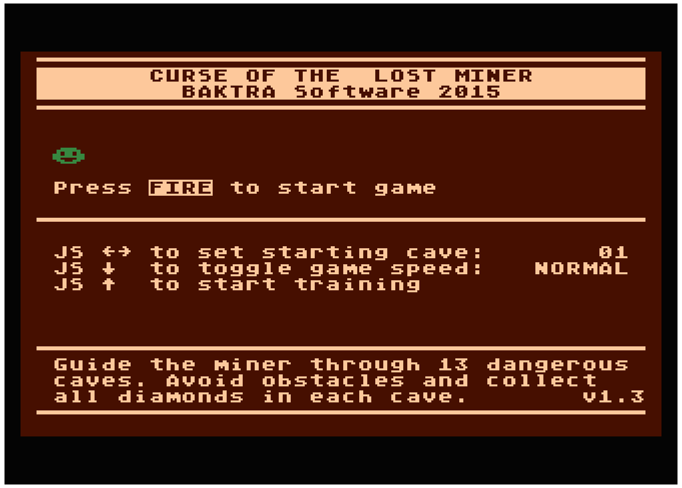
Including Game Data and Assembler Routines in a Program Written in CC65

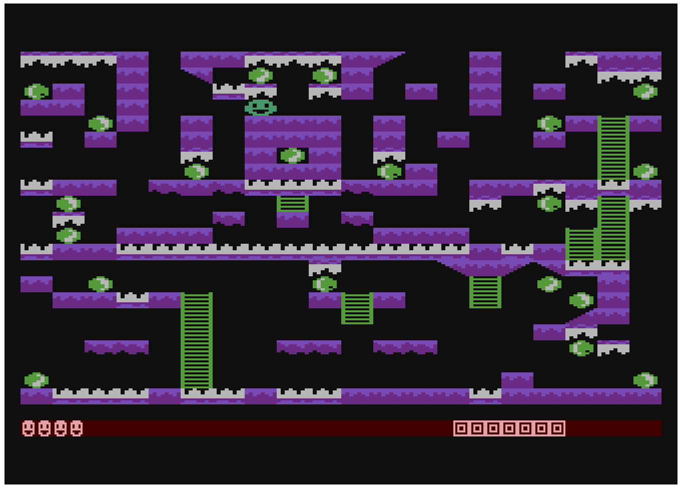
Michael Kalouš (BAKTRA Software)
Original version for the Atariáda 2016 event
English translation and updates for **Atari Sector**

Introduction

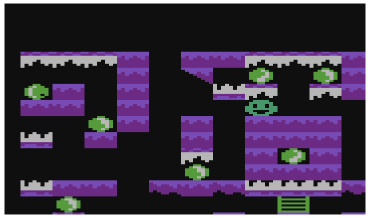
- There are several ways of including game data and assembler routines in programs written in CC65
- This text is based on experience with development of several versions of the Curse of the Lost Miner (CuLoMin) game
- I focus on **tools provided by the CC65 development environment**
- I compare this approach with usage of external linker



```
      CURSE OF THE  LOST MINER
        BAKTRA Software 2015

 Press FIRE to start game

 JS ←→ to set starting cave:       01
 JS ↓  to toggle game speed:   NORMAL
 JS ↑  to start training

 Guide the miner through 13 dangerous
 caves. Avoid obstacles and collect
 all diamonds in each cave.      v1.3
```



Game Data



- Game World (Caves)
  - A flat file created with the cave editor
  - Can be placed anywhere in the memory
- Character Sets
  - Two character sets, each 1 KB long
  - Must be placed at 1 KB boundary

QWE

- Display lists
  - Display list for game screen (GR. 12 and GR. 0, DLI)
  - Requires special treatment when crossing 1 KB boundary

LMS
JVB

- Music and Sound Effects
  - Music composed with the Raster Music Tracker (RMT) is a binary load file. The replay routine is a multi-section binary load file with further requirements described in RMT documentation.

