ld65 Users Guide

Ullrich von Bassewitz

2014-04-20

The ld65 linker combines object files into an executable file. ld65 is highly configurable and uses configuration files for high flexibility.

---

1. Overview

The ld65 linker combines several object modules created by the ca65 assembler, producing an executable file. The object modules may be read from a library created by the ar65 archiver (this is somewhat faster and more convenient). The linker was designed to be as flexible as possible. It complements the features that are built into the ca65 macroassembler:

- Accept any number of segments to form an executable module.
- Resolve arbitrary expressions stored in the object files.
- In case of errors, use the meta information stored in the object files to produce helpful error messages. In case of undefined symbols, expression range errors, or symbol type mismatches, ld65 is able to tell you the exact location in the original assembler source, where the symbol was referenced.
- Flexible output. The output of ld65 is highly configurable by a config file. Some more-common platforms are supported by default configurations that may be activated by naming the target system. The output generation was designed with different output formats in mind, so adding other formats shouldn't be a great problem.

2. Usage

2.1 Command-line option overview

The linker is called as follows:

```
---------------------------------------------------------------------------
Usage: ld65 [options] module ...
Short options:
  -(                    Start a library group
  -)                    End a library group
  -C name               Use linker config file
  -D sym=val            Define a symbol
  -L path               Specify a library search path
  -Ln name              Create a VICE label file
  -S addr               Set the default start address
  -V                    Print the linker version
  -h                    Help (this text)
  -m name               Create a map file
  -o name               Name the default output file
  -t sys                Set the target system
  -u sym                Force an import of symbol `sym'
  -v                    Verbose mode
  -vm                   Verbose map file

Long options:
  --cfg-path path       Specify a config file search path
  --config name         Use linker config file
  --dbgfile name        Generate debug information
  --define sym=val      Define a symbol
  --end-group           End a library group
  --force-import sym    Force an import of symbol `sym'
  --help                Help (this text)
  --lib file            Link this library
  --lib-path path       Specify a library search path
  --mapfile name        Create a map file
  --module-id id        Specify a module id
  --obj file            Link this object file
  --obj-path path       Specify an object file search path
  --start-addr addr     Set the default start address
  --start-group         Start a library group
  --target sys          Set the target system
  --version             Print the linker version
---------------------------------------------------------------------------
```

2.2 Command-line options in detail

Here is a description of all of the command-line options:

-(, --start-group

Start a library group. The libraries specified within a group are searched multiple times to resolve crossreferences within the libraries. Normally, crossreferences are resolved only within a library, that is the library is searched multiple times. Libraries specified later on the command line cannot reference otherwise unreferenced symbols in libraries specified earlier, because the linker has already handled them. Library groups are a solution for this problem, because the linker will search repeatedly through all libraries specified in the group, until all possible open symbol references have been satisfied.

-), --end-group

End a library group. See the explanation of the `--start-group` option.

-h, --help

Print the short option summary shown above.

-m name, --mapfile name

This option (which needs an argument that will used as a filename for the generated map file) will cause the linker to generate a map file. The map file does contain a detailed overview over the modules used, the sizes for the different segments, and a table containing exported symbols.

-o name

The -o switch is used to give the name of the default output file. Depending on your output configuration, this name *might not* be used as the name for the output file. However, for the default configurations, this name is used for the output file name.

-t sys, --target sys

The argument for the -t switch is the name of the target system. Since this switch will activate a default configuration, it may not be used together with the `-C` option. The following target systems are currently supported:

- none
- module
- apple2
- apple2enh
- atari2600
- atari
- atarixl
- atmos
- c16 (works also for the c116 with memory up to 32K)
- c64
- c128
- cbm510 (CBM-II series with 40-column video)
- cbm610 (all CBM series-II computers with 80-column video)
- geos-apple
- geos-cbm
- lunix
- lynx
- nes
- pet (all CBM PET systems except the 2001)
- plus4
- sim6502
- sim65c02
- supervision

- vic20

There are a few more targets defined but neither of them is actually supported.

-u sym[:addrsize], --force-import sym[:addrsize]

Force an import of a symbol. While object files are always linked to the output file, regardless if there are any references, object modules from libraries get only linked in if an import can be satisfied by this module. The `--force-import` option may be used to add a reference to a symbol and as a result force linkage of the module that exports the identifier.

