

# Getting Started

Power Architecture Development Platform

Version 1.0



HighTec EDV-Systeme GmbH  
Feldmannstraße 98  
D-66119 Saarbrücken

Phone: +49 681-9 26 13-16  
Email: [info@hightec-rt.com](mailto:info@hightec-rt.com)  
<http://www.hightec-rt.com>

# 1 Getting Started with the Eclipse PowerPC Development Platform UAD2+

## 1.1 Software installation

This procedure will install the PowerPC Development Platform UAD2+ on your PC. For performing a full installation with all components, you need administrator rights on your PC. If you do not have these rights, contact your system administrator.

1. Insert the PowerPC Development Platform UAD2+ disk into your CD drive. Setup should start automatically. If AutoRun is deactivated, double-click the **Setup.exe** file on the CD.
2. For the installation of the PowerPC Development Platform UAD2+ a key for unlocking is required. Please request the key from HighTec by sending an email to <mailto:eval@hightec-rt.com> and providing the following information:
  - a) First and last name
  - b) Company
  - c) Postal address
  - d) Phone
  - e) Type of controller (e.g. PowerPC)

The information will be treated confidentially by HighTec. Depending of the entered key the different products like PowerPC Compiler, IDE Eclipse, PLS debugger and real-time operating system PXROS-HR, which includes a management of the PowerPC MPU, will be installed.

**Note:**

Per default the installation is done in the directory `C:\Hightec\PowerPC` . In the following text this path is called installation directory.

## 2 Getting started with Eclipse

Eclipse is a powerful graphic interface for the PowerPC compiler toolchain, which contains a complete project management and an interface for configuring the PowerPC compiler. To become familiar with the basic features of Eclipse, it is recommended to read the on-line manuals integrated in the Eclipse Help. To run Eclipse an installation of Java JRE 6 Update 10 or higher is required.

### 2.1 Starting Eclipse

Start the desktop icon 'Eclipse for PowerPC' . Eclipse uses a so-called workspace to manage different C/C++ projects. The selection of the workspace is pre-configured but can be set to a user defined directory.

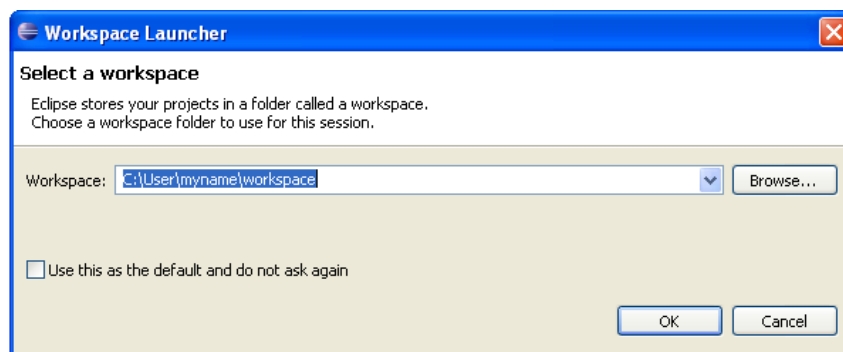


Figure 2.1: Eclipse Workspace Selection

If you launch Eclipse for the first time, a welcome screen will appear, which can be closed by clicking the cross 'x' in the upper right-hand corner of the window.

The following parts of the manual explain the workflow for creating and debugging PowerPC projects.

### 2.2 Setting up a Project

This tutorial shows how to create a simple 'timedemo' example for PowerPC in Eclipse.

#### 2.2.1 Wizards for creating a PowerPC Project

Before creating a PowerPC project, it is necessary to switch to the HighTec perspective, which should be active per default. This offers an optimized C/C++ perspective for the PowerPC. If it is not enabled please use the menu 'Window' → 'Open Perspective' → 'Other' → 'HighTec' to activate this perspective. The name of the perspective is displayed in the title bar of the workbench window.

1. The PowerPC C/C++ Project wizard is started via the menu 'File' → 'New' → 'PowerPC Project' .
2. Enter a name for your project, e.g. `myproject`, and press the 'Next' button. By default the project will be stored in the existing workspace of Eclipse.

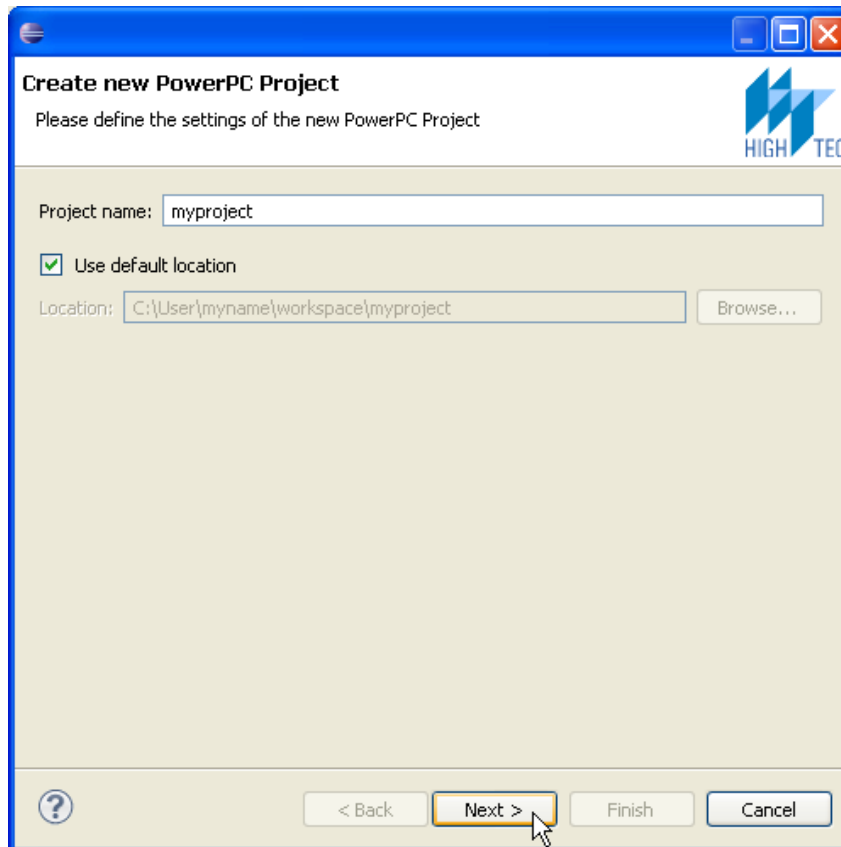


Figure 2.2: Project Wizard

3. The next dialogue shows a selection box containing the most common PowerPC boards (see [Figure 2.3](#) on page 4). If your board is not listed here, please choose one of the configurations which might be similar to your board.

