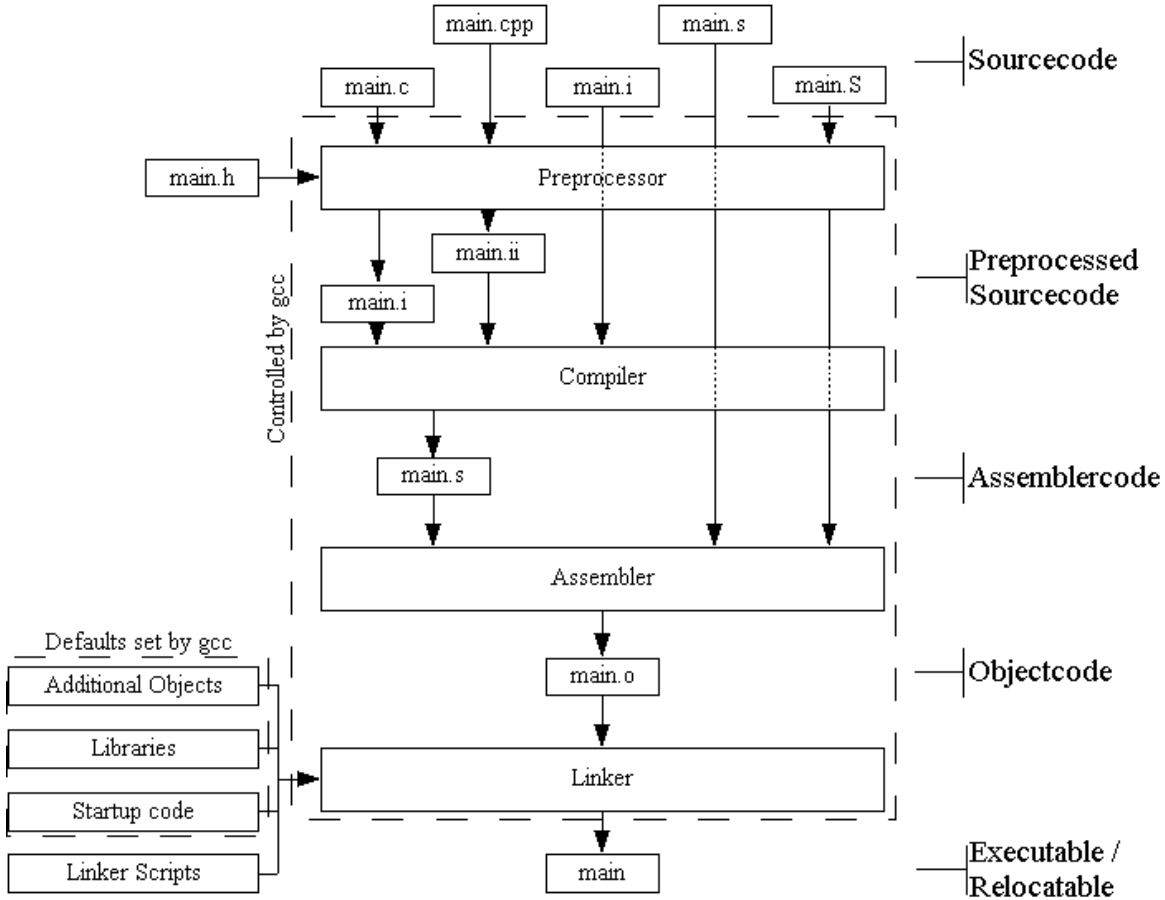
Power User Training

HighTec EDV-Systeme GmbH

Feldmannstraße 98
66119 Saarbrücken
www.hightec-rt.com

1. Build Process
   - Preprocessor
   - Compiler
   - Assembler
   - Linker
2. Additional Tools
   - tricore-objcopy
   - tricore-objdump
   - tricore-nm
   - tricore-ar
3. New Features
   - Extensions Made by HighTec
   - GNU Project
   - Planned features

Abstract


main.cpp
main.s
main.c
main.i
main.S
Sourcecode
main.h
Preprocessor
main.ii
main.i
Preprocessed Sourcecode
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

- Simplifies build process by:
  Using the appropriate tools (cpp, cc1, as, ld) for each file, dependent on its suffix
- Passing of specified options to the respective tools
- Passing of default options to the respective tools

