USING THE INCARNATION DATABASE (V4.0)

Sven van den Berghe, ***Fujitsu Laboratories of Europe***

Version 4.0.3 23rd July 2003

Version 4.0.1 14th February 2003

Using this document and studying the example IDB(s) should allow you to adapt an IDB to your requirements.

This document assumes knowledge of the Unicore architecture and of the AJO.

NOTE: The following sections are no longer valid and must be removed from any IDBs; IMPORT, EXPORT, COPY_FILE.

Their replacement is FILE_COPY (from NJS 4.0.0 Beta 10 onwards).

1 Incarnation and Execution

Executing an AbstractAction on a target system is a two-stage process. Firstly, the AbstractAction is incarnated by the NJS: incarnation converts the abstract expression of a task and its options into site and vendor specific commands. The incarnated commands are then passed to a Target System Interface (TSI), which is responsible for executing the commands.

The NJS also creates and incarnates some actions of its own e.g. actions to set up a Uspace, remove a Uspace, retrieve stdout and stderr and query the status of executing jobs.

The NJS is generic with the same code applying to all possible target systems. Since the NJS is generic, it needs to be initialised with the information that it needs for incarnation and to contact and control the TSIs. This information will be specific to each site and to each target system. This part of the NJS initialisation is done through the Incarnation Database (IDB).

1.1 Comments

- TSIs interact directly with the execution mechanism of the target system (NQS, Unix batch jobs etc) and so need to be tailored to each target system.
- TSIs run as processes separate from the NJS and on the actual target system (the NJS usually runs on a workstation remote from the target system).
- TSIs are intended to be lightweight and stateless providing efficient execution of the tasks with minimum coding and installation.
- TSIs have been written in Perl (for flexibility) but the protocol between the NJS and its TSIs is text based and so any language could be used to implement them.
- The NJS assumes that that a Bourne Shell is available.
- The TSI executes the incarnated tasks and so it is the only place where non-normal user permissions are required – the NJS can run as a normal user.

2 Syntax

An IDB is split into sections. A section is started by a section keyword and ended by the END keyword. Each section has its own set of valid keywords and subsections.

The sections are described in more detail below.

2.1 General Rules

Keywords start on a new line, there cannot be more than one keyword on a line.

Blank lines are allowed.

Any characters between a "#" and the end of a line are treated as comments.

White space is used to delimit words but any amount is allowed.

2.1.1 Word

A **word** is text delimited by white space or a comment character. The string **word** is used throughout the rest of this document to refer to text in this format.

For example:

```
this_is_a_word
```

is a **word**, but

```
this is a word
```

is not a **word** (it is a list of 4 **word**s).

2.1.2 List

A list is a group of words (up to the end of the line) delimited by white space or by commas.

```
This is a list
```

is a list.

2.1.3 End_of_line

The text in an **end_of_line** starts at the end of the keyword and continues to the end of the line.

2.1.4 Verbatim

verbatim is delimited by "[" and the matching "]". This text is copied directly into the command being created.

If the verbatim text includes "[" or "]", then opening and closing braces must match, even if escaped).

verbatim text may also be start with a double quote. It then ends at the next double quote.

Finally, where the context is unambiguous verbatim text can be delimited by white space (if the first non-space character is not a double quote or "[", then the text is delimited by white space).

Inside **verbatim** text a new line (\n) immediately preceded by a "-", indicates concatenation of the two lines (the "-" is discarded).

For example the following verbatim text:

```
[ this is a line of text
  but this two lines –
  will become one ]
```

Is placed into scripts as:

```
 this is a line of text
  but this two lines will become one
```

2.2 Pre-processing

The reading of an IDB includes a very simple pre-processing phase, which scans for lines with the format –DEFINE and –INCLUDE.

2.2.1 DEFINE

-DEFINE *string replacement*

or

#DEFINE *string replacement*

(the –DEFINE or #DEFINE must start the line, *string* must not contain spaces, *replacement* is the rest of the line, up to a “#” or “!”)

The pre-processor replaces every occurrence of *string* by *replacement* in all the following lines.

A line of the form:

`-ECHO`

or

`#ECHO`

will cause all following lines to be echoed to the log file after pre-processing.

2.2.2 *INCLUDE*

`-INCLUDE` *`file_name`*

Read the named file and place its contents at this point in the IDB. INCLUDEd files can INCLUDE other files.

2.3 Localisation of commands used by the NJS

The NJS performs a number of housekeeping tasks on Uspaces. These tasks use standard Unix commands and so are not defined in the IDB. The location of these commands may change on different systems and so the NJS will read the following values from the IDB for these commands:

`-DEFINE CAT_CMD xx` will use xx for Unix `/bin/cat`

`-DEFINE COPY_CMD xx` will use xx for Unix `/bin/cp –r`

Care must be taken to ensure that the incarnated copy command follows symbolic links and does not just copy them. This is the default behaviour on most systems. However, under Linux the default behaviour is to copy a symbolic link. This can be corrected by using “cp –rL”.

`-DEFINE FIND_CMD xx` will use xx for Unix `/bin/find`

`-DEFINE LS_CMD xx` will use xx for Unix `/bin/ls`

`-DEFINE LSA_CMD xx` will use xx for Unix `/bin/ls –A`[1]

`-DEFINE LN_CMD xx` will use xx for Unix `/bin/ln -s`

`-DEFINE MKDIR_CMD xx` will use xx for Unix `/bin/mkdir –p –m700`

`-DEFINE MV_CMD xx` will use xx for Unix `/bin/mv`

`-DEFINE PF_CMD xx` will use xx for Unix `/usr/bin/printf`

`-DEFINE RM_CMD xx` will use xx for Unix `/bin/rm`

`-DEFINE SH_CMD xx` will use xx for Unix `/bin/sh`

`-DEFINE SED_CMD xx` will use xx for Unix `/bin/sed`

`-DEFINE TR_CMD xx` will use xx for Unix `/usr/bin/tr –s`

`-DEFINE TOUCH_CMD xx` will use xx for Unix `/usr/bin/touch`

`-DEFINE MKFIFO_CMD xx will use xx for /bin/mkfifo –m600`

The setting of these commands can be tested in a running NJS by using the “test_commands” command in the “njs_admin” utility.

---

[1] list all files, including hidden, but not “.” or “..”

2.4 Capacity Resources

The values in the capacity resources are sent to clients as the limits on the resources available at the Vsite. These values are sent to the client even if the queue limits are greater than the values set here.

A Capacity Resource is a resource that needs to be requested by type and amount; for example a certain amount of memory. Most capacity resources also have limits on the amount of resource available. The syntax of a Capacity Resource description is as follows:

```
RESOURCE_NAME description
   MAXIMUM maximum
   MINIMUM minimum
   DEFAULT default
```

Where *RESOURCE_NAME* is a keyword that is the name of the resource being described, *description* is **verbatim** and is a string to describe the resource, *maximum* is the maximum of the resource allowed at the site, *minimum* is the minimum of the resource allowed by the site and *default* the amount of resource that will be allocated to n incarnated task if there is no explicit request and it is necessary to supply a value. *maximum*, *minimum* and *default* are **verbatim** and interpreted as numbers.

2.5 Action Descriptions

All action descriptions have two parts. **invocation definitions**, which define the lines of script to be used to incarnate a task in a Software Resource, and **field definitions**, which define how to define fields (perhaps in different Software Resources).

