Getting started with the PowerPC tools:

Compiler, IDE and debugger

This Application Note describes how to get started with the Cosmic Toolchain for PowerPC. Using one of the examples provided with the compiler, you will be guided through:

- compiling (understanding the main options)
- linking (using one of the provided linker files)
- and debugging (executing step by step, viewing code, memory and peripherals).

The second part of this document deals with the basics of the following common situations:

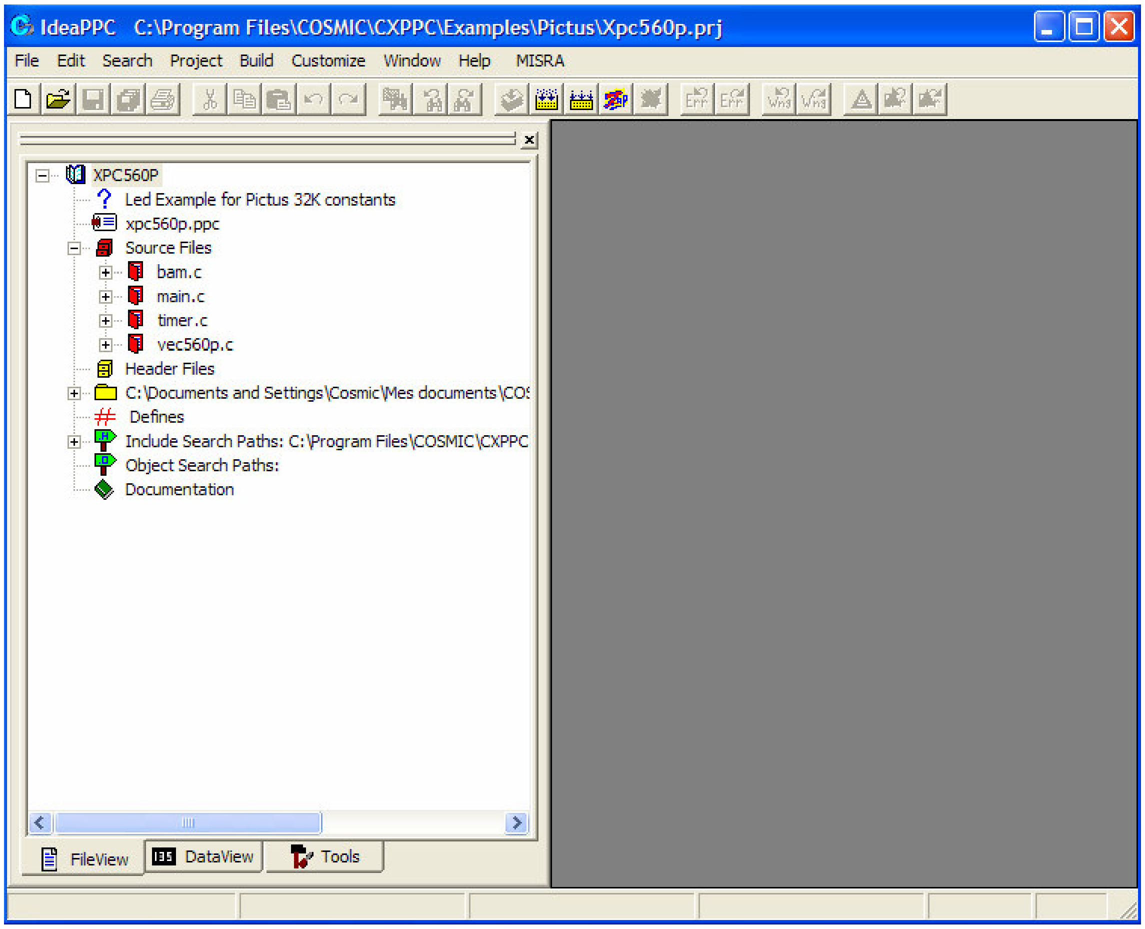
- porting the example to another chip
- using external debuggers or running standalone
- sharing projects with your Customers / Colleagues
- compiler versions

Before you get started, please consider the following important issues:

- **Some parts of this application note only apply for compiler version v4.2.8 (November 2010) and higher**
- Two versions of the PowerPc tools exist. The VLE-only version (called CxVLE) and the full version (CxPPC). This Application Note will always mention CxPPC, but most of its content is applicable to CxVLE without modifications.