  $\Rightarrow$ Configurable by means of `specs` file

  $\Rightarrow$ Preferred configuration in `tricore.specs`

File Types and Suffixes

`file.c`	C source code file
`file.cc, file.cxx, file.cpp, file.C`	C++ source code file
`file.h`	C header file
`file.i`	Preprocessed C source
`file.ii`	Preprocessed C++ soucre
`file.s`	Assembler code
`file.S`	Assembler code that is to be preprocessed

Prepocessor

- Integrated into compiler → usage: option `-E`
- Inclusion of header files
- Macro expansion
- Syntax checking of preprocessor directives
- Conditional compilation
- Replaces comments by single space
- Deletes backslash-newline sequences
- Line control

Important Options I

`-I <dir>` <dir> is apended to include paths list

`-D <macro>` Sets <macro> as defined

`-D <macro>=<value>`
Assignes value <value> to macro <macro>

`-U <macro>` Set <macro> as undefined

`-dM` Generates list of all defined macros and values

`-imacros <file>`
The macros defined in <file> are defined before any input file is processed. This option does the same as `-include` <file> except that only the macro definitions are read.

Important Options II

`-include <file>`
: Includes the contents of <file> to the preprocessed file before processing the #include directives of the input file.

`-Wall`
: Activates all optional warnings which are desirable for normal code.

`-mno-warn-skipped-cpp-directives`
: Outputs warnings when reading invalid preprocessor options

Default Options

- -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

- Translates C/C++ source code into assembler commands
- Optimizes for code size and/or execution time (optional)

-O0	No optimization (level 0)
-O1	The compiler tries to reduce code size, execution time and stack space by 'simple strategies' (level 1)
-O2	Optimizes even more and uses optimization algorithms that do 'not' have negative impact on code size or execution time (level 2)
-O3	Highest level of optimization (level 3) activates function inlining and loop unrolling

- Generation of debugging information (optional)

Major Options

`-E`	Just preprocess
`-S`	Preprocess and compile
`-c`	Preprocess, compile and assemble
`-o <outfile>`	Write output to <outfile>
`-v`	Display version information and which tools are used together with their parameters
`-O[n]`	Turn on optimization (n = 0, 1, 2, 3)
`-Os`	Optimize for code size
`-gdwarf-2`	Generate DWARF 2.0 debugging information
`-Wall`	Generate almost 'all' warnings

TriCore–specific Options (Part I)

`-mabs=<N>` Put variables of size $\leq$ N into absolute addressable area (`.zbss`)

`-mabs-data=<N>,-mabs-const=<N>` Put data/constants with a size $\leq$ N bytes in absolute addressable area.

`-msmall=<N>` Put variables of size $\leq$ N into small data area (`.sdata` and `.sbss`)

`-msmall-data=<N>,-msmall-const=<N>` Put data/constants with a size $\leq$ N bytes in small data area.

`-mtc12` Generate code for TriCore v1.2

`-mtc13` Generate code for TriCore v1.3

`-mcpu<N>` Activate workaround for silicon bug (<N> = 9, 13, 18, 24, 31, 34, 48, 50, 60, 70, 72, 76)

`-mall-errata`

TriCore–specific Options (Part II)

`-mhard-float`
: Use instructions for floating point(TriCore v1.3)

`-msoft-fdiv`
: Use software emulation for floating point divisions

`-mwarnprqa=on|off`
: Turn on/off warnings for QAC pragmas

TriCore specific options (part III)

`-masm-source-lines`
: C source code as comments in asm code

`-mversion-info`
: Generate section `.version_info`

`-maligned-data-sections`
: Generate subsections `.a1` `.a2` `.a4` `.a8` depending on alignment

`-maligned-access`
: Activate attribute `alignedaccess`

TriCore–specific Specs File

Options

`-mcpu=<MCU>`

Specify used MCU

`-mcpu-specs=<Datei>`

Do not use the default specs file tricore.specs

Excerpt of tricore.specs

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

Default options

- -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 language tasks

- Translates mnemonics into machine code
- Splitting of code/data ($\rightarrow$ sections)
- Dissolving of local references
- Generates entries of relocations for eternal references
- Generates a linkable object file

Important options

`-o <objfile>`
: Use <filename> instead of 'a.out' as output file

`--defsym <symbol>=<value>`
: Defines the symbol <sym> with <value>

`--gdwarf2`
: Gernerates DWARF2 debug information for each assembler line

`-I <dir>`
: Adding <dir> into search path of the assembler

`-a[opts][=file]`
: Gives listing in named file (Option -g necessary)

