AVR32006 : Getting started with GCC for AVR32

32-bit AVR® Microcontrollers

Application Note

Features

- This guide will give some tips and tricks for using the GNU Compiler Collection (GCC) for developing C/C++ code for AVR32.
- Optimization techniques for the compiler and linker is also provided
- The guide applies to versions of GCC for AVR32 >=4.2.1.

1 Introduction

The goal of this guide is to quickly enlighten the developer on best practices for the development of code for AVR®32 microcontrollers, as well as giving some pointers to specific features of GCC relevant to AVR32. Most of the information in this document is available in the GCC manual (which is usually a part of the GCC installation)

Note that when referring to commands in the AVR32 GNU toolchain we use the avr32-<command> form in the rest of this guide. If using the toolchain built for an AVR32 Linux® target, these commands should be replaced with avr32-linux-<command>.

Rev. 32074A-AVR-12/07

1.1 Command line options

This document refers to command line options of GCC and will explain the relevant command line options for controlling code generation as well as explaining some of the best practices when compiling source code for an AVR32 microcontroller

1.2 Generating optimised code

GCC has an extensive set of options for code-generateion, but for most users only a few options are really necessary.

When it comes to optimization there is a general switch '-O<optimization_level>' which specifies the level of optimisation used when generating the code:

Table 1-1. Generic optimization flags to gcc

Option	**Function**
-Os	Signal that the generated code should be optimised for code size. The compiler will not care about the execution performance of the generated code.
-O0	No optimisation. This is the default. GCC will generate code that is easy to debug but slower and larger than with the incremental optimization levels outlined below.
-O1 or -O	This will optimise the code for both speed and size. Most statements will be executed in the same order as in the C/C++ code and most variables can be found in the generated code. This makes the code quite suitable for debugging.
-O2	Turn on most optimizations in GCC except for some optimisations that might drastically increase code size. This also enables instruction scheduling, which allows instructions to be shuffled around to minimize CPU stall cycles because of data hazards and dependencies, for CPU architectures that might benefit from this. Overall this option makes the code quite small and fast, but hard to debug.
-O3	Turn on some extra performance optimisations that might drastically increase code size but increase performance compared to the -O2 and -O1 optimization levels. This includes performing function inlining

Table 2 Additional optimization flags for GCC

Option	**Function**
-funroll-loops	If code size is not a concern then some extra performance might be obtained by making gcc unroll loops by using the '-funroll-loops'' switch in addition to the '-O3' switch.
-ffunction-sections -fdata-sections ;	When code size is a great concern it is often helpful to be able to remove unused code and data from the final linked program. The GNU linker has support for this by performing garbage collecting at the time of linking if invoked with the '--gc-sections' option. This will cause the linker to remove unused sections from the final linked program. In order to get garbage collection granularity on a function or data item level, the switches '-ffunction-sections' and '-fdata-sections' can be given to GCC. This will cause each function or data item to be placed in its own section, which means that if the function or data item is unused it can be removed from the final linked program. This is especially useful when compiling files for big libraries where we do not want to include unused functions or data in the linked executable

