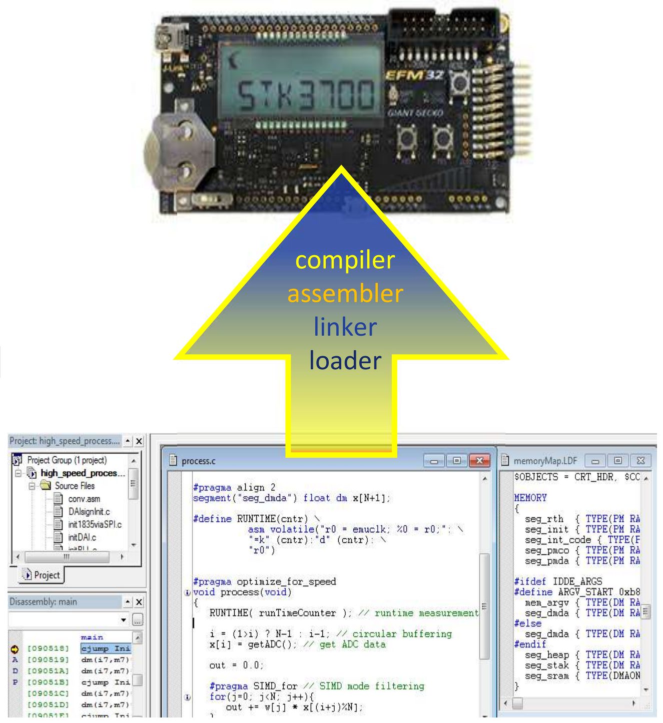
Embedded Software Development 2025.09.22.

SW development environment, compiler

Department of
Artificial Intelligence and
Systems Engineering

SW development environment

- IDE: Integrated Development Environment
- Duty of SW development environment:
  - Gives a frame for the available toolchains (program modules), like:
    - Compiler: generates low level assembly code from high level code
    - Assembler: generates machine code from low level assembly code
    - Linker: merge the numerous compilation files
  - Compilation
  - Debug
  - „Texting”: assistance in writing the code
    - code highlighting
    - automatic completion
    - tracking functions and definition of variables
    - …

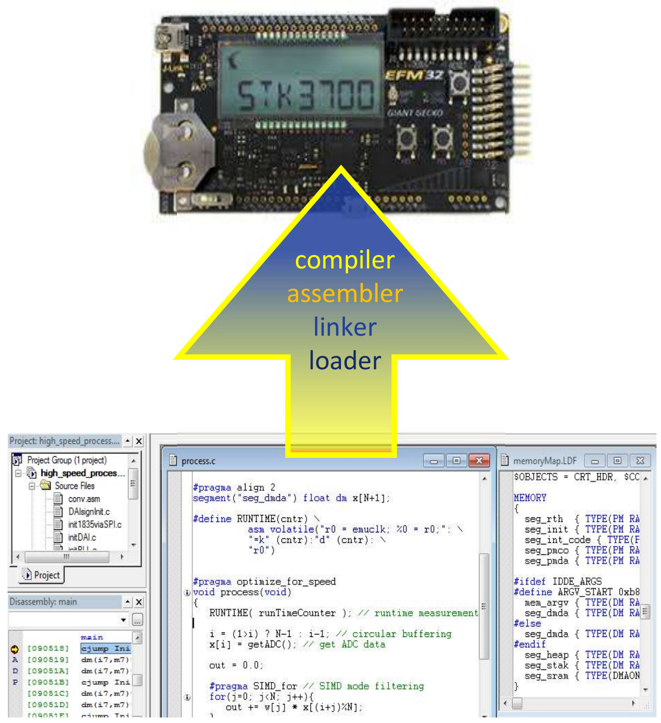


```
#pragma align 2
segment("seg_dmda") float dm x[N+1];

#define RUNTIME(cntr) \
        asm volatile("r0 = emuclk; %0 = r0;": \
        "=k" (cntr):"d" (cntr): \
        "r0")


#pragma optimize_for_speed
void process(void)
{
   RUNTIME( runTimeCounter ); // runtime measurement

   i = (1>i) ? N-1 : i-1; // circular buffering
   x[i] = getADC(); // get ADC data

   out = 0.0;

   #pragma SIMD_for // SIMD mode filtering
   for(j=0; j<N; j++){
      out += w[j] * x[(i+j)%N];
```

```
$OBJECTS = CRT_HDR, $CC

MEMORY
{
  seg_rth  { TYPE(PM RA
  seg_init { TYPE(PM RA
  seg_int_code { TYPE(F
  seg_pmco { TYPE(PM RA
  seg_pmda { TYPE(PM RA

#ifdef IDDE_ARGS
#define ARGV_START 0xb8
  mem_argv { TYPE(DM RA
  seg_dmda { TYPE(DM RA
#else
  seg_dmda { TYPE(DM RA
#endif
  seg_heap { TYPE(DM RA
  seg_stak { TYPE(DM RA
  seg_sram { TYPE(DMAON
}
```

SW development environment

- Duty of SW development environment (cont'd):
    - Handling/storing project settings
    - Downloading and running the program
    - Handling the connected embedded HW systems
    - Intelligent handling of error messages
    - Setting up the HW configuration



```
main
[090518]  cjump Ini
[090519]  dm(i7,m7)
[09051A]  dm(i7,m7)
[09051B]  cjump Ini
[09051C]  dm(i7,m7)
[09051D]  dm(i7,m7)
```

```
#pragma align 2
segment("seg_dmda") float dm x[N+1];

#define RUNTIME(cntr) \
        asm volatile("r0 = emuclk; %0 = r0;": \
        "=k" (cntr):"d" (cntr): \
        "r0")


#pragma optimize_for_speed
void process(void)
{
   RUNTIME( runTimeCounter ); // runtime measurement

   i = (1>i) ? N-1 : i-1; // circular buffering
   x[i] = getADC(); // get ADC data

   out = 0.0;

   #pragma SIMD_for // SIMD mode filtering
   for(j=0; j<N; j++){
      out += w[j] * x[(i+j)%N];
   }
```

```
$OBJECTS = CRT_HDR, $CC

MEMORY
{
  seg_rth  { TYPE(PM RA
  seg_init { TYPE(PM RA
  seg_int_code { TYPE(F
  seg_pmco { TYPE(PM RA
  seg_pmda { TYPE(PM RA

#ifdef IDDE_ARGS
#define ARGV_START 0xb8
  mem_argv { TYPE(DM RA
  seg_dmda { TYPE(DM RA
#else
  seg_dmda { TYPE(DM RA
#endif
  seg_heap { TYPE(DM RA
  seg_stak { TYPE(DM RA
  seg_sram { TYPE(DMAON
}
```

