

# Xcrypt Manual

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

September 13, 2011

# Contents

<b>I</b>	<b>General</b>	<b>4</b>
<b>1</b>	<b>Introduction</b>	<b>5</b>
1.1	Overview . . . . .	5
1.2	Environment . . . . .	5
<b>2</b>	<b>Script</b>	<b>7</b>
2.1	Module . . . . .	7
2.2	Template . . . . .	8
2.3	Job Object . . . . .	8
2.4	Environment Object . . . . .	8
2.5	Procedure . . . . .	8
2.6	Example . . . . .	8
<b>3</b>	<b>Flow</b>	<b>10</b>
3.1	State . . . . .	10
3.2	Installation . . . . .	10
3.3	Execution . . . . .	10
3.4	Interactive Usage . . . . .	11
3.5	Product . . . . .	11
<b>II</b>	<b>Details</b>	<b>13</b>
<b>4</b>	<b>Module</b>	<b>14</b>
4.1	core . . . . .	14
4.2	sandbox . . . . .	14
4.3	limit . . . . .	14
4.4	bulk . . . . .	14
4.5	successor . . . . .	15
4.6	convergence . . . . .	15
4.7	n_section_method . . . . .	15
4.8	dry . . . . .	15
4.9	invalidate . . . . .	16
<b>5</b>	<b>Template</b>	<b>17</b>
5.1	RANGE <i>i</i> . . . . .	17
5.2	id . . . . .	17
5.3	exe . . . . .	17

5.4	header	17
5.5	exe <i>i</i>	17
5.6	arg <i>i-j</i>	18
5.7	stdofile	18
5.8	stdefile	18
5.9	JS_key	18
5.10	env	18
<b>6</b>	<b>Function</b>	<b>19</b>
6.1	prepare	19
6.2	submit	21
6.3	sync	21
6.4	add_host	22
6.5	get_local_env	22
6.6	xcr_exist	23
6.7	xcr_qx	23
6.8	xcr_system	23
6.9	xcr_mkdir	24
6.10	xcr_copy	24
6.11	xcr_rename	24
6.12	xcr_symlink	25
6.13	xcr_unlink	25
6.14	get_from	25
6.15	put_into	25
6.16	add_key	26
6.17	add_prefix_of_key	26
6.18	repeat	26
6.19	set_expander	27
6.20	get_expander	27
6.21	set_separator	27
6.22	get_separator	27
6.23	check_separator	28
6.24	nocheck_separator	28
6.25	prepare_submit	28
6.26	prepare_submit_sync	28
<b>7</b>	<b>Methods of core Class</b>	<b>29</b>
7.1	workdir_member_file	29
7.2	abort	29
7.3	cancel	29
7.4	invalidate	29
<b>8</b>	<b>Option</b>	<b>30</b>
8.1	--config <i>xcryptrc</i>	30
8.2	--sched <i>job_scheduler</i>	30
8.3	--compile_only	30
8.4	--shared	30
8.5	--xbs-type <i>type</i>	30
8.6	--xbs-auto	30
8.7	--abort_check_interval <i>num</i>	30

8.8	--left_message_check_interval <i>num</i> . . . . .	31
8.9	--inventory_path <i>path</i> . . . . .	31
8.10	--verbose <i>num</i> . . . . .	31
8.11	--stack_size <i>num</i> . . . . .	31
8.12	--host <i>username@hostname</i> . . . . .	31
8.13	--wd <i>path</i> . . . . .	31
8.14	--xd <i>path</i> . . . . .	31
8.15	--scratch . . . . .	31
<b>A How to Implement Job Class Extension Modules</b>		<b>32</b>
A.1	How to Define and Use Extension Modules . . . . .	32
A.2	Scripts of Extension Modules . . . . .	32
A.3	Special Methods . . . . .	34
A.3.1	<i>new</i> . . . . .	34
A.3.2	<i>initially</i> . . . . .	34
A.3.3	<i>before</i> . . . . .	35
A.3.4	<i>start</i> . . . . .	35
A.3.5	<i>after</i> . . . . .	35
A.3.6	<i>finally</i> . . . . .	35
A.4	Ordinary methods . . . . .	35
A.4.1	<i>apply_push_valid_arg</i> . . . . .	35
A.4.2	<i>make_jobscript</i> . . . . .	35
A.4.3	<i>make_jobscript_header</i> . . . . .	35
A.4.4	<i>make_jobscript_body</i> . . . . .	36
A.4.5	<i>make_in_jobscript</i> . . . . .	36
A.4.6	<i>make_before_in_jobscript</i> . . . . .	36
A.4.7	<i>make_after_in_jobscript</i> . . . . .	36
A.4.8	<i>update_script_file</i> . . . . .	36
A.4.9	<i>update_jobscript_file</i> . . . . .	36
A.4.10	<i>update_before_in_job_file</i> . . . . .	36
A.4.11	<i>update_after_in_job_file</i> . . . . .	36
A.4.12	<i>update_all_script_files</i> . . . . .	36
A.4.13	<i>make_qsub_options</i> . . . . .	36
A.4.14	<i>qsub_make</i> . . . . .	36
A.4.15	<i>qsub</i> . . . . .	36
A.4.16	<i>qdel</i> . . . . .	36
A.4.17	<i>qdel_if_queued_or_running</i> . . . . .	36

