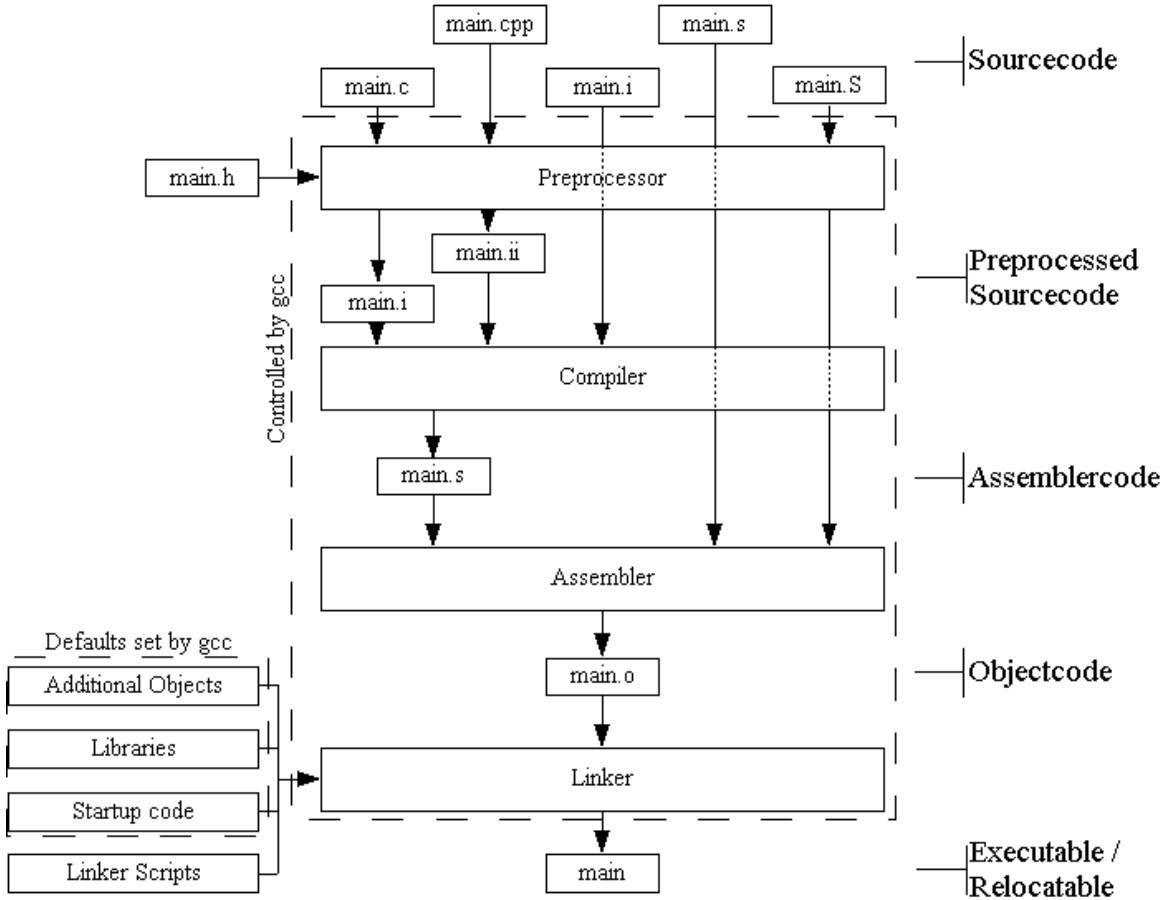
Power User Training

HighTec EDV-Systeme GmbH

Feldmannstraße 98
66119 Saarbrücken
www.hightec-rt.com

1. Entwicklungslauf
   - Preprocessor
   - Compiler
   - Assembler
   - Linker
2. Zusätzliche Werkzeuge
   - tricore-objcopy
   - tricore-objdump
   - tricore-nm
   - tricore-ar
3. Neue Features
   - Erweiterungen durch HighTec
   - GNU-Projekt
   - Ausblick

Überblick


main.cpp
main.s
main.c
main.i
main.S
Sourcecode
main.h
Preprocessor
main.ii
Preprocessed Sourcecode
main.i
Controlled by gcc
Compiler
main.s
Assemblercode
Assembler
Defaults set by gcc
Additional Objects
main.o
Objectcode
Libraries
Linker
Startup code
Linker Scripts
main
Executable / Relocatable

Compiler Driver tricore-gcc

- Vereinfacht Build Prozess durch:
  Einbindung aller benötigten Werkzeuge (cpp, cc1, as, ld) für jede Datei, abhängig von entsprechender Dateiendung
- Übergabe spezifizierter Optionen an jeweilige Werkzeuge
- Übergabe von Default Optionen an jeweilige Werkzeuge

  $\Rightarrow$ Konfiguration über `specs` Datei möglich

  $\Rightarrow$ Benutzerdefinierte Konfiguration in `tricore.specs`

Dateitypen

`file.c`	C Quellcode
`file.cc, file.cxx, file.cpp, file.C`	C++ Quellcode
`file.h`	C Headerdatei
`file.i`	Preprocessed C Quellcode
`file.ii`	Preprocessed C++ Quellcode
`file.s`	Assembler code
`file.S`	Assembler code muss Preprocessor durchlaufen

Prepocessor

- Integriert im Compiler → Verwendung Option `-E`
- Einbindung von Header Files
- Auflösen von Makros
- Überprüft Syntax von Preprocessor Direktiven
- Varianten durch konditionale Compilierung
- Ersetzen von Kommentaren durch single space
- Löschen von backslash-newline Sequenzen
- Line control

Wichtige Optionen

`-I <dir>` Setzt <dir> ans Ende des Include Pfads

`-D <macro>` Definiert <macro> als wahr (1)

`-D <macro>=<value>`
Weist <macro> den Wert <value> zu

`-U <macro>` Undefine <macro>

`-dM` Generiert Liste aller definierten Makros und Werte

`-imacros <file>`
Bevor input files durch den Preprocessor gehen, wird <file> nach Makros durchsucht

`-include <file>`
Übergabe des Inhalts von <file> an Preprocessor bevor #include Direktiven aus input file ausgewertet werden

`-Wall` Aktiviert alle optional verfügbaren Warnings

Wichtige Optionen

`-mwarn-skipped-cpp-directives`
: Ausgeben einer Warnung bei ungültigen Preprocessor Direktiven

`-mno-warn-skipped-cpp-directives`
: Default Option

Default Optionen

- -Acpu(tricore)
- -Amachine(tricore)
- -D__GNUC__=3
- -D__GNUC_MINOR__=3
- -D__GNUC_PATCHLEVEL__=0
- -D__SOFT_FLOAT__
- -Dtricore -D__tricore__ -D__tricore
- -DRIDER_B -D__TC12__ -D__RIDER_B__
- `-iprefix`
- `-iwithprefixbefore`

GNU Compiler

- Übersetzt C/C++ Quellcode in Assembler Befehle
- Optimierung Codegröße und/oder Ausführungszeit (optional)

  `-O0` Deaktivieren aller Optimierungen (Stufe 0)

  `-O1` Compiler versucht Codegröße und Ausführungszeit mit 'einfachen Mitteln' zu optimieren (Stufe 1)

  `-O2` Aktivierung aller Optimierungsalgorithmen, die sich nicht negativ auf die Ausführungszeit oder Größe auswirken (Stufe 2)

  `-O3` Höchste Optimierungsstufe (Stufe 3) aktiviert function inlining
- Generierung von Debug Information (optional)

Wichtige Optionen

`-E`	Nur Preprocessor-Lauf
`-S`	Preprocessor und Compile-Lauf
`-c`	Durchlaufen von Preprocessor, Compiler und Assembler
`-o <outfile>`	Schreibt output in <outfile>
`-v`	Anzeige der Version der eingebundenen Tools und deren Parameter
`-O[n]`	Aktiviert Optimierung (n = 0, 1, 2, 3)
`-Os`	Optimiert Codegröße
`-gdwarf-2`	Generiert DWARF 2.0 debug information
`-Wall`	Erzeugt 'alle' Warnings

TriCore spezifische Optionen (Teil I)

`-mabs=<N>` Legt Variablen deren Größe $\leq$ N in absolute addressable area (.zbss)

`-mabs-data=<N>,-mabs-const=<N>` Legt Daten/Konstanten deren Größe $\leq$ N in absolute addressable area.

