HTTrack Programming page - plugging functions

You can write external functions to be plugged in the httrack library very easily. We'll see there some examples.

The `httrack` commandline tool allows (since the 3.30 release) to plug external functions to various callbacks defined in httrack. The 3.41 release introduces a cleaned up ver
callbacks, with two major changes:

- Cleaned up function prototypes, with two arguments always passed (the caller carg structure, and the httrackp* object), convenient to pass an user-defined pointer (see `CALLBACKARG_USERDEF(carg)`)
- The httrackp* option structure can be directly accessed to plug callbacks (no need to give the callback name and function name in the commandline!)
- The callback plug is made through the CHAIN_FUNCTION() helper, allowing to chain multiple callbacks of the same type (the callbacks MUST preserve the chain by calling

References:

- the `httrack-library.h` prototype file
  Note: the *Initialization*, *Main functions*, *Options handling* and *Wrapper functions* sections are generally the only ones to be considered.
- the `htsdefines.h` prototype file, which describes callback function prototypes
- the `htsopt.h` prototype file, which describes the full httrackp* structure
- the `callbacks-example*.c` files given in the httrack archive
- the `htsjava.c` source file (the java class plugin ; overrides 'detect' and 'parse')
- the example given at the end of this document

Below the list of functions to be defined in the module (plugin).

module function name	**function description**	**function signature**

hts_plug	The module entry point. The opt structure can be used to plug callbacks, using the CHAIN_FUNCTION() macro helper. The argv optional argument is the one passed in the commandline as --wrapper parameter.
	return value: 1 upon success, 0 upon error (the mirror will then be aborted)
	
	Wrappers can be plugged inside hts_plug() using:
	`CHAIN_FUNCTION(opt, <callback name>, <our callback function name>, <our callback function optional custom pointer argument>);`
	
	Example:
	`CHAIN_FUNCTION(opt, check_html, process, userdef);`
		`extern int hts_plug(htt *opt, const char* argv)`

hts_unplug	The module exit point. To free allocated resources without using global variables, use the uninit callback (see below)	`extern int hts_unplug(h *opt);`

Note that all callbacks (except init and uninit) take as first two argument:

- the t_hts_callbackarg structure
  this structure holds the callback chain (parent callbacks defined before the current callback) pointers, and the user-defined pointer ; see `CALLBACKARG_USERDEF(car`
- the httrackp structure
  this structure, holding all current httrack options and mirror state, can be read or mofidied

Below the list of callbacks, and associated external wrappers.

callback name	**callback description**	**callback function signature**
init	Note: the use the "start" callback is advised. Called during initialization. return value: none	`void mycallback(t_hts_callba *carg);`
uninit	Note: the use os the "end" callback is advised. Called during un-initialization return value: none	`void mycallback(t_hts_callba *carg);`
start	Called when the mirror starts. The `opt` structure passed lists all options defined for this mirror. You may modify the `opt` structure to fit your needs. return value: 1 upon success, 0 upon error (the mirror will then be aborted)	`int mycallback(t_hts_ca *carg, httrackp* opt);`
end	Called when the mirror ends return value: 1 upon success, 0 upon error (the mirror will then be considered aborted)	`int mycallback(t_hts_ca *carg, httrackp* opt);`
chopt	Called when options are to be changed. The `opt` structure passed lists all options, updated to take account of recent changes return value: 1 upon success, 0 upon error (the mirror will then be aborted)	`int mycallback(t_hts_ca *carg, httrackp* opt);`
preprocess	Called when a document (which is an html document) is to be parsed (original, not yet modified document). The `html` address points to the document data address (char**), and the `length` address points to the lenth of this document. Both pointer values (address and size) can be modified to change the document. It is up to the callback function to reallocate the given pointer (using the hts_realloc()/hts_free() library functions), which will be free()'ed by the engine. Hence, return of static buffers is strictly forbidden, and the use of hts_strdup() in such cases is advised. The `url_address` and `url_file` are the address and URI of the file being processed return value: 1 if the new pointers can be applied (default value)	`int mycallback(t_hts_ca *carg, httrackp* opt, c html, int* len, const c url_address, const char url_file);`
postprocess	Called when a document (which is an html document) is parsed and transformed (links rewritten). The `html` address points to the document data address (char**), and the `length` address points to the lenth of this document. Both pointer values (address and size) can be modified to change the document. It is up to the callback function to reallocate the given pointer (using the hts_realloc()/hts_free() library functions), which will be free()'ed by the engine. Hence, return of static buffers is strictly	`int mycallback(t_hts_ca *carg, httrackp* opt, c html, int* len, const c`

	forbidden, and the use of hts_strdup() in such cases is advised. The `url_address` and `url_file` are the address and URI of the file being processed
	return value: 1 if the new pointers can be applied (default value)
		`url_address, const char`
		`url_file);`