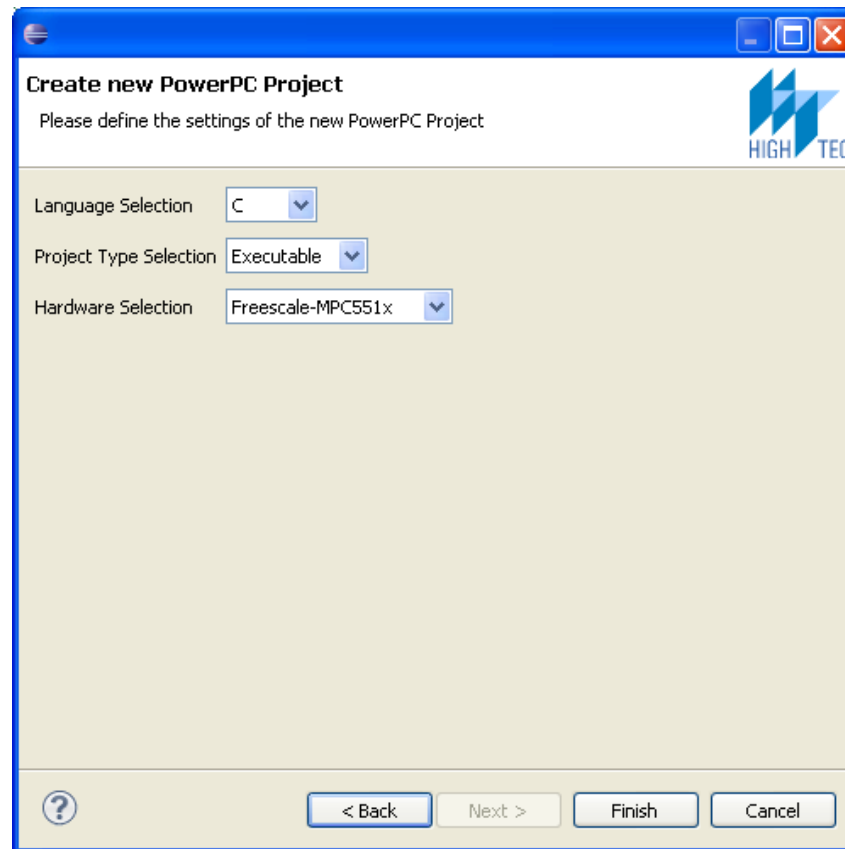


Figure 2.3: Hardware Selection

4. To open your main routine select the source file `main.c` or `hello.c` in the `src` folder of the C/C++ Project view and open the file with a double-click (see [Figure 2.4](#) on page 5) or by using the corresponding context menu.

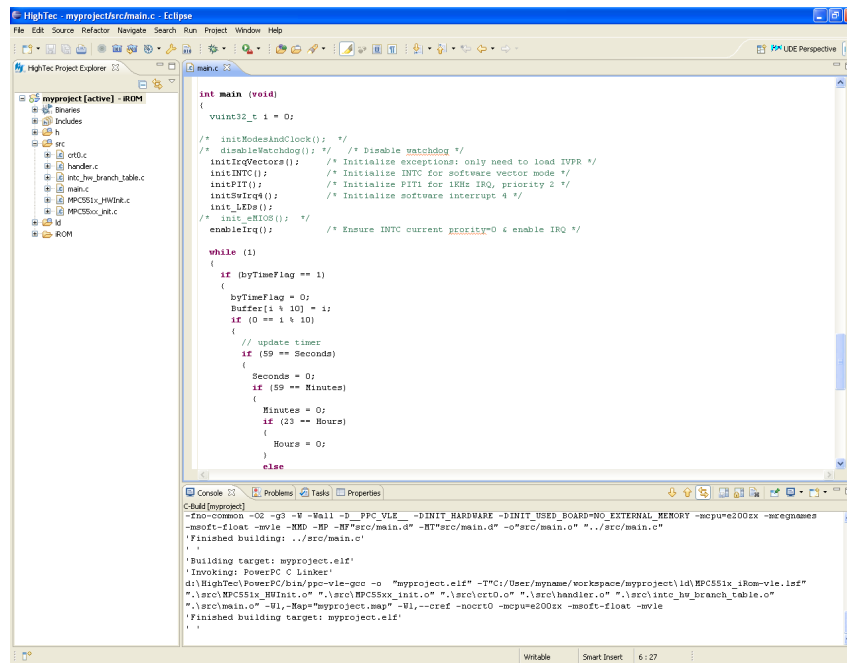




Figure 2.4: Open main routine

- The wizard generates different build targets like iRAM and iROM. You can choose a configuration by clicking  and build the active project via the build icon .

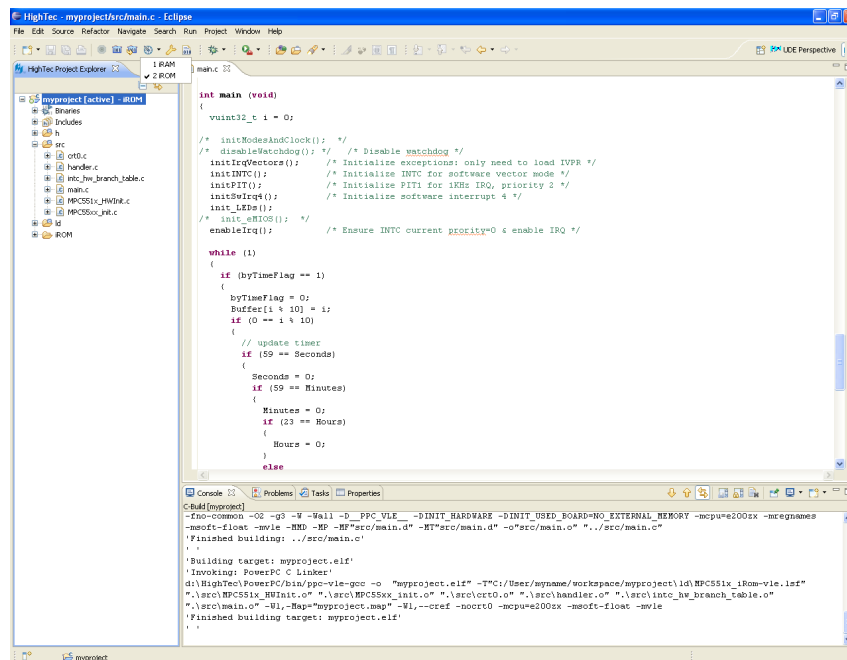


Figure 2.5: Building a project

During the build process the sources belonging to the project will be compiled and linked. The messages occurring during the build process are displayed in the 'Console window'. The build process should terminate without giving any errors or warnings.

### Meaning of build targets

**iROM** Code will be located in the internal flash.

**iRAM** Code will be located in the fast internal scratch pad ram.

**eRAM** Code will be located in an external RAM.

**eROM** Code will be located in an external flash.

For derivatives with an external bus interface you can put your code into the external RAM. The build target is called 'eRAM' .