2.5.1 Invocation Definitions

Incarnation definitions define the script lines to be used to incarnate an action. They have the following structure:

```
INVOCATION word1 word2 verbatim
```

Where **word1** is a Software Resource identifier which names the Software Resource to which the definition applies.

The Software Resource must have been defined for the TSI.

A Software Resource identifier is made up of the Software Resource name and version string separated by a hyphen (if there is no version information, then the hyphen is omitted) e.g.

GAUSSIAN-66.vA

Is the Software Resource identifier for a GAUSSIAN software resource with version 66.vA.

If **word1** is absent, then this incarnation is the default and will be used when no Software Resources are present in the task. The default incarnation is also used as the base for the incarnation building process (see below).

Every action definition must have a default incarnation

word2 is optional. If it is the string “INTERACTIVE”, then the TSI is signalled that the incarnation of tasks asking for the Software Resource should be executed “interactively”.

Note that there is no code in the current TSIs to do interactive execution and so this will usually have no effect.

The **verbatim** section defines the script lines to use. It consists of text, which is passed (almost) verbatim to the incarnated task, and fields within “<” and “>” which are substituted by the NJS according to the rules given in the field definitions and the option values chosen by the user. Every valid field for the task must appear in each incarnation definition.

Building incarnations

An incarnation is built up by applying the definitions in the order that they are encountered in the IDB if the Software Resource is selected in the task. The special field <STANDARD> can be used to substitute the incarnation so far (if <STANDARD> is selected, then no other field can be present).

For example the IDB text:

```
INVOCATION [./<RUNCOMMAND>]

INVOCATION MPI1 [aprun <RUNCOMMAND>]

INVOCATION PVM [aprun <STANDARD>

INVOCATION TIME_ON <time <STANDARD>]
```

Produces the following commands if <RUNCOMMAND> incarnates to a.out:

By default with no Software Resources in the task's resource request:

```
./a.out
```

The task selects the MPI1 Software Resource only:

```
aprun a.out
```

The task select the PVM Software Resource:

```
aprun ./a.out
```

The task selects the option to produce timing information:

```
time ./a.out
```

The task selects both MPI1 and timing:

```
time aprun a.out
```

Manipulation of strings

The incarnation of most fields can be instructed to escape shell metacharacters (*,?,$,[,]) with the syntax:

<field_name%>.

The <STANDARD> field can be modified by simple text processing:

> <field_name/A/B/> means substitute every occurrence of string A by string B in the incarnation so far
>
> <field_name+C> means append string C to the incarnation so far.

The **verbatim** text can be modified by special characters immediately before the start of a field (i.e. before the "<"):

> If the last character before a "<" is a "*", then the following field is split into words and each word prepended by the verbatim text. For example: –l*<LIBRARY> when <LIBRARY> incarnates to "m c mpi" will be written to the incarnated job as"–lm –lc –lmpi")
>
> If the last character is a before a "<" is "_", then the field and the preceding verbatim text are placed in the incarnation only if the field incarnates to a string with non-zero length.

2.5.2 *Field definitions*

Some of the fields in the invocation definition have options (e.g. compiler optimisation levels). The incarnation of these options is specified by a field definition with the following format:

```
FIELD_NAME
```

```
  OPTION_NAME incarnation
  OPTION_NAME incarnation
  ......
END
```

Incarnation is a **word**. If it is an underscore (“_”), then no text is placed in the incarnation of that field.

Fields are usually defined for all Software Resources. However, it is possible to redefine a field for a particular Software Resource by placing it within the following structure:

```
SOFTWARE_RESOURCE software_resource
...
END
```

Where *software_resource* is a **word** and is the name of the Software Resource to which these changes apply. More than one field can be modified within a SOFTWARE_RESOURCE section.

For example to redefine the naming of files so that preprocessed Fortran files have a .F extension and non-preprocessed files have a .f extension:

```
NAMING
  SOURCE_FILE .f
  ..
END
SOFTWARE_RESOURCE PREPROCESS
  NAMING
    SOURCE_FILE .F
    ..
  END
END
```

2.5.3 *Naming*

Unicore assumes certain file name extensions for certain types of files. These may not be the same as those required by a target system. The NJS is told of system specific file name extensions through this construct:

```
NAMING
FILE_TYPE name
FILE_TYPE name
..
END
```

Where *name* is the file name for this file type. If *name* starts with a full point (“.”), then it is treated as an extension and the NJS replaces incoming extensions with the system specific extension. Otherwise *name* is treated as the complete file name.

2.6 Multiple TSIs

The NJS is usually configured to use a single TSI. However, in some cases it may be desirable to use multiple TSIs (e.g. to allow cross-compilation). Secondary TSIs can be defined by placing additional EXECUTION_TSI sections in the IDB.

The secondary TSIs must be named (and their definitions must appear after the – unnamed – primary, or default, TSI).

Tasks can be specialised for secondary TSIs by repeating the section and including a DEFINITION_FOR section.

2.6.1 DEFINITION_FOR section

The DEFINITION_FOR section tells the NJS which TSIs can use the particular definition. The format is:

`DEFINITION_FOR list END`

Where *list* is a **list** of the names of previously defined TSI that can use this definition. There are two special names, ALL says that this definition applies to all TSIs and DEFAULT says that this definition applies to the default TSI.

The DEFINITION_FOR section is optional. If it is not present, then the definition is for the default TSI (the first Execution TSI in the IDB file).

3 GENERAL section

The GENERAL section describes properties that apply to the whole Vsite and to all task incarnations.

There are seven keywords recognised in the GENERAL section; USPACE_ROOT, OUTCOME_ROOT, SPOOL_ROOT, HEADER, BROKER, TextInfoResource, NumericInfoResource.

TextInfoResource and NumericInfoResource are abstract carriers of information. They are designed to pass information from the site administrators to a client and have no relevance to the operation of the NJS. One use of these resources is to replace the Resource Pages that were used in Unicore 1.0. The syntax of the information is determined by the client, suggestions are given in the examples below.

The _ROOT keywords refer to directories that the NJS needs to use to store files. The directory names are interpreted relative to the current working directory of the TSI. These directory names are not necessarily passed through a shell and so will not expand any environment variables except for two: $HOME and $USER.

The _ROOT directories must exist and should be writable by all the users that will write files to them (generally mode 777 unless there is a $HOME or $USER in the name, when they can be limited to just the user).

The default is to use the TSI's current working directory.

These directories are all cleaned up by the commands defined in the CLEANUP section.

3.1 USPACE_ROOT keyword

The USPACE_ROOT keyword tells the NJS where it can place the Uspaces of the executing AJOs. The NJS will create a subdirectory of the UPSACE_ROOT directory for each executing AJO.

The format of the USPACE_ROOT keyword is:

```
USPACE_ROOT  directory
```

Where *directory* is a **word** and is name of the directory under which the Uspaces should be created.

3.2 OUTCOME_ROOT keyword

The OUTCOME_ROOT keyword tells the NJS where it can store files that are part of the Outcome of an AJO. The NJS will create a subdirectory of the OUTCOME_ROOT directory for each executing AJO that produces Outcome files.

The format of the OUTCOME_ROOT keyword is:

```
OUTCOME_ROOT  directory
```

Where *directory* is a **word** and is the name of the directory under which the Outcomes should be stored.

3.3 SPOOL_ROOT keyword

The SPOOL_ROOT keyword tells the NJS where it can store files that are spooled by a Spool task. The NJS will create a subdirectory of the SPOOL_ROOT directory for each spooled Portfolio.

