Xcrypt Manual

E-Science Group, Nakashima Laboratory, ACCMS, Kyoto University

January 14, 2011

Contents

Part I

General

Chapter 1

Introduction

1.1 Overview

In using a high-performance computer, we usually commit job processing to a job scheduler. At this time, we often go through the following procedures:

- to create a script in its writing style depending on the job scheduler,

- to pass the script to the job scheduler, and

- to extract data from its result, create another script from the data, and pass it to the job scheduler.

However, such procedures require manual intervention cost. It therefore seems better to remove manual intervention in mid-processing by using an appropriate script language. Xcrypt is a script language for job parallelization. We can deal with jobs as objects (called *job objects*) in Xcrypt and manipulate the jobs as well as objects in an object-oriented language. Xcrypt provides some functions and modules for facilitating job generation, submission, synchronization, etc. Xcrypt makes it easy to write scripts to process job, and supports users to process jobs easily.

1.2 Environment

Xcrypt requires Perl ($\geq$ 5.8.5) and a superset of Bourne shell.
Xcrypt also requires the following outer modules:

- Sherzod Ruzmetov's Config-Simple,

- Marc Lehmann's Coro (where conftest.c is not contained), EV,

- Gurusamy Sarathy's Data-Dumper,

- Graham Barr's Error,

- Joshua Nathaniel Pritikin's Event,

- Salvador Fandiño's Net-OpenSSH,

- Daniel Muey's Recursive,

and wants Marc Lehmann's AnyEvent, common::sense, and Guard (warns if none). These modules are bundled with Xcrypt.

Chapter 2

Script
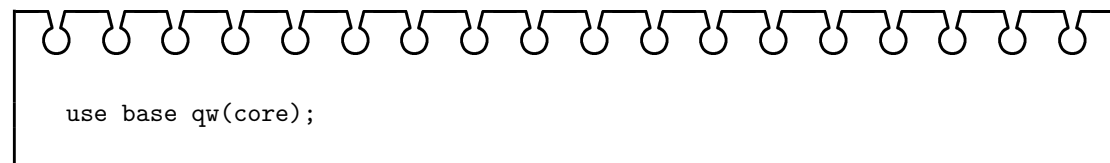
Xcrypt is a script language, and an extension of Perl. Xcrypt provides some functions and modules (not in Perl) which support how to deal with *jobs*.

An Xcrypt script consists of descriptions of

1. module,

2. template, and

3. procedure.
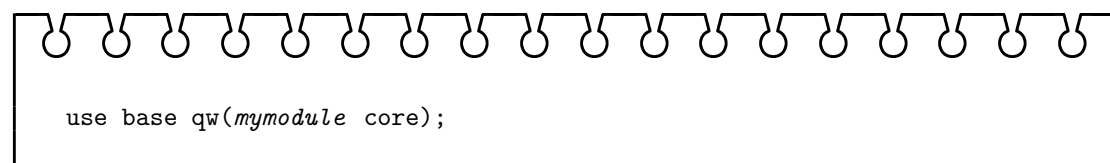
2.1   Module

Modules for job objects are used as follows,

```
use base qw(core);
```

When you use multiple modules, it is enough to write

```
use base qw(mymodule core);
```

Every module should be used in order. The details of the modules are described in Chapter 4.

Commonly-used modules can be loaded as follows,

```
use mymodule;
```

similarly to how to use modules in Perl.

2.2   Template

Xcrypt's templates are implemented as Perl's hashes. For example,

```
%mytemplate = (
  'id@' => sub { "myjob$VALUE[0]"; },
  'exe0@' => sub { "./myexe $VALUE[0]"; },
  'RANGE0' => [0,1]
);
```

Keys in templates are described in Chapter 5 in detail.

