<< Install toolchain Open "Hello World!" Build for RAM Build for flash ROM see also

Quick start: "Hello world!" example

Step 1: Install the GNU toolchain

Install the STW TriCore Software Tools by following the wizards instructions.
This will install among other things the GNU compiler for TriCore and the Code::Blocks IDE.

Step 2: Open the ESX-3XL "Hello world!" project

Start Code::Blocks and open the ESX-3XL demo project "esx3xl_demo.cbp".

This project consists of only one source file whose main program look like this:

```
int main(void)
{
   T_x_sys_system_information tSys;
   sint32 s32_Error;


   s32_Error = x_sys_init(&tSys);          // init ESX-3XL hardware, this function call is mandatory

   if (s32_Error == C_NO_ERR)              // system init OK?
   {
      // init serial interface X_SER_01 (19200, 8N1)
      s32_Error  = x_ser_init(X_SER_01, 19200, X_SER_MODE_8DB_NO_PARITY, 1u, 0u, NULL);

      // init X_SER_01 TX buffer
      s32_Error |= x_ser_init_tx_buf(X_SER_01, 100u, 0u, NULL);
   }


   if (s32_Error == C_NO_ERR)
   {
      x_ser_write_string(X_SER_01, "Hello world!");
      x_sys_set_beeper(1000);     // set beeper 1000Hz
      x_sys_delay_us(200000);     // do a short beep (0.2 seconds)
   }
   else
   {
      x_sys_set_beeper(200);      // on error: set beeper 200Hz
      x_sys_delay_us(1000000);    // do a long beep (1 second)
   }

   x_sys_set_beeper(X_OFF);       // switch beeper off


   while (true)                   // main loop...
   {
      led1_change_color();        // setup LED1 color
      x_sys_delay_us(10000);      // delay 10 milliseconds
   }
}
```

The program starts with a call of x_sys_init, the hardware initialization of the ESX-3XL main board. Each ESX-3XL application should start with this function call. Further hardware initialization may follow when a ESX-3XL controller is equipped with expansion boards.
This example program use the serial interface as output and therefore the interface must be initialized and a TX buffer must be prepared.
If anything is OK it writes the "Hello world!" string to the interface and beeps a short time. If something wents wrong it does a long beep with a low frequency.
Afterwards it changes the color of LED1 in the main infinite loop.

Step 3: Build the project for RAM debugging

To link the application into RAM space we have to use the '**Debug**' build target.

Check the setting for the linker command inside "**Project→Build options→Linker tab**" from Code::Blocks main menu:
Be sure that the 'Debug' build target project settings are active. It must be selected in the left of the build options dialog.
The linker option **-Wl,--script=..\libs\esx3xl_link_debug.ld** should be set in order that the executable output code will be linked into RAM for debugging. This linker command file sets the program start address to **0xA4000000** that is external RAM space.

Select the build target '**Debug**' at the botton bar and then build the program with "**Build→Rebuild**" from the Code::Blocks main menu. If all goes well then the **ELF file** "esx3xl_demo_debug.elf" will be generated inside of the 'output' folder.

Debugging the executable in RAM:

Be sure the debugger hardware is connected and the power supply for the debugger as well as for the ESX-3XL is switched on.

Make sure that the **Debug** build target is active in Code::Blocks.
Start the TRACE32 debugger from the Code::Blocks 'Tools' menu (see also: chapter "**Starting up with TRACE32 debugger**").
The TRACE32 program will now be started, the debugger hardware will be initialized, the executable output file will be loaded into the RAM and the program execution will be started until the main function.
Now you can step through your program, set breakpoints, watch variables and registers and so on...
You can reload the executable output file by typing "**run ram**" in the TRACE32 command line.

Step 4: Build the project to be stored permanently into flash ROM

To link the application into flash ROM space we have to use the '**Release**' build target.

Check the setting for the linker command inside "**Project→Build options→Linker tab**" from Code::Blocks main menu:
Be sure that the 'Release' build target project settings are active. It must be selected in the left of the build options dialog.
The linker option **-Wl,--script=..\libs\esx3xl_link_extflash.ld** should be set in order that the executable output code will be linked into flash ROM for storing permanently. This linker command file sets the program start address to **0xA1040000** that is external flash ROM space.

Select the build target '**Release**' at the botton bar and then build the program with "**Build→Rebuild**" from the Code::Blocks main menu. If all goes well then the **ELF file** "esx3xl_demo.elf" will be generated inside of the 'output' folder.

Additionally the application will be converted to a so-called **HEX file** "esx3xl_demo.hex". With using the HEX file the application could be loaded into the ESX-3XL controller via CAN bus by using the **WinFlash** tool.

Debugging the executable in flash ROM:

Be sure the debugger hardware is still connected and the power supply for the debugger as well as for the ESX-3XL is switched on.
Make sure that the **Release** build target is active in Code::Blocks.
Start the TRACE32 debugger from the Code::Blocks tools menu (see also chapter "**Starting up with TRACE32 debugger**").
The TRACE32 Program will now be started and the debugger script 'flash.cmm' automatically initializes the debugger hardware, enables the debugger for programming flash memories, unlocks and erase flash sectors from address 0xA1040000 up to 0xA107FFFF, loads the executable output file into this flash area and starts program execution until the main function.
Now you can step through your program, set breakpoints, watch variables and registers and so on. But debugging inside the flash memory limits the usage of breakpoints. Only two hardware breakpoints are available.
You can reload the executable output file by typing "**run flash**" in the TRACE32 command line.

see also: