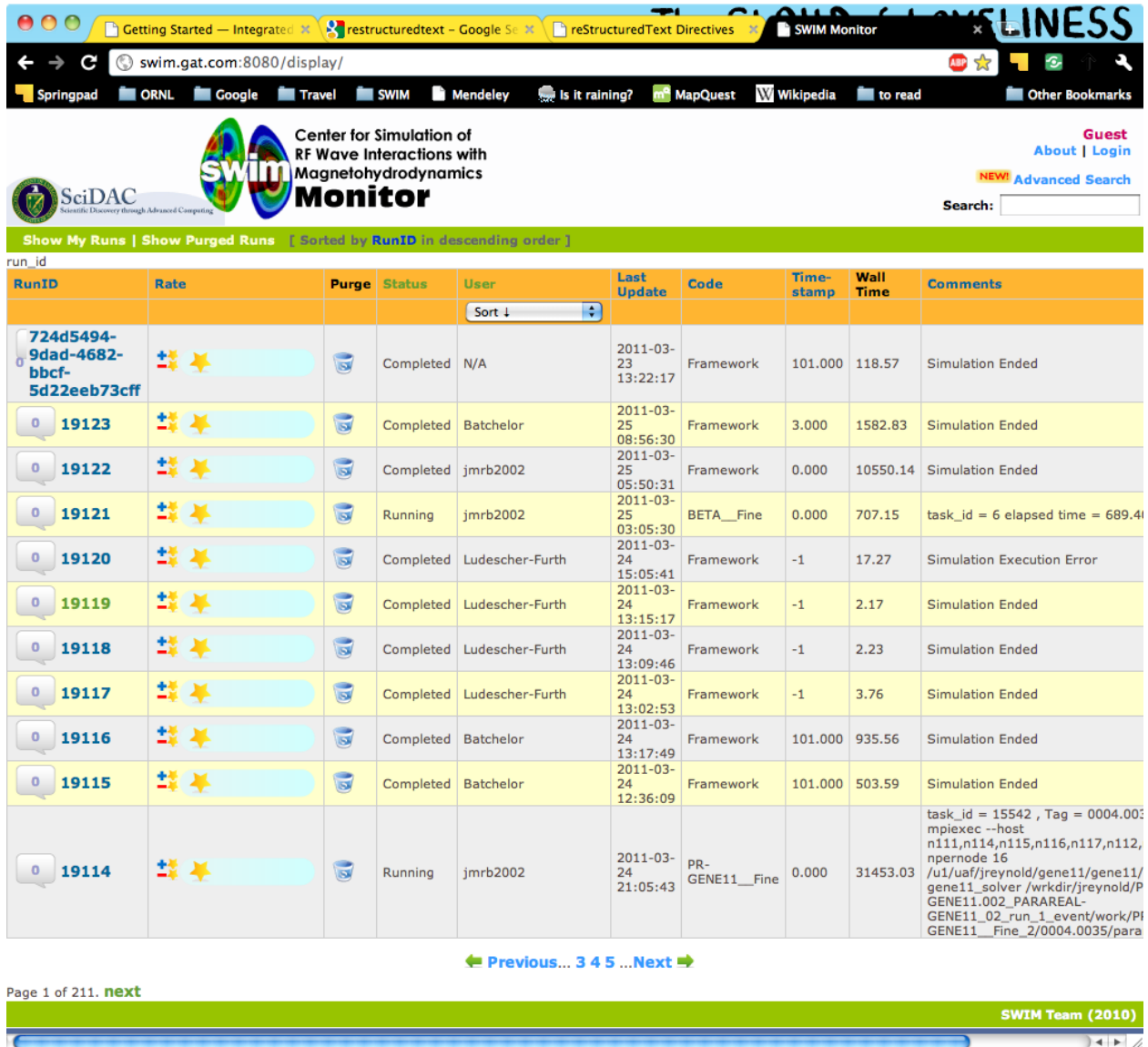
3. Edit the batch script such that *IPS\_ROOT* is set to the location of your IPS checkout:

```
IPS_ROOT=/path/to/ips
```

4. Launch batch script:

```
head_node: ~ > qsub hello_batchscript.<machine>
```

Once your job is running, you can monitor it on the [portal](#).



The screenshot shows the SWIM Monitor web interface. At the top, there's a navigation bar with links like 'Getting Started', 'reStructuredText', and 'SWIM Monitor'. Below this is a header section with the SciDAC logo and the text 'Center for Simulation of RF Wave Interactions with Magnetohydrodynamics'. A search bar is on the right. The main content area is titled 'Show My Runs | Show Purged Runs [ Sorted by RunID in descending order ]'. It contains a table with columns: RunID, Rate, Purge, Status, User, Last Update, Code, Time-stamp, Wall Time, and Comments. The table lists several simulation runs, including RunID 19123 through 19114. The status of these runs varies from 'Completed' to 'Running'. The bottom of the page shows 'Page 1 of 211. next' and a footer with 'SWIM Team (2010)'.

RunID	Rate	Purge	Status	User	Last Update	Code	Time-stamp	Wall Time	Comments
724d5494-9dad-4682-bbcf-5d22eeb73cff	++ ★		Completed	N/A	2011-03-23 13:22:17	Framework	101.000	118.57	Simulation Ended
0 19123	++ ★		Completed	Batchelor	2011-03-25 08:56:30	Framework	3.000	1582.83	Simulation Ended
0 19122	++ ★		Completed	jmr2002	2011-03-25 05:50:31	Framework	0.000	10550.14	Simulation Ended
0 19121	++ ★		Running	jmr2002	2011-03-25 03:05:30	BETA_Fine	0.000	707.15	task_id = 6 elapsed time = 689.4
0 19120	++ ★		Completed	Ludescher-Furth	2011-03-24 15:05:41	Framework	-1	17.27	Simulation Execution Error
0 19119	++ ★		Completed	Ludescher-Furth	2011-03-24 13:15:17	Framework	-1	2.17	Simulation Ended
0 19118	++ ★		Completed	Ludescher-Furth	2011-03-24 13:09:46	Framework	-1	2.23	Simulation Ended
0 19117	++ ★		Completed	Ludescher-Furth	2011-03-24 13:02:53	Framework	-1	3.76	Simulation Ended
0 19116	++ ★		Completed	Batchelor	2011-03-24 13:17:49	Framework	101.000	935.56	Simulation Ended
0 19115	++ ★		Completed	Batchelor	2011-03-24 12:36:09	Framework	101.000	503.59	Simulation Ended
0 19114	++ ★		Running	jmr2002	2011-03-24 21:05:43	PR-GENE11_Fine	0.000	31453.03	task_id = 15542 , Tag = 0004.00: mplexec --host n111,n114,n115,n116,n117,n112, npernode 16 /u1/uaf/jreynold/gene11/gene11/gene11_solver /wrkdir/jreynold/P GENE11.002_PARAREAL-GENE11_02_run_1_event/work/PI GENE11_Fine_2/0004.0035/para

Page 1 of 211. [next](#)

SWIM Team (2010)

When the simulation has finished, the output file should contain:

```
Starting IPS
Created <class 'hello_driver.HelloDriver'>
Created <class 'hello_worker.HelloWorker'>
HelloDriver: beginning step call
Hello from HelloWorker
HelloDriver: finished worker call
```

### 3.3.2 Model Physics Example

This simulation is intended to look almost like a real simulation, short of requiring actual physics codes and input data. Instead typical simulation-like data is generated from simple analytic (physics-less) models for most of the plasma state quantities that are followed by the *monitor* component. This “model” simulation includes time stepping, time varying scalars and profiles, and checkpoint/restart.

The following components are used in this simulation:



- `minimal_state_init.py`: simulation initialization for this model case
- `generic_driver.py`: general driver for many different simulations
- `model_epa_ps_file_init.py`: model equilibrium and profile advance component that feeds back data from a file in lieu of computation
- `model_RF_IC_2_mcmd.py`: model ion cyclotron heating
- `model_NB_2_mcmd.py`: model neutral beam heating
- `model_FUS_2_mcmd.py`: model fusion heating and reaction products
- `monitor_comp.py`: real monitor component used by many simulations that helps with processing of data and visualizations that are produced after a run

First, we will run the simulation from time 0 to 20 with checkpointing turned on, and then restart it from a checkpoint taken at time 12.

1. Copy the following files to your working directory:

- Configuration files:

```
/ips/doc/examples/seq_model_sim.config
/ips/doc/examples/restart_12_sec.config
```

- Batch scripts:

```
/ips/doc/examples/model_sim_bs.<machine>
/ips/doc/examples/restart_bs.<machine>
```

2. Edit the configuration files (you will need to do this in BOTH files, unless otherwise noted):

- Set the location of your web-enabled directory for the portal to watch and for you to access your data via the portal.

Franklin:

```
USER_W3_DIR = /project/projectdirs/m876/www/<username>
USER_W3_BASEURL = http://portal.nersc.gov/project/m876/<username>
```

Stix:

```
USER_W3_DIR = /p/swim/w3_html/<username>
USER_W3_BASEURL = http://w2.pppl.gov/swim/<username>
```

This step allows the framework to talk to the portal, and for the portal to access the data generated by this run.

- Edit the *IPS\_ROOT* to be the absolute path to the IPS checkout that you built. This tells the framework where the IPS scripts are:

```
IPS_ROOT = /path/to/ips
```

- Edit the *SIM\_ROOT* to be the absolute path to the output tree that will be generated by this simulation. Within that tree, there will be work directories for each of the components to execute for each time step, along with other logging files. For this example you will likely want to place the *SIM\_ROOT* as the directory where you are launching your simulations from, and name it using the *SIM\_NAME*:

```
SIM_ROOT = /current/path/${SIM_NAME}
```

- Edit the *RESTART\_ROOT* in `restart_12_sec.config` to be the *SIM\_ROOT* of `seq_model_sim.config`.

- Edit the *USER* entry that is used by the portal, identifying you as the owner of the run:

```
USER = <username>
```

3. Edit the batch script such that *IPS\_ROOT* is set to the location of your IPS checkout:

```
IPS_ROOT=/path/to/ips
```

4. Launch batch script for the original simulation:

```
head_node: ~ > qsub model_sim_bs.<machine>
```

Once your job is running, you can monitor is on the portal and it should look like this:

The screenshot shows the SWIM Monitor portal interface. At the top, there's a header with the SciDAC logo and the text "Center for Simulation of RF Wave Interactions with Magnetohydrodynamics". Below this, the "Portal Run ID: 19020" is displayed, along with the user "ssfoley". A table of simulation events is shown, with columns for Time, Seq Num, Event Type, Code, State, Wall Time, Phys Time-stamp, and Comment. The events are listed in chronological order, showing the progression of the simulation from 11:21:02 to 11:21:03.

Time	Seq Num	Event Type	Code	State	Wall Time	Phys Time-stamp	Comment
2011-03-18 11:21:03	381	IPS_END	Framework	Completed	154.94	20.000	Simulation Ended
2011-03-18 11:21:03	380	IPS_CALL_END	drivers_dbb_generic_driver	Running	154.92	20.000	Target = monitor@3:finalize(20.000)
2011-03-18 11:21:03	379	IPS_CALL_BEGIN	drivers_dbb_generic_driver	Running	154.88	20.000	Target = monitor@3:finalize(20.000)
2011-03-18 11:21:02	378	IPS_CALL_END	drivers_dbb_generic_driver	Running	154.84	20.000	Target = model_EPA@4:finalize(20.000)
2011-03-18 11:21:02	377	IPS_CALL_BEGIN	drivers_dbb_generic_driver	Running	154.81	20.000	Target = model_EPA@4:finalize(20.000)
2011-03-18 11:21:02	376	IPS_CALL_END	drivers_dbb_generic_driver	Running	154.77	20.000	Target = model_FUS@7:finalize(20.000)
2011-03-18 11:21:02	375	IPS_CALL_BEGIN	drivers_dbb_generic_driver	Running	154.74	20.000	Target = model_FUS@7:finalize(20.000)
2011-03-18 11:21:02	374	IPS_CALL_END	drivers_dbb_generic_driver	Running	154.70	20.000	Target = model_NB@6:finalize(20.000)
2011-03-18 11:21:02	373	IPS_CALL_BEGIN	drivers_dbb_generic_driver	Running	154.66	20.000	Target = model_NB@6:finalize(20.000)
2011-03-18 11:21:02	372	IPS_CALL_END	drivers_dbb_generic_driver	Running	154.63	20.000	Target = model_RF_IC_2@5:finalize(20.000)
2011-03-18 11:21:02	371	IPS_CALL_BEGIN	drivers_dbb_generic_driver	Running	154.59	20.000	Target = model_RF_IC_2@5:finalize(20.000)
2011-03-18 11:21:02	370	IPS_CHECKPOINT_END	drivers_dbb_generic_driver	Running	154.56	20.000	Components = [Model_seq_1@minimal_state_init@1, Model_seq_1@model_EPA@4, Model_seq_1@model_RF_IC_2@5, Model_seq_1@model_NB@6, Model_seq_1@model_FUS@7, Model_seq_1@monitor@3]
2011-03-18 11:21:02	369	IPS_CALL_END	drivers_dbb_generic_driver	Running	154.53	20.000	Target = monitor@3:checkpoint(20.000)

When the simulation has finished, you can run the restart version to restart the simulation from time 12:

```
head_node: ~ > qsub restart_bs.<machine>
```

The job on the portal should look like this when it is done:

Getting Started — Integrated restructuredtext — Google reStructuredText Directives SWIM Monitor

swim.gat.com:8080/detail/?id=19023

Springpad ORNL Google Travel SWIM Mendeley Is it raining? MapQuest Wikipedia to read to share Other Bookmarks

SciDAC Scientific Discovery through Advanced Computing

Center for Simulation of RF Wave Interactions with Magnetohydrodynamics

**swim Monitor**

Guest About | Login

NEW! Advanced Search

Search:

Home

Portal Run ID: **19023** Batchelor

Run Comment: restart at 12 sec equential model simulation using generic driver.py

Tokamak: ITER

Shot No: 1

Sim Name: Model\_seq\_1

Sim Runid: Model\_seq

Last Updated: 2011-03-18 13:53:46

Host: franklin

Output Prefix: N/A

Tag: sequential\_model

Logfile: N/A

Visualization: N/A

Add a new comment for 19023

Time	Seq Num	Event Type	Code	State	Wall Time	Phys Time-stamp	Comment
2011-03-18 13:53:46	304	IPS_END	Framework	Completed	116.65	20.000	Simulation Ended
2011-03-18 13:53:46	303	IPS_CALL_END	drivers_dbb_generic_driver	Running	116.62	20.000	Target = monitor@3:finalize(20.000)
2011-03-18 13:53:46	302	IPS_CALL_BEGIN	drivers_dbb_generic_driver	Running	116.59	20.000	Target = monitor@3:finalize(20.000)
2011-03-18 13:53:46	301	IPS_CALL_END	drivers_dbb_generic_driver	Running	116.55	20.000	Target = model_EPA@4:finalize(20.000)
2011-03-18 13:53:46	300	IPS_CALL_BEGIN	drivers_dbb_generic_driver	Running	116.51	20.000	Target = model_EPA@4:finalize(20.000)
2011-03-18 13:53:46	299	IPS_CALL_END	drivers_dbb_generic_driver	Running	116.48	20.000	Target = model_FUS@7:finalize(20.000)
2011-03-18 13:53:46	298	IPS_CALL_BEGIN	drivers_dbb_generic_driver	Running	116.44	20.000	Target = model_FUS@7:finalize(20.000)
2011-03-18 13:53:45	297	IPS_CALL_END	drivers_dbb_generic_driver	Running	116.41	20.000	Target = model_NB@6:finalize(20.000)
2011-03-18 13:53:45	296	IPS_CALL_BEGIN	drivers_dbb_generic_driver	Running	116.37	20.000	Target = model_NB@6:finalize(20.000)
2011-03-18 13:53:45	295	IPS_CALL_END	drivers_dbb_generic_driver	Running	116.33	20.000	Target = model_RF_IC_2@5:finalize(20.000)
2011-03-18 13:53:45	294	IPS_CALL_BEGIN	drivers_dbb_generic_driver	Running	116.30	20.000	Target = model_RF_IC_2@5:finalize(20.000)
2011-03-18 13:53:45	293	IPS_CHECKPOINT_END	drivers_dbb_generic_driver	Running	116.26	20.000	Components = [Model_seq_1@minimal_state_init@1, Model_seq_1@model_EPA@4, Model_seq_1@model_RF_IC_2@5, Model_seq_1@model_NB@6, Model_seq_1@model_FUS@7, Model_seq_1@monitor@3] Target = monitor@3:checkpoint(20.000)
2011-03-18 13:53:45	292	IPS_CALL_END	drivers_dbb_generic_driver	Running	116.24	20.000	Target = monitor@3:checkpoint(20.000)



# USER GUIDES

This directory has all of the user guides for using the IPS (see the component and portal user guides for further information pertaining to those topics). It is organized in a series of *basic IPS usage* topics and *advanced IPS usage* topics, both are chock-full of examples and skeletons.

**How do I know if I am a Basic or Advanced user?** Basic IPS usage documents contain information that is intended for those who have run a few simulations and need a refresher on how to set up and run an existing simulation. These documents will help users run or make small modifications to existing simulations, including ways the IPS and other utilities can be used to examine scientific problems.

Advanced IPS usage documents contain information for *writers* of drivers and components. These documents will help those who wish to make new components and drivers, make significant changes to an existing component or driver, examine the performance of the IPS and components, or those who would like to understand how to use the multiple levels of parallelism and asynchronous communication mechanisms effectively.

## Basic IPS Usage

**Introduction to the IPS** A handy reference for constructing and running applications, this document helps users through the process of running a simulation based on existing components. It also includes: terminology, examples, and guidance on how to translate a computational or scientific question into a series of IPS runs.

**The Configuration File - Explained:** Annotated version of the configuration file with explanations of how and why the values are used in the framework and components.

**Using the Plasma State:** Essential guide to what the Plasma State is, the data it contains, and how to use it. This will go more in-depth than the component and driver writing guide, but less than the developers guide. It should contain how the PS is supposed to be used in various coupled simulation scenarios.

**Platform Configuration File - Explained:** Annotated platform configuration file and explanation of the manual allocation specification interface.

**Examples:** Sets of config files, batch scripts and more for users to use and modify for their own purposes.

## Advanced IPS Usage

**The IPS for Driver and Component Developers:** This guide contains the elements of components and drivers, suggestions on how to construct a simulation, how to add the new component to a simulation and the repository, as well as, an IPS services guide to have handy when writing components and drivers. This guide is for components and drivers based on the *generic driver* model. More sophisticated logic and execution models are covered in the following document.

**Fundamentals of the Advanced Features of the IPS:** Explanation of the different levels of parallelism, and other advanced features of the IPS in abstract terms, followed by examples. This is for the planning stages of simulation composition.

**Performance Analysis:** How to gather performance data for the IPS and its constituent tasks using Tau.

*Examples:* A listing of example files mentioned in this section.

## 4.1 Reference Guide for Running IPS Simulations

This reference guide is designed to help you through the process of setting up a simulation to run. It provides instructions on how to change configuration files and how to build and run the IPS on a given platform, as well as, determine if the simulation is setup correctly and will produce the correct data. In the various sections the user will find a series of questions designed to help the user plan for the preparation, execution, and post-processing of a run (or series of runs).

### 4.1.1 Terminology

Before going further, some basic definitions of terms that are used in the IPS must be presented. These terms are specific to the IPS and may be used in other contexts with different meanings. These are brief definitions and designed to remind the user of their meaning.

### 4.1.2 Elements of a Simulation

**Head node** The *head node* is how this documentation refers to any login, service or head node that acts as the gateway to a cluster or MPP. It is where the Python codes and some helper scripts run, including the framework, services and components.

**Compute node** A *compute node* is a node that exists in the compute partition of a parallel machine. It is designed for running compute intensive and parallel applications.

**Batch allocation** The *batch allocation* is the set of (compute) nodes given to the framework by the system's scheduler. The framework services manage the allocation of resources and launching of tasks on compute nodes within this allocation.

**Framework** The *framework* serves as the structure that contains the components, drivers and services for the simulation(s). It provides the infrastructure for the different elements to interact. It is the piece of software that is executed, and uses the services to invoke, run and manage the drivers, components, and tasks.

**Component** A *component* is a Python class that interacts with other components (typically the driver) and tasks using the services. A *physics* component typically uses the Python class to adapt a standalone physics code to be coupled with other components. Logically, each component contributes something to the simulation at hand, whether it is a framework functionality, like a bridge to the portal, or a model of some physical phenomena, like RF heating sources.

**Task** A *task* is an executable that runs on compute node(s) launched by the services on behalf of the component. These executables are the ones who do the heavy physics computation and dominate the run time, allowing the Python components and framework to manage the orchestration and other services involved in managing a multiphysics simulation. Most often tasks are parallel codes using MPI for interprocess communication.

**Driver (Component)** The *driver* is a special component in that it is the first one to be executed for the simulation. It is responsible for invoking its constituent components, implementing the time stepping and other logic among components, and global data operations, such as checkpointing.

