HACKT CHPSIM

A simulator manual

David Fang

This manual describes the usage and operation of HACKT's `chpsim` simulator.

This document can also be found online at http://www.csl.cornell.edu/~fang/hackt/hacchpsim.

The main project home page is http://www.csl.cornell.edu/~fang/hackt/.

Short Contents

Table of Contents

List of Figures

1 Introduction

`chpsim` is a simulator for the CHP (Communicating Hardware Processes) language, which is based on Hoare's CSP (Communicating Sequential Processes) *C. A. R. Hoare. Communicating sequential processes. Communications of the ACM. 21(8):666–667, 1978.*. To distinguish this implementation from its predecessor, we name this simulator `hacchpsim`. However throughout this document, we use `chpsim` for brevity.

1.1 History

Where it all began...

History of original `chpsim`? Differences.

Inspiration from `mcc`, a concurrent dialect of the C language.

2 Usage

This chapter describes `chpsim`'s command-line options.

Usage: '`hacchpsim [options] obj-file`'

To pass a script to `hacchpsim`, use shell redirection or pipes. For example,

```
$ cat test.chpsimrc | hacchpsim -b test.haco
```

or

```
$ hacchpsim -b test.haco < test.chpsimrc
```

2.1 Option Summary

For options that take an argument, the space between the flag and the argument is optional.

none [User Option]
: With no arguments, just print a list of all command-line options, much like what is summarized below.

`-b` [User Option]
: Batch mode, non-interactive, promptless. This is useful for running scripts or piping in commands while suppressing prompts. This mode also turns off tab-completion in the interpreter. The opposing option is '`-i`'. **Note:** executables linked against `libeditline` may *require* this option for processing scripts due to a mishandling of EOF.

`-d` *checkpoint* [User Option]
: Produce a textual dump of a checkpoint binary. Exits without running the simulator.

`-f` *flag* [User Option]
: See Section 2.2 [General Flags], page 6.

`-h` [User Option]
: Help. Print command-line help and exit.

`-h` [User Option]
: Help. Print list of all interpreter commands and exit.

`-i` [User Option]
: Interactive, prompting. This is the default mode. The opposing option is '`-b`'.

`-I` *path* (repeatable) [User Option]
: Append *path* to the list of paths to search for sourcing other command scripts in the interpreter.

`-L` *path* (repeatable) [User Option]
: Append *path* to the list of paths to search for opening shared library plug-ins (modules). The equivalent command in the interpreter is [`dladdpath`], page 15. For more on building and loading shared-libraries, See Chapter 5 [Extending simulation], page 21.

`-l` *lib* (repeatable) [User Option]
: Load the *lib* shared library module for registering user-defined run-time functions. *lib* should be named *without* its file extension, for the sake of portability. For example, '`libcrunch.la`' should be referenced as '`libcrunch`', and '`chewy.so`' should be referenced as '`chewy`'. The equivalent command in the interpreter is [`dlopen`], page 15.

`-c` [User Option]
: Pass to indicate that input file is a source (to be compiled) as opposed to an object file.

`-C` *options* [User Option]
: When input is a source file, forward *options* to the compiler driver.

`-r` *file* [User Option]
: Startup the simulation already recording a trace file of every event. Trace file is automatically close when simulation exits. This is equivalent issuing `trace` command at the beginning of a simulation session.

`-t` *type* [User Option]
: Instead of expanding the whole top-level instances, only operate on the given type *type*, i.e. instantiate one instance of *type* as the top-level. This variation, however does **not** expand subinstances recursively, like the '`-T`' option. This is particularly useful for examining the CHP event structure of a particular definition.

`-T` *type* [User Option]
: Instantiate one instance of type *type* as the top-level, ignoring all previous top-level instances in the object file. This variation *does* recursively instantiate substructures. The ports of the instance of *type* (if any) will not be connected to any other processes. This is particularly useful for selecting test structures out of a collection of test structure definitions.

`-v` [User Option]
: Print version information and exit.

2.2 General Flags

For lack of better organization, many general purpose flags are folded into the '`-f`' option. Unless otherwise noted, all '`-f`' options have a '`no-`' prefixed counterpart, so '`-f no-disassemble`' is the intuitive negation of '`-f disassemble`'. Later options always override earlier options.

`-f check-structure` [User Option]
: Run additional internal graph (nodes and edges) consistency checks. Enabled by default.

`-f default` [User Option]
: Resets to default flags. Has no negation.

`-f dump-graph-alloc` [User Option]
Diagnostic tool. Produce a textual dump of expression allocation after the internal whole-program graph has been constructed.

`-f dump-dot-struct` [User Option]
Produce a textual netlist of the whole-program event graph in `dot` format[1]. A list of options that tune this output can be found in Section 2.3 [Graph Generation], page 7.

`-f run` [User Option]
Actually run the simulator's interpreter. Enabled by default. '`-f no-run`' is explicitly needed when all that is desired are diagnostic dumps.

`-f ack-loaded-fns` [User Option]
Print names of functions as they are loaded from dlopened modules. Default on. Mostly useful for diagnostics.

2.3 Graph Generation

The following flags are relevant only with '`-f dump-dot-struct`'. All of these options are also negatable with '`no-`' prefixed. Don't forget to pass '`-f no-run`' when not intending to run the interpreter.

`-f cluster-processes` [User Option]
Wrap process subgraphs into clusters, which are enveloped in rectangular outlines. Default off.

`-f show-channels` [User Option]
Label channel edges with their channel names. Default off.

`-f show-delays` [User Option]
Annotate event nodes with their delay values. Default off.

`-f show-event-index` [User Option]
Annotate event nodes with their globally allocated indices. Default off.

`-f show-instances` [User Option]
Also show allocated instances as nodes. Default off.

[1] `dot` is the name of a program (and its input language) that is part of AT&T's GraphViz package (open-source).

3 Tutorial

This chapter contains some small examples for quickly getting started and verifying that the simulator is working properly.

The examples in this chapter are taken out of the test suite and its testing library.