`-masm-source-lines`
: Intermix of C-source und assembler output

TriCore specific options

`-mtc12, -mtc13, -mtc2`
: Assembles for TC1v1.2, TC1v1.3, TC2 instructin set

`-mcpuN`
: Activates workaround for hardware bug (<N> = 9, 34, 48, 50, 60, 70, 72, 81, 82, 83, 94, 95)

`--dont-optimize`
: Prevents the optimization of mnemonic commands (esception for .optim command)

`--insn32-only`
: Uses only 32-bit opcodes (.code16 and .optim are ignored)

`--insn32-preferred`
: As `--insn32-only` but takes care for .code16

`--enforce-aligned-data`
: Alignment depending from size of the variables (sizeof)

Assembler syntax

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

- Definition / declaration of sections
- Definition of absolute terms (integer, float, string)
- Definition of labels / symbols, their validity area and type name
- Activating / deactivating of special options / source code handling

TriCore specific pseudo-opcodes

`.code16`	Uses 16-bit opcode for the next instruction
`.code32`	Uses 32-bit opcode or the next instruction
`.optim`	Tries to optimize the next instruction
`.noopt`	Disables optimization for the next instruction
`.pcptext,.pcpdata`	PCP support

Pseudo-Opcodes for bit variables

`.bit <bname>[,bexpr]`

Creates globale Bit-Variable <bname> and intitialized optional with value (0 or 1) of the absolute term bexpr. Access by symbol name and bit-position through prefix bpos:

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

`.lbit <bname>[,bexpr]`

In difference to .bit is only in the current modul visible (local scope)

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

Takes the name of bit-variable as argument and displays bit-position in the associated section

Pseudo-Opcodes for PCP

- GNU Assembler supports mulitple PCP per Chip by explicit specification in PCP sections (→linker script)
- The startup code `crt0.S` copies PCP code and datas section in associated memory scopes

`.pcptext` Access to PCP text section. Thus the assembler allows the usage of PCP mnemonics instead of TriCore instrucctions (Halfword Access)

`.pcpdata` Access by each word of PCP Data section (e.g. storage for parameters)

`.pcpinitword` .pcpinitword initPC, initDPTR, initFLAGS creates a 32-bit value, which can be used to initialize PCP register R7

Prefixes for assignation of the relocation typ

Prefix description

Prefix	calculates
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 Functions

- Linking of objects
- Combining of archive files
- Relocating of data
- Resolving of symbol references
- Provides additional diagnostic information detailed overview in map file
  - Placement of object files and symbols
  - Common symbols
  - Included archives

Important Options (Part I)

`-o <file>, --output <file>`
: Write output of linker into file <file>

`-M, --print-map`
: Write mapfile to standard output

`-Map <file>`
: Write mapping information to file <file>

`--cref`
: Creates a table of cross-references, which is included into the mapfile

`--defsym <symbol>=<expression>`
: Creates a global symbol with absolute address <expression> in the output file

Important Options (Part II)

`-R <file>, --just-symbols <file>`
: Reads symbolnames and -addresses from <file>, but does not include them into the output file

`-r, -i, --relocateable`
: Runs an incremental linker pass (creates relocatable output)

TriCore Specific Options

`-extmap=<output-option>`
: Generates a 'extended map file' with additional information (Option `--Map` or `-M` needed)

`--warn-orphan`
: Creates an error message, if there is no fixed mapping between an input and output section

`--relax-24rel`
: Relax call and jump commands, whose target addresses can not be reached by pc relative offset nor by absolute addressing

`--relax-bdata`
: Compress bit objects contained in input section `.bdata`

`--relax`
: Relax branches on certain targets, implies options `--relax-24rel` and `--relax-bdata`

Default Options and Arguments

Options

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

Arguments

- Startup Code `crt0.o`

Binutils (Part I)

tricore-ar

Generates, modifies and extracts from archives

tricore-ranlib

Generates an index for an archive

tricore-nm

Lists symbols from an object file

tricore-objcopy

Copies and translates object files into various other object formats

tricore-objdump

Displays various information from an object file

Binutils (Part II)

tricore-addr2line

Convert addresses into file name and line number

tricore-readelf

Displays the contents of an ELF object file

tricore-size

Displays section sizes and absolute size

tricore-strings

Displays the strings in a file

tricore-strip

Discards symbols

Major Options for tricore-objcopy

`-O --output-target <bfdname>`
: Generate an output file in <bfdname> format

`-j --only-section <name>`
: Just copy the specified section from input file to the output file