The name of the symbol may optionally be followed by a colon and an address-size specifier. If no address size is specified, the default address size for the target machine is used.

Please note that the symbol name needs to have the internal representation, meaning you have to prepend an underscore for C identifiers.

-v, --verbose

Using the -v option, you may enable more output that may help you to locate problems. If an undefined symbol is encountered, -v causes the linker to print a detailed list of the references (that is, source file and line) for this symbol.

-vm

Must be used in conjunction with `-m` (generate map file). Normally the map file will not include empty segments and sections, or unreferenced symbols. Using this option, you can force the linker to include all that information into the map file. Also, it will include a second `Exports` list. The first list is sorted by name; the second one is sorted by value.

-C

This gives the name of an output config file to use. See section 4 for more information about config files. -C may not be used together with `-t`.

-D sym=value, --define sym=value

This option allows to define an external symbol on the command line. Value may start with a '$' sign or with `0x` for hexadecimal values, otherwise a leading zero denotes octal values. See also the SYMBOLS section in the configuration file.

-L path, --lib-path path

Specify a library search path. This option may be used more than once. It adds a directory to the search path for library files. Libraries specified without a path are searched in the current directory, in the list of directories specified using `--lib-path`, in directories given by environment variables, and in a built-in default directory.

-Ln

This option allows you to create a file that contains all global labels and may be loaded into the VICE emulator using the `ll` (load label) command or into the Oricutron emulator using the `sl` (symbols load) command. You may use this to debug your code with VICE. Note: Older versions had some bugs in the label code. If you have problems, please get the latest VICE version.

-S addr, --start-addr addr

Using -S you may define the default starting address. If and how this address is used depends on the config file in use. For the default configurations, only the "none", "apple2" and "apple2enh" systems honor an explicit start address, all other default configs provide their own.

-V, --version

This option prints the version number of the linker. If you send any suggestions or bugfixes, please include this number.

--cfg-path path

Specify a config file search path. This option may be used more than once. It adds a directory to the search path for config files. A config file given with the `-C` option that has no path in its name is searched in the current directory, in the list of directories specified using `--cfg-path`, in directories given by environment variables, and in a built-in default directory.

--dbgfile name

Specify an output file for debug information. Available information will be written to this file. Using the `-g` option for the compiler and assembler will increase the amount of information available. Please note that debug information generation is currently being developed, so the format of the file and its contents are subject to change without further notice.

--lib file

Links a library to the output. Use this command-line option instead of just naming the library file, if the linker is not able to determine the file type because of an unusual extension.

--obj file

Links an object file to the output. Use this command-line option instead of just naming the object file, if the linker is not able to determine the file type because of an unusual extension.

--obj-path path

Specify an object file search path. This option may be used more than once. It adds a directory to the search path for object files. An object file passed to the linker that has no path in its name is searched in the current directory, in the list of directories specified using `--obj-path`, in directories given by environment variables, and in a built-in default directory.

3. Search paths

Starting with version 2.10, there are now several search-path lists for files needed by the linker: one for libraries, one for object files, and one for config files.

3.1 Library search path

The library search-path list contains in this order:

1. The current directory.
2. Any directory added with the `--lib-path` option on the command line.
3. The value of the environment variable `LD65_LIB` if it is defined.
4. A subdirectory named `lib` of the directory defined in the environment variable `CC65_HOME`, if it is defined.
5. An optionally compiled-in library path.

3.2 Object file search path

The object file search-path list contains in this order:

1. The current directory.
2. Any directory added with the `--obj-path` option on the command line.

3. The value of the environment variable `LD65_OBJ` if it is defined.
4. A subdirectory named `obj` of the directory defined in the environment variable `CC65_HOME`, if it is defined.
5. An optionally compiled-in directory.

3.3 Config file search path

The config file search-path list contains in this order:

1. The current directory.
2. Any directory added with the `--cfg-path` option on the command line.
3. The value of the environment variable `LD65_CFG` if it is defined.
4. A subdirectory named `cfg` of the directory defined in the environment variable `CC65_HOME`, if it is defined.
5. An optionally compiled-in directory.