The format of the SPOOL_ROOT keyword is:

```
SPOOL_ROOT  directory
```

Where *directory* is a **word** and is the name of the directory under which the spooled files should be stored.

3.4 HEADER keyword

The HEADER keyword supplies text that the NJS adds to the header of every script that it produces. It can be used to supply a common environment to all incarnated AbstractActions or to implement any pre-processing that is required by a site.

All the scripts created by the NJS use the Bourne shell.

The format of the HEADER keyword is:

`HEADER` *`text`*

Where *text* is **verbatim** and copied directly from the IDB to the start of every script.

3.5 BROKER keyword

The BROKER keyword tells the NJS the name of a class to load to perform resource brokering and resource checking functions. The class must implement one of the following interfaces:

- com.fujitsu.arcon.njs.interfaces.ResourceChecker
- com.fujitsu.arcon.njs.interfaces.ResourceBroker

The format of the BROKER keyword is:

`BROKER classname` *`initialisation`*

Where *classname* is a **word** and is the name of the class to load to perform the brokering functions.

Where *initialisation* is **verbatim**. This string is passed through to the initialisation of the brokering/checking class.

This keyword is optional. If it is not used, then the NJS will not perform these functions (any tasks requesting resource brokering or resource checking will fail).

If this keyword is used, then the NJS will use these classes to execute any org.unicore.ajo.CheckResources and org.unicore.ajo.CheckQoS tasks that it receives. It will also add instances of org.unicore.resources.QoSCheckResource and org.unicore.resources.ResourceCheckResource to its advertised resources.

If the implemented interface is com.fujitsu.arcon.njs.interfaces.ResourceChecker, then only org.unicore.resources.QoSCheckResource is added.

3.6 TextInfoResource

The TextInfoResource is copied into the resource description of the Vsite as an instance of org.unicore.resources.TextInfoResource. The NJS does nothing else with this resource.

This keyword can appear as many times as required.

The format of TextInfoResource is:

`TextInfoResource` *`description`*

`Tag` *`tag`*

`Value` *`value`*

Where *description* is **verbatim** and is a description of the information that is contained in the tag and value sections, *tag* is **verbatim** and defines the information type and *value* is **verbatim** and is the information.

3.7 NumericInfoResource

The NumericInfoResource is copied into the resource description of the Vsite as an instance of org.unicore.resources.NumericInfoResource. The NJS does nothing else with this resource.

This keyword can appear as many times as required.

The format of NumericInfoResource is:

```
NumericInfoResource description
Tag tag
Value value
```

Where *description* is **verbatim** and is a description of the information that is contained in the tag and value sections, *tag* is **verbatim** and defines the information type and *value* is **verbatim** and is the information, *value* must be convertible into a number.

3.8 Example

Tell the NJS to create all Uspaces as sub-directories of /UNICORE/uspaces, to put all outcomes into user specific directories under /UNICORE/uspaces/Outcomes/ and to write spooled files into each user's home file space.

Two variables are added to the environment of every script created by the NJS

Passes the remaining fields to the client.

```
GENERAL
  USPACE_ROOT /UNICORE/uspaces
  OUTCOME_ROOT /UNICORE/uspaces/Outcomes/$USER
  SPOOL_ROOT $HOME/UNICORE_SPOOL
  HEADER [
    UNICORE_JOB=true; export UNICORE_JOB
    UC_TARGET=VPP; export UC_TARGET
  ]

  TextInfoResource [ The full name of the Usite. ]
    TAG [ Site name ]
    VALUE [
      Fujitsu European Centre for Information
Technology ]

  TextInfoResource [ The short name of the Usite. ]
   TAG [ Site short name ]  VALUE [ FECIT ]

  TextInfoResource [ The Usite's primary contact
person. ]
   TAG [ Responsible contact ]
   VALUE [ Dr. David F. Snelling ]

  TextInfoResource [ The informal name of the Vsite. ]
   TAG [ Vsite name ] VALUE [ Fecit_VPP ]
  TextInfoResource [ The Usite's contact email
address. ]
   TAG [ Contact email ] VALUE [ snelling@fecit.co.uk
]


  NumericInfoResource [ Total Vsite performance
Gflop/s. ]
      TAG [ Site performance (Gflop/s) ]
      VALUE [ 8.8 ]

 # Vsite Resources
 TextInfoResource [ The Vsite architecture ]
   TAG [ Architecture ]
   VALUE [ VPP/300 ]

 TextInfoResource [ The Vsite operating system ]
   TAG [ Operating System ]
   VALUE [ UXP/V, version UXP/V ]

 TextInfoResource [XML Resource Pages]
    TAG[Unicore XML Resource Pages V1.0]
    VALUE […………]
END
```

4 EXECUTION_TSI[2] section

Describes the TSI that used by the NJS to execute all incarnated scripts.

4.1 NAME keyword

A name for the TSI being described, this is optional.

The NAME keyword should appear at most once in a EXECUTION_TSI section.

The format of the NAME keyword is:

NAME *name*

Where *name* is a **word** and is the name of the TSI.

4.2 SOURCE keyword

How the TSI will contact the NJS.

The format of the SOURCE field is:

SOURCE *machine_name in_port out_port*

Where:

machine_name is a **word** and is the name of the machine on which the TSI executes.

in_port is a **word** and is the number of the port on which the TSI processes will contact the NJS.

out_port is a **word** and is the number of the port on which the TSI daemon processes is listening for requests from the NJS to start a new process.

The NJS to TSI connections are only allowed between two specified machines using specified port numbers (this is done so that the TSI, which is a process with root privileges, can be sure that it is accepting commands from a genuine NJS). Thus the NJS will only connect to TSI running on *machine_name* .

There is a two-stage protocol to start a TSI process. When the NJS detects the need for a new TSI process it will contact the TSI shepherd process (running on *machine_name* listening on *out_port).* The TSI shepherd will create a TSI process that will contact the NJS (on *in_port*).

4.3 STORAGE keyword

The STORAGE keyword introduces a description of a (file) storage resource.

The format of the STORAGE field is:

`STORAGE` *`description path`* `NAME` *`name`* `DEFAULT` *`default`*
`MAXIMUM` *`maximum`* `MINIMUM` *`minimum`*

Where:

description is **verbatim** text that describes the resource to users

path is the path prepended to all file names on this storage

name is the name of the storage resource

default, *maximum, minimum* are the limits on the storage (in megabytes)

If the *name* is “ROOT”, then an instance of org.unicore.resources.Root is created that overrides the default instance (which has a *path* of “/” and limits from 0.0 to

---

[2] An alternative form of this keyword is EXECUTION_TSI_Q which differs only in the run time behaviour for job-status polling. The normal form assumes that a job-status poll will return results for all executing jobs, the alternative form that results are returned only for a particular job. The alternative form is much less efficient than the normal form and should only be used if the normal cannot be used. The alternative form requires changes to the TSI code.

the maximum floating point (double) value and a default of 0.01). PATH is optional.

If the *name* is "USPACE", then an instance of org.unicore.resources.USpace is created that overrides the default instance (which has a *path* of the value of USPACE_ROOT and limits from 0.0 to the maximum floating point (double) value and a default of 0.01, (a matching instance of org.unicore.resources.AlternativeUspace is also created). PATH is optional.

If the *name* is "SPOOL", then an instance of org.unicore.resources.Spool is created that overrides the default instance (which has a *path* of the value of SPOOL_ROOT and limits from 0.0 to the maximum floating point (double) value and a default of 0.01). PATH is optional.