Enter the following definitions into a HAC file '`bool-lib.hac`':

```
defproc bool_buf (chan?(bool) L; chan!(bool) R) {
bool x;
chp { *[L?(x); R!(x)] }
}

template <pbool B>
defproc bool_buf_init (chan?(bool) L; chan!(bool) R) {
bool x;
chp { x:=B; *[R!(x); L?(x)] }
}

template <><pint N; pbool B[N]>
defproc bool_source_once(chan!(bool) S) {
chp {
{;i:N: S!(B[i]) }
}
}

template <><pint N; pbool B[N]>
defproc bool_source(chan!(bool) S) {
chp {
*[
  {;i:N: S!(B[i]) }
]
}
}

defproc bool_sink(chan?(bool) B) {
bool b;
chp {
*[ B?(b) ]
}
}
```

We will be reusing this file in many examples. Instantiate some of these types in another file '`bool-test-1.hac`'.

```
import "bool-lib.hac";

chan(bool) L, R;
bool_source SRC<4,{false,false,true,true}>(L);
bool_buf B(L, R);
```

```
bool_sink SNK(R);
```

Compile the file using `haco` (or one of the template Makefiles):

```
$ haco bool-test-1.hac bool-test-1.haco
$ haccreate bool-test-1.haco bool-test-1.haco-c
```

and run `hacchpsim`:

```
$ hacchpsim bool-test-1.haco-c
```

The `hacchpsim` simulator is interactive. You should see the following prompt:

```
chpsim>
```

The following session is an example of stepping through the simulator while watching data values on channels:

```
chpsim> get L
L : chan(bool<>) L = (0) [empty]
chpsim> get R
R : chan(bool<>) R = (0) [empty]
```

`get` queries the current state of a channel or data variable. All channels are initially empty (current values are meaningless), which is indicated by `[empty]`. Executed events are printed in a table format, whose headings are given by `print-event-header`.

1. `time` is the time of the event
2. `eid` is the (global) static event ID number (corresponding to a node in the event graph)
3. `pid` is the index of the process in which the event occurred
4. `event` is text for the CHP statement that executed
5. `cause` shows the critical predecessor event's *static* ID

For the critical predecessor to be printed we need to turn it on:

```
chpsim> cause
chpsim> watchall-events
chpsim> print-event-header
        time    eid     pid     event   cause
chpsim> step 20
        0       0       4       null
        10      5       2       B.L?(B.x)
        10      1       1       SRC.S!(false)   [by:5]
        20      6       2       B.R!(B.x)       [by:5]
        20      7       3       SNK.B?(SNK.b)   [by:6]
        30      5       2       B.L?(B.x)       [by:6]
        30      4       1       SRC.S!(false)   [by:5]
        40      6       2       B.R!(B.x)       [by:5]
        40      7       3       SNK.B?(SNK.b)   [by:6]
        50      5       2       B.L?(B.x)       [by:6]
        50      3       1       SRC.S!(true)    [by:5]
        60      6       2       B.R!(B.x)       [by:5]
        60      7       3       SNK.B?(SNK.b)   [by:6]
        70      5       2       B.L?(B.x)       [by:6]
        70      2       1       SRC.S!(true)    [by:5]
```

```
80      6       2       B.R!(B.x)       [by:5]
80      7       3       SNK.B?(SNK.b)   [by:6]
90      5       2       B.L?(B.x)       [by:6]
90      1       1       SRC.S!(false)   [by:5]
100     6       2       B.R!(B.x)       [by:5]
```

Above, the simulator displays a trace of 20 events as they are executed. The first event is always the *null* event, which starts all processes. The source repeated sends values to the buffer, and the buffer sends values to the receiver. Send and receive actions execute atomically in pairs. (`B.L` and `SRC.S` both refer to channel `L`.) This trace does not actually show what values were sent over the channels. We restart this example, watching the channel data values this time:

```
chpsim> initialize
chpsim> unwatchall-events
chpsim> watch-value L R
chpsim> step 20
updated channel(s):
chan(bool<>) L = (0) [recvd]
watch:  10      5       2       B.L?(B.x)
updated channel(s):
chan(bool<>) L = (0) [empty]
watch:  10      1       1       SRC.S!(false)   [by:5]
updated channel(s):
chan(bool<>) R = (0) [sent]
watch:  20      6       2       B.R!(B.x)       [by:5]
updated channel(s):
chan(bool<>) R = (0) [empty]
watch:  20      7       3       SNK.B?(SNK.b)   [by:6]
updated channel(s):
chan(bool<>) L = (0) [recvd]
watch:  30      5       2       B.L?(B.x)       [by:6]
updated channel(s):
chan(bool<>) L = (0) [empty]
watch:  30      4       1       SRC.S!(false)   [by:5]
updated channel(s):
chan(bool<>) R = (0) [sent]
watch:  40      6       2       B.R!(B.x)       [by:5]
updated channel(s):
chan(bool<>) R = (0) [empty]
watch:  40      7       3       SNK.B?(SNK.b)   [by:6]
updated channel(s):
chan(bool<>) L = (1) [recvd]
watch:  50      5       2       B.L?(B.x)       [by:6]
updated channel(s):
chan(bool<>) L = (1) [empty]
watch:  50      3       1       SRC.S!(true)    [by:5]
updated channel(s):
```

```
chan(bool<>) R = (1) [sent]
watch:  60      6       2       B.R!(B.x)       [by:5]
updated channel(s):
chan(bool<>) R = (1) [empty]
watch:  60      7       3       SNK.B?(SNK.b)   [by:6]
updated channel(s):
chan(bool<>) L = (1) [recvd]
watch:  70      5       2       B.L?(B.x)       [by:6]
updated channel(s):
chan(bool<>) L = (1) [empty]
watch:  70      2       1       SRC.S!(true)    [by:5]
updated channel(s):
chan(bool<>) R = (1) [sent]
watch:  80      6       2       B.R!(B.x)       [by:5]
updated channel(s):
chan(bool<>) R = (1) [empty]
watch:  80      7       3       SNK.B?(SNK.b)   [by:6]
updated channel(s):
chan(bool<>) L = (0) [recvd]
watch:  90      5       2       B.L?(B.x)       [by:6]
updated channel(s):
chan(bool<>) L = (0) [empty]
watch:  90      1       1       SRC.S!(false)   [by:5]
updated channel(s):
chan(bool<>) R = (0) [sent]
watch:  100     6       2       B.R!(B.x)       [by:5]
```

A list of all commands with brief descriptions is printed with '`help all`'. Please refer to Chapter 4 [Commands], page 13 for a comprehensive description of all `hacchpsim` commands.

Exercises.

Tracing.

4 Commands

This chapter describes all of the simulator's interactive commands. Commands are organized into the following categories:

'`builtin`' Built-in interpreter commands

'`general`' General-purpose commands

'`info`' Information about the simulation state

'`modes`' Simulator execution modes

'`simulation`'
Breakpoints and step control

'`tracing`' Checkpointing and tracing

'`view`' Simulation state monitoring and feedback

4.1 `builtin` commands

Built-in commands pertain to the interpreter, and have no relation to simulation.

`help` *cmd* [Command]
Help on command or category *cmd*. '`help all`' gives a list of all commands available in all categories. '`help help`' tells you how to use `help`.

`#` ... [Command]
`comment` ... [Command]
Whole line comment, ignored by interpreter.

