

KEIL μ VISION3 (GCC COMPILER) FAQ

C BASICS

Q: How do I beginning programming the ADuC7000 Series?

A: For a quick and easy introduction to programming for the ADuC7000 series, please refer to the “Get Started Guide. For sample projects, please refer to the code examples directory installed from the QuickStart CD. If you do not have the CD, or the code examples are note present, they may be downloaded from <http://www.analog.com/microconverter>, Technical Support Resources in the Code Examples Section.

Q: How do I change (CAST) a variable’s size?

A: To convert a variable from one size to another, i.e. casting, the following syntax is used:

Destintaion_Variable = (Destination_Variable type) Source_Variable

An example of this is:

```
Unsigned char U_char;      /*      8 Bit      */
Unsigned long U_long;     /*     32 Bit     */
U_char = (unsigned char) U_long;
```

WARNING: Casting a variable may cause data loss.

Q: How do I know where my variable/constants are placed and what size they are?

A: The table below gives a guide to the size and valid data range of various variable types.

	Size	Format	Valid Data
Char	8 Bit	Signed	-128 to 127
		Unsigned	0 to 255
Short	16 Bit	Signed	-32,768 to 32767
		Unsigned	0 to 65,535
Long/Int	32 Bit	Signed	-2,147,483,648 to 2,147,483,647
		Unsigned	0 to 4,294,967,295

To ensure that a variable is always accessed in a loop, the modifier **VOLATILE** is used, signifying that the data in the variable may change unexpectedly. This ensures that the variable is not optimized out of the loop. An example of the use of the volatile modifier:

```
Volatile int i;
```

To place global variables in FLASH, use the **CONST** modifier. To place local variables in FLASH use the **STATIC CONST** modifier.

For more information on the location of variables, i.e. where in memory the variable is located, please refer to the code example located in:

```
\\ADuC_Beta702x\Code\Keil Code Examples\ADuC7024\Variable Placement
```

CODE IN RAM

Q: I have heard that it is possible to place certain **functions in RAM**, how is this done and why would I want to do it?

A: Yes, this is possible and in certain circumstances extremely desirable. To place functions in ram, using GCC, the section function attribute is used alongside the function prototype, e.g.

```
Void my_ram_function(void) __attribute__((section (".ram_func")));
```

Running functions from RAM is desirable for the following reasons:

1. Whilst running at CD = 0 to avoid the memory access penalty if running ARM code. The Flash is divided into half word, 16bits. When running the CORE in ARM mode at CD = 0, an extra memory fetch is required to construct the instruction.
2. If flash programming is required. When erasing/writing to the flash, the core is stalled until the operation is completed. To overcome this, the flash programming function may be placed in RAM and whilst programming, the core continues to execute code. Before leaving the code segment in RAM, ensure that the flash is ready to resume code operation.

An example of functions in RAM is included in

```
\\ADuC_Beta702x\Code\Keil Code Examples\ADuC7024\ FuncRam
```

NOTE : A special linker script has being used which is present in the above directory.

Q: How do I place my interrupt **Vector table in RAM** and perform a **REMAP**?

A: It is desirable to have the interrupt vector table in RAM for several reasons:

1. To ensure that an interrupt is executed as fast as possible when programming the flash.
2. It allows the vector table, which is written in ARM assembly, to be executed at the maximum possible speed if running at CD = 0

To place the Vector table in RAM requires three steps:

1. The memory regions in the linker must be changed by the user. The Data region is changed from Start 0x10000 and size 0x2000 to Start 0x10040 and size 0x1FC0. This tells the compiler to not use the first 64 bytes of RAM.
2. Construct the vector table in RAM. The vector table consists of jumps to functions. This is accomplished by either branch, B, or load, ldr PC, assembly commands. These commands must be placed in RAM locations starting at 0x10000.
3. Perform a REMAP, i.e. map RAM (0x10000) to 0x00. By default Flash (0x80000) is mapped to 0x00. This is done by the following line of C Code:

```
REMAP = 0x01;
```

A detailed example is shown in

\ADuC_Beta702x\Code\Keil Code Examples\ADuC7024\Int\RamInt

INTERRUPTS

Q: How do I write an **ISR (Interrupt Service Routine)**?

A: By default, FIQ and IRQ interrupts are enabled in the CPSR Register. An IRQ is a standard interrupt and an FIQ is a fast interrupt which has a higher priority than the IRQ. To enable an individual interrupt, the required bit in the FIQEN/IRQEN MMR (Memory Mapped Register) must be set.

To specify the ISR function the IRQ and FIQ variables must be set equal to the required Interrupt function.