`-R --remove-section <name>`
: Remove the section named <sectionname> from the output file

`--add-section <sectionname>=<filename>`
: Insert a new section named <sectionname> from <filename>

`--rename-section <old>=<new>[,<flags>]`
: Rename a section and optionally change <flag>

Examples

Intelhex Format

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

Generate a bin file

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

Remove debugging section

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

Major Options for tricore-objdump

`-D, --disassemble-all`
: Disassemble opcodes to assembler mnemonics

`-h, --section-headers`
: Overview of section headers from object file

`-t, --syms` Display symbol table's entries

`-S, --source`
: Display source code intermixed with related assembler output (implicit option `-d`)

Example: HelloSerial

Show section 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
...
```

Major Options for tricore-nm

`-f, --format=<format>`
: Use output format <format> ('bsd','sysv' or 'posix')

`-g, --extern-only`
: Just show external symbols

`-n, --numeric-sort`
: Sort symbols by their addresses

`--size-sort`
: Sort by size

`-u, --undefined-only`
: Just show undefined symbols (references to other object file)

Example: HelloSerial

Show external symbols

```
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
```

Show undefined symbols

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

```
lock_wdtcon
unlock_wdtcon
```

Major Optons for tricore-ar

`d`	Remove module <member> from archive
`m[ab]`	Move modules inside of archive
`q[f]`	Fast add files <member> to archive by appending them (no name checking for known names)
`r[ab][f][u]`	Add files <member> to archive by replacing them (name checking performed)
`x[o]`	Extract module <member> from archive

Example

Extract modules from an archive

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

Remove modules from an archive

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

Append/replace modules

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

Extensions Made by HighTec

- Bit data type
- Multiple bit-data sections
- Pragma sections
- Packed data storage
- Relative addressing
- Circular addressing
- Constants in read only section
- Mapping of default sections
- CSA overhead
- Indirect addressing for longcall
- Position independent code
- Source lines in assembler output

Derivative Specs File

- Derivative specific parameters (e.g. errata)

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

- Tags begin with $*$
- Interface for own configurations

Configuration

Example

- Content of *tc1775_errata replaced by %(tc1775_errata)
- Access to tag *TC1775 with

```
tricore-gcc -mcpu=TC1775 ...
equals
tricore-gcc -mtc12 -mcpu18 -mcpu24 -mcpu31 -mcpu34
-mcpu48 -mcpu50 ...
```

Note

An own configuration file <file> can be specified by `-mcpu-specs=<file>`.

Indirect Addressing by using longcall

Background

- Memory mapping on TC1796
- Calls from external to internes RAM via indirect addressing
- Internal RAM significant faster in execution time
  $\Rightarrow$ Map frequently used functions into internal RAM
- Introduction of an attribute longcall

Attribute longcall

longcall

Using the function attribute longcall , arbitrary functions can be called using calli , thus diminishing code size as well as execution time.

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

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

CSA Overhead

Context Save/Restore

If a jump or link instruction is used instead of call an interrupt handler saves one context save and context restore process.

- Saving of upper context (in CSA) in interrupt and trap
  - PSW (Processor Status Word)
  - A10 to A15 (Address Register)
  - D8 to D15 (Data Register)
- Restoring upper context after `ret` or `rfe`

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

$\Rightarrow$ These instructions use the return address in `%a11`

Example

`interrupt` Using a jump indirect instruction `ji %a11`

`interrupt_handler`
Function returns using `rfe`