1. Start a debug session via a new Eclipse launch configuration

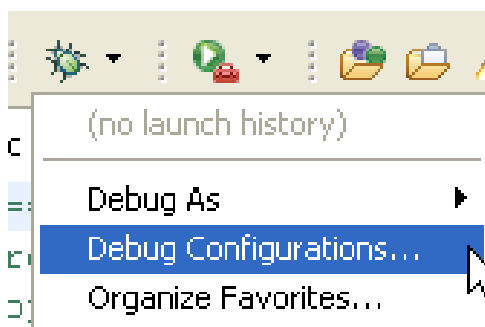



Figure 2.6: Starting a new debug configuration

The following steps are necessary to debug your application:

- a) Connect the freescale/ST main adapter to the power supply and connect the cable to the appropriate socket of the freescale/ST.
  - b) Plug the power supply of the Universal Access Device (UAD) into the power socket and connect with the UAD. For communication between the UAD and your host-PC a USB or firewire connection is required.
  - c) Use the provided JTAG cable to connect the UAD to your PowerPC target board.
2. To create a new launch configuration, double-click the Universal Debug Engine entry or click the icon 

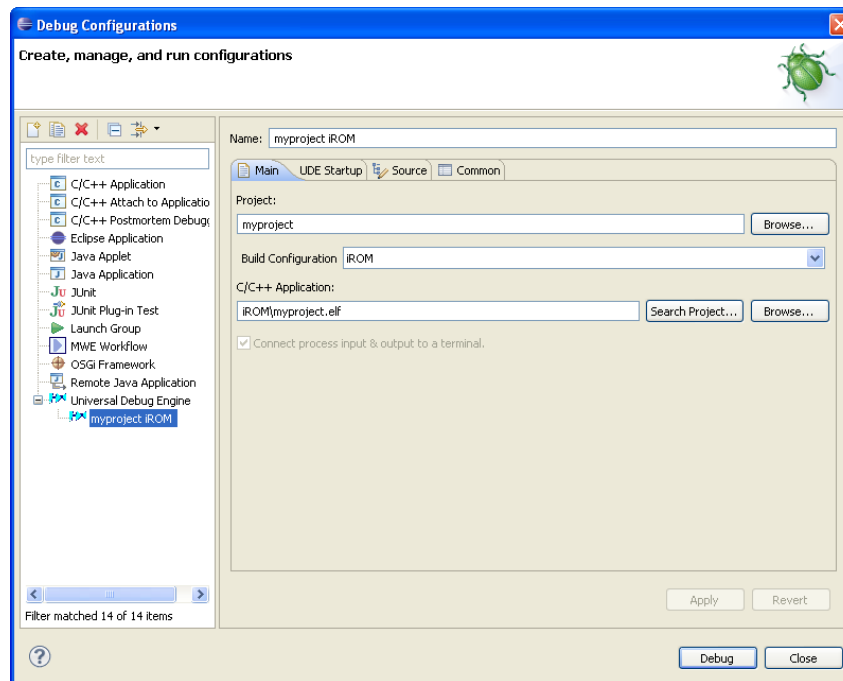


Figure 2.7: UDE Debug configuration

**Note:**

The UDE target configuration files are located in the folder `targets` of the UDE installation directory e.g. `c:\Program Files\pls\UDE3.0\targets`.

3. If a project is selected, the corresponding UDE debug configuration has to be selected. Hitting the 'Debug' button will launch the UDE debugger, and Eclipse will switch to the UDE perspective.

If you built a ROM version of your application, the UDE Memory Programming Tool will appear in the UDE perspective.

**Note:**

If the Memory Programming Tool does not appear, please start it via the menu 'Tools' → 'Flash Programming' and set the 'Enable' checkbox. Start flashing with the Program button and then exit this dialogue.

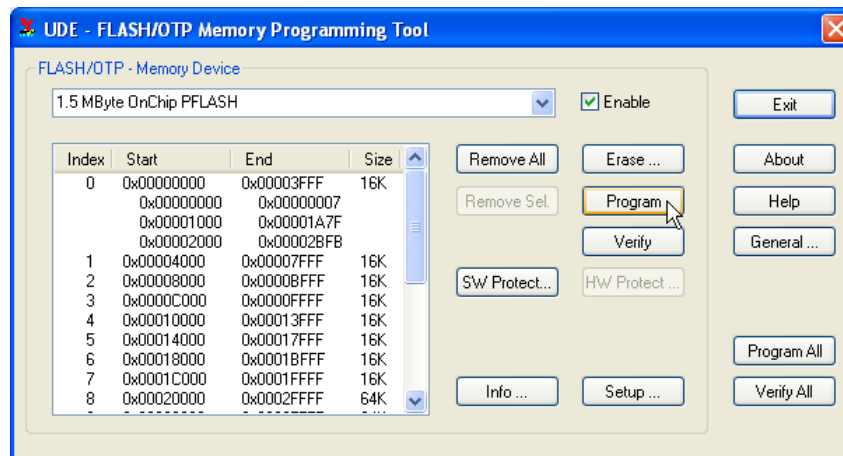



Figure 2.8: UDE Memory Programming Tool

1. Start your application via menu 'Debug' → 'Start Program Execution' or the icon 

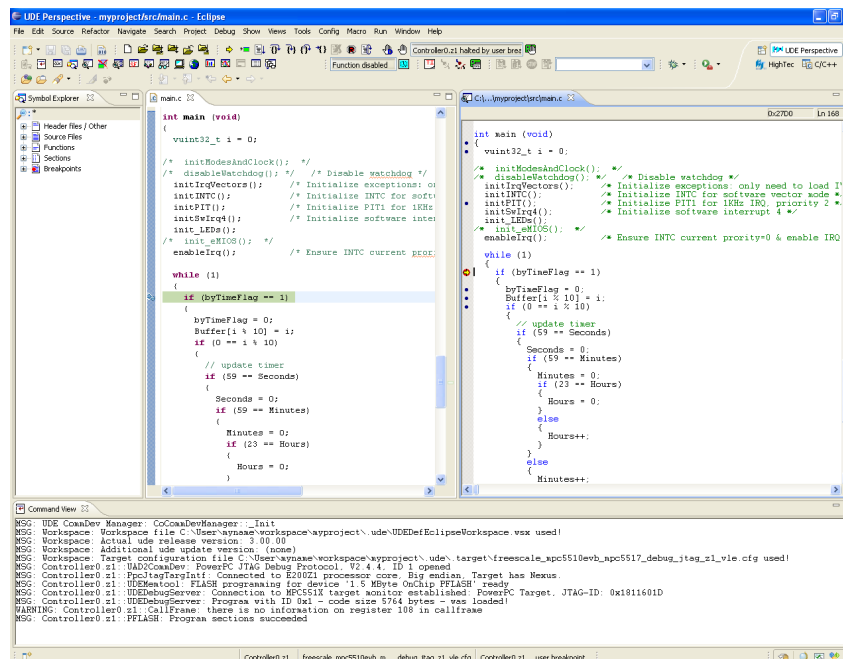


Figure 2.9: timedemo

The following text gives you a brief description of the example timedemo. A hardware interrupt triggers a software interrupt. In the software interrupt a counter SWIrq4Ctr is incremented and its hex value is set as output on 4 Leds. Before leaving the software interrupt the global variable byTimeFlag is set. This variable is used in the main-loop, which implements a clock functionality in hours, minutes and seconds. If the program is halted in the debugger you can add the corresponding variables to a watch window to see e.g. how many seconds the program was active.

