Master Project

Memory Layout Management for Embedded Single and Multi-Core Systems

Peter LORENZ

August 10, 2017

Graz University of Technology
Faculty of Electrical Engineering
Institute for Technical Informatics

I declare that I carried out this master project independently, and only with the cited source, literature and other professional sources.

In..... Date.....

Signature.....

Embedded Automotive Systems Group

Supervisors

Univ.-Prof. Dipl.-Inf. Univ. Dr.rer.nat. Marcel Carsten BAUNACH
Eng. Renata Martins GOMES

Contents

Abstract

Memory management (MM) is the heart of every operating system (OS). It is crucial for programming. MM is basically responsible for allocating memory blocks for programs and deallocate blocks when they have not been used for a while.

The linker takes care of a program's address space. Some changes in the linker scripts have impact on the memory's layout. Yet, since memory layouts are often changed, leads to a work overload checking if the changes are correct. We present our novel approach to *memory management tool* for maintaining linker scripts and at the same time simplifying the redesign of the memory layout by syntactically checking changes in the linker script.

Keywords Embedded Systems • Memory Management • Linker Scripts

1 Theoretical Background

Memory layout or also called memory map is a technical term to indicate the structure of data that resides in memory itself. In general, it contains information about total available memory and any reserved regions. There are many different meanings for the term memory layout/map. For example, a native debugger's memory map would refer to a mapping between an executable and memory regions or such as virtual memory of a process. However, in this piece of work this wide-ranged term is related to micro-controller units (MCUs) and these memory layouts are explicitly controlled by the link processes. The linker's command language has access to addresses of sections and placement of common blocks (MEMORY, SECTIONS).

To date, there is no known possibility to directly extract the memory layout from linker scripts directly at all. An approach will be explained to parse linker scripts to analyse its syntax and extract the necessary values of variables or sections and provide those values for further processing, i.e. visualization.

This thesis is structured as follows. First, a short overview of the build process is given to understand the importance of linking process and review some tools, which prepare memory layout information from different approaches. Finally, the own approach is explained, followed by implementation details.

1.1 Build Process

GCC compiles a source code by going through a sequence of intermediate steps before an executable is generated. Those steps resulting from different tools are invoked implicitly to finalize the compilation. The whole compilation process can basically be divided into four phases, whereas the last one, linking, is crucial for further explanations in this thesis.

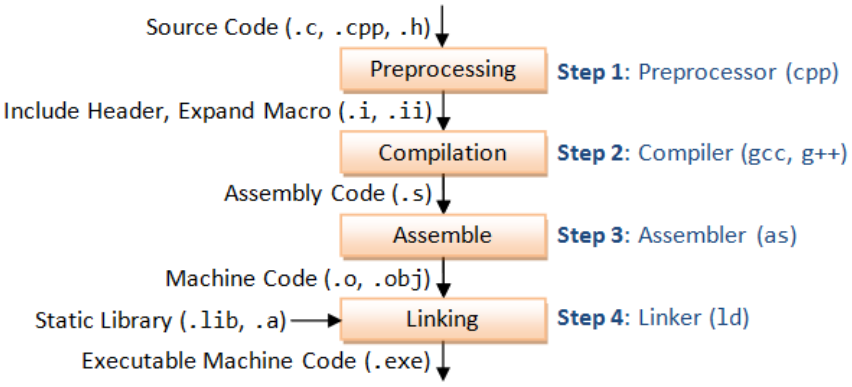

Figure 1.1: Compile, link and execute stage for a running program [NTU13]

Figure Figure 1.1 shows a summary of all 4 phases, and assuming that there are one or more source files with the file ending (*.c*, *.cpp*, *.h*). A sequence of tools process those source files:

1. **Preprocessing**: The first pass of any C/C++ compilation. All include-files, conditional compilation, instructions and macros are processed. The C preprocessor,often known as cpp, is used automatically by the C compiler to transform a C program before compilation (*cpp hello.c > hello.i*).

2. **Compilation**: The preprocessed code is translated into assembly instructions while the instruction set depends on the processor architecture target. Some compilers skip this step, because they support integrated assembler, where machine code is directly generated. The output files can manually generated with the console command *cc -S hello.c*.

3. **Assembly**: During this stage, an assembler is used to translate the assembly instructions to machine code or object code, which consists of instructions to be run on the target processor.

4. **Linking**: In this stage, we have machine code elements, but these are not connected. The linker will organise the pieces of object code so that functions in some pieces can successfully call functions in other pieces. It will also add pieces from library functions used by the program.

The output of the linker is an object file that contains all code from input object files in the same object file format. This is done by merging bss, text and data sections of the input files. As a result, all machine code from all input object files will be in the text section of the new file and all of the initialized and uninitialized variables will be put in the data and bss sections.

1.2 Overview of existing Memory Visualization Tools

Modern memory visualizations tools nowadays can not solve our problem, extracting memory information from linker scripts, but they can extract memory information in different ways. In this section, a brief overview of several memory visualization tools is given which serves us a starting point to design an own tool. The main focus stays on tools which are intended for Windows platforms, because our tool is also aimed for those platforms.

1.2.1 AMap

AMap is an open-source software to analyze *.map* files which are produced by both MS Visual Studio and GCC compilers. The installation also delivers some command line tools and can be directly used from a terminal window. The tool mainly reports the amount of memory being used by data and code. The memory information can be grouped by module, file or even section and is shown in a table.

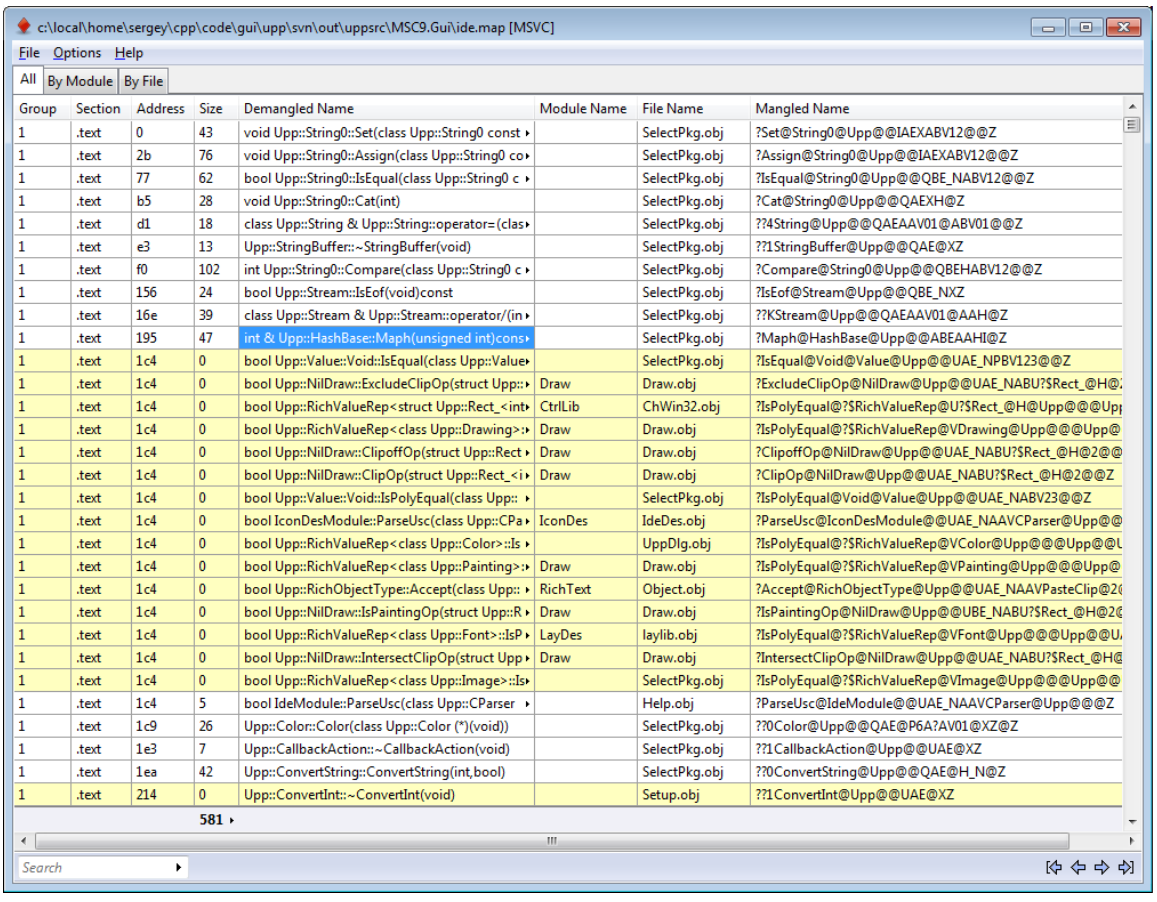

Figure 1.2: Software: AMap [Sik08]

Software link: `http://www.sikorskiy.net/prj/amap/index.html`

1.2.2 StackAnalyzer

StackAnalyzer is a commercial tool focused on the user stack optimization, especially for localizing stack overflows. After reading a *.elf* format file, StackAnalzer starts determining the worst-case scenario of the application. Beside stack analyzing it can visualize a program in a control flow or show the sections in a table with flags (readable, writable, executable, re-/allocated), byteorder and start as well end address.

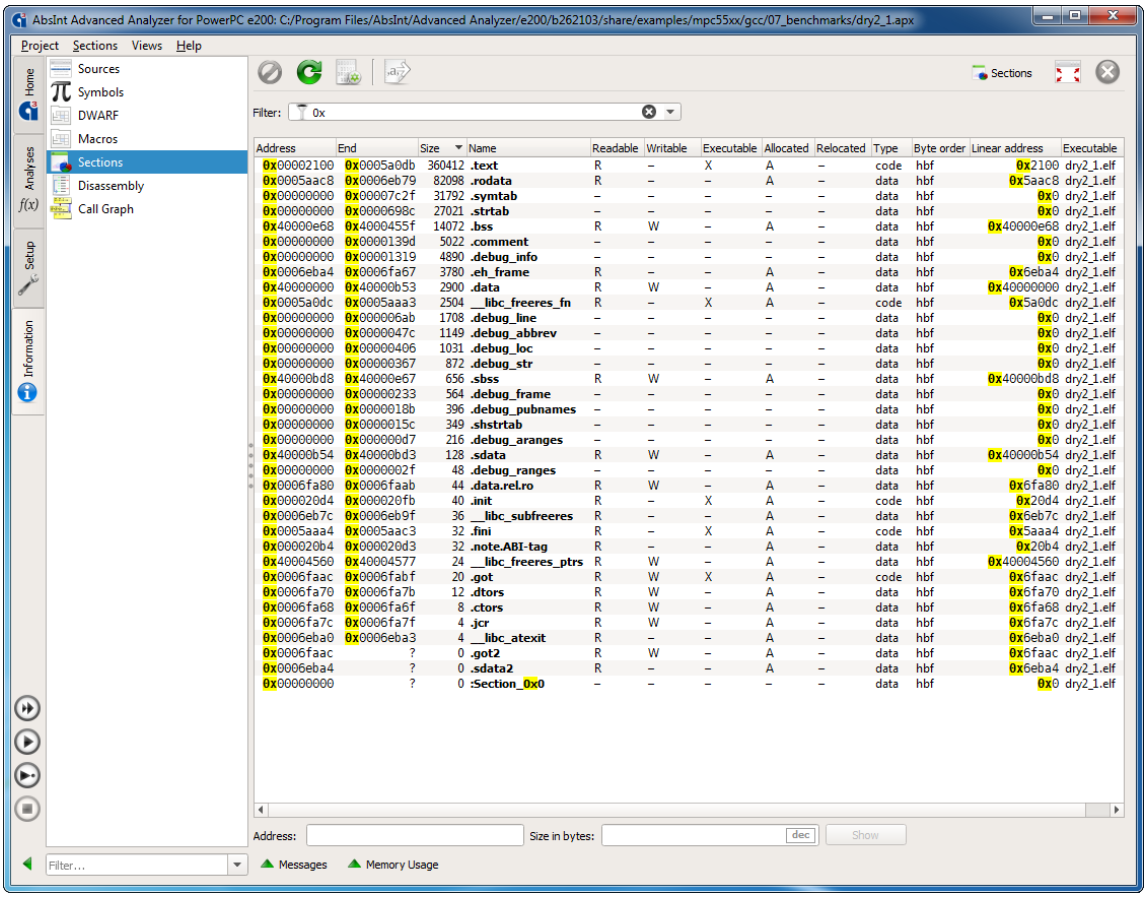

Figure 1.3: Software: Stack Analyzer [Abs98]

Software link: `https://www.absint.com/stackanalyzer/index.htm`

1.2.3 GNU tool-chain collection

GNU tool-chain collection does not even provide a visual graphical-user interface (GUI) that supports parsing *.ld* files to visualize the memory layout. There are only some simple tools provided by GNU toolchain collection, such as

```
objdump -h // dump the sections
objdump -t // dump the symbols
```

Console 1: Different Objdump Flags

Objdump is a tool for displaying information about object files with the file ending *.o*. It can be used as dissembler to show an executable in assembly form.

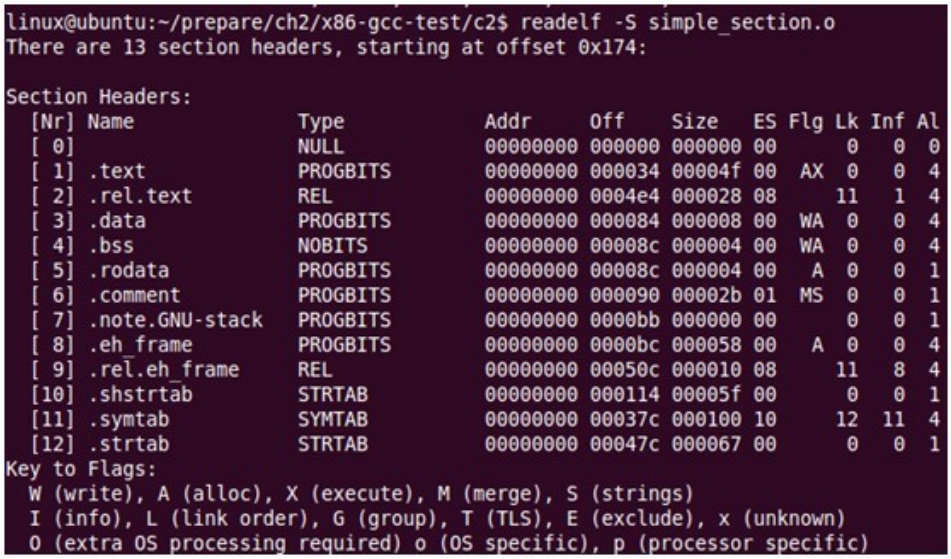

Figure 1.4: Objdump Output

An example output is shown in Figure 1.4, where a list of several sections is printed out in the shell.

Alternatives to objdump:

Readelf, for example, reads the common standard file format ELF for executables, object code, shared libraries and core dumps. The tool can show immediately where the sections (.text, .data) start and end but cannot deliver detailed information about what is inside these sections.

Unlike *readelf*, the tool *nm* only lists symbols from object files, executables including libraries, compiled object modules and shared-object files. *nm* was originally used

```
readelf // dumps elf format
nm      // dumps object files
```

Console 2: Readelf and nm CLI command

for debugging name conflicts, C++ name mangling and is an integral part of the toolchain to validate other parts.

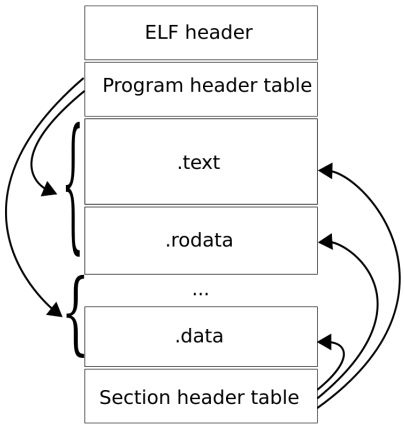

Figure 1.5: ELF Layout

Both tools have in common that they use tables to list the entries, and of course, the most fundamental part of them is a parser. In the next few chapters, some implementation details of a *.ld* parser will be shown.

1.3 Background and Related Work

The motivation of this thesis is to develop a novel approach different to the previously proposed ones to extract the relevant information from linker scripts. The state-of-the-art of the current linker scripts description tools are discussed, but they do not parse linker scripts directly. No related papers have been found.

Except the GNU Binutils [GNU98] is a collection of binary tools and a firm componented of the GNU compilation process, where linking is the last part before generating an executable. Therefore, linker scripts must be parsed and read in order to finish the parsing process. In the git repository of binutils in the subdirectory *'ld/'* the lexer file (*ldlex.l*) and the grammar file (*ldgram.y*) can be accessed.

Two programs are needed to run the parser. One is the lexer for *ldlex.l* file, which splits the source file into tokens and sends it to *lgram.y* to find the hierarchical structure of the program.

The GNU *binutils* uses for lexing the tool *FLex* and for structure checking the tool *BISON* is used. Those tools are the official GNU versions of *Lex* [LS90] and *YACC* [JS90] and can be used as open-source software. *BISON* carries the advantage that no pre-installation of the host system is required.

1.4 Approach

In fact, the number of available linker visualization tools is rather small. After enumerating some different tools, it turns out that the tools do not parse the pure linker script, they parse elf headers or some other object files to filter the relevant information[1] residing in the memory.

Our approach differs, because we want to parse directly linker scripts and save relevant information from the MEMORY section into a data structure. The data structure shall be easily reusable for different further processing purposes, such as a command-line interface (CLI) or GUI. In general, the whole process can be divided into the following steps:

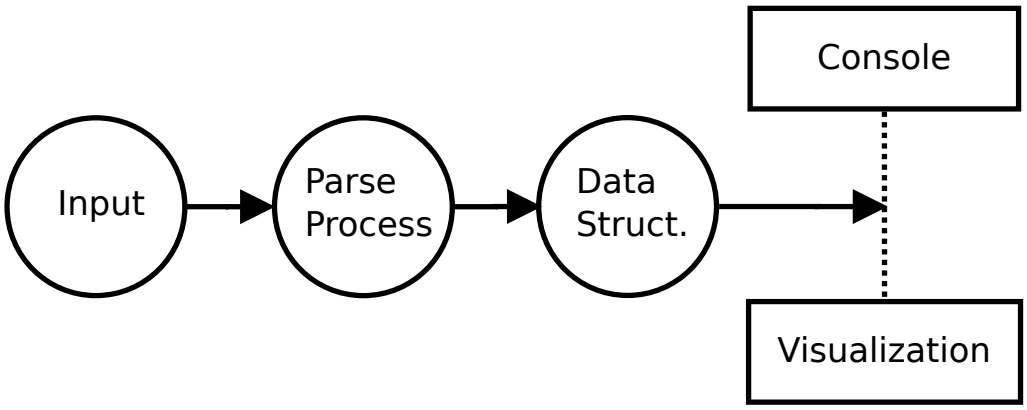

Figure 1.6: Overview from reading an linker script (input), parsing (lexer, parser) to final data structure

Basically, a linker script with the file ending *.ld* shall be read as input. A lexical analysis produces tokens during the parsing process which are forwarded to the parser itself which checks those tokens syntactically. A common structure of a linker script includes several sections like MEMORY, SECTIONS and PHDR. After extracting relevant information during the parsing step the relevant information is saved into a global data structure.

[1] relevant information: all variables and values occurring in the MEMORY section.

The global structure is used temporarily to print relevant information into the console. Moreover, the idea is that this data structure shall serve as the basics for many more applications. In Figure 1.6, example applications are sketched (Console ... Visualization).

2 Practical Implementation

In this chapter the parser is described in more detail. It has been decided to take the *BISON* grammar file and the *Flex* scanner file from *binutils* as template. The *binutils* linker parser is written in C, but for binding this parser to any GUI library, it is better to use an object-oriented (OO) language, like C++.

First of all, the tokens must be declared in a way that the parser can recognize it. Therefore, some rules must be defined:

A grammar is defined as a 4-tuple (V_N, V_T, S, Φ):

- V_N a set of non-terminals.
- V_T a set of terminals (the alphabet of a language).
- $S \in V_N$ the initial state.
- $\Phi = \alpha \rightarrow \beta$ the initial state α describes an arbitrary stringing consisting of terminals and non-terminals that contains at least one non-terminal, this means in formal $(V_N \cup V_T) * V_N(V_N \cup V_T)*$. β is an arbitrary stringing together of terminals and non-terminals including the empty set, also $(V_N \cup V_T)*$.

The program $P_{\mathcal{L}}$ decides for each word w, if the language α contains this word. The program returns true, iff (if-and-only-if) the language contains the word. This is called a **Parser**.

2.1 Project Structure

Figure 2.1 shows the project structure which is divided into three main parts, whereas **thesis** is neglected, because only literature and *latex* files are cotained:

The first one, **buildenv**, contains directories generated by *doxygen*[1]. *Doxygen* generates a documentation in two formats: a latex output and an *HTML* output, what includes sub folders. In chapter 4, in Console 3, the command `make doc` can be used to generate the *doxygen* files.

The **development** part is the major part, where the source code can be found. This part is again divided into parser (source files and the final executable can be found there) and test-benches (including linker scripts as test cases).

[1] `http://www.stack.nl/~dimitri/doxygen/manual/starting.html`

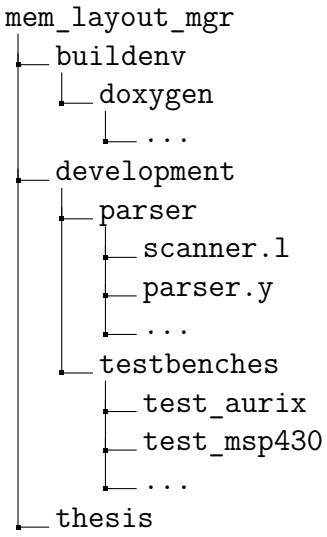

Figure 2.1: Project Directory Tree

2.1.1 Grammar

In general, parsing is the process of matching grammar definition to elements in the input data. In order to understand the further steps there is a short introduction given:

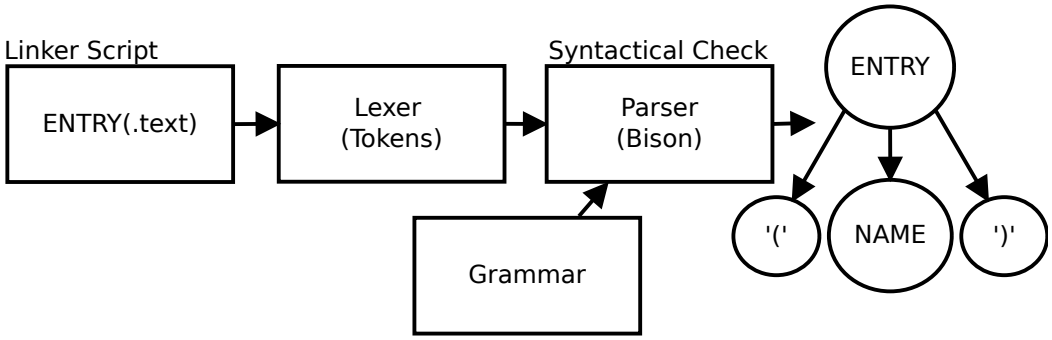

Figure 2.2: Parsing Process from a linker script over tokens generating to a final parsing tree.

In Figure 2.2, the parsing process is illustrated, where the lexer produces tokens from a linker script and sends those tokens to the parser. The parser has only to check if the tokens are in the right order. This is done via an Abstract Syntax Tree (AST). The bison parser is a bottom-up (LL-style parsing) and can work in linear to cubic time [PF11].

The input is divided into terminals (also called tokens). A token is the smallest unit of a character sequence that is significant for a grammar. Tokens are common in languages. Their delimiters are white spaces, commas, etc.

Most programming languages, as well *.ld* formats, are context-free grammars. A context-free grammar is a LL(n)-Grammar, iff:

- No left recursions occur (e.g. $L \rightarrow L\underline{a}$).
- No production with identical prefixes for the same link side exists.
- Only in one step (get to know the next token), it is possible to decide for the next rule.

These properties we can be used to form the grammar. For example, left recursion does not work at all, because it ends in an endless loop.

There are two tasks for the parser: first, check if the input is correct and second, provide the tokens in suitable data structure for later use.

```
MEMORY {
  name (attr) : ORIGIN = origin, LENGTH = len
}
```

Definition 1: Generic Memory Definition [CS94]

To the set of terminals V_T belongs { *MEMORY, ORIGIN, LENGTH, name, attr, origin, len* }. When the parser gets the keyword *MEMORY* it will initiate the AST for *MEMORY*. Then a curled bracket in the beginning and in the end must follow. In between these curled brackets several memory regions declaration can occur. Unless the right tokens are given to the parser, it will fail.

```
1  ifile_p1   : MEMORY '{' memory_spec_list '}' ifile_p1
2  memory_spec_list : memory_spec_list opt_comma memory_spec | ;
3  memory_spec : NAME attributes_opt ':' origin_spec opt_comma length_spec;
4  opt_comma   : ',' | ;
5  origin_spec : ORIGIN '=' NAME;
6  length_spec : LENGTH '=' NAME;
7  attributes_opt  : /* empty */
8                  | '(' NAME ')'
9                  | '(' '!' NAME ')'
10                 | '(' NAME '!' NAME ')';
```

Listing 1: Simplified MEMORY Grammar

The Listing 1 above is just a very simplified version of the original MEMORY grammar, but it shows all important elements. The non-terminals V_N are written in lower-case and serve for connecting the terminals.

The second important step should provide the data during the parsing process. In Figure 2.3, a schematic overview of the parsers implementation is shown.

The parsing process works basically in a way that the singleton class [FFBS04] *ParseManager* works as a central class and has all functions implemented to start the parse process. The *ParserOutput* class involves the callback function of the outcome (relevant information) of the parsing process and is finally stored in a data structure that can be easily accessed.

There are different types to be parsed. A constant term like *ENTRY(symbol)* is saved in a *ParseManager* member variable during the parsing process. Unless it is a constant, a recursion process must be done and the outcome is caught by a callback function. For example, a memory section could have more than one memory region. This is also possible for SECTIONS or PHDRs.

Therefore, an additional data structure must be defined in the header class prototype *AST*. During the parsing process, several loops are made in the grammar until all tokens are processed and that is the point where the relevant information is stored in a list. When the parsing process is finished, the outcome is stored in the data structure accessible by *ParseManager*.

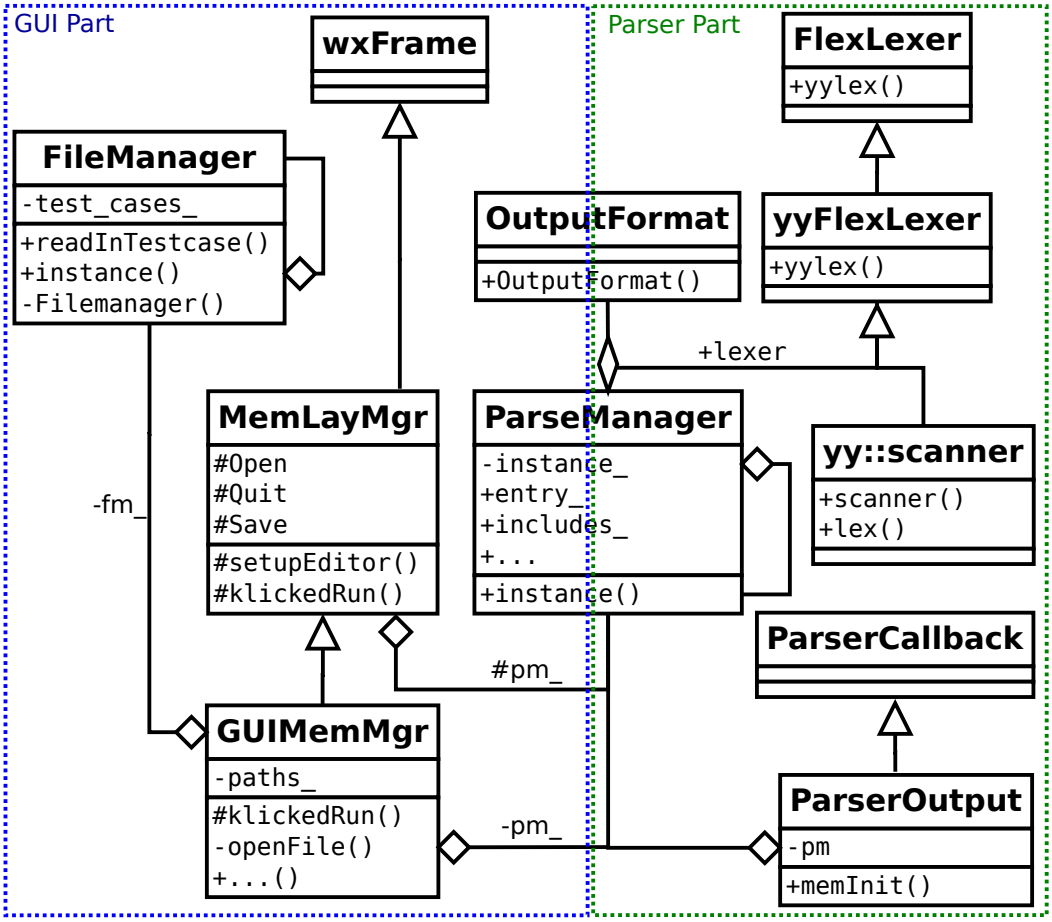

Figure 2.3: Implementation details of the project: The blue area corresponds to the GUI part and the green part represents the parser itself.

In Figure 2.3 the GUI part consists of three classes to provide basic functionality of GUI, i.e. text editor with linker script highlighting, menu bar and buttons. The class *GUIMemMgr* connects the class *FileManager* with the functionality of the GUI, but function stubs are also provided to run the class *ParseManager*. *GUIMemMgr* is inherited from *MemLayMgr* so it is only one instance or in other words, only one window on the screen is shown. If there are several windows are opened, several inheritances of *MemLayMgr* are running.

2.1.2 Includes (multiple input streams)

Commonly, linker scripts consist of several linker files. Each section (MEMORY, SECTIONS, PHDR) can consist of at least one `INCLUDE`, which looks as follows:

```
INCLUDE common.ld  // global

MEMORY {
INCLUDE mem.ld // local, in-memory
}
```

Linker Script 1: INCLUDE command presented as global and in-memory environment.

Global and local `INCLUDE` commands do only differ in the grammar file, where for example in the MEMORY section the `INCLUDE` command is declared.

During the parsing process, the scanner must generate the tokens and forwards them to the parser. This can be done during the token generation. The grammar must be advanced to states. State 0 is the normal mode until the first `INCLUDE` occurs. Then the grammar enters into the next state.

The file, which is specified by the `INCLUDE` command, is opened and read into an input stream for further processing.

The scanner is still in the state 1 and starts to send tokens to the parser. However, the parser is not aware of different states and just does the syntactical check. If all tokens have been sent to the parser the input stream is closed and the scanner continues to the next state.

2.2 Linker Scripts and the Differences

Linker scripts are given to the linker in order to specify the memory layout and to initialize the various memory sections. With memory sections is meant following memory types:

- **Data** segment: contains any global and static variables which are initialized. The values of these variables can be changed during run-time. Example: `static char* var = "some text";`
- **BSS** segment: uninitialized static variables, filled with zeros. Example: `static char* var;`
- **Text** segment: It stores the binary image of the process, which is read-only in many architectures.
- **Stack** segment: stores local variables and parameters pushed onto the user stack.

The GNU linker command language [ST11] is a collection of statements and provides an explicit control over the link process. It specifies the mapping between the linker input files and its output. It handles input files, file formats output file layout, addresses of sections and placement of common blocks.

However, the linker command language is not standardized compared to the ELF [Too01], although the GNU linker command language is the basis for most linker script languages and manufactures (Infineon, etc.) extend the GNU linker command language with their new commands.

2.2.1 AURIX

AURIX (Automotive Realtime Integrated NeXt Generation Architectur) micro controllers from Infineon are 32-bit multi-core TriCore processors. In linker scripts you can access 3 TriCore CPUs [Inf15] separately. According to this feature, for instance, the flags for memory regions must be adapted and some additional commands must be added to our grammar.

- **<name> [(<attr>)]: org = <origin> , l = <len>** Compared to standard definitions, we would have

  `<name> [(<attr>)]: ORIGIN = <origin> , LENGTH= <len>`, where the keywords *ORIGIN* and *LENGTH* can be replaced by *org* and *l*. As an example we choose

  `int_psprcpu0 (rx!c1): org = 0xc0000000, l= 24K`. We can assign memory regions to cores by giving them one of the core flags c0,...,c6. This means, if the flag is `c<nr>`, all code that is located to this memory region will run on the CPU and all previously not exported symbols which are located in this region will be only visible from the CPU.

- **REGION_MAP(<cpu>, <org_local>, <len_local>, <org_global>)** Each core has a physical memory region with a global address and a core local address. The core local data is accessed and global addresses allow to access visible symbols of other cores or global memory.

- **CORE_SYM(symbol)** This command adds a suffix to the symbol name which includes the actual core number. E.g. `CORE_SYM(symbol)` will assign the name `symbol_CPU<N>_`.
- **CORE_SEC(section)** This command is for assigning an object to a core. If you execute the linker script with the command

  `-core=CPU<N>` or in the linker script itself,each output section name that is defined in the relocatable linker script files with the form of `CORE_SEC(.name)` will be named to `CPU<N>.name`.

2.3 Summary

To sum up, linker script language is a context-free language and can be restricted to certain properties that help to write the grammar to divide-and-conquer apparently complicated linker scripts.

The grammar must be adapted to the current situation to not only syntactically check the linker script's content but also to get the information stored in a suitable data structure. This is realized by recursive structures via callback functions.

In the last part of this section, the properties of the AURIX linker script language are investigated. This language now supports controlling each core of the processor separately and consequently new command linker script commands have also been introduced. Hence, we have to extend our current grammar to these new commands and some some commands can already be parsed (see subsection 2.2.1).