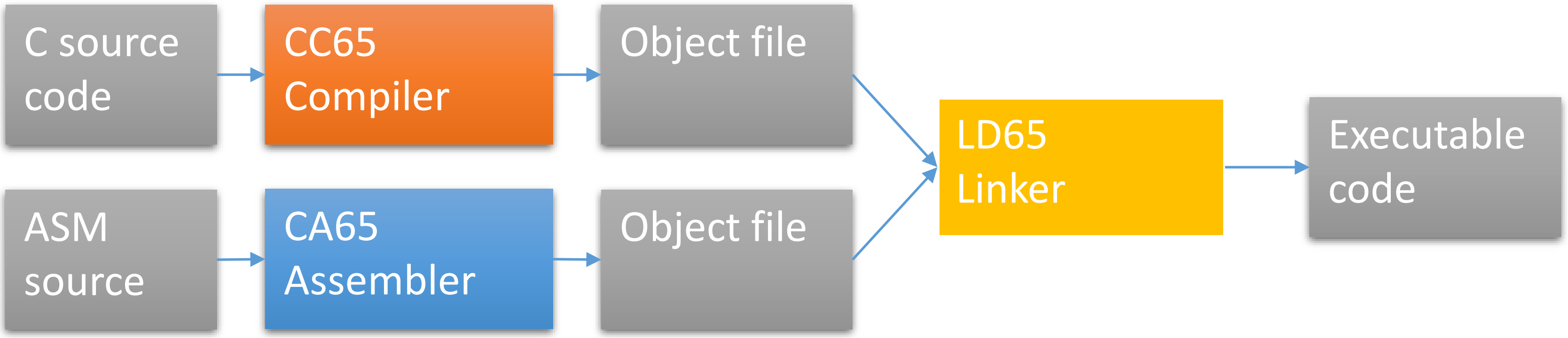
Assembler Routines

- Interrupt Handlers
  - VBI – Handler, setting VBI vector using the official OS service
  - DLI – Handler, setting DLI vector. Color and character set changes.
- Music and Sound Effects
  - RMT has API for assembler only, so we must call RMT using assembler
- Others
  - Code that jumps to the COLDSV vector when the ESC key is pressed in the main menu. We must do so, because the game destroys DOS.

Important Memory Areas

- Display Memory for Game Screen
  - Portion for the cave (GRAPHICS 12)
  - Portion for the status bar (GRAPHICS 0)
  - Display memory requires special treatment when crossing 4 KB boundary
- Area for Player-Missile-Graphics (PMG)
  - Single-line resolution
  - Area must start at 2-KB boudnary

How CC65 and CA65 Work



- Object File
  - Machine Code
  - Data
  - **Symbolic, unresolved addresses** in the machine code and data
  - Information about **segments** the machine code and data belongs to
- Linker creates a file that contains **executable** machine code with symbolic addresses resolved to concrete numbers
  - Binary load file, cartridge image, or even tape image

A Brief Digression for the Sake of Clear Terminology

- In this presentation
  - A **section** is a portion of a DOS 2 binary load file with start address, end address and data
  - A **segment** is a named block of code and data processed by CC65, CA65, and LD66

More on Segments

- A segment is a block of machine code and/or data
- All generated code belongs to a particular segment
- Motto: *"Everything in a segment, nothing against the segment, nothing outside a segment"*
- Code and data that are somehow related should be placed to the same segment

Segments in CC65

- CC65 uses 4 segments by default
  - **CODE** for code inside functions
  - **DATA** for mutable data
  - **RODATA** pro immutable data
  - **BSS** for data that is not initialized to particular values
- Other Segments
  - Libraries specific for given platform can use other segments
  - ZP, STARTUP, SYSCHK
- Segment Usage
  - DATA – For variables defined outside functions or static variables initialized to a particular value
  - RODATA – Constants defined outside functions, typically string literals
  - BSS – Variables defined outside functions not initialized to a particular value
  - Automatic variables are stored in stack, of course

Example

```
int counter;
char message = “my message”;
int maxTasks = 2;

int main(void) {
    int a=0;

    a=counter + maxtasks;
    printf(“%d\n”,a);
}
```

BSS

RODATA

DATA

!! STACK !!

