Atari specific information for cc65

Shawn Jefferson and Christian Groessler

An overview over the Atari runtime system as it is implemented for the cc65 C compiler.

---

9. CONIO implementation

10. Technical details

11. Other hints

12. License

---

1. Overview

This file contains an overview of the Atari runtime system as it comes with the cc65 C compiler. It describes the memory layout, Atari specific header files, available drivers, and any pitfalls specific to that platform.

The Atari runtime support comes in two flavors: `atari` and `atarixl`. The `atari` target supports all Atari 8-bit computers, the `atarixl` only supports XL type or newer machines (excluding the 600XL).

The `atarixl` runtime makes the whole 64K of memory available, with the exception of the I/O area at $D000 - $D7FF. Since the `atarixl` runtime has some limitations, it is recommended to use the `atari` target unless lack of memory dictates the use of the `atarixl` target.

Please note that Atari specific functions are just mentioned here, they are described in detail in the separate function reference. Even functions marked as "platform dependent" may be available on more than one platform. Please see the function reference for more information.

2. Binary format

The Atari DOS executable file format supports more than one load block (*chunk*).

The default binary output format generated by the linker for the Atari target is a machine language program with a standard executable header (FF FF <load chunk #1> ... <load chunk #n>). A load chunk has the format [<2 byte start address> <2 bytes end address> <chunk data>]. A run vector is added to the end of the file ($02E0 $02E1 <run vector>) and is calculated using the `start` label in crt0.s. (Technically the run vector is also a load chunk, but is not regarded as such here.)

An `atari` program has two load chunks, an `atarixl` program has three load chunks. The load chunks are defined in the linker configuration files. For more detailed information about the load chunks see the chapter Technical details. For the discussion here it's sufficient to know that the first load chunk(s) do preparation work and the main part of the program is in the last load chunk.

The values determining the size of the main part of the program (the second load chunk for `atari`, the third load chunk for `atarixl`) are calculated in the crt0.s file from the __STARTUP_LOAD__ and __BSS_LOAD__ values. Be aware of that if you create a custom linker config file and start moving segments around (see section Reserving a memory area inside the program).

3. Memory layout

3.1 atari target

The default linker config file assumes that the BASIC ROM is disabled (or the BASIC cartridge unplugged). This gives a usable memory range of [$2000-$BC1F]. The library startup code examines the current memory configuration, which depends on the size of the installed memory and cartridges. It does so by using the value in the MEMTOP ($2E5) variable as highest memory address the program can use. The initial stack pointer, which is the upper bound of memory used by the program, is set to this value, minus an optionally defined __RESERVED_MEMORY__ value.

The default load address of $2000 can be changed by creating a custom linker config file or by using the "--start-addr" cl65 command line argument or the "--start-addr" or "-S" ld65 command line arguments.

Please note that the first load chunk (which checks the available memory) will always be loaded at $2E00, regardless of the specified start address. This address can only be changed by a custom linker config file.

Special locations:

Text screen

The text screen depends on the installed memory size and cartridges and can be obtained from the SAVMSC variable ($58).

Stack

The C runtime stack is located at MEMTOP and grows downwards, regardless of how your linker config file is setup. This accommodates the different memory configurations of the Atari machines, as well as having a cartridge installed. You can override this behaviour by writing your own crt0.s file and linking it to your program (see also Final note).

Heap

The C heap is located at the end of the program and grows towards the C runtime stack.

3.2 atarixl target

The startup code rearranges the memory as follows:

1. Screen memory and display list are moved below the program start address.
2. The ROM is disabled, making the memory in the areas [$C000-$CFFF] and [$D800-$FFF9] available.
3. Character generator data is copied from ROM to the CHARGEN location specified in the linker config file. This is (in the default `atarixl.cfg` file) at the same address as where it is in ROM ($E000, it can be changed, see atarixl chargen location). With the character generator at $E000, there are two upper memory areas available, [$D800-$DFFF] and [$E400-$FFF9].

With the default load address of $2400 this gives a usable memory range of [$2400-$CFFF].

Please note that the first load chunk (which checks the system compatibility and available memory) will always be loaded at $2E00, regardless of the specified start address. This address can only be changed by a custom linker config file.

Special locations:

Text screen

The text screen depends on the selected load address ($2400 by default), and resides directly before that address, rounded to the next lower page boundary. The screen memory's start address can be obtained from the SAVMSC variable ($58).

Stack

The C runtime stack is located at end of the MAIN memory area ($CFFF) and grows downwards.

Heap

The C heap is located at the end of the program (end of BSS segment) and grows towards the C runtime stack.

4. Linker configurations

The ld65 linker comes with default config files for the Atari. There are two targets for the Atari, `atari` and `atarixl`. The default config file for `atari` is selected with `-t atari`, and the default config file for `atarixl` is selected with `-t atarixl`. The Atari package comes with additional secondary linker config files which can be used via `-t atari -C <configfile>` (for `atari` target) or `-t atarixl -C <configfile>` (for `atarixl` target).