### crto.c

This file contains the start-up for the corresponding Power Architecture derivative.

For example the

- stack
- small data pointers
- trap table

are initialized.

#### **init\_hw\_branch\_table.c**

This file contains the configuration of the interrupt vector table. In the timedemo a hardware and software interrupt are configured. The software interrupt `Swlrq4Handler` is entered within the interrupt table as parameter via the macro `INTR_HND` and the hardware interrupt is enabled with `Pit1Handler`.

#### **handler.c**

This file contains the implementation of a hardware interrupt `Pit1Handler` and software interrupt `Swlrq4Handler`. These functions use the attribute `naked` which avoids the generation of a prologue and an epilogue by the compiler.

#### **<board>.h**

This header file contains the description of the special function registers. The different SFRs are implemented as bit fields. The user can access the different bits of a SFR as struct element.

#### **<board>\_HWInit.c**

Contains functions which can configure the MMU, external bus interface etc. The configuration to use VLE mode is included here.

#### **<board>\_<buid-configuration>-vle.lsf**

The file is the so-called linker description file and contains the the memory layout of the Power Architecture device and the location of different input sections and symbols.

### **2.2.2 HighTec Toolbar**

The PowerPC Development Platform UAD2+ provides different extensions to the standard Eclipse CDT which are accessible from the HighTec toolbar. The icons have the following meaning:



A project can be set as active via this button or via the corresponding context menu. A project remains active even a different project from the Project Explorer is selected.



Build active project with current configuration.



Re-build active project with current configuration.



You can switch between different build configurations.



Project settings of active project will appear.

## 2.3 Modifying the file list of a project

### 2.3.1 Removing files from a project

An Eclipse project contains files which can belong to different build targets. For example you can select a file in the C/C++ Project explorer and use the context menu 'Exclude from build' to remove this file in different build targets and at the same time but also keep within in your file system.

### 2.3.2 Adding files to a project

The `timedemo` project can be used as a starting point for your software projects. The procedure to extend the projects with new source and header files is as described below:

1. In the C/C++ Projects view, right-click the name of the project, e.g. `myproject`, and select the menu 'New' → 'Source File' .
2. Specify a source folder and a name for the new file. Use the extension `.c` to handle C source files, the extension `.cpp` for C++ files and `.S` for assembler.

In the Template field you can select a code template for your source file. Choose the 'Default C source' template to insert a default code or 'None' if you want to start with an empty file. To enable a standard design for source files you can use the 'Configure' button to place a predefined content in the template. When creating new C/C++ source and header files, the content will be inserted automatically at the beginning of a file.

Eclipse provides different ways of adding a file to a project

1. Import a file, which physically copies a file into the project folder within the workspace
2. Create a file in the project folder
3. Create a link to an existing file in the project folder

### 2.3.3 Project Templates

After setting up a project with a configuration for compiler switches, optimization and PowerPC specific switches, you may want to use this project as a template for new projects. Select the project in the C/C++ Project view and use the context menu copy. Afterwards choose the context menu paste and enter a new name for the copied project.

### 2.3.4 Configuring the UDE Debug menu

The Universal Debug Engine integrated in Eclipse is a high-end debug solution. In the default UDE Debug perspective a lot of features are accessible via toolbar icons. If you want to minimize the available toolbar icons in the UDE perspective, please use the so-called 'Framework-Filter' feature.

1. In the UDE perspective choose 'Config' → 'Debug-Server Configuration'
2. In the subsequent dialogue you have to select Basic Settings within the Tree Framework

3. Check 'Enable hiding of GUI frameworks elements'
4. Apply your changes via the OK button and restart Eclipse so that your changes can take affect
5. In the UDE perspective choose the menu 'Config' → 'Debug-Server Configuration' and navigate to the menu 'Framework' → 'Framework-Filter' . Here you can modify the visibility of UDE toolbar icons and menu. These modification require a restart of Eclipse

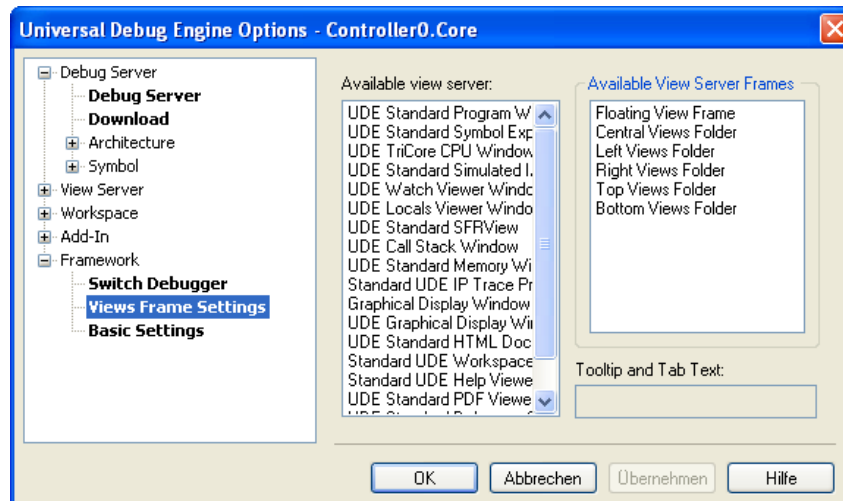


Figure 2.10: Configure UDE menus

### 2.3.5 Searching within Eclipse

Eclipse provides various powerful search mechanism, for example:

1. Select a text in the Eclipse editor and call the context menu 'Search Text' → 'Project' . All occurrences will be displayed in the Search console.
2. Call the menu 'Search' → ' C/C++' . Within this dialogue you can use different scopes which ease the search in large software projects.

### 2.3.6 Help System

The user's guide of the PowerPC Development Platform UAD2+ is accessible via the menu 'Help' → ' Help Contents' . Please select the entry 'HighTec EDV-Systeme' → 'PowerPC' → ' Development Platform' from the contents to open the corresponding help (see [Figure 2.11](#) on page 12).