2.3   Job Object
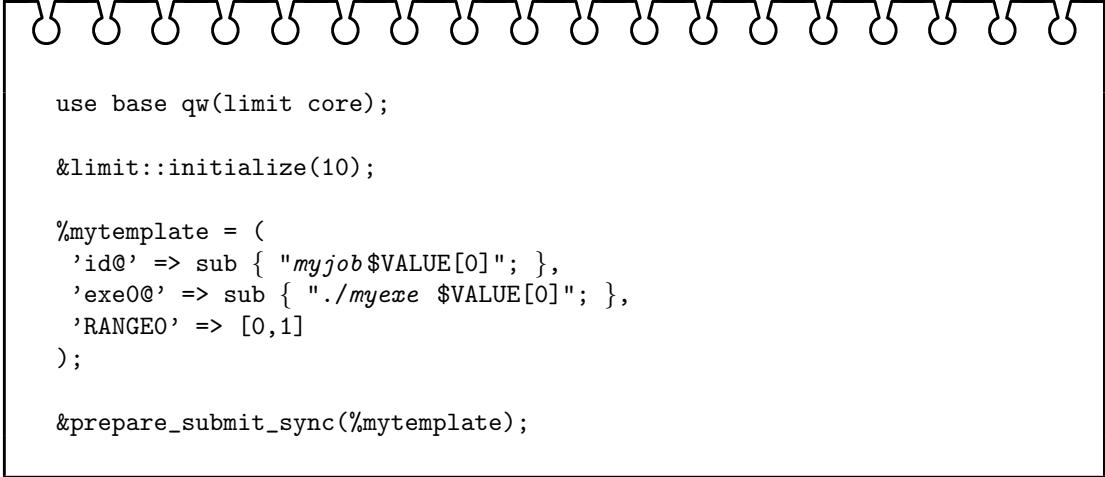
Xcrypt's job object are implemented as Perl's objects (blessed hash references). In Xcrypt, job objects should be typically created from templates by a built-in function &prepare (Chapter 6 in detail).

2.4   Procedure

Procedures of job processing are described in Xcrypt (and Perl) instead of manually carried out. Xcrypt's functions are described in Chapter 6.

2.5   Example

An example script is as follows,

```
use base qw(limit core);

&limit::initialize(10);

%mytemplate = (
 'id@' => sub { "myjob$VALUE[0]"; },
 'exe0@' => sub { "./myexe $VALUE[0]"; },
 'RANGE0' => [0,1]
);

&prepare_submit_sync(%mytemplate);
```

Chapter 3

Flow

In this chapter, we introduce how jobs are processed.

3.1 State

Any job has one of the following states:

initialized:	the job is initialized or aborted,
prepared:	the same as initialized (for backward compatibility)
submitted:	the job is submitted,
queued:	the job is queued,
running:	the job is running,
done:	the job is done,
finished:	the job is finished,
aborted:	the job is aborted.

3.2 Installation

Edit `xcrypt/source-me.sh` in order to set some environment variables, and

```
$ source source-me.sh
```

Then, continue the following installation procedure:

```
$ cd $XCRYPT/cpan; ./do-install.sh
```

3.3 Execution

Edit `$HOME/.xcryptrc` as follows,

```
  [default]
  sched = t2k_kyoto
  queue = eh
  [special]
  sched = t2k_kyoto
  queue = ocuppied
```

along your environment[1].

Next, move to the working directory (e.g., `$HOME/wd`)

```
$ cd $HOME/wd
```

and write an Xcrypt script (e.g., `sample.xcr`). See Section 2.5 in order to know how to write.

Finally, execute Xcrypt with the script:

```
$ $XCRYPT/bin/xcrypt sample.xcr
```

If you use Emacs, then the following description in `.emacs.el` helps you.

```
  (add-to-list 'auto-mode-alist '("\\.xcr" . perl-mode))
```

3.4   Interactive Usage

```
$ $XCRYPT/bin/xcrypt myscript.xcr
```

makes Xcrypt to interpret *myscript*.`xcr`.

```
$ $XCRYPT/bin/xcryptstat
```

shows states of jobs. In detail, use the `--help` option.

```
$ $XCRYPT/bin/xcryptdel myjob [myjob...]
```

makes states of unfinished jobs aborted. For deleting all the jobs, use the `--all` option. For making (not necessarily unfinished) jobs aborted, use the `--uninitialize` option. For making jobs finished, use the `--finish` option. For forgetting states of all jobs, use the `--clean` option.