If the *name* is "HOME", then an instance of org.unicore.resources.Home is created that overrides the default instance (which has a *path* of "$HOME" and limits from 0.0 to the maximum floating point (double) value and a default of 0.01). PATH is optional.

If the *name* is "TEMP", then an instance of org.unicore.resources.Temp is created. PATH is required.

For all other values for *name* an instance of org.unicore.resources.StorageServer is created with the given name. . PATH is required.

4.3.1 Example

```
STORAGE "Test storage server" /foo/bar NAME on_foo
DEFAULT 10 MAXIMUM 100 MINIMUM 1

STORAGE on_root NAME root DEFAULT 10 MAXIMUM 100
MINIMUM 1

STORAGE on_uspace NAME uspace DEFAULT 10 MAXIMUM 100
MINIMUM 1

STORAGE on_spool  NAME spool DEFAULT 10 MAXIMUM 100
MINIMUM 1

STORAGE on_home  NAME Home $HOME DEFAULT 10 MAXIMUM
100 MINIMUM 1

STORAGE on_temp NAME temp /local/temp DEFAULT 10
MAXIMUM 100 MINIMUM 1
```

4.4 NODE keyword

The NODE keyword describes a *Capacity Resource* for the number of nodes available to the EXECUTION_TSI.

The NODE keyword must appear once in the EXECUTION_TSI section. The limits set by this keyword must be accepted by at least one of the defined queues (if any queues are defined).

4.4.1 Example

```
Node [ The number of nodes available to a batch job ]

  DEFAULT [ 64 ]

  MAXIMUM [ 256 ]

  MINIMUM [ 1 ]
```

This tells the NJS that it has between 0 and 256 nodes available for batch jobs on the TSI, with a default value of 64 nodes.

4.5 PROCESSOR keyword

The PROCESSOR keyword describes a *Capacity Resource* for the number of processors per node available to this EXECUTION_TSI.

The systems that can be described in the IDB are those with a collection of homogeneous shared memory nodes, each node having the same number of processors.

The PROCESSOR keyword is optional. It defaults to 1 processor per node for all fields.

The limits set by this keyword must be accepted by at least one of the defined queues (if any queues are defined).

4.5.1 Example

```
Processor [ The number of processors per node ]

  DEFAULT [ 16 ]

  MAXIMUM [ 16 ]

  MINIMUM [ 1 ]
```

This tells the NJS that it has between 1 and 16 processors per node, with a default value of 16 processors per node.

4.6 SOFTWARE_RESOURCE keyword

Description of a Software Resource supported by the EXECUTION_TSI.

If a Software Resource is used to modify the incarnation of an action, then it must appear in the EXECUTION_TSI section(s) for which the action is being defined.

The format of a SOFTWARE_RESOURCE is:

SOFTWARE_RESOURCE *description type name version*

Where *description* is **verbatim** and can be used to describe the Software resource to users. *Description* is optional.

Where *type* is a **word** (either APPLICATION or CONTEXT) which describes the type of Software resource being defined. Contexts include libraries (e.g MPI) or execution contexts (such as a timing run or a run with debug). *Type* is optional (but may be made required in a future release of the NJS).

Where *name* is a **word** and is the name of the Software Resource and *version* is a **word** and is the version information of the Software Resource (*version* is optional).

Note that the Software Resource identifier that is used in the definitions of the incarnations combines the name and version information (see Section 2.5).

4.6.1 Example

SOFTWARE_RESOURCE [A weather forecast model] APPLICATION IFS 3.45

4.7 APPLICATION keyword

The APPLICATION keyword is an alternative way to describe a SOFTWARE_RESOURCE of type APPLICATION.

The format of APPLICATION is:

APPLICATION *description name version meta_data_file_name*

Where *description* is **verbatim** and can be used to describe the Software resource to users. *Description* is optional.

Where *name* is a **word** and is the name of the Software Resource and *version* is a **word** and is the version information of the Software Resource (*name* and *version* are required).

Note that the Software Resource identifier that is used in the definitions of the incarnations combines the name and version information (see Section 2.5).

meta_data_file_name is the name of a file whose contents are read where the NJS starts up and are sent with the Application resource as its "MetaData" field (see the AJO documentation). (*meta_data_file_name* is optional)

4.7.1 *Example*

APPLICATION "A weather forecast model" IFS 3.45 gui.xml

4.8 CPUTIME keyword

The CPUTIME keyword describes a *Capacity Resource* for the amount of time that a task (batch job) is allowed with the addition of another field to describe the speed of the processors[3].

The units of the fields are seconds.

The format of the additional field is:

`RATE number`

Where *number* is **verbatim** (interpreted as a number) and is the (peak) floating point performance of a processor in megaflop per second per processor

The CPUTIME keyword must appear once in each EXECUTION_TSI section. The limits set by this keyword must be accepted by at least one of the defined queues (if any queues are defined).

4.8.1 *Example*

```
CPUTime [ Limits on time for jobs ]
      DEFAULT [ 1000 ]
      MAXIMUM [ 14400 ]
      MINIMUM [ 1 ]
      RATE [ 900 ]
```

This tells the NJS that the site allows between 1 and 14400 seconds time and that each of the processors is rated at 900 megaflop per second.

4.9 MEMORY keyword

The MEMORY keyword describes a *Capacity Resource* for the amount of memory that a task requires.

The units of the fields are megabytes per node.

The MEMORY keyword must appear once in each EXECUTION_TSI section. The limits set by this keyword must be accepted by at least one of the defined queues (if any queues are defined)..

4.9.1 *Example*

```
Memory [ Limits on the memory (per node) of a batch
job ]
  DEFAULT [ 1000 ]
  MAXIMUM [ 1280 ]
  MINIMUM [ 1 ]
```

This tells the NJS that each node has between 1 and 1280 megabytes available and that a default value of 1000 megabytes per node will be supplied.

---

[3] The CPUTIME keyword generates instances of the org.unicore.resources.FloatingPoint and the org.unicore.resources.RunTimes resouce classes.

4.10 PER_NODE_LIMITS keyword

The PER_NODE_LIMITS is optional. If it appears, then the NJS creates memory resource requests for the batch sub-system that are expressed as per-node (this is the default).

The format of the PER_NODE_LIMITS keyword is:

```
PER_NODE_LIMITS
```

4.11 PER_JOB_LIMITS keyword

The PER_JOB_LIMITS is optional. If it appears, then the NJS creates memory resource requests for the batch sub-system that are expressed as per-job i.e. are the sum of the requirements for each processor in the job.

Note that the Unicore AJO expects all memory requests to be per-node so setting this option means that all requests from a user for memory are multiplied by the number of requested nodes before being sent to the BSS.

Note also that the resource description (MEMORY keyword) is *always* per-node.

The format of the PER_JOB_LIMITS keyword is:

```
PER_JOB_LIMITS
```

4.12 PER_PROCESSOR_LIMITS keyword

The PER_PROCESSOR_LIMITS is optional. If it appears, then the NJS creates memory resource requests for the batch sub-system that are expressed as per-processor.

Note that the Unicore AJO expects all memory requests to be per-node so setting this option means that all requests from a user for memory are divided by the number of requested processors per node before being sent to the BSS.

Note also that the resource description (MEMORYkeyword) is *always* per-node.