`-msmall=<N>` Legt Variablen deren Größe $\leq$ N in small data area (.sdata und .sbss)

`-msmall-data=<N>,-msmall-const=<N>` Legt Daten/Konstanten deren Größe $\leq$ N in small data area.

`-mtc12` Generiert code für TriCore v1.2

`-mtc13` Generiert code für TriCore v1.3

`-mcpu<N>` Aktiviert workaround für Hardware Bug (<N> = 9, 13, 18, 24, 31, 34, 48, 50, 60, 70, 72, 76, ...)

TriCore spezifische Optionen (Teil II)

`-mhard-float`
: Verwendet floating instructions (TC1v1.3)

`-moptfp`
: Verwendet optimierte single float Emulation

`-msoft-fdiv`
: Verwendet Software Emulation für floating point Divisionen

`-mwarnprqa=on|off`
: Warnings für QAC pragmas einstellbar

TriCore spezifische Optionen (Teil III)

`-masm-source-lines`
: C-Sourcecode als Kommentar im Assemblercode

`-mversion-info`
: Erstellt Sektion `.version_info`

`-maligned-data-sections`
: Erzeugt Subsections `.a1` `.a2` `.a4` `.a8` abhängig vom Alignment

`-maligned-access`
: Aktiviert Attribut `alignedaccess`

TriCore spezifische specs-Datei

Optionen

`-mcpu=<CPU>`

Setzt die verwendete CPU

`-mcpu-specs=<Datei>`

Ersetzt das Standard-`tricore.specs`-File

Auszug

```
*tc1796_errata:
-mcpu48=1 -mcpu60 -mcpu70 -mcpu72 -mcpu76 -mno-all-errata
...
*TC1796:
    -mtc13 %(tc1796_errata) -mhard-float
```

Default Optionen

- -fargument-alias
- -fbranch-count-reg
- -fcommon
- -ffunction-cse
- -fgcse-lm
- -fgcse-sm
- -fgnu-linker
- -fident
- -fkeep-static-consts
- -fmath-errno
- -fomit-frame-pointer
- -fpeephole
- -freg-struct-return
- -fsched-interblock
- -fsched-spec
- -ftrapping-math
- -funsigned-bitfields
- -fzero-initialized-in-bss
- -mtc12

Assembler Aufgaben

- Übersetzt mnemonics in Maschinen Code
- Teilung von Code/Daten ($\rightarrow$ sections)
- Auflösen lokaler Referenzen
- Generiert Relokationseinträge für externe Referenzen
- Generiert eine linkfähige Objektdatei

Wichtige Optionen

`-o <objfile>`
: Verwendt <filename> anstatt 'a.out' als Ausgabedatei

`--defsym <symbol>=<value>`
: Definiert das Symbol <sym> mit <value>

`--gdwarf2`
: Generiert DWARF2 Debug Information für jede Assemblerzeile

`-I <dir>`
: Fügt <dir> in Suchpfad von Assembler hinzu

`-a[opts][=file]`
: Schreibt Listing in angegebene Datei (Option -g erforderlich)

`-masm-source-lines`
: Intermix von C-Source und Assembler Output

TriCore spezifische Optionen

`-mtc12, -mtc13, -mtc2`
: Assembliert für TC1v1.2, TC1v1.3, TC2 Befehlssatz

`-mcpuN`
: Aktiviert workaround für Hardware Bug (<N> = 9, 34, 48, 50, 60, 70, 72)

`--dont-optimize`
: Unterbindet Optimierung von mnemonic Befehlen (Ausnahme bei .optim Befehl)

`--insn32-only`
: Nur 32-bit opcodes verwenden (.code16 und .optim werden ignoriert)

`--insn32-preferred`
: Wie `--insn32-only` berücksichtigt aber .code16

`--enforce-aligned-data`
: Alignment abhängig von Größe einer Variablen (sizeof)

Assembler Syntax

```
.pseudo_opcode__name [option(s)]
```

- Definition / Angabe Sektionen
- Definition von Konstanten (integer, float, string)
- Definition von Labels / Symbolen, deren Gültigkeitsbereich und Typangabe
- Aktivieren / Deaktivieren von speziellen Optionen / Quellcode-Behandlung

TriCore spezifische Pseudo-Opcodes

`.code16`	Verwendet 16-bit opcode für nächsten Befehl
`.code32`	Verwendet 32-bit opcode für nächsten Befehl
`.optim`	Versucht nächsten Befehl zu optimieren
`.noopt`	Unterdrückt Optimierung für nächsten Befehl
`.pcptext,.pcpdata`	PCP Unterstützung

Pseudo-Opcodes für Bit-Variablen

`.bit <bname>[,bexpr]`

Erzeugt globale Bit-Variable <bname> und initialisiert optional mit dem Wert (0 oder 1) des absoluten Ausdruck bexpr. Zugriff über Symbolname und Bit-Position über Prefix bpos:

```
.bit foo;
st.t foo,bpos:foo,1
```

`.lbit <bname>[,bexpr]`

Im Unterschied zu .bit nur im aktuellen Modul sichtbar (local scope)

`.bpos, .bposb, .bposh, .bposw`

Übernimmt Name der Bit-Variable als Argument und gibt Bit-Position in zugehöriger Sektion aus

Pseudo-Opcodes für PCP

- GNU Assembler unterstützt mehrere PCP per Chip durch explizite Aufgliederung in PCP sections (→linker script)
- Der Startup code `crt0.S` kopiert PCP Code und Daten Sektion in zugehörige Speicherbereiche

`.pcptext` Zugriff auf PCP text Sektion. Somit erlaubt Assembler Verwendung PCP mnemonics anstatt TriCore Befehle (Halfword Zugriff)

`.pcpdata` Wordweiser Zugriff auf PCP Daten Sektion (z.B. Ablage für Parameter)

`.pcpinitword` `.pcpinitword initPC, initDPTR, initFLAGS` erzeugt einen 32-bit Wert, der zum Initialisieren von PCP Register R7 verwendet werden kann

Prefixes zur Bestimmung vom Relokationtyp

Prefix Beschreibung

Prefix	berechnet
hi:	((sym_or_expr + 0x8000) >> 16)
lo:	(sym_or_expr & 0xFFFF)
sm:	(16−bit offset into SDA)
up:	((sym_or_expr >> 16) & 0xFFFF)
bpos:	Bitposition .bit foo; st.t foo,bpos:foo,1

Default Optionen

- `-mtc12`
- `-o <file>`

Linker Aufgaben

- Bindet verschiedene Objekte
- Kombiniert Archiv Dateien
- Reloziert die Daten
- Löst Symbol Referenzen auf
- Stellt zusätzliche Diagnostikinformation zur Verfügung
- Im Mapfile detaillierter Überblick
  - Ablage der Objektdateien und Symbole
  - Common symbols zugewiesen werden
  - Eingefügte Archive

Wichtige Optionen (Teil I)

`-o <file>, --output <file>`
: Ausgabedatei des Linker in <file> schreiben

`-M, --print-map`
: Schreibt Mapfile auf Standard Output

`-Map <file>`
: Schreibt die Mapfile Information in Datei <file>

`--cref`
: Erzeugt eine Tabelle mit Querverweisen, die mit `-Map` in die Datei eingefügt wird