```
#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); /* uses ji %a11*/
 ifoo(1); /* uses ji %a11*/
 *IntSrc = 0x1001;
 __asm__ volatile ("enable");
 *IntSrc |= 0x8000;
...
```

Planned features for GCC 4.0

General Optimisation Improvements

- Improved usability of profile feedback and coverage test
- Improved inlining heuristics for C, Objective-C, C++. Call graph based on out-of-order inlining is now activated when using option -O2
- Improved optimisation strategies
- Global optimisation module

Planned features for GCC 4.0

SSA

Contains two high-level intermediate languages (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 Addressing Modes
- Regular Addressing
- Register relative addressing
- Absolute addressing

5 Inline Assembler
- Assembly Language Template

6 Attributes and Pragmas
- Attributes for Addressing
- Attributes for Data Storage
- Attributes for Sections
- Pragmas

7 Data Type _bit
- Introduction
- Linking
- Map File
- Bit Fields

Regular Addressing

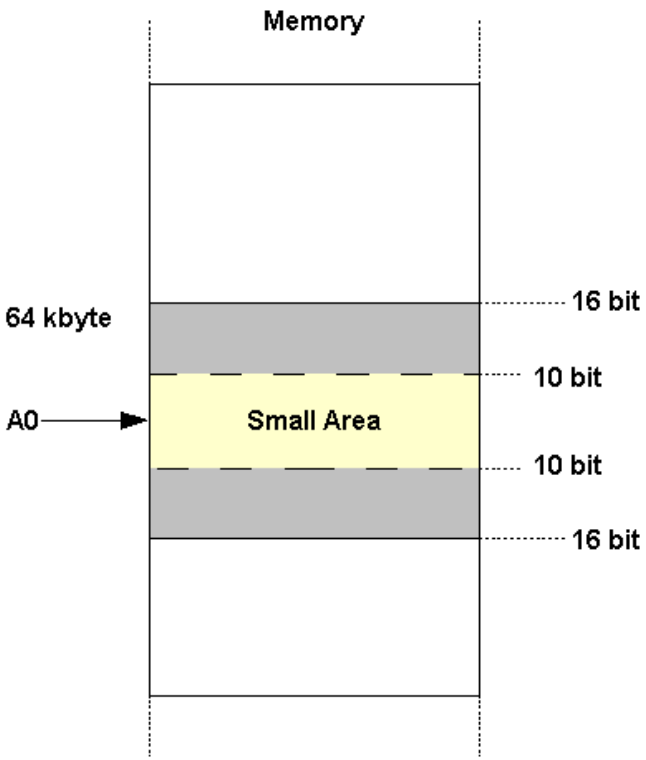
Source code

```
c1 = foo;
```

Generated code

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

Register relative addressing



- Addressing relativ to base address in register
- Address is composed by addition / subtraction of an offset and a base address
- 10 or 16 bit offset
  - 10 bit offset available for all memory operations
  - 16 bit offset available for many memory operations

Variable Declaration

- Attribute asection or pragma section
- Section name .sdata or .sbss
- Own sections begins with .sdata. or .sbss.
- Flag t for 10bit, flag s for 16bit

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

or

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

Small addressable sections

- At most four small data sections in executable
- To be addressed via different registers

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

Linker Script I

- Output sections .sdata/.sbss are provided in default linker script
- In case, that output sections .sdataX/.sbssX are not defined in linker script, the linker creates them, if small data area $> 64k$
- .sbss and .sbss.* per default in output section .sbss
- .sdata and .sdata.* per default in output section .sdata
  → .sdataX/.sbssX before .sdata/.sbss in linker script
- Startup code copies all data and initialises .sbss with 0
  → Sections .sdataX have to copied in __copy_table
  → Sections .sbssX have to cleared in __clear_table

Linker Script II

- Each small data area has a symbol
- Default: symbols corresponds to start adress + 32768
- Symbols are initialised automatically, if not defined in the linker script

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

Example

Target

- Define variables in different small addressed sections

Approach

1. Declare variables
2. Define additional output section .sdata2
3. Define additional output section .sbss2
4. adjust __copy_table

Variables

```
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) :
  ...
```

Map File

Output of results in in map file (shortened)

```
.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));
...
```

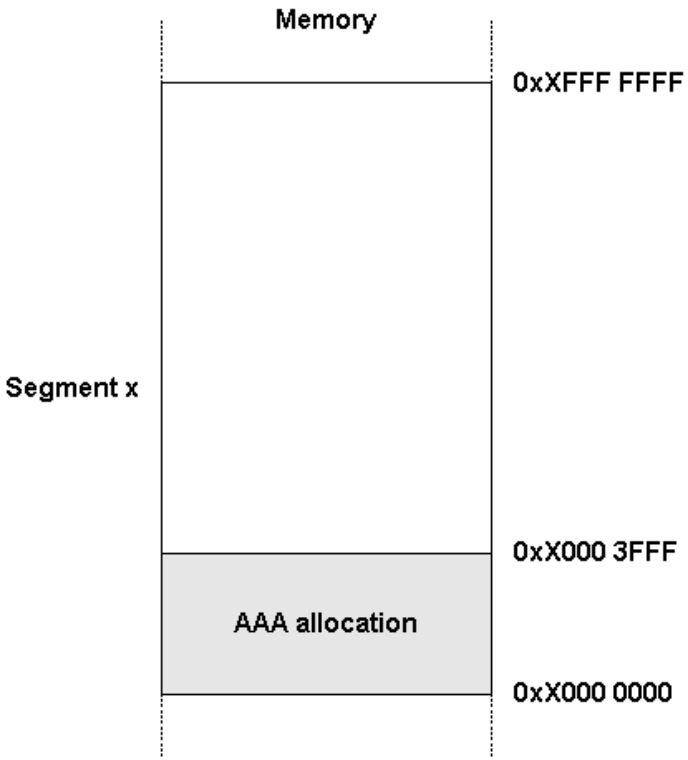
Original __clear_table

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

__clear_table with .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 addressing



- Absolute addressable range at beginning of each segment
- 16 segments available
- 16 kByte abs. addressable per segment
- Addressed with 18bit address:
  - → 4 bit address the segment
  - → 14 bit offset within the segment

Variable declarations