---

[1]`SGE`, `t2k_tsukuba`, `t2k_tokyo`, `t2k_kyoto`, and `sh` are available. In the case of `sh`, jobs are dealt with as processes in OS. The default is `sh`

3.5   Product

Xcrypt creates the following in the working directory during and after its execution.

*myjob*_$XCRJOBSCHED.sh

is a job script passed to a job scheduler or a Bourne shell script executed, regarding OS as a job scheduler, respectively.

*myjob*_stdout

is a file storing the job's standard output. When `stdofile` is defined, the file is renamed as its value.

*myjob*_stderr

is a file storing the job's standard error. When `stdefile` is defined, the file is renamed as its value.

inv_watch

is a directory containing log and other files for retry. When `--inventory_path` is defined, the directory is renamed as its value.

Part II

Details

Chapter 4

Module

In this chapter, we introduce some modules available in Xcrypt scripts.

4.1 core

This module is the Xcrypt core module, and required to be read in order to use anything particular to Xcrypt.

4.2 sandbox

A directory of the name

$$\texttt{join('\_', (\$myjob->\{id\}, @VALUE))}$$

is created for each job (called a *job working directory*). Job-processing is done in the job working directory.
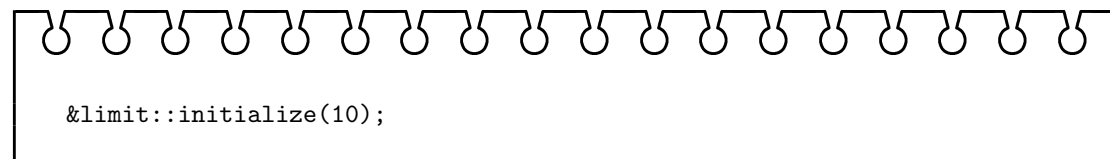
The following can be defined in templates.

`linkedfile`i: a soft link of the file (whose name is its value) is created in the job working directory.

`copiedfile`i: the indicated file is copied to the job working directory.

4.3 limit

This module limits the number of jobs submitted simultaneously. In order to limit the number of jobs to 10, for example, it is enough to write as follows,

```
&limit::initialize(10);
```

4.4 `bulk_simple`

This module gathers jobs and returns one job of the same denotation. For bulked jobs, the function &bulk ignores any member except exe*i*, exe*i*_j, initially, before, after, and finally. Typically, you can write as follows,

```
my $template = { 'id' => 'bulked_job_', 'JS_queue' => 'myqueue' };
my @jobs0 = &prepare(%template0);
my @jobs1 = &prepare(%template1);
my @bulk = &bulk_simple::bulk($template, @jobs0, @jobs1);
&submit(@bulk);
```

4.5 `successor`

This module indicates job objects which can be defined declaratively. For example, in order to define job objects of the name %x, %y, write:

```
...
'successor' => ['x', 'y'],
...
```

using the key successor in the template.

4.6 `convergence`

This module provides a function for a Plan-Do-Check-Action (PDCA) cycle, to deal with convergence of difference of job's results. The keys initialvalue, isConvergent, inputfile, sweepname, outputfile, and extractrules can be used in templates.

4.7 `n_section_method`

This module provides n-section method, a root-finding algorithm. The only difference from bisection method[1] is the number of sections.

The values partition and epsilon denote a partition number and an error, respectively. An interval is expressed by x_left and x_right. The values y_left and y_right are values on x_left and x_right. Typically, we can call the function n_section_method with these keys, e.g.,

---

[1]http://en.wikipedia.org/wiki/Bisection_method/

```
&n_section_method::n_section_method(%job,
    'partition' => 12, 'epsilon' => 0.01,
    'x_left'  => -1,  'x_right' => 10,
    'y_left'  => 0.5, 'y_right' => -5
);
```

4.8 dry

This module provides job-processing in dry mode (skipping any command execution). Description in a template

```
...
'dry' => 1,
...
```

makes any job (derived from this hash) to be processed in dry mode.