SW development environments for embedded systems

- Not easy to provide comprehensive summary since unlike PC approach, in the embedded field many processors and platforms and so many development environments exist
  - Special features → special compilers
  - Different architectures and instruction set
- Debugging is difficult since the processor is an autonomous unit that cannot be accessed directly by PC
- Relationship between the compiler and the graphical user interface (GUI):
  - Compiler (and other supplementary program tools) and the GUI build up a complex system (e.g. provided by the manufacturer)
  - General toolchain (e.g. gcc compiler) + editor (e.g. Eclipse env.)

An example: Simplicity Studio

- SW development environment in the course: Silicon Laboratories (SiLabs): Simplicity Studio
- Architecture:
  - Eclipse-based GUI
  - gcc-based compiler
  - GUI helps in exploiting the services offered by the compiler

Compilation steps

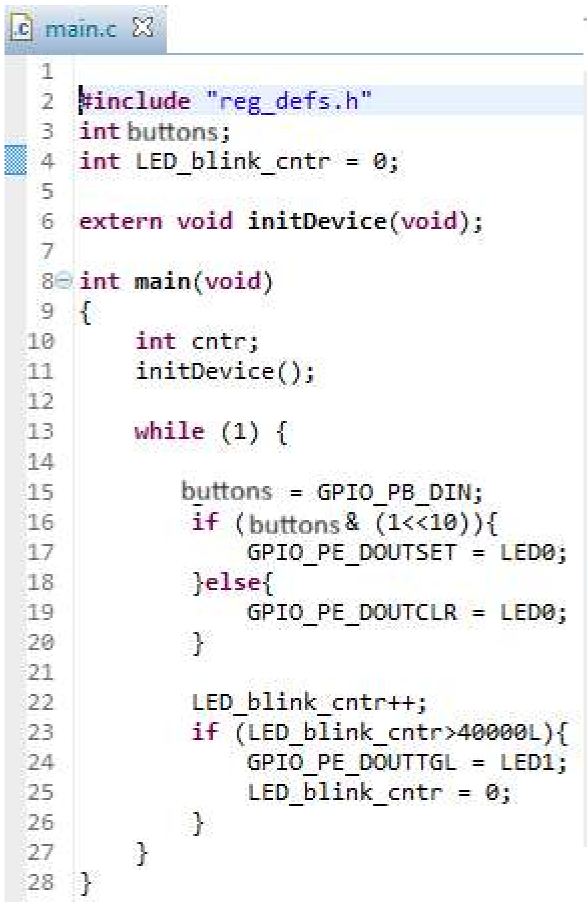
- Source codes in C → Assembly code/object file
- Assembly code → object file
  - object file: the file compiled into a machine code + extra auxiliary information for the linker
- Linker: integrates object files
  - Generating the whole machine code from the program
  - Based on the auxiliary information places real addresses in the code (e.g. resolving function- or variable links from different C-language files)
  - Storing variables and functions in the memory
    - Linker file contains information of which variable stored into which segment of the memory

Compilation steps

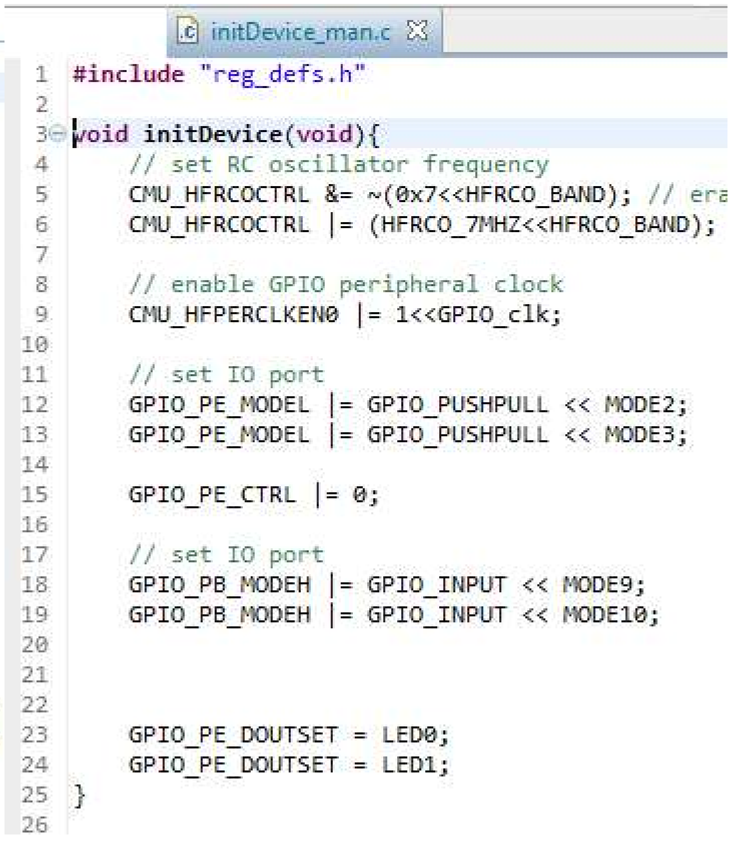
- Typical ‘intercompilation’ files containing auxiliary information:
  - .i : file processed by the preprocessor (e.g. substitution of #define-s)
  - .s : asm file
  - .o : object file
  - .d : file containing dependencies (e.g. main.c file contains init.c)
  - .axf: (ARM) object file containing (among others) debug information
  - .map: memory map
- Final result of the compilation: files that can be loaded on the embedded unit, e.g. development board (.hex, .bin, …)