`echo` *output* [Command]
Print *output* to stdout. **Note:** multiple spaces in *output* are compacted into single spaces by the interpreter's tokenizer.

`exit` [Command]
`quit` [Command]
Exit the simulator.

`abort` [Command]
Exit the simulator with a fatal (non-zero) exit status.

`precision` [*n*] [Command]
Sets the precision of real-valued numbers to be printed. Without an argument, this command just reports the current precision.

`repeat` *n cmd...* [Command]
Repeat a command *cmd* a fixed number of times, *n*. If there are any errors in during command processing, the loop will terminate early with a diagnostic message.

`history` [*start* [*end*]] [Command]
Prints command history. If no arguments given, then print entire command history. If only *start* is given, print to the most recent line. If *start* is negative, count backwards from last line. If *end* is positive, count forward from *start*. If *end* is negative, count backward from last line.

`history-noninteractive` [*on*|*off*] [Command]
Controls the recording of non-interactive commands in the history.

`history-save` *file* [Command]
Writes command line history to file *file*.

`history-rerun` *start* [*end*] [Command]
Reruns a set of previous commands. *start* is the first line to rerun. If *end* is omitted, only one line is rerun. If *end* is negative, count backwards from the most recent to determine the last line to run in the range. If *end* is positive, take that as the number of lines to execute from *start*, inclusive.

`interpret` [Command]
Open an interactive subshell of the interpreter, by re-opening the standard input stream. This is useful when you want to break in the middle of a non-interactive script and let the user take control temporarily before returning control back to the script. `Ctrl-D` sends the EOF signal to exit the current interactive level of input and return control to the parent.

`!` *shell-cmd* [Command]
Shell escape. Execute *shell-cmd* in parent shell, like the '`system`' library function, e.g. '`!date`'. Note: this preserves the rest of the line after the '!' verbatim.

The following commands are related to command aliases. Every command line given to the interpreter recursively expands the first token if it has a known alias. Aliases may reference to other aliases in the first token. The interpreter is smart enough to catch cyclic aliases and report an error.

`alias` *cmd args* [Command]
Defines an alias, whereby the interpreter expands *cmd* into *args* before interpreting the command. *args* may consist of multiple tokens. This is useful for shortening common commands.

`aliases` [Command]
Print a list of all known aliases registered with the interpreter.

`unalias` *cmd* [Command]
Undefines an existing alias *cmd*.

`unaliasall` [Command]
Undefines *all* aliases.

The following commands emulate a directory like interface for navigating the instance hierarchy, reminiscent of shells. By default, *all instance references are relative to the current working directory*, just like in a shell. Prefix with '`::`' to use absolute (from-the-top) reference. Go up levels of hierarchy with '`../`' prefix. The hierarchy separator is '`.`' (dot).

`cd` *dir* [Command]
Changes current working level of hierarchy.

`pushd` *dir* [Command]
Pushes new directory onto directory stack.

`popd` [Command]
Removes last entry on directory stack.

`pwd` [Command]
Prints current working directory.

`dirs` [Command]
Prints entire directory stack.

The following command is useful for showing each executed command.

`echo-commands` *arg* [Command]
Enables or disables echoing of each interpreted command and tracing through sourced script files. *arg* is either `"on"` or `"off"`. Default off.

4.2 `general` commands

The following commands relate to sourcing script files. Scripts may source other scripts. Cyclic scripts are detected and diagnosed as errors.

`source` *script* [Command]
Loads commands to the interpreter from the *script* file. File is searched through include paths given by the [‘`-I`’], page 5 command-line option or the [`addpath`], page 15 command.

`addpath` *path* [Command]
Appends *path* to the search path for sourcing scripts.

`paths` [Command]
Print the list of paths searched for source scripts.

The following commands relate to extending the simulator with user-defined functions in dynamically loaded shared libraries. More on shared modules can be found in Chapter 5 [Extending simulation], page 21.

`dladdpath` *paths ...* [Command]
Append *paths* to the list of paths to search for opening shared library modules. This is useful if you simply forget (or are too lazy) to pass the corresponding paths on the command-line. See also [the ‘`-L`’ option], page 5.

`dlopen` *lib* [Command]
Open shared library *lib* for loading external user-defined functions. Library is found by searching through user-specified load paths and the conventional library path environment variables. The command-line equivalent is the [‘`-l`’ option], page 6, following the same naming guidelines.

`dlpaths` [Command]
Prints the list of paths used in searching for dlopen-ing modules.

`dlcheckfunc` *funcs ...* [Command]
For each function named in *funcs*, report whether or not it has been bound to a symbol in a dynamically loaded module. Never errors out. See [command `dlassertfunc`], page 16.

`dlassertfunc` *funcs ...* [Command]
Error out if any function named in *funcs* is unbound to a module symbol. Useful for making sure a set of symbols is resolved before any execution begins. See [command `dlcheckfunc`], page 15.

`dlfuncs` [Command]
Print list of registered functions, from dlopened modules.

4.3 `info` commands

`assert-queue` [Command]
Error out if the event queue is empty. Useful as a quick check for deadlock.

`assertn-queue` [Command]
Error out if the event queue is not empty.

`queue` [Command]
Print an ordered list of all events in the checking event queue and execution event queue.

`dump-event` *event-id* [Command]
Print status information about event number *event-id*.

`dump-event-source` *event-id* [Command]
Print full-context of the source in which event *event-id* occurs.

`dump-all-event-source` [Command]
Print full-context of the source for all events.

`dump-state` [Command]
Print textual summary of entire state of simulation.

`get` *inst* [Command]
Print the state information about instance named *inst*. The name *inst* need not be canonical. Information includes current run-time value, if applicable.

`print-event-header` [Command]
Prints a table header suitable for interpreting printed event records.

`subscribers` *inst* [Command]
Print a list of all events currently subscribed to the value of variable *inst*. Such events are alerted for rechecking when value of *inst* changes.

`subscribers-all` [Command]
Print a list of all events currently subscribed to any variables.

`time` [Command]
Print the current simulator time.

`what` *inst* [Command]
Prints the type of the named instance *inst*, along with its canonical name.

`who` *inst* [Command]
`who-newline` *name* [Command]
Print all equivalent aliases of instance *name*. The '`-newline`' variant separates names by line instead of spaces for improved readability.

`ls` *name* [Command]
List immediate subinstances of the instance named *name*.

4.4 `modes` commands

`null-event-delay` [*delay*] [Command]
Without the *delay* argument, prints the value of the delay used for "trivial" events. With the *delay* argument, sets the said delay value.

`timing` *mode* [Command]
Select timing mode for event delays. *mode* can be one of the following:

'`uniform`' Use the same delay for all events, set by `uniform-delay`.