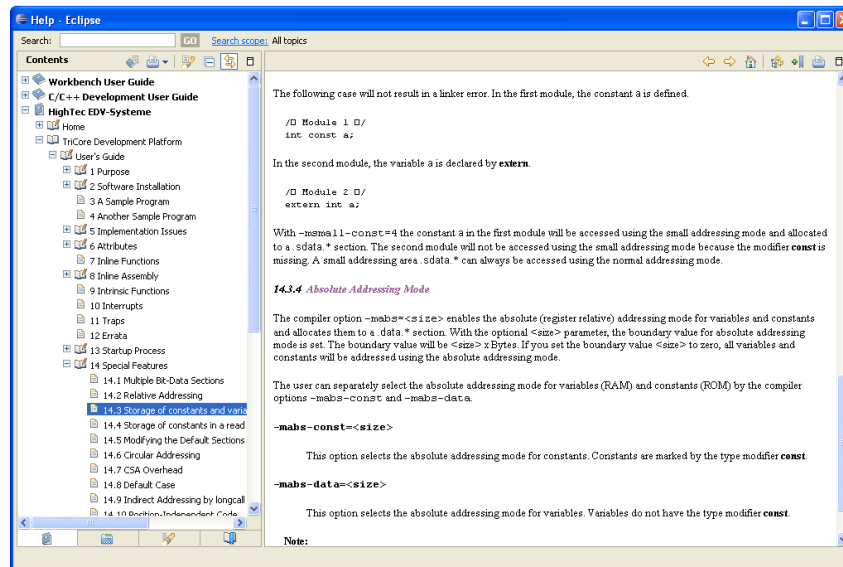


Figure 2.11: Integrated Help

### 2.3.7 Migrating projects

If you have projects which are created with a former version of the PowerPC Development Platform UAD2+, then you have to update your project description. If you build such a project an error will indicate the problem. In the console Problems you can select the corresponding entry and eliminate the problem by executing the context menu 'Quick Fix'.

### 2.3.8 Eclipse Macros

Eclipse provides different macros which can be accessed within the project settings or pre/post-build steps. One use-case is to use some of the macros to make the Eclipse projects portable.

#### File-specific macros

The set of file-context macros is predefined by the Managed Build System (MBS). Neither a tool-integrator nor a user can define new file-context macros.

#### $\${InputFileName}$

Represents the input file name. The input file has the following meaning:

1. If a tool does not accept building multiple files of the primary input type with one tool invocation, the input file is the file of the primary input type being built.
2. If a tool accepts building multiple files of the primary input type with one tool invocation the input file is undefined and the macros representing the input file contain information about one of the inputs of the primary input type being built.

**`#{InputFileExt}`**

Represents the extension of the input file.

**`#{InputFileBaseName}`**

Represents the base name of the input file. That is the file name with an extension stripped.

**`#{InputFileRelPath}`**

Represents the input file path relative to the builder current directory.

**`#{InputDirRelPath}`**

Represents the input file directory path relative to the builder current directory.

**`#{OutputFileName}`**

Represents the output file name. The output file has the following meaning:

1. If a tool is not capable of producing multiple files of the primary output type with one tool invocation the output file is the file of the primary output type that is built with a given tool invocation.
2. If a tool is capable of producing multiple files of the primary output type with one tool invocation the output file is undefined and the macros representing the output file contain information about one of the files of the primary output type that are built with a given tool invocation.

**`#{OutputFileExt}`**

Represents the output file extension.

**`#{OutputFileBaseName}`**

Represents the output file base name. That is the output file name with an extension stripped.

**`#{OutputFileRelPath}`**

Represents the output file path relative to the current builder directory.

**`#{OutputDirRelPath}`**

Represents the output file directory path relative to the current builder directory.

**Configuration-specific macros****`#{ConfigName}`**

Represents the name of a given configuration.

**`#{ConfigDescription}`**

Represents the description of a given configuration.

**`#{BuildArtifactFileName}`**

Represents the name of the build artifact.

**`#{BuildArtifactFileExt}`**

Represents the extension of the build artifact.

**`#{BuildArtifactFileBaseName}`**

Represents the base name of the build artifact.

**`#{BuildArtifactFilePrefix}`**

Represents the prefix of the build artifact.

**`\${TargetOsList}`**

Represents the list of the target OS names.

**`\${TargetArchList}`**

Represents the list of the target Arch names.

**Project-specific macros****`\${ProjName}`**

Represents the name of a given project.

**`\${ProjDirPath}`**

Represents the absolute path of a given project.

**Workspace-specific macros****`\${WorkspaceDirPath}`**

Represents the workspace absolute path.

**`\${DirectoryDelimiter}`**

Represents the directory delimiter used on the system. That is the `\` for Win32 systems and the `/` for Unix-like systems

This could be useful, e.g. in the case a user needs the absolute path of an input file.

The absolute path would be represented in the following way: `WDirectoryDelimiterinputFileRelPath$CSD`

**`\${PathDelimiter}`**

Represents the default path delimiter used on the system to separate paths in the path environment variables. That is the `;` for Win32 systems and the `:` for Unix-like systems

This might be used in the environment variable definition

**CDT/Eclipse installation-specific macros****`\${EclipseVersion}`**

Represents the current Eclipse version.

**`\${CDTVersion}`**

Represents the current CDT version.

**`\${MBSVersion}`**

Represents the current MBS version.

**`\${HostOsName}`**

Represents the operating system name on which eclipse is running.

**`\${HostArchName}`**

Represents the architecture name on which eclipse is running.

## 3 FAQ

### 3.1 Examples and Board Support Package

#### 3.1.1 What is the structure of the examples and the BSP?

There are a number of derivatives for the PowerPC which differ in the configuration of settings which are tied very closely to the underlying hardware such as timer unit, pins for LEDs, and interface configuration (serial, TCP). The different settings for the included examples are summarised in the board support package directory `bsp`. It is thus possible to compile one example for several derivatives. Only the derivative-specific files have to be included in the board support package. Instructions on the settings for the derivatives can be found at [subsection 3.2.1](#) on page 15.

#### 3.1.2 How can I use target-specific header files?

The PowerPC Development Platform UAD2+ contains target-specific header files, which may be used for a direct access to the SFRs of a special derivative. You may use these header files by including either the common header file for your target or by only including the special header file(s) for the peripherals you are using. The first alternative includes a huge number of files and will slow down compiling, yet there is no risk of a mix-up since the correct header file is automatically included.

### 3.2 Using the Compiler and the Tools

#### 3.2.1 How can I configure the compiler to support my derivatives?

The compiler has a set of pre-configured derivatives which may be set by the option `-mcpu=<derivative>`.