Compilation process example

- Example: handling buttons:
  - Inintialization
  - Read button state, setting LED
  - LED blinks repeatedly
- Files:
  - main.c
  - initDevice_man.c
  - reg_defs.h
  - startup_gcc_efm32gg.s (startup code: provided by the manufacturer for initialization purposes)

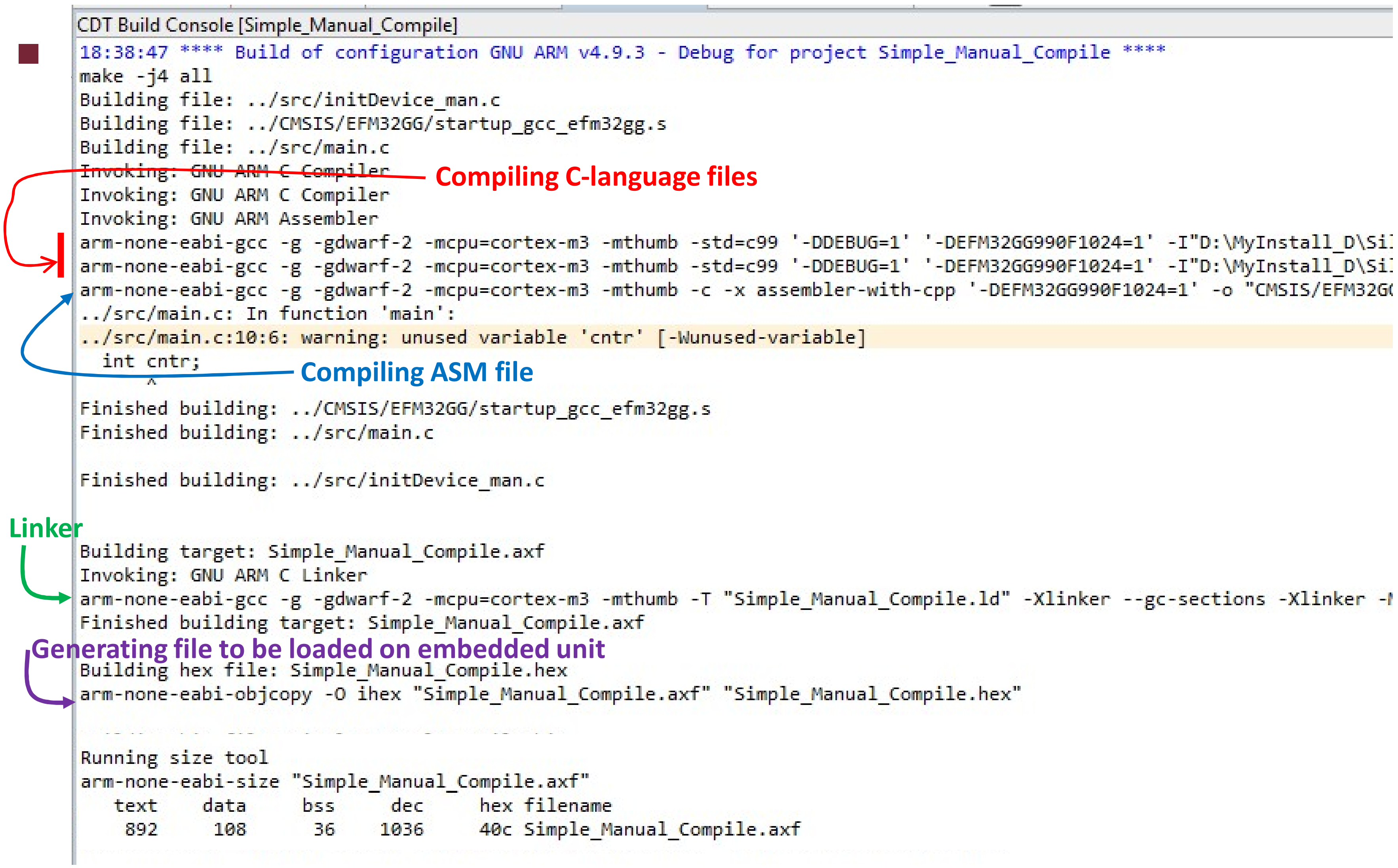

main.c

```c

#include "reg_defs.h"
int buttons;
int LED_blink_cntr = 0;

extern void initDevice(void);

int main(void)
{
    int cntr;
    initDevice();

    while (1) {

        buttons = GPIO_PB_DIN;
         if (buttons & (1<<10)){
              GPIO_PE_DOUTSET = LED0;
         }else{
              GPIO_PE_DOUTCLR = LED0;
         }

         LED_blink_cntr++;
         if (LED_blink_cntr>40000L){
              GPIO_PE_DOUTTGL = LED1;
              LED_blink_cntr = 0;
         }
    }
}
```


initDevice_man.c

```c
#include "reg_defs.h"

void initDevice(void){
    // set RC oscillator frequency
    CMU_HFRCOCTRL &= ~(0x7<<HFRCO_BAND); // era
    CMU_HFRCOCTRL |= (HFRCO_7MHZ<<HFRCO_BAND);

    // enable GPIO peripheral clock
    CMU_HFPERCLKEN0 |= 1<<GPIO_clk;

    // set IO port
    GPIO_PE_MODEL |= GPIO_PUSHPULL << MODE2;
    GPIO_PE_MODEL |= GPIO_PUSHPULL << MODE3;

    GPIO_PE_CTRL |= 0;

    // set IO port
    GPIO_PB_MODEH |= GPIO_INPUT << MODE9;
    GPIO_PB_MODEH |= GPIO_INPUT << MODE10;



    GPIO_PE_DOUTSET = LED0;
    GPIO_PE_DOUTSET = LED1;
}