4.1 atari config files

default config file (atari.cfg)

The default configuration is tailored to C programs. It creates files which have a default load address of $2000.

The files generated by this config file include the "system check" load chunk. It can optionally be left out, see Getting rid of the "system check" load chunk.

atari-asm.cfg

This config file aims to give the assembler programmer maximum flexibility. All program segments (CODE, DATA, etc.) are optional.

By default it creates regular DOS executable files, which have a default load address of $2E00. It's also possible to generate an image of just the program data without EXE header, load address, or (auto-)start address. To you so, you have to define the symbols __AUTOSTART__ and __EXEHDR__ when linking the program. Therefore, to generate a "plain" binary file, pass the options "-D__AUTOSTART__=1 -D__EXEHDR__=1" to the linker. It's also possible to create a non auto-starting program file, by defining only the __AUTOSTART__ symbol. Such a program has to be run manually after being loaded by DOS (for example by using the "M" option of DOS 2.5). Defining only the __EXEHDR__ symbol will create a (useless) file which doesn't conform to the DOS executable file format (like a "plain" binary file) but still has the "autostart" load chunk appended.

The sections of the file which the defines refer to (__AUTOSTART__ for the autostart trailer, __EXEHDR__ for the EXE header and load address) is *left out*, keep this in mind.

The values you assign to the two symbols __AUTOSTART__ and __EXEHDR__ don't matter.

atari-asm-xex.cfg

This config file allows writing multi segment binaries easily, without having to write the header explicitly on each segment.

It is similar to the atari-asm.cfg above, but uses the ATARI (xex) file format support on LD65 instead of the standard binary output, so it does not have the __AUTOSTART nor the __EXEHDR__ symbols.

Note that each MEMORY area in the configuration file will have it's own segment in the output file with the correct headers, and you can specify and init address INITAD) for each memory area.

atari-cart.cfg

This config file can be used to create 8K or 16K cartridges. It's suited both for C and assembly language programs.

By default, an 8K cartridge is generated. To create a 16K cartridge, pass the size of the cartridge to the linker, like "-D__CARTSIZE__=0x4000". The only valid values for __CARTSIZE__ are 0x2000 and 0x4000.

The option byte of the cartridge can be set with the __CARTFLAGS__ value, passed to the linker. The default value is $01, which means that the cartridge doesn't prevent the booting of DOS.

The option byte will be located at address $BFFD. For more information about its use, see e.g. "Mapping the Atari".

atari-cassette.cfg

This config file can be used to create cassette boot files. It's suited both for C and assembly language programs.

The size of a cassette boot file is restricted to 32K. Larger programs would need to be split in more parts and the parts to be loaded manually.

To write the generated file to a cassette, a utility (w2cas.com) to run on an Atari is provided in the util directory of atari target dir.

atari-xex.cfg

This config file shows how to write a binary using the ATARI (xex) file format support on LD65, this simplifies the memory areas and allows to add new memory areas easily without writing new headers and trailers.

Note that the default C library includes the system-check chunk, so in this linker configuration we suppress the importing of the header and trailer for this chunk by defining the standard import symbols to a 0 value. For the initialization address of the system-check chunk, the INITAD is set directly in the configuration.

4.2 atarixl config files

default config file (`atarixl.cfg`)

The default configuration is tailored to C programs. It creates files which have a default load address of $2400.

The files generated by this config file include the "system check" load chunk. It can optionally be left out, see Getting rid of the "system check" load chunk.

`atarixl-largehimem.cfg`

This is the same as the default config file, but it rearranges the high memory beneath the ROM into one large block. In order for this config file to work, the runtime library has to be recompiled with a special define. See the file `libsrc/atari/Makefile.inc` in the source distribution.

The files generated by this config file include the "system check" load chunk. It can optionally be left out, see Getting rid of the "system check" load chunk.

`atarixl-xex.cfg`

Similar to the `atari-xex.cfg` above, this config file shows how to write a binary using the ATARI (xex) file format support on LD65.

In addition to the suppressing of the system-check headers and trailers, this also suppresses the shadow-ram-preparation headers and trailers, but does this by defining an "UNUSED" memory area that is not written to the output file.

5. Platform specific header files

Programs containing Atari specific code may use the `atari.h` header file.

This also includes access to operating system locations (e.g. hardware shadow registers) by a structure called "`OS`". The names are the usual ones you can find in system reference manuals. Example:

```
...
    OS.savmsc = ScreenMemory;
    OS.color4 = 14;                         // white frame
    if (OS.stick0 != 15 || OS.ch != 255)    // key or stick input?
...
```

Please note that memory location 762/$2FA is called "`char_`" while the original name "`char`" conflicts with the C keyword.

If you like to use the OS names and locations for the original Atari 800 operating system, please "`#define OSA`" before including the `atari.h` header file. If you like to target the floating point register model of revision 2 machines, put a "`#define OS_REV2`" before including `atari.h`.

Access to the Basic programming language zero page variables is established by the structure "`BASIC`".

