

# The CPC manual

Juliusz Chroboczek, Gabriel Kerneis  
<jch@pps.jussieu.fr>, <kerneis@pps.jussieu.fr>

15 November 2010

# Chapter 1

## The CPC language

CPC is a programming language designed for writing concurrent programs, for example network servers and clients. CPC is implemented as a source-to-source translation from CPC into plain C using a technique known as *translation into Continuation Passing Style* (CPS) [SW74, Plø75].

The main abstraction provided by CPC is a *CPC thread*. From the programmer's view, a CPC thread roughly corresponds to other programming languages' notion of *thread* or *lightweight process*, except that it has no identity: there is no *thread identifier* that can be used to kill or suspend a given CPC thread.

A CPC thread can be in one of two states: *attached* to the CPC scheduler, and *detached*. At a given time, the set of all attached threads are scheduled cooperatively, and an attached thread cannot only be preempted by other attached threads because of explicit programmer action. A detached thread, on the other hand, is associated to a native operating-system thread, and is scheduled by the operating system asynchronously with respect to all other threads.

### 1.1 Structure of a CPC program

Just like a plain C program, a CPC program is a set of functions. Functions in a CPC program are partitioned into “cps” functions and “native” functions; a global constraint is that a cps function can only ever be called by another cps function, never by a native function. The precise set of contexts where a cps function can be called is defined in Sec. 1.3.

Intuitively, cps code is “interruptible”: when in the attached state, it is possible to interrupt the flow of a block of cps code in order to pass control to another piece of code or to wait for an event to happen. Native code, on the other hand, is “atomic”: if a sequence of native code is executed in attached state, it must be completed before anything else is allowed to run.

Technically, native function calls are executed by using the machine's native stack. Cps function calls, on the other hand, are executed by using a lightweight stack-like structure known as a continuation. This arrangement makes CPC context switches extremely fast; the tradeoff is that a cps function call is an order of magnitude slower than a native call. Thus, computationally expensive code should be implemented in native code whenever possible.

Execution of a CPC program starts at a native function called `main`. This

function usually starts by registering a number of continuations with the CPC runtime (using `cpc_spawn`, Section 1.8), and then passes control to the CPC runtime (by calling `cpc_main_loop`, Section 1.5).

## 1.2 Reserved words

CPC is a conservative extension of the 1999 edition of the C programming language; thus, the syntax of CPC is defined as a set of productions to be added to the grammar defined in the ISO C99 standard [ISO99].

In addition to the reserved words in C99, CPC reserves the words `cps`, `cpc_spawn` and `cpc_attached`.

## 1.3 CPS contexts

Any instruction, declaration, or function definition in CPC can be in *cps context* or in *native context*. Cps context is defined as follows:

- the body of a `cps` function is in cps context (Sec. 1.4);
- the body of a `cpc_spawn` statement is in cps context (Sec. 1.8);
- the body of a `cpc_attached` statement is in cps context (Sec. 1.7).

Any construct that is not in cps context is said to be in native context.

## 1.4 CPS functions

function-specifier ::= `cps`

Functions can be declared as being CPS-converted by adding `cps` to the list of functions specifiers. The effect of such a declaration is to put the body of the function in cps context, thus making it possible to use most of the CPC features.

block-item ::= function-definition

Functions can be defined within other functions, as in Algol-family languages; the inner function can access the variables bound by the outer one. Only cps functions can be inner functions.

Free variables of inner functions are copies of the variables of the enclosing function; thus, a change to the value of the free variable is not visible in the enclosing function<sup>1</sup>. The free variables are initialized whenever the inner function is called; thus, their initial value does not depend on the location of the inner function within the outer one.

Moreover:

- array cannot appear as free variables,
- variable-length arrays must not be used in cps context.

---

<sup>1</sup>Except if the variable is declared `static`.

## 1.5 Bootstrapping

```
void cpc_main_loop(void);
```

`cpc_main_loop` Since `main` is a native function, some means is necessary to pass control to cps code. The function `cpc_main_loop` invokes the CPC scheduler; it returns when all threads have been exhausted (i.e. where there is nothing more to do).

## 1.6 CPC statements and primitive cps functions

CPC provides a set of primitive cps functions, allowing the programmer to schedule threads and wait for some events. It is important to understand that those primitive functions could not be defined outside the CPC runtime: they must have access to the internals of the scheduler to operate. Since they are cps functions, they are valid only in cps context.