Opening the example project

When you first install the Cosmic tools for PowerPC, you will see the "Cosmic CxPPC v4.2.8" icon appear on your desktop. When you double click on this icon, you start IDEA (the Cosmic Integrated Development Environment). IDEA will first show a screen with some credits and information, including a version number which is the version number of IDEA itself, and **NOT** the version number of the compiler or debugger (see the end of this document for more information about compiler versions).
Once IDEA is started, go to Project -> Load, browse to directory C:\Program Files\COSMIC\CXPPC\Examples\Pictus (assuming you installed the tools in their standard directory) and select the file Xpc560p.prj. Your screen should look like:


IdeaPPC C:\Program Files\COSMIC\CXPPC\Examples\Pictus\Xpc560p.prj
File Edit Search Project Build Customize Window Help MISRA
FileView
DataView
Tools

XPC560P
Led Example for Pictus 32K constants
xpc560p.ppc
Source Files
bam.c
main.c
timer.c
vec560p.c
Header Files
C:\Documents and Settings\Cosmic\Mes documents\COS
Defines
Include Search Paths: C:\Program Files\COSMIC\CXPPC
Object Search Paths:
Documentation

The project just loaded is an example that was developed for the P&E board and uses this board's features (LEDs and buttons) to allow testing/running the code on real hardware. This example is written for a Pictus device (so plug in the Pictus mini-module if you want to see it working), but, as we will see later, it is quite easy to port it to similar devices (Bolero, Spectrum, …, or to any VLE-only device with no MMU)

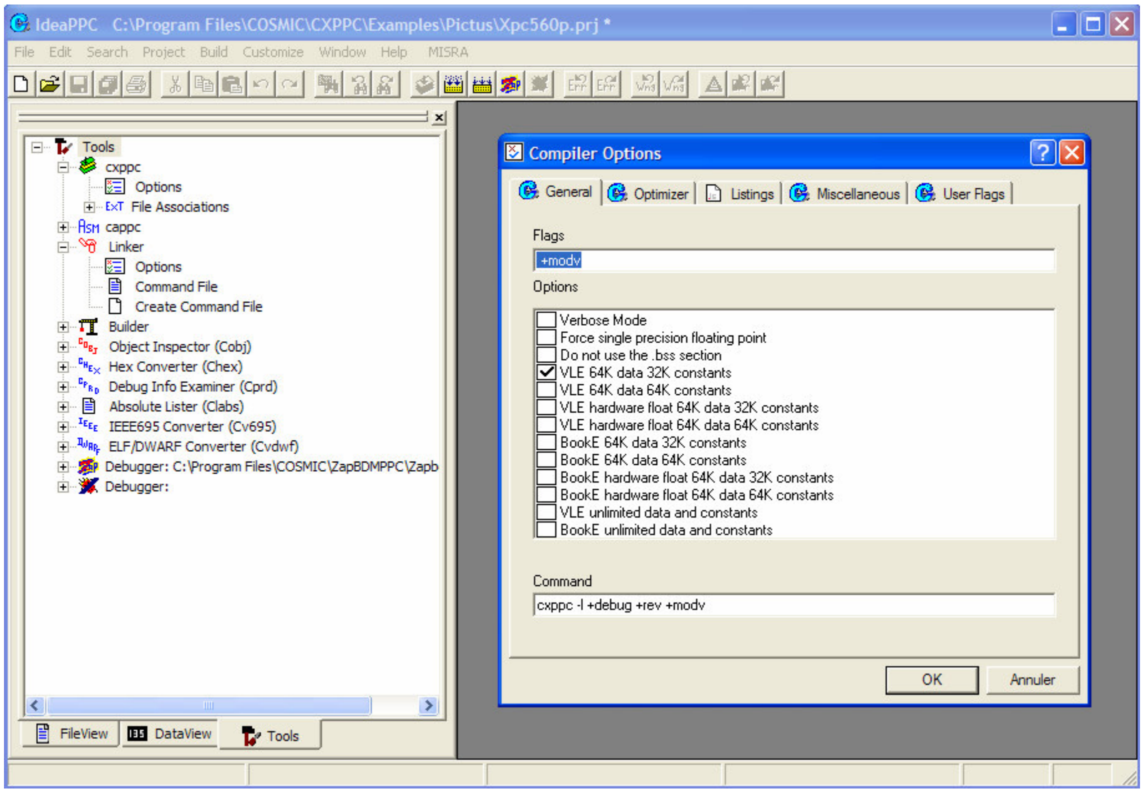
The main information in this screenshot is:

- the list of source files. Our example is made of 4 C files and no assembler files.
- the name of the executable image (xpc560p.ppc).
- the "working directory" (<My Documents>\COSMIC\CXPPC\Examples\Pictus, just below the empty line named Header Files). Within IDEA, every time a file is not specified with a full path, it will be searched for in the Working Directory

The main buttons that we will use are the three in the middle of the top line (those that are not greyed-out):

- The "ZAP" button will launch the ZAP debugger and automatically load the application. Note that it must first be configured by going into menu Build -> Setup Tools -> Debugger and choosing Zapbgppc.exe in its installation directory.
- The "ReBuild all" button (just to the left of the ZAP button) will rebuild the whole application
- The "Build" button (the left one) will build the application by only recompiling those files that need to be recompiled (shown in red in the list) and relinking all the objects.

Let's now click on the "Tools" tab at the bottom on the screen and take a look at how we can configure the compiler and related tools. After right clicking on the line "cxppc" and selecting "Options", the screen should look like:



IdeaPPC   C:\Program Files\COSMIC\CXPPC\Examples\Pictus\Xpc560p.prj *

File Edit Search Project Build Customize Window Help MISRA

- Tools
  - cxppc
    - Options
    - File Associations
  - cappc
  - Linker
    - Options
    - Command File
    - Create Command File
  - Builder
  - Object Inspector (Cobj)
  - Hex Converter (Chex)
  - Debug Info Examiner (Cprd)
  - Absolute Lister (Clabs)
  - IEEE695 Converter (Cv695)
  - ELF/DWARF Converter (Cvdwf)
  - Debugger: C:\Program Files\COSMIC\ZapBDMPPC\Zapb
  - Debugger:

FileView DataView Tools

Compiler Options

General | Optimizer | Listings | Miscellaneous | User Flags

Flags

+modv

Options

- [ ] Verbose Mode
- [ ] Force single precision floating point
- [ ] Do not use the .bss section
- [✓] VLE 64K data 32K constants
- [ ] VLE 64K data 64K constants
- [ ] VLE hardware float 64K data 32K constants
- [ ] VLE hardware float 64K data 64K constants
- [ ] BookE 64K data 32K constants
- [ ] BookE 64K data 64K constants
- [ ] BookE hardware float 64K data 32K constants
- [ ] BookE hardware float 64K data 64K constants
- [ ] VLE unlimited data and constants
- [ ] BookE unlimited data and constants

Command

```
cxppc -l +debug +rev +modv
```

OK Annuler

The left window contains a list of tools that can be individually configured by clicking on them.

The most important ones are

- The compiler (cxppc). It will be discussed below

- The linker. By clicking on “command file” you open the linker command file, which is the file where you specify what to link (the list of objet files, the libraries..) and where to link it in memory (the memory addresses and sizes). The link command file must be customized when you change chip derivative (There is another application note for this).
- The Builder: this is where you specify what actions must be executed when you build your projet; for example, if you want an elf/dwarf file to be produced (in order to debug with external debuggers).