```
/* allocate variables 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
```

Map File

Output of results in map file

```
.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

- Some instructions can not be optimally represented by C code
- An improvement is to integrate assembler code in C sources
- Machine specific code for machine specific actions
- Operands of the inline assembler can be C expressions
- Textual replacement of inline assembler code by the compiler ('BlackBox')
- Integration into macros is possible

Syntax

Inline Assembler Statement Syntax

```
__asm__ volatile ("<Assembly Language Template>"
                  : List of output operands
                  : List of input operands
                  : List of clobbers );
```

- For ANSI compatibility: __asm__ und __volatile__
- At most 30 operands

Assembly Language Template

- One or more valid assebler instructions
- Whitespaces allowed
- The compiler interprets the template as a string
  - The instructions are double quoted
  - Instructions are separated by \n
  - Each line is a string (concatenation)

C Expressions in a Template

- C expressions can be included in a assembly language template
- Operands are referenced %<number>
- <number> equals to the number of the expression in the list of operands
- Alternative: an operand can be referenced by %[<name>], if it has a name attached to it

Example

Inline Assembler Statement:

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

Generated Code

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

Attribute absdata

Syntax

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

- Absolute addressing of variable
- Variable has to be allocated in an absolutely addressed output section in the linker script file
- In general, the attribute is not used explicitely (implicit use by attribute section or asection)
- used only with extern declarations

Attribute smalldata

Syntax

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

- Addressing of variable ('small') relative to register
- Variable has to be allocated in a small addressed output section within the linker script
- In general, the attribute is not used explicitely (implicit use by attribute section or asection)
- used only with extern declarations

Attribute aligned

Syntax

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

- Stores variables and functions aligned
- Alignment has to be greater than default alignment
- Alignment has to be a power of 2 (2^1, 2^2, 2^4)

Attribute packed

Syntax

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

- When stored aligned, there is space between structure elements
- Structure can be stored packed: no space between structure elements
- Access to variables of more than one byte in size is suboptimal

