

CT Praktikum: Modulare Programmierung – Linker

1 Einleitung

In diesem Praktikum lernen Fehler zu beheben bei der Verwendung von fremden Libraries und wie Sie den Debugger dazu bringen, auch in die Library Funktionen hinein zu springen. Zusätzlich lernen Sie die Output Daten des Linkers zu lesen.

Das Projekt besteht aus einer Library mit dazu passenden Header Files.

Sie müssen der Entwicklungsumgebung angeben, wo die Header Files liegen, fehlende Include Direktiven im main.c angeben, und schliesslich angeben, wo der Linker die Library findet.

Danach geht es darum, mit dem erfolgreich gebildeten Programm zu debuggen. Dabei lernen Sie, wie Sie beim Builden zwischen Libraries mit und ohne Debug Information wechseln, und wie Sie dem Debugger sagen, wo er die Sourcen für Source-Line-Debugging findet.

2 Lernziele

- Sie können Compiler und Linker Fehlermeldungen, welche im Zusammenhang mit modularer Programmierung entstehen können, interpretieren und korrigieren
- Sie können Libraries inklusive notwendiger Header Files einbinden
- Sie können ELF Symbol Sections ausgeben und interpretieren
- Sie können Linker Map Files interpretieren

3 Aufgaben

3.1 Compiler und Linker Fehlermeldungen interpretieren und korrigieren

In einem ersten Schritt soll das unfertige Projekt zum Laufen gebracht werden. Dazu müssen die Fehlermeldungen des Präprozessors, des Compilers und des Linkers interpretiert und korrigiert werden.

Siehe dazu die Task-Liste in der Datei *main.c*.

Modulare Programmierung bedeutet für dieses Projekt, dass neben dem Code im *app* Ordner zusätzliche Library Header Files im *. \inc* Ordner liegen. Dieser Ordner muss an geeigneter Stelle in den Projekt Properties (C/C++ Tab) angegeben werden, damit diese Header Dateien vom Präprozessor/Compiler auch gefunden werden.

Analog müssen die einzelnen Libraries im Linker Tab der Projekt Properties unter *Misc Controls* dem *lib* Ordner angegeben werden. Geben sie dazu die benötigte Library in diesem Feld ein (z.B. *lib\read_write.lib*)

Wenn Sie alle Fehler erfolgreich korrigiert haben, können sie das Programm auf das CT-Board laden.

Die Funktion des Programms ist simpel: Wenn Sie auf T0 drücken, wird beim ersten Mal drücken von dunkel auf ein fixes Muster gewechselt, mit jedem weiteren T0 Drücken, werden die LEDs invertiert.

3.2 Debugging

- a) Versuchen Sie das Programm im Debugger in Einzelschritten auszuführen (**Step/F11**). Was beobachten Sie bei der Funktion `read8(BUTTONS)`? (Infos zu Debugging finden Sie auf CT Board Wiki (<https://ennis.zhaw.ch>) unter «Compile and Debugging»)

Was beobachten Sie?

Man kann nicht in die read methode steppen

- b) Ersetzen Sie im Linker Tab die Referenz auf `lib\read_write.lib` mit `lib_debug\read_write.lib`. Was beobachten Sie wenn Sie nun kompilieren, linken und debuggen?
- c) Schliesslich ersetzen sie die Referenz durch `lib_debug_with_src\read_write.lib`. Beim Debugging mit Sourcen müssen Sie dem Debugger zusätzlich noch angeben, wo sich die Source-Files genau befinden. Im Library File selbst befinden sich nämlich keine Source-Files, lediglich die Symbole. Geben sie hierfür im *Command Window* des Debuggers folgenden Befehl ein:

```
set src = C:\<Ihr Pfad>\project\lib_debug_with_src
```

Der Debugger weiss jetzt, wo sich die Source-Files befinden. Achten Sie darauf, dass es im Pfad kein Leerzeichen hat.

Was beobachten Sie wenn Sie nun kompilieren, linken und debuggen?

Was ist der Grund für das veränderte Verhalten?