5.1 Atari specific functions

The functions and global variable listed below are special for the Atari. See the function reference for declaration and usage.

- get_ostype
- get_tv
- _dos_type
- _gtia_mkcolor
- _getcolor
- _getdefdev
- _graphics
- _is_cmdline_dos
- _rest_vecs
- _save_vecs
- _scroll
- _setcolor
- _setcolor_low
- _sound
- waitvsync

5.2 Hardware access

The following pseudo variables declared in the `atari.h` header file do allow access to hardware located in the address space. Some variables are structures, accessing the struct fields will access the chip registers.

`GTIA_READ` and `GTIA_WRITE`

The `GTIA_READ` structure allows read access to the GTIA. The `GTIA_WRITE` structure allows write access to the GTIA. See the `_gtia.h` header file located in the include directory for the declaration of the structure.

`POKEY_READ` and `POKEY_WRITE`

The `POKEY_READ` structure allows read access to the POKEY. The `POKEY_WRITE` structure allows write access to the POKEY. See the `_pokey.h` header file located in the include directory for the declaration of the structure.

`ANTIC`

The `ANTIC` structure allows read access to the ANTIC. See the `_antic.h` header file located in the include directory for the declaration of the structure.

`PIA`

The `PIA` structure allows read access to the PIA 6520. See the `_pia.h` header file located in the include directory for the declaration of the structure.

5.3 Display lists

A major feature of the Atari graphics chip "ANTIC" is to process instructions for the display generation. cc65 supports constructing these display lists by offering defines for the instructions. In conjunction with the "void"-variable extension of cc65, display lists can be created quite comfortable:

```
...
unsigned char ScreenMemory[100];

void DisplayList =
{
    DL_BLK8,
    DL_BLK8,
    DL_BLK8,
    DL_LMS(DL_CHR20x8x2),
    ScreenMemory,
    DL_CHR20x8x2,
    DL_CHR20x8x2,
    DL_CHR20x8x2,
    DL_BLK4,
    DL_CHR20x8x2,
    DL_JVB,
    &DisplayList
};
...
OS.sdlst = &DisplayList;
...
```

Please inspect the `_antic.h` header file to determine the supported instruction names. Modifiers on instructions can be nested without need for an order:

```
DL_LMS(DL_HSCROL(DL_VSCROL(DL_DLI(DL_MAP80x4x2))))
```

Please mind that ANTIC has memory alignment requirements for "player missile graphics"-data, font data, display lists and screen memory. Creation of a special linker configuration with appropriate aligned segments and switching to that segment in the c-code is usually necessary. A more memory hungry solution consists in using the posix_memalign() function in conjunction with copying your data to the allocated memory.

5.4 Character mapping

The Atari has two representations for characters:

1. ATASCII is character mapping which is similar to ASCII and used by the CIO system of the OS. This is the default mapping of cc65 when producing code for the atari target.
2. The internal/screen mapping represents the real value of the screen ram when showing a character.

For direct memory access (simplicity and speed) enabling the internal mapping can be useful. This can be achieved by including the "`atari_screen_charmap.h`" header.

A word of caution: Since the `0x00` character has to be mapped in an incompatible way to the C-standard, the usage of string functions in conjunction with internal character mapped strings delivers unexpected results regarding the string length. The end of strings are detected where you may not expect them (too early or (much) too late). Internal mapped strings typically support the "`mem...()`" functions.

For assembler sources the macro "`scrcode`" from the "`atari.mac`" package delivers the same feature.

You can switch back to the ATASCII mapping by including "`atari_atascii_charmap.h`".

Example:

```
#include <atari_screen_charmap.h>
char* pcScreenMappingString = "Hello Atari!";

#include <atari_atascii_charmap.h>
char* pcAtasciiMappingString = "Hello Atari!";
```

5.5 Keyboard codes

For direct keyboard scanning in conjunction with e.g. the OS location "CH" (764/$2FC), all keyboard codes are available as defined values on C and assembler side.

Example:

```
...
    while (!kbhit());
    switch (OS.ch)
    {
        case KEY_RETURN:
        ...
        case KEY_SPACE:
        ...
        case KEY_1:
        ...
    }
...
```

You can find the C defines in the file "`atari.h`" or "`atari.inc`" for the assembler variant.

6. Loadable drivers

The names in the parentheses denote the symbols to be used for static linking of the drivers.