3 Future Work

This thesis has introduced a novel approach to parser linker scripts. The basis for a GUI as well function stubs are also provided. An extension of the original *Binutils* grammar file was necessary to extract relevant information and save it into a easily accessible data structure for further processing (Console, GUI visualization, and so on). Moreover, the grammar was extended to some AURIX Tricore syntax from the documentation [Inf15]. The implemented AURIX commands are listed in subsection 2.2.1, but still bunches of commands are not yet implemented.

Nevertheless, there is still a lot of work to be done and this can not be considered as a small project. Linker script command languages vary from different MCU manufacturer and this is the reason why parsing linker scripts are updated with new commands. Hence, the grammar file must also be updated. The advantage of the current software architecture is that the parser can be compiled entirely separate from the GUI or vice versa.

The parser and GUI form the basis for illustrating the memory layout. The vector graphic panel needs to be developed to draw the relevant information residing in the data structure. An example is provided in Figure 3.1. The advantage of vector graphics is that there is no information loss and illustrated relevant information should be exportable to the format PDF.

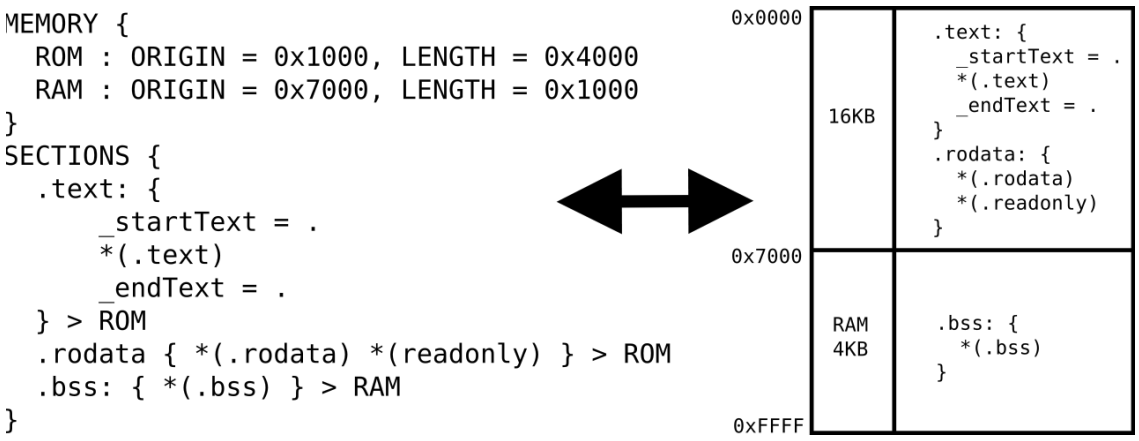

Figure 3.1: Memory Mapping from linker script (left) to a vector graphic (right)

A consistent software structure is necessary to store and reopen projects. The current idea is that files with same names will not be opened again. So the same project

is not opened too many times. On the other hand, a structure for saving files should be planned in future versions. Momentarily, the editor supports that every included file (linker script command: `INCLUDE <filename>`) results in a new editor tab which involves syntax highlighting and some other preset settings.

Concerning the open issue opening linker scripts into tabs. There are 2 possible solutions: The first one is implemented in the current version and uses the *BISON* parser provided idea let the parser via file streams open included files and remember current parsed position and start parsing the included files. After parsing of the included file has ended, the *BISON* parser continues parsing from the last remembered point. This approach is very hard to debug and therefore time consuming.

We are interested to remove the automatic recognition of included files by the parser and shift this task to the class *GuiMemMgr*. When the parser encounters an included file, the *FileManager* saves the included file in a list and opens (following the FIFO principle) the file as well and saves the content of the file as data type string. Then, the *GuiMemMgr* calls the *ParseManager*, which parses the only the opened file, because the opened file is inserted into the current linker script (where INCLUDE command was called) and the parser continuous parsing seamlessly the linker scripts.

4 Appendix

4.1 Software Requirements

- mingw5.2 C++11
- winFlex 2.6 and winBison 3.0.4
- wxWidget 3.1.0

4.2 Graphical User Interface

A GUI skeleton has been designed. The implementation compromises method stubs for the different functionalities listed below such as a "Run" button or menu strip. The following listing corresponds to Figure 4.1 on the next page.

Menu strip

- File: Two sub items are implemented. For example, the functionality for opening a linker script (like in the menu below - file path) or closing a file is provided.
- Help: A new window pops up with a written text about the software requirements.

Run Button and File Path: The functionality of the "Run" button is that a given file in the file path shall be opened. If there exists other includes, they also will also be opened in a new tab.

Text Editor: The text editor is based on *scintilla*[1]. A syntax highlighting for linker scripts languages is not supported and therefore the editor's language support is extended to specific linker scripts commands.

Memory - Vectorized Panel: The purpose of this panel visualization. After parsing the linker scripts, the relevant information is provided and then this information is examined for drawing vector objects in the panel. As a result, a memory block with its sections is visualized.

Constants - Table of relevant information: This table is intended to show the relevant information in the form of a table.