Option	Function
-fno-common –fsection-anchors	By default GCC uses common symbols for declaring global un-initialised variables. This has the advantage that if several common symbols with the same name exist, they will be merged together at link time. The disadvantage is that GCC then has no control over where the symbol will be placed relative to other symbols. Letting GCC know the relative placement of global variables declared in the same file as where they are used will cause GCC to take more advantage of section anchors. Section anchors are enabled with the '-fsection-anchors' switch or by specifying any optimisation options other than the default '-O1' for the AVR32. Normally when GCC creates code for accessing global or static variables in memory it first need to get the address of this variable into a register. This can be done with the 'lda.w' pseudo instruction, which we will come back to when discussing linker relaxing. When we have the address GCC can insert a memory access instruction to access the variable. If we are accessing several global or static variables in the same function and GCC knows where these variables will be placed relative to each other, then it would be more optimal to get the address of one of the variables and then access the other variables by just adding an offset to this address. This is what section anchoring does, and the address from where all the variables can be accessed by using an offset is called the section anchor. This means that disabling the use of common symbols by using the '-fno-common' switch will often lead to better code when section anchors are used and the code uses many un-initialised global variables. Note that using the '-fdata-section' switch will clobber GCC's section anchor optimisations since each variable will get its own section and GCC will no longer know the relative position of the variables.
-ffast-math -mfast-float	When using floating-point computations in your code it is often not necessary to be fully IEEE® 754 compliant with regards to rounding and handling of special numbers like +/- 0.0, Inf and NaN. This special handling often creates a significant overhead on the floating-point operations generated by GCC. The '-ffast-math' switch makes GCC able to optimise floating-point code to some degree at the cost of not being fully IEEE compliant, but good enough for most applications. The switch '–mfast-float', which is an AVR32 specific switch, causes fast, non-ieee compliant versions of some of the optimised AVR32 floating-point library functions to be used. This switch is by default enabled if the '-ffast-math' switch is used.
-masm-addr-pseudos	This option is enabled by default and causes GCC to output the pseudo instructions call and lda.w for calling direct functions and loading symbol addresses respectively. It can be turned off by specifying the switch '-mno-asm-addr-pseudos' The advantage of using these pseudo-instructions is that the linker can optimise these instructions at link time if linker relaxing is enabled. The '-mrelax' option can be passed to GCC to signal to the assembler that it should generate a relaxable object file.
-mimm-in-const-pool	When GCC needs to move immediate values not suitable for a single move instruction into a register, it has two possible choices; it can put the constant into the code somewhere near the current instruction (the constant pool) and then use a single load instruction to load the value or it can use two immediate instruction for loading the value directly without using a memory load. If a load from the code memory is faster than executing two simple one-cycle immediate instructions, then putting these immediate values into the constant pool will be most optimal for speed. This is often true for MCU architectures implementing an instruction cache, whereas architectures with code executing from internal flash will probably need several cycles for loading values from code memory. By default GCC will use the constant pool for AVR32 products with an instruction cache and two immediate instructions for flashbased MCUs. This can be overridden by using the option '-mimm-in-const-pool' or its negated option '-mno-imm-in-const-pool'.

Option	Function
-muse-rodata-section	By default GCC will output read-only data into the code (.text) section. If the code memory is slow it might be more optimal for performance to put read-only data into another faster memory, if available. This can be done by specifying the switch '-muse-rodata-section' which makes GCC put read-only data into the .rodata section. Then the linker file can specify where the content of the .rodata section should be placed. For systems running code from flash this might however mean that the read-only data must be placed in flash and then copied over to another memory at startup, which means that extra memory usage is required with this scheme.

2 Linker Relaxing

When GCC converts source code to assembly instructions, information about symbol location is missing. This is not know until after linking, which means that GCC must be pessimistic and always assume a worst-case when selecting instructions depending on the address of a symbol.

An example is loading the address of a symbol into a register. GCC does not know what address the symbol will get after the link-stage, so a single move instruction can not be used. A move instruction *(mov)* can only accept immediate integer values ranging from -1048576 to 1048575. Loading the symbol from the constant pool[1] or use two instructions must be done. This is not as effective / fast as a single move instruction.

GCC can not do this optimisation, so Atmel made it possible for the linker do it. This is done by making GCC output the pseudo instructions *lda.w*, to load a symbol address, and *call*, for making a call to the address of a symbol. The linker can then, if the input file is tagged as relaxable, convert a pseudo instruction into the best possible instruction with regards to the final symbol address.

This is called linker relaxing. It makes it possible to convert a *call* into an *rcall* instruction if the distance to the called function is close enough to the calling instruction, or it can convert an *lda.w* into an immediate subtraction relative to the address of the instruction, if the address of the symbol is close enough to allow this, or into an immediate move instruction if theabsolute symbol-address allows this. To enable the linker to convert an *lda.w* into an immediate move instruction, the option *'—direct-data'* must be given to the linker.