`--defsym <symbol>=<expression>`
: Erzeugt ein globales Symbol in der Ausgabedatei, mit der absoluten Adresse <expression>

Wichtige Optionen (Teil II)

`-R <file>, --just-symbols <file>`
: Liest aus <file> Symbolnamen und deren Adresse, ohne sie in die Ausgabedatei einzufügen (Referenz auf Symbol/Shared Libraries)

`-r, -i, --relocateable`
: Durchführen eines inkrementellen Linkerlauf (erzeugt relokatierbare Ausgabe)

TriCore spezifische Optionen

`-extmap=<output-option>`
: Generiert ein 'extended map file' mit zusätzlicher Information (Option `--Map` oder `-M` erforderlich)

`--warn-orphan`
: Erzeugte eine Fehlermeldung, wenn keine feste Abbildung zwischen einer Input und Output Sektion besteht

`--relax-24rel`
: Relax call und jump Befehle deren Zieladresse weder durch PC-relativen Offset noch durch absolute Adressierung erreicht werden kann

`--relax-bdata`
: Komprimiert bit Objekte aus input Sektion `.bdata`

`--relax`
: Impliziert Optionen `--relax-24rel` und `--relax-bdata`

Default Optionen / Argumente

Optionen

- `-L <library path>`
- `-lgcc`
- `-lc`
- `-los`
- `-lc`
- `-lgcc`

Argumente

- Startup Code `crt0.o`

Binutils (Teil I)

tricore-ar

Erzeugt, modifiziert und extrahiert aus Archiven

tricore-ranlib

Generiert einen Index für Archiv Inhalt

tricore-nm

Auflistung von Symbolen aus Objektdatei

tricore-objcopy

Kopiert und Übersetzt Objektdateien in verschiedene Formate

tricore-objdump

Zeigt verschiedene Informationen aus Objektdatei an

Binutils (Teil II)

tricore-addr2line

Konvertiert Adressen in Dateinamen und Zeilennummer

tricore-readelf

Anzeige des Inhalts einer ELF Format Datei

tricore-size

Auflistung von Datei-Sektionsgröße und absolute Größe

tricore-strings

Auflistung der Strings aus einer Datei

tricore-strip

Entfernen von Symbolen

Wichtige Optionen für tricore-objcopy

`-O --output-target <bfdname>`
: Erzeugt eine Ausgabedatei im <bfdname> Format

`-j --only-section <name>`
: Kopiert nur die ausgewählte Sektion der Eingabedatei in die Ausgabedatei

`-R --remove-section <name>`
: Entfernt Sektionen mit Name <sectionname> aus der Ausgabedatei

`--add-section <sectionname>=<filename>`
: Fügt eine neue Sektion <sectionname> aus <filename> in die Datei

`--rename-section <old>=<new>[,<flags>]`
: Umbenennen von Sektionen und optional ändern von <flag>

Beispiele

Intelhex Format

```
tricore-objcopy -O ihex Input.elf Output.hex
```

Erzeugung einer Binär-Datei

```
tricore-objcopy -O binary Input.elf Output
```

Entfernen der Debug Sektion

```
tricore-objcopy -R .debug_info Output.elf
```

Wichtige Optionen für tricore-objdump

`-D, --disassemble-all`
: Zeigt die Assembler mnemonics für Maschinebefehle an

`-h, --section-headers`
: Überblick der Sektion Headers aus der Objektdatei

`-t, --syms`
: Zeigt Einträge für Symboltabelle an

`-S, --source`
: Zeigt Quellcode gemischt mit zugehörigem Assembler Output an (Impliziert Option `-d`)

Beispiele HelloSerial

Anzeige der Sektion Headers

```
tricore-objdump -h triuart.o
```

```
triuart.o:     file format elf32-tricore
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000176  00000000  00000000  00000034  2**1
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 .data         00000010  00000000  00000000  000001b0  2**3
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000000  00000000  00000000  000001c0  2**3
                  ALLOC
  3 .debug_abbrev 00000112  00000000  00000000  000001c0  2**0
                  CONTENTS, READONLY, DEBUGGING
  4 .debug_info   00000673  00000000  00000000  000002d2  2**0
                  CONTENTS, RELOC, READONLY, DEBUGGING
...
```

Wichtige Optionen für tricore-nm

`-f, --format=<format>`
: Verwendet als Ausgabeformat <format> ('bsd','sysv' oder 'posix')

`-g, --extern-only`
: Zeigt nur externe Symbole an

`-n, --numeric-sort`
: Sortiert Symbole nach Adressen

`--size-sort`
: Sortiert Symbole nach Größe

`-u, --undefined-only`
: Zeigt nur nicht definierte Symbole an (Zugriff auf externe Objektdatei)

Beispiel HelloSerial

Anzeige externer Symbole

```
tricore-objdump -g triuart.o
```

```
00000126 T _in_uart
00000060 T _init_uart
00000102 T _out_uart
0000014c T _poll_uart
00000000 T get_cpu_frequency
         U lock_wdtcon
         U unlock_wdtcon
```

Anzeige nicht definierter Symbole

```
tricore-objdump -u triuart.o
```

```
lock_wdtcon
unlock_wdtcon
```

Wichtige Optionen für tricore-ar

`d`	Löscht aus Archiv das Modul mit Namen <member>
`m[ab]`	Verschiebt Teile in einem Archiv
`q[f]`	Schnelles Hinzufügen durch Anfügen der Dateien <member> in Archiv (Überprüft nicht auf gleiche Name)
`r[ab][f][u]`	Fügt die Dateien <member> ... durch Ersetzung in das Archiv (Prüfung der Namen)
`x[o]`	Extrahiert Teil <member> aus dem Archiv

Beispiel

Extrahieren von Modulen aus einem Archiv

```
tricore-ar -x <name>.a
```

Löschen eines Moduls aus einem Archiv

```
tricore-ar -d <name>.a <member>.o
```

Anfügen/Ersetzen eines Moduls aus einem Archiv

```
tricore-ar -r <name>.a <member>.o
```

Erweiterungen durch HighTec

- Bitdatentyp
- Multiple Bit-Data Sections
- Pragma Sections
- Dichte Ablage von Daten
- Relative Addressing
- Circular Addressing
- Constants in Read Only Section
- Mapping of Default Sections
- CSA Overhead
- Indirect Addressing bei longcall
- Position Independent Code
- Source Lines in Assembler Output

Derivative Specs File

- Derivat spezifische Einstellungen (z.B. Errata)

```
...
*tc1775_errata:
-mcpu18 -mcpu24 -mcpu31 -mcpu34 -mcpu48 -mcpu50
...
*TC1775:
    -mtc12 %(tc1775_errata)
...
```

- Tags beginnen mit $*$
- Schnittstelle für eigene Konfiguration

Konfiguration

Beispiel

- Mit %(tc1775_errata) Inhalt von *tc1775_errata ersetzt
- Zugriff auf Tag *TC1775 mit

```
tricore-gcc -mcpu=TC1775 ...
```

entspricht

```
tricore-gcc -mtc12 -mcpu18 -mcpu24 -mcpu31 -mcpu34
-mcpu48 -mcpu50 ...
```

Hinweis

Mit `-mcpu-specs=<file>` lässt sich eigene Konfigurationsdatei <file> angeben.