**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 confstest.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,

- H.Merijn Brand and Jochen Wiedmann's Text-CSV\_XS,

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

# Chapter 2

## Script

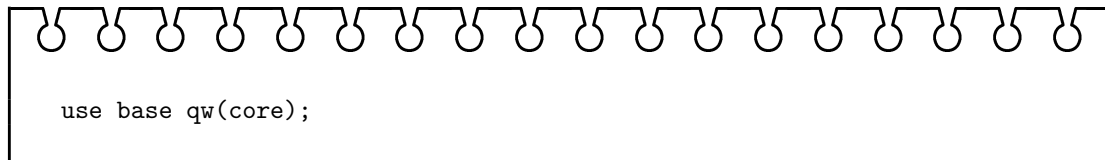
Xcrypt is a script language, and an extension of Perl. Xcrypt provides some functions and modules (not in Perl) which support how to deal with *jobs*.

An Xcrypt script consists of descriptions of

1. module,
2. template, and
3. procedure.

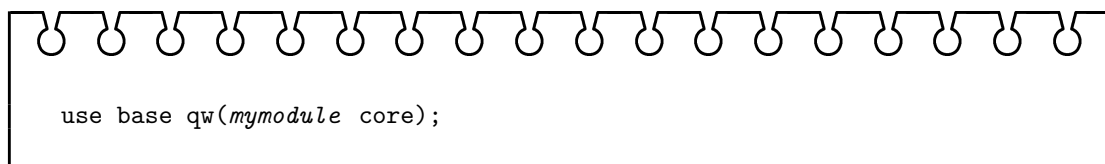
### 2.1 Module

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(my module 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 objects 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 Environment Object

Xcrypt's environment objects are implemented as Perl's objects (blessed hash references). In Xcrypt, environment objects should be typically created from hash references by a built-in function `&add_host` (Chapter 6 in detail).

## 2.5 Procedure

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

## 2.6 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 $XCRIPT/cpan; ./do-install.sh
```

If necessary, edit `$XCRIPT/etc/xcryptrc` along your environment.

Edit `$XCRIPT/bin/wrapper` along your environment for the purpose of making it possible to skip setting user environment variables.

### 3.3 Execution

If necessary, edit `$HOME/.xcryptrc` along your environment.

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

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

and write an Xcrypt script (e.g., `sample.xcr`). See Section 2.6 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.

### 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

```
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.

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

`copiedfilei`: 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

This module gathers jobs and returns one job of the same denotation. In detail, see the manual `bulk_module_manual.pdf`.

## 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<sup>1</sup> 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.,

```
&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
);
```

---

<sup>1</sup>[http://en.wikipedia.org/wiki/Bisection\\_method/](http://en.wikipedia.org/wiki/Bisection_method/)



## 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 *RANGE*

Its value is an array. The value is used for creating job objects. In detail, see Section 6.1.

### 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

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

is available.

### 5.3 *exe*

Its value denotes a Perl function. The function is executed as follows,

```
$ perl exe.pl
```

where *exe.pl* consists of only the function.

### 5.4 *header*

Its value denotes the set of Perl modules. The modules are read by the above *exe*.

### 5.5 *exe*

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

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

with `argij` explained below.

## 5.6 `argij`

Its values are arguments of a command.

## 5.7 `stdofile`

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

## 5.8 `stdefile`

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

## 5.9 `JSkey`

## 5.10 `env`

Its value denotes environments by which jobs generated by templates are surrounded. By default, an environment (denoting a local-modeemb Xcrypt execution) is embedded into templates. When you intend to execute jobs at remote hosts, it is enough to embed such an environment (typically returned by the embedded function `add_host`). In detail, see `add_host` in Section 6.