The format of the PER_PROCESSOR_LIMITS keyword is:

```
PER_PROCESSOR_LIMITS
```

4.13 DO_NOT_SEND_EMAIL keyword

The DO_NOT_SEND_EMAIL keyword is optional. Usually the NJS will submit batch jobs so that they will cause email to be sent whenever the batch job changes state (e.g. starts and finishes executing). This email is sent to the email address sent with the AJO. If a batch sub-system or site does not support the sending of external email, then this keyword can be used to stop the NJS submitting jobs with this option set.

The format of the DO_NOT_SEND_EMAIL keyword is:

```
DO_NOT_SEND_EMAIL
```

4.14 QSTAT_XLOGIN keyword

The QSTAT_XLOGIN keyword is passed by the NJS to the TSI whenever it requests a listing of the state of all batch jobs from the TSI.

The TSI sets its uid to this value before executing the query command (under NQS this is qstat). There are two uses of this keyword. Firstly it prevents the TSI from executing a command as root (all other commands are executed as an xlogin). It also provides a way for sites to fulfil the requirement that the qstat command return the state of *all* batch jobs (some sites limit the return from qstat to just the user’s jobs). If the QSTAT_XLOGIN is made an (NQS) administrator of the queues, then all jobs will be visible to it.

The format of the QSTAT_XLOGIN keyword is:

`QSTAT_XLOGIN` *`xlogin`*

Where *`xlogin`* is a word and is the login to use for the qstat commands.

This key word is required once for every Execution TSI.

4.15 OVERHEAD keyword

The OVERHEAD keyword tells the NJS the amount of time added to the execution of a typical script by the pre and post processing commands added by a site.

The format of the OVERHEAD keyword is:

```
OVERHEAD time
```

Where `time` is a **word** and is the amount by which the requested job time must be increased to account for the extra processing.

4.16 QUEUE subsection

The queue subsection tells the NJS which batch queues are available and what the resources limits are on these queues.

The format of the queues subsection is:

```
QUEUE

  NAME name

  PROCESSORS min max

  NODES min max

  TIME min max

  MEMORY min max

END
```

Where *name* is a **word** and is the name of the batch queue being described and *max* and *min* are **word**s which are the limits of each resource

- PROCESSORS is the limits on the number of processors per node (optional)
- NODES is the node count.
- TIME is the CPU time in seconds (for nodes with performance as described in the CPUTIME section)
- MEMORY is the per-node memory limits on the queue, in megabytes.

If the PER_JOB_LIMITS option is set, then the MEMORY limits are treated as job totals.

If the PER_PROCESSOR_LIMITS option is set, then the MEMORY limits are treated as job totals.

There are as many QUEUE subsections as there are queues on the target system.

Queue sections are optional, if there is no QUEUE description, then a single unnamed queue is assumed that has the limits of the resources described in the FLOATINGPOINT, MEMORY and other appropriate *Capacity Resource* sections.

There must always be a name for the queue. If the target BSS does not allow names for its queue, then using the string “noname” for the IDB queue name means that the TSI does not supply a queue name during job submittal.

Every queue name that appears in a PRIORITY should be described by a QUEUE section and every queue defined by a QUEUE section should appear in at least one PRIORITY.

4.17 PRIORITY subsection

The priority subsection tells the NJS how it should map each of the Unicore priorities to local queues. This is optional unless queues are explicitly described when it is required.

The format of the priority subsection is:

```
PRIORITY
  HIGH list
  DEVELOPMENT list
  NORMAL list
  LOW list
  WHENEVER list
END
```

Where *list* is a list of names of queues that can be used for tasks requesting the priority. The list entries are delimited by a comma, space or tab. Each list must have at least one element.

High, development, normal, low and whenever can appear 0 or 1 times.

The algorithm used to select a queue for a task is first to see if a queue in the Priority list matching the task's priority will accept jobs with the task's resource requirements. If not, then the next lower priority is searched.

4.18 Example of PRIORITY and QUEUE subsections

4.18.1 Definition for a single unnamed queue setup

```
QUEUE
  NAME noname
  MEMORY 1 2048
  TIME 1 7200
  NODES 0 256
END
PRIORITY
  HIGH noname
  DEVELOPMENT noname
  NORMAL noname
  LOW noname
  WHENEVER noname
END
```

4.18.2 Definition for multiple batch queues

```
QUEUE
  NAME fast
  MEMORY 1 2048
  TIME 1 7200
  NODES 0 16
END
QUEUE
  NAME slow
  MEMORY 1 2048
  TIME 1 7200
  NODES 0 256
END

PRIORITY
  HIGH fast
  DEVELOPMENT fast
  NORMAL slow
  LOW slow
  WHENEVER slow
END
```

In this system jobs that request less than 16 nodes and specify HIGH or DEVELOPMENT as their priority will be submitted to the FAST queue, All other jobs, including those specifying HIGH or DEVELOPMENT as their priority but with more than 16 nodes, will be submitted to the SLOW queue.

5 RUN section

The RUN section tells the NJS how to incarnate instances of org.unicore.ajo.ExecuteTask (run an executable). The executable can be a file whose name is supplied by the NJS (based on the user's input) or a script to be interpreted using one of four script interpreters.

The RUN section is an *Action description*. See the introductory section (2.5) on Action Descriptions for the syntax of these, including the INVOCATION keyword.

It requires invocations for all four of these contexts: PERL, KORN_SHELL, BOURNE_SHELL and C_SHELL.

The invocation definitions take a single field, RUNCOMMAND, which is incarnated by the NJS to the file containing the file to be executed.

The RUNCOMMAND is optional so UserTasks do not have to supply an executable where the incarnation does not use RUNCOMMAND.

The file name incarnated for RUNCOMMAND is prefixed by the full path to the Uspace (unless RUNNCOMMAND is preceded by a path e.g. "./").

Any Software Resources used to modify an invocation must be defined in the description of the TSIs for which the RUN is being described.

Note that the Software Resource identifier uses both the name and the version information from the SOFTWARE_RESOURCE keyword (se section 2.5).

The RUN section also recognises three predefined Software Resources:

- TIME_ON: used when UserTask.isTimeMeasured() returns true
- DEBUG: used when UserTask.is.runForDebug() returns true
- PROFILE: used when UserTask.is.runForProfile() returns true

The RUN section requires two field definitions: VERSION and VERBOSE (see Section 2.5.2)

VERSION controls the printing from the incarnated executable of version information it has two options (ON and OFF).

VERBOSE controls the printing from the incarnated executable of verbose (possibly debug) information it has two options (ON and OFF).

5.1 Defining Applications

An Application can be defined using the APPLICATION keyword.

The APPLICATION keyword has the following form:

`APPLICATION` *`name version`*

Where *name* is the name of the application and is **required** and *version* is the optional version information for the application.

The APPLICATION keyword introduces a section (terminated by the next END keyword). The APPLICATION, INVOCATION, DECSRIPTION, FILE and DATA keywords are recognised within this section.

The syntax is:

`APPLICATION` *`name version`*

An inner APPLICATION keyword has the same syntax as the outer and introduces a related application. It behaves the same way as an outer APPLICATION except that the name of the Application Resource is prefixed by the name of the outer Application followed by a ":", e.g. `OuterApp:InnerApp`

`INVOCATION`

The same syntax as the INVOCATION keyword

`DESCRIPTION verbatim`

Where *verbation* placed in the description field of the SiftwareResource.

`FILE file_name`

Where *file_name* is the name of a file whose contents will be appended to the Metadata field of the Application Resource representing this application.

`DATA verbatim`

Where *verbatim* is appended to the Application Resource's metadata field.