[1] http://www.scintilla.org/SciTE.html

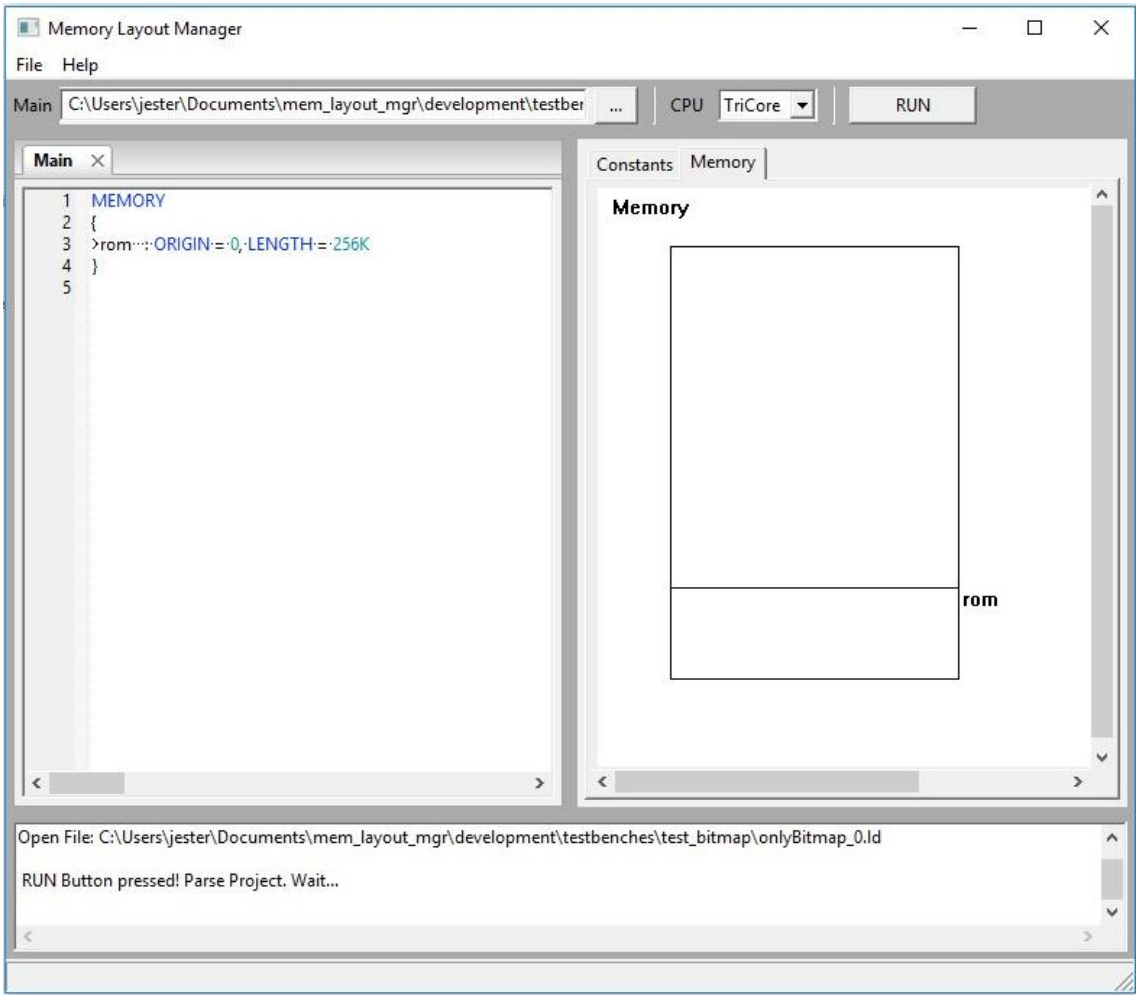

Figure 4.1: GUI - A simple linker script is opened with the correct syntax highlighting.

4.3 Usage

This section gives a short overview on how to execute the binary as well as how to compile the code. First of all, enter the directory `mem_layout_mgr/development/parser/`. Then execute the following in the shell: `make help`. A window with different options will appear (Console 3).

Description

make all: calls the next two make options in the list and compiles it with the compiler of *mingw*. This command makes sure to execute the parsing and scanning in the right order.

make parse: only executes the bison parser command.

make scan: only executes the scanner command.

gram: executes both commands (`make parse` and `make scan`) above, but only if it is not already compiled.

make doc: generates *HTML* files and a *PDF* as documentation.

make run: runs the binary file.

```
**********************************
*** BUILDING MM HELP ****
**********************************
<make all>       ; will call gram
<make parse>     ; bison parser
<make scan>      ; lexer
<make gram>      ; bison lexer
<make doc>
<make run>
```

Console 3: After executing the command: `make` help

Bibliography

[Abs98] AbsInt Angewandte Informatik GmbH. *Stack Analyzer*. https://www.absint.com/stackanalyzer/shot7.png. 1998

[CS94] Chamberlain, Steve ; Support, Cygnus: *Using LD, The GNU Linker*. Version 1.45. UTAH, January 1994. – https://www.math.utah.edu/docs/info/ld_3.html#SEC13

[FFBS04] Freeman, Elisabeth ; Freeman, Eric ; Bates, Bert ; Sierra, Kathy: *Head First Design Patterns*. O' Reilly & Associates, Inc., 2004. – ISBN 0596007124

[GNU98] GNU: *Binutils*. 1998. – https://www.gnu.org/software/binutils/

[Inf15] Infineon Technologies AG: *TriCore Development Platform*, 2015. – Manual

[JS90] Johnson, Stephen C. ; Sethi, Ravi: UNIX Vol. II. Philadelphia, PA, USA : W. B. Saunders Company, 1990. – ISBN 0–03–047529–5, Kapitel Yacc: A Parser Generator, S. 347–374

[LS90] Lesk, M. E. ; Schmidt, E.: UNIX Vol. II. Philadelphia, PA, USA : W. B. Saunders Company, 1990. – ISBN 0–03–047529–5, Kapitel Lex&Mdash;a Lexical Analyzer Generator, S. 375–387

[NTU13] NTU: *GCC and Make Compiling, Linking and Building C/C++ Applications*. 2013. – https://www3.ntu.edu.sg/home/ehchua/programming/cpp/gcc_make.html

[PF11] Parr, Terence ; Fisher, Kathleen: LL(*): The Foundation of the ANTLR Parser Generator. (2011), S. 425–436. ISBN 978–1–4503–0663–8

[Sik08] Sikorskiy, Sergey. *AMap*. http://www.sikorskiy.net/prj/amap/images/amap.05.01.png. 2008

[ST11] Steve, Chamberlain ; Taylor, Ian L.: *The GNU linker, LD (GNU Binutils)*. Red Hat Inc, 2011

[Too01] Tool Interface Standards: *Executable and Linkable Format (ELF)*. Version 1.2, 2001