**Init (Component)** The *init* component is a special in that it is invoked by the framework and is the first one to be executed for the simulation. It is responsible for performing any initialization needed by the driver before it begins its execution cycle.

**Port** A *port* is a category of component that can be implemented by different component implementations, i.e., components that wrap codes that different mathematical models of the same phenomenon. Each component that has the same port must implement the same interface (i.e., implement functions with the same names - in the

IPS all components implement “init”, “step”, and “finalize”), and provide the same functionality in a coupled simulation. In most cases, this means that it updates the same values in the plasma state. Drivers use the port name of a component to obtain a reference for that component at run time, as specified in the configuration file.

**Services** The framework *services* provide APIs for setting up the simulation, and managing data, resources, tasks, component invocations, access to configuration data and communication via an event service during execution. For more details, see our published [papers](#), [IPS developer documentation](#) and [code listings](#). Component writers should check out the [services API](#) for relevant services and tips on how to use them.

**Data files** Each component specifies the input and output *data files* it needs for a given simulation. These file names and locations are used to stage data in and out for each time step. Note that these are not the same as the *plasma state files*, in that *data files* are component local (and thus private).

**Plasma State files** The *plasma state* is a utility and set of files that allow multiple components to contribute values to a set of files representing the shared data about the plasma. These shared files are specified in the configuration file and access is managed through the framework services data management API. Component writers may need to write scripts to translate between plasma state files and the files expected/generated by the underlying executable.

**Configuration file** The *configuration file* allows the user to describe how a simulation is to be run. It uses a third-party Python package called [ConfigObj](#) to easily parse the shell-like, hierarchical syntax. In the configuration file there are sections describing the following aspects of the simulation. They are all explained in further detail in [The Configuration File - Explained](#).

**Platform Configuration file** The *platform configuration file* contains platform specific information needed by the framework for task and resource management, as well as paths needed by the portal and configuration manager. These rarely change, so the version in the top level of the IPS corresponding to the platform you are running on should be used.

**Batch script** The *batch script* tells the batch scheduler how and what to run, including the number of processes and nodes for the allocation, the command to launch the IPS, and any other information that the batch scheduler needs to know to run your job. There are some examples in the [examples](#) directory.

**Portal** The *portal* is a web portal set up to monitor SWIM runs. There are some additional capabilities with regard to saving and visualizing data associated with the run that are in development. See the [portal documentation](#) for more information and up-to-date capabilities.

### 4.1.3 Sample workflow

This section consists of an outline of how the IPS is intended to be used. It will walk you through the steps from forming an idea of what to run, through running it and analyzing the results. This will also serve as a reference for running IPS simulations. If you are not comfortable with the elements of an IPS simulation, then you should start with the sample simulations in [Getting Started](#) and review the terminology above.

#### Problem Formation

Before embarking on a simulation experiment, the problem that you are addressing needs to be determined. The problem may be a computational one where you are trying to determine if a component works properly, or an experiment to determine the scalability or sensitivity to computation parameters, such as time step length or number of particles. The problem may pertain to a study of how a component, or set of components, compare to previous results or real data. The problem may be to figure out for a set of variations which one produces the most stable plasma conditions. In each case, you will need to determine:

- what components are needed to perform this experiment?
- what input files must be obtained, prepared or generated (for each component and the simulation as a whole)?

- does this set of components make sense?
- what driver(s) are needed to perform this experiment?
- do new components and drivers need to be created?
- does it make sense to run multiple simulations in a single IPS instance?
- how will multiple simulations effect the computational needs and amount of data that is produced?
- what plasma state files are needed?
- where will initial plasma state values (and those not modeled by components in this scenario) come from?
- how much compute time and resources are needed for each task? the simulation as a whole?
- are there any restrictions on where or when this experiment can be run?
- how will the output data be analyzed?
- where will the output data go when the simulation is completed?
- when and where will the output data be analyzed?

Once you have a plan for constructing, managing and analyzing the results of your simulation(s), it is time to begin preparation.

### A Brief Introduction to Writing and Modifying Components

In many cases, new components or modifications to existing components need to be made. In this section, the anatomy of a component and a driver are explained for a simple invocation style of execution (see [Advanced User Guide](#) for more information on creating components and drivers with complex logic, parallelism and asynchronous control flow).

Each component is derived from the `Component` class, meaning that each IPS component inherits a few base capabilities, and then must augment them. Each IPS component must implement the following function bodies for the component class:

**`init(self, timeStamp=0)`** This function performs pre-simulation setup activities such as reading in global configuration parameters, checking configuration parameters, updating input files and internal state. (Component configuration parameters are populated *before* `init` is ever called.)

**`step(self, timeStamp=0)`** This function is the main part of the component. It is responsible for launching any tasks, and managing the input, output and plasma state during the course of the step.

**`finalize(self, timeStamp=0)`** This function is called after the simulation has completed and performs any clean up that is required by the component. Typically there is nothing to do.

**`checkpoint(self, timeStamp=0)`** This function performs a checkpoint for the component. All of the files marked as restart files in the configuration file are automatically staged to the checkpoint area. If the component has any internal knowledge or logic, or if there are any additional files that are needed to restart, this should be done explicitly here.

**`restart(self, timeStamp=0)`** This function replaces `init` when restarting a simulation from a previous simulation step. It should read in data from the appropriate files and set up the component so that it is ready to compute the next step.

To create a new component, there are two ways to do it, start from “scratch” by copying and renaming the skeleton component (`ips/doc/examples/skeleton_component.py`) to your desired location <sup>1</sup>, or by modifying an existing component (e.g., `ips/doc/examples/example_component.py`). When creating your new component, keep in mind that it should be somewhat general and usable in multiple contexts. In general, for things that

---

<sup>1</sup> Components are located in the `ips/components/` directory and are organized by *port name*, followed by implementation name. It is also common to put input files and helper scripts in the directory as well.



change often, you will want to use component configuration variables or input files to drive the logic or set parameters for the tasks. For more in depth information about how to create components and add them to the build process, see *Developing Drivers and Components for IPS Simulations*.

When changing an existing component that will diverge from the existing version, be sure to create a new version. If you are editing an existing component to make it better, be sure to document what you changexs.

## Setup Simulation

At this point, all components and drivers should be added to the repository, and any makefiles modified or created (see *makefile section* of component writing guide). You are now ready to set up the execution environment, build the IPS, and prepare the input and configuration files.

## Execution Environment

First, the platform on which to run the simulation must be determined. When choosing a platform, take in to consideration:

- The parallelism of the tasks you are running
  - Does your problem require 10s, 100s or 1000s of cores?
  - How well do your tasks take advantage of “many-core” nodes?
- The location of the input files and executables
  - Does your input data exist on a suitable platform?
  - Is it reasonable to move the data to another machine?
- Time and CPU hours
  - How much time will it take to run the set of simulations for the problem?
  - Is there enough CPU time on the machine you want to use?
- Dealing with results
  - Do you have access to enough hard drive space to store the output of the simulation until you have the time to analyze and condense it?

Once you have chosen a suitable platform, you may build the IPS like so:

```
host ~ > cd <path to ips>
host ips > . swim.bashrc.<machine_name>
host ips > svn up
host ips > make clean
host ips > cp config/makeconfig.<machine_name> config/makeconfig.local
host ips > make
host ips > make install
```

Second, construct input files or edit the appropriate ones for your simulation. This step is highly dependent on your simulation, but make sure that you check for the following things (and recheck after constructing the configuration file!):

- Does each component have all the input files it needs?
- Are there any global initial files, and are they present? (This includes any plasma state and non-plasma state files.)
- For each component input file: Are the values present, valid, and consistent?

- For the collection of files for each component: Are the values present, valid, and consistent?
- For the collection of files for each simulation: Are the values present, valid, and consistent?
- Do the components model all of the targeted domain and phenomena of the experiment?
- Does the driver use the components you expect?
- Does the driver implement the data dependencies between the components as you wish?

Third, you must construct the configuration file. It is helpful to start with a configuration file that is related to the experiment you are working on, or you may start from the example configuration file, and edit it from there. Some configuration file values are user specific, some are platform specific, and others are simulation or component specific. It may be helpful to save your personal versions on each machine in your home directory or some other persistent storage location for reuse and editing. These tend not to be good files to keep in subversion, however there are some examples in the example directory to get you started. The most common and required configuration file entries are explained here. For more a more complete description of the configuration options, see [The Configuration File - Explained](#).

- User Data Section:

```
USER_W3_DIR = <location of your web directory on this platform>
USER_W3_BASEURL = <URL of your space on the portal>
USER = <user name> # Optional, if missing the unix username is used
```

Set these values to the www directory you created for your own runs, a matching url for the portal to store your run info, and your user name (this is used on the portal to identify simulations you run). These should be the same for all of your runs on a given platform.

- Simulation Info Section:

```
RUN_ID = <short name of run>
TOKAMAK_ID = <name of the tokamak>
SHOT_NUMBER = 1
...
SIM_NAME = ${RUN_ID}_${SHOT_NUMBER}

OUTPUT_PREFIX =

IPS_ROOT = <location of built ips>
SIM_ROOT = <location of output tree>

RUN_COMMENT = <used by portal to help identify what ran and why>
TAG = <grouping string>
...
SIMULATION_MODE = NORMAL
RESTART_TIME =
RESTART_ROOT = ${SIM_ROOT}
```

In this section the simulation is described and key locations are specified. *RUN\_COMMENT* and *TAG*, along with *RUN\_ID*, *TOKAMAK\_ID*, and *SHOT\_NUMBER* are used by the portal to describe this simulation. *RUN\_ID*, *TOKAMAK\_ID*, and *SHOT\_NUMBER* are commonly used to construct the *SIM\_NAME*, which is often used in as the directory name of the *SIM\_ROOT*. The *IPS\_ROOT* is the top-level of the IPS source tree that you are using to execute this simulation. And finally, the *SIMULATION\_MODE* and related items identify the simulation as a *NORMAL* or *RESTART* run.

- Logging Section:

```
LOG_FILE = ${RUN_ID}_sim.log
LOG_LEVEL = DEBUG | WARN | INFO | CRITICAL
```

The logging section defines the name of the log file and the default level of logging for the simulation. The log file for the simulation will contain all logging messages generated by the components in this simulation. Logging messages from the framework and services will be written to the framework log file. The *LOG\_LEVEL* may be the following and may differ from the framework log level (in order of most verbose to least) <sup>2</sup>:

- *DEBUG* - all messages are produced, including debugging messages to help diagnose problems. Use this setting for debugging runs only.
- *INFO* - these are messages stating what is happening, as opposed to what is going wrong. Use this logging level to get an idea of how the different pieces of the simulation interact, without extraneous messages from the debugging level.
- *WARN* - these messages are produced when the framework or component expects different conditions, but has an alternative behavior or default value that is also valid. In most cases these messages are harmless, but may indicate a behavior that is different than expected. This is the most common logging level.
- *ERROR* - conditions that throw exceptions typically also produce an error message through the logging mechanism, however not all errors result in the failure of a component or the framework.
- *CRITICAL* - only messages about fatal errors are produced. Use this level when using a well known and reliable simulation.

- Plasma State Section:

```
PLASMA_STATE_WORK_DIR = ${SIM_ROOT}/work/plasma_state

# Config variables defining simulation specific names for plasma state files
CURRENT_STATE = ${SIM_NAME}_ps.cdf
PRIOR_STATE = ${SIM_NAME}_psp.cdf
NEXT_STATE = ${SIM_NAME}_psn.cdf
CURRENT_EQDSK = ${SIM_NAME}_ps.geq
CURRENT_CQL = ${SIM_NAME}_ps_CQL.dat
CURRENT_DQL = ${SIM_NAME}_ps_DQL.nc
CURRENT_JSJSK = ${SIM_NAME}_ps.jso

# List of files that constitute the plasma state
PLASMA_STATE_FILES1 = ${CURRENT_STATE} ${PRIOR_STATE} ${NEXT_STATE} ${CURRENT_EQDSK}
PLASMA_STATE_FILES2 = ${CURRENT_CQL} ${CURRENT_DQL} ${CURRENT_JSJSK}
PLASMA_STATE_FILES = ${PLASMA_STATE_FILES1} ${PLASMA_STATE_FILES2}
```

Specifies the naming convention for the plasma state files so the framework and components can manipulate and reference them in the config file and during execution. The initial file locations are also specified here.

- Ports Section:

```
[PORTS]
    NAMES = INIT DRIVER MONITOR EPA RF_IC NB FUS

# Required ports - DRIVER and INIT
[[DRIVER]]
    IMPLEMENTATION = GENERIC_DRIVER

[[INIT]]
    IMPLEMENTATION = minimal_state_init
# Physics ports
[[RF_IC]]
    IMPLEMENTATION = model_RF_IC

[[FP]]
```

<sup>2</sup> For more information and guidance about how the Python logging module works, see the Python logging module [tutorial](#).

```
IMPLEMENTATION = minority_model_FP

[[FUS]]
    IMPLEMENTATION = model_FUS

[[NB]]
    IMPLEMENTATION = model_NB

[[EPA]]
    IMPLEMENTATION = model_EPA

[[MONITOR]]
    IMPLEMENTATION = monitor_comp_4
```

The ports section specifies which ports are included in the simulation and which implementation of the port is to be used. Note that a *DRIVER* must be specified, and a warning will be issued if there is no *INIT* component present at start up. The value of *IMPLEMENTATION* for a given port *must* correspond to a component description below.

- Component Configuration Section:

```
[<component name>]
    CLASS = <port name>
    SUB_CLASS = <type of component>
    NAME = <class name of component implementation>
    NPROC = <# of procs for task invocations>
    BIN_PATH = ${IPS_ROOT}/bin
    INPUT_DIR = ${DATA_TREE_ROOT}/<location of input directory>
    INPUT_FILES = <input files for each step>
    OUTPUT_FILES = <output files to be archived>
    PLASMA_STATE_FILES = ${CURRENT_STATE} ${NEXT_STATE} ${CURRENT_EQDSK}
    RESTART_FILES = ${INPUT_FILES} <extra state files>
    SCRIPT = ${BIN_PATH}/<component implementation>
```

For each component, fill in or modify the entry to match the locations of the input, output, plasma state, and script locations. Also, be sure to check the *NPROC* entry to suit the problem size and scalability of the executable, and add any component specific entries that the component implementation calls for. The data tree is a SWIM-public area where simulation input data can be stored. It allows multiple users to access the same data and have reasonable assurance that they are indeed using the same versions. On franklin the data tree root is `/project/projectdirs/m876/data/`, and on stix it is `/p/swim1/data/`. The plasma state files must be part of the simulation plasma state. It may be a subset if there are files that are not needed by the component on each step. Additional component-specific entries can also appear here to signal a piece of logic or set a data value.

- Checkpoint Section:

```
[CHECKPOINT]
    MODE = WALLTIME_REGULAR
    WALLTIME_INTERVAL = 15
    NUM_CHECKPOINT = 2
    PROTECT_FREQUENCY = 5
```

This section specifies the checkpoint policy you would like enforced for this simulation, and the corresponding parameters to control the frequency and number of checkpoints taken. See the comments in the same configuration file or the configuration file [documentation](#). If you are debugging or running a component or simulation for the first time, it is a good idea to take frequent checkpoints until you are confident that the simulation will run properly. For guidance on specifying the checkpoint interval, see [Fundamentals of the Advanced Features of the IPS](#).

- Time Loop Section:

```
[TIME_LOOP]
  MODE = REGULAR
  START = 0.0
  FINISH = 20.0
  NSTEP = 5
```

This section sets up the time loop to help the driver manage the time progression of the simulation. If you are debugging or running a component or simulation for the first time, it is a good idea to take very few steps until you are confident that the simulation will run properly.

Lastly, double-check that your input files and config file are both self-consistent and make physics sense.

## Run Simulation

Now, that you have everything set up, it is time to construct the batch script to launch the IPS. Just like the configuration files, this is something that tends to be user specific and platform specific, so it is a good idea to keep local copy in a persistent directory on each platform you tend to use for easy modification.

As an example, here is a skeleton of a batch script for Franklin:

```
#!/bin/bash
#PBS -A <project code for accounting>
#PBS -N <name of simulation>
#PBS -j oe                                # joins stdout and stderr
#PBS -l walltime=0:6:00
#PBS -l mppwidth=<number of *cores* needed>
#PBS -q <queue to submit job to>
#PBS -S /bin/bash
#PBS -V

IPS_ROOT=<location of IPS root>
cd $PBS_O_WORKDIR
umask=0222

$IPS_ROOT/bin/ips [--config=<config file>]+ \
  --platform=$IPS_ROOT/franklin.conf \
  --log=<name of log file> \
  [--debug] \
  [--nodes=<number of nodes in this allocation>] \
  [--ppn=<number of processes per node for this allocation>]
```

Note that you can only run one instance of the IPS per batch submission, however you may run multiple simulations in the same batch allocation by specifying multiple `--config=<config file>` entries on the command line. Each config file must have a unique file name, and `SIM_ROOT`. The different simulations will share the resources in the allocation, in many cases improving the resource efficiency, however this may make the execution time of each individual simulation a bit longer due to waiting on resources. For more information on running multiple simulations, see *Fundamentals of the Advanced Features of the IPS*.

The IPS also needs information about the platform it is running on (`--platform=$IPS_ROOT/franklin.conf`) and a log file (`--logfile=<name of log file>`) for the framework output. Platform files for commonly used platforms are provided in the top-level of the ips directory. It is strongly recommended that you use the appropriate one for launching IPS runs. See *Platforms and Platform Configuration* for more information on how to use or create these files.

Lastly, there are some optional command line arguments that you may use. `--debug` will turn on debugging information from the framework. `--nodes` and `--ppn` allow the user to manually set the number of nodes and processes per node for the framework. This will override any detection by the framework and should be used with caution. It is, however, a convenient way to run the ips on a machine without a batch scheduler.

Once your job is running, you can watch their progress on the [portal](#). Note that each *simulation* will appear on the portal, so multiple simulation jobs will look like multiple simulations that all started around the same time.

### Analysis and/or Debugging

Once your run (or set of runs) is done, it is time to look at the output. First, we will examine the structure of the output tree:

```

${SIM_ROOT}/
  ${PORTAL_RUNID}
    File containing the portal run ids that are associated with this directory. There can be
    more than one.
  <platform config file>
  <simulation configuration files>
    Each simulation configuration file that used this sim root.
  restart/
    <each checkpoint>/
      <each component>/
        Directory containing the restart files for this checkpoint
  simulation_log/
    Directory containing the event log for each runid.
  simulation_results/
    <each time step>/
      components/
        <each component>/
          Directory containing the output files for the given component at
          the given step.
        <each component>/
          Directory containing the output files for each step. File names are appended
          with the time step to avoid collisions.
  simulation_setup/
    <each component>/
      Directory containing the input files from the beginning of the simulation.
  work/
    <each component>/
      Directory where the component computes from time step to time step. Left-
      over input and output files from the last step will be present at the end of the
      simulation.
```

There are a few tools for visualizing (and light analysis) of a run or set of runs:

- Portal web interface to PCMF: This tool is a web interface to the PCMF tool (see below). It has recently been integrated into the portal for quick and remote viewing. For more in depth analysis, viewing and printing of graphs from the monitor component, use the more powerful standalone version of PCMF.
- PCMF: A tool to Plot and Compare multiple Monitor Files (`ips/components/monitor/monitor_4/PCMF.py`) is the local Python version of the web tool. It uses Matplotlib to generate plots of the different values in the plasma state over the course of the simulation. It also allows you to generate graphs for more than one set of monitor files. Examples and instructions are located in the repo and are coming soon to this documentation.
- ELVis: This tool graphs values from netCDF (plasma state) files through a web browser plugin or using the Java client.

Using these utilities, your own scripts or manual inspection results can be analyzed, or bugs found. Debugging a coupled simulation is more complicated than debugging a standalone code. Here are some things to consider when a problem is encountered:

- Problems using the framework
  - Was an exception thrown? If so, what was it and where did it come from? If you don't understand the exception, talk to a framework developer.
  - Was something missing in the configuration file?
  - Were the components invoked and tasks launched as expected?
  - Did you use the proper implementation of the component and executable?
  - Was your compute environment/permissions/batch allocation set up properly?
- Data between components
  - Does each component update all the values in the plasma state it needs to?
  - Does each component update all output files it uses internally properly?
  - Are the components updating the plasma state in the right order?
- Physics code problem
  - Did a task return an error code?
  - Does the component check for a bad return code and handle it properly?
  - Is the code that is launched have the proper command line arguments?
  - Are the input and output files properly adapted to the executable?
  - Does the executable fail in standalone mode?
  - Was the executable built properly?
  - Were all necessary input and source files found?

If you are working out a problem, it is always good to:

- Turn on debugging output using the `--debug` flag on the command line, and setting the `LOG_LEVEL` in the configuration file to `DEBUG`.
- Turn on debugging output in physics codes to see what is going on during each task.
- Use frequent checkpoints to restart close to where the problem starts.
- Reduce the number of time steps to the minimum needed to produce the problem.
- Only change one thing before rerunning the simulation to determine what fixes the problem.