Indirect Addressing by longcall

Hintergrund

- Memory Mapping bei TC1796
- Zugriff von extern auf internes RAM über indirekte Adressierung
- Internes RAM schneller in der Ausführung

  $\Longrightarrow$ Häufig verwendete Funktionen in internes RAM mappen
- Einführung eines Attributs longcall

Attribut longcall

longcall

Mit dem Funktionsattribut longcall können beliebige Funktionen mit calli aufgerufen werden. Der Code und die Ausführungszeit werden verringert.

```
extern void func02(void) __attribute__((longcall));
void func01(void) __attribute__((longcall));

void func01(void)
{
/* do something */;
}
```

CSA Overhead

Context Save/Restore

Wenn eine jump oder link Instruktion anstatt eines call verwendet wird, erzeugt ein interrupt handler einen context save and context restore Prozess.

- Sicherung Upper Context (in CSA) bei Interrupt und Trap
  - PSW (Processor Status Word)
  - A10 to A15 (Address Register)
  - D8 to D15 (Data Register)
- Wiederherstellen upper context nach ret oder rfe

TriCore bietet alternative jump and link instruction (jl , jla , jli)

$\Rightarrow$ Diese Instruktionen verwenden die Return-Adresse aus %a11

Beispiel

`interrupt` Eine jump indirect Instruktion `ji %a11` verwenden

`interrupt_handler`
Funktion kehrt mit `rfe` zurück

```
#include <machine/cint.h>
extern void ifoo(int) __attribute__ ((interrupt));
extern void ihfoo(void) __attribute__ ((interrupt_handler));

int lf, nf, iif, ihf;
unsigned int * IntSrc = (unsigned int *)0xf7e0fffc;
int main(void)
{
 _install_int_handler(3,ifoo,0);
...
 ifoo(1); /* verwendet ji %a11*/
 *IntSrc = 0x1001;
 __asm__ volatile ("enable");
 *IntSrc |= 0x8000;
...
```

Ausblick für GCC 4.0

Allgemeine Optimierungsverbesserungen

- Bedienbarkeit von Profile Feedback und Coverage Test ist verbessert
- Inlining heuristics für C, Objective-C, C++ deutlich verbessert. Call graph basierend auf out-of-order inlining ist jetzt durch Option -O2 aktiviert
- Verbesserte Optimierungsstrategien
- Globale Optimierung Module

Ausblick für GCC 4.0

SSA

Beinhaltet zwei high-level intermediate Sprachen (GENERIC and GIMPLE).

- Scalar replacement of aggregates
- Constant and Value range propagation
- Partial redundancy elimination
- Load and store motion
- Strength reduction
- Dead store elimination
- Dead and unreachable code elimination
- Autovectorization
- Tail recursion by accumulation
- Loop interchange

4 Adressierungsarten
- Normale Adressierung
- Registerrelative Adressierung
- Absolute Adressierung

5 Inline Assembler
- Assembly Language Template

6 Attribute und Pragmas
- Attribute für die Adressierung
- Attribute für die Datenablage
- Attribute für Sektionen
- Pragmas

7 Bit-Datentyp
- Einführung
- Linken
- Mapfile
- Bitfelder

Normale Adressierung

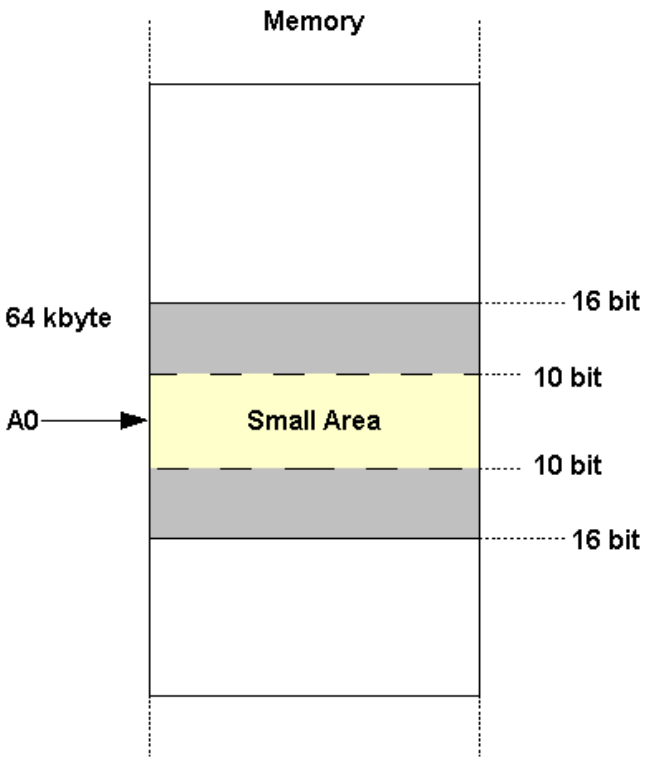
Quellcode

```
c1 = foo;
```

Generierter Code

```
movh.a %a15, HI:foo
lea %a15, [%a15] LO:foo
ld.b %d15, [%a15] 0
```

Registerrelative Adressierung



- Adressierung relativ zu Basisadresse in Register
- Adresse wird durch Addition / Subtraktion von Offset und Basisadresse gebildet
- 10 oder 16 Bit Offset
  - 10 Bit Offset für alle Memory Operationen verfügbar
  - 16 Bit Offset für viele Memory Operationen verfügbar

Deklaration von Variablen

- Attribut asection oder pragma section
- Sektionsname .sdata oder .sbss
- Eigene Sektionen fangen mit .sdata. oder .sbss. an
- Flag t für 10bit, Flag s für 16bit

```
char c __attribute__((asection \
  (".sdata.byte", "a=1", "f=awt")));
```

oder

```
#pragma section .sdata.byte 1 awt
char c;
#pragma section
```

Small adressierbare Sektionen

- Vier small data Sektionen im Executable möglich
- Adressiert über verschiedene Register

Name	Register
.sdata / .sbss	A0
.sdata2 / .sbss2	A1
.sdata3 / .sbss3	A8
.sdata4 / .sbss4	A9

Linker Script I

- Output Sections .sdata/.sbss im default Linker Script vorgesehen
- Wenn Output Sections .sdataX/.sbssX nicht im Linker Script definiert wurden, legt der Linker sie an, wenn Small Data Area $> 64k$
- .sbss und .sbss.* per default in Output Section .sbss
- .sdata und .sdata.* per default in Output Section .sdata
  → .sdataX/.sbssX vor .sdata/.sbss im Linker Script
- Startupcode kopiert alle Daten und initialisiert .sbss mit 0
  → Sektionen .sdataX müssen in __copy_table aufgeführt sein
  → Sektionen .sbssX müssen in __clear_table aufgeführt sein

Linker Script II

- Jede Small Data Area hat ein Symbol
- Default: Symbole entsprechen Anfangsadresse + 32768
- Symbole werden automatisch gesetzt, wenn nicht im Linker Script definiert

Symbol	SDA
_SMALL_DATA_	.sdata/.sbss
_SMALL_DATA2_	.sdata2/.sbss2
_SMALL_DATA3_	.sdata3/.sbss3
_SMALL_DATA4_	.sdata4/.sbss4

Beispiel

Ziel

- Variablen in verschiedenen small adressierte Sektionen definieren

Vorgehensweise

1. Variablen deklarieren
2. Zusätzliche Outputsection .sdata2 definieren
3. Zusätzliche Outputsection .sbss2 definieren
4. __copy_table anpassen
5. __clear_table anpassen

Variablen

```
char c1 __attribute__((section (".sdata")));

char c2 __attribute__((asection \
  (".sdata2.byte", "a=1", "f=awt")));

int i1 __attribute__((asection \
  (".sdata.int", "a=4", "f=aws")));
```

Variable	O-Section
c1	.sdata
c2	.sdata2
i1	.sdata

Original Linker Script File

```
.sdata : AT(LOADADDR(.data) + SIZEOF(.data))
{
  . = ALIGN(8) ;
  SDATA_BASE = ABSOLUTE(.) ;
  PROVIDE(__sdata_start = .);
  *(.sdata)
  *(.sdata.*)
  . = ALIGN(8) ;
} > ext_dram
.sbss ALIGN(8) (NOLOAD) :
{
  PROVIDE(__sbss_start = .);
  *(.sbss)
  *(.sbss.*)
} > ext_dram
```