'`random`' Use a high-entropy random variable delay.

'`per-event`'
Use the delay specified by each individual event.

`seed48` [*int int int*] [Command]
Corresponds to libc's seed48 function. With no argument, print the current values of the internal random number seed. With three (unsigned short) integers, sets the random number seed. Note: the seed is automatically saved and restored in checkpoints.

`uniform-delay` [*delay*] [Command]
The uniform delay value only takes effect in the `uniform` timing mode. Without the *delay* argument, prints the value of the delay. With the *delay* argument, sets the said delay value.

4.5 `simulation` commands

`initialize` [Command]
Resets the variable state of the simulation, while preserving other settings such as mode and breakpoints.

`reset` [Command]
Similar to `initialize`, but also resets all modes to their default values.

The following commands run the simulation. Simulation is interrupted if a run-time error occurs, or a breakpoint is tripped. *`Ctrl-c`* or '`SIGINT`' (from '`kill -INT`') interrupts the simulation and returns control back to the interpreter in interactive mode.

`advance` *delay* [Command]
Advances the simulation *delay* units of time.

`advance-to` *t* [Command]
Advances the simulation *until* time *t*.

`step` *n* [Command]
Advances the simulation by *n* steps.

`run` [Command]
Runs the simulation until the event queue is empty, if ever.

The following commands pertain to breakpoints.

`break-event` *event-id* [Command]
Stop the simulation when event *event-id* executes.

`break-value` *inst* [Command]
Stop the simulation when variable *inst* is written, event when its value does not change.

`unbreak-event` *event-id* [Command]
Remove breakpoint on event *event-id*.

`unbreak-value` *inst* [Command]
Remove breakpoint on variable *inst*.

`show-event-breaks` [Command]
List all event breakpoints.

`show-value-breaks` [Command]
List all variable breakpoints.

`unbreakall-events` [Command]
Removes all event breakpoints.

`unbreakall-values` [Command]
Removes all variable breakpoints.

4.6 `tracing` commands

Checkpointing is useful for saving long simulations. Checkpoint files are only valid for simulations that load *the same object file* that was used to produce the checkpoint. A few minimal consistency checks are performed to alert the user of a mistake.

The structure of the whole-program (after state allocation) is not retained in the checkpoint; rather, it is regenerated from the object file. **Note:** Only the state of variables (their values) and events is written to the checkpoint. Simulator modes, breakpoints, watchpoints, and dlopen-ed modules are *not* preserved in checkpoints, nor are they affected by loading checkpoints.

TODO: periodic checkpointing, via `auto-save`.

`save` *ckpt* [Command]
Saves the current simulator state to a checkpoint file *ckpt* that can be restored later. Overwrites *ckpt* if it already exists.

`load` *ckpt* [Command]
: Restores the simulator state (variables and events) from a checkpoint file *ckpt*. Loading a checkpoint will not overwrite the current status of the auto-save file, the previous autosave command will keep effect. Loading a checkpoint, however, will close any open tracing streams.

Entire execution traces may be saved away for offline analysis. Again the structure of the whole-program (after state allocation) is not recorded in the trace; rather, it is assumed from the object file.

TODO: Section on trace file details and internals?

`trace` *file* [Command]
: Record events to tracefile *file*. Overwrites *file* if it already exists. A trace stream is automatically closed when the `initialize` or `reset` commands are invoked. See the '`-r`' option for starting up the simulator with a newly opened trace stream.

`trace-file` [Command]
: Print the name of the currently opened trace file.

`trace-close` [Command]
: Finish writing the currently opened trace file by flushing out the last epoch and concatenating the header with the stream body. Trace is automatically closed when the simulator exits.

`trace-dump` *file* [Command]
: Produce textual dump of trace file contents in *file*.

`trace-flush-interval` *steps* [Command]
: If *steps* is given, set the size of each epoch according to the number of events executed, otherwise report the current epoch size. This regulates the granularity of saving traces in a space-time tradeoff.

`trace-flush-notify` [*0*|*1*] [Command]
: Enable (1) or disable (0) notifications when trace epochs are flushed.

4.7 `view` commands

The `view` category commands controls what information is printed by the simulator as it executes events. Watchpoints are similar to breakpoints in the feedback that is printed, but without interrupting simulation.

`cause` [Command]
: Show causes of events when events are printed.

`nocause` [Command]
: Turn off `cause` in feedback.

`watchall-queue` [Command]
: Print events as they enter the event queue (either for checking or execution). This is generally recommended for debugging, as it prints *a lot* of information.

`nowatchall-queue` [Command]
Disables `watchall-queue`.

`watch-event` *event-id* [Command]
Watchpoint. Print event *event-id* each time it executes, without interrupting.

`unwatch-event` *event-id* [Command]
Remove watchpoint on event *event-id*.

`watch-value` *inst* [Command]
Print events that write to *inst* as they execute.

`unwatch-value` *inst* [Command]
Stop watching *inst*.

`show-event-watches` [Command]
Print list of all watched events.

`show-event-values` [Command]
Print list of all watched variables.

`watchall-events` [Command]
Print all events as they execute, *regardless of whether or not they are explicitly watched*.

`nowatchall-events` [Command]
Stop printing all events, but keep printing events that are explicitly listed watchpoints. This is particularly useful for temporarily watching all events in detail, and later restoring only explicitly watched events.

5 Extending the simulator

This chapter describes the procedure for providing user-defined functions through an external shared library module (or plug-in).

For this chapter (and in your own work) we *strongly* recommend including the following template Makefile in your own working Makefile:

```
# "Makefile"
include prefix/share/hackt/mk/hackt-lt.mk
```

where `prefix` refers to the base installation path of the tools. This Makefile template simplifies compilation and linking tremendously, and also provide suffix compilation rules for HAC object files. We will refer to some definitions found in '`hackt-lt.mk`' throughout this section.

5.1 CHP Function Calls

In CHP, function calls may appear in expressions or as standalone statements. Function call syntax is similar to that of C, and functions may take arbitrarily many arguments, or no arguments at all.

```
defproc func(chan?(int) A, B, chan!(int) C) {
  int a, b;
  chp {
    *[ A?(a), B?(b);
       alert_me();          // alert_me is a yet undefined function
       C!(twiddle(a,b))     // twiddle is a yet undefined function
     ]
  }
}
```

Definitions such as the above can be compiled (by `haco`) all the way through creation (`haccreate`) and state allocation (`hacalloc`) without errors. All such nonmeta (run-time) functions are only bound at *run-time*. A consequence of such late binding is that the types and number of parameters of function calls cannot be checked at compile time.