## 4.2 Developing Drivers and Components for IPS Simulations

This section is for those who wish to modify and write drivers and components to construct a new simulation scenario. It is expected that readers are familiar with IPS terminology, the directory structure and have looked at some existing drivers and components before attempting to modify or create new ones. This guide will describe the elements of a simulation, how they work together, the structure of drivers and components, IPS services API, and a discussion of control flow, data flow and fault management.

### 4.2.1 Elements of a Simulation

When constructing a new simulation scenario, writing a new component or even making small modifications to existing components and drivers, it is important to consider and understand how the pieces of an IPS simulation work together. An IPS simulation scenario is specified in the *configuration file*. This file tells the framework how to set up the output tree for the data files, which components are needed and where the implementation is located, time loop and checkpoint parameters, and input and output files for each component and the simulation as a whole are specified. The *framework* uses this information to find the pieces of code and data that come together to form the simulation, as well as provide this information to the components and driver to manage the simulation and execution of tasks<sup>3</sup>.

The framework provides *services* that are used by components to perform data, task, resource and configuration management, and provides an event service for exchanging messages with internal and external entities. While these services are provided as a single API to component writers, the documentation (and underlying implementation) divides them into groups of methods to perform related actions. *Data management* services include staging input, output and plasma state files, changing directories, and saving task restart files, among others. The framework will perform these actions for the calling component based on the files specified in the configuration file and within the method call maintaining coherent directory spaces for each component's work, previous steps, checkpoints and globally accessible data to insure that name collisions do not corrupt data and that global files are accessed in a well-defined manner<sup>4</sup>. Services for *task management* include methods for component method invocations, or *calls*, and executable launch on compute nodes, or *task launches*. The task management portion of the framework works in conjunction with the IPS resource manager to execute multiple parallel executables within a single batch allocation, allowing IPS simulations to efficiently utilize compute resources, as data dependencies allow. The IPS task manager provides blocking and non-blocking versions of `call` and `launch_task`, including a notion of *task pools* and the ability to wait for the completion of any or all calls or tasks in a group. These different invocation and launch methods allow a component writer to manage the control flow and implement data dependencies between components and tasks. This task management interface hides the resource management, platform specific, task scheduling, and process interactions that are performed by the framework, allowing component writers to express their simulations and component coupling more simply. The *configuration manager* primarily reads the configuration file and instantiates the components for the simulation so that they can interact over the course of the simulation. It also provides an interface for accessing key data elements from the configuration file, such as the time loop, handles to components and any component specific items listed in the configuration file.

There are three classes of components: framework, driver, and general purpose (physics components fall into this category). In the IPS, each component executes in a separate process (a child of the framework) and implements the following methods:

**`init(self, timeStamp=0)`** This function performs pre-simulation setup activities such as reading in global configuration parameters, checking configuration parameters, updating input files and internal state. (Component configuration parameters are populated *before* `init` is ever called.)

---

<sup>3</sup> Tasks are the binaries that are launched by components on compute nodes, where as components are Python scripts that manage the data movements and execution of the tasks (with the help of IPS services). In general, the component is aware of the driver and its existence within a coupled simulation, and the task does not.

<sup>4</sup> The IPS uses an agreed upon file format and associated library to manage global (shared) data for the simulation, called the Plasma State. It is made up of a set of netCDF files with a defined layout so that codes can access and share the data. At the beginning of each step the component will get a local copy of the current plasma state, execute based on these values, and then update the plasma state values that it changed to the global copy. There are data management services to perform these actions, see *Data Management API*.



**step(self, timeStamp=0)** This function is the main part of the component. It is responsible for launching any tasks, and managing the input, output and plasma state during the course of the step.

**finalize(self, timeStamp=0)** This function is called after the simulation has completed and performs any clean up that is required by the component. Typically there is nothing to do.

**checkpoint(self, timeStamp=0)** This function performs a checkpoint for the component. All of the files marked as restart files in the configuration file are automatically staged to the checkpoint area. If the component has any internal knowledge or logic, or if there are any additional files that are needed to restart, this should be done explicitly here.

**restart(self, timeStamp=0)** This function replaces `init` when restarting a simulation from a previous simulation step. It should read in data from the appropriate files and set up the component so that it is ready to compute the next step.

The component writer will use the services API to help perform data, task, configuration and event management activities to implement these methods.

This document focuses on helping (physics) component and driver writers successfully write new components. It will take the writer step-by-step through the process of writing basic components. Detailed discussions of multiple levels of parallelism, fault tolerance strategies, performance and resource utilization considerations, and asynchronous coordination of simulations can be found in the [advanced topics](#) documentation.

## 4.2.2 Writing Components

In this section, we take you through the steps of adding a new component to the IPS landscape. It will cover where to put source code, scripts, binaries and inputs, how to construct the component, how to add the component to the IPS build system, and some tips to make this process smoother.

### Adding a New Binary

The location of the binary does not technically matter to the framework, as long as the path can be constructed by the component and the permissions are set properly to launch it when the time comes. There are two recommended ways to express the location of the binary to the component:

1. For stable and shared binaries, the convention is to put them in the platform's *PHYS\_BIN*. This way, the *PHYS\_BIN* is specified in the platform configuration file and the component can access the location of the binary relative to that location on each machine. See [Platforms and Platform Configuration](#).
2. The location of the binary is specified in the component's section of the simulation configuration file. This way, the binary can be specified just before runtime and the component can access it through the framework services. This convention is typically used during testing, experimentation with new features in the code, or other circumstances where the binary may not be stable, fully compatible with other components, or ready to be shared widely.

### Data Coupling Preparation

Once you have your binary built properly and available, it is time to work on the data coupling to the other components in a simulation. This is a component specific task, but it often takes conversation with the other physicists in the group as to what values need to be communicated and to develop an understanding of how they are used.

When the physics of interest is identified, adapters need to be written to translate IPS-style inputs (from the Plasma State) to the inputs the binary is expecting, and a similar adapter for the output files. More details on how to use the Plasma State and adapting binaries can be found in the [Plasma State Guide](#).

## Create a Component

Now it is time to start writing the component. At this point you should have an idea of how the component will fit into a coupled simulation and the types of activities that will need to happen during the *init*, *step*, and *finalize* phases of execution.

1. Create a directory for your component (if you haven't already). The convention in the IPS repository is to put component scripts and helpers in `ips/components/<port_name>/<component_name>`, where *port\_name* is the “type” of component, and the *component\_name* is the implementation of that “type” of component. Often, *component\_name* will contain the name of the code it executes. If there is already a component directory and existing components, then you may want to make your own directory within the existing component's directory or just add your component in that same directory.
2. Copy the skeleton component (`ips/doc/examples/skeleton_comp.py`) to the directory you choose or created. Be sure to name it such that others will easily know what the component does. For example, a component for TORIC, a code that models radio frequency heating in plasmas, is found in `ips/components/rf/toric/` and called `rf_ic_toric_mcmd.py`.
3. Edit skeleton. Components should be written such that the inputs, outputs, binaries and other parameters are specified in the configuration file or appear in predictable locations *across platforms*. The skeleton contains an outline, in comments, of the activities that a generic component does in each method invocation. You will need to fill in the outline with your own calls to the services and any additional activities in the appropriate places. Take a look at the other example components in the `ips/doc/examples/` or `ips/components/` for guidance. The following is an outline of the changes that need to be made:
  - (a) Change the name of the class and update the file to use that name every where it says `# CHANGE EXAMPLE TO COMPONENT NAME`.
  - (b) Modify `init` to initialize the input files that are needed for the first step. Update shared files as needed.
  - (c) Modify `step` to use the appropriate *prepare\_input* and *process\_output* executables. Make sure all shared files that are changed during the course of the task execution are saved to their proper locations for use by other components. Make sure that all output files that are needed for the next step are copied to archival location. If a different task launch mechanism is required, modify as needed. See [Task Launch API](#) for related services.
  - (d) Modify `finalize` to do any clean up as needed.
  - (e) Modify `checkpoint` to save all files that are needed to restart from later.
  - (f) Modify `restart` to set up the component to resume computation from a checkpointed step.

While writing your component, be sure to use `try...except` blocks<sup>5</sup> to catch problems and the services logging mechanisms to report critical errors, warnings, info and debug messages. It is *strongly* recommended that you use exceptions and the services logging capability for debugging and output. Not catching exceptions in the component can lead to the driver or framework catching them in a weird place and it will likely take a long time to track down where the problem occurred. The logging mechanism in the IPS provides time stamps of when the event occurred, the component that produced the message, as well as a nice way to format the message information. These messages are written to the log file (specified in the configuration file for the simulation) atomically, unlike normal print statements. Absolute ordering is not guaranteed across different components, but ordering within the same component is guaranteed. See [Logging API](#) for more information on when to use the different logging levels.

At this point, it might be a good idea to start the documentation of the component in `ips/doc/component_guides/`. You will find a *README* file in `ips/doc/` that explains how to build and write IPS documentation, and another in the `ips/doc/component_guides/` on what information to include in your component documentation.

---

<sup>5</sup> Tutorial on exceptions

## Makefile

Once you are satisfied with the implementation of the component, it is time to construct and edit the Makefiles such that the component is built properly by the framework. The Makefile will build your executables and move scripts to `${IPS_ROOT}/bin`.

1. If you do not already have a makefile in the directory for your new component, copy the examples (`ips/doc/examples/Makefile` and `ips/doc/examples/Makefile.include`) to your component directory.
2. List all executables to be compiled in *EXECUTABLES* and scripts in *SCRIPTS*.

```
EXECUTABLES = do_toric_init prepare_toric_input process_toric_output \
              process_toric_output_mcmd # Ptoric.e Storic.e
SCRIPTS = rf_ic_toric.py rf_ic_toric_mcmd.py
TARGETS = $(EXECUTABLES)
```

3. Make targets for each executable. Do not remove targets *all*, *install*, *clean*, *distclean*, and *.depend*.
4. Add any libraries that are needed to `ips/config/makeconfig.local`. (This is where *LIBS* and the various fortran flags are defined.)
5. Add component to top-level Makefile. Toric example:

```
TORIC_COMP_DIR=components/rf/toric/src
TORIC_COMP=.TORIC
```

6. Add component dir to *COMPONENT\_DIRS*:

```
COMPONENTS_DIRS=$(AORSA_COMP_DIR) \
                $(TORIC_COMP_DIR) \
                $(BERRY_INIT_COMP_DIR) \
                $(CHANGE_POWER_COMP_DIR) \
                $(BERRY_CQL3D_INIT_COMP_DIR) \
                $(CHANGE_POWER_COMP_DIR) \
                $(CQL3D_COMP_DIR) \
                $(ELWASIF_DRIVER_COMP_DIR) \
                ...
```

7. Add component to *COMPONENTS*:

```
COMPONENTS=$(AORSA_COMP) \
            $(TORIC_COMP) \
            $(BERRY_AORSA_INIT_COMP) \
            $(BERRY_CQL3D_INIT_COMP) \
            $(CHANGE_POWER_COMP) \
            $(CQL3D_COMP) \
            $(BERRY_INIT_COMP) \
            $(ELWASIF_DRIVER_COMP) \
            ...
```

Now you should be able to build the IPS with your new component.

## Testing and Debugging a Component

Now it is time to construct a simulation to test your new component. There are two ways to test a new component. The first is to have the IPS just run that single component without a driver, by specifying your component as the driver. The second is to plug it into an existing driver. The former will test only the task launching and data movement capabilities.

The latter can also test the data coupling and call interface to the component. This section will describe how to xtest your component using an existing driver (including how to add the new component to the driver).

As you can see in the example component, almost everything is specified in the configuration file and read at run-time. This means that the configuration of components is vitally important to their success or failure. The entries in the component configuration section are made available to the component automatically, thus a component can access them by `self.<entry_name>`. This is useful in many cases, and you can see in the example component that `self.NPROC` and `self.BIN_PATH` are used. Global configuration parameters can also be accessed using services call `get_config_param(<param_name>)` ([API](#)).

Drivers access components by their port names (as specified in the configuration file). To add a new component to the driver you will either need to add a new port name or use an existing port name. `ips/components/drivers/dbb/generic_driver.py` is a good all-purpose driver that most components should be able to use. If you are using an existing port name, then the code should just work. It is recommended to go through the driver code to make sure the component is being used in the expected manner. To add a new port name, you will need to add code to `generic_driver.step()`:

- get a reference to the port (`self.services.get_port(<name of port>)`)
- call “init” on that component (`self.services.call(comp_ref, “init”)`)
- call “step” on that component (`self.services.call(comp_ref, “step”)`)
- call “finalize” on that component (`self.services.call(comp_ref, “finalize”)`)

The following sections of the configuration file may need to be modified. If you are not adding the component to an existing simulation, you can copy a configuration file from the examples directory and modify it.

### 1. Plasma State (Shared Files) Section

You will need to modify this section to include any additional files needed by your component:

```
# Where to put plasma state files as the simulation evolves
PLASMA_STATE_WORK_DIR = ${SIM_ROOT}/work/plasma_state
CURRENT_STATE = ${SIM_NAME}_ps.cdf
PRIOR_STATE = ${SIM_NAME}_psp.cdf
NEXT_STATE = ${SIM_NAME}_psn.cdf
CURRENT_EQDSK = ${SIM_NAME}_ps.geq
CURRENT_CQL = ${SIM_NAME}_ps_CQL.nc
CURRENT_DQL = ${SIM_NAME}_ps_DQL.nc
CURRENT_JSDBSK = ${RUN_ID}_ps.jso

# What files constitute the plasma state
PLASMA_STATE_FILES1 = ${CURRENT_STATE} ${PRIOR_STATE}
                     ${NEXT_STATE}
PLASMA_STATE_FILES2 = ${PLASMA_STATE_FILES1} ${CURRENT_EQDSK}
                     ${CURRENT_CQL} ${CURRENT_DQL}
PLASMA_STATE_FILES = ${PLASMA_STATE_FILES2} ${CURRENT_JSDBSK}
```

### 2. Ports Section

You will need to add the component to the ports section so that it can be properly detected by the framework and driver. An entry for *DRIVER* must be specified, otherwise the framework will abort. Also, a warning is produced if there is no *INIT* component. Note that all components added to the *NAMES* field must have a corresponding subsection.

```
[PORTS]
  NAMES = INIT DRIVER MONITOR EPA NB
  [[DRIVER]]
    IMPLEMENTATION = EPA_IC_FP_NB_DRIVER
  [[INIT]]
```

```

        IMPLEMENTATION = minimal_state_init
    [[RF_IC]]
        IMPLEMENTATION = model_RF_IC

    ...

```

### 3. Component Description Section

The ports section just defines which components are going to be used in this simulation, and point to the section where they are described. The component description section is where those definitions take place:

```

[TSC]
    CLASS = epa
    SUB_CLASS =
    NAME = tsc
    NPROC = 1
    BIN_PATH = ${IPS_ROOT}/bin
    INPUT_DIR = ${IPS_ROOT}/components/epa/tsc
    INPUT_FILES = inputa.I09001 sprsina.I09001config_nbi_ITER.dat
    OUTPUT_FILES = outputa tsc.cgm inputa log.tsc ${PLASMA_STATE_FILES}
    SCRIPT = ${BIN_PATH}/epa_nb_iter.py

```

The component section starts with a label that matches what is listed as the implementation in the ports section. These *must* match or else the framework will not find your component and the simulation will fail before it starts (or worse, use the wrong implementation!). *CLASS* and *SUBCLASS* typically refer to the directory hierarchy and are sometimes used to identify the location of the source code and input files. Note that *NAME* must match the python class name that implements the component. *NPROC* is the number of *processes* that the binary needs to use when launched on compute nodes. The *BIN\_PATH* will almost always be `${IPS_ROOT}/bin` and refers to the location of any binaries you wish to use in your component. The Makefile will move your component script to `${IPS_ROOT}/bin` when you build the IPS, and should do the same to any binaries that are produced from the targets in the Makefile. If you have pre-built binaries that exist in another location, an additional entry in the component description section may be a convenient place to put it. *INPUT\_DIR*, *INPUT\_FILES* and *OUTPUT\_FILES* specify the location and names of the input and output files, respectively. If a subset of plasma states files is all that is required by the component, they are specified here (*PLASMA\_STATE\_FILES*). If the entry is omitted, *all* of the plasma state files are used. This prevents the full set of files to be copied to and from the component's work directory on every step, saving time and space. Lastly, *SCRIPT* is the Python script that contains the component code, specifically the Python class in *NAME*. Additionally, any component specific values maybe specified here to control logic or set data values that change often.

### 4. Time Loop Section

This may need to be modified for your component or the driver that uses your new component. During testing, a small number of steps is appropriate.

```

# Time loop specification (two modes for now) EXPLICIT | REGULAR
# For MODE = REGULAR, the framework uses the variables START, FINISH, and NSTEP
# For MODE = EXPLICIT, the framework uses the variable VALUES (space separated
# list of time values)
[TIME_LOOP]
    MODE = EXPLICIT
    VALUES = 75.000 75.025 75.050 75.075 75.100 75.125

```

## Tips

This section contains some useful tips on testing, debugging and documenting your new component.

- General:

- Naming is important. You do not want the name of your component to overlap with another, so make sure it is unique.
- Be sure to commit all the files and directories that are needed to build and run your component. This means the executables, Makefiles, component script, helper scripts and input files.
- Testing:
  - To test a new component, first run it as the driver component of a simulation all by itself. This will make sure that the component itself works with the framework.
  - The next step is to have a driver call just your new component to make sure it can be discovered and called by the driver properly.
  - The next step is to determine if the component can exchange global data with another component. To do this run two components in a driver and verify they are exchanging data properly.
  - When testing IPS components and simulations, it may be useful to turn on debugging information in the IPS and the underlying executables.
  - If this is a time stepping simulation, a small number of steps is useful because it will lead to shorter running times, allowing you to submit the job to a debug or other faster turnaround queue.
- Debugging:
  - Add logging messages (*services.info()*, *services.warning()*, etc.) to make sure your component does what you think it does.
  - Remove other components from the simulation to figure out which one or which interaction is causing the problem
  - Take many checkpoints around the problem to narrow in on the problem.
  - Remove concurrency to see if one component is overwriting another's data.
- Documentation:
  - Document the component code such that another person can understand how it works. It helps if the structure remains the same as the example component.
  - Write a description of what the component does, the inputs it uses, outputs it produces, and what scenarios and modes it can be used in in the component documentation section, [Component Guides](#).

### 4.2.3 Writing Drivers

The driver of the simulation manages the control flow and synchronization across components via time stepping or implicit means, thus orchestrating the simulation. There is only one driver per simulation and it is invoked by the framework and is responsible for invoking the components that make up the simulation scenario it implements. It is also responsible for managing data at the simulation level, including checkpoint and restart activities.

Before writing a driver, it is a good idea to have the components already written. Once the components that are to be used are chosen the data coupling and control flow must be addressed.

In order to couple components, the data that must be exchanged between them and the ordering of updates to the plasma state must be determined. Once the data dependencies are identified (which components have to run before the next, and which ones can run at the same time). You can write the body of the driver. Before going through the steps of writing a driver, review the [method invocation API](#) and plan which methods to use during the main time loop. If you are writing a driver that uses the event service for synchronization, see [Advanced Features](#) for instructions and examples.

The framework will invoke the methods of the *INIT* and *DRIVER* components over the course of the simulation, defining the execution of the run:

- `init_comp.init()` - initialization of initialization component
- `init_comp.step()` - execution of initialization work
- `init_comp.finalize()` - cleanup and confirmation of initialization
- `driver.init()` - any initialization work (typically empty)
- `driver.step()` - the bulk of the simulation
  - get references to the ports
  - call *init* on each port
  - get the time loop
  - implement logic of time stepping
  - during each time step:
    - \* perform pre-step logic that may stage data or determine which components need to run or what parameters are given to each component
    - \* call *step* on each port (as appropriate)
    - \* manage global plasma state at the end of each step
    - \* checkpoint components (frequency of checkpoints is controlled by framework)
  - call *finalize* on each component
- `driver.finalize()` - any clean up activities (typically empty)

It is recommended that you start with the `ips/components/drivers/dbb/generic_driver.py` and modify it as needed. You will most likely be changing: how the components are called in the main loop (the generic driver calls each component in sequence), the pre-step logic phase, and what ports are used. The data management and checkpointing calls should remain unchanged as their behavior is controlled in the configuration file.

The process for adding a new driver to the IPS is the same as that for the component. See the appropriate sections above for adding a component.

## 4.2.4 IPS Services API

The IPS framework contains a set of managers that perform services for the components. A component uses the services API to access them, thus hiding the complexity of the framework implementation. Below are descriptions of the individual function calls grouped by type. To call any of these functions in a component replace *ServicesProxy* with *self.services*. The *services* object is passed to the component upon creation by the framework.

### Component Invocation

Component invocation in the IPS means one component is calling another component's function. This API provides a mechanism to invoke methods on components through the framework. There are blocking and non-blocking versions, where the non-blocking versions require a second function to check the status of the call. Note that the *wait\_call* has an optional argument (*block*) that changes when and what it returns.

`ServicesProxy.call(component_id, method_name, *args)`

Invoke method *method\_name* on component *component\_id* with optional arguments *\*args*. Return result from invoking the method.