6.1 Graphics drivers

atari	atarixl	screen resolution	display pages
atr3.tgi (atr3_tgi)	atrx3.tgi (atrx3_tgi)	40x24x4 (CIO mode 3, ANTIC mode 8)	1
atr4.tgi (atr4_tgi)	atrx4.tgi (atrx4_tgi)	80x48x2 (CIO mode 4, ANTIC mode 9)	1
atr5.tgi (atr5_tgi)	atrx5.tgi (atrx5_tgi)	80x48x4 (CIO mode 5, ANTIC mode A)	1
atr6.tgi (atr6_tgi)	atrx6.tgi (atrx6_tgi)	160x96x2 (CIO mode 6, ANTIC mode B)	1
atr7.tgi (atr7_tgi)	atrx7.tgi (atrx7_tgi)	160x96x4 (CIO mode 7, ANTIC mode D)	1
atr8.tgi (atr8_tgi)	atrx8.tgi (atrx8_tgi)	320x192x2 (CIO mode 8, ANTIC mode F)	1
atr8p2.tgi (atr8p2_tgi)	atrx8p2.tgi (atrx8p2_tgi)	320x192x2 (CIO mode 8, ANTIC mode F)	2
atr9.tgi (atr9_tgi)	atrx9.tgi (atrx9_tgi)	80x192x16b (CIO mode 9, ANTIC mode F, GTIA mode $40)	1
atr9p2.tgi (atr9p2_tgi)	atrx9p2.tgi (atrx9p2_tgi)	80x192x16b (CIO mode 9, ANTIC mode F, GTIA mode $40)	2
atr10.tgi (atr10_tgi)	atrx10.tgi (atrx10_tgi)	80x192x9 (CIO mode 10, ANTIC mode F, GTIA mode $80)	1
atr10p2.tgi (atr10p2_tgi)	atrx10p2.tgi (atrx10p2_tgi)	80x192x9 (CIO mode 10, ANTIC mode F, GTIA mode $80)	2

atr11.tgi (atr11_tgi)	atrx11.tgi (atrx11_tgi)	80x192x16h (CIO mode 11, ANTIC mode F, GTIA mode $C0)	1
atr14.tgi (atr14_tgi)	atrx14.tgi (atrx14_tgi)	160x192x2 (CIO mode 14, ANTIC mode C)	1
atr15.tgi (atr15_tgi)	atrx15.tgi (atrx15_tgi)	160x192x4 (CIO mode 15, ANTIC mode E)	1
atr15p2.tgi (atr15p2_tgi)	atrx15p2.tgi (atrx15p2_tgi)	160x192x4 (CIO mode 15, ANTIC mode E)	2

Many graphics modes require more memory than the text screen which is in effect when the program starts up. Therefore the programmer has to tell the program beforehand the memory requirements of the graphics modes the program intends to use.

On the `atari` target his can be done by using the __RESERVED_MEMORY__ linker config variable. The number specified there describes the number of bytes to subtract from the top of available memory as seen from the runtime library. This memory is then used by the screen buffer.

On the `atarixl` target the screen memory resides below the program load address. In order to reserve memory for a graphics mode, one simply uses a higher program load address. There are restrictions on selectable load addresses, see Selecting a good program load address.

The numbers for the different graphics modes presented below should only be seen as a rule of thumb. Since the screen buffer memory needs to start at specific boundaries, the numbers depend on the current top of available memory. The following numbers were determined by a BASIC program.

graphics mode	reserved memory
0	1
1	1
2	1
3	1
4	1
5	182
6	1182
7	3198
8	7120
9	7146
10	7146
11	7146
12	162
13	1
14	3278
15	7120
16	1
17	1
18	1
19	1
20	1
21	184
22	1192
23	3208
24	7146
25	7146
26	7146
27	7146
28	162
29	1
30	3304
31	7146

reserved memory required for different graphics modes

The values of "1" are needed because the graphics command crashes if it doesn't have at least one byte available. This seems to be a bug of the Atari ROM code.

Default drivers: `atr8.tgi (atr8_tgi)` and `atrx8.tgi (atrx8_tgi)`.

6.2 Extended memory drivers

Currently there is only one extended memory driver. It manages the second 64K of a 130XE.

atari	atarixl
atr130.emd (atr130_emd)	atrx130.emd (atrx130_emd)

6.3 Joystick drivers

Currently there are two joystick drivers available:

atari	atarixl	description
atrstd.joy (atrstd_joy)	atrxstd.joy (atrxstd_joy)	Supports up to two/four standard joysticks connected to the joystick ports of the Atari. (Four on the pre-XL systems, two on XL or newer.)
atrmj8.joy (atrmj8_joy)	atrxmj8.joy (atrxmj8_joy)	Supports up to eight standard joysticks connected to a MultiJoy adapter.

Default drivers: `atrstd.joy (atrstd_joy)` and `atrxstd.joy (atrxstd_joy)`.

6.4 Mouse drivers

Currently there are five mouse drivers available:

atari	atarixl	description
atrjoy.mou (atrjoy_mou)	atrxjoy.mou (atrxjoy_mou)	Supports a mouse emulated by a standard joystick.
atrst.mou (atrst_mou)	atrxst.mou (atrxst_mou)	Supports an Atari ST mouse.
atrami.mou (atrami_mou)	atrxami.mou (atrxami_mou)	Supports an Amiga mouse.
atrtrk.mou (atrtrk_mou)	atrxtrk.mou (atrxtrk_mou)	Supports an Atari trakball.
atrtt.mou (atrtt_mou)	atrxtt.mou (atrxtt_mou)	Supports an Atari touch tablet.