An example of this is shown below:

To specify the IRQ and FIQ functions, equate the relevant function to either IRQ or FIQ. In the example the IRQ function is My_IRQ_Function.

```
IRQ = My_IRQ_Function;           //Specify Interrupt Service
FIQ = My_FIQ_Function;          //Specify Interrupt Service
```

The functions My_IRQ_Functions and My_FIQ_Function are user defined functions.

To enable the external interrupt 0 and the general purpose timer and ADC to generate a FIQ the following code is used.

```
IRQEN = XIRQ0_BIT+GP_TIMER_BIT;//Enable XIRQ0 and Timer1
                                     Where GP_TIMER_BIT    0x00000008
                                     XIRQ0_BIT              0x00008000
FIQEN = ADC_BIT;                    //Enable ADC FIQ
                                     Where ADC_BIT          0x00000080
```

The interrupt functions, My_IRQ_Functions and My_FIQ_Function, may be defined as below

```
// Function Prototypes
void My_IRQ_Functions(void);           void My_FIQ_Functions(void);

void My_IRQ_Functions(){               void My_FIQ_Functions(){
    return; // Write IRQ Code Here     return; // Write FIQ Code Here
}
```

A detailed example may be seen in:

\ADuC_Beta702x\Code\Keil Code Examples\ADuC7024\INT\Mult

STDIO LIBRARY

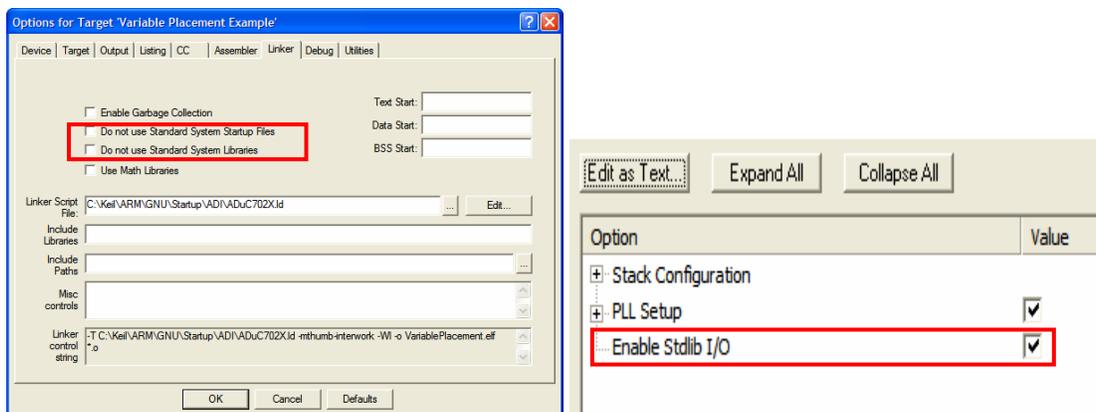
Q: How do I use **PRINTF** and **SCANF** statements, i.e. How do I use **STDIO**?

A: To use **STDIO** functions, please ensure that the following project options are selected:
Project->Options for Target.->Linker Tab

Ensure that “Do not use Standard System Startup files” and “Do not use Standard System Libraries” are deselected.

Startup.s: Startup Code File

When opened in the configuration wizard, if opened in text mode right click inside file and select “Open in Configuration wizard”, ensure that “Enable **STDLIB I/O**” selected.



The user is also required to include two C source files, serial.c and syscalls.c, for Serial IO via the UART. These files are located in:

\Keil\ARM\INC\ADI

Using **STDIO** impacts considerably on the size of code. The inclusion of the Standard system startup files adds approximately 5.5K to the user code, If **PRINTF** is used, a further 20K is required. If **SCANF** is used in conjunction with **PRINTF**, then a further 5K is required. When **SCANF** is used in isolation from **PRINTF**, the additional code size required is approximately 13K.

- Q: Is it possible to redirect the output/input of the STDIO library from the UART to, for example, the SPI port?
- A: To redirect the output/input of the STDIO library requires modification of the putchar and getchar functions located in the serial.c file. This file is included must be included in all projects which make use of the STDIO library. For example to use SCANF targeting the SPI port, getchar would be changed by the user from:

```
int getchar (void) {          /* Read character from Serial Port */

    while(!(0x01==(COMSTA0 & 0x01)))
    {}
    return (COMRX);
}
```

to:

```
int getchar (void) {          /* Read character from Serial Port */

    while(!(0x01==(SPISTA & 0x03)))
    {}
    return (SPIRX);
}
```