`ServicesProxy.call_nonblocking(component_id, method_name, *args)`

Invoke method *method\_name* on component *component\_id* with optional arguments *\*args*. Return *call\_id*.



ServicesProxy.**wait\_call**(*call\_id*, *block=True*)

If *block* is `True`, return when the call has completed with the return code from the call. If *block* is `False`, raise `ipsExceptions.IncompleteCallException` if the call has not completed, and the return value is it has.

ServicesProxy.**wait\_call\_list**(*call\_id\_list*, *block=True*)

Check the status of each of the call in *call\_id\_list*. If *block* is `True`, return when *all* calls are finished. If *block* is `False`, raise `ipsExceptions.IncompleteCallException` if *any* of the calls have not completed, otherwise return. The return value is a dictionary of *call\_ids* and return values.

## Task Launch

The task launch interface allows components to launch and manage the execution of (parallel) executables. Similar to the component invocation interface, the behavior of *launch\_task* and the *wait\_task* variants are controlled using the *block* keyword argument and different interfaces to *wait\_task*.

ServicesProxy.**launch\_task**(*nproc*, *working\_dir*, *binary*, *\*args*, *\*\*keywords*)

Launch *binary* in *working\_dir* on *nproc* processes. *\*args* are any arguments to be passed to the binary on the command line. *\*\*keywords* are any keyword arguments used by the framework to manage how the binary is launched. Keywords may be the following:

- *task\_ppn* : the processes per node value for this task
- *block* : specifies that this task will block (or raise an exception) if not enough resources are available to run immediately. If `True`, the task will be retried until it runs. If `False`, an exception is raised indicating that there are not enough resources, but it is possible to eventually run. (default = `True`)
- *tag* : identifier for the portal. May be used to group related tasks.
- *logfile* : file name for `stdout` (and `stderr`) to be redirected to for this task. By default `stderr` is redirected to `stdout`, and `stdout` is not redirected.
- *whole\_nodes* : if `True`, the task will be given exclusive access to any nodes it is assigned. If `False`, the task may be assigned nodes that other tasks are using or may use.
- *whole\_sockets* : if `True`, the task will be given exclusive access to any sockets of nodes it is assigned. If `False`, the task may be assigned sockets that other tasks are using or may use.

Return *task\_id* if successful. May raise exceptions related to opening the logfile, being unable to obtain enough resources to launch the task (`ipsExceptions.InsufficientResourcesException`), bad task launch request (`ipsExceptions.ResourceRequestMismatchException`, `ipsExceptions.BadResourceRequestException`) or problems executing the command. These exceptions may be used to retry launching the task as appropriate.

**Note:** This is a nonblocking function, users must use a version of `ServicesProxy.wait_task()` to get result.

ServicesProxy.**wait\_task**(*task\_id*)

Check the status of task *task\_id*. Return the return value of the task when finished successfully. Raise exceptions if the task is not found, or if there are problems finalizing the task.

ServicesProxy.**wait\_task\_nonblocking**(*task\_id*)

Check the status of task *task\_id*. If it has finished, the return value is populated with the actual value, otherwise `None` is returned. A `KeyError` exception may be raised if the task is not found.

ServicesProxy.**wait\_tasklist**(*task\_id\_list*, *block=True*)

Check the status of a list of tasks. If *block* is `True`, return a dictionary of return values when *all* tasks have completed. If *block* is `False`, return a dictionary containing entries for each *completed* task. Note that the dictionary may be empty. Raise `KeyError` exception if *task\_id* not found.



`ServicesProxy.kill_task(task_id)`

Kill launched task *task\_id*. Return if successful. Raises exceptions if the task or process cannot be found or killed successfully.

`ServicesProxy.kill_all_tasks()`

Kill all tasks associated with this component.

The task pool interface is designed for running a group of tasks that are independent of each other and can run concurrently. The services manage the execution of the tasks efficiently for the component. Users must first create an empty task pool, then add tasks to it. The tasks are submitted as a group and checked on as a group. This interface is basically a wrapper around the interface above for convenience.

`ServicesProxy.create_task_pool(task_pool_name)`

Create an empty pool of tasks with the name *task\_pool\_name*. Raise exception if duplicate name.

`ServicesProxy.add_task(task_pool_name, task_name, nproc, working_dir, binary, *args, **key-words)`

Add task *task\_name* to task pool *task\_pool\_name*. Remaining arguments are the same as in `ServicesProxy.launch_task()`.

`ServicesProxy.submit_tasks(task_pool_name, block=True)`

Launch all unfinished tasks in task pool *task\_pool\_name*. If *block* is `True`, return when all tasks have been launched. If *block* is `False`, return when all tasks that can be launched immediately have been launched. Return number of tasks submitted.

`ServicesProxy.get_finished_tasks(task_pool_name)`

Return dictionary of finished tasks and return values in task pool *task\_pool\_name*. Raise exception if no active or finished tasks.

`ServicesProxy.remove_task_pool(task_pool_name)`

Kill all running tasks, clean up all finished tasks, and delete task pool.

## Miscellaneous

The following services do not fit neatly into any of the other categories, but are important to the execution of the simulation.

`ServicesProxy.get_working_dir()`

Return the working directory of the calling component.

The structure of the working directory is defined using the configuration parameters *CLASS*, *SUB\_CLASS*, and *NAME* of the component configuration section. The structure of the working directory is:

`${SIM_ROOT}/work/${CLASS}_${SUB_CLASS}_${NAME}_<instance_num>`

`ServicesProxy.update_time_stamp(new_time_stamp=-1)`

Update time stamp on portal.

`ServicesProxy.send_portal_event(event_type='COMPONENT_EVENT', event_comment='')`

Send event to web portal.

## Data Management

The data management services are used by the components to manage the data needed and produced by each step, and for the driver to manage the overall simulation data. There are methods for component local, and simulation global files, as well as replay component file movements. Fault tolerance services are presented in another section.

Staging of local (non-shared) files:

`ServicesProxy.stage_input_files(input_file_list)`

Copy component input files to the component working directory (as obtained via a call to `ServicesProxy.get_working_dir()`). Input files are assumed to be originally located in the directory variable `INPUT_DIR` in the component configuration section.

`ServicesProxy.stage_output_files(timestamp, file_list, keep_old_files=True)`

Copy associated component output files (from the working directory) to the component simulation results directory. Output files are prefixed with the configuration parameter `OUTPUT_PREFIX`. The simulation results directory has the format:

```
${SIM_ROOT}/simulation_results/<timestamp>/components/${CLASS}_${SUB_CLASS}_${NAME}_${SEQ_NUM}
```

Additionally, plasma state files are archived for debugging purposes:

```
${SIM_ROOT}/history/plasma_state/<file_name>_${CLASS}_${SUB_CLASS}_${NAME}<timestamp>
```

Copying errors are not fatal (exception raised).

Staging of global (plasma state) files:

`ServicesProxy.stage_plasma_state()`

Copy current plasma state to work directory.

`ServicesProxy.update_plasma_state(plasma_state_files=None)`

Copy local (updated) plasma state to global state. If no plasma state files are specified, component configuration specification is used. Raise exceptions upon copy.

`ServicesProxy.merge_current_plasma_state(partial_state_file, logfile=None)`

Merge partial plasma state with global state. Partial plasma state contains only the values that the component contributes to the simulation. Raise exceptions on bad merge. Optional *logfile* will capture `stdout` from merge.

Staging of replay files:

`ServicesProxy.stage_replay_output_files(timestamp)`

Copy output files from the replay component to current sim for physics time *timestamp*. Return location of new local copies.

`ServicesProxy.stage_replay_plasma_files(timestamp)`

Copy plasma state files from the replay component to current sim for physics time *timestamp*. Return location of new local copies.

## Configuration Parameter Access

These methods access information from the simulation configuration file.

`ServicesProxy.get_port(port_name)`

Return a reference to the component implementing port *port\_name*.

`ServicesProxy.get_config_param(param)`

Return the value of the configuration parameter *param*. Raise exception if not found.

`ServicesProxy.set_config_param(param, value, target_sim_name=None)`

Set configuration parameter *param* to *value*. Raise exceptions if the parameter cannot be changed or if there are problems setting the value.

`ServicesProxy.get_time_loop()`

Return the list of times as specified in the configuration file.

## Logging

The following logging methods can be used to write logging messages to the simulation log file. It is *strongly* recommended that these methods are used as opposed to print statements. The logging capability adds a timestamp and identifies the component that generated the message. The syntax for logging is a simple string or formatted string:

```
self.services.info('beginning step')
self.services.warning('unable to open log file %s for task %d, will use stdout instead',
                    logfile, task_id)
```

There is no need to include information about the component in the message as the IPS logging interface includes a time stamp and information about what component sent the message:

```
2011-06-13 14:17:48,118 drivers_ssfoley_branch_test_driver_1 DEBUG    __initialize__(): <branch_test
2011-06-13 14:17:48,125 drivers_ssfoley_branch_test_driver_1 DEBUG    Working directory /scratch/scr
2011-06-13 14:17:48,129 drivers_ssfoley_branch_test_driver_1 DEBUG    Running - CompID = branch_test
2011-06-13 14:17:48,130 drivers_ssfoley_branch_test_driver_1 DEBUG    _init_event_service(): self.co
2011-06-13 14:17:51,934 drivers_ssfoley_branch_test_driver_1 INFO     ('Received Message ',)
2011-06-13 14:17:51,934 drivers_ssfoley_branch_test_driver_1 DEBUG    Calling method init args = (0,
2011-06-13 14:17:51,938 drivers_ssfoley_branch_test_driver_1 INFO     ('Received Message ',)
2011-06-13 14:17:51,938 drivers_ssfoley_branch_test_driver_1 DEBUG    Calling method step args = (0,
2011-06-13 14:17:51,939 drivers_ssfoley_branch_test_driver_1 DEBUG    _invoke_service(): init_task
2011-06-13 14:17:51,939 drivers_ssfoley_branch_test_driver_1 DEBUG    _get_service_response(REQUEST|
2011-06-13 14:17:51,952 drivers_ssfoley_branch_test_driver_1 DEBUG    _get_service_response(REQUEST|
2011-06-13 14:17:51,954 drivers_ssfoley_branch_test_driver_1 DEBUG    Launching command : aprun -n 4
2011-06-13 14:17:51,961 drivers_ssfoley_branch_test_driver_1 DEBUG    _invoke_service(): getTopic (
2011-06-13 14:17:51,962 drivers_ssfoley_branch_test_driver_1 DEBUG    _get_service_response(REQUEST|
2011-06-13 14:17:51,972 drivers_ssfoley_branch_test_driver_1 DEBUG    _get_service_response(REQUEST|
2011-06-13 14:17:51,972 drivers_ssfoley_branch_test_driver_1 DEBUG    _invoke_service(): sendEvent
2011-06-13 14:17:51,973 drivers_ssfoley_branch_test_driver_1 DEBUG    _get_service_response(REQUEST|
2011-06-13 14:17:51,984 drivers_ssfoley_branch_test_driver_1 DEBUG    _get_service_response(REQUEST|
2011-06-13 14:17:51,987 drivers_ssfoley_branch_test_driver_1 DEBUG    _invoke_service(): getTopic (
2011-06-13 14:17:51,988 drivers_ssfoley_branch_test_driver_1 DEBUG    _get_service_response(REQUEST|
2011-06-13 14:17:52,000 drivers_ssfoley_branch_test_driver_1 DEBUG    _get_service_response(REQUEST|
2011-06-13 14:17:52,000 drivers_ssfoley_branch_test_driver_1 DEBUG    _invoke_service(): sendEvent
2011-06-13 14:17:52,000 drivers_ssfoley_branch_test_driver_1 DEBUG    _get_service_response(REQUEST|
2011-06-13 14:17:52,012 drivers_ssfoley_branch_test_driver_1 DEBUG    _get_service_response(REQUEST|
2011-06-13 14:17:52,012 drivers_ssfoley_branch_test_driver_1 DEBUG    _invoke_service(): finish_task
```

The table below describes the levels of logging available and when to use each one. These levels are also used to determine what messages are produced in the log file. The default level is WARNING, thus you will see WARNING, ERROR and CRITICAL messages in the log file.

Level	When it's used
DE- BUG	Detailed information, typically of interest only when diagnosing problems.
INFO	Confirmation that things are working as expected.
WARN- ING	An indication that something unexpected happened, or indicative of some problem in the near future (e.g. "disk space low"). The software is still working as expected.
ER- ROR	Due to a more serious problem, the software has not been able to perform some function.
CRITI- CAL	A serious error, indicating that the program itself may be unable to continue running.

For more information about the logging module and how to use it, see [Logging Tutorial](#).

```
ServicesProxy.log(*args)
    Wrapper for ServicesProxy.info().
```

`ServicesProxy.debug(*args)`

Produce **debugging** message in simulation log file. Raise exception for bad formatting.

`ServicesProxy.info(*args)`

Produce **informational** message in simulation log file. Raise exception for bad formatting.

`ServicesProxy.warning(*args)`

Produce **warning** message in simulation log file. Raise exception for bad formatting.

`ServicesProxy.error(*args)`

Produce **error** message in simulation log file. Raise exception for bad formatting.

`ServicesProxy.exception(*args)`

Produce **exception** message in simulation log file. Raise exception for bad formatting.

`ServicesProxy.critical(*args)`

Produce **critical** message in simulation log file. Raise exception for bad formatting.

## Fault Tolerance

The IPS provides services to checkpoint and restart a coupled simulation by calling the checkpoint and restart methods of each component and certain settings in the configuration file. The driver can call *checkpoint\_components*, which will invoke the checkpoint method on each component associated with the simulation. The component's *checkpoint* method uses *save\_restart\_files* to save files needed by the component to restart from the same point in the simulation. When the simulation is in restart mode, the *restart* method of the component is called to initialize the component, instead of the *init* method. The *restart* component method uses the *get\_restart\_files* method to stage in inputs for continuing the simulation.

`ServicesProxy.save_restart_files(timestamp, file_list)`

Copy files needed for component restart to the restart directory:

```
${SIM_ROOT}/restart/${timestamp}/components/${CLASS}_${SUB_CLASS}_${NAME}
```

Copying errors are not fatal (exception raised).

`ServicesProxy.checkpoint_components(comp_id_list, time_stamp, Force=False, Protect=False)`

Selectively checkpoint components in *comp\_id\_list* based on the configuration section *CHECKPOINT*. If *Force* is *True*, the checkpoint will be taken even if the conditions for taking the checkpoint are not met. If *Protect* is *True*, then the data from the checkpoint is protected from clean up. *Force* and *Protect* are optional and default to *False*.

The *CHECKPOINT\_MODE* option controls determines if the components checkpoint methods are invoked.

Possible *MODE* options are:

**WALLTIME\_REGULAR:** checkpoints are saved upon invocation of the service call *checkpoint\_components()*, when a time interval greater than, or equal to, the value of the configuration parameter *WALLTIME\_INTERVAL* had passed since the last checkpoint. A checkpoint is assumed to have happened (but not actually stored) when the simulation starts. Calls to *checkpoint\_components()* before *WALLTIME\_INTERVAL* seconds have passed since the last successful checkpoint result in a NOOP.

**WALLTIME\_EXPLICIT:** checkpoints are saved when the simulation wall clock time exceeds one of the (ordered) list of time values (in seconds) specified in the variable *WALLTIME\_VALUES*. Let  $[t_0, t_1, \dots, t_n]$  be the list of wall clock time values specified in the configuration parameter *WALLTIME\_VALUES*. Then  $\text{checkpoint}(T) = \text{True}$  if  $T \geq t_j$ , for some  $j$  in  $[0, n]$  and there is no other time  $T_1$ , with  $T > T_1 \geq t_j$  such that  $\text{checkpoint}(T_1) = \text{True}$ . If the test fails, the call results in a NOOP.

**PHYSTIME\_REGULAR:** checkpoints are saved at regularly spaced “physics time” intervals, specified in the configuration parameter *PHYSTIME\_INTERVAL*. Let *PHYSTIME\_INTERVAL* = *PTI*, and the physics

time stamp argument in the call to `checkpoint_components()` be `pts_i`, with  $i = 0, 1, 2, \dots$ . Then `checkpoint(pts_i) = True` if  $pts_i \geq n \cdot PTI$ , for some  $n$  in  $1, 2, 3, \dots$  and  $pts_i - pts_{prev} \geq PTI$ , where `checkpoint(pts_prev) = True` and  $pts_{prev} = \max(pts_0, pts_1, \dots, pts_{i-1})$ . If the test fails, the call results in a NOOP.

**PHYSTIME\_EXPLICIT:** checkpoints are saved when the physics time equals or exceeds one of the (ordered) list of physics time values (in seconds) specified in the variable `PHYSTIME_VALUES`. Let  $[pt_0, pt_1, \dots, pt_n]$  be the list of physics time values specified in the configuration parameter `PHYSTIME_VALUES`. Then `checkpoint(pt) = True` if  $pt \geq pt_j$ , for some  $j$  in  $[0, n]$  and there is no other physics time  $pt_k$ , with  $pt > pt_k \geq pt_j$  such that `checkpoint(pt_k) = True`. If the test fails, the call results in a NOOP.

The configuration parameter `NUM_CHECKPOINT` controls how many checkpoints to keep on disk. Checkpoints are deleted in a FIFO manner, based on their creation time. Possible values of `NUM_CHECKPOINT` are:

- `NUM_CHECKPOINT = n`, with  $n > 0 \rightarrow$  Keep the most recent  $n$  checkpoints
- `NUM_CHECKPOINT = 0 \rightarrow` No checkpoints are made/kept (except when *Force* = True)
- `NUM_CHECKPOINT < 0 \rightarrow` Keep ALL checkpoints

Checkpoints are saved in the directory `${SIM_ROOT}/restart`

`ServicesProxy.get_restart_files(restart_root, timeStamp, file_list)`

Copy files needed for component restart from the restart directory:

```
<restart_root>/restart/<timeStamp>/components/${CLASS}_${SUB_CLASS}_${NAME}_${SEQ_NUM}
```

to the component's work directory.

Copying errors are not fatal (exception raised).

## Event Service

The event service interface is used to implement the web portal connection, as well as for components to communicate asynchronously. See the [Advanced Features](#) documentation for details on how to use this interface for component communication.

`ServicesProxy.publish(topicName, eventName, eventBody)`

Publish event consisting of *eventName* and *eventBody* to topic *topicName* to the IPS event service.

`ServicesProxy.subscribe(topicName, callback)`

Subscribe to topic *topicName* on the IPS event service and register *callback* as the method to be invoked when an event is published to that topic.

`ServicesProxy.unsubscribe(topicName)`

Remove subscription to topic *topicName*.

`ServicesProxy.process_events()`

Poll for events on subscribed topics.

## 4.3 The Configuration File - Explained

This section will detail the different sections and fields of the configuration file and how they relate to a simulation. The configuration file is designed to let the user to easily set data items used by the framework, components, tasks, and the portal from run to run. There are user specific, platform specific, and component specific entries that need to be modified or verified before running the IPS in the given configuration. After a short overview of the syntax of

the package used by the framework to make sense of the configuration file, a detailed explanation of each line of the configuration file is presented.

### 4.3.1 Syntax and the ConfigObj module

`ConfigObj` is a Python package for reading and writing config files. The syntax is similar to shell syntax (e.g., use of `$` to reference variables), uses square brackets to create named sections and nested subsections, comma-separated lists and comments indicated by a `#`.

In the example configuration file below, curly braces (`{ }`) are used to clarify references to variables with underscores (`_`). Any left-hand side value can be used as a variable after it is defined. Additionally, any platform configuration value can be referenced as a variable in the configuration file as well.

### 4.3.2 Configuration File - Line by Line

**Platform Configuration Override Section** It is possible for the configuration file to override entries in the platform configuration file. It is rare and users should use caution when overriding these values. See *Platform Configuration File - Explained* for details on these values.

```
#HOST =
#MPIRUN =
#PHYS_BIN_ROOT =
#DATA_TREE_ROOT =
#PORTAL_URL =
#RUNID_URL =
#NODE_ALLOCATION_MODE =
```

#### User Data Section

The following items are specific to the user and should be changed accordingly. They will help you to identify your runs in the portal (*USER*), and also store the data from your runs in particular web-enabled locations for post-processing (*USER\_W3\_DIR* on the local machine, *USER\_W3\_BASEURL* on the portal). All of the items in this section are optional.

```
USER_W3_DIR = /project/projectdirs/m876/www/ssfoley
USER_W3_BASEURL = http://portal.nersc.gov/project/m876/ssfoley
USER = ssfoley # Optional, if missing the unix username is used
```

**Simulation Information Section** These items describe this configuration and is used for describing and locating its output, information for the portal, and location of the source code of the IPS.

**\*\* Mandatory items:** *SIM\_ROOT*, *SIM\_NAME*, *LOG\_FILE*, *RUN\_COMMENT*, *IPS\_ROOT*

*RUN\_ID*, *TOKOMAK\_ID*, *SHOT\_NUMBER* - identifiers for the simulation that are helpful for SWIM users. They are often used to form a hierarchical name for the simulation, identifying related runs.