check_html	Called when a document (which may not be an html document) is to be parsed. The `html` address points to the document data, of lenth `len`. The `url_address` and `url_file` are the address and URI of the file being processed
	return value: 1 if the parsing can be processed, 0 if the file must be skipped without being parsed
		`int mycallback(t_hts_ca`
		`*carg, httrackp* opt, c`
		`html, int len, const ch`
		`url_address, const char`
		`url_file);`

query	Called when the wizard needs to ask a question. The `question` string contains the question for the (human) user
	return value: the string answer ("" for default reply)
		`const char*`
		`mycallback(t_hts_callba`
		`*carg, httrackp* opt, c`
		`char* question);`

query2	Called when the wizard needs to ask a question	`const char*`
		`mycallback(t_hts_callba`
		`*carg, httrackp* opt, c`
		`char* question);`

query3	Called when the wizard needs to ask a question	`const char*`
		`mycallback(t_hts_callba`
		`*carg, httrackp* opt, c`
		`char* question);`

loop	Called periodically (informational, to display statistics)
	return value: 1 if the mirror can continue, 0 if the mirror must be aborted
		`int mycallback(t_hts_ca`
		`*carg, httrackp* opt, l`
		`back, int back_max, int`
		`back_index, int lien_to`
		`lien_ntot, int stat_tim`
		`hts_stat_struct* stats)`

check_link	Called when a link has to be tested. The `adr` and `fil` are the address and URI of the link being tested. The passed `status` value has the following meaning: 0 if the link is to be accepted by default, 1 if the link is to be refused by default, and -1 if no decision has yet been taken by the engine
	return value: same meaning as the passed `status` value ; you may generally return -1 to let the engine take the decision by itself
		`int mycallback(t_hts_ca`
		`*carg, httrackp* opt, c`
		`char* adr, const char*`
		`status);`

check_mime	Called when a link download has begun, and needs to be tested against its MIME type. The `adr` and `fil` are the address and URI of the link being tested, and the `mime` string contains the link type being processed. The passed `status` value has the following meaning: 0 if the link is to be accepted by default, 1 if the link is to be refused by default, and -1 if no decision has yet been taken by the engine
	return value: same meaning as the passed `status` value ; you may generally return -1 to let the engine take the decision by itself
		`int mycallback(t_hts_ca`
		`*carg, httrackp* opt, c`
		`char* adr, const char*`
		`const char* mime, int s`

pause	Called when the engine must pause. When the `lockfile` passed is deleted, the function can return
	return value: none
		`void`
		`mycallback(t_hts_callba`
		`*carg, httrackp* opt, c`
		`char* lockfile);`

filesave	Called when a file is to be saved on disk
	return value: none
		`void`
		`mycallback(t_hts_callba`
		`*carg, httrackp* opt, c`
		`char* file);`

filesave2	Called when a file is to be saved or checked on disk
	The hostname, filename and local filename are given. Two additional flags tells if the local file is new (is_new), if the local file is to be modified (is_modified), and if the file was not updated remotely (not_updated).
	(!is_new && !is_modified): the file is up-to-date, and will not be modified
	(is_new && is_modified): a new file will be written (or an updated file is being written)
	(!is_new && is_modified): a file is being updated (append)
	(is_new && !is_modified): an empty file will be written ("do not recatch locally erased files")
	not_updated: the file was not re-downloaded because it was up-to-date (no data transfered again)
	
	return value: none
		`void`
		`mycallback(t_hts_callba`
		`*carg, httrackp* opt, c`
		`char* hostname, const c`
		`filename, const char* l`
		`int is_new, int is_modi`
		`not_updated);`

linkdetected	Called when a link has been detected
	return value: 1 if the link can be analyzed, 0 if the link must not even be considered
		`int mycallback(t_hts_ca`
		`*carg, httrackp* opt, c`
		`link);`

linkdetected2	Called when a link has been detected
	return value: 1 if the link can be analyzed, 0 if the link must not even be considered
		`int mycallback(t_hts_ca`
		`*carg, httrackp* opt, c`
		`link, const const char*`
		`tag_start);`

xfrstatus	Called when a file has been processed (downloaded, updated, or error)
	return value: must return 1
		`int mycallback(t_hts_ca`
		`*carg, httrackp* opt, l`
		`back);`

savename	Called when a local filename has to be processed. The `adr_complete` and `fil_complete` are the address and URI of the file being saved ; the `referer_adr` and `referer_fil` are the address and URI of the referer link. The `save` string contains the local filename being used. You may modifiy the `save` string to fit your needs, up to 1024 bytes (note: filename collisions, if any, will be handled by the engine by renaming the file into file-2.ext, file-3.ext ..).
	return value: must return 1
		`int mycallback(t_hts_ca`
		`*carg, httrackp* opt, c`
		`char* adr_complete, cons`
		`fil_complete, const cha`
		`referer_adr, const char`
		`referer_fil, char* save`