```

Compilation process example



CDT Build Console [Simple_Manual_Compile]
Compiling C-language files
Compiling ASM file
Linker
Generating file to be loaded on embedded unit

```
18:38:47 **** Build of configuration GNU ARM v4.9.3 - Debug for project Simple_Manual_Compile ****
make -j4 all
Building file: ../src/initDevice_man.c
Building file: ../CMSIS/EFM32GG/startup_gcc_efm32gg.s
Building file: ../src/main.c
Invoking: GNU ARM C Compiler
Invoking: GNU ARM C Compiler
Invoking: GNU ARM Assembler
arm-none-eabi-gcc -g -gdwarf-2 -mcpu=cortex-m3 -mthumb -std=c99 '-DDEBUG=1' '-DEFM32GG990F1024=1' -I"D:\MyInstall_D\Si
arm-none-eabi-gcc -g -gdwarf-2 -mcpu=cortex-m3 -mthumb -std=c99 '-DDEBUG=1' '-DEFM32GG990F1024=1' -I"D:\MyInstall_D\Si
arm-none-eabi-gcc -g -gdwarf-2 -mcpu=cortex-m3 -mthumb -c -x assembler-with-cpp '-DEFM32GG990F1024=1' -o "CMSIS/EFM32G
../src/main.c: In function 'main':
../src/main.c:10:6: warning: unused variable 'cntr' [-Wunused-variable]
  int cntr;
      ^
Finished building: ../CMSIS/EFM32GG/startup_gcc_efm32gg.s
Finished building: ../src/main.c

Finished building: ../src/initDevice_man.c


Building target: Simple_Manual_Compile.axf
Invoking: GNU ARM C Linker
arm-none-eabi-gcc -g -gdwarf-2 -mcpu=cortex-m3 -mthumb -T "Simple_Manual_Compile.ld" -Xlinker --gc-sections -Xlinker -
Finished building target: Simple_Manual_Compile.axf

Building hex file: Simple_Manual_Compile.hex
arm-none-eabi-objcopy -O ihex "Simple_Manual_Compile.axf" "Simple_Manual_Compile.hex"

Running size tool
arm-none-eabi-size "Simple_Manual_Compile.axf"
   text    data     bss     dec     hex filename
    892     108      36    1036     40c Simple_Manual_Compile.axf
```

‘Manual’ compilation

- Let us be a ‘manual’ compiler

```
SET COMP="d:\MyInstall_D\SiliconLabs\SimplicityStudio\developer\toolchains\gnu_arm\4.9_2015q3\bin\„
```

Compilation of C files:

```
%COMP%arm-none-eabi-gcc
        -g -gdwarf-2 -mcpu=cortex-m3 -mthumb -std=c99
        -D DEBUG=1 -D EFM32GG990F1024=1 -I ./src
        -O0 -Wall -c -fmessage-length=0 -mno-sched-prolog -fno-builtin -ffunction-sections -fdata-sections
        -MMD -MP -MF"initDevice_man.d" -MT"initDevice_man.o"
        -o "initDevice_man.o" "initDevice_man.c„

%COMP%arm-none-eabi-gcc
        -g -gdwarf-2 -mcpu=cortex-m3 -mthumb -std=c99
        -D DEBUG=1 -D EFM32GG990F1024=1 -I ./src
        -O0 -Wall -c -fmessage-length=0 -mno-sched-prolog -fno-builtin -ffunction-sections -fdata-sections
        -MMD -MP -MF"main.d" -MT"main.o"
        -o "main.o" "main.c„
```

Compilation of ASM file:

```
%COMP%arm-none-eabi-gcc
        -g -gdwarf-2 -mcpu=cortex-m3 -mthumb -c -x assembler-with-cpp
        -D EFM32GG990F1024=1
        -o "startup_gcc_efm32gg.o" "startup_gcc_efm32gg.s"
```

‘Manual’ compilation

- Let us be a ‘manual’ compiler

Linking:

```
%COMP%arm-none-eabi-gcc
        -g -gdwarf-2 -mcpu=cortex-m3 -mthumb
        -T "Simple_Manual_Compile.ld" -Xlinker --gc-sections -Xlinker -Map="Simple_Manual_Compile.map"
        --specs=nano.specs
        -o Simple_Manual_Compile.axf
        "startup_gcc_efm32gg.o" "main.o" "initDevice_man.o"
        -Wl,--start-group -lgcc -lc -lnosys -Wl,--end-group
```

Generating the file to be downloaded to the embedded device

```
%COMP%arm-none-eabi-objcopy -O ihex "Simple_Manual_Compile.axf" "Simple_Manual_Compile.hex"
```

Calculating size:

```
%COMP%arm-none-eabi-size "Simple_Manual_Compile.axf"
```

Generating dissassemby file:

```
%COMP%arm-none-eabi-objdump -S --disassemble Simple_Manual_Compile.axf > Simple_Manual_Compile.dump
```

Meaning of compile switches (C compiler)

```
%COMP%arm-none-eabi-gcc
        -g -gdwarf-2 -mcpu=cortex-m3 -mthumb -std=c99
        -D DEBUG=1 -DBLINK_DELAY=4000L -D EFM32GG990F1024=1 -I ./src
        -O0 -Wall -c -fmessage-length=0 -mno-sched-prolog -fno-builtin -ffunction-sections -fdata-sections
        -MMD -MP -MF"initDevice_man.d" -MT"initDevice_man.o"
        -o "initDevice_man.o" "initDevice_man.c„
```

- -g -gdwarf-2: saving debug information into dwarf-2 format
- -D DEBUG=1 –D BLINK_DELAY=4000L -D EFM32GG990F1024=1 : as if these variables have been given by #define in all our program files. By this, conditional compilation or general parameters can be given, e.g. type of processor
- -I ./src: libraries can be given where to search for included files
- -mcpu=cortex-m3: type of CPU for which the compilation is done
- -mthumb: thumb instruction set (16-bit reduced instruction set)
- -std=c99: the C-language standard used
- -O0: optimization level: 0, no optimization
  - Possible levels: O0…O3, Os: optimization for size
- -Wall -c -fmessage-length=0: all warnings are on, messages are not truncated (instead of 0 truncation length can be given)

Meaning of compile switches (C compiler)

```
%COMP%arm-none-eabi-gcc
            -g -gdwarf-2 -mcpu=cortex-m3 -mthumb -std=c99
             -D DEBUG=1 -DBLINK_DELAY=4000L -D EFM32GG990F1024=1 -I ./src
            -O0 -Wall -c -fmessage-length=0 -mno-sched-prolog -fno-builtin -ffunction-sections -fdata-sections
            -MMD -MP -MF"initDevice_man.d" -MT"initDevice_man.o"
            -o "initDevice_man.o" "initDevice_man.c„
```