*OUTPUT\_PREFIX* - used to prevent collisions and overwriting of different simulations using the same *SIM\_ROOT*.

*SIM\_NAME* - used to identify the simulation on the portal, and often to name the output tree.

*LOG\_FILE* - name of the log file for this simulation. The framework log file is specified at the command line.

*LOG\_LEVEL* - sets the logging level for the simulation. If empty, the framework log level is used, which defaults to *WARNING*. See *Logging* for details on the logging capabilities in the IPS. Possible values: *DEBUG*, *INFO*, *WARNING*, *ERROR*, *EXCEPTION*, *CRITICAL*.

*IPS\_ROOT* - location of the ips source code that will be used.

*SIM\_ROOT* - location of output tree. This directory will be created if it does not exist. If the directory already exists, then data files will be added, possibly overwriting existing data.

```
RUN_ID = Model_seq           # Identifier for this simulation run
TOKAMAK_ID = ITER
SHOT_NUMBER = 1             # Identifier for specific case for this tokamak
                              # (should be character integer)

SIM_NAME = ${RUN_ID}_${TOKAMAK_ID}_${SHOT_NUMBER}

OUTPUT_PREFIX =
LOG_FILE = ${RUN_ID}_sim.log
LOG_LEVEL = DEBUG           # Default = WARNING

# Root of IPS (contains framework and component source code)
IPS_ROOT = /scratch/scratchdirs/ssfoley/ips

# Simulation root - path of the simulation directory that will be constructed
# by the framework
SIM_ROOT = /scratch/scratchdirs/ssfoley/seq_example

# Description of the simulation for the portal
SIMULATION_DESCRIPTION = sequential model simulation using generic driver.py
RUN_COMMENT = sequential model simulation using generic driver.py
TAG = sequential_model      # for grouping related runs
```

### Simulation Mode

This section describes the mode in which to run the simulation. All values are optional.

*SIMULATION\_MODE* - describes whether the simulation is starting from *init* (*NORMAL*) or restarting from a check-point (*RESTART*). The default is *NORMAL*. For *RESTART*, a restart time and directory must be specified. These values are used by the driver to control how the simulation is initialized. *RESTART\_TIME* must coincide with a check-point save time. *RESTART\_DIRECTORY* may be *\$SIM\_ROOT* if there is an existing current simulation there, and the new work will be appended, such that it looks like a seamless simulation.

*NODE\_ALLOCATION\_MODE* - sets the default execution mode for tasks in this simulation. If the value is *EXCLUSIVE*, then tasks are assigned whole nodes. If the value is *SHARED*, sub-node allocation is used so tasks can shared nodes thus using the allocation more efficiently. It is the users responsibility to understand how node sharing will impact the performance of their tasks.

```
SIMULATION_MODE = NORMAL    # NORMAL | RESTART
RESTART_TIME = 12           # time step to restart from
RESTART_ROOT = ${SIM_ROOT}
NODE_ALLOCATION_MODE = EXCLUSIVE # SHARED | EXCLUSIVE
```

### Plasma State Section

The locations and names of the plasma state files are specified here, along with the directory where the global plasma state files are located in the simulation tree. It is common to specify groups of plasma state files for use in the component configuration sections. These files should contain all the shared data values for the simulation so that they can be managed by the driver.

```
PLASMA_STATE_WORK_DIR = ${SIM_ROOT}/work/plasma_state

# Config variables defining simulation specific names for plasma state files
CURRENT_STATE = ${SIM_NAME}_ps.cdf
PRIOR_STATE = ${SIM_NAME}_psp.cdf
NEXT_STATE = ${SIM_NAME}_psn.cdf
CURRENT_EQDSK = ${SIM_NAME}_ps.geq
```

```
CURRENT_CQL = ${SIM_NAME}_ps_CQL.dat
CURRENT_DQL = ${SIM_NAME}_ps_DQL.nc
CURRENT_JSDSK = ${SIM_NAME}_ps.jsd

# List of files that constitute the plasma state
PLASMA_STATE_FILES1 = ${CURRENT_STATE} ${PRIOR_STATE} ${NEXT_STATE} ${CURRENT_EQDSK}
PLASMA_STATE_FILES2 = ${CURRENT_CQL} ${CURRENT_DQL} ${CURRENT_JSDSK}
PLASMA_STATE_FILES = ${PLASMA_STATE_FILES1} ${PLASMA_STATE_FILES2}
```

### Ports Section

The ports section identifies which ports and their associated implementations that are to be used for this simulation. The ports section is defined by [PORTS]. *NAMES* is a list of port names, where each needs to appear as a subsection (e.g., [[DRIVER]]). Each port definition section must contain the entry *IMPLEMENTATION* whose value is the name of a component definition section. These are case sensitive names and should be named such that someone familiar the components of this project has an understanding of what is being modeled. The only mandatory port is *DRIVER*. It should be named *DRIVER*, but the implementation can be anything, as long as it is defined. If no *INIT* port is defined, then the framework will produce a warning to that effect. There may be more port definitions than listed in *NAMES*.

```
[PORTS]
  NAMES = INIT DRIVER MONITOR EPA RF_IC NB FUS

# Required ports - DRIVER and INIT
  [[DRIVER]]
    IMPLEMENTATION = GENERIC_DRIVER

  [[INIT]]
    IMPLEMENTATION = minimal_state_init

# Physics ports

  [[RF_IC]]
    IMPLEMENTATION = model_RF_IC

  [[FP]]
    IMPLEMENTATION = minority_model_FP

  [[FUS]]
    IMPLEMENTATION = model_FUS

  [[NB]]
    IMPLEMENTATION = model_NB

  [[EPA]]
    IMPLEMENTATION = model_EPA

  [[MONITOR]]
    IMPLEMENTATION = monitor_comp_4
```

### Component Configuration Section

Component definition and configuration is done in this “section.” Each component configuration section is defined as a section (e.g., [model\_RF\_IC]). Each entry in the component configuration section is available to the component at runtime using that name (e.g., *self.NPROC*), thus these values can be used to create specific simulation cases using generic components. Variables defined within a component configuration section are local to that section, but values may be defined in terms of the simulation values defined above (e.g., *PLASMA\_STATE\_FILES*, and *IPS\_ROOT*).

**\*\* Mandatory entries: *SCRIPT*, *NAME*, *BIN\_PATH*, *INPUT\_DIR***



*CLASS* - commonly this is the port name or the first directory name in the path to the component implementation in `ips/components/`.

*SUB\_CLASS* - commonly this is the name of the code or method used to model this port, or the second directory name in the path to the component implementation in `ips/components/`.

*NAME* - name of the class in the Python script that implements this component.

*NPROC* - number of processes on which to launch tasks.

*BIN\_PATH* - path to script and any other helper scripts and binaries. This is used by the framework and component to find and execute helper scripts and binaries.

*BINARY* - the binary to launch as a task. Typically, these binaries are found in the

*PHYS\_BIN* or some subdirectory therein. Otherwise, you can make your own variable and put the directory where the binary is located there.

*INPUT\_DIR* - directory where the input files (listed below) are found. This is used during initialization to copy the input files to the work directory of the component.

*INPUT\_FILES* - list of files (relative to *INPUT\_DIR*) that need to be copied to the component work directory on initialization.

*OUTPUT\_FILES* - list of output files that are produced that need to be protected and archived on a call to `services.ServicesProxy.stage_output_files()`.

*PLASMA\_STATE\_FILES* - list of plasma state files used and modified by this component. If not present, then the files specified in the simulation entry *PLASMA\_STATE\_FILES* is used.

*RESTART\_FILES* - list of files that need to be archived as the checkpoint of this component.

*NODE\_ALLOCATION\_MODE* - sets the default execution mode for tasks in this component. If the value is *EXCLUSIVE*, then tasks are assigned whole nodes. If the value is *SHARED*, sub-node allocation is used so tasks can share nodes thus using the allocation more efficiently. If no value or entry is present, the simulation value for *NODE\_ALLOCATION\_MODE* is used. It is the users responsibility to understand how node sharing will impact the performance of their tasks. This can be overridden using the *whole\_nodes* and *whole\_sockets* arguments to `services.ServicesProxy.launch_task()`.

Additional values that are specific to the component may be added as needed, for example certain data values like *PPN*, paths to and names of other executables used by the component or alternate *NPROC* values are examples. It is the responsibility of the component writer to make sure users know what values are required by the component and what the valid values are for each.

```
[model_epa]
    CLASS = epa
    SUB_CLASS = model_epa
    NAME = model_epa
    NPROC = 1
    BIN_PATH = ${IPS_ROOT}/bin
    INPUT_DIR = ${DATA_TREE_ROOT}/model_epa/ITER/hy040510/t20.0
    INPUT_STATE_FILE = hy040510_002_ps_epa__tsc_4_20.000.cdf
    INPUT_EQDSK_FILE = hy040510_002_ps_epa__tsc_4_20.000.geq
    INPUT_FILES = model_epa_input.nml ${INPUT_STATE_FILE} ${INPUT_EQDSK_FILE}
    OUTPUT_FILES = internal_state_data.nml
    PLASMA_STATE_FILES = ${CURRENT_STATE} ${NEXT_STATE} ${CURRENT_EQDSK}
    RESTART_FILES = ${INPUT_FILES} internal_state_data.nml
    SCRIPT = ${BIN_PATH}/model_epa_ps_file_init.py

[monitor_comp_4]
    CLASS = monitor
    SUB_CLASS =
```

```
NAME = monitor
NPROC = 1
W3_DIR = ${USER_W3_DIR}           # Note this is user specific
W3_BASEURL = ${USER_W3_BASEURL}   # Note this is user specific
TEMPLATE_FILE= basic_time_traces.xml
BIN_PATH = ${IPS_ROOT}/bin
INPUT_DIR = ${IPS_ROOT}/components/monitor/monitor_4
INPUT_FILES = basic_time_traces.xml
OUTPUT_FILES = monitor_file.nc
PLASMA_STATE_FILES = ${CURRENT_STATE}
RESTART_FILES = ${INPUT_FILES} monitor_restart monitor_file.nc
SCRIPT = ${BIN_PATH}/monitor_comp.py
```

## Checkpoint Section

This section describes when checkpoints should be taken by the simulation. Drivers should be written such that at the end of each step there is a call to `services.ServicesProxy.checkpoint_components()`. This way the services use the settings in this section to either take a checkpoint or not.

Selectively checkpoint components in *comp\_id\_list* based on the configuration section *CHECKPOINT*. If *Force* is True, the checkpoint will be taken even if the conditions for taking the checkpoint are not met. If *Protect* is True, then the data from the checkpoint is protected from clean up. *Force* and *Protect* are optional and default to False.

The *CHECKPOINT\_MODE* option controls determines if the components checkpoint methods are invoked. Possible *MODE* options are:

**WALLTIME\_REGULAR:** checkpoints are saved upon invocation of the service call `checkpoint_components()`, when a time interval greater than, or equal to, the value of the configuration parameter *WALLTIME\_INTERVAL* had passed since the last checkpoint. A checkpoint is assumed to have happened (but not actually stored) when the simulation starts. Calls to `checkpoint_components()` before *WALLTIME\_INTERVAL* seconds have passed since the last successful checkpoint result in a NOOP.

**WALLTIME\_EXPLICIT:** checkpoints are saved when the simulation wall clock time exceeds one of the (ordered) list of time values (in seconds) specified in the variable *WALLTIME\_VALUES*. Let  $[t_0, t_1, \dots, t_n]$  be the list of wall clock time values specified in the configuration parameter *WALLTIME\_VALUES*. Then `checkpoint(T) = True` if  $T \geq t_j$ , for some  $j$  in  $[0, n]$  and there is no other time  $T_1$ , with  $T > T_1 \geq t_j$  such that `checkpoint(T_1) = True`. If the test fails, the call results in a NOOP.

**PHYSTIME\_REGULAR:** checkpoints are saved at regularly spaced “physics time” intervals, specified in the configuration parameter *PHYSTIME\_INTERVAL*. Let *PHYSTIME\_INTERVAL* = PTI, and the physics time stamp argument in the call to `checkpoint_components()` be *pts\_i*, with  $i = 0, 1, 2, \dots$ . Then `checkpoint(pts_i) = True` if  $pts_i \geq n \cdot PTI$ , for some  $n$  in  $1, 2, 3, \dots$  and  $pts_i - pts_{prev} \geq PTI$ , where `checkpoint(pts_prev) = True` and  $pts_{prev} = \max(pts_0, pts_1, \dots, pts_{i-1})$ . If the test fails, the call results in a NOOP.

**PHYSTIME\_EXPLICIT:** checkpoints are saved when the physics time equals or exceeds one of the (ordered) list of physics time values (in seconds) specified in the variable *PHYSTIME\_VALUES*. Let  $[pt_0, pt_1, \dots, pt_n]$  be the list of physics time values specified in the configuration parameter *PHYSTIME\_VALUES*. Then `checkpoint(pt) = True` if  $pt \geq pt_j$ , for some  $j$  in  $[0, n]$  and there is no other physics time  $pt_k$ , with  $pt > pt_k \geq pt_j$  such that `checkpoint(pt_k) = True`. If the test fails, the call results in a NOOP.

The configuration parameter *NUM\_CHECKPOINT* controls how many checkpoints to keep on disk. Checkpoints are deleted in a FIFO manner, based on their creation time. Possible values of *NUM\_CHECKPOINT* are:

- *NUM\_CHECKPOINT* =  $n$ , with  $n > 0$  → Keep the most recent  $n$  checkpoints
- *NUM\_CHECKPOINT* = 0 → No checkpoints are made/kept (except when *Force* = True)
- *NUM\_CHECKPOINT* < 0 → Keep ALL checkpoints

Checkpoints are saved in the directory `${SIM_ROOT}/restart`

```
[CHECKPOINT]
MODE = WALLTIME_REGULAR
WALLTIME_INTERVAL = 15
NUM_CHECKPOINT = 2
PROTECT_FREQUENCY = 5
```

### Time Loop Section

The time loop specifies how time progresses for the simulation in the driver. It is not required by the framework, but may be required by the driver. Most simulations use the time loop section to specify the number and frequency of time steps for the simulation as opposed to hard coding it into the driver. It is a helpful tool to control the runtime of each step and the overall simulation. It can also be helpful when looking at a small portion of time in the simulation for debugging purposes.

*MODE* - defines the following entries. If mode is *REGULAR* – *START*, *FINISH* and *NSTEP* are used to generate a list of times of length *NSTEP* starting at *START* and ending at *FINISH*. If mode is *EXPLICIT* – *VALUES* contains the (whitespace separated) list of times that are to be modeled.

```
[TIME_LOOP]
MODE = REGULAR
START = 0.0
FINISH = 20.0
NSTEP = 5
```

## 4.4 Platforms and Platform Configuration

This section will describe key aspects of the platforms that the IPS has been ported to, key locations relevant to the IPS, and the platform configuration settings in general and specific to the platforms described below.

**Important Note** - while this documentation is intended to remain up to date, it may not always reflect the current status of the machines. If you run into problems, check that the information below is accurate by looking at the websites for the machine. If you are still having problems, contact the framework developers.

### 4.4.1 Ported Platforms

Each subsection will contain information about the platform in question. If you are porting the IPS to a new platform, these are the items that you will need to know or files and directories to create in order to port the IPS. You will also need a platform configuration file (*described below*). Available queue names are listed with the most common ones in **bold**.

The platforms below fall into the following categories:

- general production machines - large production machines on which the majority of runs (particularly production runs) are made.
- experimental systems - production or shared machines that are being used by a subset of SWIM members for specific research projects. These systems may also be difficult for others to get accounts.
- formerly used systems - machines that the IPS was ported to but we either do not have time on that machine, it has been retired by its hosting site, or it is not in wide use anymore.
- single user systems - laptop or desktop machines for testing small problems.

## General Production

### Franklin

Franklin is a Cray XT4 managed by [NERSC](#).

- Account: You must have an account at NERSC and be added to the SWIM project's group (m876) to log on and access the set of physics binaries in the *PHYS\_BIN*.
- Logging on - `ssh franklin.nersc.gov -l <username>`
- Architecture - 9,572 nodes, 4 cores per node, 8 GB memory per node
- Environment:
  - OS - Cray Linux Environment (CLE)
  - Batch scheduler/Resource Manager - PBS, Moab
  - [Queues](#) - **debug**, **regular**, low, premium, interactive, xfer, iotask, special
  - Parallel Launcher (e.g., mpirun) - aprun
  - Node Allocation policy - exclusive node allocation
- Project directory - `/project/projectdirs/m876/`
- Data Tree - `/project/projectdirs/m876/data/`
- Physics Binaries - `/project/projectdirs/m876/phys-bin/phys/`
- WWW Root - `/project/projectdirs/m876/www/<username>`
- WWW Base URL - `http://portal.nersc.gov/project/m876/<username>`

### Hopper

Hopper is a Cray XE6 managed by [NERSC](#).

- Account: You must have an account at NERSC and be added to the SWIM project's group (m876) to log on and access the set of physics binaries in the *PHYS\_BIN*.
- Logging on - `ssh hopper.nersc.gov -l <username>`
- Architecture - 6384 nodes, 24 cores per node, 32 GB memory per node
- Environment:
  - OS - Cray Linux Environment (CLE)
  - Batch scheduler/Resource Manager - PBS, Moab
  - [Queues](#) - **debug**, **regular**, low, premium, interactive
  - Parallel Launcher (e.g., mpirun) - aprun
  - Node Allocation policy - exclusive node allocation
- Project directory - `/project/projectdirs/m876/`
- Data Tree - `/project/projectdirs/m876/data/`
- Physics Binaries - `/project/projectdirs/m876/phys-bin/phys/`
- WWW Root - `/project/projectdirs/m876/www/<username>`

- WWW Base URL - `http://portal.nersc.gov/project/m876/<username>`

## Stix

Stix is a SMP hosted at PPPL.

- Account: You must have an account at PPPL to access their Beowulf systems.
- Logging on:
  1. Log on to the PPPL vpn (<https://vpn.pppl.gov>)
  2. `ssh <username>@portal.pppl.gov`
  3. `ssh portalr5`
- Architecture - 80 cores, 440 GB memory
- Environment:
  - OS - linux
  - Batch scheduler/Resource Manager - PBS (Torque), Moab
  - Queues - **smppq** (this is how you specify that you want to run your job on stix)
  - Parallel Launcher (e.g., mpirun) - mpiexec (MPICH2)
  - Node Allocation policy - node sharing allowed (whole machine looks like one node)
- Project directory - `/p/swim1/`
- Data Tree - `/p/swim1/data/`
- Physics Binaries - `/p/swim1/phys/`
- WWW Root - `/p/swim/w3_html/<username>`
- WWW Base URL - `http://w3.pppl.gov/swim/<username>`

## Experimental Systems

### Swim

Swim is a SMP hosted by the [fusion theory group](#) at ORNL.

- Account: You must have an account at ORNL and be given an account on the machine.
- Logging on - `ssh swim.ornl.gov -l <username>`
- Architecture - ? cores, ? GB memory
- Environment:
  - OS - linux
  - Batch scheduler/Resource Manager - None
  - Parallel Launcher (e.g., mpirun) - mpirun (OpenMPI)
  - Node Allocation policy - node sharing allowed (whole machine looks like one node)
- Project directory - None
- Data Tree - None

- Physics Binaries - None
- WWW Root - None
- WWW Base URL - None

## Pacman

Pacman is a linux cluster hosted at [ARSC](#).

- Account: You must have an account to log on and use the system.
- Logging on - ?
- Architecture:
  - 88 nodes, 16 cores per node, 64 GB per node
  - 44 nodes, 12 cores per node, 32 GB per node
- Environment:
  - OS - Red Hat Linux 5.6
  - Batch scheduler/Resource Manager - Torque (PBS), Moab
  - [Queues](#) - debug, standard, standard\_12, standard\_16, bigmem, gpu, background, shared, transfer
  - Parallel Launcher (e.g., mpirun) - mpirun (OpenMPI?)
  - Node Allocation policy - node sharing allowed
- Project directory - ?
- Data Tree - ?
- Physics Binaries - ?
- WWW Root - ?
- WWW Base URL - ?

## Iter

Iter is a linux cluster (?) that is hosted ???.

- Account: You must have an account to log on and use the system.
- Logging on - ?
- Architecture - ? nodes, ? cores per node, ? GB memory per node
- Environment:
  - OS - linux
  - Batch scheduler/Resource Manager - ?
  - Queues - ?
  - Parallel Launcher (e.g., mpirun) - mpiexec (MPICH2)
  - Node Allocation policy - node sharing allowed
- Project directory - /project/projectdirs/m876/

- Data Tree - /project/projectdirs/m876/data/
- Physics Binaries - /project/projectdirs/m876/phys-bin/phys/
- WWW Root - ?
- WWW Base URL - ?

## Odin

Odin is a linux cluster hosted at [Indiana University](#).