sendhead	Called when HTTP headers are to be sent to the remote server. The `buff` buffer contains text headers, `adr` and `fil` the URL, and `referer_adr` and `referer_fil` the referer URL. The `outgoing` structure contains all information related to the current slot.
	return value: 1 if the mirror can continue, 0 if the mirror must be aborted
		`int mycallback(t_hts_ca`
		`*carg, httrackp* opt, c`
		`buff, const char* adr,`
		`char* fil, const char*`
		`referer_adr, const char`
		`referer_fil, htsblk* ou`

receivehead	Called when HTTP headers are recevived from the remote server. The `buff` buffer contains text headers, `adr` and `fil` the URL, and `referer_adr` and `referer_fil` the referer URL. The `incoming` structure contains all information related to the current slot.
	return value: 1 if the mirror can continue, 0 if the mirror must be aborted
		`int mycallback(t_hts_ca`
		`*carg, httrackp* opt, c`
		`buff, const char* adr,`
		`char* fil, const char*`
		`referer_adr, const char`
		`referer_fil, htsblk* in`

detect	Called when an unknown document is to be parsed. The `str` structure contains all information related to the document.
	return value: 1 if the type is known and can be parsed, 0 if the document type is unknown
		`int mycallback(t_hts_ca`
		`*carg, httrackp* opt,`
		`htsmoduleStruct* str);`

parse	The `str` structure contains all information related to the document.
	return value: 1 if the document was successfully parsed, 0 if an error occured
		`int mycallback(t_hts_ca`
		`*carg, httrackp* opt,`
		`htsmoduleStruct* str);`

Note: the optional libhttrack-plugin module (libhttrack-plugin.dll or libhttrack-plugin.so), if found in the library environment, is loaded automatically, and its `hts_plug()` fun
called.

An example is generally more efficient than anything else, so let's write our first module, aimed to stupidely print all parsed html files:

```
/* system includes */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* standard httrack module includes */
#include "httrack-library.h"
#include "htsopt.h"
#include "htsdefines.h"

/* local function called as "check_html" callback */
static int process_file(t_hts_callbackarg /*the carg structure, holding various information*/*carg, /*the option settings*/httrackp
                        /*other parameters are callback-specific*/
                        char* html, int len, const char* url_address, const char* url_file) {
  void *ourDummyArg = (void*) CALLBACKARG_USERDEF(carg);    /*optional user-defined arg*/

  /* call parent functions if multiple callbacks are chained. you can skip this part, if you don't want previous callbacks to be ca
  if (CALLBACKARG_PREV_FUN(carg, check_html) != NULL) {
    if (!CALLBACKARG_PREV_FUN(carg, check_html)(CALLBACKARG_PREV_CARG(carg), opt,
                                                html, len, url_address, url_file)) {
        return 0;  /* abort */
      }
  }

  printf("file %s%s content: %s\n", url_address, url_file, html);
  return 1;  /* success */
}

/* local function called as "end" callback */
static int end_of_mirror(t_hts_callbackarg /*the carg structure, holding various information*/*carg, /*the option settings*/httrack
  void *ourDummyArg = (void*) CALLBACKARG_USERDEF(carg);    /*optional user-defined arg*/

  /* processing */
  fprintf(stderr, "That's all, folks!\n");

  /* call parent functions if multiple callbacks are chained. you can skip this part, if you don't want previous callbacks to be ca
  if (CALLBACKARG_PREV_FUN(carg, end) != NULL) {
    /* status is ok on our side, return other callabck's status */
    return CALLBACKARG_PREV_FUN(carg, end)(CALLBACKARG_PREV_CARG(carg), opt);
  }

  return 1;  /* success */
}

/*
module entry point
the function name and prototype MUST match this prototype
*/
EXTERNAL_FUNCTION int hts_plug(httrackp *opt, const char* argv) {
  /* optional argument passed in the commandline we won't be using here */
  const char *arg = strchr(argv, ',');
  if (arg != NULL)
    arg++;

  /* plug callback functions */
  CHAIN_FUNCTION(opt, check_html, process_file, /*optional user-defined arg*/NULL);
  CHAIN_FUNCTION(opt, end, end_of_mirror, /*optional user-defined arg*/NULL);

  return 1;  /* success */
}

/*
module exit point
the function name and prototype MUST match this prototype
*/
EXTERNAL_FUNCTION int hts_unplug(httrackp *opt) {
  fprintf(stder, "Module unplugged");

  return 1;  /* success */
}
```

Compile this file ; for example:

```
gcc -O -g3 -shared -o mylibrary.so myexample.c
```

and plug the module using the commandline ; for example:

```
httrack --wrapper mylibrary http://www.example.com
```

or, if some parameters are desired:

```
httrack --wrapper mylibrary,myparameter-string http://www.example.com
```

(the "myparameter-string" string will be available in the 'arg' parameter passed to the hts_plug entry point)