With the above example, instantiate some environment of sources and sinks:

```
chan(int) X, Y, Z;
func F(X, Y, Z);

chp { *[X!(1)] }     // value source
chp { *[Y!(2)] }
chp { *[Z?]    }     // value sink
```

If your file is called '`chptest.hac`', run `make chptest.haco-a`. This will compile, create, and allocate the state in an object file. If you only make the '`.haco`' or '`.haco-c`' object files, `hacchpsim` will automatically compile the object file as much as necessary.

You can now run the simulation:

```
$ hacchpsim chptest.haco-a
chpsim> watchall-events
chpsim> run
```

```
...
```
error Eventually tries to call unbound function `alert_me`.

Such errors can be caught earlier using the [dlassertfunc], page 16 command.

Next we compile a libary to provide these missing functions.

5.2 Shared Module Creation

There are two parts to building a shared library module: compiling and linking.

5.2.1 Compiling module sources

A typical `chpsim`-module source file (C++) is organized as follows:

```
// "chptest.hac"
// include headers
#include <sim/chpsim/chpsim_dlfunction.h>

// using declarations
USING_CHPSIM_DLFUNCTION_PROLOGUE

// function definitions
static
void
my_alert(void) {
  // your code here
}

// a module export macro
CHP_DLFUNCTION_LOAD_DEFAULT("alert_me", my_alert)

static
int_value_type
compute(const int_value_type a, const int_value_type b) {
  // return some function of a and b
}

CHP_DLFUNCTION_LOAD_DEFAULT("twiddle", compute)
```

The header '`sim/chpsim/chpsim_dlfunction.h`' should have been installed in the `pkgincludedir`, '`$(prefix)/include/hackt/`'. This header defines the macros and prototypes used in the rest of the source. To add the header path to the search paths during compilation (actually, preprocessing), either pass the path directly '`-I` *`pkgincludedir`*', or pass it indirectly using shell expansion and `hackt-config`: '`-I` `` `hackt-config --cflags` ``'. The latter method is preferred because it works across hosts with tools installed in different paths. The '`hackt-lt.mk`' template Makefile appends this flag to *CPPFLAGS* for you automatically, when compiling `chpsim` module objects.

`USING_CHPSIM_DLFUNCTION_PROLOGUE` [Macro]
This just imports certain type names from the header into the current namespace with C++ using-directives. The details are not important. For compatibility, one should always use this macro and let the preprocessor expand its definition.

`CHP_DLFUNCTION_LOAD_DEFAULT` *name sym* [Macro]
This is the macro that is responsible for binding the library symbol *sym* to a name of the user's choice *name*, a string. Name binding occurs automatically as soon as the module is loaded (by dlopen). (If you must know, this is achieved through static object initialization.) The C++ function symbol *sym* must be a prototype that uses the restricted set of types: `int_value_type` and such.

`REGISTER_DLFUNCTION_RAW` *name sym* [Macro]
This macro is responsible for binding the library symbol *sym* to a name of the user's choice *name*, a string. The function prototype of *sym* must be of the form: '`chp_function_return_type (*) (const chp_function_argument_list_type&)`'.

`chp_function_return_type` [Data type]
This is the return type used in `chpsim`'s run-time environment. This is a memory-managed pointer (reference-counted) to an abstract expression type. Defined in '`Object/expr/dlfunction_fwd.h`'.

`chp_function_argument_list_type` [Data type]
This is the argument list type passed to all registered CHP functions. From this list-type, arguments are automatically extracted and passed to native C++ functions by `CHP_DLFUNCTION_LOAD_DEFAULT`. This is typically a list of memory-managed pointers to abstract expression types. Defined in '`Object/expr/dlfunction_fwd.h`'.

You may have noticed that the '`compute`' function references return types and argument type `int_value_type`. A few such types are defined in the interface to `chpsim`'s run-time. These types are defined in the header '`Object/expr/types.h`'.

`int_value_type` [Data type]
The signed integer data type, corresponding to '`int<W>`' in CHP, typically defined to the host machine's native integer type.

`bool_value_type` [Data type]
The boolean data type, corresponding to '`bool`' in CHP, typically defined to a C++ `bool`, or the smallest character type.

`real_value_type` [Data type]
The floating-point data type, corresponding to '`real`' in CHP, typically defined to `float` or `double`.

`string_value_type` [Data type]
The string data type, typically defined to a C++ `std::string`. **Recommendation:** When defining functions that take this type as an argument, pass it by reference, for example: '`void dump_string(const string_value_type&);`'.

All functions that are registered with `CHP_DLFUNCTION_LOAD_DEFAULT()` are required to use only the above types in argument types and return types (and `void`). If your function uses different but convertible types, then write a call-wrapper that uses only the allowed types and forwards the arguments and return values. This is necessary when compiling and linking against symbols that belong to libraries beyond your control, or when you simply don't want to alter an existing library. When in doubt, it is always safe to use a such a wrapper. It is possible to change these types (say, to increase precision) if the entire suite of HACKT tools is re-compiled.

Compiling the source file for a shared library requires some additional measures. Fortunately, with the aid of conveniently installed template Makefiles, the complexities are hidden[1]. '`hackt-lt.mk`' provides a suffix rule for compiling C++ files ending with '`.cc`' to Libtool-wrapped object files '`.lo`'.

For every C++ source file ('`.cc`') that is to be linked into the `chpsim` module, its corresponding object file should be referenced with the '`.lo`' extension (for Libtool object). The next section describes how to correctly link a `chpsim` module.

5.2.2 Linking module libraries

In your working Makefile, you will refer to target libraries with a '`.la`' extension (Libtool archive). The '`.la`' extension replaces what would normally be '`.so`', '`.dylib`', '`.dll`', or the native shared-object extension. Libtool provides a platform-independent abstraction of shared libraries, so the user need not worry about these details. The target library name need not be prefixed with '`lib`', since it is being dlopened as a module (plug-in). Suppose the above source file was named '`foo.cc`', and our target library is '`bar.la`', one might write in the Makefile:

```
# "Makefile" (continued)
# list of dependent libraries (-l...)
bar_la_LIBADD =

# required flags
bar_la_LDFLAGS = $(CHPSIM_MODULE_FLAGS)

# -L search paths to dependent libraries
# bar_la_LDFLAGS +=

# object file list
bar_la_OBJECTS = foo.lo

bar.la: $(bar_la_OBJECTS)
        $(CXXLINK) $(bar_la_LDFLAGS) $(bar_la_OBJECTS) $(bar_la_LIBADD)
```

The '`bar.la`' to '`bar_la`' name canonicalization is borrowed from Automake's variable naming convention. We've referenced some variables in the Makefile, defined in '`hackt-lt.mk`':

[1] Such complexities include additional compiler flags for shared-library objects, such as PIC (position-independent-code).