Applications defined using the APPLICATION keyword do not need separate definitions for their INVOCATION and their Software Resource; the complete definition is contained within the APPLICATION section.

Note that there is also a keyword named APPLICATION as part of the EXECUTION_TSI section. This has a different syntax and effect.

5.1.1 Example

The following lines of an IDB define an application "Package" that contains two related executables "Part1" and "Part2". The NJS's resources will contain Application Resources that are named "Package", "Package:Part1" and "Package:Part2".

```
APPLICATION Package v1
  DESCRIPTION "This is the main package"
  INVOCATION [/usr/apps/package/v1/bin/package data ]
  APPLICATION Part1
    DECSRIPTION "Prepare files for Package"
    INVOCATION [ …… ]
  END
  APPLICATION Part2
    DECSRIPTION "Tidy up after execution"
    INVOCATION [ …… ]
  END
  DATA [ First line of metadata ]
  FILE meta_data_file
  DATA [ Last line of metadata ]
END
```

5.2 Making Decisions

The Unicore AJO model contains two type of AbstractAction that can be controlled by values passed from an executable to the NJS (If and RepeatGroup). The value used to perform this control is called a "Decision".

The NJS sets the environment variable "UC_DECISION_FILE" for all executables. This contains the name of a file whose contents are read after execution of the job and returned to the NJS as the executable's Decision.

The NJS creates a Context (see Section 4.6) called "MakeReturnCodeDecision" which will make the executable's return code the Decision of the executable. This context is always available.

More advanced applications may wish to make non-numeric decisions. This can be done by modifying the incarnation as in the example below (note that jobs using this incarnation must not include the MakeReturnCodeDecision Context in their resources)

```
# The value of $the_last is interpreted by the NJS
# as the exit code of the execution
# $UC_DECISION_FILE will be set by the NJS to be
# the file that it reads for the Decision

INVOCATION test [ ./executable;
the_last=$?; the_end=1;
case $the_last in
0) echo "done" > $UC_DECISION_FILE
   ;;
1) echo "no file" > $UC_DECISION_FILE
   ;;
2) echo "no grid" > $UC_DECISION_FILE
   ;;
3) echo "not converged" > $UC_DECISION_FILE;
   $the_last = 0; # Get the NJS to treat this as
success
   ;;
*) echo $the_last > UC_DECISION_FILE
   ;;
exit $the_last

]
```

$the_last and $the_end are variables used by the NJS ($the_last is the return code of the executable, $the_end=1 indicates that execution got to the executable).

With this code the Decision is “done” if the executable returned 0 and as the return code is 0 the execution is considered successful by the NJS. If the return code is 1 or 2, then the NJS sets the Decisions to “no file” or “no grid” and fails execution. A return code of 3 would usually be regarded as a failed in execution by the NJS, however this code resets this to a successful execution but with a Decision of “not converged” to distinguish it from the done/successful.

5.3 Example

```
RUN

  INVOCATION [ <RUNCOMMAND> ]

  INVOCATION PERL

      [SHELL=/usr/bin/perl; $SHELL <RUNCOMMAND> ]

  INVOCATION KORN_SHELL

      [SHELL=/usr/bin/ksh; $SHELL <RUNCOMMAND> ]

  INVOCATION BOURNE_SHELL

      [SHELL=/usr/bin/sh; $SHELL <RUNCOMMAND> ]

  INVOCATION C_SHELL

      [SHELL=/usr/bin/csh; $SHELL <RUNCOMMAND> ]


  INVOCATION MPI-1
```

```
      [mpprun -n $UC_NODES -a <RUNCOMMAND> ]


  INVOCATION CPMD [/local/bin/our_packages/cpmd filea
]
END
```

All script interpreters are found in /usr/bin and this invocation ensures that the environment variable SHELL is set to the value of the executed shell and not one inherited from the scripts used by the TSI.

MPI is a Software Resource with identifier “MPI-1” (with a version “1” so the name here must concatenate these with a “-“).

The last definition does not use the RUNCOMMAND field and so all incarnations of the CPMD resource will execute the fixed string. There is no need to import the CPMD executable, but the AJO must ensure that *filea* exists.

6 FILE_COPY section

The FILE_COPY section tells the NJS how to copy file to and from Uspaces.

FILE_COPY is an Action definition section. See the introductory section (2.4) on Action Descriptions for the syntax of these, including the INVOCATION keyword.

The FILE_COPY section requires invocation definitions for two options of the task, these are: OVERWRITE and NO_OVERWRITE.

> OVERWRITE indicates that the user is prepared to allow the copied files to overwrite any existing files.
>
> NO_OVERWRITE indicates that the user wants the task to fail if the copy would overwrite any existing files (writing into existing directories is always allowed).

Each invocation requires two fields: SOURCE and DESTINATION. SOURCE is incarnated by the NJS into a list of file (or directory) names that need to be copied to DESTINATION (which is either a file or a directory). Directory copied are recursive.

NOTE: This section replaces the IMPORT, EXPORT and FILE_COPY sections that were used in versions of the NJS before 4.0.0 (and some beta releases of the 4.0.0 NJS)

6.1 Example

For a sample incarnation see the example IDB supplied with the NJS release.

7 CLEANUP section

The commands in the CLEANUP section are executed by the NJS when it needs to cleanup a directory created by the execution of an AJO. These directories include:

- the AJO's Uspace (which is cleaned up at the end of the execution of the AJO),
- a directory used to hold any Outcome files for an AJO (which is cleaned up when the NJS removes the AJO after receiving a RetrieveOutcomeAck for it)
- directories used to hold Spooled files (which are cleaned up by an UnSpool task)

The CLEANUP section is an *Action Description*. See the introductory section (2.4) on Action Descriptions for the syntax of these, including the INVOCATION keyword.

The CLEANUP section needs just one INVOVATION description (it does not recognise any conexts).

The invocation description requires one field, DIRECTORY, which is the directory to be cleaned up

7.1 Example

```
CLEANUP

 INVOCATION

     [ echo "Leaving Uspace intact for debug
<DIRECTORY>"]

END


# keep a Uspace but move out of the way for loops

CLEANUP

 INVOCATION

     [ mv <DIRECTORY> <DIRCETORY>_`date`]

END



CLEANUP

  INVOCATION [ rm –rf <DIRECTORY> ]

END
```

8 LIST_DIRECTORY section

The LIST_DIRECTORY section tells the NJS how produce listings of files within the storage servers.

LIST_DIRECTORY is an Action definition section. See the introductory section (2.4) on Action Descriptions for the syntax of these, including the INVOCATION keyword.

LIST_DIRECTORY requires invocation definitions for three options of the task, these are: AS_FILE, RECURSIVE and NOT_RECURSIVE. AS_FILE limits the listing to the file; the other options will list directory contents. RECURSIVE indicates the listing should include all subdirectories.

Each invocation description requires one field: TARGET. TARGET is the name of the file (directory) to be listed.

The Perl script “tsi_ls” supplied with the TSI produces listings of the required format. You need to change the IDB entry to refer to this script as installed on your system.

8.1 Format of returned listing

The NJS expects a certain format for the listing.

Listing starts with the line:

```
START_LISTING
```

and ends with the line:

```
END_LISTING
```

The files are listed in depth-first order. Each time a sub-directory is found the entry for the sub-directory file is listed and then entries for all the file in the subdirectory are listed.

The format for each listing line is:

Character 0 is blank, except:

If character 0 is '-', then the this line contains extra information about the file described in the previous line. This line is copied without change into the ListDirectory outcome entry for the file.