4. Detailed workings

The linker does several things when combining object modules:

First, the command line is parsed from left to right. For each object file encountered (object files are recognized by a magic word in the header, so the linker does not care about the name), imported and exported identifiers are read from the file and inserted in a table. If a library name is given (libraries are also recognized by a magic word, there are no special naming conventions), all modules in the library are checked if an export from this module would satisfy an import from other modules. All modules where this is the case are marked. If duplicate identifiers are found, the linker issues warnings.

That procedure (parsing and reading from left to right) does mean that a library may only satisfy references for object modules (given directly or from a library) named *before* that library. With the command line

```
ld65 crt0.o clib.lib test.o
```

the module `test.o` must not contain references to modules in the library `clib.lib`. But, if it does, you have to change the order of the modules on the command line:

```
ld65 crt0.o test.o clib.lib
```

Step two is, to read the configuration file, and assign start addresses for the segments and define any linker symbols (see Configuration files).

After that, the linker is ready to produce an output file. Before doing that, it checks its data for consistency. That is, it checks for unresolved externals (if the output format is not relocatable) and for symbol type mismatches (for example a zero-page symbol is imported by a module as an absolute symbol).

Step four is, to write the actual target files. In this step, the linker will resolve any expressions contained in the segment data. Circular references are also detected in this step (a symbol may have a circular reference that goes unnoticed if the symbol is not used).

Step five is to output a map file with a detailed list of all modules, segments and symbols encountered.

And, last step, if you give the `-v` switch twice, you get a dump of the segment data. However, this may be quite unreadable if you're not a developer. :-)

5. Configuration files

Configuration files are used to describe the layout of the output file(s). Two major topics are covered in a config file: The memory layout of the target architecture, and the assignment of segments to memory areas.

In addition, several other attributes may be specified.

Case is ignored for keywords, that is, section or attribute names, but it is *not* ignored for names and strings.

5.1 Memory areas

Memory areas are specified in a `MEMORY` section. Let's have a look at an example (this one describes the usable memory layout of the C64):

```
MEMORY {
    RAM1:  start = $0800, size = $9800;
    ROM1:  start = $A000, size = $2000;
    RAM2:  start = $C000, size = $1000;
    ROM2:  start = $E000, size = $2000;
}
```

As you can see, there are two RAM areas and two ROM areas. The names (before the colon) are arbitrary names that must start with a letter, with the remaining characters being letters or digits. The names of the memory areas are used when assigning segments. As mentioned above, case is significant for those names.

The syntax above is used in all sections of the config file. The name (`ROM1` etc.) is said to be an identifier, the remaining tokens up to the semicolon specify attributes for this identifier. You may use the equal sign to assign values to attributes, and you may use a comma to separate attributes, you may also leave both out. But you *must* use a semicolon to mark the end of the attributes for one identifier. The section above may also have looked like this:

```
# Start of memory section
MEMORY
{
    RAM1:
        start $0800
        size $9800;
    ROM1:
        start $A000
        size $2000;
    RAM2:
        start $C000
        size $1000;
    ROM2:
        start $E000
        size $2000;
}
```

There are of course more attributes for a memory section than just start and size. Start and size are mandatory attributes, that means, each memory area defined *must* have these attributes given (the linker will check that). I will cover other attributes later. As you may have noticed, I've used a comment in the example above. Comments start with a hash mark (`#'), the remainder of the line is ignored if this character is found.

5.2 Segments

Let's assume you have written a program for your trusty old C64, and you would like to run it. For testing purposes, it should run in the `RAM` area. So we will start to assign segments to memory sections in the `SEGMENTS` section:

```
SEGMENTS {
    CODE:   load = RAM1, type = ro;
    RODATA: load = RAM1, type = ro;
    DATA:   load = RAM1, type = rw;
    BSS:    load = RAM1, type = bss, define = yes;
}
```

What we are doing here is telling the linker, that all segments go into the RAM1 memory area in the order specified in the SEGMENTS section. So the linker will first write the CODE segment, then the RODATA segment, then the DATA segment - but it will not write the BSS segment. Why? Here enters the segment type: For each segment specified, you may also specify a segment attribute. There are four possible segment attributes:

```
ro      means readonly
rw      means read/write
bss     means that this is an uninitialized segment
zp      a zeropage segment
```

So, because we specified that the segment with the name BSS is of type bss, the linker knows that this is uninitialized data, and will not write it to an output file. This is an important point: For the assembler, the BSS segment has no special meaning. You specify, which segments have the bss attribute when linking. This approach is much more flexible than having one fixed bss segment, and is a result of the design decision to supporting an arbitrary segment count.