Linker relaxing is enabled in the linker by passing the *'—relax'* option to the linker. If using GCC as a frontend for the linker, this option is automatically passed to the linker when using *'-O2'* or *'-O3'* or explicitly using the *'-mrelax'* option. Marking the output objects from GCC as relaxable is done by giving the assembler the *'--linkrelax'* option. This option is automatically passed on to the assembler from GCC when using *'-O2'* or *'-O3'* or explicitly using the *'-mrelax'* option.

[1] When compiling Position Independent Code (PIC) by using the *'–fpic'* switch or using the avr32-linux toolchain this is not entirely true since symbols addresses are accessed vie the Global Offset Table (GOT)

3 Debugging

GCC for AVR32 uses the Dwarf 2 standard for attaching debugging information to the generated code. Using the -g switch when invoking GCC enables this debug info. For ease of debugging it is recommended to not turn on too much code optimisation in GCC when compiling for debugging.

It is, however, possible to debug even if full optimisation is turned on. Problems such as variables that the compiler optimised away can then be unavailable, functions that are inlined and statements that are executed completely out of order or even optimised away. When linker relaxing is enabled it can be very hard to debug since the address of instructions can be changed by the linker. The linker can insert, remove or change the size of certain instructions. This can cause the source code to seem completely uncorrelated with the instructions being executed.

When debugging, it can sometimes be useful to see the assembly code generated by GCC, interleaved with the source code, in a listing file. Making GCC pass a special option further to the assembler to signal that a listing output should be generated can do this.

To make GCC pass options further to the assembler is done by using the *'-Wa,<assembler-option>'* option. The option for making the assembler generate a listing containing the high level source-code interleaved with the generated assembly code is done with the option *'-ahl=<listing-file>'*. Thus, for generating the listing file *foo.lst* the following option has to be given to GCC: *'-Wa,-ahl=foo.lst'*. Generating a listing containing source code and assembly instructions does, of course, require that debug info is enabled when invoking GCC by using the *'-g'* option.

The listings generated by the assembler do not contain any information on where the generated instructions are placed in memory. This is not known until after linking. Also, the pseudo instructions *lda.*w and *call* are not real instructions and we do not know which real instructions these will be replaced with until after linking. To make a listing showing the source code interleaved with the assembly code from the finally linked executable is possible by using the *avr32-objdump* utility (available in tehe binutils package). This can be done if the output executable from the linker is an Elf file, which is the default output format, containing debug information. Just run *'avr32-objdump –S <elf-file>'* and you will get the listing of the complete program with the address location of each instruction.

4 AVR32 specific features in GCC

In addition to support standard C/C++ source code, GCC for AVR32 has some extra AVR32 specific features to allow easier access to the optimized AVR32 instruction set and architecture. This applies particularly to the Digital Signal Processing (DSP) and parallel execution instructions (Single Instruction Multiple Data – SIMD). They come in the form of attributes and builtins to the GNU Compiler.

4.1 Attributes

An attribute is a GCC extension that can be applied to functions or variables to specify non-standard handling of their implementation. The syntax for an attribute is the following:

__attribute__((<attribute-name>(<attribute-argument>)))

4.1.1 interrupt

The interrupt function-attribute is used to signal that a function is an interrupt handler. The difference in the generated code from an ordinary function is that a function tagged with the interrupt attribute will return using a *rete* instruction and has a different scheme for pushing needed registers on the stack. The interrupt attribute takes an optional argument specifying the shadow-register mode for the interrupt. Registers that are shadowed is not required to be saved on the stack before being used in the function. The AVR32 architecture has three possible shadowing modes: *full*, *half* or *none*[2]. Specifying no argument to the attribute defaults to *none*. Writing an interrupt handler function for an interrupt with *half* shadow mode can then be done like this:

```
      __attribute__((interrupt("half"))) void bar(void) {
… }
```

Note that the EVBA system register and the interrupt controller must be set up before actually starting to use interrupts..