All mouse devices connect to joystick port #0.

Default drivers: `atrst.mou (atrst_mou)` and `atrxst.mou (atrxst_mou)`.

Mouse callbacks

There are two mouse callbacks available.

The "text mode" callbacks (`mouse_txt_callbacks`) display the mouse cursor as a "diamond" character on the standard "GRAPHICS 0" text mode screen. The mouse cursor character can be changed by an assembly file defining the character by exporting the zeropage symbol `mouse_txt_char`. The default file looks like this:

```
.export mouse_txt_char : zp = 96     ; 'diamond' screen code
```

The "P/M" callbacks (`mouse_pm_callbacks`) use Player-Missile graphics for the mouse cursor. The cursor shape can be changed, too, by an assembly file. Here's the default shape definition:

```
.export mouse_pm_bits
.export mouse_pm_height    : zeropage
```

```
        .export mouse_pm_hotspot_x : zeropage
        .export mouse_pm_hotspot_y : zeropage
        .rodata
mouse_pm_bits:
        .byte   %11110000
        .byte   %11000000
        .byte   %10100000
        .byte   %10010000
        .byte   %10001000
        .byte   %00000100
        .byte   %00000010
mouse_pm_height = * - mouse_pm_bits
; hot spot is upper left corner
mouse_pm_hotspot_x = 0
mouse_pm_hotspot_y = 0
```

`mouse_pm_bits` defines the shape of the cursor, `mouse_pm_height` defines the number of bytes in `mouse_pm_bits`. `mouse_pm_hotspot_x` and `mouse_pm_hotspot_y` define the position in the shape where "the mouse points to". When using this callback page #6 ($600 - $6FF) is used for the P/M graphics data and no P/M graphics can otherwise be used by the program. The height of the shape (`mouse_pm_height`) must not exceed 32 lines since the callback routines cannot handle more than 32 lines.

The default callbacks definition (`mouse_def_callbacks`) is an alias for the "P/M" callbacks.

6.5 RS232 device drivers

Currently there is one RS232 driver. It supports up to 9600 baud, requires hardware flow control (RTS/CTS) and uses the R: device (therefore an R: driver needs to be installed). It was tested with the 850 interface module.

atari	atarixl
atrrdev.ser (atrrdev_ser)	atrxrdev.ser (atrxrdev_ser)

7. Limitations

7.1 Realtime clock

Access to the realtime clock is supported only when running on SpartaDOS-X. There needs to be a realtime clock driver installed. This is normally the case in the default installation (CONFIG.SYS) of SpartaDOS-X. A missing realtime clock driver in SpartaDOS-X is not supported, and the program may crash when calling the `clock_settime()` or `clock_gettime()` functions.

The resolution of the realtime clock driver is 1 second.

7.2 atarixl target

- The display is cleared at program start and at program termination. This is a side effect of relocating the display memory below the program start address.
- Not all possible CIO and SIO functions are handled by the runtime stub code which banks the ROM in and out. All functions used by the runtime library are handled, though.
- The `_sys()` function is not supported.
- It is not compatible with DOSes or other programs using the memory below the ROM.

8. DIO implementation

The Atari supports disk drives with either 128 or 256 byte sectors. The first three sectors of any disk are always 128 bytes long though. This is because the system can only boot from 128 bytes sectors.

Therefore the DIO read and write functions transfer only 128 bytes for sectors 1 to 3, regardless of the type of diskette.

9. CONIO implementation

The console I/O is speed optimized therefore support for XEP80 hardware or f80.com software is missing. Of course you may use stdio.h functions.

`cprintf` targets a 40 character line. On a 20-column display this has the unexpected effect of a blank line after your text. On such displays you can either use for example `gotoxy(20,0)` to target the "next" line, or you can switch to `write()` function which does not have this side effect.

10. Technical details

10.1 atari

Load chunks

An `atari` program contains two load chunks.

1. "system check"
   This load chunk is always loaded at address $2E00, and checks if the system has enough memory to run the program. It also checks if the program start address is not below MEMLO. If any of the checks return false, the loading of the program is aborted.
   The contents of this chunk come from the SYSCHKCHNK memory area of the linker config file.
2. main program
   This load chunk is loaded at the selected program start address (default $2000) and contains all of the code and data of the program.
   The contents of this chunk come from the MAIN memory area of the linker config file.

10.2 atarixl

General operation

The `atarixl` target banks out the ROM while the program is running in order to make more memory available to the program.

The screen memory is by default located at the top of available memory, $BFFF if BASIC is not enabled, $9FFF if BASIC is enabled. Therefore, in order to create a largest possible continuous memory area, the screen memory is moved below the program load address. This gives a memory area from <program load addr> to $CFFF.

The startup code installs wrappers for interrupt handlers and ROM routines. When an interrupt or call to a ROM routine happens, the wrappers enable the ROM, call the handler or routine, and disable the ROM again.

The "wrapping" of the ROM routines is done by changing the ROM entry point symbols in `atari.inc` to point to the wrapper functions.