Output Section .sdata2

```
.sdata2 : AT(LOADADDR(.data) + SIZEOF(.data))
{
  SDA2_BASE = ABSOLUTE(.) ;
  . = ALIGN(8) ;
  *(.sdata2.byte)
  . = ALIGN(8) ;
} > ext_dram
. = ALIGN(8) ;
_SMALL_DATA2_ = SDA2_BASE + 512;

.sdata : AT(LOADADDR(.sdata2) + SIZEOF(.sdata2))
  ...
```

Output Section .sbss2

```
.sbss2 (NOLOAD) :
{
  . = ALIGN(8) ;
  *(.sbss2.byte)
  . = ALIGN(8)
} > ext_dram

.sbss ALIGN(8) (NOLOAD) :
  ...
```

Mapfile

Ausgabe der Ergebnisse im Mapfile (gekürzt)

```
.sdata2 0xa0080718 0x1
 *(.sdata.byte)
 .sdata2.byte 0xa0080718 0x1 main.o
                0xa0080718 c2
.sdata 0xa0080728 0x8
 *(.sdata)
 .sdata 0xa0080728 0x1 main.o
                0xa0080728 c1
 *(.sdata.*)
 *fill* 0xa0080729 0x3 00
 .sdata.int 0xa008072c 0x4 main.o
                0xa008072c i1
```

Original __copy_table

```
PROVIDE(__copy_table = .) ;
LONG(LOADADDR(.data));
LONG(ABSOLUTE(DATA_BASE));
LONG(SIZEOF(.data));

LONG(LOADADDR(.sdata));
LONG(ABSOLUTE(SDATA_BASE));
LONG(SIZEOF(.sdata));
...
```

__copy_table mit .sdata2

```
PROVIDE(__copy_table = .) ;
LONG(LOADADDR(.data));
LONG(ABSOLUTE(DATA_BASE));
LONG(SIZEOF(.data));

LONG(LOADADDR(.sdata));
LONG(ABSOLUTE(SDATA_BASE));
LONG(SIZEOF(.sdata));

LONG(LOADADDR(.sdata2));
LONG(ABSOLUTE(SDA2_BASE));
LONG(SIZEOF(.sdata2));
...
```

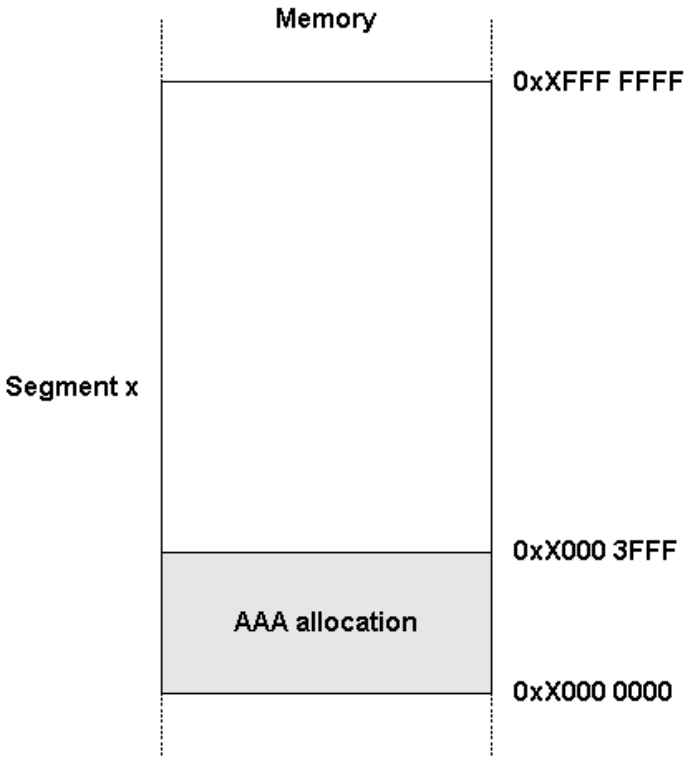
Original __clear_table

```
PROVIDE(__clear_table = .) ;
LONG(0 + ADDR(.bss));
LONG(SIZEOF(.bss));
LONG(0 + ADDR(.sbss));
LONG(SIZEOF(.sbss));
...
```

__clear_table mit .sbss2

```
PROVIDE(__clear_table = .) ;
LONG(0 + ADDR(.bss));
LONG(SIZEOF(.bss));
LONG(0 + ADDR(.sbss));
LONG(SIZEOF(.sbss));
LONG(0 + ADDR(.sbss2));
LONG(SIZEOF(.sbss2));
...
```

Absolute Adressierung



- Absolut adressierbarer Bereich am Anfang jedes Segments
- 16 Segmente verfügbar
- 16 kByte abs. adressierbar pro Segment
- Adressiert über 18bit-Adresse:
  - → 4 Bit adressieren Segment
  - → 14 Bit Offset im Segment

Deklaration von Variablen

```
/* allocate varibales according to their alignments */
#pragma section .zdata.myabsdata_1 1 awz
char char1;
char char2;
char char3;
#pragma section
#pragma section .zdata.myabsdata_4 4 awz
int int1;
int int2;
#pragma section
#pragma section .zdata.myabsdata_2 2 awz
short short1;
short short2;
#pragma section
```

Linker Script File

```
.zdata :
  {
    ZDATA_BASE = . ;
    *(.bdata)
    . = ALIGN(8) ;

    *(SORT(.zdata.myabsdata_*))
    *(.zdata)
    *(.zdata.*)
    ZBSS_END = . ;
  } > ext_cram
```

Mapfile

Ausgabe der Ergebnisse im Mapfile

```
.zdata.myabsdata_1 0xa0000000 0x3 pragma.o
                0xa0000000 char1
                0xa0000001 char2
                0xa0000002 char3
*fill* 0xa0000003 0x1 00
.zdata.myabsdata_2 0xa0000004 0x4 pragma.o
                0xa0000004 short1
                0xa0000006 short2
.zdata.myabsdata_4 0xa0000008 0x8 pragma.o
                0xa0000008 int1
                0xa000000c int2
```

Inline Assembler

- Manche Instruktionen können nicht optimal durch C-Statements repräsentiert werden
- Besser: Assembler Code direkt im C-Quellcode
- Maschinenspezifischer Code für Aktionen auf Maschinenebene
- Operanden für den Inline Assembler können C-Expressions sein
- Textuelle Ersetzung des Inline Assembler Codes durch den Compiler ('BlackBox')
- Einbindung in ein Makro möglich

Syntax

Inline Assembler Statement Syntax

```
__asm__ volatile ("<Assembly Language Template>"
                  : Liste der Output Operanden
                  : Liste der Input Operanden
                  : Liste der Clobbers );
```

- Für ANSI-Compatibilität: __asm__ und __volatile__
- Max. 30 Operanden möglich

Assembly Language Template

- Eine oder mehrere gültige Assemblerinstruktionen
- Leerzeichen und Tabs erlaubt
- Der Compiler sieht das Template nur als String an
  - Die Instruktionen sind durch doppelte Hochkommata umgeben
  - Trennung von verschiedenen Instruktionen durch \n
  - Jede Zeilen als eigenen String (Concatenation)

C-Expressions im Template

- C-Expressions können im Assembly Language Template aufgenommen werden
- Operanden werden als %<number> referenziert
- <number> entspricht der Nummer der Expression in der Operandenliste
- Alternative: Referenzierung über %[<name>], wenn dem Operanden ein Name zugewiesen wurde