- Account: You must have an account to log on and use the system.
- Logging on - `ssh odin.cs.indiana.edu -l <username>`
- Architecture - 128 nodes, 4 cores per node, ? GB memory per node
- Environment:
  - OS - GNU/Linux
  - Batch scheduler/Resource Manager - Slurm, Maui
  - Queues - there is only one queue and it does not need to specified in the batchscript
  - Parallel Launcher (e.g., mpirun) - mpirun (OpenMPI)
  - Node Allocation policy - node sharing allowed
- Project directory - None
- Data Tree - None
- Physics Binaries - None
- WWW Root - None
- WWW Base URL - None

## Sif

Sif is a linux cluster hosted at [Indiana University](#).

- Account: You must have an account to log on and use the system.
- Logging on - `ssh sif.cs.indiana.edu -l <username>`
- Architecture - 8 nodes, 8 cores per node, ? GB memory per node
- Environment:
  - OS - GNU/Linux
  - Batch scheduler/Resource Manager - Slurm, Maui
  - Queues - there is only one queue and it does not need to specified in the batchscript
  - Parallel Launcher (e.g., mpirun) - mpirun (OpenMPI)
  - Node Allocation policy - node sharing allowed
- Project directory - None
- Data Tree - None

- Physics Binaries - None
- WWW Root - None
- WWW Base URL - None

### Retired/Formerly Used Systems

#### Viz/Mhd

Viz/mhd are SMP machines hosted at PPPL. These systems appear not to be online any more.

- Account: You must have an account at PPPL to access their Beowulf systems.
- Logging on:
  1. Log on to the PPPL vpn (<https://vpn.pppl.gov>)
  2. `ssh <username>@portal.pppl.gov`
- Architecture - ? cores, ? GB memory
- Environment:
  - OS - linux
  - Batch scheduler/Resource Manager - PBS (Torque), Moab
  - Parallel Launcher (e.g., mpirun) - mpiexec (MPICH2)
  - Node Allocation policy - node sharing allowed (whole machine looks like one node)
- Project directory - /p/swim1/
- Data Tree - /p/swim1/data/
- Physics Binaries - /p/swim1/phys/
- WWW Root - /p/swim/w3\_html/<username>
- WWW Base URL - <http://w3.pppl.gov/swim/<username>>

#### Pingo

Pingo was a Cray XT5 hosted at ARSC.

- Account: You must have an account to log on and use the system.
- Logging on - ?
- Architecture - 432 nodes, 8 cores per node, ? memory per node
- Environment:
  - OS - ?
  - Batch scheduler/Resource Manager - ?
  - Parallel Launcher (e.g., mpirun) - aprun
  - Node Allocation policy - exclusive node allocation
- Project directory - ?
- Data Tree - ?



- Physics Binaries - ?
- WWW Root - ?
- WWW Base URL - ?

## Jaguar

Jaguar is a Cray XT5 managed by OLCF.

- Account: You must have an account for the OLCF and be added to the SWIM project group for accounting and files sharing purposes, if we have time on this machine.
- Logging on - `ssh jaguar.ornl.gov -l <username>`
- Architecture - 13,688 nodes, 12 cores per node, 16 GB memory per node
- Environment:
  - OS - Cray Linux Environment (CLE)
  - Batch scheduler/Resource Manager - PBS, Moab
  - Queues - debug, production
  - Parallel Launcher (e.g., mpirun) - aprun
  - Node Allocation policy - exclusive node allocation
- Project directory - ?
- Data Tree - ?
- Physics Binaries - ?
- WWW Root - ?
- WWW Base URL - ?

## Single User Systems

The IPS can be run on your laptop or desktop. Many of the items above are not present or relevant in a laptop/desktop environment. See the next section for a sample platform configuration settings.

### 4.4.2 Platform Configuration File

The platform configuration file contains platform specific information that the framework needs. Typically it does not need to be changed for one user to another or one run to another (except for manual specification of allocation resources). For *most* of the platforms above, you will find platform configuration files of the form `ips/<machine name>.conf`. It is not likely that you will need to change this file, but it is described here for users working on experimental machines, manual specification of resources, and users who need to port the IPS to a new machine.

```
HOST = franklin
MPIRUN = aprun
PHYS_BIN_ROOT = /project/projectdirs/m876/phys-bin/phys/
DATA_TREE_ROOT = /project/projectdirs/m876/data
DATA_ROOT = /project/projectdirs/m876/data/
PORTAL_URL = http://swim.gat.com:8080/monitor
RUNID_URL = http://swim.gat.com:4040/runid.esp
```

```
#####
# resource detection method
#####
NODE_DETECTION = checkjob # checkjob | qstat | pbs_env | slurm_env

#####
# manual allocation description
#####
TOTAL_PROCS = 16
NODES = 4
PROCS_PER_NODE = 4

#####
# node topology description
#####
CORES_PER_NODE = 4
SOCKETS_PER_NODE = 1

#####
# framework setting for node allocation
#####
# MUST ADHERE TO THE PLATFORM'S CAPABILITIES
# * EXCLUSIVE : only one task per node
# * SHARED : multiple tasks may share a node
# For single node jobs, this can be overridden allowing multiple
# tasks per node.
NODE_ALLOCATION_MODE = EXCLUSIVE # SHARED | EXCLUSIVE
```

**HOST** name of the platform. Used by the portal.

**MPIRUN** command to launch parallel applications. Used by the task manager to launch parallel tasks on compute nodes. If you would like to launch a task directly without the parallel launcher (say, on a SMP style machine or workstation), set this to “eval” – it tells the task manager to directly launch the task as `<binary> <args>`.

**\*\_ROOT** locations of data and binaries. Used by the configuration file and components to run the tasks of the simulation.

**\*\_URL** portal URLs. Used to connect to and communicate with the portal.

**NODE\_DETECTION** method to use to detect the number of nodes and processes in the allocation. If the value is “manual,” then the manual allocation description is used. If nothing is specified, all of the methods are attempted and the first one to succeed will be used. Note, if the allocation detection fails, the framework will abort, killing the job. See [Porting the IPS](#) for more information <sup>6</sup>.

**TOTAL\_PROCS** number of processes in the allocation <sup>7</sup>.

**NODES** number of nodes in the allocation <sup>3</sup>.

**PROCS\_PER\_NODE** number of processes per node (ppn) for the framework <sup>8</sup>.

**CORES\_PER\_NODE** number of cores per node <sup>9</sup>.

**SOCKETS\_PER\_NODE** number of sockets per node <sup>1</sup>.

---

<sup>6</sup> Currently the porting documentation is under construction. Use python script `ips/framework/utils/test_resource_parsing.py` to determine which automatic parsing works for the platform in question. If nothing works, use the manual settings and contact the framework developers to look into developing a method for automatically detecting the allocation.

<sup>7</sup> Only used if manual allocation is specified, or if no detection mechanism is specified and none of the other mechanisms work first. It is the users responsibility for this value to make sense.

<sup>8</sup> Used in manual allocation detection and will override any detected ppn value (if smaller than the machine maximum ppn).

<sup>9</sup> This value should not change unless the machine is upgraded to a different architecture or implements different allocation policies.

**NODE\_ALLOCATION\_MODE** ‘EXCLUSIVE’ for one task per node, and ‘SHARED’ if more than one task can share a node <sup>1</sup>. Simulations, components and tasks can set their node usage allocation policies in the configuration file and on task launch.

Due to the recent changes in the framework regarding resource management, some platforms may not have platform configuration files in the repository. Below is a list of those that are in the repo and work with the recent changes to the framework.

- franklin
- hopper
- odin
- sif
- stix <sup>10</sup>
- swim <sup>5</sup>

In addition to these files, there is `ips/workstation.conf`, a sample platform configuration file for a workstation. It assumes that the workstation:

- does not have a batch scheduler or resource manager
- may have multiple cores and sockets
- does not have portal access
- will manually specify the allocation

```
HOST = workstation
MPIRUN = mpirun # eval
PHYS_BIN_ROOT = /home/<username>/phys-bin
DATA_TREE_ROOT = /home/<username>/swim_data
DATA_ROOT = /home/<username>/swim_data
#PORTAL_URL = http://swim.gat.com:8080/monitor
#RUNID_URL = http://swim.gat.com:4040/runid.esp

#####
# resource detection method
#####
NODE_DETECTION = manual # checkjob | qstat | pbs_env | slurm_env | manual

#####
# manual allocation description
#####
TOTAL_PROCS = 4
NODES = 1
PROCS_PER_NODE = 4

#####
# node topology description
#####
CORES_PER_NODE = 4
SOCKETS_PER_NODE = 1

#####
# framework setting for node allocation
#####
```

<sup>10</sup> These need to be updated to match the “allocation” size each time. Alternatively, you can just use the *command line* to specify the number of nodes and processes per node.

```
# MUST ADHERE TO THE PLATFORM'S CAPABILITIES
# * EXCLUSIVE : only one task per node
# * SHARED : multiple tasks may share a node
# For single node jobs, this can be overridden allowing multiple
# tasks per node.
NODE_ALLOCATION_MODE = SHARED # SHARED | EXCLUSIVE
```

## 4.5 Plasma State Guide

### Coming Soon

This is where documentation for using the plasma state will go.

## 4.6 Fundamentals of the Advanced Features of the IPS

### Coming Soon

Information about multiple levels of parallelism and how to run multiple simulations effectively.

## 4.7 Performance Analysis

### Coming Soon

This article describes how to analyze the performance of IPS simulations at a variety of levels from the framework itself, to the underlying physics binaries.

## 4.8 Examples Listing

This document contains a list and brief description of all the files in the examples directory.

# COMPONENT GUIDES

This directory contains documentation for the use and development of the various components.

**To component documentation writers:** Documentation should include

- Information on how to use components in the IPS
  - config file settings
  - other component inputs
  - component outputs
  - point(s) of contact
  - location of relevant files
  - settings and modes of execution
  - execution characteristics or limitations
  - (brief) description of the role it plays in a coupled simulation
- Pointers
  - underlying application documentation
  - other relevant information

List of Components:

- Monitor Component - (location of documentation) (brief description)
- Portal Bridge Component - (location of documentation) (brief description)
- FTB Bridge Component - (location of documentation) (brief description)
- AORSA - (location of documentation) (brief description)
- TSC - (location of documentation) (brief description)
- NUBEAM - (location of documentation) (brief description)
- GENRAY - (location of documentation) (brief description)



# DEVELOPER GUIDES

This directory contains documentation to help developers navigate, augment and edit code associated with SWIM. Component developer documents are located in *Component Guides*.

## IPS Developer Documents

- *IPS Design Documentation*
- *The IPS Framework and Managers*

*Plasma State for Developers*

*Guide to Porting the IPS*

## 6.1 IPS Design Documentation

**Coming Soon**

## 6.2 The IPS Framework and Managers

**Coming Soon**

More detailed information for developers in the framework, managers and services.

## 6.3 Plasma State for Developers

**Coming Soon**

## 6.4 Guide to Porting the IPS

**Coming Soon**

This document will describe the procedure for porting the IPS to a new platform. It is designed for developers since this process unfortunately requires changes to the resource management piece of code in the framework. We hope to make this unnecessary in the future.

There is a work around for porting. You can use the manual resource specification interface in the platform configuration file. See *Platforms and Platform Configuration* for details.





# PORTAL GUIDES

## Coming Soon

This chapter contains various documents on how to use the portal and develop additional portal utilities.



---

# CODE LISTINGS

## 8.1 IPS

# local version The Integrated Plasma Simulator (IPS) Framework. This framework enables loose, file-based coupling of certain class of nuclear fusion simulation codes.

For further design information see

- Wael Elwasif, David E. Bernholdt, Aniruddha G. Shet, Samantha S. Foley, Randall Bramley, Donald B. Batchelor, and Lee A. Berry, *The Design and Implementation of the SWIM Integrated Plasma Simulator*, in The 18th Euromirco International Conference on Parallel, Distributed and Network - Based Computing (PDP 2010), 2010.
- Samantha S. Foley, Wael R. Elwasif, David E. Bernholdt, Aniruddha G. Shet, and Randall Bramley, *Extending the Concept of Component Interfaces: Experience with the Integrated Plasma Simulator*, in Component - Based High - Performance Computing (CBHPC) 2009, 2009, (extended abstract).
- D Batchelor, G Alba, E D'Azevedo, G Bateman, DE Bernholdt, L Berry, P Bonoli, R Bramley, J Breslau, M Chance, J Chen, M Choi, W Elwasif, S Foley, G Fu, R Harvey, E Jaeger, S Jardin, T Jenkins, D Keyes, S Klasky, S Kruger, L Ku, V Lynch, D McCune, J Ramos, D Schissel, D Schnack, and J Wright, *Advances in Simulation of Wave Interactions with Extended MHD Phenomena*, in Horst Simon, editor, SciDAC 2009, 14-18 June 2009, San Diego, California, USA, volume 180 of Journal of Physics: Conference Series, page 012054, Institute of Physics, 2009, 6pp.
- Samantha S. Foley, Wael R. Elwasif, Aniruddha G. Shet, David E. Bernholdt, and Randall Bramley, *Incorporating Concurrent Component Execution in Loosely Coupled Integrated Fusion Plasma Simulation*, in Component-Based High-Performance Computing (CBHPC) 2008, 2008, (extended abstract).
- D. Batchelor, C. Alba, G. Bateman, D. Bernholdt, L. Berry, P. Bonoli, R. Bramley, J. Breslau, M. Chance, J. Chen, M. Choi, W. Elwasif, G. Fu, R. Harvey, E. Jaeger, S. Jardin, T. Jenkins, D. Keyes, S. Klasky, S. Kruger, L. Ku, V. Lynch, D. McCune, J. Ramos, D. Schissel, D. Schnack, and J. Wright, *Simulation of Wave Interactions with MHD*, in Rick Stevens, editor, SciDAC 2008, 14-17 July 2008, Washington, USA, volume 125 of Journal of Physics: Conference Series, page 012039, Institute of Physics, 2008.
- Wael R. Elwasif, David E. Bernholdt, Lee A. Berry, and Don B. Batchelor, *Component Framework for Coupled Integrated Fusion Plasma Simulation*, in HPC-GECCO/CompFrame 2007, 21-22 October, Montreal, Quebec, Canada, 2007.

**Authors** Wael R. Elwasif, Samantha Foley, Aniruddha G. Shet

**Organization** Center for Simulation of RF Wave Interactions with Magnetohydrodynamics (CSWIM)

## 8.2 Framework

```
class ips.Framework (config_file_list, log_file, platform_file_name, debug=False, ftb=False, ver-
                    bose_debug=False, cmd_nodes=0, cmd_ppn=0)
```

```
critical (*args)
```

Produce **critical** message in simulation log file. Raise exception for bad formatting.

```
debug (*args)
```

Produce **debugging** message in simulation log file. Raise exception for bad formatting.

```
error (*args)
```

Produce **error** message in simulation log file. Raise exception for bad formatting.

```
exception (*args)
```

Produce **exception** message in simulation log file. Raise exception for bad formatting.

```
get_inq ()
```

Return handle to the Framework's input queue object ([multiprocessing.Queue](#))

```
info (*args)
```

Produce **informational** message in simulation log file. Raise exception for bad formatting.

```
log (*args)
```

Wrapper for [Framework.info\(\)](#).

```
register_service_handler (service_list, handler)
```

Register a call back method to handle a list of framework service invocations.

- *handler*: a Python callable object that takes a [messages.ServiceRequestMessage](#).
- *service\_list*: a list of service names to call *handler* when invoked by components. The service name must match the *target\_method* parameter in [messages.ServiceRequestMessage](#).

```
run ()
```

Run the communication outer loop of the framework.

This method implements the core communication and message dispatch functionality of the framework. The main phases of execution for the framework are:

1. Invoke the `init` method on all framework-attached components, blocking pending method call termination.
2. Generate method invocation messages for the remaining public method in the framework-centric components (i.e. `step` and `finalize`).
3. Generate a queue of method invocation messages for all public framework accessible components in the simulations being run. framework-accessible components are made up of the **Init** component (if it exists), and the **Driver** component. The generated messages invoke the public methods `init`, `step`, and `finalize`.
4. Dispatch method invocations for each framework-centric component and physics simulation in order.

Exceptions that propagate to this method from the managed simulations causes the framework to abort any pending method invocation for the source simulation. Exceptions from framework-centric component aborts further invocations to that component.

When all method invocations have been dispatched (or aborted), [Framework.terminate\\_sim\(\)](#) is called to trigger normal termination of all component processes.

```
terminate_sim (status=0)
```

Terminate all active component instances by invoking the `terminate` method on each one.

**warning** (\*args)

Produce **warning** message in simulation log file. Raise exception for bad formatting.

## 8.3 Data Manager

**class** `dataManager.DataManager` (*fwk*)

The data manager facilitates the movement and exchange of data files for the simulation.

**merge\_current\_plasma\_state** (*msg*)

Merge partial plasma state file with global master. Newly updated plasma state copied to caller's workdir. Exception raised on copy error.

*msg.args*:

0.partial\_state\_file

1.target\_state\_file

2.log\_file: stdout for merge process if not None

**process\_service\_request** (*msg*)

Invokes the appropriate public data manager method for the component specified in *msg*. Return method's return value.

**stage\_plasma\_state** (*msg*)

Copy plasma state files from source dir to target dir. Return 0. Exception raised on copy error.

*msg.args*:

0.plasma\_files

1.source\_dir

2.target\_dir

**update\_plasma\_state** (*msg*)

Copy plasma state files from source dir to target dir. Return 0. Exception raised on copy error.

*msg.args*:

0.plasma\_files

1.source\_dir

2.target\_dir

## 8.4 Task Manager

**class** `taskManager.TaskManager` (*fwk*)

The task manager is responsible for facilitating component method invocations, and the launching of tasks.

**build\_launch\_cmd** (*nproc*, *binary*, *cmd\_args*, *ppn*, *max\_ppn*, *nodes*, *accurateNodes*, *partial\_nodes*, *task\_id*, *core\_list*='')

Construct task launch command to be executed by the component.

*nproc* - number of processes to use *binary* - binary to launch *cmd\_args* - additional command line arguments for the binary *ppn* - processes per node value to use *max\_ppn* - maximum possible ppn for this allocation *nodes* - comma separated list of node ids *accurateNodes* - if `True`, launch on nodes in *nodes*, otherwise the parallel launcher determines the process placement *partial\_nodes* - if `True` and *accurateNodes* and *task\_launch\_cmd* == 'mpirun', a host file is created specifying

the exact placement of processes on cores. *core\_list* - used for creating host file with process to core mappings

**finish\_task** (*finish\_task\_msg*)

Cleanup after a task launched by a component terminates

*finish\_task\_msg* is expected to be of type `messages.ServiceRequestMessage`

Message args:

0.*task\_id*: task id of finished task

1.*task\_data*: return code of task

**get\_call\_id** ()

Return a new call id

**get\_task\_id** ()

Return a new task id

**init\_call** (*init\_call\_msg*, *manage\_return=True*)

Creates and sends a `messages.MethodInvokeMessage` from the calling component to the target component. If *manage\_return* is `True`, a record is added to *outstanding\_calls*. Return call id.

Message args:

0.*method\_name*

1.+ arguments to be passed on as method arguments.

**init\_task** (*init\_task\_msg*)

Allocate resources needed for a new task and build the task launch command using the binary and arguments provided by the requesting component. Return launch command to component via `messages.ServiceResponseMessage`. Raise exception if task can not be launched at this time (`ipsExceptions.BadResourceRequestException`, `ipsExceptions.InsufficientResourcesException`).

*init\_task\_msg* is expected to be of type `messages.ServiceRequestMessage`

Message args:

0.*nproc*: number of processes the task needs

1.*binary*: full path to the executable to launch

2.*tpn*: processes per node for this task. (0 indicates that the default ppn is used.)

3.*block*: whether or not to wait until the task can be launched.

4.*wnodes*: `True` for whole node allocation, `False` otherwise.

5.*wsocks*: `True` for whole socket allocation, `False` otherwise.

6.+ *cmd\_args*: any arguments for the executable

**init\_task\_pool** (*init\_task\_msg*)

Allocate resources needed for a new task and build the task launch command using the binary and arguments provided by the requesting component.

*init\_task\_msg* is expected to be of type `messages.ServiceRequestMessage`

Message args:

0.*task\_dict*: dictionary of task names and objects

**initialize** (*data\_mgr*, *resource\_mgr*, *config\_mgr*, *ftb*)

Initialize references to other managers and key values from configuration manager.

**printCurrTaskTable ()**

Prints the task table pretty-like.

**process\_service\_request (msg)**

Invokes the appropriate public data manager method for the component specified in *msg*. Return method's return value.

**return\_call (response\_msg)**

Handle the response message generated by a component in response to a method invocation on that component.

*reponse\_msg* is expected to be of type `messages.MethodResultMessage`

**wait\_call (wait\_msg)**

Determine if the call has finished. If finished, return any data or errors. If not finished raise the appropriate blocking or nonblocking exception and try again later.

*wait\_msg* is expected to be of type `messages.ServiceRequestMessage`

Message args:

0.*call\_id*: call id for which to wait

1.*blocking*: determines the wait is blocking or not

## 8.5 Resource Manager

**class resourceManager.ResourceManager (fwk)**

The resource manager is responsible for detecting the resources allocated to the framework, allocating resources to task requests, and maintaining the associated bookkeeping.

**add\_nodes (listOfNodes)**

Add node entries to `self.nodes`. Typically used by `initialize()` to initialize `self.nodes`. May be used to add nodes to a dynamic allocation in the future.