If you specify "type = bss" for a segment, the linker will make sure that this segment does only contain uninitialized data (that is, zeroes), and issue a warning if this is not the case.

For a bss type segment to be useful, it must be cleared somehow by your program (this happens usually in the startup code - for example the startup code for cc65-generated programs takes care about that). But how does your code know, where the segment starts, and how big it is? The linker is able to give that information, but you must request it. This is, what we're doing with the "define = yes" attribute in the BSS definitions. For each segment, where this attribute is true, the linker will export three symbols.

```
__NAME_LOAD__   This is set to the address where the
                segment is loaded.
__NAME_RUN__    This is set to the run address of the
                segment. We will cover run addresses
                later.
__NAME_SIZE__   This is set to the segment size.
```

Replace NAME by the name of the segment, in the example above, this would be BSS. These symbols may be accessed by your code.

Now, as we've configured the linker to write the first three segments and create symbols for the last one, there's only one question left: Where does the linker put the data? It would be very convenient to have the data in a file, wouldn't it?

5.3 Output files

We don't have any files specified above, and indeed, this is not needed in a simple configuration like the one above. There is an additional attribute "file" that may be specified for a memory area, that gives a file name to write the area data into. If there is no file name given, the linker will assign the default file name. This is "a.out" or the one given with the -o option on the command line. Since the default behaviour is OK for our purposes, I did not use the attribute in the example above. Let's have a look at it now.

The "file" attribute (the keyword may also be written as "FILE" if you like that better) takes a string enclosed in double quotes (`"") that specifies the file, where the data is written. You may specify the same file several times, in that case the data for all memory areas having this file name is written into this file, in the order of the memory areas defined in the MEMORY section. Let's specify some file names in the MEMORY section used above:

```
MEMORY {
    RAM1:  start = $0800, size = $9800, file = %O;
    ROM1:  start = $A000, size = $2000, file = "rom1.bin";
    RAM2:  start = $C000, size = $1000, file = %O;
    ROM2:  start = $E000, size = $2000, file = "rom2.bin";
}
```

The %O used here is a way to specify the default behaviour explicitly: %O is replaced by a string (including the quotes) that contains the default output name, that is, "a.out" or the name specified with the -o option on the command line. Into this file, the linker will first write any segments that go into RAM1, and will append then the segments for RAM2, because the memory areas are given in this order. So, for the RAM areas, nothing has really changed.

We've not used the ROM areas, but we will do that below, so we give the file names here. Segments that go into ROM1 will be written to a file named "rom1.bin", and segments that go into ROM2 will be written to a file named "rom2.bin". The name given on the command line is ignored in both cases.

Assigning an empty file name for a memory area will discard the data written to it. This is useful, if the memory area has segments assigned that are empty (for example because they are of type bss). In that case, the linker will create an empty output file. This may be suppressed by assigning an empty file name to that memory area.

The %O sequence is also allowed inside a string. So using

```
MEMORY {
    ROM1:  start = $A000, size = $2000, file = "%O-1.bin";
    ROM2:  start = $E000, size = $2000, file = "%O-2.bin";
}
```

would write two files that start with the name of the output file specified on the command line, with "-1.bin" and "-2.bin" appended respectively. Because '%' is used as an escape char, the sequence "%%" has to be used if a single percent sign is required.