- -mno-sched-prolog: in case of functions their header (stack pointer handling, parameter handling…) is coded separately in a non-optimized manner, not included into the function body -> easier to read and debug the assembly code
- -fno-builtin: the embedded C-language functions are not optimized but appears in the compiled code -> easier to read and debug the assembly code
- -ffunction-sections -fdata-sections: the compiler will not mix the data and the program but they are stored in dedicated memory segments -> easier to debug
- -MMD -MP -MF"initDevice_man.d" -MT"initDevice_man.o„: generate the dependency structure of files and saves it into a file with extension of .d (e.g. which file uses variables and functions of other files)
  - Example: content of main.d: src/main.o: ../src/main.c ../src/reg_defs.h
- -o "initDevice_man.o" "initDevice_man.c„: from initDevice_man.c file initDevice_man.o output object file if generated

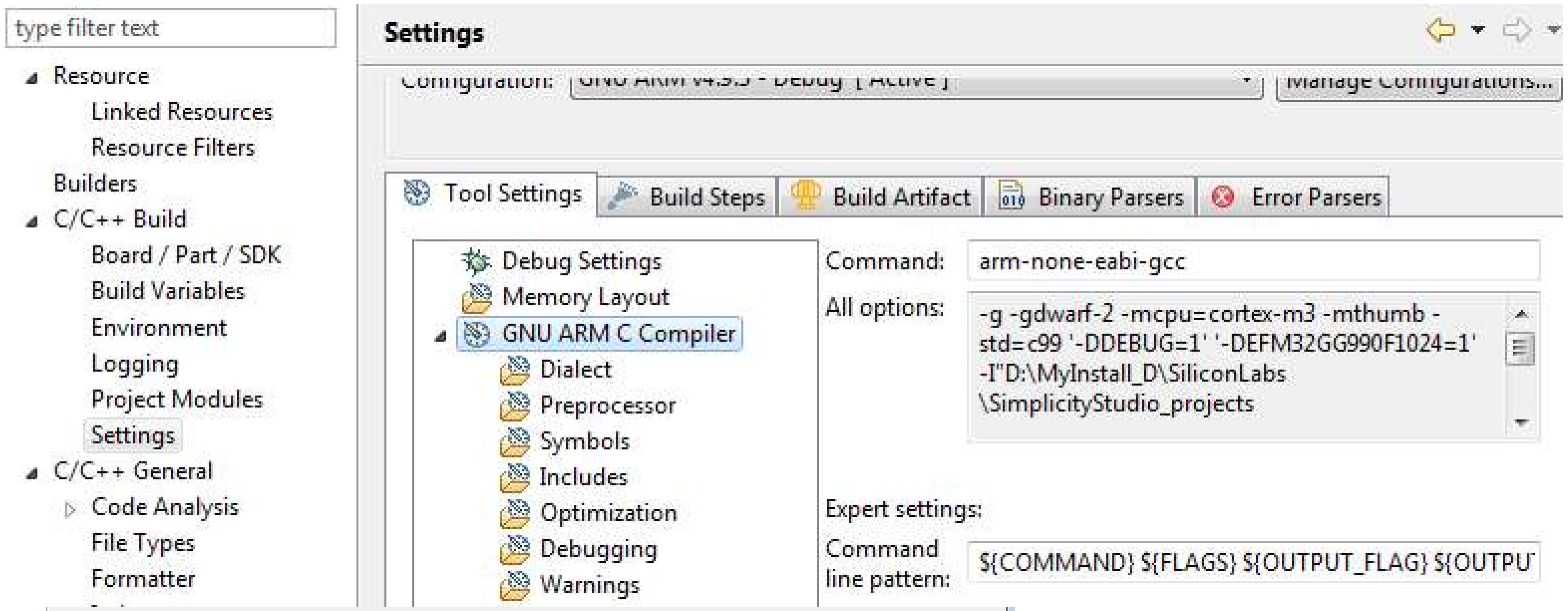
Meaning of compile switches (linker)

```
%COMP%arm-none-eabi-gcc
          -g -gdwarf-2 -mcpu=cortex-m3 -mthumb
          -T "Simple_Manual_Compile.ld" -Xlinker --gc-sections -Xlinker -Map="Simple_Manual_Compile.map"
          --specs=nano.specs
          -o Simple_Manual_Compile.axf
          "startup_gcc_efm32gg.o" "main.o" "initDevice_man.o"
          -Wl,--start-group -lgcc -lc -lnosys -Wl,--end-group
```

- -T "Simple_Manual_Compile.ld„: linker file. This file defines at which memory address the data program code should be stored. The memory can be segmented into more parts
- -Xlinker: the command followed by this switch is passed to the linker
- -Xlinker --gc-sections: tries to leave out the non-used functions (only if they are compiled using switches -ffunction-sections and -fdata-sections)
- -Xlinker -Map="Simple_Manual_Compile.map„: providing map file
- --specs=nano.specs: special command file given to the linker
- -o Simple_Manual_Compile.axf: output file
- "startup_gcc_efm32gg.o" "main.o" "initDevice_man.o„: these files are linked into a single source file
- -Wl,--start-group -lgcc -lc -lnosys -Wl,--end-group: not interested for us

