# C interpreter linking and relocation

*Mark Russell*

University of Kent

## 1. Introduction

This document describes the linking and relocation scheme used by the ups C interpreter. Read it if you have to maintain or extend the code.

We first give a brief overview of the system, and then a detailed description of each step of the compilation, linking, loading and execution process.

You will need a copy of the ups source to hand as you read this document.

## 2. Overview

C interpreter linking and relocation is more complex than usual because we do not build a monolithic linked output file. Instead the output file contains references to the .ocx files that were used in the link. The .ocx files are read on demand at run time. A .ocx file is read when a function defined in it is called. Some .ocx files may never be loaded on a given run.

Data that can only be referenced by functions in a .ocx file is stored there. This is to reduce the size of the linked cx file. Relocation on this data is done at .ocx load time; the relocation information is also stored in the .ocx file. In this description we call this data 'private'.

Data that can potentially be referenced by functions outside the .ocx file is stored in the linked cx file. It also appears in the .ocx file, but only so that the linker can copy in to the cx file. We call this data 'public'.

You might think that variables declared with the C keyword 'static' would always be private, but this is not so. If the address of such a variable is used to initialise a non-static variable then the static variable could be referenced via the non-static one (and not by name). This effect can propogate - consider a statically initiailised linked list with all but the first element declared static.

As there is no monolithic text area there is more to a function call than jumping to a text offset. Instead a function address consists of an index into a table of files and an offset into the per-file text area. There is a fair amount of book-keeping involved in dealing with external function calls.

## 3. Phases of compilation

There are three phases between source text and an executing program:

Compilation
: This consists of parsing the source (.c) file and generating code. The output is written to an object file with the suffix .ocx. This contains the generated text and initialised data, along with relocation, linking and optionally debugging information.

Linking
: In this phase a specified list of .ocx files is linked to form a C interpreter 'binary'. This involves resolving external function and data references. The .ocx file list can include libraries, which are simply multiple .ocx files concatenated together.

  The result of a successful link appears to the user like an executable, but in fact it uses the '#!' mechanism to run the loader binary. The file contains a list of the .ocx file pathnames, and the relocation information needed to correctly load them as necessary.

Loading
: When the user invokes a C interpreter binary the loader is invoked via the kernel '#!' mechanism. The loader driver allocates some space for the stack, then reads the cx.out file. It then jumps to the

entry point (specified in the header). The text for the initial function is stored in the cx.out file; it sets up argv and calls main.

The rest of this document describes this process in detail.

### 4. The `text_t` structure

`compile`() takes a pointer to a parse tree for a single .c file and generates code, data and relocation information. All this information is stored in the text descriptor `text_t`, defined in `ci_compile.h`. The fields in a `text_t` include the following:

`textword_t *tx_text`
: A pointer to the start of the generated code. The code may contain unresolved references to external functions and data, as well as unrelocated references to data.

`reloc_t *tx_var_relocs`
: A list of relocations needed because of references to variables. Added to by `ci_add_var_reloc`() and processed by `make_relocs`().

`reloc_t *tx_string_relocs`
: Relocations needed because of references to string literals. Added to by `ci_add_string_reloc`() and processed by `make_relocs`().

`funcreloc_t *tx_func_relocs`
: Relocations needed for pointers to functions. These are needed because a function pointer consists of a 16 bit file index and a 16 bit function number, and the file indices are not known until link time.

`var_t *tx_varlist`
: A list of file-scope variables generated by the parse. This includes entries generated by `extern` declarations.

`vlist_t *tx_vlist`
: A list of non-automatic variables that have been referenced. Unlink `tx_varlist`, this list includes local static variables and excludes variables declared `extern` that are never referenced.

### 5. Compilation steps

`ci_compile_for_linking`() takes a handle on a parse tree (a `parse_id_t`) and returns a `linkinfo_t` describing the text, data and relocation and symbol table information needed to write a .ocx file..