# 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<sup>1</sup>, 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]);
```

where \$VALUE[0] refers to an element of the value (i.e., array) of RANGE0.

Declarative description is also available as follows,

---

<sup>1</sup>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 (0, 2), (0, 4), (1, 2), (1, 4) which can be referred by \$VALUE[0] and \$VALUE[1], respectively.

This can be also written as

```

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

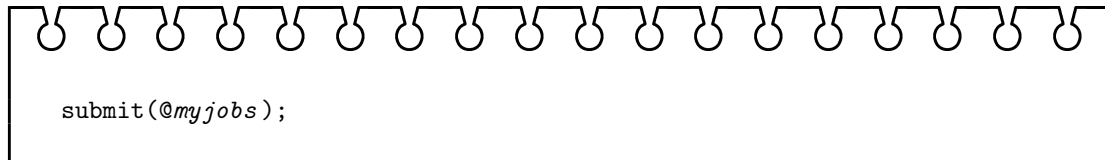
@jobs = prepare(%mytemplate);

```

## 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.

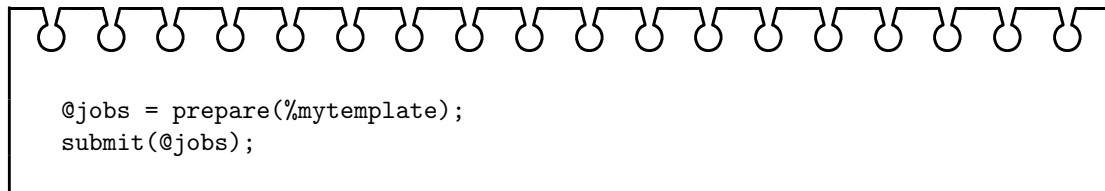
### Format



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

### Example

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



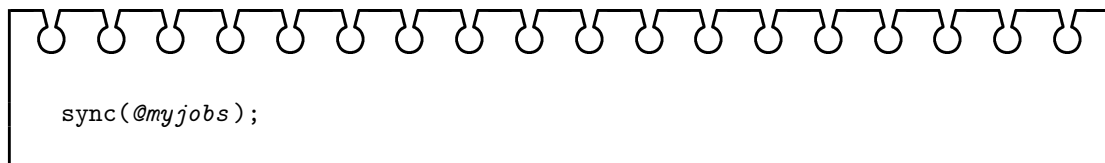
```
@jobs = prepare(%mytemplate);  
submit(@jobs);
```

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 add\_host

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

Entries of `host` and `sched` are required. By default, `wd` and `xd` have the values of `$HOME` and `$XCRIPT` as values, respectively.

### Format

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

### Example

```
$env = add_host({'host' => 'foo@bar',
                'sched' => 'torque',
                'wd' => '/home/foo/myapp',
                'xd' => '/usr/local/share/xcrypt'});
%template = ('id' => 'myjob', 'exe0' => './myexe', 'env' => $env);
```

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

## 6.5 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);
```

## 6.6 xcr\_exist

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

### Format

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

## 6.7 xcr\_qx

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

### Format

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

## 6.8 xcr\_system

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



### Format

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

## 6.9 xcr\_mkdir

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

### Format

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

## 6.10 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);
```

## 6.11 xcr\_rename

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

### Format

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

## 6.12 xcr\_symlink

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

**Format**

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

## 6.13 xcr\_unlink

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

**Format**

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

## 6.14 get\_from

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

**Format**

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

## 6.15 put\_into

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

### Format

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

## 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

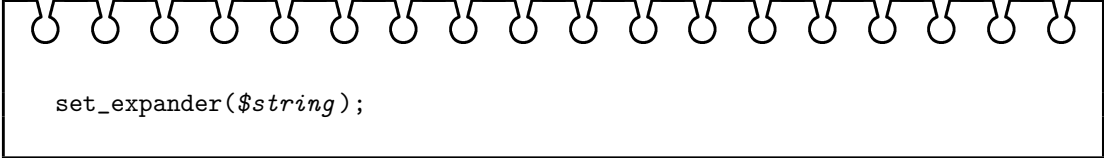
This function takes an Xcrypt's script code (denoted as *mystring*) and an integer *i*, and evaluates it each *i* seconds.

### Format

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

## 6.19 set\_expander

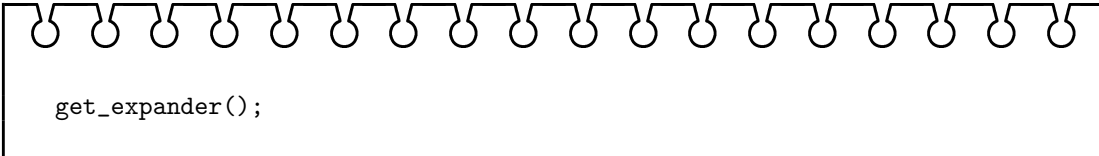
Format



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

## 6.20 get\_expander

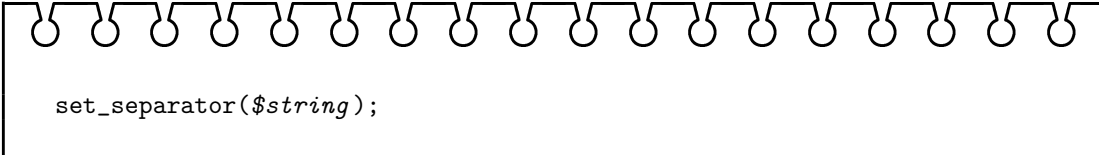
Format