4.1.2 naked

The naked function-attribute is used for functions where GCC should not output any prologue or epilogue. This can be used when writing a complete function with inline assembly and the programmerneeds to handle the prologue and epilogue. An example of a naked function with just a *nop* and *ret* instruction return the value of the input argument:

```
__attribute__((naked)) void foobar(int a)
      {
            asm volatile (“nop; ret r12”);
      }
```

This example function will output nothing more than a *nop* and *ret* instruction, no prologue and epilogue.

4.2 Built-in functions

Sometimes the user might want to force the insertion of special instructions into the code generated by GCC. It may be for inserting system specific operations not possible to describe in C/C++ or for inserting optimum instructions for operations hard to express or for the compiler to recognize in C/C++. This is what GCC built-in functions are there for.

Calling a built-in function looks like a normal function-call but really expands to one or more inlined instructions for performing the given built-in operation. GCC comes with a number of target independent built-in functions that can be really handy for the programmer.

[2] See the AVR32 architecture manual for a description of the shadowing modes, and the technical reference manuals for the shadowing modes implemented for a given part.

GCC for AVR32 also includes some AVR32 specific built-in functions for inserting special instructions from the AVR32 instruction set. The advantage of built-in functions is that it is easy to use from C/C++ code while still giving the power and low-level control as you get in assembly programming. Most of these built-in functions could also be implemented with inline assembly, but it is generally better to use built-in functions because inline assembly gives GCC no idea aboutwhich instructions are inserted, whereas built-in functions provides this information.

Here is a list of all the AVR32 specific built-ins function that corresponds to a single instruction:

```
int __builtin_sats (int /*Rd*/,int /*sa*/, int /*bn*/)
int __builtin_satu (int /*Rd*/,int /*sa*/, int /*bn*/)
int __builtin_satrnds (int /*Rd*/,int /*sa*/, int /*bn*/)
int __builtin_satrndu (int /*Rd*/,int /*sa*/, int /*bn*/)
short __builtin_mulsathh_h (short, short)
int __builtin_mulsathh_w (short, short)
short __builtin_mulsatrndhh_h (short, short)
int __builtin_mulsatrndwh_w (int, short)
int __builtin_mulsatwh_w (int, short)
int __builtin_macsathh_w (int, short, short)
short __builtin_satadd_h (short, short)
short __builtin_satsub_h (short, short)
int __builtin_satadd_w (int, int)
int __builtin_satsub_w (int, int)
long long __builtin_mulwh_d(int, short)
long long __builtin_mulnwh_d(int, short)
long long __builtin_macwh_d(long long, int, short)
long long __builtin_machh_d(long long, short, short)

void __builtin_musfr(int);
int __builtin_mustr(void);
int __builtin_mfsr(int /*Status Register Address*/)
void __builtin_mtsr(int /*Status Register Address*/, int
/*Value*/)
int __builtin_mfdr(int /*Debug Register Address*/)
void __builtin_mtdr(int /*Debug Register Address*/, int
/*Value*/)
void __builtin_cache(void * /*Address*/, int /*Cache
Operation*/)
void __builtin_sync(int /*Sync Operation*/)
void __builtin_tlbr(void)
void __builtin_tlbs(void)
void __builtin_tlbw(void)
void __builtin_breakpoint(void)
int __builtin_xchg(void * /*Address*/, int /*Value*/ )
void __builtin_cop(int/*cpnr*/, int/*crd*/, int/*crx*/,
int/*cry*/, int/*op*/)
int __builtin_mvcr_w(int/*cpnr*/, int/*crs*/)
void __builtin_mvrc_w(int/*cpnr*/, int/*crd*/,
int/*value*/)
long long __builtin_mvcr_d(int/*cpnr*/, int/*crs*/)
void __builtin_mvrc_d(int/*cpnr*/, int/*crd*/, long,
long/*value*/)
```