Beispiel

Inline Assembler Statement:

```
__asm__ volatile ("or %0, %1, %2"
        : "=d" (a)
        : "d" (b), "d" (c))
```

Generierter Code

```
or %d15, %d2, %d4
```

Attribut absdata

Syntax

```
extern int absint __attribute__ ((absdata));
```

- Variable wird absolut adressiert
- Variable muss im Linker Script File in eine absolut adressierte Output Section allokiert werden
- Attribut wird normalerweise nicht explizit benutzt (Implizite Benutzung durch Attribut section oder asection)
- Anwendung nur bei extern-Deklarationen

Attribut smalldata

Syntax

```
extern int smallint __attribute__ ((smalldata));
```

- Variable wird registerrelativ ('small') adressiert
- Variable muss im Linker Script File in eine small adressierte Output Section allokiert werden
- Attribut wird normalerweise nicht explizit benutzt (Implizite Benutzung durch Attribut section oder asection)
- Anwendung nur bei extern-Deklarationen

Attribut aligned

Syntax

```
int alignint __attribute__ ((aligned(8)));
```

- Legt Variablen und Funktionen aligned ab
- Alignment nur möglich größer als default Alignment
- Alignment muss Potenz von 2 sein (2^1, 2^2, 2^4)

Attribut packed

Syntax

```
typedef struct{
    char s1;
    int i1;
} __attribute__ ((packed)) struct_t;
```

- Bei aligned Ablage ist Platz zwischen Strukturelementen
- Struktur kann gepackt werden: Keine Lücken zwischen Strukturelementen
- Kein Optimaler Zugriff auf Variablen mit mehr als einem Byte Größe

Beispiel 1

Quellcode

```
typedef struct{
    char s1;
    int i1;
} struct_t;
...
struct_t str;
...
str.i1 = 24;
```

Generierter Code

```
movh.a %a15,HI:str
lea %a15,[%a15] LO:str
mov %d15, 24
st.w [%a15] 4, %d15
```

Beispiel 2

Quellcode

```
typedef struct{
    char s1;
    int i1;
} __attribute__ ((packed)) \
    struct_t;
...
struct_t str;
...
str.i1 = 24;
```

Generierter Code

```
movh.a %a15,HI:str
lea %a15,[%a15] LO:str
ld.b %d15, [%a15] 1
and %d15, %d15, 0
or %d15, %d15, 24
st.b [%a15] 1, %d15
ld.b %d15, [%a15] 2
and %d15, %d15, 0
st.b [%a15] 2, %d15
ld.b %d15, [%a15] 3
and %d15, %d15, 0
st.b [%a15] 3, %d15
ld.b %d15, [%a15] 4
and %d15, %d15, 0
st.b [%a15] 4, %d15
```

Attribut alignedaccess

Syntax

```
int* foo __attribute__ ((alignedaccess("4")));
```

- Im PRAM ist nur 4-Byte Zugriff erlaubt
- Legt die Zugriffsart auf Variablen fest
- Mögliche Zugriffe: char (1), short (2) und int(4)
- `-maligned-access` ist Voraussetzung

Attribut interrupt

Syntax

```
void foo (void) __attribute__ ((interrupt));
```

- Bei Interrupt Service Routinen muss der Upper Context nicht erneut gesichert werden
- jl auf die Funktion in der Interrupt Vector Table
- Am Ende der Funktion: ji zurück in die Interrupt Vector Table
- rfe kommt in der Interrupt Vector Table

Attribut interrupt_handler

Syntax

```
void foo (void) __attribute__ ((interrupt_handler));
```

- jump auf die Funktion in der Interrupt Vector Table
- Am Ende der Funktion: rfe, also direktes Beenden der Interrupt Service Routine

Attribut longcall

Syntax

```
void foo (void) __attribute__ ((longcall));
```

- Funktion wird durch calli statt call aufgerufen
- call kann nur ± 16MByte adressieren
- Vom aktuellen PC weiter entfernte Funktionen müssen durch calli aufgerufen werden.

Attribut section

Syntax

```
int foo __attribute__ ((section(".foo")));
```

- Legt Variablen und Funktionen in benutzerdefinierte Sektionen ab

Attribut asection

Syntax

```
__attribute__ ((asection("<name>", "a=<align>", "f=<flags>")))
```

Parameter

`name`	Name der Sektion
`align`	Alignment als 2er-Potenz
`flags`	Zusätzliche Flags für die Sektion

$\Rightarrow$ asection einsetzbar für Funktionen und Variablen

Flags

`a`	allocatable (immer gesetzt)
`x`	executable
`w`	writable
`p`	PCP section
`t`	small adressiert (10 Bit)
`s`	small adressiert (16 Bit)
`z`	absolut adressiert
`b`	Bitsection

Sonstiges

- Registerrelativ ('small') adressierte Sektionen müssen mit .sdata. oder .sbss. beginnen
- Wenn die Sektion Code enthält: Flag x muss gesetzt sein

Pragma section I

Syntax

```
#pragma section <name> [<alignment>] [<flags>]
  /* Objekte */
#pragma section
```

- Attribut asection nur möglich für eine Variable / Funktion
- pragma section für beliebige Zahl von Variablen / Funktionen

Pragma section II

- Gleiche Flags wie bei Attribut asection
- Gleiche Vorgaben für Alignment und Namen (.sdata. bzw. .zdata.)
- Pragma sections dürfen nicht verschachtelt werden
- Flag x gesetzt: Nur Funktionen werden lokatiert
- Flag x nicht gesetzt: Nur Variablen werden lokatiert
- Sektionsnamen müssen gültige C-Identifier sein (.sbss.1 ist verboten)

Pragma branch I

- Attribut ist in Vorbereitung
- Default Branches können für if−else- und switch−case-Statements angegeben werden.
- Die Pragmas müssen immer direkt vor der if-, else- oder case-Instruktion stehen
- Ausnahme: Kommentare
- Das Pragma gilt immer für das jeweils nächste Statement
- Bei `-mwarn-pragma-branch` wird eine Warning ausgegeben, wenn für eine Instruktion kein Default Branch angegeben wird.

Pragma branch II

`#pragma branch_if_default`

Der if-Case ist der Standardcase

`#pragma branch_else_default`

Der else-Case ist der Standardcase

`#pragma branch_if_not_default`

Synonym für branch_else_default

`#pragma branch_case_default`

Markiert den Standardcase innerhalb eines switch-Statements

`#pragma branch_no_default`

Weder der if- noch der else-Case sind Standardcase

Bit-Datentyp

- _bit ist Typ für Bitvariablen
- Drei Möglichkeiten:
  - Global nichtinitialisiert: `_bit bit1;`
  - Global initialisiert: `_bit bit2 = 1;`
  - Lokal im Modul initialisiert: `static _bit bit3;`

Implementation

- Implementiert als unsigned int
- Wertebereich 0 und 1
- Bei cast und Zuweisung von längerem integer-Typ bekommt die Bitvariable immer das LSB zugewiesen

Der Compiler unterscheidet die Behandlung von Zuweisungen und Abfragen. Grundsätzlich gilt:

Hinweis

In Abfragen wird immer auf den Datentyp integer expandiert und bei Zuweisungen auf den kleinsten beteiligten Datentyp verringert.