Jetzt sind die source files die methode verfügbar und es kann c code dargestellt werden statt nur deassembled stuff

3.3 Symbole extrahieren

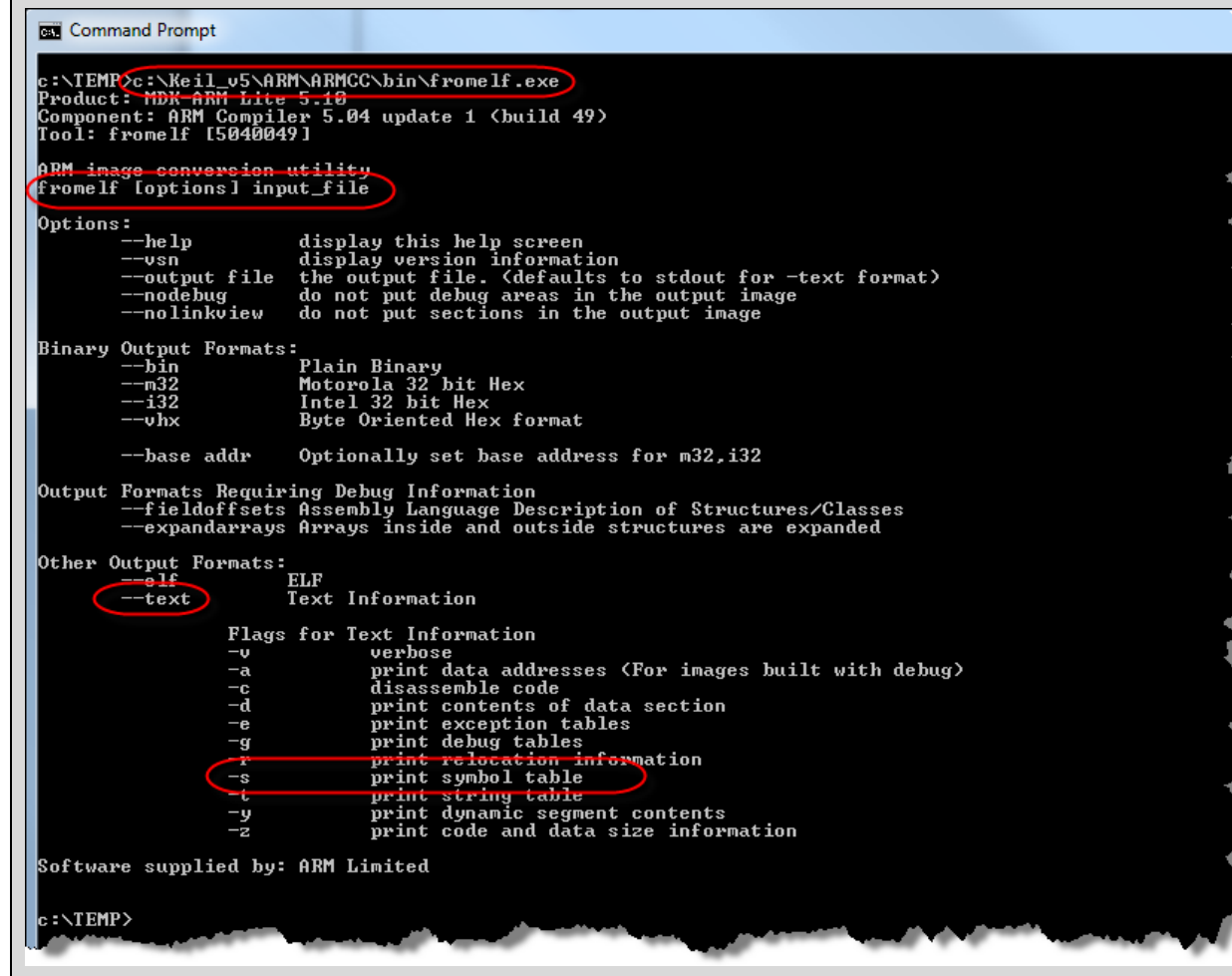
Das Tool *fromelf.exe* kann den Inhalt der binären ELF Dateien in lesbarer Form ausgeben. Die Objekt Dateien (file.o), die Libraries (file.lib) und die Programme (file.axf) sind alle in ELF File Format gegeben.