If character 0 is ‘<’, then all files in a sub-directory have been listed and the listing is continuing with the parent directory. This is required even when the listing is non-recursive.

Character 1 is 'D' if the file is a directory

Character 2 is "R" if the file is readable by the Xlogin (effective uid/gid)

Character 3 is "W" if the file is writable by the Xlogin (effective uid/gid)

Character 4 is "X" if the file is executable by the Xlogin (effective uid/gid)

Character 5 is "O" if the file is owned by the Xlogin (effective uid/gid)

Character 6 is a space.

Until the next space is a decimal integer which is the size of the file in bytes.

Until the next space is a decimal integer which is the last modification time of the file in seconds since the Unix epoch.

Until the end of line is the full path name of the file

Every line is terminated by \n

8.2 Example

```
LIST_DIRECTORY
  INVOCATION NOT_RECURSIVE [perl tsi_ls N <TARGET> ]
  INVOCATION RECURSIVE     [perl tsi_ls R <TARGET> ]
  INVOCATION AS_FILE       [perl tsi_ls A <TARGET> ]
END
```

9 RENAME_FILE section

The RENAME_FILE section tells the NJS how to move files within the storage servers. It is the incarnation of the RenameFile abstract task.

RENAME_FILE is an Action definition section. See the introductory section (2.4) on Action Descriptions for the syntax of these, including the INVOCATION keyword.

RENAME_FILE requires invocation descriptions for two options of the task, these are: OVERWRITE and NO_OVERWRITE.

> OVERWRITE indicates that the user is prepared to allow the renamed files to overwrite any existing files.
>
> NO_OVERWRITE indicates that the user wants the task to fail if the rename would overwrite any existing files.

Each invocation description requires two fields: SOURCE and DESTINATION. SOURCE is incarnated by the NJS into the name of the file to be moved to DESTINATION.

9.1 Example

```
RENAME_FILE
 INVOCATION OVERWRITE [ /bin/mv -f <SOURCE> <DESTINATION>]
 INVOCATION NO_OVERWRITE [
     if [ ! -f <DESTINATION> ]
      then
       /bin/mv  <SOURCE> <DESTINATION>
      else
       # Fail since a destination file exists
       /bin/printf "Rename failed. <DESTINATION> exists.\n" 1>&2
       the_last=1
       exit $the_last
     fi
  ]
END
```

10 SYMBOLIC_LINK section

The SYMBOLIC_LINK section tells the NJS how to make symbolic links between files in a Storage Server. It is the incarnation of the SymbolicLink abstract task.

SYMBOLIC_LINK is an Action definition section. See the introductory section (2.4) on Action Descriptions for the syntax of these, including the INVOCATION keyword.

SYMBOLIC_LINK requires invocation descriptions for two options of the task, these are:

> OVERWRITE, indicates that the user is prepared to allow the link file to overwrite any existing files.
>
> NO_OVERWRITE, indicates that the user wants the task to fail if the link would overwrite any existing files.

Each invocation description requires two fields: TARGET and LINK. TARGET is incarnated by the NJS into the source file to be linked to LINK.

10.1 Example

```
SYMBOLIC_LINK

  INVOCATION OVERWRITE [

      if [ -f <LINK> ]

      then

        RM_CMD <LINK>

      fi

      LN_CMD -s <TARGET> <LINK>

  ]

  INVOCATION NO_OVERWRITE [LN_CMD –s <TARGET> <LINK>]

END
```

11 DELETE_FILE section

The DELETE_FILE section tells the NJS how to delete files on the storage servers. It is the incarnation of the DeleteFile abstract task.

This is an Action Description. See the introductory section (2.4) on Action Descriptions for the syntax of these, including the INVOCATION keyword.

The invocation requires one field: TARGET. TARGET is incarnated by the NJS into the name of the file to be deleted.

11.1 Example

```
DELETE_FILE

  INVOCATION [ /bin/rm <TARGET> ]

END
```

12 CHANGE_PERMISSIONS section

The CHANGE_PERMISSIONS section tells the NJS how to change the owners permissions of files on the storage servers.

This is an Action Decsription. See the introductory section (2.4) on Action Descriptions for the syntax of these, including the INVOCATION keyword.

The invocation requires two fields: FILE and ARGUMENTS.

> FILE is incarnated by the NJS into the name of the file whose permissions are to be changed.
>
> ARGUMENTS is incarnated into the final permissions for the file in the form: u=rwx (some, all or none of r, w and x will be present).

12.1 Example

```
CHANGE_PERMISSIONS

  INVOCATION [ chmod <ARGUMENTS> <FILE> ]

END
```

13 FORTRAN section

The FORTRAN section tells the NJS how to incarnate instances of FortranTask which compiles sources files written in Fortran.

The FORTRAN section is an *Action description*. See the introductory section (2.4) on Action Descriptions for the syntax of these, including the INVOCATION keyword.

The FORTRAN section recognises a special Software Resource “PREPROCESS” for compilations that require a pass of the pre-processor[4]. Example:

```
INVOCATION PREPROCESS [ <STANDARD> —Cpp ]
```

File name extensions

The NJS needs to know the file name extensions used by the Fortran compiler for the following file types:

- source files
- object files
- source files to pass through the pre-processor
- listing files
- source files after passing through the pre-processor (used when a pre-processing only run is requested)

The NJS is told of the extensions using in a NAMING section:

```
NAMING
    SOURCE_FILE extension
    OBJECT_FILE extension
    PREPROCESS_FILE extension
    LISTING_FILE extension
    PREPROCESSED_FILE extension
END
```

For example[5]:

```
NAMING
    SOURCE_FILE .f90
    OBJECT_FILE .o
    PREPROCESS_FILE .F90
    LISTING_FILE .lis
    PREPROCESSED_FILE .f90
END
```

13.1 Invocation definition

The FORTRAN invocation definition must contains the following keywords:

<SOURCES>: Will be replaced by the names of the source files to be compiled.

---

[4] If this is used, then the file name extension for pre-process files should be the same as normal Fortran source files (usually "“f90")

<INCLUDES>: Will be replaced by the names of the directories to be searched for include files.

For example, the invocation string:

```
-I*<INCLUDES>
```

when given a list of directory names (e.g dir1 dir2 dir3) will incarnate as:

```
-Idir1 –Idir2 –Idir3
```

<VERSION>: Will be replaced by the incarnation of the user's choice for output of the compiler version information. This requires a Field Definition (see below)

<VERBOSE>: Will be replaced by the incarnation of the user's choice for verbose output by the compile. This requires a Field Definition (see below)

<LISTINGLEVEL>: Will be replaced by the incarnation of the user's choice of the level of listing produced by the compiler. This requires a Field Definition (see below)

<OPTIMISATIONLEVEL>: Will be replaced by the incarnation of the user's choice for the amount of optimisation to be done by the compiler. This requires a Field Definition (see below)

<DOUBLELENGTH>: Will be replaced by the incarnation of the user's requested default double length. This requires a Field Definition (see below)

<REALLENGTH>: Will be replaced by the incarnation of the user's requested default real length. This requires a Field Definition (see below)

<INTEGERLENGTH>: Will be replaced by the incarnation of the user's requested default integer length. This requires a Field Definition (see below)

<SOURCEFORM>: Will be replaced by the flag to tell the compiler the form of the Fortran source code. This requires a Field Definition (see below)

<ARGCHECKING>: Will be replaced by the flag to request argument checking code to be inserted by the compiler (if requested by the user). This requires a Field Definition (see below)

<BOUNDSCHECKING>: Will be replaced by the flag to request bounds checking code to be inserted by the compiler (if requested by the user). This requires a Field Definition (see below)

<PREPROCESSONLY>: Will be replaced by the flag to request only a pre-processing run compiler (if requested by the user). This requires a Field Definition (see below)

<DEFINENAME>: Will be replaced by the names and values of the (pre-processing) tokens that the user wants to set.