4.9 invalidate

This module invalidates jobs of which running time is more than allotted_time (can be defined in templates).

Chapter 5

Template

In this chapter, we introduce keys and values available in templates by default.

5.1 RANGEi

key@ denotes the one whose postfix is the character @ (e.g., exe0@).
   Any word of ASCII printable characters except

$$@_{\sqcup}"\$\%\&'/:;<=>?[\backslash]'\{|\}$$

is available for RANGEi's values.
   @ means
   values are array references, function references, (or scalar although not recommended).

5.2 id

Its value is a word. The value is used for creating job objects and identifying the job objects as their prefixes. Any word of ASCII printable characters except

$$@_{\sqcup}"\$\%\&'/:;<=>?[\backslash]'\{|\}$$

is available.

5.3 exe$i$

Its value denotes a command. The command is executed as follows,

```
$  myexe0 myarg0_0 ...
$  myexe1 myarg1_0 ...
⋮
```

with argi_j explained below.

5.4 argi_j

Its values are arguments of a command.

5.5 `stdofile`

The standard output is stored in the indicated file. The default is `stdout`.

5.6 `stdefile`

The standard error is stored in the indicated file. The default is `stderr`.

5.7 JS_*key*

Chapter 6

Function

In this chapter, we introduce built-in functions.

6.1 `prepare`

This function takes a job definition hash and parameters of references[1], and returns an array of job objects.

Format

```
prepare(%template);
```

Example

```
@jobs = prepare('id@' => sub { "myjob $VALUE[0]"; },
                'exe0@' => sub { "./myexe0 $VALUE[0]"; },
                'exe1@' => sub { "./myexe1 $VALUE[0]"; },
                'RANGE0' => [0,1]);
```

Declarative description is also available as follows,

---

[1]In this manual, references do not denote type globs.

```
%mytemplate = (
    'id@' => sub { "myjob$VALUE[0]"; },
    'exe0@' => sub { "./myexe0 $VALUE[0]"; },
    'exe1@' => sub { "./myexe1 $VALUE[0]"; },
    'RANGE0' => [0,1]
);

@jobs = prepare(%mytemplate);
```

Advanced

It is possible to generate job objects by using multiple parameters. For example,

```
%mytemplate = (
    'id@' => sub { "myjob$VALUE[0]_$VALUE[1]"; },
    'exe0@' => sub { "./myexe $VALUE[0] $VALUE[1]"; },
    'RANGE0' => [0,1],
    'RANGE1' => [2,4]
);
@jobs = prepare(%mytemplate);
```

creates 4 job objects. This is the same as

```
%mytemplate = (
    'id@' => sub { "myjob$VALUE[0]_$VALUE[1]"; },
    'exe0@' => sub { "./myexe $VALUE[0] $VALUE[1]"; },
    'RANGES' => [[0,1],[2,4]]
);
@jobs = prepare(%mytemplate);
```

6.2 submit

This function takes an array of job objects and passes the jobs (corresponding to the job objects)
to a job scheduler. Its return value is also the array of job objects.

Format

```
submit(@myjobs);
```

Example

Typically, this function takes a return value of `prepare`.

```
@jobs = prepare(%mytemplate);
submit(@jobs);
```

It is possible to define job references without using `prepare` (although not recommended).

6.3 sync

This function takes an array of job objects and synchronizes the job objects. Its return value is the array of job objects.

Format

```
sync(@myjobs);
```

Example

Typically, this function takes a return value of `prepare` (same as `submit`).

```
@jobs = prepare(%mytemplate);
submit(@jobs);
sync(@jobs);
```

6.4 `xcr_exist`

This function returns 1 if `$file` exists (0 unless) at `$env{location}`.

Format

```
xcr_exist(\%env, $file);
```

6.5 `xcr_qx`

This function returns `$command`'s standard output at `$env{location}` as an array.