5.4 LOAD and RUN addresses (ROMable code)

Let us look now at a more complex example. Say, you've successfully tested your new "Super Operating System" (SOS for short) for the C64, and you will now go and replace the ROMs by your own code. When doing that, you face a new problem: If the code runs in RAM, we need not to care about read/write data. But now, if the code is in ROM, we must care about it. Remember the default segments (you may of course specify your own):

```
CODE            read-only code
RODATA          read-only data
DATA            read/write data
BSS             uninitialized data, read/write
```

Since BSS is not initialized, we must not care about it now, but what about DATA? DATA contains initialized data, that is, data that was explicitly assigned a value. And your program will rely on these values on startup. Since there's no way to remember the contents of the data segment, other than storing it into one of the ROMs, we have to put it there. But unfortunately, ROM is not writable, so we have to copy it into RAM before running the actual code.

The linker won't copy the data from ROM into RAM for you (this must be done by the startup code of your program), but it has some features that will help you in this process.

First, you may not only specify a "load" attribute for a segment, but also a "run" attribute. The "load" attribute is mandatory, and, if you don't specify a "run" attribute, the linker assumes that load area and run area are the same. We will use this feature for our data area:

```
SEGMENTS {
    CODE:   load = ROM1, type = ro;
    RODATA: load = ROM2, type = ro;
    DATA:   load = ROM2, run = RAM2, type = rw, define = yes;
    BSS:    load = RAM2, type = bss, define = yes;
}
```

Let's have a closer look at this SEGMENTS section. We specify that the CODE segment goes into ROM1 (the one at $A000). The readonly data goes into ROM2. Read/write data will be loaded into ROM2 but is run in RAM2. That means that all references to labels in the DATA segment are relocated to be in RAM2, but the segment is written to ROM2. All your startup code has to do is, to copy the data from its location in ROM2 to the final location in RAM2.

So, how do you know, where the data is located? This is the second point, where you get help from the linker. Remember the "define" attribute? Since we have set this attribute to true, the linker will define three external symbols for the data segment that may be accessed from your code:

```
__DATA_LOAD__   This is set to the address where the segment
                is loaded, in this case, it is an address in
                ROM2.
__DATA_RUN__    This is set to the run address of the segment,
                in this case, it is an address in RAM2.
__DATA_SIZE__   This is set to the segment size.
```

So, what your startup code must do, is to copy __DATA_SIZE__ bytes from __DATA_LOAD__ to __DATA_RUN__ before any other routines are called. All references to labels in the DATA segment are relocated to RAM2 by the linker, so things will work properly.

There's a library subroutine called copydata (in a module named copydata.s) that might be used to do actual copying. Be sure to have a look at it's inner workings before using it!

5.5 Other MEMORY area attributes

There are some other attributes not covered above. Before starting the reference section, I will discuss the remaining things here.

You may request symbols definitions also for memory areas. This may be useful for things like a software stack, or an I/O area.

```
MEMORY {
    STACK:  start = $C000, size = $1000, define = yes;
}
```

This will define some external symbols that may be used in your code:

```
__STACK_START__         This is set to the start of the memory
                        area, $C000 in this example.
__STACK_SIZE__          The size of the area, here $1000.
__STACK_LAST__          This is NOT the same as START+SIZE.
                        Instead, it is defined as the first
                        address that is not used by data. If we
                        don't define any segments for this area,
                        the value will be the same as START.
__STACK_FILEOFFS__      The binary offset in the output file. This
                        is not defined for relocatable output file
                        formats (o65).
```

A memory section may also have a type. Valid types are

```
ro      for readonly memory
rw      for read/write memory.
```

The linker will assure, that no segment marked as read/write or bss is put into a memory area that is marked as readonly.

Unused memory in a memory area may be filled. Use the "fill = yes" attribute to request this. The default value to fill unused space is zero. If you don't like this, you may specify a byte value that is used to fill these areas with the "fillval" attribute. If there is no "fillval" attribute for the segment, the "fillval" attribute of

the memory area (or its default) is used instead. This means that the value may also be used to fill unfilled areas generated by the assembler's .ALIGN and .RES directives.

The symbol %S may be used to access the default start address (that is, the one defined in the FEATURES section, or the value given on the command line with the -S option).

To support systems with banked memory, a special attribute named bank is available. The attribute value is an arbitrary 32-bit integer. The assembler has a builtin function named .BANK which may be used with an argument that has a segment reference (for example a symbol). The result of this function is the value of the bank attribute for the run memory area of the segment.