```
get_expander();
```

## 6.21 set\_separator

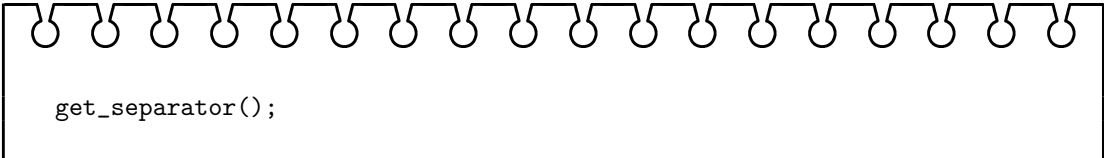
Format



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

## 6.22 get\_separator

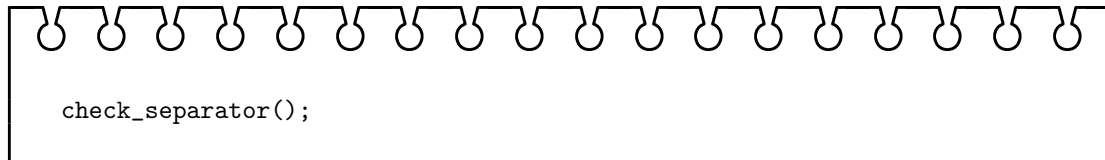
Format



```
get_separator();
```

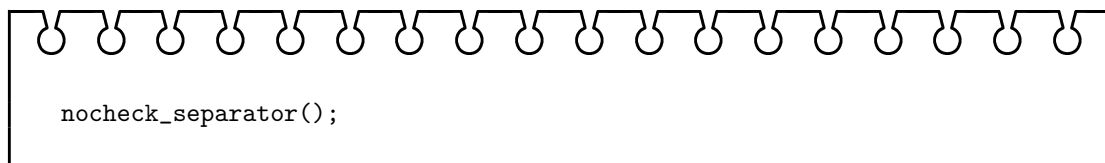
## 6.23 check\_separator

Format



## 6.24 nocheck\_separator

Format



## 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 `--config xcryptrc`

reads the *xcryptrc*, `~/.xcryptrc`, or `$XCRYPT/etc/xcryptrc`.

### 8.2 `--sched job_scheduler`

takes a batch job scheduler.

### 8.3 `--compile_only`

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

### 8.4 `--shared`

does not copy files between local and remote sites in remote-execution mode.

### 8.5 `--xbs-type type`

use Keiji Yamamoto's XBS<sup>1</sup> internally for generating job scripts of the *type*.

### 8.6 `--xbs-auto`

use XBS internally for generating job scripts (of an inferred type).

### 8.7 `--abort_check_interval num`

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

---

<sup>1</sup><http://www.il.is.s.u-tokyo.ac.jp/~yamamoto/xbs/>

**8.8** `--left_message_check_interval` *num*

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

**8.9** `--inventory_path` *path*

a path at that logs are located.

**8.10** `--verbose` *num*

**8.11** `--stack_size` *num*

**8.12** `--host` *username@hostname*

a host to that jobs are submitted.

**8.13** `--wd` *path*

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

**8.14** `--xd` *path*

**8.15** `--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,

```

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.,

```
%template = { ..., my_instance_member=>value, ... }
```

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.,

```
$mymodule::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<sup>1</sup>.

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

```
$class->NEXT::new($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 **news** in children classes call the **new** in the **core**.

### A.3.2 initially

to be written...

---

<sup>1</sup>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

```
$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 `starts` in children classes call the `start` in the `core`.

### A.3.5 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.3.6 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`.