The compilation steps performed by `ci_compile_for_linking`() are:

- Compile the parse tree, by calling `compile`(). This produces the text area (pointed to by `tx_text`) and a list of relocations in the text area (`tx_var_relocs`, `tx_string_relocs`, and `tx_func_relocs`). This also adds local static variables to `tx_vlist`, and adds any static initialisations to `tx_initlist`.
- Set the `VA_EXT_RELOC` (external relocation needed) flag for any variables referenced in initialisers for variables with external linkage (this effect can propogate).
- Assign addresses for all non-automatic variables defined (not just declared) in this file. A variable can have internal or external linkage (`VA_EXT_RELOC`) and go in either data or bss (`VA_HAS_INITIALISER`) giving four possible locations. As part of the address assignment we calculate the sizes of the four areas.
- Add space to the internal and external data areas for string initialisers and strings referenced from text.
- Allocate and zero space for the internal and external data areas, then perform the static initialisations listed in `tx_initlist`. This can add variable, string and function relocations in the internal or external data area.
- Allocate the `linkinfo_t` structure that is the output of `ci_compile_for_linking`(). Set the `linkinfo_t` pointers and sizes for the text and internal and external data areas, and the sizes for the internal and external bss areas.

- Call make_extvars() to build the lists of defined and undefined variables, and adjust the addresses of internal static bss variables (adding li_data_size as internal bss follows internal data).

  Undefined variables go on the li_undef_vars list, with ln_addr set to zero. va_addr is set to the index into the list (this only becomes correct when the list is reversed later).

  Defined variables go on the li_vars list, with a dummy name if the variable is really static but is referenced by an external variable. ln_addr is set to the value of va_addr, with the addresses of bss variables made relative to the *end* of external bss rather than the start. Bss addresses are thus always negative - this is used by cvtwords() to distinguish bss from data addresses. Then va_addr is set to the index into the li_vars list, offset by the length of the li_undef_vars list.

- Call make_relocs() to construct the linkinfo_t relocation lists from the text_t lists tx_var_relocs, tx_string_relocs and tx_func_relocs.

  make_relocs builds five output lists. Note that a reloc_pos_t contains just a location, whereas an ext_reloc_t contains a location and an index.

  reloc_pos_t *li_extdata_funcrelocs
  : References to function pointers (to functions in this file) in external data. Only the location (relative to the start of external data) is needed, as the function index is already stored in the data location. These are resolved by cvtwords() before any .ocx files are loaded.

  ext_reloc_t *li_extdata_relocs
  : References to external variables in external data. These are also resolved by cvtwords.

  ext_reloc_t *li_ext_relocs
  : References to external variables in text or internal data. The offset (if any) is stored at the location itself, and the relocation entry contains the index of the variable (as placed in va_addr by make_extvars()). These are resolved when a .ocx file is loaded by load_text().

  reloc_pos_t *li_static_rps
  : References to internal variables and strings in text or internal data. Also resolved by load_text().

  reloc_pos_t *li_static_funcrelocs
  : References to function pointers (to functions in this file) in text or internal data. Resolved by load_text().

  We do some relocation resolution in make_relocs(), as we know the size of the text and data areas as well as the addresses of internal variables. This means, for example that we can store internal relocation as simply an offset from the start of the text area, with no need to store which variable is referred to. We do not need to record whether an internal relocation is in the text or data area - we just add the size of the text area to the location for relocations in internal data.

  make_relocs() also copies string initialisers into the internal or external data areas.

- Call make_funcinfo() to build the table of function addresses and the list of external functions. The function address table contains offsets from the start of the text area, and is ultimately used by the virtual machine to do a function call (function pointers contain an index into this table).

  The list of external functions is used later at link time to resolve cross-file function calls.

- Call make_symfuncs() to build symbol table information describing functions. This consists of the name, text address and block pointer of each function, as well a table for each function mapping text addresses to source file line numbers. The block pointer points to the local variable and type information as produced by the parse.

- Call make_symvars() to build a table of global variable names defined in this file. This table is (currently) only used to quickly check if a name is defined in a given source file before loading it to resolve a symbol.

## 6. .ocx files

The output of `ci_compile_for_linking`() is a `linkinfo_t`() structure. This is normally written to a .ocx file, which is similar in function to a compile .o file. The structure of a .ocx file is described by the `cx_header_t` structure (defined in `xc_load.h`). The .ocx file is just a file representation of the information in a `linkinfo_t`().

.ocx files exist to support separate compilation. They are produced when multiple .c files are compiled. In subsequent compilations the .ocx file can be used instread of the .c file. In this case the `linkinfo_t` structure is read from the file (by `ci_read_cx_file`()) rather than built from source code by `ci_compile_file`().