In addition to these functions, CPC extends the C language with two statements – `cpc_spawn` and `cpc_attached` – and some syntactic sugar (e.g. `cpc-detached`), described in the following sections.

```
statement ::= cpc-statement
```

## 1.7 Detaching and reattaching

A continuation can be scheduled to be run by a native thread; intuitively, the continuation “becomes” a native thread. When this happens, we say that the continuation has been *detached* from the CPC scheduler, and *attached* to some thread pool. The opposite operation is known as *attaching* a detached continuation back to the CPC scheduler. It is of course possible to migrate a detached continuation directly from one thread pool to another.

```
typedef cpc_sched;  
cpc_sched *cpc_default_scheduler, *cpc_default_threadpool;  
  
cpc_sched *cpc_threadpool_get(int);  
int cpc_threadpool_release(cpc_sched *);  
  
cps cpc_sched *cpc_attach(cpc_sched *pool);  
cpc_sched *cpc_get_sched();
```

`cpc_attach` The `cpc_attach` statement attaches the current continuation to the given scheduler. If the scheduler is a thread pool, the following statements are executed in a dedicated native thread, within the thread pool. If the scheduler is the special `cpc_default_scheduler`, the current continuation is scheduled by the CPC scheduler. This function returns a pointer to the scheduler you are coming from.

**Warning:** currently, `cpc_default_scheduler` is a NULL pointer, but this is *not* guaranteed to hold in the future. Always use the `cpc_default_scheduler` variable.

`cpc_threadpool_get` The function `cpc_threadpool_get` returns a pointer to a new thread pool. It takes one argument: the maximum number of threads in the pool (threads are created dynamically). This argument is cropped to `MAX_THREADS`<sup>2</sup> if it is outside the `[1; MAX_THREADS]` interval. This function shall not be called outside the main loop (it is not thread-safe).

`cpc_threadpool_release` Conversely, `cpc_threadpool_release` releases a given thread pool. This function does not block. It returns 0 if the pool has been released, -1 otherwise (most notably when there are some threads still detached). In that case, you shall retry later. This function can be called in any mode and context, but notice that it will always fail if you try to release the pool you are attached to. Conversely, it should always succeed when `cpc_main_loop` has returned.

`cpc_default_threadpool` A pointer to the default thread pool, initialised during the call to `cpc_main_loop`.

`cpc_get_sched` The function `cpc_get_sched` returns the current scheduler. Although this is not a cps function, it should be treated as such: this function is valid in cps context only, and no pointer to it should be taken.

## Syntactic sugar

`cpc_attached`

`cpc-statement ::= cpc_attached (expression) statement`

The body of a `cpc_attached` statement is run attached to the scheduler given as its argument: an implicit `s = cpc_attach(expression)` is executed upon entering the body, and a `cpc_attach(s)` is executed when the end of the statement is reached, and before any `return` statement inside the body. Some constructs are forbidden in the body of a `cpc_attached` statement because they disrupt the flow of control: `goto`, `break`, `continue` and labels.

*Note:* it would be possible to allow the forbidden constructs, with a smart detection of when one jumps in or out of the body; CPC used to do this in the past. But I (Gabriel) believe this is “too much of magic” with a potentially unhealthy semantics. Therefore, the programmer has to take care of such tricky cases by himself.

`cpc_is_detached`, `cpc_detach`, `cpc_detached` Three macros to test if the current thread is detached, and attach to the default thread pool if not already detached.

```
#define cpc_is_detached() \
    (cpc_get_sched() != cpc_default_sched)
#define cpc_detach() \
    cpc_attach(cpc_is_detached () ? \
    cpc_get_sched() : cpc_default_threadpool)
#define cpc_detached \
```

---

<sup>2</sup>`MAX_THREADS` is an internal constant of the CPC runtime, currently set to 20.

```
cpc_attached(cpc_is_detached () ? \  
cpc_get_sched() : cpc_default_threadpool)
```

## 1.8 Cooperating: yielding, spawning

```
cpc-statement ::= cpc_spawn statement
```

```
cps void cpc_yield();  
cps void cpc_done();
```

**cpc\_yield** The function `cpc_yield` causes the current continuation to be suspended, and placed at the end of the queue of runnable continuations. Control is passed back to the CPC main loop. This statement has no effect in detached state.

**cpc\_done** The function `cpc_done` causes the current continuation to be discarded, and control to be passed back to the main CPC loop.

**cpc\_spawn** The `cpc_spawn` statement creates a new attached thread that executes the argument to `cpc_spawn` and places it at the end of the queue of runnable continuations. If this argument contains free variables, they are handled like free variables in CPS functions (Section 1.4). Execution then proceeds after the `cpc_spawn` statement (control is *not* passed back to the main CPC loop). This statement is valid in arbitrary context.