For example, the invocation string:

```
-D *<DEFINENAME>
```

will incarnate the name/value pairs (steve,waugh) (justin,langer) as:

```
-D steve=waugh –D justin=langer
```

<UNDEFINENAME>: Will be replaced by the names of the (pre-processing) tokens that the user wants to unset.

<DEBUG>: Will be replaced by the flag to turn debugging code generation on (if requested by the user). This requires a Field Definition (see below)

<PROFILE>: Will be replaced by the flag to turn profiling code generation on (if requested by the user). This requires a Field Definition (see below)

13.2 Field definitions

The format of the following sections is: field_name (option1, option2 ….) followed by an example:

VERSION (ON, OFF)

```
VERSION
    ON -version
    OFF _
END
```

VERBOSE (ON, OFF)

```
VERBOSE
    ON -v
    OFF _
END
```

LISTINGLEVEL (NONE, ON, FULL)

```
LISTINGLEVEL
    NONE _
    ON –L1
    FULL –L9
END
```

OPTIMISATIONLEVEL (NONE, DEFAULT, AGGRESSIVE)

```
OPTIMISATIONLEVEL
    NONE –O0
    DEFAULT _
    AGGRESSIVE –Wv,-Of
END
```

DOUBLELENGTH (DON’T_CARE, EIGHT_BYTES, SIXTEEN_BYTES)

```
DOUBLELENGTH
    DON’T_CARE _
    EIGHT_BYTES –AQ
    SIXTEEN_BYTES -Aq
END
```

REALLENGTH (DON’T_CARE, FOUR_BYTES, EIGHT_BYTES)

```
REALLENGTH
    DON’T_CARE _
    FOUR_BYTES –AD
    EIGHT_BYTES -Ad
END
```

INTEGERLENGTH (DON’T_CARE, FOUR_BYTES, EIGHT_BYTES)

```
INTEGERLENGTH
```

```
    DON’T_CARE _
    FOUR_BYTES _
    EIGHT_BYTES —CcI4I8
END
```

SOURCEFORM (FIXED, FREE)

```
SOURCEFORM
    FIXED -Ffixed
    FREE -Ffree
END
```

ARGCHECKING (ON,OFF)

```
ARGCHECKING
    ON -Da
    OFF _
END
```

BOUNDSCHECKING (ON,OFF)

```
BOUNDSCHECKING
    ON -Ds
    OFF _
END
```

PREPROCESSONLY (ON,OFF)

```
PREPROCESSONLY
    ON —cpp_only
    OFF _
END
```

DEBUG (ON,OFF)

```
DEBUG
    ON —g 3
    OFF _
END
```

PROFILE (ON,OFF)

```
PROFILE
    ON —p 3
    OFF _
END
```

13.3 Example

14 LINK section

The LINK section tells the NJS how to incarnate instances of LinkTask which links object files to an executable.

The LINK section is an *Action description*. See the introductory section (2.4) on Action Descriptions for the syntax of these, including the INVOCATION keyword.

File name extensions

The NJS needs to know the file name extensions used by the linker for the following file types:

- object files
- libraries
- linker map files

The NJS is told of the extensions using in a NAMING section:

```
NAMING

    OBJECT_FILE extension

    LIBRARY extension

    MAP_FILE extension

END
```

For example[6]:

```
NAMING

    OBJECT_FILE .o

    LIBRARY .a

    MAP_FILE .map

END
```

14.1 Invocation definition

The LINK invocation definition must contains the following keywords:

<EXECUTABLE>: Will be replaced by the name of the executable file created bythe link.

<OBJECTS>: Will be replaced by the names of the object files to be linked.

<**LIBRARIES**>: Will be replaced by the libraries to be searched during the link, for example:

```
-l<LIBRARIES>
```

when given a list of libraries (dir1 dir2 dir3) will incarnate as:

```
-ldir1 –ldir2 –ldir3
```

<VERSION>: Will be replaced by the incarnation of the user's choice for output of the compiler version information. This requires a Field Definition (see below)

<VERBOSE>: Will be replaced by the incarnation of the user's choice for verbose output by the compile. This requires a Field Definition (see below)

<MAPLEVEL>: Will be replaced by the incarnation of the user's choice of the level of map produced by the linker. This requires a Field Definition (see below)

<DEBUG>: Will be replaced by the flag to request that the executable include debugging code at the level requested by the user. This requires a Field Definition (see below)

<PROFILE>: Will be replaced by the flag to request that the executable include profiling code at the level requested by the uesr This requires a Field Definition (see below)

14.2 Field definitions

The format of the following sections is: field_name (option1, option2 ….) followed by an example:

VERSION (ON, OFF)

```
VERSION

    ON -version

    OFF _

END
```

VERBOSE (ON, OFF)

```
VERBOSE

    ON -v

    OFF _

END
```

MAPLEVEL (NONE, ON, FULL)

```
MAPLEVEL

    NONE _

    ON —M1

    FULL —M9

END
```

DEBUG (ON,OFF)

```
DEBUG

    ON —g 3

    OFF _

END
```

PROFILE (ON,OFF)

```
PROFILE

    ON —profile —lproflib.a

    OFF _

END
```

14.3 Example

15 COPY_PF section

The COPY_PF section tells the NJS how to copy the files in Portfolios between directories. It is used by during the incarnation of a number of tasks.

There is usually no need to supply incarnation rules for COPY_PF as the NJS contains its own incarnation.

COPY_PF is an Action definition section. See the introductory section (2.4) on Action Descriptions for the syntax of these, including the INVOCATION keyword.

Each invocation description can contain four fields:

SDIR: the directory containing the files to be copied

SPF: list of file names to be copied

DDIR: destination directory

The invocation should copy the files in SDIR only, the NJS will copy the Portfolio descriptions.

15.1 Example

```
COPY_PF

  INVOCATION [ /bin/cp -r <SPF> <DDIR> ]

END
```

16 MAKE_FIFO section

The MAKE_FIFO section tells the NJS how to create FIFO files. It is the incarnation of the MakeFifo abstract task.

MAKE_FIFO is an Action definition section. See the introductory section (2.4) on Action Descriptions for the syntax of these, including the INVOCATION keyword.

MAKE_FIFO requires invocation definitions for two options of the task, these are: OVERWRITE and NO_OVERWRITE.

> OVERWRITE indicates that the user is prepared to allow the new FIFOs to overwrite any existing files.
>
> NO_OVERWRITE indicates that the user wants the task to fail if the FIFO would overwrite any existing files.

Each invocation description requires one field: TARGET , the name of the FIFO.

This section is optional. If there is no MAKE_FIFO section, the NJS will incarnate MakeFifo tasks as in the example below.

16.1 Example

```
MAKE_FIFO

  INVOCATION NO_OVERWRITE [ /bin/mkfifo –m600 <TARGET> ]

  INVOCATION OVERWRITE [

     RM_CMD –f <TARGET>

     /bin/mkfifo –m600 <TARGET>

END
```