The semantics as well as the input and output operands for all these built-in functions is explained in the AVR32 architecture manual, by looking at the semantics of the functionheader after removing *__builtin*

4.3 Byteswapping

GCC provides two built-in functions for byte- swapping. This does not map to a single instruction but selects which instruction to insert depending on the input or output operands being stored in registers or memory. The *bswap_16* built-in swaps bytes in a 16-bit halfword while *bswap_32* swaps bytes in a 32-bit word:

```
short __builtin_bswap_16(short)
int __builtin_bswap_32(int)
```

In addition GCC has some generic built-in functions; *__builtin_ffs*, *__builtin_clz* and *__builtin_ctz* that utilizes instructions such as *clz* and *brev* for AVR32 targets.

4.4 Atomic instructions

AVR32 GCC also has support for the atomic built-ins provided by GCC. These can be used to implement atomic operations on memory, which is useful in operating systems for handling synchronization between processes. These builtin functions make use of the *stcond* and *xchg* instructions for the AVR32 architecture. These instructions have been designed with these types of operations in mind. Some of the available atomic operations are:

```
int  __sync_fetch_and_<op> (int *ptr, int value)
int  __sync_<op>_and_fetch (int *ptr, int value)
bool __sync_bool_compare_and_swap ( int *ptr, int oldval,
                                    int newval)
int __sync_val_compare_and_swap (   int *ptr, int oldval,
                                    int newval)
int __sync_lock_test_and_set (int *ptr, int value)
```

Where <*op*> denotes the operations: *add*, *sub*, *or*, *and*, *xor* and *nand*.

Documentation of the builtin functions can be found under various chapters in the GCC manual. Examples are found in:*"Extensions to the C Language Family"* ,"*Built-in functions for atomic memory access*", "*Other built-in functions provided by GCC*" and "*Built-in Functions Specific to Particular Target Machines*".

4.5 Inline Assembly

GCC supports using inline assembly, which means that you can freely mix C/C++ code with assembly code in your source code. You can pass data back and forth between variables in C/C++ and low-level instructions in assembly.

The inline assembly syntax is:

```
asm ("<assembly-code>"  : "<constraint>" (<output-variable>),
                        "<constraint>" (<output-variable2>),
                                …
                        "<constraint>" (<output-variablen>)
                        : "<constraint>" (<input-variable>),
                        "<constraint>" (<input-variable2>),
                                …
                        "<constraint>" (<input-variablen>)
                        : "<clobber1>" … "<clobbern>");
```

Where a constraint is a code for what type of operand is allowed for the given operand in the inline assembly expression. A detailed description of constraints is given in the chapter "*Constraints for asm Operands*" in the GCC manual. Operands are referred to with *%<operand-number>* where the *operand-number* is the number of the operand in the operand list.

Here is an example of using inline assembly to use the *macwh.d* instruction:

```
static inline long long macwh_d (long long acc,
                                       int a, short b)
      {
            asm ("mulwh.d %0, %1, %2" : "+r"(acc)
                                      : "r" (a),
                                        "r" (b));
            return acc;
      }
```

Here we have used the most common constraint, *'r'*, which means that the operand needs to be in a register. Normally output operands need an *'='* character before the constraint character to specify that the operand is an output, but in this case the *acc* output operand is both an input and output so the *'+'* character should be used instead.

For a more detailed description of inline assembly see the chapter "*Assembler Instructions with C Expression Operands*" in the GCC manual.

5 From source-code to executable

The process of going from source code to a final executable program is quite simple. First all source code files have to be compiled, in order to be converted to assembly code. The assembler then converts the generated assembly code to object files. All object files are then linked together by the linker to produce the final executable program. Here we will learn that flexibility of the *avr32-gcc* command that can be used as a wrapper for all these tasks.