*listOfNodes* is a list of tuples (*node name*, *cores*). `self.nodes` is a dictionary where the keys are the *node names* and the values are `node_structure.Node` structures.

Return total number of cores.

**begin\_RM\_report ()**

Print header information for resource usage reporting file.

**check\_core\_cap (nproc, ppn)**

Determine if it is currently possible to allocate *nproc* processes with a ppn of *ppn* without further restrictions.. Return `True` and list of nodes to use if successful. Return `False` and empty list if there are not enough available resources at this time, but it is possible to eventually satisfy the request. Exception raised if the request can never be fulfilled.

**check\_whole\_node\_cap (nproc, ppn)**

Determine if it is currently possible to allocate *nproc* processes with a ppn of *ppn* and whole nodes. Return `True` and list of nodes to use if successful. Return `False` and empty list if there are not enough available resources at this time, but it is possible to eventually satisfy the request. Exception raised if the request can never be fulfilled.

**check\_whole\_sock\_cap (nproc, ppn)**

Determine if it is currently possible to allocate *nproc* processes with a ppn of *ppn* and whole sockets. Return `True` and list of nodes to use if successful. Return `False` and empty list if there are not enough available resources at this time, but it is possible to eventually satisfy the request. Exception raised if the request can never be fulfilled.

**get\_allocation** (*comp\_id, nproc, task\_id, whole\_nodes, whole\_socks, task\_ppn=0*)

Traverse available nodes to return:

If *whole\_nodes* is `True`:

- *shared\_nodes*: `False`
- *nodes*: list of node names
- *ppn*: processes per node for launching the task
- *max\_ppn*: processes that can be launched
- *accurateNodes*: `True` if *nodes* uses the actual names of the nodes, `False` otherwise.

If *whole\_nodes* is `False`:

- *shared\_nodes*: `True`
- *nodes*: list of node names
- *node\_file\_entries*: list of slots to be used to launch the task
- *ppn*: processes per node for launching the task
- *max\_ppn*: processes that can be launched
- *accurateNodes*: `True` if *nodes* uses the actual names of the nodes, `False` otherwise.

Aguments:

- *nproc*: the number of requested processes (int)
- *comp\_id*: component identifier, must be unique with respect to the framework (string)
- *task\_id*: task identifier from TM (int)
- *method*: name of method (string)
- *task\_ppn*: ppn for this task (optional) (int)

**initialize** (*dataMngr, taskMngr, configMngr, fib, cmd\_nodes=0, cmd\_ppn=0*)

Initialize resource management structures, references to other managers (*dataMngr, taskMngr, configMngr*), and feature settings (*fib*).

Resource information comes from the following in order of priority:

- command line specification (*cmd\_nodes, cmd\_ppn*)
- detection using parameters from platform config file
- manual settings from platform config file

The second two sources are obtained through `resourceHelper.getResourceList()`.

**printRMState** ()

Print the node tree to `stdout`.

**process\_FTB\_events** (*topic, event*)

**process\_service\_request** (*msg*)

**release\_allocation** (*task\_id, status*)

Set resources allocated to task *task\_id* to available. *status* is not used, but may be used to correlate resource failures to task failures and implement task relaunch strategies.

**report\_RM\_status** (*notes=''*)

Print current RM status to the reporting\_file ("resource\_usage") Entries consist of:



- time in seconds since beginning of time (`__init__` of RM)
- # cores that are available
- # cores that are allocated
- % allocated cores
- # processes launched by task
- % cores used by processes
- notes (a description of the event that changed the resource usage)

**sendEvent** (*eventName, info*)

wrapper for constructing and publishing EM events

---

**class** `node_structure.Node` (*name, socks, cores, p*)

Models a node in the allocation.

- name*: name of node, typically actual name from resource detection phase.
- task\_ids, owners*: identifiers for the tasks and components that are currently using the node.
- allocated, available*: list of sockets that have cores allocated and available. A socket may appear in both lists if it is only partially allocated.
- sockets*: list of sockets belonging to this node
- avail\_cores*: number of cores that are currently available.
- total\_cores*: total number of cores that can be allocated on this node.
- status*: indicates if the node is 'UP' or 'DOWN'. Currently not used, all nodes are considered functional..

**allocate** (*whole\_nodes, whole\_sockets, tid, o, procs*)

Mark *procs* number of cores as allocated subject to the values of *whole\_nodes* and *whole\_sockets*. Return the number of cores allocated and their corresponding slots, a list of strings of the form:

<socket name>:<core name>

**print\_sockets** (*fname=''*)

Pretty print of state of sockets.

**release** (*tid, o*)

Mark cores used by task *tid* and component *o* as available. Return the number of cores released.

**class** `node_structure.Socket` (*name, cps*)

Models a socket in a node.

- name*: identifier for the socket
- task\_ids, owners*: identifiers for the tasks and components that are currently using the socket.
- allocated, available*: lists of cores that are allocated and available.
- cores*: list of `Core` objects belonging to this socket
- avail\_cores*: number of cores that are currently available.
- total\_cores*: total number of cores that can be allocated on this socket.

**allocate** (*whole, tid, o, num\_procs*)

Mark *num\_procs* cores as allocated subject to the value of *whole*. Return a list of strings of the form:

<socket name>:<core name>

**print\_cores** (*fname*='')

Pretty print of state of cores.

**release** (*tid*)

Mark cores that are allocated to task *tid* as available. Return number of cores set to available.

**class** node\_structure.**Core** (*name*)

Models a core of a socket.

- name*: name of core

- is\_available*: boolean value indicating the availability of the core.

- task\_id, owner*: identifiers of the task and component using the core.

**allocate** (*tid, o*)

Mark core as allocated.

**release** ()

Mark core as available.

---

The Resource Helper file contains all of the code needed to figure out what host we are on and what resources we have. Taking this out of the resource manager will allow us to test it independent of the IPS.

resourceHelper.**getResourceList** (*services, host, ffb, partial\_nodes=False*)

Using the host information, the resources are detected. Return list of (<node name>, <processes per node>), cores per node, sockets per node, processes per node, and `True` if the node names are accurate, `False` otherwise.

resourceHelper.**get\_checkjob\_info** ()

Use `checkjob $PBS_JOBID` to get the node names and core counts of allocation. Typically works in a Cray environment.

**Note:** Two formats for outputting resource information.

- 1.[node\_id:tasks\_per\_node]+

- 2.([comma separated list of node\_ids and node\_id ranges]\*tasks\_per\_node)+

resourceHelper.**get\_pbs\_info** ()

Access info about allocation from PBS environment variables:

PBS\_NNODES PBS\_NODEFILE

resourceHelper.**get\_qstat\_jobinfo** ()

Use `qstat -f $PBS_JOBID` to get the number of nodes and ppn of the allocation. Typically works on PBS systems.

resourceHelper.**get\_slurm\_info** ()

Access environment variables set by Slurm to get the node names, tasks per node and number of processes.

SLURM\_NODELIST SLURM\_TASKS\_PER\_NODE or SLURM\_JOB\_TASKS\_PER\_NODE  
SLURM\_NPROC

resourceHelper.**get\_topo** (*services*)

Uses `hwloc` library calls in C program `topo_disco` to detect the topology of a node in the allocation. Return the number of sockets and the number of cores.

**Note:** Not available on all platforms.

resourceHelper.**manual\_detection** (*services*)

Use values listed in platform configuration file.

## 8.6 Component

**class** `component.Component` (*services, config*)

Base class for all IPS components. Common set up, connection and invocation actions are implemented here.

**checkpoint** (*timestamp=0.0*)

Produce some default debugging information before the rest of the code is executed.

**finalize** (*timestamp=0.0*)

Produce some default debugging information before the rest of the code is executed.

**init** (*timestamp=0.0*)

Produce some default debugging information before the rest of the code is executed.

**restart** (*timestamp=0.0*)

Produce some default debugging information before the rest of the code is executed.

**step** (*timestamp=0.0*)

Produce some default debugging information before the rest of the code is executed.

**terminate** (*status*)

Clean up services and call `sys_exit`.

## 8.7 Configuration Manager

**class** `configurationManager.ConfigurationManager` (*fwk, config\_file\_list, platform\_file\_name*)

The configuration manager is responsible for parsing the simulation and platform configuration files, creating the framework and simulation components, as well as providing an interface to accessing items from the configuration files (e.g., the time loop).

**class** `SimulationData` (*sim\_name*)

Structure to hold simulation data stored into the `sim_map` entry in the `configurationManager` class

`ConfigurationManager.getPort` (*sim\_name, port\_name*)

Deprecated since version 1.0: Use `get_port()`

`ConfigurationManager.get_component_map` ()

Return a dictionary of simulation names and lists of component references. (May only be the driver, and init (if present)???)

`ConfigurationManager.get_config_parameter` (*sim\_name, param*)

Return value of *param* from simulation configuration file for *sim\_name*.

`ConfigurationManager.get_driver_components` ()

Return a list of driver components, one for each sim.

`ConfigurationManager.get_framework_components` ()

Return list of framework components.

`ConfigurationManager.get_framework_logger` (*sim\_name*)

Return framework logger for simulation *sim\_name*.

`ConfigurationManager.get_init_components` ()

Return list of init components.

`ConfigurationManager.get_platform_parameter` (*param, silent=False*)

Return value of platform parameter *param*. If *silent* is `False` (default) `None` is returned when *param* not found, otherwise an exception is raised.

`ConfigurationManager.get_port(sim_name, port_name)`  
Return a reference to the component from simulation *sim\_name* implementing port *port\_name*.

`ConfigurationManager.get_sim_names()`  
Return list of names of simulations.

`ConfigurationManager.get_sim_parameter(sim_name, param)`  
Return value of *param* from simulation configuration file for *sim\_name*.

`ConfigurationManager.initialize(data_mgr, resource_mgr, task_mgr, ffb)`  
Parse the platform and simulation configuration files using the `ConfigObj` module. Create and initialize simulation(s) and their components, framework components and loggers.

`ConfigurationManager.process_service_request(msg)`  
Invokes public configuration manager method for a component. Return method's return value.

`ConfigurationManager.set_config_parameter(sim_name, param, value, target_sim_name)`  
Set the configuration parameter *param* to value *value* in *target\_sim\_name*. If *target\_sim\_name* is the framework, all simulations will get the change. Return *value*.

`ConfigurationManager.terminate(status)`  
Terminates all processes attached to the framework. *status* not used.

## 8.8 Services

`class services.ServicesProxy(fwk, fwk_in_q, svc_response_q, sim_conf, log_pipe_name)`

`add_task(task_pool_name, task_name, nproc, working_dir, binary, *args, **keywords)`  
Add task *task\_name* to task pool *task\_pool\_name*. Remaining arguments are the same as in `ServicesProxy.launch_task()`.

`call(component_id, method_name, *args)`  
Invoke method *method\_name* on component *component\_id* with optional arguments *\*args*. Return result from invoking the method.

`call_nonblocking(component_id, method_name, *args)`  
Invoke method *method\_name* on component *component\_id* with optional arguments *\*args*. Return *call\_id*.

`checkpoint_components(comp_id_list, time_stamp, Force=False, Protect=False)`  
Selectively checkpoint components in *comp\_id\_list* based on the configuration section *CHECKPOINT*. If *Force* is `True`, the checkpoint will be taken even if the conditions for taking the checkpoint are not met. If *Protect* is `True`, then the data from the checkpoint is protected from clean up. *Force* and *Protect* are optional and default to `False`.

The *CHECKPOINT\_MODE* option controls determines if the components checkpoint methods are invoked.

Possible *MODE* options are:

**WALLTIME\_REGULAR:** checkpoints are saved upon invocation of the service call `checkpoint_components()`, when a time interval greater than, or equal to, the value of the configuration parameter *WALLTIME\_INTERVAL* had passed since the last checkpoint. A checkpoint is assumed to have happened (but not actually stored) when the simulation starts. Calls to `checkpoint_components()` before *WALLTIME\_INTERVAL* seconds have passed since the last successful checkpoint result in a NOOP.

**WALLTIME\_EXPLICIT:** checkpoints are saved when the simulation wall clock time exceeds one of the (ordered) list of time values (in seconds) specified in the variable `WALLTIME_VALUES`. Let  $[t_0, t_1, \dots, t_n]$  be the list of wall clock time values specified in the configuration parameter `WALLTIME_VALUES`. Then  $\text{checkpoint}(T) = \text{True}$  if  $T \geq t_j$ , for some  $j$  in  $[0, n]$  and there is no other time  $T_1$ , with  $T > T_1 \geq t_j$  such that  $\text{checkpoint}(T_1) = \text{True}$ . If the test fails, the call results in a NOOP.

**PHYSTIME\_REGULAR:** checkpoints are saved at regularly spaced “physics time” intervals, specified in the configuration parameter `PHYSTIME_INTERVAL`. Let `PHYSTIME_INTERVAL = PTI`, and the physics time stamp argument in the call to `checkpoint_components()` be `pts_i`, with  $i = 0, 1, 2, \dots$ . Then  $\text{checkpoint}(\text{pts}_i) = \text{True}$  if  $\text{pts}_i \geq n \text{ PTI}$ , for some  $n$  in  $1, 2, 3, \dots$  and  $\text{pts}_i - \text{pts}_{\text{prev}} \geq \text{PTI}$ , where  $\text{checkpoint}(\text{pts}_{\text{prev}}) = \text{True}$  and  $\text{pts}_{\text{prev}} = \max(\text{pts}_0, \text{pts}_1, \dots, \text{pts}_{i-1})$ . If the test fails, the call results in a NOOP.

**PHYSTIME\_EXPLICIT:** checkpoints are saved when the physics time equals or exceeds one of the (ordered) list of physics time values (in seconds) specified in the variable `PHYSTIME_VALUES`. Let  $[\text{pt}_0, \text{pt}_1, \dots, \text{pt}_n]$  be the list of physics time values specified in the configuration parameter `PHYSTIME_VALUES`. Then  $\text{checkpoint}(\text{pt}) = \text{True}$  if  $\text{pt} \geq \text{pt}_j$ , for some  $j$  in  $[0, n]$  and there is no other physics time  $\text{pt}_k$ , with  $\text{pt} > \text{pt}_k \geq \text{pt}_j$  such that  $\text{checkpoint}(\text{pt}_k) = \text{True}$ . If the test fails, the call results in a NOOP.

The configuration parameter `NUM_CHECKPOINT` controls how many checkpoints to keep on disk. Checkpoints are deleted in a FIFO manner, based on their creation time. Possible values of `NUM_CHECKPOINT` are:

- `NUM_CHECKPOINT = n`, with  $n > 0 \rightarrow$  Keep the most recent  $n$  checkpoints
- `NUM_CHECKPOINT = 0 \rightarrow` No checkpoints are made/kept (except when *Force* = True)
- `NUM_CHECKPOINT < 0 \rightarrow` Keep ALL checkpoints

Checkpoints are saved in the directory `${SIM_ROOT}/restart`

**create\_task\_pool** (*task\_pool\_name*)

Create an empty pool of tasks with the name *task\_pool\_name*. Raise exception if duplicate name.

**critical** (\*args)

Produce **critical** message in simulation log file. Raise exception for bad formatting.

**debug** (\*args)

Produce **debugging** message in simulation log file. Raise exception for bad formatting.

**error** (\*args)

Produce **error** message in simulation log file. Raise exception for bad formatting.

**exception** (\*args)

Produce **exception** message in simulation log file. Raise exception for bad formatting.

**getGlobalConfigParameter** (*param*)

Deprecated since version 1.0: Use `ServicesProxy.get_config_param()`

**getPort** (*port\_name*)

Deprecated since version 1.0: Use `ServicesProxy.get_port()`

**getTimeLoop** ()

Deprecated since version 1.0: Use `ServicesProxy.get_time_loop()`

**get\_config\_param** (*param*)

Return the value of the configuration parameter *param*. Raise exception if not found.

**get\_finished\_tasks** (*task\_pool\_name*)

Return dictionary of finished tasks and return values in task pool *task\_pool\_name*. Raise exception if no active or finished tasks.

**get\_port** (*port\_name*)

Return a reference to the component implementing port *port\_name*.

**get\_restart\_files** (*restart\_root*, *timeStamp*, *file\_list*)

Copy files needed for component restart from the restart directory:

```
<restart_root>/restart/<timeStamp>/components/$CLASS_${SUB_CLASS}_${NAME}_${SEQ_NUM}
```

to the component's work directory.

Copying errors are not fatal (exception raised).

**get\_time\_loop** ()

Return the list of times as specified in the configuration file.

**get\_working\_dir** ()

Return the working directory of the calling component.

The structure of the working directory is defined using the configuration parameters *CLASS*, *SUB\_CLASS*, and *NAME* of the component configuration section. The structure of the working directory is:

```
${SIM_ROOT}/work/$CLASS_${SUB_CLASS}_${NAME}<instance_num>
```

**info** (\*args)

Produce **informational** message in simulation log file. Raise exception for bad formatting.

**kill\_all\_tasks** ()

Kill all tasks associated with this component.

**kill\_task** (*task\_id*)

Kill launched task *task\_id*. Return if successful. Raises exceptions if the task or process cannot be found or killed successfully.

**launch\_task** (*nproc*, *working\_dir*, *binary*, \*args, \*\*keywords)

Launch *binary* in *working\_dir* on *nproc* processes. \*args are any arguments to be passed to the binary on the command line. \*\*keywords are any keyword arguments used by the framework to manage how the binary is launched. Keywords may be the following:

- *task\_ppn* : the processes per node value for this task
- *block* : specifies that this task will block (or raise an exception) if not enough resources are available to run immediately. If `True`, the task will be retried until it runs. If `False`, an exception is raised indicating that there are not enough resources, but it is possible to eventually run. (default = `True`)
- *tag* : identifier for the portal. May be used to group related tasks.
- *logfile* : file name for `stdout` (and `stderr`) to be redirected to for this task. By default `stderr` is redirected to `stdout`, and `stdout` is not redirected.
- *whole\_nodes* : if `True`, the task will be given exclusive access to any nodes it is assigned. If `False`, the task may be assigned nodes that other tasks are using or may use.
- *whole\_sockets* : if `True`, the task will be given exclusive access to any sockets of nodes it is assigned. If `False`, the task may be assigned sockets that other tasks are using or may use.

Return *task\_id* if successful. May raise exceptions related to opening the logfile, being unable to obtain enough resources to launch the task (`ipsExceptions.InsufficientResourcesException`), bad task launch request (`ipsExceptions.ResourceRequestMismatchException`),

`ipsExceptions.BadResourceRequestException`) or problems executing the command. These exceptions may be used to retry launching the task as appropriate.

**Note:** This is a nonblocking function, users must use a version of `ServicesProxy.wait_task()` to get result.

**launch\_task\_pool** (*task\_pool\_name*)

Construct messages to task manager to launch each task. Used by `TaskPool` to launch tasks in a task\_pool.

**launch\_task\_resilient** (*nproc, working\_dir, binary, \*args, \*\*keywords*)  
**not used**

**log** (*\*args*)

Wrapper for `ServicesProxy.info()`.

**merge\_current\_plasma\_state** (*partial\_state\_file, logfile=None*)

Merge partial plasma state with global state. Partial plasma state contains only the values that the component contributes to the simulation. Raise exceptions on bad merge. Optional *logfile* will capture stdout from merge.

**process\_events** ()

Poll for events on subscribed topics.

**publish** (*topicName, eventName, eventBody*)

Publish event consisting of *eventName* and *eventBody* to topic *topicName* to the IPS event service.

**remove\_task\_pool** (*task\_pool\_name*)

Kill all running tasks, clean up all finished tasks, and delete task pool.

**save\_restart\_files** (*timeStamp, file\_list*)

Copy files needed for component restart to the restart directory:

```
${SIM_ROOT}/restart/${timestamp}/components/${CLASS}_${SUB_CLASS}_${NAME}
```

Copying errors are not fatal (exception raised).

**send\_portal\_event** (*event\_type='COMPONENT\_EVENT', event\_comment=''*)

Send event to web portal.

**setMonitorURL** (*url=''*)

Send event to portal setting the URL where the monitor component will put data.

**set\_config\_param** (*param, value, target\_sim\_name=None*)

Set configuration parameter *param* to *value*. Raise exceptions if the parameter cannot be changed or if there are problems setting the value.

**stageCurrentPlasmaState** ()

Deprecated since version 1.0: Use `ServicesProxy.stage_plasma_state()`

**stageInputFiles** (*input\_file\_list*)

Deprecated since version 1.0: Use `ServicesProxy.stage_input_files()`

**stageOutputFiles** (*timeStamp, output\_file\_list*)

Deprecated since version 1.0: Use `ServicesProxy.stage_output_files()`

**stage\_input\_files** (*input\_file\_list*)

Copy component input files to the component working directory (as obtained via a call to `ServicesProxy.get_working_dir()`). Input files are assumed to be originally located in the directory variable *INPUT\_DIR* in the component configuration section.