Führen Sie *fromelf.exe* in einem Command Prompt aus.

Ein Command Prompt öffnen Sie indem sie cmd.exe ausführen.

Das Tool *fromelf.exe* wird über diesen Pfad ausgeführt (Pfad kann abweichen je nachdem wo KEIL installiert wurde): *C:\Keil_v5\ARM\ARMCC\bin\fromelf.exe*.

z.B.



```
CA: Command Prompt
c:\TEMP>c:\Keil_v5\ARM\ARMCC\bin\fromelf.exe
Product: MDK-ARM Lite 5.10
Component: ARM Compiler 5.04 update 1 (build 49)
Tool: fromelf [5040049]

ARM image conversion utility
fromelf [options] input_file

Options:
--help          display this help screen
--vsn          display version information
--output file   the output file. (defaults to stdout for -text format)
--nodebug       do not put debug areas in the output image
--nolinkview    do not put sections in the output image

Binary Output Formats:
--bin          Plain Binary
--m32          Motorola 32 bit Hex
--i32          Intel 32 bit Hex
--vhex         Byte Oriented Hex format

--base addr    Optionally set base address for m32,i32

Output Formats Requiring Debug Information
--fieldoffsets Assembly Language Description of Structures/Classes
--expandarrays Arrays inside and outside structures are expanded

Other Output Formats:
--elf          ELF
--text         Text Information

Flags for Text Information
-v            verbose
-a            print data addresses (For images built with debug)
-c            disassemble code
-d            print contents of data section
-e            print exception tables
-g            print debug tables
-r            print relocation information
-s            print symbol table
-t            print string table
-y            print dynamic segment contents
-z            print code and data size information

Software supplied by: ARM Limited

c:\TEMP>
```

Generieren Sie für *Objects\toggle.o*, *Objects\main.o* und *lib\read_write.lib* die Symbol Tabelle. Fokussieren Sie auf Code und Data Einträge und ignorieren Sie Debug Einträge

Welches sind lokale Symbole?

Welches sind exportierte Symbole?

Welches sind importierte Symbole (referenzierte Symbole)?

Hinweis: Gesucht sind nicht die Namen der einzelnen Symbole, sondern mit welcher Bezeichnung diese in der generierten Tabelle hinten markiert werden.

Lokal	Importiert	Exportiert
Bind: Lc	Sec: Ref	Bind: Gb and no Ref

Vergleichen Sie die obige Antwort mit dem entsprechenden Header File.

Was ist mit den als exportierten Symbolen gemeldeten Einträgen, die nicht im Header File stehen?

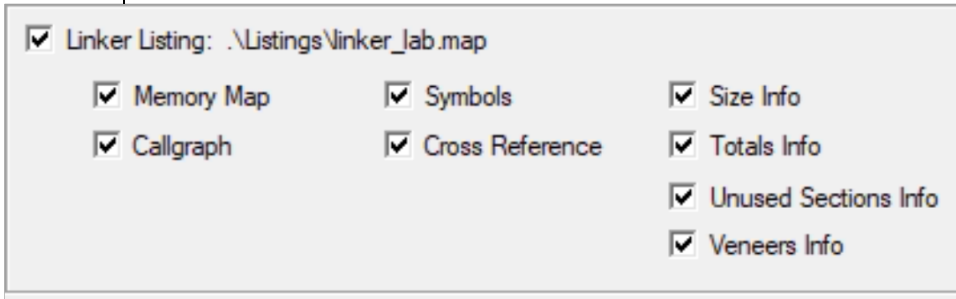
Von einem anderen include?

Wenn Sie eine Library haben und keine dazu passende Header Files, können Sie dann mit dem **fromelf.exe** Tool alle nötigen Informationen aus der Library extrahieren um selber ein Header File zu schreiben? Fehlt etwas?

Der funktionskopf ist unvollständig. Es ist nicht klar welche Parameter übergeben werden müssen. Die implementation ist auch nicht vorhanden

3.4 Linker Map Interpretieren

Beim Linken wird ein Map File kreiert. Prüfen Sie, welche der Informationen unter "Project" → "Options for Target ..." → Tab "Listing" im Map File vorkommen.