5.1 Compiling

Using the *avr32-gcc* command invokes the GCC for AVR32 compiler. The path to the source file(s) is given as input to the command. The *avr32-gcc* command is really a front-end for some underlying tools.

First the *avr32*-gcc command calls the pre-processor that parses all pre-processor directives. Then, unless given the *'-E'* switch, which tells *avr32-gcc* to only pre-process, it calls the actual compiler that converts C/C++ code to assembly instructions.

Next, the assembler is called, unless the *'-S'* switch is given, which tells *avr32-gcc* to only generate assembly output, and generates an object file. If not using the *'-c'* switch, *avr32-gcc* will even execute the linker

Invoking avr32-*gcc* can actually do the whole process of compiling, assembling and linking.

Here is an example of compiling one of the two files in the commonly used dhrystone benchmark. We are compiling *dhry_1.c* in order to make the output object file *dhry_1.o*:

avr32-gcc –c –g –O3 -fno-common -funroll-loops –Wa,-ahl=dhry_1.lst dhry_1.c –o dhry_1.o

Note the use of optimisation options and that we are signalling that debugging info should be included in the generated object file. With the *'-c'* option we are telling the *avr32-gcc* front-end not to call the linker and just to invoke the assembler in order to produce an object file output. We have also added an option for making the assembler output a listing with interleaved source and assembly code in the file *dhry_1.lst*.

When compiling C++ sources the command *avr32-g++* can also be used. This command will by default treat *.c* and *.h* files as C++ sources.

5.2 Assembling

As you have seen, the assembler does not need to be invoked directly when compiling source code with *avr32-gcc*. The *avr32-gcc* command can take care of this. If assembling hand-written assembly code, the assembler can be invoked directly by using the *avr32-as* command. The *avr32-gcc* command can however also be used for this task.

All files with ending *'.S'* and *'.s'* will be treated as assembly files and compilation will be skipped and the assembler will be called directly. For *'.S'* files the C pre-processor will also be invoked, which means that it is possible to use pre-processing directives in these types of files.

5.3 Linking

Linking canbe invoked with the *avr32-gcc* command. If invoked without the *'-c'* option, *avr32-gcc* will try to link together all the files specified as input. If not all input files are object files it will first try to compile and/or assemble any C/C++ or assembly files based on the filetype guessed from the filename before calling the linker.

5.3.1 Default linker options

When avr32-*gcc* is used as a front-end for linking it will by default include the GCC runtime library (libgcc), C-library (libc), a startup file (typically crt0.o) and some other files required at runtime by GCC. The GCC runtime library is required by GCC for some operations that need library calls, such as floating-point operations etc. The C-library is required if using any of the C-library functions in your program. The startup file is responsible for doing any initialisation needed, such as clearing un-initialised variables, setting the stack-pointer etc., before calling the main function of the program.

The linker can be called directly by using the *avr32-ld* command, but then the user is responsible to provide all the needed libraries and object files needed for the linking to succeed. This can often be practical for small assembly-code where no runtime library or C-library is needed.

5.3.2 Linker scripts

The linking process needs information about code and data memory location. Using a linker script provides this. If specifying which device you are compiling code for by using the *'-mpart=<part>'* option in *avr32-gcc,* a default linker script for that device will be used.

These linker scripts can be found under the *avr32/lib/ldscripts* folder of your installation. If you have special needs and the default linker script is not working for your project, you can specify your own linker script to the linker. This can be done by using the *'-T<linker-script>'* option to *avr32-ld*.

If using *avr32-gcc* for linking the option for passing parameters to the linker *'-Wl,*<linker-*option>'* can be used like this: *'-Wl,-T<linker-script>'* in order to pass the linker script option forward to the linker. We will not go into the details of how to write linker files scripts here. The linker manual contains a detailed description of the powerful linker script command language. It might also be worth taking a look at the default linker scripts used by the linker, and maybe using this as a basis if writing your own scripts.