CODE

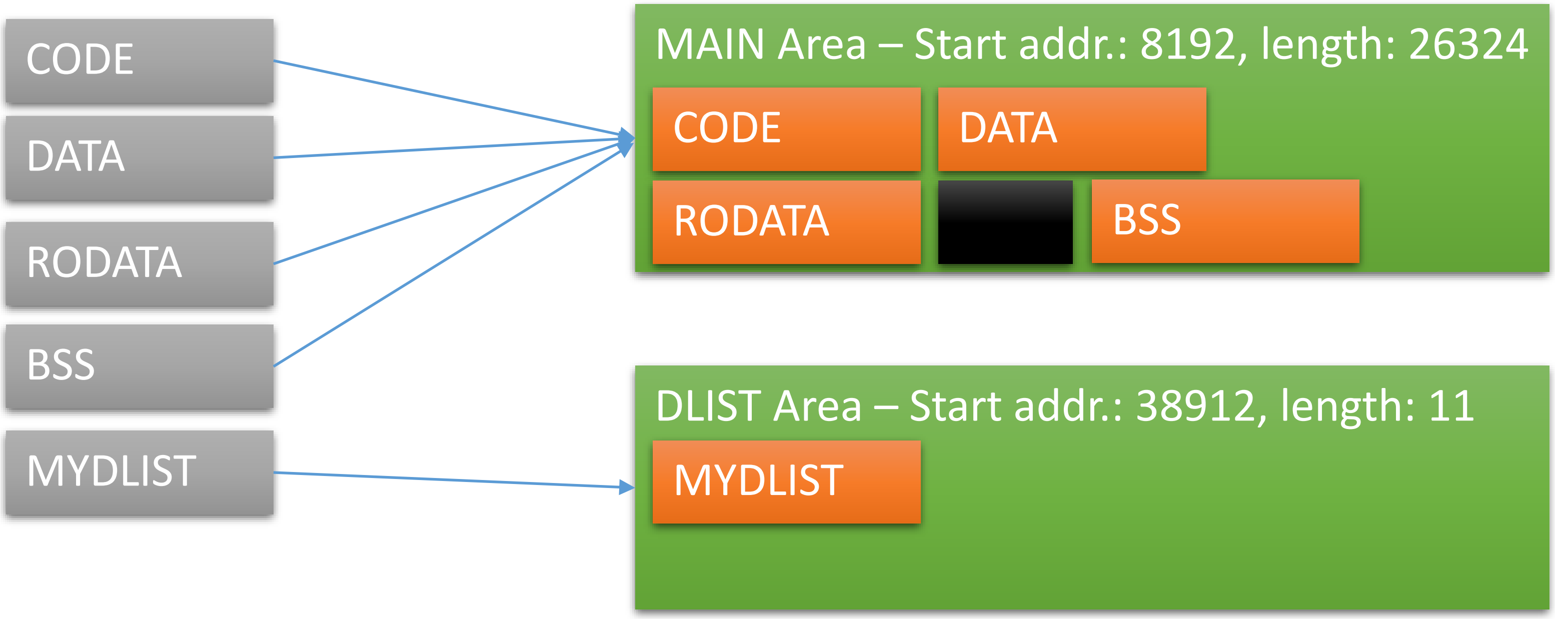
Segments in CA65

- Each piece of code or data belongs to a particular segment
- Determined by the `.segment "segname"` directive
- Example

```
.segment "code"
      lda #10
      sta ten
.segment "mydlist"
mydl: .byte 112,112,112    ; Blank
      .byte 66             ; LMS GR.0
      .word screenmem      ; Screen area
      .byte 2,2            ; 2x GR.0
      .byte 65             ; JVB
      .word mydl           ; From the top
```