## 1.9 Synchronisation: condition variables

```
typedef struct cpc_condvar cpc_condvar;  
  
cpc_condvar *cpc_condvar_get(void);  
cpc_condvar *cpc_condvar_retain(cpc_condvar*);  
void cpc_condvar_release(cpc_condvar*);  
int cpc_condvar_count(cpc_condvar*);  
  
cps int cpc_wait(cpc_condvar *cond);  
void cpc_signal(cpc_condvar *);  
void cpc_signal_all(cpc_condvar *);
```

**cpc\_wait** The function `cpc_wait` places the current continuation on the list of continuations waiting on the condition variable passed as argument to `cpc_wait`. Control is passed back to the CPC loop. This function accepts two additional optional arguments, specifying a timeout in seconds and microseconds. See `cpc_sleep` below for more details. This function returns `CPC_CONDVAR` or `CPC_TIMEOUT`, depending on the event which woke it up. This function is only valid in attached state.

`cpc_signal` The function `cpc_signal` causes the first of the continuations waiting on the condition variable passed as argument to be moved to the tail of the queue of runnable continuations. Execution proceeds at the instruction following the call to `cpc_signal`. This statement is valid in arbitrary context, but only in attached state.

`cpc_signal_all` The function `cpc_signal_all` causes all of the continuations waiting on the condition variable passed as argument to be moved to the tail of the queue of runnable continuations. This function guarantees that the continuations will be run in the order in which they were suspended. This statement is valid in arbitrary context, but only in attached state.

## 1.10 Sleeping

```
cps int cpc_sleep(int sec, int usec, cpc_condvar *cond);
int cpc_gettimeofday(stuct timeval *tv);
```

`cpc_sleep` The function `cpc_sleep` takes three arguments: a time in seconds, a time in microseconds, and a condition variable. It causes the current thread to be suspended until either the specified amount of time has passed, or the condition variable is signalled, whichever happens first. It returns, respectively, `CPC_TIMEOUT` or `CPC_CONDVAR`.

The third argument can be omitted if no interruption is necessary. The second argument can be omitted if sub-second accuracy is not needed.

The form with a third argument is only valid in attached state.

The following calls:

```
cpc_wait(cond, sec);
cpc_wait(cond, sec, usec);
```

are syntactic sugar for, respectively,

```
cpc_sleep(sec, 0, cond);
cpc_sleep(sec, usec, cond);
```

`cpc_gettimeofday` The function `cpc_gettimeofday` is a non-blocking equivalent of `gettimeofday`. In detached mode it simply calls it. In attached mode, it returns the latest time measured by the scheduler. (The timezone is set to `NULL`.) This function is only valid in `cps` context.

`cpc_time` The function `cpc_time` is a non-blocking equivalent of `time`. In detached mode it simply calls it. In attached mode, it returns the latest time measured by the scheduler. This function is only valid in `cps` context.

## 1.11 Waiting for I/O

```
cps int cpc_io_wait(int fd, int direction, cpc_condvar *cond);
void cpc_signal_fd(int fd, int direction);
```

`cpc_io_wait` The function `cpc_io_wait` takes three arguments: a file descriptor, a direction, and a condition variable. The direction can be one of `CPC_IO_IN`, meaning input, or `CPC_IO_OUT`, meaning output.

This function causes the current continuation to be suspended until either the given file descriptor is available for I/O in the given direction, or the given condition variable is signalled, whichever happens first. It returns `CPC_IO_IN`, `CPC_IO_OUT` or `CPC_CONDVAR`, depending on what happened. It may also return `-1` if some error occurred (e.g. in detached mode, when the call to `poll` fails).

The form with a third argument is only valid in attached state.

`cpc_signal_fd` The function `cpc_signal_fd` wakes up the continuations waiting for an event `direction` on the `fd` file descriptor. It shall only be called in the same thread as the CPC scheduler (but not necessarily in `cps` context) and will only wake up attached threads.

## 1.12 Limitations and implementation notes

Not all legal C code is allowable in CPC. Some of the limitations described below are fundamental to the implementation technique of CPC; others are just artefacts of the current implementation, and will be lifted in a future version.

### 1.12.1 Fundamental limitations

The use of the `longjmp` library function, and its variants, is not allowed in CPC code. The use of `alloca` in `cps` context yields unpredictable behaviour.

### 1.12.2 Current limitations

The current implementation is based on CIL. The CIL limitations therefore apply. You should read the *CIL Limitations* and *Known Bugs and Limitations* sections of the CIL user manual (see `doc/cil.tex`).

Other current limitations are described in each relevant section of the present document.