Format

```
xcr_qx(\%env, $command);
```

6.6 `xcr_system`

This function returns `$command`'s return value at `$env{location}`.

Format

```
xcr_system(\%env, $command);
```

6.7 `xcr_mkdir`

This function makes a directory of the name `$dir` at `$env{location}`.

Format

```
xcr_mkdir(\%env, $dir);
```

6.8 xcr_copy

This function copies $file_or_dir0 to $file_or_dir1 at $env{location}.

Format

```
xcr_copy(\%env, $file_or_dir0, $file_or_dir1);
```

6.9 xcr_rename

This function rename $file0 to $file1 at $env{location}.

Format

```
xcr_rename(\%env, $file0, $file1);
```

6.10 xcr_symlink

This function links $file as $link in $dir at $env{location}.
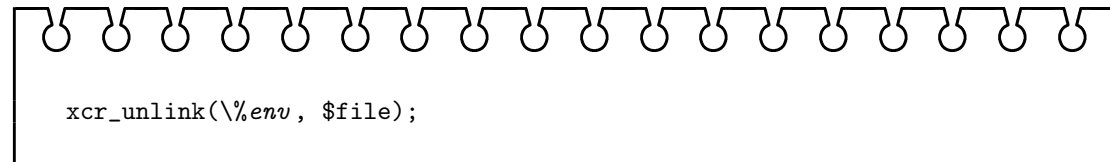
Format

```
xcr_symlink(\%env, $file, $dir, $link);
```

6.11 xcr_unlink

This function removes `$file` at `$env{location}`.

Format

```
xcr_unlink(\%env, $file);
```

6.12 get_from

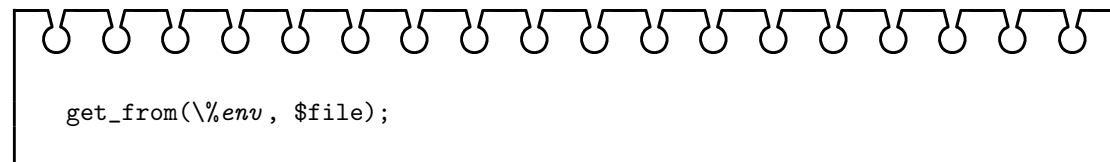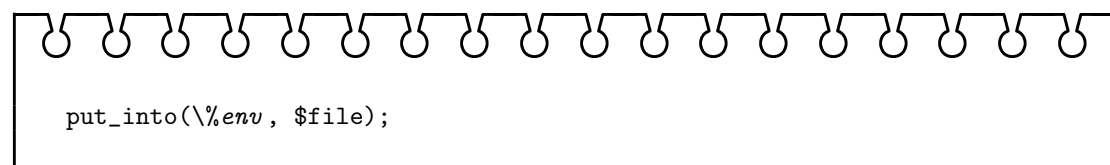This function gets `$file` from `$env{wd}` in `$env{location}`.

Format

```
get_from(\%env, $file);
```

6.13 put_into

This function puts `$file` into `$env{wd}` in `$env{location}`.

Format

```
put_into(\%env, $file);
```

6.14 add_host

This function takes a hash that denotes a host (containing its environment), and returns a reference that denotes it.

Format

```
add_host(\%env);
```

Example

```
$env = add_host({'host' => 'foo@bar', 'wd' => '/home/foo'});
%template = ('id' => 'myjob', 'exe0' => './myexe', 'env' => $env);
```

6.15 get_local_env

This function returns a reference that denotes the local host (containing its environment).

Format

```
get_local_env();
```

Example

```
$env = get_local_env();
%template = ('id' => 'myjob', 'exe0' => './myexe', 'env' => $env);
```

6.16 add_key

This function takes an array of words and makes it available as keys in job definition hashes.
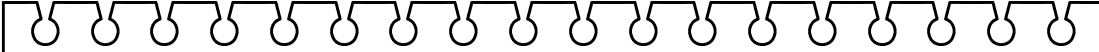
Format

```
add_key(@words);
```

6.17 add_prefix_of_key

This function takes an array of words and makes it available as prefixes of keys in job definition hashes.
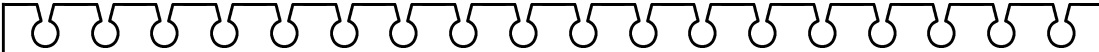
Format

```
add_prefix_of_key(@words);
```

6.18 repeat

This function takes an Xcrypt's script code (denoted as $mystring$) and an integer i, and evaluates it each i seconds.