#### 3.2.2 How to enable VLE support in the GNU compiler?

The HighTec GNU compiler for PowerPC supports the VLE instructions of E200 Power Architecture family.

- |                           |  |
|---------------------------|--|
| <code>-mcpu=e200zx</code> | This compiler option enables the code generation for E200 family. If this option is enabled the compiler passes the option <code>-me200zx</code> to the assembler. This option is set per default.   |
| <code>-mvle</code>        | This option generates standard EBOOK instructions and passes this option to the assembler. If this option is enable then the assembler will convert the EBOOK instructions in corresponding VLE instructions. In this pass the code sections <code>.text</code> are renamed to <code>.text_vle</code> and the VLE sections are marked with the section flag <code>SHF_PPC_VLE</code> . This option is set per default. |

**--warn-flags** Since the compiler support the mix of VLE and non-VLE code the linker must be able to check the consistency of the code. Since the VLE code sections are marked with the flag `SHF_PPC_VLE` the linker is able to check the compatibility of the sections which are linked. The linker option **--warn-flags** enables the check.

In the linker description file the additional function `FLAGS` can be used to specify the flags of an output section

### Example

```
.text_vle : FLAGS(axv)
{
    *(.text_vle*)
}
```

In this example the flags `axv` (allocatable, executable, vle) are set for the output section `.text_vle`.

Within the source code an input section (e.g. VLE section) can be generated via the pragma syntax

```
#pragma section ".text_vle" axv
/* the function which should be put in the VLE section .text_vle */
#pragma section
```

## 3.2.3 How to enable floating point support in the Power Architecture?

Some derivatives of the Power Architecture include a single instruction multiple data unit for or single and double precision instructions. The so-called SPE must be initialised in the start-up code the `crt0.c`. The compiler option `-mspe` enables the generation of optimized instructions of the SPE unit. If this unit is not available the compiler option `-msoft-float` can be used as software emulation.

```
volatile __attribute__((vector_size(8))) int vd0={1,1},vd1={1,1},vd2={1,1};

int main (void)
{
    ...
        /* SPE sample */
        vd1 += vd2;
        vd2 = (vd1 / vd2);
    ...
}
```

## 3.2.4 How can I force the linker to use a function from a defined library?

The case may occur in a software project that functions with identical names exist in different libraries. A symbol is created for each function; thus for functions with identical names there will be several symbols of the same name. Using one of these functions will

cause a linker error due to the conflict of symbols. To avoid this situation, follow these steps:

1. Add the linker option `-u <symbol>` or `--undefined=<symbol>` (Replace `<symbol>` by the name of the function).
2. The library including the function you wish to use must be the first to be passed to the linker.

Due to the option `-u <symbol>` the symbol is undefined thus forcing the linker to resolve the reference by taking the first matching symbol it encounters in the library.

### 3.2.5 How do I resolve references in libraries?

If you pass a list of libraries to the linker, the linker will parse the libraries in the given order and try to resolve the references. If the linker fails to resolve all the references, e.g. due to the existence of cyclic references, then the following message will be issued:

```
undefined reference to symbol name
```

The linker can also automatically resolve cyclic references so that the order of libraries becomes irrelevant. The corresponding linker option is `-Wl,--start-group <list> -Wl,--end-group`. `<list>` is used here a placeholder for your list of libraries.

### 3.2.6 How can I allocate a data or function/section to a fixed address?

To place a section at a specific address, you have to apply this address in the section definition in the linker script file. I.e.

```
.data.mydata 0xd0001040 :
{
    *(.data.mydata*)
}
```

this linker script command tells the linker to put all input sections `.data.mydata*` of all input objects into the output section `.data.mydata` and locate this section at the address `0xd0001040`. If you like to locate only the input sections of some moduls into the output section, you can exchange the wildcard `*` with the module name, i.e

```
.data.mydata 0xd0001040 :
{
    mymodul1.o(.data.mydata*)
    mymodul2.o(.data.mydata*)
}
```

To locate a function at specific address, you can use the compiler option `-ffunction-sections` to tell the compiler to define an own section for each function. The compiler will name the section `.text.<functionname>`. To locate this function at a specific address you can follow the above description. If you have several modules with you can also add the modulname in the linker script command like

```
.myfunction 0xa0001000 :  
{  
modul.o(.text.functionname)  
}
```

This command will allocate the section `.text.functionname` of the object `modul.o` into the output section `.myfunction` and locate this section at the address `0xa0001000`.

A different solution is to define a memory region at the desired address and then the corresponding output section e.g. `.mysection` should be redirected to the defined memory region.

```
.mysection :  
{  
...  
} > name_memory_region
```

Definitions of memory areas in the Linker Description File are prefixed with `MEMORY`.

**Note:**

A combination of both methods is not permitted.

### 3.2.7 How can I locate a variable at a fix address?

To locate a variable at a fix address the following proceeding is advisable.

1. Define an output section e.g. `.myvar`
2. Assign the section to fix address
3. Define the symbol/variable `VAR_NAME` at the current address (the dot `.` is used to set the location counter).
4. The command `LONG(<initial value>)` will initialize the variable and in addition the necessary memory is allocated.

The following snippet of the linker description file shows the corresponding sequence.

```
SECTION {  
...  
  
    .myvar 0xd0008000 {  
        VAR_NAME = . ;  
        LONG(<initial value>) ;  
    }  
}
```

The variable can be accessed from the C/C++ code like:

```
extern type VAR_NAME;
```

### 3.2.8 How can I take a look at the intermediate files generated by the compiler?

Whenever you compile C/C++ files, so-called preprocessed intermediate files are generated that are normally stored in a temporary directory and deleted after completion of the compiling process. These files, which are suffixed `.i`, `.s` and `.o`, can be preserved if the option `-save-temps` is set.

### 3.2.9 How can I pass options to the assembler and linker?

The GCC Compiler Driver controls all compiler calls, assembler calls and linker calls during the process of generating an executable. Notation for passing parameters is as follows:

- `-Wa,<Options>` `<Options>` are passed from `ppc-vle-gcc` directly to the assembler. Multiple options are to be separated by commas.
- `-Wl,<Options>` `<Options>` are passed from `ppc-vle-gcc` directly to the linker. Multiple options are to be separated by commas.
- `-T <file>` `<File>` is used as a Linker Description File. This file configures the memory layout and the storage location for source code and data.

### 3.2.10 How can I speed up the build process?

The `ppc-vle-gcc` can be started several times simultaneously thereby allowing parallel compilation processes that can considerably speed up the build process. This effect is particularly obvious with the multi-core architectures of personal computers. In Eclipse, the settings are defined in 'Properties' → 'C/C++-Build' on the 'Behaviour' tab of the 'Use parallel build' menu. Make supports parallel build via the `-j <Number of Builds>` command line option.