Example 1

Sourcecode

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

Generated Code

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

Example 2

Sourcecode

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

Generated 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")));
```

- Only 4-byte access allowed in PRAM
- Defines access type to variables
- Possible access: char (1), short (2) und int(4)
- `-maligned-access` is a precondition

Attribute interrupt

Syntax

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

- For interrupt service routines, the upper context hasn't to be saved another time
- jl to the function in the interrupt vector table
- At function's end: returns by using ji

Attribute interrupt_handler

Syntax

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

- jump to the functions in the interrupt vector table
- At function's end: returns by using rfe

Attribute longcall

Syntax

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

- Function is called by calli instead of call
- call can only address ± 16MByte
- Remote functions have to be called by calli

Attribute section

Syntax

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

- Stores variables and functions in user defined sections

Attribute asection

Syntax

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

Parameters

`name`	Section's name
`align`	Alignment in a power of 2
`flags`	Additional flags for section

$\Rightarrow$ asection is useable for functions as well as variables

Flags

`a`	allocatable (always set)
`x`	executable
`w`	writable
`p`	PCP section
`t`	small addressable (10 Bit)
`s`	small addressable (16 Bit)
`z`	absolut addressable
`b`	Bitsection

Miscellaneous

- Register relative ('small') addressed sections must start with .sdata. or .sbss.
- If the section contains code: flag x must be set

Pragma section I

Syntax

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

- Attribute asection only valid for one variable / function
- pragma section for an arbitrary number of variables / functions

Pragma section II

- Flags are the same as for attribute asection
- Same defaults for alignment and names (.sdata. resp. .zdata.)
- Pragma sections may not be nested
- Flag x set: only functions are locatated
- Flag x not set: only variables are locatated
- Names of sections have to be valid C identifiers (.sbss.1 is forbidden)

Pragma branch I

- Default branches can be defined for if−else- and switch−case statements else- oder case-Instruktion stehen
- The pragmas have to be placed immediately before the if-, else- or case instruction
- Exception: comments
- Each pragma applies to the next statement
- For `-mwarn-pragma-branch` a warning is raised, if no default branch is defined for an instruction

Pragma branch II

`#pragma branch_if_default`

The if-case is the default

`#pragma branch_else_default`

The else-case is the default branch_else_default

`#pragma branch_if_not_default`

synonym for branch_else_default

`#pragma branch_case_default`

Marks the default case within a switch block

`#pragma branch_no_default`

Neither the if- nor the else-case are default

Data Type _bit

- _bit declares bit variable
- Three flavours:
  - Global, not initialized: `_bit bit1;`
  - Global, initialized: `_bit bit2 = 1;`
  - Local to module, not initialized: `static _bit bit3;`
  - Local to module, initialized: `static _bit bit4 = 1;`

Implementation

- Implemented as unsigned int
- Range is 0 and 1
- Type casts and assignments from larger integral types will assign the LSB

GCC distinguishes between assignment and conditionals. Generally speaking:

Note

Conditionals will expand the bit to an int. In contrast, assignments will cast it to the smallest data type involved.

Permitted Operations

- Assignments
- Logic and binary unary and binary operators
- Testing
- Pointer to bits within a struct

Forbidden Operations

- ++, −− (post/pre increment/decrement)
- Unary minus (-b1)
- Indirection (address operator &)
- +, −, ∗, /, %, <<, >>
- +=, −=, ∗=, /=, %=, <<=, >>=
- Indirection (array/pointer/address)

Linking of Bit Variables

- The compiler reserves one byte per bit
- The linker packs bits into bytes (linker option `--relax` or `--relax-bdata`)
- Default sections: .bdata and .bbss
- User defined sections must be prefixed .bdata. or .bbss.
- Flag b of attributes asection and of pragma section must be set

Bits in the Map File

- Bit variables are displayed just like ordinary variables
- In addition to its address the position within the byte is displayed:

  0xa00001f0.2

Definition of Bit Fields

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

- Alignment if size is <= 8 bits: 1 byte
- Alignment if size is > 8 bits: 4 byte
- EABI: A bit field must not cross more than one 16 bit boundary
- volatile bit fields force access via ldmst

8 Sections
- Default and Bit Data Sections
- Addressing modes
- PCP, C++, Debug Sections

9 Linker Script File
- Internal Functions
- MEMORY and SECTION Command
- Initialising and Locating
- Locating Relative to End Address

10 Recommendations
- Programming the PCP
- Example

TriCore Sections (Part I)

Default Sections

`.text`	Code section
`.data`	Initialised data is in '.data'
`.bss`	Not initialised data is in '.bss'
`.rodata`	Storage of write-protected data
`.version_info`	Information about the compiler used for this module

For these default sections, the following subsections exist:
`.a1` `.a2` `.a4` `.a8`.

Bit Data Sections

`.bbss`	Not initialised bit data is placed in section '.bbss'
`.bdata`	Bit variables are placed in '.bdata'

TriCore Sections (Part II)

Small addressable sections

`.sdata`	Section '.sdata' stores initialised data which is addressable by small data area pointer (%a0)
`.sbss`	Not initialised data in section '.sbss', addressable by small data area pointer (%a0)
`.sdata.rodata`	Storage of write-protected data, which is addressed small

Absolutely Addressable Sections

`.zdata`	Initialised data, absolutely addressed
`.zbss`	Not initialised data, absolutely addressed
`.zrodata`	Write-protected data data, absolutely addressed

TriCore Sections (Part III)

PCP Sections

`.pcptext`	PCP Code Section
`.pcpdata`	PCP Data Section

C++ Sections

`.eh_frame`	Exception handling frame for C++ exceptions
`.ctors`	Section for constructors
`.ctors`	Section for destructors

Debug Sections

`.debug_<name>`
: Diverse Debug Sections

Basics

- Each object file consists of sections
- Each sections has a name and a size
- Loadable and non-loadable sections
- Sections, which include debug informations
- Mapping of input and output sections and memory partitioning
- Loadable and allocatable output sections have two addresses:

VMA

The virtual memory address specifies the address for program execution

LMA

The load memory address specifies the load address of a section

Internal Functions (Part I)

`ABSOLUTE(<exp>)`
: Returns the value of <exp>

`ADDR(<section>)`
: Returns the absolute addresse (VMA) of <section>

`LOADADDR(<section>)`
: Returns the absolute load memory address of <section>

`ALIGN(<exp>)`
: Returns the location counter (.) adjusted to the next <exp> boundary

Internal Functions (Part II)

`DEFINED(<symbol>)`

If <symbol> is defined and exists in the global symbol table, the return value is one, zero else

`PROVIDE(<symbol> = <expression>)`

Defines a symbol only, if it is referenced and not yet defined in an included object file zurck

`SIZEOF(<section>)`

Returns the size of the section in bytes

Important File Commands

`INCLUDE <filename>`
: Includes the linker script <filename>

`INPUT(<file>, <file>, ...), ...`
: Includes the given files into the link run

`GROUP(<file>, <file>, ...), ...`
: Similar to `INPUT`, except that the files should be archives.

`OUTPUT(<filename>)`
: `OUTPUT` returns the name of the output file

`SEARCH DIR(<path>)`
: Expands the search path by <path>

`STARTUP(<filename>)`
: The file, which has to be linked first

MEMORY Command

Describes location and size of memory partitions

Attributes

Attribute	Description
r	Read only
w	Read and write
p	PCP memory
x	Executable section
a	Allocatable section
i or l	Initialised section
!	Inverts all following attributes

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 Sections with Wildcard Patterns

'*'	Matches any number of arbitrary characters
'?'	Matches any single character
[<chars>]	Matches a single occurence of any of ¡chars¿. Can be specified as a range of characters, e.g. '[a-z]'.
\	Quotes the following pattern

Sorting of Data

Normally, the linker will place files and sections matched by wildcards in the order in which they are seen during the link. You can change this by using the SORT keyword, which appears before a wildcard pattern in parentheses (e.g., SORT(.text)). When the SORT keyword is used, the linker will sort the files or sections into ascending order by name before placing them in the output file.

SECTION Command

- Symbol assignment
- Description of output sections
- Rules for mapping input to output sections

> can be used to locate output sections in defined memory areas.

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
```