`ci_read_cx_file`() will read libraries as well as single .ocx files. A library simply consists of multiple concatenated .ocx files (and indeed is built with *cat*(1)). Libraries are handled just like individual .ocx files, except that a .ocx file within a library is only loaded if it satisfies an existing unresolved external variable or function reference.

## 7. Linking

Unless the user directs otherwise, the .c and .ocx files named on the command line are linked into a single executable output file. There are three steps:

- `ci_make_link_id`() builds an `olinkinfo_t`. This holds global link information such as the list of .ocx files, and a list of defined and undefined external variables.

  `ci_make_link_id`() inserts a reference to `main`, as well as definitions for the compiler builtins `exit`(), `__cx_setjmp`() and `__cx_longjmp`().
- Each `linkinfo_t` (whether produced from a .c file, a .ocx file or a member of a library) is passed to `link_file`(), which updates the global `olinkinfo_t` link information with the link information in the `linkinfo_t`.
- Finally, `ci_link`() checks for unresolved or duplicate external names, and if all is well writes the executable output file.

### 7.1. Processing `linkinfo_t` structures

`link_file`() takes a `linkinfo_t` and a `cx_header_t` and builds a `fileinfo_t` from it which is pushed onto the `ol_files` list. The `cx_header_t` is a copy of the structure at the front of a .ocx file; it contains data copied from the `linkinfo_t` structure.

`link_file`() also updates the `ol_extnames` list of defined and undefined external variables and functions. `link_file`() makes deep copies of everything from the `linkinfo_t` that must be preserved for the final link, because all the data hanging off the `linkinfo_t` is freed shortly after the `link_file`() call.

The only `fileinfo_t` fields which are not copied directly from the `linkinfo_t` and `cx_header_t` structures are `fx_funcrefs`, `fx_varrefs` and `fx_vardefs`. These three fields are arrays of pointers to entries in the `ol_extnames` list which are built as a result of doing name resolution.

First, undefined variables and functions (`li_undef_vars` and `li_undef_funcs`) are looked up. If they not found then they are added to the `ol_extnames` list as undefined. The resulting pointer to an `extname_t` is added to table of undefined functions and variables (`fx_funcrefs` and `fx_varrefs` respectively). There is very rudimentary consistency checking, in that a function may not satisfy a variable reference, and vice versa.

Variables and functions defined in this file (`li_vars` and `li_funcs`) are then added to the `ol_extnames` list by calling `resolve_names`(). This fills in the `fx_vardefs` array of pointers into the `ol_extnames` lists. There is no need for a similar table of functions defined in this file, [but I currently don't know why]. Functions and variables in the `ol_extnames` list are indicated as undefined by having the `en_file` field set to `NULL`; if we find such entries we set `en_file` to point at the current `fileinfo_t`, thus marking the entry as defined. If we find an entry with `en_file` already set we complain about multiple definitions.

When all the files have been processed in this way the only `ol_extnames` entries still undefined should be references to builtin functions and variables (see below).

### 7.2. Writing the executable

Once all the input files have been passwd to `link_file`(), the compiler driver calls `ci_link`() to resolve any undefined symbols and write the output file.

`ci_link`() first scans the `ol_extnames` list for undefined symbols (entries with `en_file` set to `NULL`). Such entries are looked up with the `get_libaddr`() function supplied as an argument to `link_file`(). If this lookup fails, we complain about the undefined symbol and do not write an executable file. Otherwise we set `en_addr` to the address returned by `(*get_libaddr)`() and point `en_file` for the entry at a special `fileinfo_t` (`libfxbuf`) with `fx_index` set to -1.

If all undefined symbols are successfully resolved we go on to write the output file. This consists of the following:

- A header `x_header_t` giving the sizes of the various segments, and some other stuff.
- An array of .ocx file entries (one for each entry in the `ol_files` list).
- An array of all the indices into the external names tables. This consists of the concatenation of the `fx_funcrefs`, `fx_varrefs` and `fx_vardefs` entries for each entry in the `ol_files` list.
- All the external data (the concatenation of all the `fx_extdata` areas).
- All the external data relocations.
- All the external function pointer relocations.
- The `ol_extnames` list. Only the file index and address for each entry is saved; the names are not needed.
- Optional symbol table information: the concatenation of the `fx_symfuncs` tables, followed by the concatenation of the `fx_symvars` tables.
- The strings table. Strings (such as .ocx file pathnames) are stored as offsets into this table.

This file is read back in at execution time by `ci_load`().