5.6 Other SEGMENT attributes

Segments may be aligned to some memory boundary. Specify "align = num" to request this feature. Num must be a power of two. To align all segments on a page boundary, use

```
SEGMENTS {
    CODE:   load = ROM1, type = ro, align = $100;
    RODATA: load = ROM2, type = ro, align = $100;
    DATA:   load = ROM2, run = RAM2, type = rw, define = yes,
            align = $100;
    BSS:    load = RAM2, type = bss, define = yes, align = $100;
}
```

If an alignment is requested, the linker will add enough space to the output file, so that the new segment starts at an address that is dividable by the given number without a remainder. All addresses are adjusted accordingly. To fill the unused space, bytes of zero are used, or, if the memory area has a "fillval" attribute, that value. Alignment is always needed, if you have used the .ALIGN command in the assembler. The alignment of a segment must be equal or greater than the alignment used in the .ALIGN command. The linker will check that, and issue a warning, if the alignment of a segment is lower than the alignment requested in an .ALIGN command of one of the modules making up this segment.

For a given segment you may also specify a fixed offset into a memory area or a fixed start address. Use this if you want the code to run at a specific address (a prominent case is the interrupt vector table which must go at address $FFFA). Only one of ALIGN or OFFSET or START may be specified. If the directive creates empty space, it will be filled with zero, of with the value specified with the "fillval" attribute if one is given. The linker will warn you if it is not possible to put the code at the specified offset (this may happen if other segments in this area are too large). Here's an example:

```
SEGMENTS {
    VECTORS: load = ROM2, type = ro, start = $FFFA;
}
```

or (for the segment definitions from above)

```
SEGMENTS {
    VECTORS: load = ROM2, type = ro, offset = $1FFA;
}
```

The "align", "start" and "offset" attributes change placement of the segment in the run memory area, because this is what is usually desired. If load and run memory areas are equal (which is the case if only the load memory area has been specified), the attributes will also work. There is also an "align_load" attribute that may be used to align the start of the segment in the load memory area, in case different load and run areas have been specified. There are no special attributes to set start or offset for just the load memory area.

A "fillval" attribute may not only be specified for a memory area, but also for a segment. The value must be an integer between 0 and 255. It is used as the fill value for space reserved by the assembler's .ALIGN and .RES commands. It is also used as the fill value for space between sections (part of a segment that comes from one object file) caused by alignment, but not for space that preceeds the first section.

To suppress the warning, the linker issues if it encounters a segment that is not found in any of the input files, use "optional=yes" as an additional segment attribute. Be careful when using this attribute, because a missing segment may be a sign of a problem, and if you're suppressing the warning, there is no one left to tell you about it.

5.7 The FILES section

The FILES section is used to support other formats than straight binary (which is the default, so binary output files do not need an explicit entry in the FILES section).

The FILES section lists output files and as only attribute the format of each output file. Assigning binary format to the default output file would look like this:

```
FILES {
    %O: format = bin;
}
```

The only other available output format is the o65 format specified by Andre Fachat (see the 6502 binary relocation format specification). It is defined like this:

```
FILES {
    %O: format = o65;
}
```

The necessary o65 attributes are defined in a special section labeled FORMAT.

5.8 The FORMAT section

The FORMAT section is used to describe file formats. The default (binary) format has currently no attributes, so, while it may be listed in this section, the attribute list is empty. The second supported format, o65, has several attributes that may be defined here.

```
FORMATS {
    o65: os = lunix, version = 0, type = small,
         import = LUNIXKERNEL,
         export = _main;
}
```

5.9 The FEATURES section

In addition to the MEMORY and SEGMENTS sections described above, the linker has features that may be enabled by an additional section labeled FEATURES.

The CONDES feature

CONDES is used to tell the linker to emit module constructor/destructor tables.

```
FEATURES {
    CONDES: segment = RODATA,
            type = constructor,
            label = __CONSTRUCTOR_TABLE__,
            count = __CONSTRUCTOR_COUNT__;
}
```

The CONDES feature has several attributes:

segment

This attribute tells the linker into which segment the table should be placed. If the segment does not exist, it is created.