Format

```
repeat(mystring, i);
```

6.19 set_expander

Format

```
set_expander($string);
```

6.20 get_expander
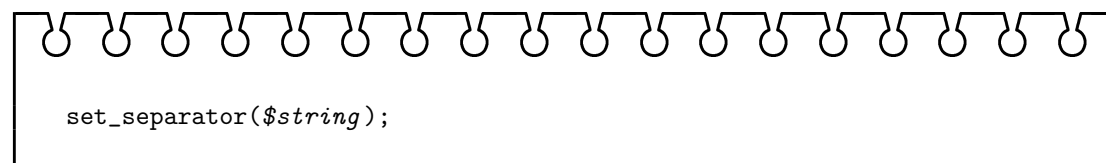
Format

```
get_expander();
```

6.21 set_separator
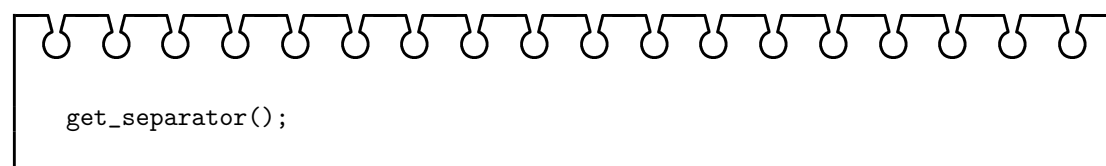
Format

```
set_separator($string);
```

6.22 get_separator
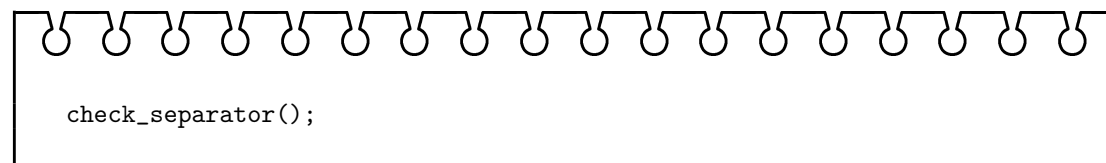
Format

```
get_separator();
```

6.23 check_separator
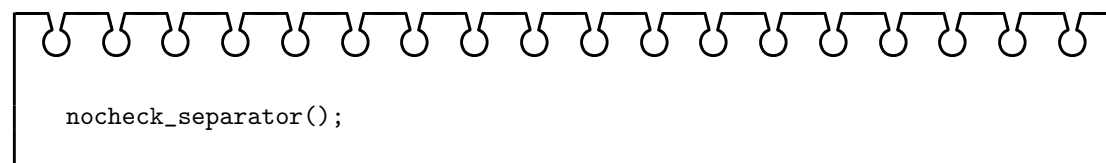
Format

```
check_separator();
```

6.24 nocheck_separator

Format

```
nocheck_separator();
```

6.25 prepare_submit

This function makes `prepare` and `submit` applied to job objects generated by `prepare`. The composition of `prepare` and `submit` is done at each job object.

6.26 prepare_submit_sync

This function is an abbreviation of prepare_submit and sync. Its format follows prepare.

Chapter 7

Methods of `core` Class

This chapter explains methods defined in `core` class. These methods can be used by end users. The methods that are defined in `core` class but should not be used directly by end users are listed in Appendix A.4.

7.1   `workdir_member_file`

to be written...

7.2   `abort`

to be written...

7.3   `cancel`

to be written...

7.4   `invalidate`

to be written...

Chapter 8

Option

8.1 --ini *entry*

uses the entry in `$HOME/.xcryptrc` as config.