Setting of compile switches

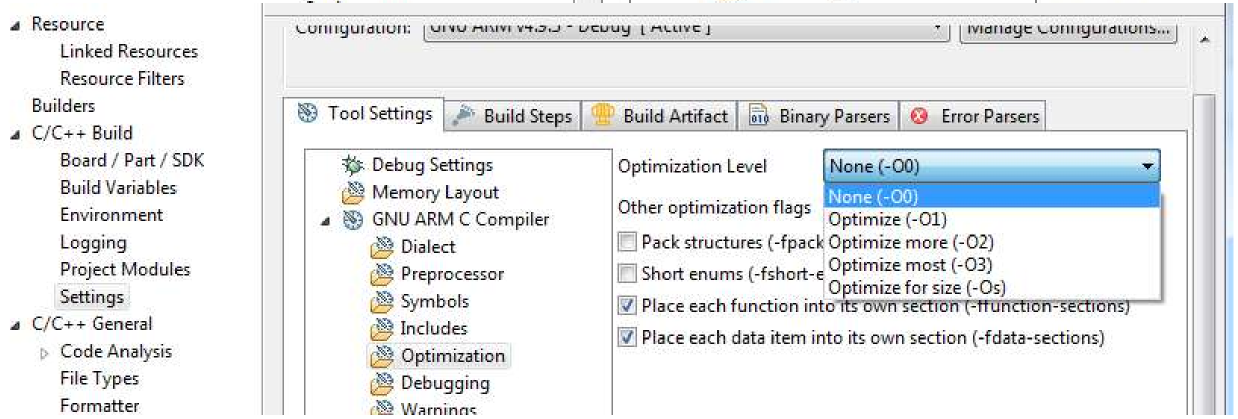
- Development environ. generates appropriate switches



Command: arm-none-eabi-gcc

All options: -g -gdwarf-2 -mcpu=cortex-m3 -mthumb -std=c99 '-DDEBUG=1' '-DEFM32GG990F1024=1' -I"D:\MyInstall_D\SiliconLabs \SimplicityStudio_projects

Command line pattern: ${COMMAND} ${FLAGS} ${OUTPUT_FLAG} ${OUTPU



Optimization Level: None (-O0)

Other optimization flags

☐ Pack structures (-fpack

☐ Short enums (-fshort-e

☑ Place each function into its own section (-ffunction-sections)

☑ Place each data item into its own section (-fdata-sections)

Standard configurations

- Generally standard configurations exist, typically:
  - Debug: for development
    - Contains more debug information, code can be read better, using switch **-mno-sched-prolog**
  - Release: final product

No optimization

Debug:

-g -gdwarf-2 -mcpu=cortex-m3 -mthumb -std=c99 **'-DDEBUG=1'** '-DEFM32GG990F1024=1' **-O0** -Wall -c -fmessage-length=0 **-c -save-temps -mno-sched-prolog -fno-builtin** -ffunction-sections -fdata-sections

Saves temporary files as well (e.g.assembly)

Keep the function header in one piece

Release:

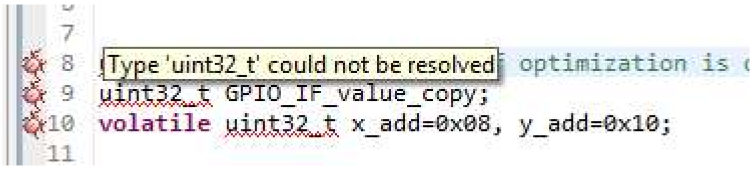
-g -gdwarf-2 -mcpu=cortex-m3 -mthumb -std=c99 **'-DNDEBUG=1'** '-DEFM32GG990F1024=1' **-O3** -Wall -c -fmessage-length=0 -ffunction-sections -fdata-sections

Configuration of compiler in source code

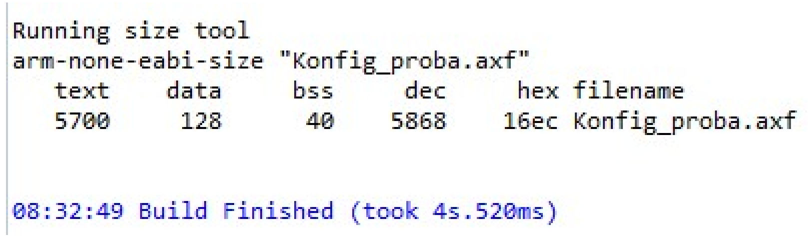
- **#pragma** directive: giving compiler specific settings
- #pragma GCC optimize("O3")
  - Setting optimization level for a certain code segment
- #pragma optimize for speed
  - E.g. Analog Devices DSP-s: a kind of optimization again
- #pragma SIMD_for
  - Where SIMD (Single Instruction Multiple Dada) is applicable
- #pragma message “message” -> e.g. “it needs more development”
  - Writes a message during compilation
- #pragma push → #pragma pop: saving and fetching the settings
- #pragma once: the header file is included only once

Development-compiler relationship

- Development environment and compiler are two separated SW units
- Theoretically the same rules are applied for both but inconsistencies may occur
  - Example:
    - development environment finds and error (uint32 cannot be resolved) but
    - the compiler compiles the project without even a warning



```
Type 'uint32_t' could not be resolved optimization is
uint32_t GPIO_IF_value_copy;
volatile uint32_t x_add=0x08, y_add=0x10;
```



```
Running size tool
arm-none-eabi-size "Konfig_proba.axf"
   text    data     bss     dec     hex filename
   5700     128      40    5868    16ec Konfig_proba.axf


08:32:49 Build Finished (took 4s.520ms)
```

Automatic compilation

- Many compiler use command `make`
  - Originally developed for UNIX system as an auxiliary program (used since 1976)
  - Can be used for automate compilation (or in other cases, generally when files has to be generated from other files based on certain rules, e.g. automatic program installation)
  - The `makefile` contains compilation rules
  - The compiler calls the *make* program that search for the *make file* of the project. Based on the rules found in the makefile, the source code is compiled and files are generated.
  - *Make* command has switches, e.g. –jN, N: number of processes run parallel (like in Simplicity Studio)
- The standardized structure makes possible the use of the same compiler for various graphical development environment or even the manual compilation

structure of `makefile`

- The makefile contains rules
- Structure of rules (dependencies)

target file: precondition(s)

instruction(s) [starts with a Tab]

Example:

```
main.o: main.c
        gcc -o main.o  main.c
```

The main.o file depends on main.c file (generated from that). A main.o file is generated by using command gcc

- Instruction(s) are executed if:
  - If the target file still not exists
  - The program checks the dates of target and precondition files in the dependencies. Instructions are executed only if precondition files are generated later than the target files
    - Checking dates saves time by not performing unnecessary compilation