**Note:**

During the build process, the compiler usually generates temporary intermediate files that are processed by the assembler. Setting the compiler option `-pipe` causes further speed-up by omitting the generation of intermediate files and, instead, transferring the results of the compiler directly to the assembler via pipes.

### 3.2.11 How do I use the alias attribute?

With the attribute `alias` one variable can be accessed with different names. These names do not occupy additional memory resources. Typical Use-Cases are:

- Ensure the compatibility of variables for different software versions
- Compliance to naming convention of supplier software

Here a small example:

```
int oldname = 1;

extern int newname __attribute__((alias("oldname"))); // declaration

int foo (int bar)
{
    bar = oldname;
    bar = newname;
    return bar;
}
```

The definition `int oldname = 1` will reserve memory for the variable. The following declaration with the `alias` attribute will create a symbol without allocating new memory. If you use the option `-Wl,-Map=<name>.map` and then inspect the generated mapfile, you will see that both variables have the same address. The variables are identical and can be accessed with both names.

### 3.2.12 How can I manage the compiler settings globally?

If certain compiler options, such as switched-on warnings, are to be set as default values, make files are often used for the build process wherein such options, include paths or other details are defined for the compiler. A much more elegant solution is, however, to pass a configuration file to the compiler which contains the desired settings. The input of a file can be interpreted as command line options. Use the following syntax in the command line to read the options from a file.

The input of the configuration is specified by the `@<configuration file>` option.

### 3.2.13 How can I disassemble a hex file?

A hex-file can be disassembled like:

```
ppc-vle-objdump -m <architecture>:v<coreversion> -D <file>.hex
```

With the option `-m` the architecture can be passed. `-D` makes a disassemble of all sections.

### 3.2.14 How can I store data in packed structures?

With aligned storage, empty spaces between structural elements and/or data can occur. There are applications, however, that require data storage in a structure without gaps. An example for such a case is the communication between a microcontroller and an external component that expects data without gaps in a certain format. This functionality is activated by the `packed` attribute.

#### Example

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

**Note:**

This method of packed storage prevents an optimised access to variables.

### 3.2.15 Why does the compiler not use any FPU instructions?

The GNU Compiler complies to the ANSI standard. This means that, by default, a number 1.0 is regarded as **double**, not as **float**. In the below example, a **double** is converted to **float**. Of course, such a code is not efficient for a with built-in FPU.

```
float x_f = 1.01
int
main (void)
{
float y_f:
y_f = 1.0/x_f; /* divide double by float */
return 0;
```

If the compiler option `-fshort-double` is used, the compiler will assume identical sizes for **float** and **double**. This option does not comply to the ANSI standard since doubles have a higher resolution than floats, meaning the result of the latter is less accurate. The advantage is, however, an improved efficiency of the FPU.

**Note:**

Instead of using the option `-fshort-double`, float should be explicitly declared as such, i.e. `1.0f` should be stated instead of `1.0`.

### 3.2.16 How can I use floating point functions from a library?

The mathematical library `libm.a` contains trigonometric functions like `sin()`. These functions do not use floating point operations. If you want to use the floating-point variant of e.g. `sin()` then you have to call explicitly `sinf()`. All the floating-point functions have the suffix `f`.

### 3.2.17 How can I have Instruction Pointer Profiling displayed by the UDE?

Set the 'Timer used for Time Measurement' entry in the 'Debug' tab of the 'Target Interface Setup' to `STM_TIM1`. Then click the clock symbol in the UDE to open the 'Program execution time' dialogue box. Tick the 'Enable program execution time measurement' checkbox and set 'Single step timer mode'. The time interval can be set to 10 microseconds. To start recording, you have to run your application in the debugger and switch on the display in 'Views' → 'Instr. Ptr. Trace Profiling'.

### 3.2.18 How can I initialise variables within the UDE?

Part of the memory of an application is often used for calibration information which are initialised with different configurations at startup. UDE offers the possibility to automatise

such processes by initialising variables via a script. Below, an excerpt is shown for the Timerdemo of the UDE installation.

```
function GetTime()
{
var Debugger;
Debugger=Workspace.CoreDebugger(0);

var HoursVar=Debugger.Expression("ucHours");
var MinutesVar=Debugger.Expression("ucMinutes");
var SecondsVar=Debugger.Expression("ucSeconds");

Debugger.ConOutput("Get time ...\n");
Debugger.ConOutput("Hours: " + HoursVar.Value + "\n");
Debugger.ConOutput("Minutes: " + MinutesVar.Value + "\n");
Debugger.ConOutput("Seconds: " + SecondsVar.Value + "\n");

}

function SetTime()
{
var Debugger;
Debugger=Workspace.CoreDebugger(0);

var HoursVar=Debugger.Expression("ucHours");
var MinutesVar=Debugger.Expression("ucMinutes");
var SecondsVar=Debugger.Expression("ucSeconds");

Debugger.ConOutput("Set time ...\n");
HoursVar.Value=1;
MinutesVar.Value=23;
SecondsVar.Value=45;
}
```

### 3.2.19 How can I save resources with Simulated I/O

A computer usually has some communication interfaces to the outside world. The most simple of these are written input and output via a command line or text files. These interfaces enable a program to react to user input, and to output data on a screen or in files, e.g. logfiles.

Input and output on a command line presuppose the existence of a keyboard and a display (usually a monitor), or access to a file system. However, these preconditions are rarely fulfilled in the embedded sector. One possibility of communication is via interfaces, such as the serial port or the I<sup>2</sup>C bus. In many cases, these cannot be used for input/output purposes since they are occupied during the application at the target. That's why the On-Chip Debug System (OCDS) existing at the target is used for input and output. This system is connected to the debugger during debugging. The debugger has an exact knowledge of the executed code and can therefore display any data output by the application, via a simulated, virtual display, the so-called Simulated I/O.

Input/output functions such as `printf()` or `getchar()` are included. These functions access low-level functions which, in 'normal' operating systems, send or receive characters to and from the operating system `write()` and `read()`.