Erlaubte Operatoren

- Zuweisung
- Logische unäre und binäre Operatoren
- Testoperatoren

Verbotene Operatoren

- ++, −− (post/pre increment/decrement)
- Unäres Minus (-b1)
- Indirection (Addressoperator &)
- +, −, ∗, /, %, <<, >>
- +=, −=, ∗=, /=, %=, <<=, >>=
- indirection (array/pointer/address)

Linken von Bitvariablen

- Compiler sieht ein Byte pro Bit vor
- Linker packt Bits in Bytes zusammen (Linkeroption `--relax` oder `--relax-bdata`)
- Default Sektionen: .bdata und .bbss
- Vom Benutzer definierte Sektionen müssen mit .bdata. oder .bbss. beginnen
- Flag b muss bei Attribut asection und bei pragma section gesetzt sein

Bits im Mapfile

- Bitvariablen werden im Mapfile wie andere Variablen dargestellt
- Zusätzlich zur Adresse des Bytes wird die Bitposition im Byte angegeben:

  0xa00001f0.2

Definition von Bitfeldern

```
struct {
  unsigned int b0 : 1;
  unsigned int b1 : 1;
  unsigned int : 2;
  unsigned int b4 : 1;
  unsigned int b5to7 : 3;
} reg8;
```

- Alignment bei Größe <= 8 Bit: 1 Byte
- Alignment bei Größe > 8 Bit: 4 Byte
- EABI: Bitfield darf keine zwei 16-Bit Boundaries überschreiten
- Bei volatile -Bitfield Zugriff über ldmst

8 Sektionen
- Default und Bit Daten Sektionen
- Adressierungsarten
- PCP, C++, Debug Sektionen

9 Linker Skript Datei
- Eingebaute Funktionen
- MEMORY und SECTION Kommando
- Initialisierung und Lokatierung
- Lokatierung relativ zur Endadresse

10 Empfehlungen
- Programmierung des PCP
- Beispiel

TriCore Sektionen (Teil I)

Default Sektionen

`.text`	Sektion für Befehle (Code)
`.data`	Initialisierte Daten stehen in '.data'
`.bss`	Nicht initialisierte Daten liegen in '.bss'
`.rodata`	Ablage von schreibgeschützten Daten
`.version_info`	Informationen über den Compiler, mit dem das Modul übersetzt wurde

Für diese Default Sections existieren die Subsections `.a1` `.a2` `.a4` `.a8`.

Bit Daten Sektionen

`.bbss`	Nicht initialisierte Bit Daten liegen in Sektion '.bbss'
`.bdata`	Bit Variablen werden in '.bdata' abgelegt

TriCore Sektionen (Teil II)

Small adressierbare Sektionen

`.sdata`	Sektion '.sdata' speichert initialisierte Daten, die über small data area pointer (%a0) adressierbar sind
`.sbss`	Nicht initialisierte Daten in der Sektion '.sbss' über small data area pointer (%a0) adressierbar
`.sdata.rodata`	Ablage von schreibgeschützten Daten, die small adressiert werden können

Absolut adressierbare Sektionen

`.zdata`	Initialisierte Daten, absolut adressierbar
`.zbss`	Nicht initialisierte Daten, absolut adressierbar
`.zrodata`	Ablage von schreibgeschützten Daten, die absolut adressiert werden können

TriCore Sektionen (Teil III)

PCP Sektionen

`.pcptext`	PCP Code Sektion
`.pcpdata`	PCP Data Sektion

C++ Sektionen

`.eh_frame`	Exception handling frame für C++ exceptions
`.ctors`	Sektion für Konstruktoren
`.dtors`	Sektion für Destruktoren

Debug Sektionen

`.debug_<name>`
Diverse Debug Sektionen

Grundlagen

- Jede Objektdatei besteht aus Sektionen
- Jede Sektion besitzt einen Namen und eine Größenangabe
- Unterscheidung von ladbaren und nicht ladbaren Sektionen
- Weitere Formen sind Sektionen mit Debug Information
- Abbildungsvorschrift von Input und Output Sektionen und Speicheraufteilung
- Lad- und allokierbare Output Sektionen haben zwei Adressen:

VMA

Die Virtual Memory Address gibt die Adresse für die Ausführung im Programm an

LMA

Die Load Memory Address spezifiziert die Ladeadresse einer Sektion

Eingebaute Funktionen (Teil I)

`ABSOLUTE(<exp>)`
: Gibt den Wert von <exp> zurück

`ADDR(<section>)`
: Rückgabewert ist die absolute Adresse (VMA) von <section>

`LOADADDR(<section>)`
: Rückgabewert ist die absolute Load Memory Address von <section>

`ALIGN(<exp>)`
: Gibt den location counter (.) ausgerichtet an der nächsten <exp> Grenze zurück

Eingebaute Funktionen (Teil II)

`DEFINED(<symbol>)`
: Falls <symbol> definiert ist und in der globalen Symboltabelle steht, ist der Rückgabewert Eins ansonsten Null

`PROVIDE(<symbol> = <expression>)`
: Definiert ein Symbol nur dann, wenn es referenziert wird und in den eingebundenen Objektdateien noch nicht definiert ist

`SIZEOF(<section>)`
: Liefert die Größe der Sektion in Bytes zurück

Wichtige Dateikommandos

`INCLUDE <filename>`
: Einbinden der Linker Skript Datei <filename>

`INPUT(<file>, <file>, ...), ...`
: Einbinden der angegebenen Dateien im Linklauf

`GROUP(<file>, <file>, ...), ...`
: Im Gegensatz zu INPUT sollten Archivdateien angegeben werden

`OUTPUT(<filename>)`
: OUTPUT gibt den Namen der Ausgabedatei an

`SEARCH DIR(<path>)`
: Die SEARCH DIR Anweisung erweitert den Suchpfad um <path>

`STARTUP(<filename>)`
: Angabe der Datei, die als erstes gelinkt wird

MEMORY Kommando

Beschreibt Ablageort und Größe von Speicherbereichen

Bedeutung der Attribute

Attribut	**Beschreibung**
r	Nur Lesezugriff
w	Lese- und Schreibzugriff
p	PCP Memory
x	Ausführbare Sektion
a	Allokierbare Sektion
i oder I	Initialisierte Sektion
!	Invertiert Bedeutung der Attribute

Linker Description File Memory Region

```
MEMORY
{
  ext_cram (arx!p): org = 0xa0000000, len = 512K
  ext_dram (aw!xp): org = 0xa0080000, len = 1M
  int_cram (arx!p): org = 0xc0000000, len = 0x8000
  int_dram (aw!xp): org = 0xd0000000, len = 0x8000
  pcp_data (awp!x): org = 0xf0010000, len = 32K
  pcp_text (arxp): org = 0xf0020000, len = 16K
}
```

Input Sektionen mit Platzhaltermuster

'*'	Übereinstimmung mit jeglicher Art von Zeichen
'?'	Übereinstimmung mit einzelnen Zeichen
[<chars>]	Angabe eines Übereinstimmungsbereiches z.B. '[a-z]'
\	Escapesequenz für Platzhaltermuster

Sortierung der Daten

Normalerweise werden die Platzerhalter in der Suchreihenfolge beim Linken ersetzt. Mit der Verwendung des Schlüsselworts SORT vor einem Platzhaltermuster (z.B. SORT(.text*)) werden die Dateien oder Sektionen in alphabetischer Reihenfolge vom Linker in die Ausgabedatei geschrieben

SECTION Kommando

- Symbolzuweisungen
- Beschreibung für Output Sektion
- Abbildungsvorschriften von Input zu Output Sektionen

Output Sektionen können mit > in definierte Speicherbereiche gelegt werden

Linker Description File Sections

```
SECTIONS
 {
 .zdata :
  {
    ZDATA_BASE = . ;
    *(.bdata)
    . = ALIGN(8) ;
    *(SORT(.zdata.myabsdata_*))
    *(SORT(.zdata.myabsdata*))
    *(.zdata)
    *(.zdata.*)
    *(.gnu.linkonce.z.*)
    ZDATA_END = . ;
  } > ext_cram
```

Beschreibung Output Section

Syntax

```
section [address] [(type)] : [AT(lma)]
{
    output-section-command
    output-section-command
} [>region] [AT>lma_region] [:phdr :phdr  ] [=fillexpr]
```

Beispiel

```
.pcpdata 0xf0010000 :
AT ( ADDR (.text) + SIZEOF (.text) )
 {
    PRAM_BASE = . ;
    *(.pcpdata)
    PRAM_END = . ;
    .text
 }
```

Initialisierung der Tabellen (Teil I)

```
.rodata :
{
  . = ALIGN(8) ;
  *(.rodata)
  *(.rodata.*)
  *(.gnu.linkonce.r.*)
  *(.rodata1)
  *(.toc)
  /*
   * Create the clear and copy tables
   * that tell the startup code
   * which memory areas to clear and
   * to copy, respectively.
   */
  . = ALIGN(4) ;
```

Initialisierung der Tabellen (Teil II)

```
  PROVIDE(__clear_table = .) ;
  LONG(0 + ADDR(.bss)); LONG(SIZEOF(.bss));
  LONG(0 + ADDR(.sbss)); LONG(SIZEOF(.sbss));
  LONG(0 + ADDR(.zbss)); LONG(SIZEOF(.zbss));
  LONG(-1); LONG(-1);
  PROVIDE(__copy_table = .) ;
  LONG(LOADADDR(.data)); LONG(ABSOLUTE(DATA_BASE));
                            LONG(SIZEOF(.data));
  LONG(LOADADDR(.sdata)); LONG(ABSOLUTE(SDATA_BASE));
                            LONG(SIZEOF(.sdata));
  LONG(LOADADDR(.pcpdata)); LONG(ABSOLUTE(PRAM_BASE));
                            LONG(SIZEOF(.pcpdata));
  LONG(LOADADDR(.pcptext)); LONG(ABSOLUTE(PCODE_BASE));
                            LONG(SIZEOF(.pcptext));
  LONG(-1); LONG(-1); LONG(-1);
} > ext_cram
```

Lokatierung der Sektionen (Teil I)

PCP Text Sektion

```
.pcptext : AT(_etext)
{
  . = ALIGN(4) ;
  PCODE_BASE = . ;
  *(.pcptext)
  *(.pcptext.*)
  . = ALIGN(4) ;
  PCODE_END = . ;
} > pcp_text
. = ALIGN(4) ;
```

oder

```
.pcptext :
{
...
} > pcp_text AT > ext_cram
. = ALIGN(4) ;
```

Lokatierung der Sektionen (Teil II)

PCP Data Sektion

```
.pcpdata : AT(LOADADDR(.pcptext) + SIZEOF(.pcptext))
{
  . = ALIGN(4) ;
  PRAM_BASE = . ;
  *(.pcpdata)
  *(.pcpdata.*)
  . = ALIGN(4) ;
  PRAM_END = . ;
} > pcp_data
```

Lokatierung der Sektionen (Teil III)

Data Sektion

```
.data : AT(LOADADDR(.pcpdata) + SIZEOF(.pcpdata))
{
  . = ALIGN(8) ;
  DATA_BASE = ABSOLUTE(.) ;
  *(.ownsection)
  *(.data)
  *(.data.*)
  *(.gnu.linkonce.d*)
  SORT(CONSTRUCTORS)
  DATA_END = ABSOLUTE(.) ;
} > ext_dram
. = ALIGN(8)
```

Lokatierung relativ zur Endadresse (Teil I)

Problem

Lokatieren einer Sektion relativ zu einer festen Endadresse

Der Linker kann Sektion nur relativ zu einer Anfangsadresse lokatieren und insbesondere keine Vorwärtsreferenzen behandeln

Lösung

Linken mit zwei Linkläufen

Erster Linklauf

Größe der Sektion wird bestimmt

Zweiter Linklauf

Definition der Anfangsadresse als (Endadresse - Größe)

Lokatierung relativ zur Endadresse (Teil II)

Einträge in Linker Description File

```
___startof_libc_mydata = 0xa0800000 -
          (MYDATA_END - MYDATA_BASE) ;

.mydata DEFINED (___startof_libc_mydata) ?
                      ___startof_libc_mydata : .
      : AT(LOADADDR(.sdata) + SIZEOF(.sdata))
{
  MYDATA_BASE = ABSOLUTE (.) ;
  c.o(.mydata)
  c2.o(.mydata)
  MYDATA_END = ABSOLUTE(.) ;
}
```

Lokatierung relativ zur Endadresse (Teil III)

Erster Linklauf

```
tricore-ld -T ld.scr -o prog.pass1 <objectliste>
```

Ermitteln der Größe aus dem Linkergebnis mit mksyms

```
#!/bin/sh
tricore-lsyms --name=^___startof_ prog.pass1 |
 while read sym; do
  set $sym
  echo $3 = 0x$1\;
done > prog.syms
```

ergibt

```
___startof_libc_mydata = 0xa07fffec;
```

Zweiter Linklauf

```
tricore-ld -T prog.syms -T ld.scr -o prog
<objectliste>
```

Lokatierung relativ zur Endadresse (Teil IV)

Dazugehöriger Makefile

```
SHELL=sh
prog: c.o c2.o ld.scr FRC
        $(LD) -T ld.scr -Map $(addsuffix .map1,$@) \
            -o $(addsuffix .pass1,$@) \
            $(CRT0) c.o c2.o $(LDFLAGS)
        $(SHELL) ./mksyms $(addsuffix .pass1, $@) \
            > $(addsuffix .syms,$@)
        $(LD) -T $(addsuffix .syms,$@) $(sizeof) -T ld.scr\
            -extmap=N -Map $(addsuffix .map, $@) \
            -o $@ $(CRT0) c.o c2.o $(LDFLAGS)
        $(RM) $(addsuffix .pass1,$@) $(addsuffix .syms,$@)\
            $(addsuffix .map1,$@)
```

Programmierung des PCP

- PCP Code und PCP Daten haben eigene Sektionen: .pcptext und .pcpdata
- Im default Linker Script vorgesehen: .pcptext.* und .pcpdata.*
- Funktionen und Variablen werden über Attribut section oder pragma section in diese Sektionen gelegt
- PCP Code und PCP Daten werden vom Startup-Code in entsprechenden Speicher kopiert
- Innerhalb PCP Sektion nur Inline Assembler erlaubt

Beispiel

```
#pragma section .pcpdata
int pcpint;
char pcpchar;
#pragma section

void pcpadd (void) __attribute__ ((section(".pcptext")));

void pcpadd (void)
{
    __asm__ __volatile__(...);
}
```

Mapfile

Ausgabe im Mapfile (gekürzt)

```
.pcptext 0xf0020000 0x1c load address 0xa0001da8
 *(.pcptext)
 .pcptext 0xf0020000 0x1a main.o
                0xf0020000 pcpadd
 *(.pcptext.*)

.pcpdata 0xf0010000 0x8 load address 0xa0001dc4
 *(.pcpdata)
 .pcpdata 0xf0010000 0x8 main.o
                0xf0010004 pcpchar
                0xf0010000 pcpint
 *(.pcpdata.*)
```

Beispiel

```
void test(void)
{
    static unsigned short in_u16;
    static unsigned char out1_u8, out2_u8;
/* Optimal */
#if CASE==2
    out1_u8 = (unsigned char )(unsigned short)(in_u16 > 255 ? 255 : in_u16);
    out2_u8 = out1_u8;
#endif
/* Unguenstig */
#if CASE==3
    out1_u8 = (unsigned char)(in_u16 > 255 ? (unsigned short)255 : in_u16);
    out2_u8 = out1_u8;
#endif
}
```

Kontakt

Noch weitere Fragen?

Kontaktieren Sie uns unter:

HighTec EDV-Systeme GmbH
Stammsitz & Entwicklung
Feldmannstraße 98
D-66119 Saarbrücken
Tel.: 0681/92613-16 Fax: -26
mailto:support@hightec-rt.com

www.hightec-rt.com