`CHPSIM_MODULE_FLAGS` [Makefile variable]
: Flags that tell Libtool to link the shared library to be suitable for dlopening (dynamic loading). Value should remain unmodified.

`CXXLINK` [Makefile variable]
: The aggregate link command (without arguments).

  Invokes `hackt-libtool` as a link wrapper. Should remain unmodified. Depends on the *CXX* Makefile variable.

`CXX` [Makefile variable]
: The user should define the C++ compiler, which is also to be invoked as the linker. Autoconf users may wish to set this automatically through a `configure` script, e.g. `CXX = @CXX@` in Makefile.in.

Other relevant variables are also provided:

`HACKT_LIBTOOL` [Makefile variable]
: Defined to `hackt-libtool`, which is expected to be in the PATH. This is a renamed copy of the '`libtool`' script that was configured during the compilation of the tools. This has the advantage of storing and re-using all of the flags needed for building shared libraries on the host platform, thus saving the user from having to do any configure-detection when using `chpsim`.

`HACKT_CONFIG` [Makefile variable]
: Defined to `hackt-config`, which is expected to be in the PATH. This script contains package installation information such as include header paths and libaries. For building `chpsim` modules, only a few compile time options are needed, no additional libraries are needed.

`CPPFLAGS` [Makefile variable]
`CHPSIM_OBJECT_CPPFLAGS` [Makefile variable]
: Expands to flags needed to compile `chpsim` module source files. Gratuitously appled to all libtoolized compilations. *CPPFLAGS* may be appended by the user, but *CHPSIM_OBJECT_CPPFLAGS* may not.

`CXXFLAGS` [Makefile variable]
: Intially empty, may be appended by the user for tuning compilation.

Summary: defining *CXX* suffices to successfully build `chpsim` module '`bar.la`' in the above example. One word of caution: the '`.la`' file is merely a placeholder that tells libtool where to find the actual built archives, which are actually built in the '`.libs`' subdirectory. Don't expect to be able to move these files arbitrarily without breaking. (There's still a good chance of it working because the libraries are not built for use in an installed location.)

Now you can build the module with '`make bar.la`'.

5.3 Run-time Module Loading

With our built module, we can now load it into `hacchpsim`. One way is to pass libraries on the command line using the ['-l' option], page 6 and the ['-L' option], page 5.

```
$ hacchpsim -lbar chptest.haco-a
loaded function: 'alert_me'
loaded function: 'twiddle'
chpsim> watchall-events
chpsim> step 40
```

The alternative is to add library paths and load libraries after the simulator is launched.

```
$ hacchpsim chptest.haco-a
chpsim> dlopen bar
loaded function: 'alert_me'
loaded function: 'twiddle'
chpsim> watchall-events
chpsim> step 40
```

Loading libraries on the command-line and in the interpreter is allowed, and works as expected. `chpsim` can load arbitrarily many libraries (within the limits of the operating system), and as long as the registered function names don't conflict.

5.4 Run-time Diagnostics

`chpsim` performs some run-time checks on the dynamically loaded functions when they are called. Run-time errors will occur under any of the following conditions:

- If a function is called but yet unbound
- If function name-collision occurs while loading modules
- A function is passed an argument with the wrong type
- A function is passed too few arguments
- The called function throws an exception, or calls `abort` or `terminate`

Note that passing too many arguments does not trigger an error, the excess arguments are simply dropped.

5.5 An Example

A complete example (from the test suite) is installed under:

'*prefix*`/share/hackt/doc/examples/chpsim-function/`'

The files contained therein are '`README`', '`hello.cc`', '`hello-test.hac`', and '`Makefile.in`'.

It is very similar to the example described earlier in this chapter. To run this example (VPATH make using remote source directory), do the following:

1. Create a new empty working directory, and enter it.

   For example, '`mkdir play && cd play`'.
2. Copy the example's '`Makefile.in`' locally to '`Makefile`'.
3. Edit the '`Makefile`' (instructions included therein):

a. Point `srcdir` and `VPATH` to
'*prefix*`/share/hackt/doc/examples/chpsim-function`'.
b. Delete the other variables in the preamble (optional).
c. Point the include line to the installed '`hackt-lt.mk`'.
d. Set *CXX* to your C++ compiler.

If you run `make`, it will perform all the necessary compilation steps and dump a short run of the simulation to an output file '`hello-test.chpsimrc-out`'.

Another output file '`confirm-exec`' is produced to confirm that all the necessary toolchain executables are found in your PATH. Some additional clean targets are also provided.

If you want to modify the examples, the (non-VPATH) alternative is to copy the whole example directory and dispense with setting `srcdir` and `VPATH`.

5.6 Global Initialization

External functions often access global variables outside the scope of the simulator. Global variables and objects in modules can be initialized in several ways. One way is to initialize object through their class constructors. Global objects are constructed in source order in their respective translation units, which is also how functions are registered. Plain-old-data, however, are often initialized through a function call. Another way is to do initialization work (directly, or through function call) in a static object constructor.

```
class module_init {
public:
    module_init() {
        // initializations and resource acquisitions...
    }
    ~module_init() {
        // deallocations and resource releasing...
    }
};

static const module_init __init__;
```

Above, the `__init__` object's constructor will perform the initializations and resource allocations for this module. Since modules are not explicitly unloaded (by '`dlclose`'), module destruction will not occur until `chpsim` exits. [Ask me if you need destruction earlier.]

C++ Reminder: global object destruction occurs in the reverse order of construction. C Reminder: the order of initialization between different translation units (even in the same module) is undefined, and should not be relied upon.

5.7 Module Rationale

The goals of our implementation of run-time bound user-defined functions were:

- Convenience: eliminate unnecessary compilation of the toolchain

- Ease: installed Makefile templates provide simple interface *without* expecting the user to deal with additional tools such as Autoconf, Automake, and Libtool
- Flexibility: be able to re-bind functions by simulating the same object file with different modules
- Portability: provide a consistent interface for shared library management across all platforms (thanks to GNU Libtool!)
- Safety: provide reasonable run-time type checking on functions

You be the judge of how we fared.

6 Event-driven Execution

Describe event-driven execution algorithm. Do we really need to devote an entire chapter to this? (Mutters to self: really need to organize this document...)