`type`

Describes the type of the routines to place in the table. Type may be one of the predefined types `constructor`, `destructor`, `interruptor`, or a numeric value between 0 and 6.

`label`

This specifies the label to use for the table. The label points to the start of the table in memory and may be used from within user-written code.

`count`

This is an optional attribute. If specified, an additional symbol is defined by the linker using the given name. The value of this symbol is the number of entries (*not* bytes) in the table. While this attribute is optional, it is often useful to define it.

`order`

An optional attribute that takes one of the keywords `increasing` or `decreasing` as an argument. Specifies the sorting order of the entries within the table. The default is `increasing`, which means that the entries are sorted with increasing priority (the first entry has the lowest priority). "Priority" is the priority specified when declaring a symbol as `.CONDES` with the assembler, higher values mean higher priority. You may change this behaviour by specifying `decreasing` as the argument, the order of entries is reversed in this case.

Please note that the order of entries with equal priority is undefined.

`import`

This attribute defines a valid symbol name, that is added as an import to the modules defining a constructor/destructor of the given type. This can be used to force linkage of a module if this module exports the requested symbol.

Without specifying the `CONDES` feature, the linker will not create any tables, even if there are `condes` entries in the object files.

For more information see the `.CONDES` command in the ca65 manual.

The STARTADDRESS feature

`STARTADDRESS` is used to set the default value for the start address, which can be referenced by the `%S` symbol. The builtin default for the linker is $200.

```
FEATURES {
    # Default start address is $1000
    STARTADDRESS:       default = $1000;
}
```

Please note that order is important: The default start address must be defined *before* the `%S` symbol is used in the config file. This does usually mean, that the `FEATURES` section has to go to the top of the config file.

5.10 The SYMBOLS section

The configuration file may also be used to define symbols used in the link stage or to force symbols imports. This is done in the SYMBOLS section. The symbol name is followed by a colon and symbol attributes.

The following symbol attributes are supported:

addrsize

The addrsize attribute specifies the address size of the symbol and may be one of

- zp, zeropage or direct
- abs, absolute or near
- far
- long or dword.

Without this attribute, the default address size is abs.

type

This attribute is mandatory. Its value is one of export, import or weak. export means that the symbol is defined and exported from the linker config. import means that an import is generated for this symbol, eventually forcing a module that exports this symbol to be included in the output. weak is similar as export. However, the symbol is only defined if it is not defined elsewhere.

value

This must only be given for symbols of type export or weak. It defines the value of the symbol and may be an expression.

The following example defines the stack size for an application, but allows the programmer to override the value by specifying --define __STACKSIZE__=xxx on the command line.

```
SYMBOLS {
    # Define the stack size for the application
    __STACKSIZE__:  type = weak, value = $800;
}
```

6. Special segments

The builtin config files do contain segments that have a special meaning for the compiler and the libraries that come with it. If you replace the builtin config files, you will need the following information.

6.1 ONCE

The ONCE segment is used for initialization code run only once before execution reaches main() - provided that the program runs in RAM. You may for example add the ONCE segment to the heap in really memory constrained systems.

6.2 LOWCODE

For the LOWCODE segment, it is guaranteed that it won't be banked out, so it is reachable at any time by interrupt handlers or similar.

6.3 STARTUP

This segment contains the startup code which initializes the C software stack and the libraries. It is placed in its own segment because it needs to be loaded at the lowest possible program address on several platforms.

6.4 ZPSAVE

The ZPSAVE segment contains the original values of the zeropage locations used by the ZEROPAGE segment. It is placed in its own segment because it must not be initialized.

7. Copyright

ld65 (and all cc65 binutils) are (C) Copyright 1998-2005 Ullrich von Bassewitz. For usage of the binaries and/or sources the following conditions do apply:

This software is provided 'as-is', without any expressed or implied warranty. In no event will the authors be held liable for any damages arising from the use of this software.

Permission is granted to anyone to use this software for any purpose, including commercial applications, and to alter it and redistribute it freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not claim that you wrote the original software. If you use this software in a product, an acknowledgment in the product documentation would be appreciated but is not required.
2. Altered source versions must be plainly marked as such, and must not be misrepresented as being the original software.
3. This notice may not be removed or altered from any source distribution.