For ROM functions which require input or output buffers, the wrappers copy the data as required to buffers in low memory.

Load chunks

An `atarixl` program contains three load chunks.

1. "system check"
   This load chunk is always loaded at address $2E00, and checks if the system is suitable for running the program. It also checks if there is enough room between MEMLO and the program start address to move the text mode screen buffer there. If any of the checks return false, the loading of the program is aborted.
   The contents of this chunk come from the SYSCHKCHNK memory area of the linker config file.
2. "shadow RAM prepare"
   The second load chunk gets loaded to the selected program load address (default $2400). It moves the screen memory below the program load address, copies the character generator from ROM to its new place in RAM, and copies the parts of the program which reside in high memory below the ROM to their place. The high memory parts are included in this load chunk.
   At the beginning of this load chunk there is a .bss area, which is not part of the EXE file. Therefore the on-disk start address of this load chunk will be higher than the selected start address. This .bss area (segment LOWBSS) contains the buffers for the double buffering of ROM input and output data. If you add contents to this segment be aware that the contents won't be zero initialized by the startup code.
   The contents of this chunk come from the SRPREPCHNK memory area of the linker config file.

3. main program
   This load chunk is loaded just above the LOWBSS segment, replacing the code of the previous load chunk. It contains all remaining code and data sections of the program, including the startup code.
   The contents of this chunk come from the RAM memory area of the linker config file.

Moving screen memory below the program start address

When setting a graphics mode, the ROM looks at the RAMTOP location. RAMTOP describes the amount of installed memory in pages (RAMTOP is only one byte). The screen memory and display list are placed immediately below RAMTOP.

Now in order to relocate the screen memory to lower memory, the startup code puts a value into RAMTOP which causes the ROM routines to allocate the display memory below the program start address and then it issues a ROM call to setup the regular text mode.

Selecting a good program load address

Due to the movement of the screen memory below the program start, there are some load addresses which are sub-optimal because they waste memory or prevent a higher resolution graphics mode from being enabled.

There are restrictions at which addresses screen memory (display buffer and display list) can be placed. The display buffer cannot cross a 4K boundary and a display list cannot cross a 1K boundary.

The startup code takes this into account when moving the screen memory down. If the program start address (aligned to the next lower page boundary) minus the screen buffer size would result in a screen buffer which spans a 4K boundary, the startup code lowers RAMTOP to this 4K boundary.
The size of the screen buffer in text mode is 960 ($3C0) bytes. So, for example, a selected start address of $2300 would span the 4K boundary at $2000. The startup code would adjust the RAMTOP value in such way that the screen memory would be located just below this boundary (at $1C40). This results in the area [$2000-$22FF] being wasted. Additionally, the program might fail to load since the lowest address used by the screen memory could be below MEMLO. (The lowest address used in this example would be at $1C20, where the display list would allocated.)

These calculations are performed by the startup code (in the first two load chunks), but the startup code only takes the default 40x24 text mode into account. If the program later wants to load TGI drivers which set a more memory consuming graphics mode, the user has to pick a higher load address. Using higher resolution modes there is a restriction in the ROM that it doesn't expect RAMTOP to be at arbitrary values. The Atari memory modules came only in 8K or 16K sizes, so the ROM expects RAMTOP to only have values in 8K steps. Therefore, when using the highest resolution modes the program start address must be at an 8K boundary.

Character generator location

The default `atarixl` linker config file (`atarixl.cfg`) leaves the character generator location at the same address where it is in ROM ($E000). This has the disadvatage to split the upper memory into two parts ([$D800-$DFFF] and [$E400-$FFF9]). For applications which require a large continuous upper memory area, an alternative linker config file (`atarixl-largehimem.cfg`) is provided. It relocates the character generator to $D800, providing a single big upper memory area at [$DC00-$FFF9].

With the character generator at a different address than in ROM, the routines which enable and disable the ROM also have to update the chargen pointer. This code is not enabled by default. In order to enable it, uncomment the line which sets CHARGEN_RELOC in `libsrc/atari/Makefile.inc` and recompile the `atarixl` runtime library.

11. Other hints

11.1 Function keys

Function keys are mapped to Atari + number key.

11.2 Passing arguments to the program

Command line arguments can be passed to `main()` when the used DOS supports it.

1. Arguments are separated by spaces.
2. Leading and trailing spaces around an argument are ignored.
3. The first argument passed to `main` is the program name.
4. A maximum number of 16 arguments (including the program name) are supported.

11.3 Interrupts

The runtime for the Atari uses routines marked as `.INTERRUPTOR` for interrupt handlers. Such routines must be written as simple machine language subroutines and will be called automatically by the VBI handler code when they are linked into a program. See the discussion of the `.CONDES` feature in the assembler manual.

Please note that on the Atari targets the `.INTERRUPTOR`s are being run in NMI context. The other targets run them in IRQ context.

11.4 Reserving a memory area inside a program

(This section is primarily applicable to the `atari` target, but the principles apply to `atatixl` as well.)