**Note:**

The `printf` function is not very efficient for embedded targets. Output of integers via the `iprintf` instruction (`iprintf (%d, int)`) can considerably save resources compared to the common `printf` instruction:

```
iprintf(%d, int)
```

## 3.3 Tools for displaying modifying object files

**ar** creates, modifies and extracts from archives,

**ranlib** generates an index for archive contents,

**nm** lists symbols from the object file,

**ppc-vle-objcopy** copies and converts object files to various formats,

**ppc-vle-objdump** displays various information from object files,

**ppc-vle-addr2line** converts addresses to file name and line number,

**ppc-vle-readelf** displays the contents of an ELF format file,

**ppc-vle-size** lists file section sizes and absolute size,

**ppc-vle-strings** lists the strings from a file,

**ppc-vle-strip** removes symbols.

### 3.3.1 Objcopy

Important options include the following:

`-O --output-target <bfdname>`

Creates an output file in `!bfdname!` format;

`-j --only-section <name>`

Copies the selected section of the input file to the output file;

`-R --remove-section <name>`

Removes sections with the name `<sectionname>` from the output file;

`--add-section <sectionname>=filename`

Adds a new section `<sectionname>` from `<filename>` to the file;

`--rename-section <old>=<new> [, <flags>]`

Renames and modifies sections. von `<flag>`

Examples:

### Intelhex Format

```
ppc-vle-objcopy -O ihex Input.elf Output.hex
```

### Creating a binary file

```
ppc-vle-objcopy -O binary Input.elf Output
```

### Removing the debug section

```
ppc-vle-objcopy -R .debug info Output.elf
```

## 3.3.2 Objdump

- D, --disassemble-all  
Displays the assembler mnemonics for machine instructions;
- h, --section-headers  
Summary of the section headers from the object file;
- j, --section=<NAME>  
Displays the information of the section <NAME> exclusively;
- t, --syms  
Displays the entries for the symbol table;
- S, --source  
Displays the source code together with the corresponding assembler output (implicit option -d).

Examples:

```
ppc-vle-objdump -h main.o
```

## 3.3.3 NM

- f, --format=<format>  
Uses <format> as the output format; ('bsd', 'sysv' oder 'posix')
- g, --extern-only  
Displays external symbols exclusively;
- n, --numeric-sort  
Sorts symbols according to addresses;
- size-sort  
Sorts symbols according to size;
- u, --undefined-only  
Displays undefined symbols exclusively (access to external object file).

Examples:

### Displaying external symbols

```
ppc-vle-objdump -g main.o
```


### Displays undefined symbols

```
ppc-vle-objdump -u main.o
```

### 3.3.4 How can I convert a program to Intel Hex format?

The tool `objcopy` supports different output formats. To generate an Intel Hex format, you can use the command line:

```
ppc-vle-objcopy -O ihex <elf-inputfile> <hex-outputfile>
```

Another possibility is to add this command as a post-built step in Eclipse. The command can be added in the 'Pre/post built steps' tab of the compiler settings .

```
cmd /c ppc-vle-objcopy -O ihex <ututFileName$O> <ututFileName$O>.hex
```

### 3.3.5 How can I strip unneeded information from an object file?

The software of ECUs consist of software modules from different suppliers. The interest of these companies is to protect their software know-how. Therefore it is advisable to strip all information from the object, which is not relevant for building the software. Normally the build process only requires the relocation information of the object file.

The command line tool `ppc-vle-strip` removes unneeded information from the object file.

```
ppc-vle-strip -x -X -g --strip-unneeded <objectfile>.o
```

### 3.3.6 How can I change the IP Address of my target?

Some PXROS examples use the PXROS TCP/IP stack `PXtcp`. These examples communicate via TCP/IP and use the IP address 192.168.0.199 as their default address. To include the board where the example is running into your network, you have to change the IP address to an address matching your network. This can be done in the file `tcpipcfg.h` of the sample program. This file is part of the project if you open the example in Eclipse.

### 3.3.7 How can I configure PXview for tracing?

The real-time operating system PXROS/PXROS-HR, internal services such as sending and receiving messages, scheduling etc, are traced in the target. The trace information can be transferred to a host PC via a TCP or UART-interface, and be visualised via PXview. A detailed description of PXview is provided in the corresponding PXview manual.

The following functions are used for configuring PXview. The complete initialisation process is included in the file `Sample.c` of the PXROS-HR example `PXROS-HR-Sample`.

```
PxTraceCtrl(PXTraceSetTicksPerSecond, (PxArg_t) PxTickGetTicksFromMilliseconds(1000))
    Set the time base of the PXview trace.
```

```
PxTraceCtrl(PXTraceSetState, (PxArg_t) PXV_ALL)
    Trace user entries and events.
```

```
PxTraceCtrl(PXTraceSetTraceState, (PxArg_t) PXVT_ALL)
    Trace all operating system services of PXROS-HR. You can select different services to be included in the trace. The trace is configured with the aid of a so-called trace mask (see the table below for details).
```

```
PxTraceCtrl (PxTraceSetPxrosTrace, (PxArg_t) TRACE_START)
```

Start PXview trace. Terminate tracing With the parameter TRACE\_STOP.

PXVT_DELAY	Delay Jobs
PXVT_DIE	Die-Server
PXVT_ERROR	Errors
PXVT_EVENT	PXROS Events
PXVT_GENID	Generation ID
PXVT_GSREF	Global references
PXVT_HND	Handler
PXVT_INTS	Interrupt Service Routines
PXVT_MBX	Mailbox
PXVT_MC	Memory classes
PXVT_MODE	Mode bits
PXVT_MSG	Messages
PXVT_NAME	PXROS Nameserver
PXVT_OBJ	Objects
PXVT_OPOOL	Objects pools
PXVT_PE	Periodic PXROS Events
PXVT_PREF	Protected references
PXVT_PRIV	Privilege levels
PXVT_REM	Remote handler
PXVT_SEMA	Semaphore
PXVT_TASK	Tasks
PXVT_TIMESLICE	Timeslices
PXVT_TO	PXROS Timeout Events
PXVT_TRACE	Trace events

### 3.3.8 How can I add user-defined marks in PXview?

The PXview trace is available without instrumenting your source code. If you wish to add user-defined 'markers' to the trace, add the function call `UserEntry` or `UserEvent` in your source code. These functions provide two parameters to transfer user-defined data. The prototypes of the functions are:

```
void PxvAddUserEntry (PxTask_t From,
                    unsigned long Data1,
                    unsigned long Data2);
```

```
void PxvAddUserEvent (PxTask_t From,
                    unsigned long Type,
                    unsigned long Data1,
                    unsigned long Data2);
```

### 3.3.9 How do I install a Handler in PXROS-HR?

An interrupt in PXROS-HR is defined by the following prototype:

```
void HandlerFunction (PxArg_t arg)
{
...
}
```

To register such an interrupt handler, a so-called interrupt object is required. This object is requested via the statement:

```
IntObj = PxInterruptRequest(PXOpoolTaskdefault);
```

Since in PXROS-HR, a handler is always assigned to a task, this instruction should be executed at the beginning of the task in question. Installing a handler within a task is done by the following function:

```
PxIntInstallHandler(INTERRUPT_NR, IntObj, (void (*)(PxArg_t))HandlerFunction, (PxArg_t )
```

**Note:**

Within a handler function, PXROS-HR services may only be used in connection with the extension `_Hnd`.

### 3.3.10 Why can't I debug via the Pxlinit function?

The function `Pxlinit` in the main-loop of your PXROS-HR application will never return, meaning a step over this function in the debugger is not possible. This function `Pxlinit` performs the initialisation process of your PXROS application.