Placing Segments to Memory Areas

- Our code and data is in the devoted segments
- We need to place our segments to **memory areas** and write the memory areas to the resulting executable file
- This is done by the LD65 linker and defined in the linker's configuration file


CODE
DATA
RODATA
BSS
MYDLIST
MAIN Area – Start addr.: 8192, length: 26324
CODE
DATA
RODATA
BSS
DLIST Area – Start addr.: 38912, length: 11
MYDLIST

Attributes of Segments

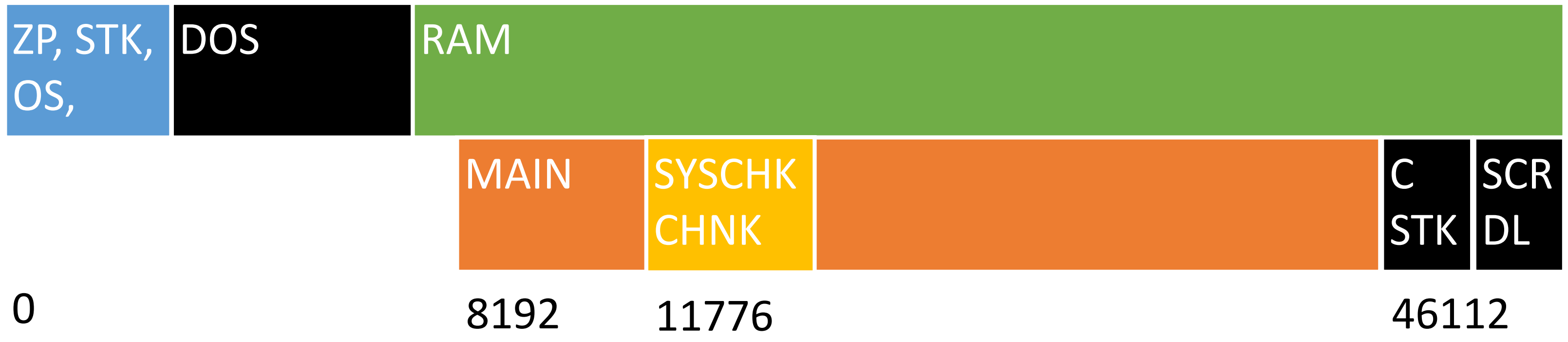
- Name of the segment ("code")
- Memory area for loading ("MAINROM")
- Memory area for execution ("MAINRAM")
- Type (rw, ro, zp, bss).
  - Segments with bss type are not written to the output file
- Mandatory or optional presence of the segment
  - If the linker does not find the segment in the object code, it can display an error message
- Fixed start address of the segment
- Alignment of the segment and value of the padding bytes

Attributes of Memory Areas

- Name ("MAINRAM”)
- Start Address
- Length of the area
- Output file (the data of the memory area are written there). Not mandatory. Data can be written to "nowhere".
- Type (rw, ro)
- Padding of unused areas and values of the padding bytes. Useful when creating cartridge images that have to be exactly 8 KB long.

Default Layout for the Atari Target (-t atari)