‘manual’ makefile

- Compilation of previous example given in makefile

```
# this is a comment
#giving the path of compiler COMP in a variable. Later can be used $COMP$ as a reference of the compiler
COMP := d:\MyInstall_D\SiliconLabs\SimplicityStudio\developer\toolchains\gnu_arm\4.9_2015q3\bin\\

#all: default target. Final file is axf file
all: Simple_Manual_Compile.axf

#First dependence: what is needed for generating Simple_Manual_Compile.axf file:
#If files with .o extensions are not available then based on the applicable rules those are generated (see next page)

Simple_Manual_Compile.axf: startup_gcc_efm32gg.o main.o initDevice_man.o
        @echo ' ‘#@echo: writing text
        @echo 'Simple_Manual_Compile.axf compilation‘
         # link command is given here (see previous example)
        $(COMP)arm-none-eabi-gcc -g -gdwarf-2 -mcpu=cortex-m3 -mthumb -T Simple_Manual_Compile.ld
        -Xlinker --gc-sections -Xlinker -Map=Simple_Manual_Compile.map --specs=nano.specs
        -o Simple_Manual_Compile.axf startup_gcc_efm32gg.o main.o initDevice_man.o –
        Wl,--start-group -lgcc -lc -lnosys -Wl,--end-group
         # hex file generation
        $(COMP)arm-none-eabi-objcopy -O ihex Simple_Manual_Compile.axf Simple_Manual_Compile.hex
```

‘manual’ makefile

```
# cont.

# startup_gcc_efm32gg.s  compilation by assembler. Generation of the startup_gcc_efm32gg.o file.
startup_gcc_efm32gg.o: startup_gcc_efm32gg.s
                @echo ' '
                @echo 'startup_gcc_efm32gg.s compilation'
                $(COMP)arm-none-eabi-gcc -g -gdwarf-2 -mcpu=cortex-m3 -mthumb -c -x assembler-with-cpp
                -D EFM32GG990F1024=1 -o startup_gcc_efm32gg.o startup_gcc_efm32gg.s

# Compilation of C-language files → generation of object files (compilation parameters are found in the previous
example)
main.o: main.c
                @echo ' '
                @echo 'main.c compilation'
                $(COMP)arm-none-eabi-gcc -g -gdwarf-2 -mcpu=cortex-m3 -mthumb -std=c99 -D DEBUG=1
                 -D EFM32GG990F1024=1 -O0 -Wall -c -fmessage-length=0 -mno-sched-prolog -fno-builtin
                -ffunction-sections -fdata-sections -MMD -MP -MFmain.d -MTmain.o -o main.o main.c

initDevice_man.o: initDevice_man.c
                @echo ' '
                @echo 'initDevice_man.c compilation'
                $(COMP)arm-none-eabi-gcc -g -gdwarf-2 -mcpu=cortex-m3 -mthumb -std=c99 -D DEBUG=1
                -D EFM32GG990F1024=1 -O0 -Wall -c -fmessage-length=0 -mno-sched-prolog -fno-builtin
                -ffunction-sections -fdata-sections -MMD -MP -MFinitDevice_man.d -MTinitDevice_man.o
                -o initDevice_man.o initDevice_man.c
```

structure of `makefile`

- Previous examples are ‘simple’. Make program includes many parameters, it has automatic variables, using them results compact but hard-to-understand rules.:
  - $@: name of target
  - $<: list of preconditions
  - Extension-based rules, example: generate every c file into object file:
    - .c.o:

      ```
      gcc  $<  -o $@
      ```
  - Pattern matching (%: all non-zero string):
    - %.o: %.c

      ```
      gcc  $<  -o $@
      ```
- Variables in the program can be accessed between $$. Example:
  - PATH=C:\MCU\
  - $PATH$header.h → C:\MCU\header.h

Basic properties of make

- Typical targets: all, clean
  - all: compilation of everything (see previous examples)
  - clean: delete generated files. Worth to use it when something behaves in a strange manner, e.g. a file has been modified but the consequences cannot be seen.
    - example:

```
clean:
        rm *.c *.o  Simple_Manual_Compile.axf
```

  - .PHONY: all clean dependents
    - Indicates that these are not real targets, therefore no need to generate 'all' file
- Some variables are declared implicitly, like:
  - $(CC) : C compiler
  - $(CFLAGS) : parameters of C compiler
  - $(LDFLAGS): linker flags
  - $(RM) : remove command

Makefile hierarchy of template project

- Editing makefile manually is extremely rare since in most cases it is generated by the development environment.
- Example: makefile hierarchy of template project
  - The makefile found in the source library includes other makefiles that are necessary for the compilation of other files of the project
- See some example of makefiles in a simplified form (some parts are ignored for better understanding)

makefile (automatically generated)

```
################################################################################
# Automatically-generated file. Do not edit!
################################################################################

-include ../makefile.init

RM := rm -rf

# All of the sources participating in the build are defined here
-include sources.mk
-include src/subdir.mk
-include CMSIS/EFM32GG/subdir.mk
-include subdir.mk
-include objects.mk

-include ../makefile.defs
```

Including makefiles that belong other source files of the project

makefile (automatically generated)

```
# All Target
all: Simple_Manual_Compile.axf

# Tool invocations
Simple_Manual_Compile.axf: $(OBJS) $(USER_OBJS)
		@echo 'Building target: $@'
		@echo 'Invoking: GNU ARM C Linker'
		arm-none-eabi-gcc[…comp. switches…]Simple_Manual_Compile.axf "./CMSIS/EFM32GG/startup_gcc_efm32gg.o"
		"./src/initDevice_man.o" "./src/main.o…
		@echo 'Finished building target: $@'
		@echo ' '

		@echo 'Building hex file: Simple_Manual_Compile.hex'
		arm-none-eabi-objcopy -O ihex "Simple_Manual_Compile.axf" "Simple_Manual_Compile.hex"
		@echo ' '

		@echo 'Running size tool'
		arm-none-eabi-size "Simple_Manual_Compile.axf"
		@echo ' '

# Other Targets
clean:
		-$(RM) $(EXECUTABLES)$(OBJS)$(C_DEPS) Simple_Manual_Compile.axf
		-@echo ' ‘
```

Rule for .axf file: generated from what object files and how (switches are ignored for better understanding)