8.2 --sched *job_scheduler*

takes a batch job scheduler.

8.3 --print_log

prints the log (in particular, status of job objects), in detail.

8.4 --compile_only

only makes a Perl script from the Xcrypt scritp, and does not exec the Perl script.

8.5 --shared

does not copy files at a shared system in remote-execution mode.

8.6 --xbs-type *type*

use Keiji Yamamoto's XBS[1] internally (*site* in config).

8.7 --xbs-auto

8.8 --abort_check_interval *num*

an interval of checking whether jobs are aborted or not (sec).

---

[1]`http://www.il.is.s.u-tokyo.ac.jp/~yamamoto/xbs/`

8.9   `--left_message_check_interval` *num*

an interval of checking what states jobs have (sec).

8.10   `--inventory_path` *path*

a path at that logs are located.

8.11   `--verbose` *num*

8.12   `--stack_size` *num*

8.13   `--host` *username@hostname*

a host to that jobs are submitted.

8.14   `--wd` *path*

a working directory in that Xcrypt is executed at a remote host.

8.15   `--xd` *path*

8.16   `--scratch`

executes Xcrypt without recovering the states of jobs in the previous execution.

Appendix A

How to Implement Job Class Extension Modules

Any job object generated by the Xcrypt's function `prepare` belongs to the class `core`, defined by `$XCRYPT/lib/core.pm`.Xcrypt users and developers can extend the class `core` by defining modules and consequently expand the function of Xcrypt. In this chapter, we introduce how to implement such extension modules.

A.1 How to Define and Use Extension Modules

In order to define an extension module of the name *mymodule*, it is enough for Xcrypt developers to put it into any directory designated by `$XCRYPT/lib/` (or `$PERL5LIB`).

Then Xcrypt users can use the extension module by simply indicating their name on the header of his/her script as follows:

```
use base (...  mymodule ... core);
```

A.2 Scripts of Extension Modules

A definition script for an extension module is typically described as follows,

```perl
package  mymodule;


use strict;
use ...;

&add_key('my_instance_member', ...);

my $my_class_member;

# special methods
sub new {
  my $class = shift;
  my $self = $class->NEXT::new(@_);
  ...
  return bless $self, $class;
}

sub before { ... }

sub start
{
  my $self = shift;
  ...
  $self->NEXT::start();
  ...
}

sub after { ... }

# general methods
sub another_method
{
  ...
}
```

In the following, we make an explanation for each component of the script.

1. Definition of the module name: is designated by `package`. The module name must coincide with the file name without its extension (`.pm`).

2. Use of Perl modules: is declared by using `use` as in typical Perl programs.

3. Addition of instance variables: is performed by the function `add_key`. The added instance variables are accessible as attributes of the job objects by writing, e.g.,

$job->{my_instance_member}

in Xcrypt scripts and modules. Also, by writing, e.g.,

$$\texttt{\%template = \{ ..., my\_instance\_menber=>}value\texttt{, ...\}}$$

users can set values to them.

4. Definition of class variables: is done in the usual way in object-oriented programming, i.e., class variables are defined as global variables in packages. The variables can be accessed, e.g.,

$$\texttt{\$}mymodule\texttt{::my\_class\_member}$$

5. Definition of methods: is defined in the usual way, i.e., methods added and extended in modules are defined as top-level functions in packages. Note that some methods with particular names have special meanings as explained in the next section.

A.3   Special Methods

Xcrypt gives special meanings to the following class methods.

A.3.1   new

The method `new` is a class method, the so-called *constructor*. The method `new` in the most specialized class (the left-most module declared on the script header) is called.

The method `new` takes the following arguments:

1. the package name (= `user`) to which an Xcrypt script belongs,

2. a reference to a job object[1].

Note that `new` is applied to each of multiple objects generated by `prepare`.

In the body of a method, the method `new` in the parent class is called as

$$\texttt{\$class->NEXT::new(\$self,\$obj)}$$

where `$class` and `$obj` are the class name and reference to the object, the arguments of `new`, respectively.