### 1.12.3 Time complexity of CPC operations

The current implementation of CPC implements all the CPS operations in constant time, with the following exceptions:

- when a continuation is queued on two structures simultaneously (because of `cpc_sleep` or `cpc_io_wait` with a non-null last argument), invoking it requires dequeuing it from the second queue, which takes linear time in the worst case;
- the `cpc_sleep` instruction runs in worst-case time proportional to logarithm of the number of currently sleeping continuations (thanks to a heap-based sleeping queue).

## Chapter 2

# The CPC library

The functions in the CPC library are themselves written in CPC, using only the primitives documented in Chapter 1.

All the functions in the CPC core library are declared in the file `cpc_lib.h`.

### 2.1 Barriers

```
typedef struct cpc_barrier cpc_barrier;  
  
cpc_barrier *cpc_barrier_get(int count);  
cps void cpc_barrier_await(cpc_barrier *barrier);
```

A barrier is a synchronisation construct that allows a set of continuations to be woken up at the same time. A barrier is conceptually a queue of continuations and a count of continuations remaining to wait for.

`cpc_barrier_get` The function `cpc_barrier_get` returns a new barrier initialised to wait for `count` continuations.

`cpc_barrier_await` The function `cpc_barrier_await` causes the current continuation to wait on the barrier given in argument. This function first decrements the barrier's count; if the count reaches zero, it wakes up all of the continuations waiting on the barrier. Otherwise, it suspends the current continuation.

The function `cpc_barrier_await` guarantees that the continuations are run in the order in which they were suspended.

### 2.2 Input/Output

#### 2.2.1 Setting up file descriptors

```
int cpc_setup_descriptor(int fd, int nonagle);
```

The function `cpc_setup_descriptor` sets up the file descriptor `fd` into non-blocking mode, making it suitable for use by the CPC runtime. If `nonagle` is

true (non-zero), the descriptor is assumed to refer to a socket and has the Nagle algorithm disabled (the socket option `TCP_NODELAY` is set).

This function returns 1 in case of success, -1 in case of failure.

### 2.2.2 Input/Output

```
cps int cpc_write(int fd, void *buf, size_t count);
cps int cpc_write_timeout(int fd, void *buf, size_t count,
                          int secs, int micros);
cps int cpc_read(int fd, void *buf, size_t count);
cps int cpc_read_timeout(int fd, void *buf, size_t count,
                         int secs, int micros);
```

The functions `cpc_write` and `cpc_read` are CPC's versions of the `write` and `read` system calls. They return the number of octets read/written in case of success; in case of failure, they return -1 with `errno` set.

The versions with `timeout` appended return after `secs` seconds and `micros` microseconds if no I/O has been possible. In this case, they return -1 with `errno` set to `EAGAIN`.

### 2.2.3 Timeouts

```
typedef struct cpc_timeout cpc_timeout;

cps cpc_timeout *cpc_timeout_get(int secs, int usecs);
cps void cpc_timeout_restart(cpc_timeout *timeout);
cpc_condvar *cpc_timeout_condvar(cpc_timeout *timeout);
int cpc_timeout_expired(cpc_timeout *timeout);
void cpc_timeout_destroy(cpc_timeout *timeout);
```

Timeouts are tiny data structures holding a condition variable, which is signaled after some time (`cpc_timeout_get`). A timeout might be reset (`cpc_timeout_restart`) or destroyed (`cpc_timeout_destroy`). To avoid race conditions, you cannot reset a timeout which has already expired: destroy it and create a new one instead.

You can get the condition variable associated with the timeout using `cpc_timeout_condvar`, and check if the timeout has already expired with `cpc_timeout_expired`. Beware race conditions: it is important to check if the timeout has already expired before waiting on the condition variable, since the latter is only signaled once.

## Chapter 3

# Tips and tricks

### Using ctags

```
ctags -R --langmap=c:+.cpc *
```

**Vim setup** Put the following in your `.vimrc`:

```
" CPC
au BufNewFile,BufRead *.cpc setf c
au BufNewFile,BufRead *.cpc syn keyword cType cps
au BufNewFile,BufRead *.cpc syn keyword cStatement cpc_spawn cpc_detached cpc_attached
```

**Emacs setup** Put the following in your `.emacs`:

```
;;; CPC
(push '("\.cpc$" . c-mode) auto-mode-alist)
```

# Bibliography

- [ISO99] Information technology — programming language C. International standard ISO/IEC 9899:1999, 1999.
- [Plo75] G. D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theoretical Computer Science*, 1:125–159, 1975. Also published as Memorandum SAI–RM–6, School of Artificial Intelligence, University of Edinburgh, Edinburgh, 1973.
- [SW74] Christopher Strachey and Christopher P. Wadsworth. Continuations: A mathematical semantics for handling full jumps. Technical Monograph PRG–11, Oxford University Computing Laboratory, Programming Research Group, Oxford, England, 1974.