The Atari 130XE maps its additional memory into CPU memory in 16K chunks at address $4000 to $7FFF. One might want to prevent this memory area from being used by cc65. Other reasons to prevent the use of some memory area could be to reserve space for the buffers for display lists and screen memory.

The Atari executable format allows holes inside a program, e.g. one part loads into $2E00 to $3FFF, going below the reserved memory area (assuming a reserved area from $4000 to $7FFF), and another part loads into $8000 to $BC1F.

Each load chunk of the executable starts with a 4 byte header which defines its load address and size. In the following linker config files these headers are named HEADER and SECHDR (for the MEMORY layout), and accordingly NEXEHDR and CHKHDR (for the SEGMENTS layout).

Low code and high data example

Goal: Create an executable with 2 load chunks which doesn't use the memory area from $4000 to $7FFF. The CODE segment of the program should go below $4000 and the DATA and RODATA segments should go above $7FFF.

The main problem is that the EXE header generated by the cc65 runtime lib is wrong. It defines a single load chunk with the sizes/addresses of the STARTUP, LOWCODE, ONCE, CODE, RODATA, and DATA segments, in fact, the whole user program (we're disregarding the "system check" load chunk here).

The contents of the EXE header come from the EXEHDR and MAINHDR segments. The EXEHDR segment just contains the $FFFF value which is required to be the first bytes of the EXE file.
The MAINHDR are defined in in crt0.s. This cannot be changed without modifying and recompiling the cc65 atari runtime library. Therefore the original contents of this segment must be discarded and be replaced by a user created one. This discarding is done by assigning the MAINHDR segment to the (new introduced) DISCARD memory area. The DISCARD memory area is thrown away in the new linker config file (written to file ""). We add a new FSTHDR segment for the chunk header of the first chunk.

The user needs to create a customized linker config file which adds new memory areas and segments to hold the new header data for the first load chunk and the header data for the second load chunk. Also an assembly source file needs to be created which defines the contents of the new header data for the two load chunks.

This is an example of a modified cc65 Atari linker configuration file (split.cfg):

```
SYMBOLS {
    __STACKSIZE__:       value = $800   type = weak;    # 2K stack
    __RESERVED_MEMORY__: value = $0000, type = weak;
}
FEATURES {
    STARTADDRESS: default = $2E00;
}
MEMORY {
    ZP: start = $82, size = $7E, type = rw, define = yes;

    HEADER: start = $0000, size = $2, file = %O;        # first load chunk

    FSTHDR: start = $0000, size = $4, file = %O;        # second load chunk
    RAMLO: start = %S, size = $4000 - %S, file = %O;

    DISCARD: start = $4000, size = $4000, file = "";

    SECHDR: start = $0000, size = $4, file = %O;        # second load chunk
    RAM: start = $8000, size = $3C20, file = %O;        # $3C20: matches upper bound $BC1F
}
SEGMENTS {
    EXEHDR: load = HEADER, type = ro;

    MAINHDR: load = DISCARD, type = ro;

    NEXEHDR: load = FSTHDR, type = ro;                  # first load chunk
    STARTUP: load = RAMLO, type = ro, define = yes;
    LOWCODE: load = RAMLO, type = ro, define = yes, optional = yes;
```

```
    ONCE: load = RAMLO, type = ro, optional = yes;
    CODE: load = RAMLO, type = ro, define = yes;

    CHKHDR: load = SECHDR, type = ro;                   # second load chunk
    RODATA: load = RAM, type = ro, define = yes;
    DATA: load = RAM, type = rw, define = yes;
    BSS: load = RAM, type = bss, define = yes;

    ZEROPAGE: load = ZP, type = zp;
    AUTOSTRT: load = RAM, type = ro;                    # defines program entry point
}
FEATURES {
    CONDES: segment = ONCE,
            type = constructor,
            label = __CONSTRUCTOR_TABLE__,
            count = __CONSTRUCTOR_COUNT__;
    CONDES: segment = RODATA,
            type = destructor,
            label = __DESTRUCTOR_TABLE__,
            count = __DESTRUCTOR_COUNT__;
}
```

A new memory area DISCARD was added. It gets loaded with the contents of the (now unused) MAINHDR segment. But the memory area isn't written to the output file. This way the contents of the MAINHDR segment get discarded.

The newly added NEXEHDR segment defines the correct chunk header for the first intended load chunk. It puts the STARTUP, LOWCODE, ONCE, and CODE segments, which are the segments containing only code, into load chunk #1 (RAMLO memory area).

The header for the second load chunk comes from the new CHKHDR segment. It puts the RODATA, DATA, BSS, and ZPSAVE segments into load chunk #2 (RAM memory area).