Deleting all files: executed when Clean Project is called

Example for an included file (subdir.mk)

```
################################################################################
# Automatically-generated file. Do not edit!
################################################################################
# Add inputs and outputs from these tool invocations to the build variables
C_SRCS += \../src/initDevice_man.c \../src/main.c

OBJS += \./src/initDevice_man.o \./src/main.o

C_DEPS += \./src/initDevice_man.d \./src/main.d

# Each subdirectory must supply rules for building sources it contributes
src/initDevice_man.o: ../src/initDevice_man.c
            @echo 'Building file: $<'
            @echo 'Invoking: GNU ARM C Compiler'
            arm-none-eabi-gcc [… comp. switches …] -o "$@" "$<"
            @echo 'Finished building: $<'
            @echo ' '
```

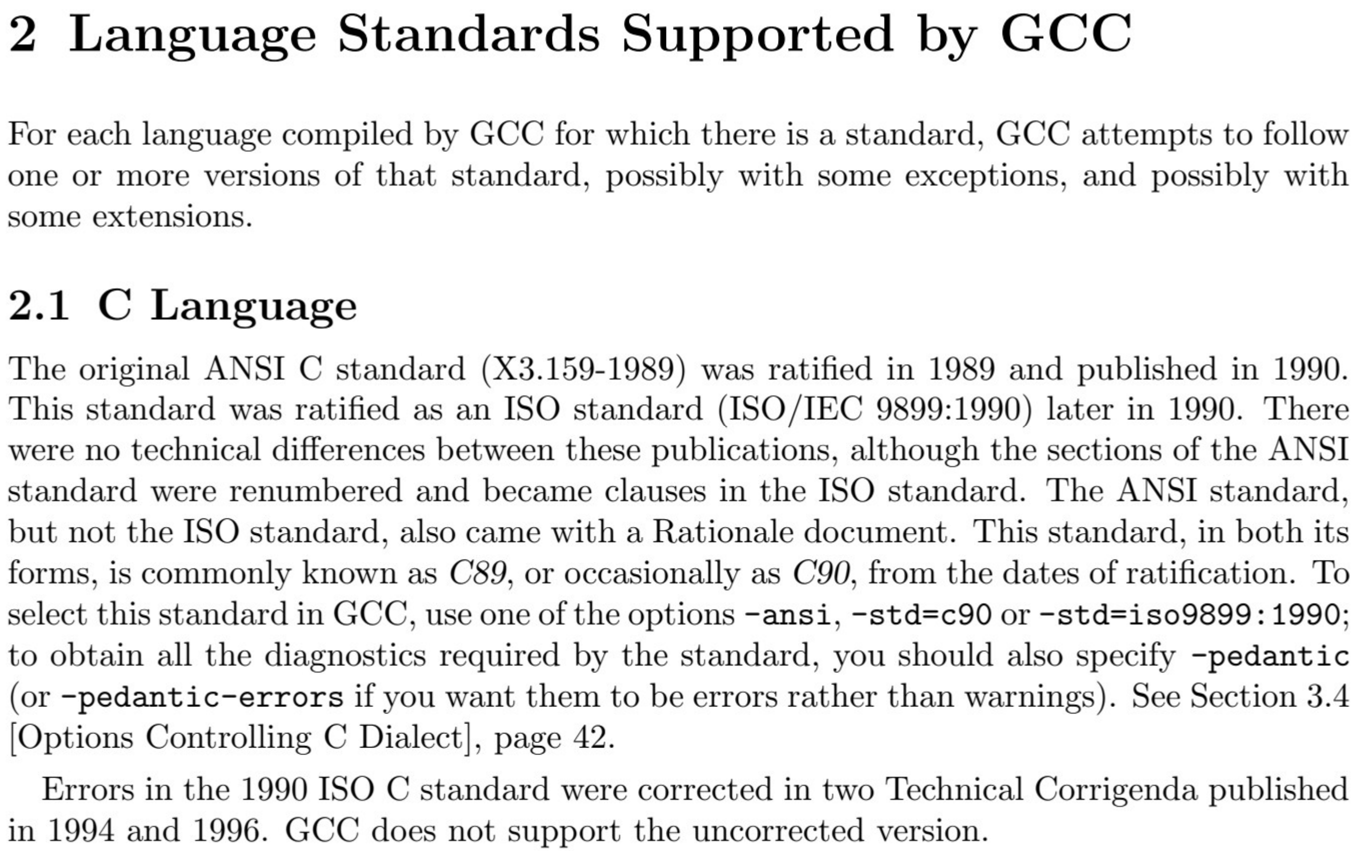
Rule for the compilation of initDevice_man.c file

```
src/main.o: ../src/main.c
            @echo 'Building file: $<'
            @echo 'Invoking: GNU ARM C Compiler'
            arm-none-eabi-gcc [… comp. switches …] $@" "$<"
            @echo 'Finished building: $<'
            @echo ' '
```

Rule for the compilation of main.c file

GCC compiler manual (.pdf)

- Search the internet for GCC.pdf to find the comprehensive description of the GCC compiler
- Example: See below the explanation of C language versions:



2 Language Standards Supported by GCC

For each language compiled by GCC for which there is a standard, GCC attempts to follow one or more versions of that standard, possibly with some exceptions, and possibly with some extensions.

2.1 C Language

The original ANSI C standard (X3.159-1989) was ratified in 1989 and published in 1990. This standard was ratified as an ISO standard (ISO/IEC 9899:1990) later in 1990. There were no technical differences between these publications, although the sections of the ANSI standard were renumbered and became clauses in the ISO standard. The ANSI standard, but not the ISO standard, also came with a Rationale document. This standard, in both its forms, is commonly known as *C89*, or occasionally as *C90*, from the dates of ratification. To select this standard in GCC, use one of the options `-ansi`, `-std=c90` or `-std=iso9899:1990`; to obtain all the diagnostics required by the standard, you should also specify `-pedantic` (or `-pedantic-errors` if you want them to be errors rather than warnings). See Section 3.4 [Options Controlling C Dialect], page 42.

Errors in the 1990 ISO C standard were corrected in two Technical Corrigenda published in 1994 and 1996. GCC does not support the uncorrected version.