5.3.3 Linking C++ object code

When linking (and compiling) C++ programs, the *avr32-g++* command is invoked. This command will automatically add the C++ library (libstdc++), which is most likely required by your C++ code, in addition to the other libraries *avr32*-gcc adds when linking.

6 Example use of GCC

Here is an example of linking the two files of the Dhrystone benchmark into a final executable by using the *avr32-gcc* command:

avr32-gcc *–O3 –Wl,--direct-data dhry_1.o dhry_2.o –o dhry.elf*

Specifying the *'-O3'* to *avr32-gcc* causes the *´--relax'* option to be passed on to the linker. This could also have been done by omitting *'-O3'* and using *'-mrelax'* instead.

Optimization the Dhrystone application even more is done by invoking more optimization options:

avr32-gcc –c –g –O3 -fno-common -funroll-loops –Wl,--direct-data dhry_1.c dhry_2.c –o dhry.elf

Refer to the above chapter for an explanation of the options.

7 Optimizations and results on the Dhrystone benchmark

We have used the Dhrystone benchmark as an example on the effects of using the different optimisation options. We have compiled the benchmark for the AVR32 UC3 architecture with different optimisation options[3]:

Table 7-1. Results of optimization flags on the Dhrystone benchmark

GCC options	Linker options	Dhrystone MIPS / MHz	Code size dhry_1.c	Code size dhry_2.c
-Os –fno-common		0,93	2892	258
-O0		0,47	3612	926
-O1 –fno-common		0,9	3136	310
-O2 –fno-common		1,08	3128	314
-03 –fno-common		1,26	3288	290
-03 –fno-common –funroll-loops		1,27	3288	286
-03 –fno-common –funroll-loops	--relax	1,35	3288	286
-03 –fno-common –funroll-loops	--relax –direct-data	1,38	3288	286

Notice the large code size and bad performance when using no optimisation ('-O0'). Using no optimisation should really just be used for debugging as can be clearly seen here. We can also see that when optimising for code size, the performance is in between what we get with '-O1' and '-O2'. Also take notice the extra performance gain we get when enabling linker relaxing and direct-data in the linker.

Using the '-ffunction-sections' GCC option and the linker option '--gc-sections' when compiling and linking dhrystone gave a size reduction on about 8-9% for the final executable when linking with the newlib c-library and GCC runtime library. The newlib library has already been compiled with the '-ffunction-sections' option, so using the garbage collector in the linker to remove unused sections was able to remove some unused functions.

These numbers were obtained by running with code in embedded flash using 0 Wait-States and data in embedded SRAM. The flash-controller master is set to last default configuration in the HMATRIX.

The code size here is before linking, taken from the object files. When turning on linker relaxing, the code size may actually become smaller in the final executable.

[3] We used the 1.3.2-0 version of the free AVR32 GNU toolchain from Atmel when compiling this benchmark.

Headquarters

Atmel Corporation
2325 Orchard Parkway
San Jose, CA 95131
USA
Tel: 1(408) 441-0311
Fax: 1(408) 487-2600

International

Atmel Asia
Room 1219
Chinachem Golden Plaza
77 Mody Road Tsimshatsui
East Kowloon
Hong Kong
Tel: (852) 2721-9778
Fax: (852) 2722-1369

Atmel Europe
Le Krebs
8, Rue Jean-Pierre Timbaud
BP 309
78054 Saint-Quentin-en-Yvelines Cedex
France
Tel: (33) 1-30-60-70-00
Fax: (33) 1-30-60-71-11

Atmel Japan
9F, Tonetsu Shinkawa Bldg.
1-24-8 Shinkawa
Chuo-ku, Tokyo 104-0033
Japan
Tel: (81) 3-3523-3551
Fax: (81) 3-3523-7581

Product Contact

Web Site
www.atmel.com

Technical Support
avr@atmel.com

Sales Contact
www.atmel.com/contacts

Literature Request
www.atmel.com/literature