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.

Memory areas

Memory areas are specified in a MEMORY section. Lets have a look at an example:

```
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 these 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.

Segments

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? Enter the segment type: For each segment specified, you may also specify a segment attribute. There are five possible segment attributes:

ro	means readonly
wprot	same as ro but will be marked as write protected in the VICE label file if -Lp is given
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). 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?

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 specifiy 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.

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 other way to remember the contents of the data segment, than storing it into one of the ROMs, we have to put it there. But unfortunately, ROM is not writeable, so we have to copy it into RAM before running the actual code.

The linker cannot help you copying the data from ROM into RAM (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 it's 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.

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 three 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 it 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.

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. This value is also used to fill unfilled areas generated by the assemblers .ALIGN and .RES directives.

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 divideable 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 the 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 a .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;
}
```

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 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.

File names may be empty, data from segments assigned to a memory area with an empty file name is discarded. This is useful, if the a 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 symbol %S may be used to access the default start address (that is, $200 or the value given on the command line with the -S option).

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. It is defined like this:

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

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

Features

In addition to the MEMORY and SEGMENTS sections described above, the linker has features that may be enabled by an additional section labeled FEATURES. Currently, one such feature is available: 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 or destructor, 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

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.

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