6.1 Event ordering

Figure: Obsolete: CHP simulator event life cycle

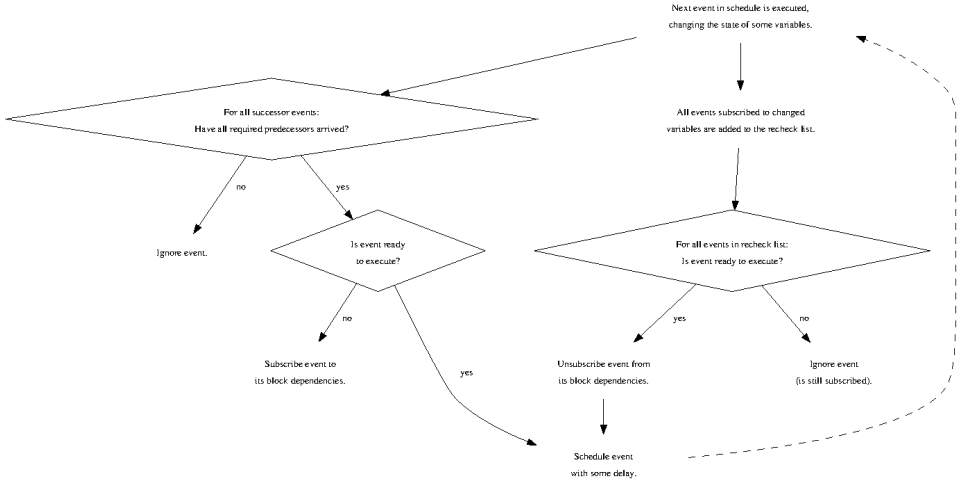
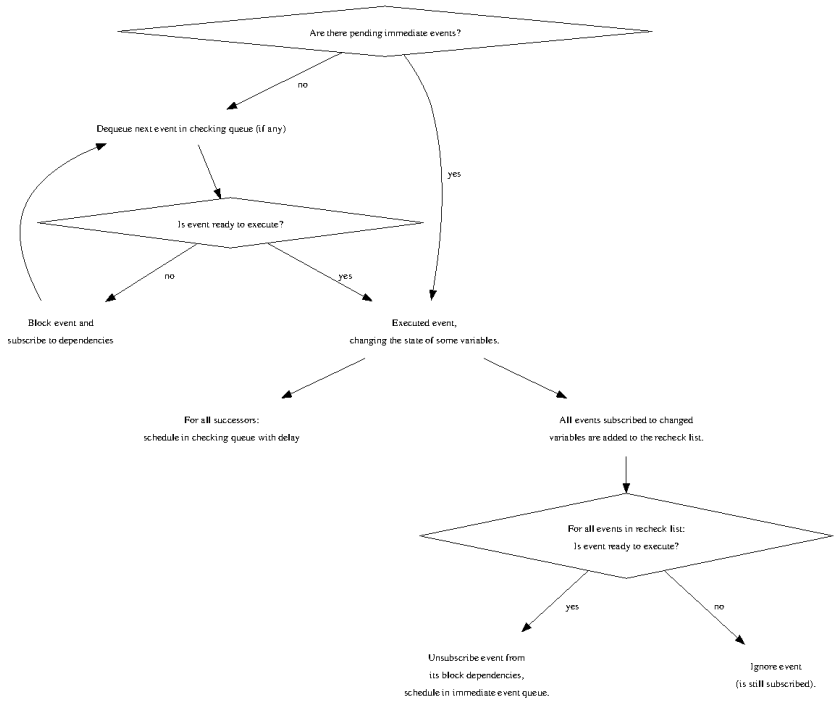


Figure: CHP simulator event algorithm



The whole-program event graph is composed of concurrent event graphs representing each process. The edges in the event graph represent the legal orderings between events, including cycles. Any process's execution trace is a projection of the entire program's execution trace (and all possible traces).

6.2 Timing

The delays associated with each event are *prefix* delays, i.e. the delays are applied before the event is *checked for the first time*[1]. In other words, every successor of an event that just executed is scheduled for future checking using the delay of each successor. When an event is checked for the first time, it is either blocked or executed immediately. When an event is blocked, it is subscribed to its dependencies (or some conservative superset thereof). When an event is unblocked, it is unsubscribed from its dependencies and placed into the *immediate execution queue* because it has already paid its delay up front, and thus should not be delayed further. The immediate execution queue contains only unblocked events that take precedence over the checking queue.

[1] One reason why a prefix-delay model was chosen was to facilitate the pseudo-atomic execution of send-receive event pairs. Pseudo-atomicity arises from the fact paired-events are still executed individually, but guaranteed to share the exact same timestamp. Had we chosen to use suffix or infix delays, delays would be applied *after* unblocking, resulting in potentially different timestamps.

7 Standard Library Functions

This section describes the usage of existing library functions. These library functions can be loaded by dlopening '`stdchpfn.la`', which is located in '*`pkglibdir`*' (like '`prefix/lib/hackt`').

To load the standard library:

```
$ hacchpsim -Lprefix/lib/hackt -lstdchpfn foo.obj
```

To help with passing the correct library flags, you can also invoke '`hackt-config --ldflags`' to automatically expand the flags:

```
$ hacchpsim `hackt-config --ldflags` -lstdchpfn foo.obj
```

7.1 Function Descriptions

This section gives brief usage descriptions of all functions available from the standard `chpsim` library.

7.1.1 Diagnostics

`assert` *z* [Function]
: Run-time invariant check. If *z* is false, throw an exception and halt the simulator, who is expected to produce some diagnostic message.

7.1.2 Conditionals

`bcond` *z a b* [Function]
: Conditional expression, for boolean rvalues. If *z* is true, return *a*, else return *b*. NOTE: both expressions *a* and *b* are evaluated *unconditionally*.

`zcond` *z a b* [Function]
: Conditional expression, for integer rvalues. If *z* is true, return *a*, else return *b*. NOTE: both expressions *a* and *b* are evaluated *unconditionally*.

`rcond` *z a b* [Function]
: Conditional expression, for floating-point (real) rvalues. If *z* is true, return *a*, else return *b*. NOTE: both expressions *a* and *b* are evaluated *unconditionally*.

`strcond` *z a b* [Function]
: Conditional expression, for string rvalues. If *z* is true, return *a*, else return *b*. NOTE: both expressions *a* and *b* are evaluated *unconditionally*.

`select` *index args...* [Function]
: Returns expression *args[index]*, where *index* is 0-based. Throws run-time exception if *index* is out-of-range.

7.1.3 Strings

`strcat` *args...* [Fucntion]
`sprint` *args...* [Fucntion]
`tostring` *args...* [Fucntion]
: Stringify all *args* and concatenate into a single string (returned). This can be used to convert argument types to a string. Does not include terminating newline.

`strtoz` *str* [Function]
Convert string *str* to an integer. Throws run-time exception if conversion fails.

`strtob` *str* [Function]
Convert string *str* (0 or 1) to a boolean. Throws run-time exception if conversion fails.