Description Output Section

Syntax

```
section [address] [(type)] : [AT(lma)]
 {
    output-section-command
    output-section-command

 } [>region] [AT>lma_region] [:phdr :phdr ] [=fillexpr]
```

Example

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

Initialising of Tables (Part 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) ;
```

Initialising of Tables (Part 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
```

Locating of Sections (Part I)

PCP text section

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

or

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

Locating of Sections (Part II)

PCP data section

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

Locating of Sections (Part II)

Data section

```
.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)
```

Locating Relative to End Addresse (Part I)

Problem

Locating of a section relative to a fixed end address

The linker can only locate sections relative to a start address and is particularly not able to handle forward references.

Solution

Two link passes

First pass

Compute size of section

Second pass

Define start address as (End address - size)

Locating Relative to End Addresse (Part II)

Linker description file entries

```
___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(.) ;
}
```

Locating Relative to End Addresse (Part III)

First link pass

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

Computing of size the from the link outcome using mksyms

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

results in

```
___startof_libc_mydata = 0xa07fffec;
```

Second Pass of Linking

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

Locating Relative to End Addresse (Part IV)

Corresponding 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) $(addsuffix .pass1,$@) $(addsuffix .syms,$@)\
            $(addsuffix .map1,$@)
```

Programming the PCP

- PCP code and PCP data have their own sections: .pcptext und .pcpdata
- Provided in the default linker script: .pcptext.* und .pcpdata.*
- Functions and variables are placed in these sections by using attribute section or pragma section
- The startup code copies PCP code and PCP data into suitable memory areas
- Only inline assebler allowed inside a PCP section

Example

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

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

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

Mapfile

Output in mapfile (abbreviated)

```
.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.*)
```

Example

```
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
/* Suboptimal */
#if CASE==3
    out1_u8 = (unsigned char)(in_u16 > 255 ? (unsigned short)255 : in_u16);
    out2_u8 = out1_u8;
#endif
}
```

Contact

Any further questions?

Contact us:

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