Typically, each `new` calls `new` in his parent class with the same two arguments, processes its return value (an object), and returns `bless` *reference to the object*, *the class name* as return values.

In the module `core`, `new` is defined. The `new` creates a job directory, soft links, and copies of files (explained in Section 3.5). Note that this required procedure is skipped unless `new`s in children classes call the `new` in the `core`.

A.3.2   initially

to be written...

---

[1]The object members has values in the template passed to the function `prepare`.

A.3.3 `before`

In Xcrypt, application of the function `submit` (cf. Section 6.2) makes a job object's state `prepared`. The methods `before`s are applied to a job object of the state `prepared` (cf. Section 3.1). Its argument is a reference to the job object. The order of calling `before`s is in such a way from children to parents classes. Return values of the methods are abandoned.

A.3.4 `start`

The methods `start`s are applied to a job object after `before`s to the job objects are applied. Its argument is a reference of the job object. The method `start` in the most specialized class (the left-most module declared on the script header) is called.

In the body of a method, the method `new` in the parent class is called as

$$\texttt{\$obj->NEXT::start()}$$

where `$obj` is the reference to the object.

In the module `core`, `start` is defined. The `start` creates a job script and submits the job to a job scheduler. Note that this required procedure is skipped unless `start`s in children classes call the `start` in the `core`.

A.3.5 `before_in_job`

The definition of this method is serialized (using the Data::Dumper module), sent to a job execution node, and interpreted and invoked by the Perl process executed in the node just *before* executing `exe`i. The job object is also serialized and passed to the method as its argument. A global variable defined in (local) Xcrypt is referred to only if it is specified by the `transfer_variable` job template member. Updates of the transferred variable are not reflected to the Xcrypt process.

A.3.6 `after_in_job`

The definition of this method is serialized (using the Data::Dumper module), sent to a job execution node, and interpreted and invoked by the Perl process executed in the node just *after* executing `exe`i. The job object is also serialized and passed to the method as its argument. A global variable defined in (local) Xcrypt is referred to only if it is specified by the `transfer_variable` job template member. Updates of the transferred variable are not reflected to the Xcrypt process.

A.3.7 `after`

In Xcrypt, a completion notice of a job submitted by the method `core::start` makes the job object's state `done`. The methods `after`s are applied to a job object with the state `done` (cf. Section 3.1). Its argument is a reference to the job object. The order of calling `after`s is in such a way from parents to children classes. Return values of the methods are abandoned.

A.3.8 `finally`

to be written...

A.4 Ordinary methods

A developer of Xcrypt modules can add ordinary methods and extended preexisting ordinary methods defined in `core.pm` in the manner of the object oriented Perl programming. This section lists the ordinary methods defined in `core.pm`. Note that the methods that can be called by end users are already listed in Chapter 7, which can be extended by Xcrypt module developers, too.

A.4.1 apply_push_valid_arg

A.4.2 make_jobscript

called by `qsub_make`.

A.4.3 make_jobscript_header

called by `make_jobscript`

A.4.4 make_jobscript_body

called by `make_jobscript`

A.4.5 make_in_jobscript

called by `make_before_in_jobscript` and `make_after_in_jobscript`

A.4.6 make_before_in_jobscript

called by `qsub_make`.

A.4.7 make_after_in_jobscript

called by `qsub_make`.

A.4.8 update_script_file

called by `update_jobscript_file`, `update_before_in_job_file`, and `update_after_in_job_file`.

A.4.9 update_jobscript_file

calls `update_script_file`

A.4.10 update_before_in_job_file

calls `update_script_file`

A.4.11 update_after_in_job_file

calls `update_script_file`

A.4.12 update_all_script_files

It calls `update_jobscript_file`, `update_before_in_job_file`, and `update_after_in_job_file`.

A.4.13 make_qsub_options

called by `qsub_make`.

A.4.14 qsub_make

called by `qsub`.

A.4.15 qsub

A.4.16 qdel

A.4.17 qdel_if_queued_or_running

conditionally calls `qdel`.