The contents of the new NEXEHDR and CHKHDR segments come from this file (split.s):

```
    .import __CODE_LOAD__, __BSS_LOAD__, __CODE_SIZE__
    .import __DATA_LOAD__, __RODATA_LOAD__, __STARTUP_LOAD__

    .segment "NEXEHDR"
    .word    __STARTUP_LOAD__
    .word    __CODE_LOAD__ + __CODE_SIZE__ - 1

    .segment "CHKHDR"
    .word    __RODATA_LOAD__
    .word    __BSS_LOAD__ - 1
```

Compile with

```
cl65 -t atari -C split.cfg -o prog.com prog.c split.s
```

Low data and high code example

Goal: Put RODATA and DATA into low memory and STARTUP, LOWCODE, ONCE, CODE, BSS, ZPSAVE into high memory (split2.cfg):

```
SYMBOLS {
    __STACKSIZE__:       value = $800   type = weak;    # 2K stack
    __RESERVED_MEMORY__: value = $0000, type = weak;
}
FEATURES {
    STARTADDRESS: default = $2E00;
}
MEMORY {
    ZP: start = $82, size = $7E, type = rw, define = yes;

    HEADER: start = $0000, size = $2, file = %O;        # first load chunk

    FSTHDR: start = $0000, size = $4, file = %O;        # second load chunk
    RAMLO: start = %S, size = $4000 - %S, file = %O;

    DISCARD: start = $4000, size = $4000, file = "";

    SECHDR: start = $0000, size = $4, file = %O;        # second load chunk
    RAM: start = $8000, size = $3C20, file = %O;        # $3C20: matches upper bound $BC1F
}
SEGMENTS {
    EXEHDR: load = HEADER, type = ro;                     # discarded old EXE header

    MAINHDR: load = DISCARD, type = ro;

    NEXEHDR: load = FSTHDR, type = ro;                  # first load chunk
    RODATA: load = RAMLO, type = ro, define = yes;
    DATA: load = RAMLO, type = rw, define = yes;

    CHKHDR: load = SECHDR, type = ro;                   # second load chunk
```

```
    STARTUP: load = RAM, type = ro, define = yes;
    ONCE: load = RAM, type = ro, optional = yes;
    CODE: load = RAM, type = ro, define = yes;
    BSS: load = RAM, type = bss, define = yes;

    ZEROPAGE: load = ZP, type = zp;
    AUTOSTRT: load = RAM, type = ro;                    # defines program entry point
}
FEATURES {
    CONDES: segment = ONCE,
            type = constructor,
            label = __CONSTRUCTOR_TABLE__,
            count = __CONSTRUCTOR_COUNT__;
    CONDES: segment = RODATA,
            type = destructor,
            label = __DESTRUCTOR_TABLE__,
            count = __DESTRUCTOR_COUNT__;
}
```

New contents for NEXEHDR and CHKHDR are needed (split2.s):

```
    .import __STARTUP_LOAD__, __BSS_LOAD__, __DATA_SIZE__
    .import __DATA_LOAD__, __RODATA_LOAD__

    .segment "NEXEHDR"
    .word    __RODATA_LOAD__
    .word    __DATA_LOAD__ + __DATA_SIZE__ - 1

    .segment "CHKHDR"
    .word    __STARTUP_LOAD__
    .word    __BSS_LOAD__ - 1
```

Compile with

```
cl65 -t atari -C split2.cfg -o prog.com prog.c split2.s
```

Final note

There are two other memory areas which don't appear directly in the linker config file. They are the stack and the heap.

The cc65 runtime lib places the stack location at the end of available memory. This is dynamically set from the MEMTOP system variable at startup. The heap is located in the area between the end of the BSS segment and the top of the stack as defined by __STACKSIZE__.

If BSS and/or the stack shouldn't stay at the end of the program, some parts of the cc65 runtime lib need to be replaced/modified.

common/_heap.s defines the location of the heap and atari/crt0.s defines the location of the stack by initializing sp.

11.5 Upgrading from an older cc65 version

If you are using a customized linker config file you might get some errors regarding the MAINHDR segment. Like this:

```
ld65: Error: Missing memory area assignment for segment 'MAINHDR'
```

The old "HEADER" memory description contained six bytes: $FFFF and the first and last memory address of the program. For the "system check" load chunk this had to be split into two memory assignments The "HEADER" now only contains the $FFFF. The main program's first and last memory address were moved to a new segment, called "MAINHDR", which in the new linker config file goes into its own memory area (also called "MAINHDR").

A simple way to adapt your old linker config file is to add the following line to the "SEGMENTS" section:

```
MAINHDR: load = HEADER, type = ro;
```

11.6 Getting rid of the "system check" load chunk

If, for some reason, you don't want to include the "system check" load chunk, you can do so by defining the symbol __SYSTEM_CHECK__ when linking the program. The "system check" chunk doesn't include vital parts of the program. So if you don't want the system checks, it is save to leave them out. This is probably mostly interesting for debugging.

When using cl65, you can leave it out with this command line:

```
cl65 -Wl -D__SYSTEM_CHECK__=1 <arguments>
```

The value you assign to __SYSTEM_CHECK__ doesn't matter. If the __SYSTEM_CHECK__ symbol is defined, the load chunk won't be included.

12. License

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.