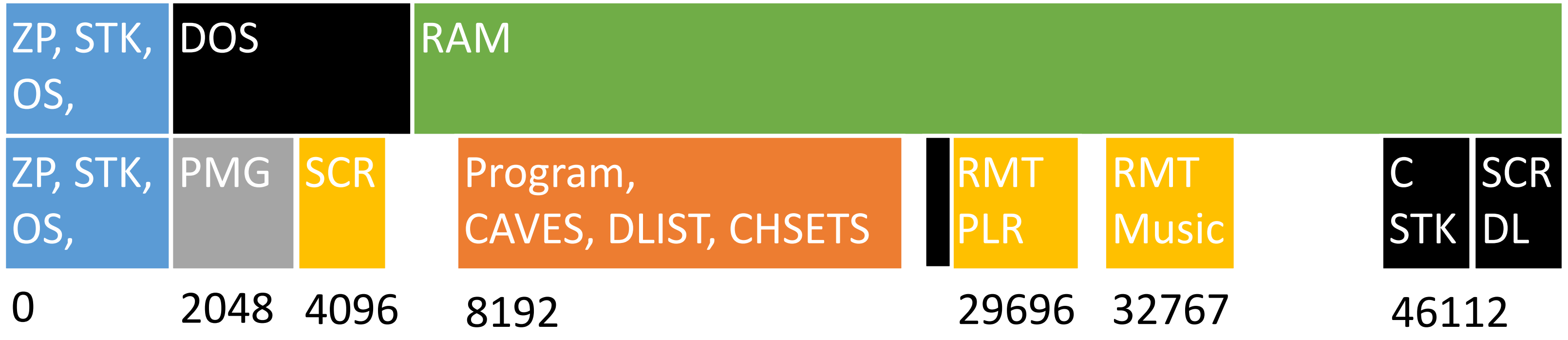
- Important Memory Areas
  - MAIN (for the CODE, RODATA, DATA, and BSS segments)
  - SYSCHKCHNK (Code for system check)
- Other Areas
  - Binary load file header, headers of the sections of the binary load file, INIT vector, RUN vector. There are also areas devoted to the C stack and screen display used by the conio library at the very end of the available memory.
- Memory Map



Thinking about the Data - CuLoMin

- Basic Conditions
  - The game is not that big
  - DOS is needed to load the game, but then it is not needed.
- Using Memory
  - The main program stays at 8192
  - All game data (except RMT music and replay routine) is added to the memory area where the main program resides – with appropriate alignment of course. **We align the data**, so it doesn't cross the boundaries that require special treatment.
  - RMT music and replay routines are placed above the main program
  - PMG area and area for screen display occupy addresses where DOS was residing (below the main program and data)

Data in Memory



- Data stored in external files is included using assembler
- For each game data, we define a separate segment with required alignment.
- For RMT, we need a separate memory area
- Beginning of each segment starts with an assembler label. These labels help us to address the data from the main program

Assembler Routines

- Assembler routines are placed to a separate module
- We use the "CODE" segment, so the assembler routines are stored together with the main program
- Variables are placed in the "DATA" segment
- We export symbols that must be visible from the main program

Now Take Look at the Code

- `culomin_cc65\gamedata.asm`
- `culomin_cc65\routines.asm`
- `culomin_cc65\main.c`
- `culomin_cc65\linker.cfg`
- `culomin_cc65\compile.bat`

Using External Linker

- Processing and Advantages
  - The external linker will just append segments to the binary load file
  - Simple, straightforward, familiar to assembler programmers
- Disadvantages
  - Addresses of the game data are not directly visible in the main program
  - Addresses of the game data must be make visible for the main program, for example by using symbolic constants created with the `#define` preprocessor directive.

Using External Linker – The Code

- The minilinker.jar is used as an external linker. It requires Java to be installed.
- Refer to the following files
  - `culomin_extlinker\main.c`
  - `culomin_extlinker\rmt_sup.asm`
  - `culomin_extlinker\tools\readme.txt`
  - `culomin_extlinker\linkfile`
  - `culomin_extlinker\compile.bat`

Conclusion

- Conclusions
  - It is not that difficult to include game data in C programs. The CC65 development environment provides all necessary tools.
  - LD65 can be configured to obey certain restrictions for data placement (alignment of the character sets, display lists, PMG areas)
  - Data in form of binary load files represents only a minor problem.
  - It is best to plan the data layout in advance
  - It is easy to create multiple outputs (.xex, .cas, .bin)
- What Next
  - Study the included source code in greater detail
  - Refer to the CC65 documentation
  - Study the .cfg files shipped with CC65,
  - Study the source code of the Atari library shipped with CC65