`strtor` *str* [Function]
Convert string *str* to real (floating-point) value. Throws run-time exception if conversion fails.

7.1.4 Input/Output

The first group of functions operate on '`stdin`' and '`stdout`' for interactiion with the user. The `scan` input functions with throw a run-time exception upon conversion failure.

`echo` *args...* [Function]
`cout` *args...* [Function]
`print` *args...* [Function]
Prints all arguments sequentially to '`stdout`'. The `print` variant includes a terminating newline, while the others do not.

`printerr` *args...* [Function]
`cerr` *args...* [Function]
Prints all arguments sequentially to '`stderr`'. The `printerr` variant includes a terminating newline, while the others do not.

`zscan` [Function]
`dzscan` [Function]
Read an integer from '`stdin`'. Use with caution, because events in the simulator are relatively asynchronous. The 'd' in the `dzscan` command alias is for decimal, base-10. See also `zscan_prompt`.

`zscan_prompt` *str* [Function]
`dzscan_prompt` *str* [Function]
Same as `zscan`, but takes a prompt string *str* as an argument and prints it to prompt the user.

`bzscan` [Function]
Reads an integer, expected in binary, containing only 0's and 1's. Input should exclude any "0b" prefix.

`bzscan_prompt` *str* [Function]
Prompts use to enter an integer in binary.

`xzscan` [Function]
Reads an integer, expected in hexadecimal. Input may include an optional "0x" prefix.

`xzscan_prompt` *str* [Function]
Prompts use to enter an integer in hexadecimal.

`bscan` [Function]
: Read a boolean (0 or 1) from '`stdin`'. Use with caution, because events in the simulator are relatively asynchronous. See also `bscan_prompt`.

`bscan_prompt` *str* [Function]
: Same as `bscan`, but takes a prompt string *str* as an argument and prints it to prompt the user.

`sscan` [Function]
: Reads a newline-terminated string from '`stdin`'.

`sscan_prompt` *str* [Function]
: Same as `sscan`, but takes a prompt string *str* as an argument and prints it to prompt the user.

TODO: `rscan` is not yet available, but is trivial to add.

I/O can also operate on file streams.

`fopen` *file* [Function]
: Open file *file* for writing, overwrite previous contents. Subsequent calls to `fprint` will still continue to append to the file. If the file stream is already open, do nothing. Return true if the stream is opened successfully (or was already open).

`fappend` *file* [Function]
: Like `fopen`, except *file* is first opened in append mode, to not overwrite existing file. Call this before `fprint` to append to *file*.

`fprint` *file args...* [Function]
: Print *args* to file *file* by appending. Throw run-time exception if opening file fails. File streams are automatically closed and flushed upon library closing.

`fclose` *file* [Function]
: Close and flush file input and output stream(s) *file*.

`fflush` *file* [Function]
: Flush output file stream *file*.

`fzscan` *file* [Function]
`fdzscan` *file* [Function]
: Read the next integer from input file *file*. Expects integer in decimal. Automatically opens new input file stream when referenced first time.

`fbzscan` *file* [Function]
: Same as `fzscan`, but expects integer in binary.

`fxzscan` *file* [Function]
: Same as `fzscan`, but expects integer in binary.

`fbscan` *file* [Function]
: Read the next boolean from input file *file*. Automatically opens new input file stream when referenced first time.

`fsscan` *file* [Function]
: Read the next boolean from input file *file*. Automatically opens new input file stream when referenced first time.

The following variants automatically restart a file stream once it reaches the end.

`fzscan_loop` *file* [Function]
`fdzscan_loop` *file* [Function]
: Read the next integer from input file *file*. Re-opens file to beginning after EOF is reached.

`fbzscan_loop` *file* [Function]
: Like `fzscan_loop`, but expects integer in binary.

`fxzscan_loop` *file* [Function]
: Like `fzscan_loop`, but expects integer in hexadecimal.

`fbscan_loop` *file* [Function]
: Read the next boolean from input file *file*. Re-opens file to beginning after EOF is reached.

`fsscan_loop` *file* [Function]
: Read the next boolean from input file *file*. Re-opens file to beginning after EOF is reached.

7.1.5 Operating System Library

System-related operations are also supported.

`system` *cmd* [Function]
: Execute the command *cmd* in the parent shell. Returns the exit status.

7.1.6 Bit-manipulation Library

The following functions are provided as fast implementations of low-level bitwise manipulation functions.

`parity` *int* [Function]
: Returns parity of *int*, true if odd-parity, false if even-parity.

`popcount` *int* [Function]
: Returns the number of set bits in the binary representation of *int*.

`clz32` *int* [Function]
: Returns the number of leading 0s in the 32b binary representation of *int*. (Leading 0s are in the more significant bit positions.) Result is undefined if *int* is 0.

`ctz` *int* [Function]
: Returns the number of trailing 0s in the binary representation of *int*. (Trailing 0s are in the less significant bit positions.) Result is undefined if *int* is 0.

`ffs` *int* [Function]
: Returns the 1-indexed position of the first set bit in the binary representation of *int*. Position is counted from the least significant bits. Returns 0 if *int* is 0.

`msb` *int* [Function]
Return the 0-indexed position of the most significant bit. Undefined for *int* 0.

`lsb` *int* [Function]
Return the 0-indexed position of the least significant bit. Undefined for *int* 0.

7.2 Library Use Example

An example project that uses some standard library functions is installed under '*pkg-datadir*`/doc/examples/chpsim-stdlib/`'.

7.3 Function Renaming

This section describes ways in which one can customize the standard library. Perhaps you don't like the names chosen, or you would like to remap selected functions to your own versions.

The source file for '`stdchpfn.la`' is also installed as '`stdchpfn.cc`' in the same location (`pkglibdir`). It merely contains function registration macros that map functions defined in '`libstdchpfn.la`' to names used in `chpsim`. One can use this file as a starting point for remapping to functions to different names, or names to different functions. To re-use the existing underlying functions, just link your new library against '`-L`*pkglibdir* `-lstdchpfn`' (in *LIBADD*, by Libtool/Automake convention), which resolves to '`libstdchpfn.la`', not the dlopenable module, '`stdchpfn.la`'.

7.4 Library Organization

The intentional separation between '`libstdchpfn.la`' and '`stdchpfn.la`' demonstrates how one can build a `chpsim` module on top of an existing C or C++ library without modifying it.

Q: When passing a module to `chpsim` with '`-l`' or dlopen, how does the system know what dependent libraries to load? A: The '`.la`' Libtool archives encode library dependencies for shared libraries, and (Libtool) libltdl's lt_dlopen automatically takes care of dependent libraries for the user.

Command Index

Variable Index

Concept Index

O

P

R

S

T

U

V

W