**stage\_output\_files** (*timeStamp, file\_list, keep\_old\_files=True*)

Copy associated component output files (from the working directory) to the component simulation results



directory. Output files are prefixed with the configuration parameter *OUTPUT\_PREFIX*. The simulation results directory has the format:

```
${SIM_ROOT}/simulation_results/<timeStamp>/components/${CLASS}_${SUB_CLASS}_${NAME}_${SEQ_NUM}
```

Additionally, plasma state files are archived for debugging purposes:

```
${SIM_ROOT}/history/plasma_state/<file_name>_${CLASS}_${SUB_CLASS}_${NAME}_<timeStamp>
```

Copying errors are not fatal (exception raised).

**stage\_plasma\_state()**

Copy current plasma state to work directory.

**stage\_replay\_output\_files(*timeStamp*)**

Copy output files from the replay component to current sim for physics time *timeStamp*. Return location of new local copies.

**stage\_replay\_plasma\_files(*timeStamp*)**

Copy plasma state files from the replay component to current sim for physics time *timeStamp*. Return location of new local copies.

**submit\_tasks(*task\_pool\_name*, *block*=True)**

Launch all unfinished tasks in task pool *task\_pool\_name*. If *block* is True, return when all tasks have been launched. If *block* is False, return when all tasks that can be launched immediately have been launched. Return number of tasks submitted.

**subscribe(*topicName*, *callback*)**

Subscribe to topic *topicName* on the IPS event service and register *callback* as the method to be invoked when an event is published to that topic.

**unsubscribe(*topicName*)**

Remove subscription to topic *topicName*.

**updatePlasmaState()**

Deprecated since version 1.0: Use `ServicesProxy.update_plasma_state()`

**updateTimeStamp(*newTimeStamp*=-1)**

Deprecated since version 1.0: Use `ServicesProxy.update_time_stamp()`

**update\_plasma\_state(*plasma\_state\_files*=None)**

Copy local (updated) plasma state to global state. If no plasma state files are specified, component configuration specification is used. Raise exceptions upon copy.

**update\_time\_stamp(*new\_time\_stamp*=-1)**

Update time stamp on portal.

**wait\_call(*call\_id*, *block*=True)**

If *block* is True, return when the call has completed with the return code from the call. If *block* is False, raise `ipsExceptions.IncompleteCallException` if the call has not completed, and the return value is it has.

**wait\_call\_list(*call\_id\_list*, *block*=True)**

Check the status of each of the call in *call\_id\_list*. If *block* is True, return when *all* calls are finished. If *block* is False, raise `ipsExceptions.IncompleteCallException` if *any* of the calls have not completed, otherwise return. The return value is a dictionary of *call\_ids* and return values.

**wait\_task(*task\_id*)**

Check the status of task *task\_id*. Return the return value of the task when finished successfully. Raise exceptions if the task is not found, or if there are problems finalizing the task.



**wait\_task\_nonblocking** (*task\_id*)

Check the status of task *task\_id*. If it has finished, the return value is populated with the actual value, otherwise `None` is returned. A `KeyError` exception may be raised if the task is not found.

**wait\_task\_resilient** (*task\_id*)  
not used

**wait\_tasklist** (*task\_id\_list*, *block=True*)

Check the status of a list of tasks. If *block* is `True`, return a dictionary of return values when *all* tasks have completed. If *block* is `False`, return a dictionary containing entries for each *completed* task. Note that the dictionary may be empty. Raise `KeyError` exception if *task\_id* not found.

**warning** (*\*args*)

Produce **warning** message in simulation log file. Raise exception for bad formatting.

**class** `services.Task` (*task\_name*, *nproc*, *working\_dir*, *binary*, *\*args*, *\*\*keywords*)

Container for task information:

- name*: task name
- nproc*: number of processes the task needs
- working\_dir*: location to launch task from
- binary*: full path to executable to launch
- \*args*: arguments for *binary*
- \*\*keywords*: keyword arguments for launching the task. See `ServicesProxy.launch_task()` for details.

**class** `services.TaskPool` (*name*, *services*)

Class to contain and manage a pool of tasks.

**add\_task** (*task\_name*, *nproc*, *working\_dir*, *binary*, *\*args*, *\*\*keywords*)

Create `Task` object and add to *queued\_tasks* of the task pool. Raise exception if task name already exists in task pool.

**get\_finished\_tasks\_status** ()

Return a dictionary of exit status values for all tasks that have finished since the last time finished tasks were polled.

**submit\_tasks** (*block=True*)

Launch tasks in *queued\_tasks*. Finished tasks are handled before launching new ones. If *block* is `True`, the number of tasks submitted is returned after all tasks have been launched and completed. If *block* is `False` the number of tasks that can immediately be launched is returned.

**submit\_tasks\_old** (*block=True*)

Deprecated since version Experimental: Use `TaskPool.submit_tasks()`

**terminate\_tasks** ()

Kill all active tasks, clear all queued, blocked and finished tasks.

## 8.9 Other Utilities

### 8.9.1 IPS Exceptions

**exception** `ipsExceptions.AllocatedNodeDownException` (*identifier*, *tid*, *comp\_id*)

Exception is raised when an allocated node is discovered to be faulty. The task manager should catch the exception and do something with it.

**exception** `ipsExceptions.BadResourceRequestException (caller_id, tid, request, deficit)`  
Exception raised by the resource manager when a component requests a quantity of resources that can never be satisfied during a `get_allocation()` call

**exception** `ipsExceptions.BlockedMessageException (msg, reason)`  
Exception Raised by the any manager when a blocking service invocation is made, and the invocation result is not readily available.

**exception** `ipsExceptions.IncompleteCallException (callID)`  
Exception Raised by the taskManager when a nonblocking `wait_call()` method is invoked before the call has finished.

**exception** `ipsExceptions.InsufficientResourcesException (caller_id, tid, request, deficit)`  
Exception Raised by the resource manager when not enough resources are available to satisfy an `allocate()` call

**exception** `ipsExceptions.InvalidResourceSettingsException (t, spn, cpn)`  
Exception raised by the resource helper to indicate inconsistent resource settings.

**exception** `ipsExceptions.NonexistentResourceException (identifier)`  
Exception for any time nonexistent (nodes) are tried to be used

**exception** `ipsExceptions.ReleaseMismatchException (caller_id, tid, old_alc, old_avc, new_alc, new_avc)`  
Exception raised by the resource manager when a release allocation request accounting yields unexpected results.

**exception** `ipsExceptions.ResourceRequestMismatchException (caller_id, tid, nproc, ppn, max_procs, max_ppn)`  
Exception raised by the resource manager when it is possible to launch the requested number of processes, but not on the requested number of processes per node.

## 8.9.2 IPS Utilities

**ipsutil.copyFiles** (*src\_dir, src\_file\_list, target\_dir, prefix='', keep\_old=False*)  
Copy files in *src\_file\_list* from *src\_dir* to *target\_dir* with an optional prefix. If *keep\_old* is `True`, existing files in *target\_dir* will not be overridden, otherwise files can be clobbered (default). Wild-cards in file name specification are allowed, *target\_dir* is created if it doesn't already exist.

**ipsutil.getTimeString** (*timeArg=None*)  
Return a string representation of *timeArg*. *timeArg* is expected to be an appropriate object to be processed by `time.strftime()`. If *timeArg* is `None`, current time is used.

**class** `messages.ExitMessage (sender_id, receiver_id, status, *args)`  
Message used to communicate the exit status of a component.

- sender\_id*: component id that is telling the component to die (framework)
- receiver\_id*: component id that is to die
- status*: either `Messages.SUCCESS` or `Messages.FAILURE` indicating if the exit request is due to the simulation finishing successfully or in error.
- \*args*: other information passed to the component to die.

**class** `messages.Message (sender_id, receiver_id)`  
Base class for all IPS messages. **Should not be used in actual communication.**

**get\_message\_id** ()

**class** `messages.MethodInvokeMessage (sender_id, receiver_id, call_id, target_method, *args)`  
Message used by components to invoke methods on other components.

- sender\_id*: component id of the sender
- receiver\_id*: component id of the receiver
- call\_id*: identifier of the call (generated by caller)
- target\_method*: method to be invoked on the receiver
- \*args*: arguments to be passed to the *target\_method*

**class** `messages.MethodResultMessage` (*sender\_id, receiver\_id, call\_id, status, \*args*)

Message used to relay the return value after a method invocation.

- sender\_id*: component id of the sender (callee)
- receiver\_id*: component id of the receiver (caller)
- call\_id*: identifier of the call (generated by caller)
- status*: either `Message.SUCCESS` or `Message.FAILURE` indicating the success or failure of the invocation.
- \*args*: other information to be passed back to the caller.

**class** `messages.ServiceRequestMessage` (*sender\_id, receiver\_id, target\_comp\_id, target\_method, \*args*)

Message used by components to request the result of a service action by one of the IPS managers.

- sender\_id*: component id of the sender
- receiver\_id*: component id of the receiver (framework)
- target\_comp\_id*: component id of target component (typically framework)
- target\_method*: name of method to be invoked on component *target\_comp\_id*
- \*args*: any number of arguments. These are specific to the target method.

**class** `messages.ServiceResponseMessage` (*sender\_id, receiver\_id, request\_msg\_id, status, \*args*)

Message used by managers to respond with the result of the service action to the calling component.

- sender\_id*: component id of the sender (framework)
- receiver\_id*: component id of the receiver (calling component)
- request\_msg\_id*: id of request message this is a response to.
- status*: either `Message.SUCCESS` or `Message.FAILURE`
- \*args*: any number of arguments. These are specific to type of response.

`sendPost.sendEncodedMessage` (*url, msg*)

## 8.10 Framework Components

**class** `portalBridge.PortalBridge` (*services, config*)

Framework component to communicate with the [SWIM web portal](#).

**class** `SimulationData`

Container for simulation data.

`PortalBridge.get_elapsed_time()`

Return total elapsed time since simulation started in seconds (including a possible fraction)

`PortalBridge.init(timestamp=0.0)`

Try to connect to the portal, subscribe to `_IPS_MONITOR` events and register callback `process_event()`.

`PortalBridge.init_simulation(sim_name, sim_root)`

Create and send information about simulation `sim_name` living in `sim_root` so the portal can set up corresponding structures to manage data from the sim.

`PortalBridge.process_event(topicName, theEvent)`

Process a single event `theEvent` on topic `topicName`.

`PortalBridge.send_event(sim_data, event_data)`

Send contents of `event_data` and `sim_data` to portal.

`PortalBridge.step(timestamp=0.0)`

Poll for events.

# INDEXES AND TABLES

- *genindex*
- *modindex*
- *search*



# PYTHON MODULE INDEX

## c

component, [71](#)  
configurationManager, [71](#)

## d

dataManager, [65](#)

## i

ips, [63](#)  
ipsExceptions, [77](#)  
ipsutil, [78](#)

## m

messages, [78](#)

## p

portalBridge, [79](#)

## r

resourceHelper, [70](#)  
resourceManager, [67](#)

## s

sendPost, [79](#)  
services, [72](#)

## t

taskManager, [65](#)





# INDEX

## A

add\_nodes() (resourceManager.ResourceManager method), 67  
add\_task() (services.ServicesProxy method), 72  
add\_task() (services.TaskPool method), 77  
allocate() (node\_structure.Core method), 70  
allocate() (node\_structure.Node method), 69  
allocate() (node\_structure.Socket method), 69  
AllocatedNodeDownException, 77

## B

BadResourceRequestException, 77  
begin\_RM\_report() (resourceManager.ResourceManager method), 67  
BlockedMessageException, 78  
build\_launch\_cmd() (taskManager.TaskManager method), 65

## C

call() (services.ServicesProxy method), 72  
call\_nonblocking() (services.ServicesProxy method), 72  
check\_core\_cap() (resourceManager.ResourceManager method), 67  
check\_whole\_node\_cap() (resourceManager.ResourceManager method), 67  
check\_whole\_sock\_cap() (resourceManager.ResourceManager method), 67  
checkpoint() (component.Component method), 71  
checkpoint\_components() (services.ServicesProxy method), 72  
Component (class in component), 71  
component (module), 71  
ConfigurationManager (class in configurationManager), 71  
configurationManager (module), 71  
ConfigurationManager.SimulationData (class in configurationManager), 71  
copyFiles() (in module ipsutil), 78  
Core (class in node\_structure), 70  
create\_task\_pool() (services.ServicesProxy method), 73  
critical() (ips.Framework method), 64

critical() (services.ServicesProxy method), 73

## D

DataManager (class in dataManager), 65  
dataManager (module), 65  
debug() (ips.Framework method), 64  
debug() (services.ServicesProxy method), 73

## E

error() (ips.Framework method), 64  
error() (services.ServicesProxy method), 73  
exception() (ips.Framework method), 64  
exception() (services.ServicesProxy method), 73  
ExitMessage (class in messages), 78

## F

finalize() (component.Component method), 71  
finish\_task() (taskManager.TaskManager method), 66  
Framework (class in ips), 64

## G

get\_allocation() (resourceManager.ResourceManager method), 67  
get\_call\_id() (taskManager.TaskManager method), 66  
get\_checkjob\_info() (in module resourceHelper), 70  
get\_component\_map() (configurationManager.ConfigurationManager method), 71  
get\_config\_param() (services.ServicesProxy method), 73  
get\_config\_parameter() (configurationManager.ConfigurationManager method), 71  
get\_driver\_components() (configurationManager.ConfigurationManager method), 71  
get\_elapsed\_time() (portalBridge.PortalBridge method), 79  
get\_finished\_tasks() (services.ServicesProxy method), 73  
get\_finished\_tasks\_status() (services.TaskPool method), 77  
get\_framework\_components() (configurationManager.ConfigurationManager method), 71  
get\_framework\_logger() (configurationManager.ConfigurationManager method), 71

`get_init_components()` (configurationManager.ConfigurationManager method), 71  
`get_inq()` (ips.Framework method), 64  
`get_message_id()` (messages.Message method), 78  
`get_pbs_info()` (in module resourceHelper), 70  
`get_platform_parameter()` (configurationManager.ConfigurationManager method), 71  
`get_port()` (configurationManager.ConfigurationManager method), 71  
`get_port()` (services.ServicesProxy method), 74  
`get_qstat_jobinfo()` (in module resourceHelper), 70  
`get_restart_files()` (services.ServicesProxy method), 74  
`get_sim_names()` (configurationManager.ConfigurationManager method), 72  
`get_sim_parameter()` (configurationManager.ConfigurationManager method), 72  
`get_slurm_info()` (in module resourceHelper), 70  
`get_task_id()` (taskManager.TaskManager method), 66  
`get_time_loop()` (services.ServicesProxy method), 74  
`get_topo()` (in module resourceHelper), 70  
`get_working_dir()` (services.ServicesProxy method), 74  
`getGlobalConfigParameter()` (services.ServicesProxy method), 73  
`getPort()` (configurationManager.ConfigurationManager method), 71  
`getPort()` (services.ServicesProxy method), 73  
`getResourceList()` (in module resourceHelper), 70  
`getTimeLoop()` (services.ServicesProxy method), 73  
`getTimeString()` (in module ipsutil), 78

## I

`IncompleteCallException`, 78  
`info()` (ips.Framework method), 64  
`info()` (services.ServicesProxy method), 74  
`init()` (component.Component method), 71  
`init()` (portalBridge.PortalBridge method), 79  
`init_call()` (taskManager.TaskManager method), 66  
`init_simulation()` (portalBridge.PortalBridge method), 80  
`init_task()` (taskManager.TaskManager method), 66  
`init_task_pool()` (taskManager.TaskManager method), 66  
`initialize()` (configurationManager.ConfigurationManager method), 72  
`initialize()` (resourceManager.ResourceManager method), 68  
`initialize()` (taskManager.TaskManager method), 66  
`InsufficientResourcesException`, 78  
`InvalidResourceSettingsException`, 78  
`ips` (module), 63  
`ipsExceptions` (module), 77  
`ipsutil` (module), 78

## K

`kill_all_tasks()` (services.ServicesProxy method), 74  
`kill_task()` (services.ServicesProxy method), 74

## L

`launch_task()` (services.ServicesProxy method), 74  
`launch_task_pool()` (services.ServicesProxy method), 75  
`launch_task_resilient()` (services.ServicesProxy method), 75  
`log()` (ips.Framework method), 64  
`log()` (services.ServicesProxy method), 75

## M

`manual_detection()` (in module resourceHelper), 70  
`merge_current_plasma_state()` (dataManager.DataManager method), 65  
`merge_current_plasma_state()` (services.ServicesProxy method), 75  
`Message` (class in messages), 78  
`messages` (module), 78  
`MethodInvokeMessage` (class in messages), 78  
`MethodResultMessage` (class in messages), 79

## N

`Node` (class in node\_structure), 69  
`NonexistentResourceException`, 78

## P

`PortalBridge` (class in portalBridge), 79  
`portalBridge` (module), 79  
`PortalBridge.SimulationData` (class in portalBridge), 79  
`print_cores()` (node\_structure.Socket method), 69  
`print_sockets()` (node\_structure.Node method), 69  
`printCurrTaskTable()` (taskManager.TaskManager method), 66  
`printRMState()` (resourceManager.ResourceManager method), 68  
`process_event()` (portalBridge.PortalBridge method), 80  
`process_events()` (services.ServicesProxy method), 75  
`process_FTB_events()` (resourceManager.ResourceManager method), 68  
`process_service_request()` (configurationManager.ConfigurationManager method), 72  
`process_service_request()` (dataManager.DataManager method), 65  
`process_service_request()` (resourceManager.ResourceManager method), 68  
`process_service_request()` (taskManager.TaskManager method), 67  
`publish()` (services.ServicesProxy method), 75

## R

`register_service_handler()` (ips.Framework method), 64  
`release()` (node\_structure.Core method), 70  
`release()` (node\_structure.Node method), 69  
`release()` (node\_structure.Socket method), 70

release\_allocation() (resourceManager.ResourceManager method), 68  
 ReleaseMismatchException, 78  
 remove\_task\_pool() (services.ServicesProxy method), 75  
 report\_RM\_status() (resourceManager.ResourceManager method), 68  
 resourceHelper (module), 70  
 ResourceManager (class in resourceManager), 67  
 resourceManager (module), 67  
 ResourceRequestMismatchException, 78  
 restart() (component.Component method), 71  
 return\_call() (taskManager.TaskManager method), 67  
 run() (ips.Framework method), 64

## S

save\_restart\_files() (services.ServicesProxy method), 75  
 send\_event() (portalBridge.PortalBridge method), 80  
 send\_portal\_event() (services.ServicesProxy method), 75  
 sendEncodedMessage() (in module sendPost), 79  
 sendEvent() (resourceManager.ResourceManager method), 69  
 sendPost (module), 79  
 ServiceRequestMessage (class in messages), 79  
 ServiceResponseMessage (class in messages), 79  
 services (module), 72  
 ServicesProxy (class in services), 72  
 set\_config\_param() (services.ServicesProxy method), 75  
 set\_config\_parameter() (configurationManager.ConfigurationManager method), 72  
 setMonitorURL() (services.ServicesProxy method), 75  
 Socket (class in node\_structure), 69  
 stage\_input\_files() (services.ServicesProxy method), 75  
 stage\_output\_files() (services.ServicesProxy method), 75  
 stage\_plasma\_state() (dataManager.DataManager method), 65  
 stage\_plasma\_state() (services.ServicesProxy method), 76  
 stage\_replay\_output\_files() (services.ServicesProxy method), 76  
 stage\_replay\_plasma\_files() (services.ServicesProxy method), 76  
 stageCurrentPlasmaState() (services.ServicesProxy method), 75  
 stageInputFiles() (services.ServicesProxy method), 75  
 stageOutputFiles() (services.ServicesProxy method), 75  
 step() (component.Component method), 71  
 step() (portalBridge.PortalBridge method), 80  
 submit\_tasks() (services.ServicesProxy method), 76  
 submit\_tasks() (services.TaskPool method), 77  
 submit\_tasks\_old() (services.TaskPool method), 77  
 subscribe() (services.ServicesProxy method), 76

## T

Task (class in services), 77

TaskManager (class in taskManager), 65  
 taskManager (module), 65  
 TaskPool (class in services), 77  
 terminate() (component.Component method), 71  
 terminate() (configurationManager.ConfigurationManager method), 72  
 terminate\_sim() (ips.Framework method), 64  
 terminate\_tasks() (services.TaskPool method), 77

## U

unsubscribe() (services.ServicesProxy method), 76  
 update\_plasma\_state() (dataManager.DataManager method), 65  
 update\_plasma\_state() (services.ServicesProxy method), 76  
 update\_time\_stamp() (services.ServicesProxy method), 76  
 updatePlasmaState() (services.ServicesProxy method), 76  
 updateTimeStamp() (services.ServicesProxy method), 76

## W

wait\_call() (services.ServicesProxy method), 76  
 wait\_call() (taskManager.TaskManager method), 67  
 wait\_call\_list() (services.ServicesProxy method), 76  
 wait\_task() (services.ServicesProxy method), 76  
 wait\_task\_nonblocking() (services.ServicesProxy method), 76  
 wait\_task\_resilient() (services.ServicesProxy method), 77  
 wait\_tasklist() (services.ServicesProxy method), 77  
 warning() (ips.Framework method), 64  
 warning() (services.ServicesProxy method), 77