Erklären Sie anhand des Linker Map Files, wie das Memory Map aussieht.

Wo sind welche Konstanten, welche Funktionen und welche Daten abgelegt?

Konstanten und Funktionen: Er_RO

Daten: ER_RW

Stack: ER_ZI

3.5 Bewertung

Die lauffähigen Programme müssen präsentiert werden. Die einzelnen Studierenden müssen die Lösungen und den Quellcode verstanden haben und erklären können.

Bewertungskriterien	Gewichtung
Das Programm ist gemäss Aufgabe 3.1 auf dem Board ausführbar.	1/4
Antworten gegeben für Aufgabe 3.2	1/4
Antworten gegeben für Aufgabe 3.3	1/4
Antworten gegeben für Aufgabe 3.4	1/4

Image Entry point : 0x08000229

Load Region LR_1 (Base: 0x08000000, Size: 0x0000097c, Max: 0xffffffff, ABSOLUTE)

Execution Region ER_RO (Exec base: 0x08000000, Load base: 0x08000000, Size: 0x00000978, Max: 0xffffffff, ABSOLUTE)

Exec Addr	Load Addr	Size	Type	Attr	Idx	E Section Name	Object
0x08000000	0x08000000	0x000001ac	Data	RO	29	RESET	startup_ctboard.o
0x080001ac	0x080001ac	0x0000007c	Code	RO	21	.text	datainit_ctboard.o
0x08000228	0x08000228	0x00000024	Code	RO	30	*.text	startup_ctboard.o
0x0800024c	0x0800024c	0x00000004	Code	RO	140	.text	read_write.lib(read.o)
0x08000250	0x08000250	0x00000004	Code	RO	159	.text	read_write.lib(write.o)
0x08000254	0x08000254	0x00000008	Code	RO	35	.text.__system	system_ctboard.o
0x0800025c	0x0800025c	0x0000014c	Code	RO	55	.text.hal_fmc_init_sram	hal_fmc.o
0x080003a8	0x080003a8	0x000000f8	Code	RO	73	.text.hal_gpio_init_alternate	hal_gpio.o
0x080004a0	0x080004a0	0x00000124	Code	RO	71	.text.hal_gpio_init_output	hal_gpio.o
0x080005c4	0x080005c4	0x00000004	Code	RO	111	.text.hal_pwr_set_overdrive	hal_pwr.o
0x080005c8	0x080005c8	0x00000084	Code	RO	123	.text.hal_rcc_reset	hal_rcc.o
0x0800064c	0x0800064c	0x00000074	Code	RO	127	.text.hal_rcc_set_osc	hal_rcc.o
0x080006c0	0x080006c0	0x00000064	Code	RO	131	.text.hal_rcc_setup_clock	hal_rcc.o
0x08000724	0x08000724	0x000000d8	Code	RO	129	.text.hal_rcc_setup_pll	hal_rcc.o
0x080007fc	0x080007fc	0x00000024	Code	RO	2	.text.main	main.o
0x08000820	0x08000820	0x0000013c	Code	RO	37	.text.system_enter_run	system_ctboard.o
0x0800095c	0x0800095c	0x0000001c	Code	RO	13	.text.toggle	toggle.o

Execution Region ER_RW (Exec base: 0x20000000, Load base: 0x08000978, Size: 0x00000004, Max: 0xffffffff, ABSOLUTE)

Exec Addr	Load Addr	Size	Type	Attr	Idx	E Section Name	Object
0x20000000	0x08000978	0x00000001	Data	RW	15	.data.value	toggle.o

Execution Region ER_ZI (Exec base: 0x20000008, Load base: 0x0800097c, Size: 0x00002008, Max: 0xffffffff, ABSOLUTE)

Exec Addr	Load Addr	Size	Type	Attr	Idx	E Section Name	Object
0x20000008	-	0x00000001	Zero	RW	4	.bss.last	main.o
0x20000009	0x0800097c	0x00000007	PAD				
0x20000010	-	0x00002000	Zero	RW	27	STACK	startup_ctboard.o