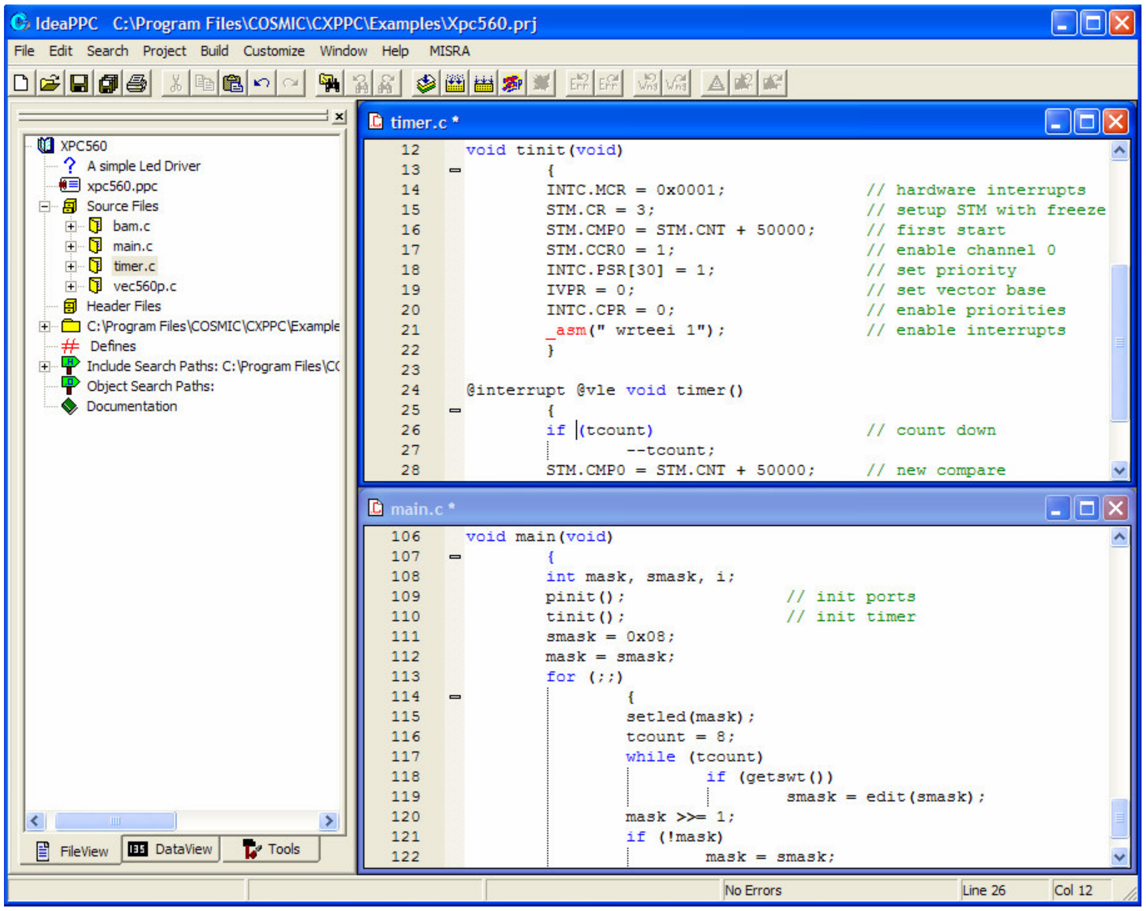
Let’s now view the compiler options. When you right click on cxppx->options, the box “Compiler Options” appears, as shown in the screen capture above.
Inside this box you can set all the compiler options: the main ones are grouped in categories (one for each tab in the top row) and can be selected with the mouse (example: to select the +modv memory model, you just check the button “VLE 64k Data 32k constants”), while the less-used options can be entered, in command-line style, after clicking on the “User Flags” tab.
At all times, the bottom line (called “Command” and that must NOT be edit directly), shows the command line that will be passed to the compiler with the set of options currently selected.
The typical compiler options for a “small” PPC projet (less than 64k RAM, less than 32k constants, no code size limit) are those shown in the screenshot and explained below; if your project is too big for these options please read the application note “PPC_Compiler_Options”.

```
cxppc +debug +rev +modv
```

- +debug: tells the compiler to produce debug information. This is necessary if you want to be able to debug your application at the source level (see the source code, execute step by step, set breakpoints..), regardless of the debugger you will use
- +rev: tells the compiler to reverse the order of bits in the bitfields. You need this option in order to use most non-Cosmic header files.
- +modv: tells the compiler to use the “VLE” memory model (see the “PPC_Compiler_Options” application note for more details)

Building and Debugging

We are now ready to build and debug our example project using ZAP.
Within IDEA, hit the "ReBuild All" button: all the files will be compiled and their icon will become yellow, as shown below


IdeaPPC C:\Program Files\COSMIC\CXPPC\Examples\Xpc560.prj
File Edit Search Project Build Customize Window Help MISRA
XPC560
A simple Led Driver
xpc560.ppc
Source Files
bam.c
main.c
timer.c
vec560p.c
Header Files
C:\Program Files\COSMIC\CXPPC\Example
Defines
Include Search Paths: C:\Program Files\C(
Object Search Paths:
Documentation
FileView
DataView
Tools
timer.c *
main.c *
No Errors
Line 26
Col 12

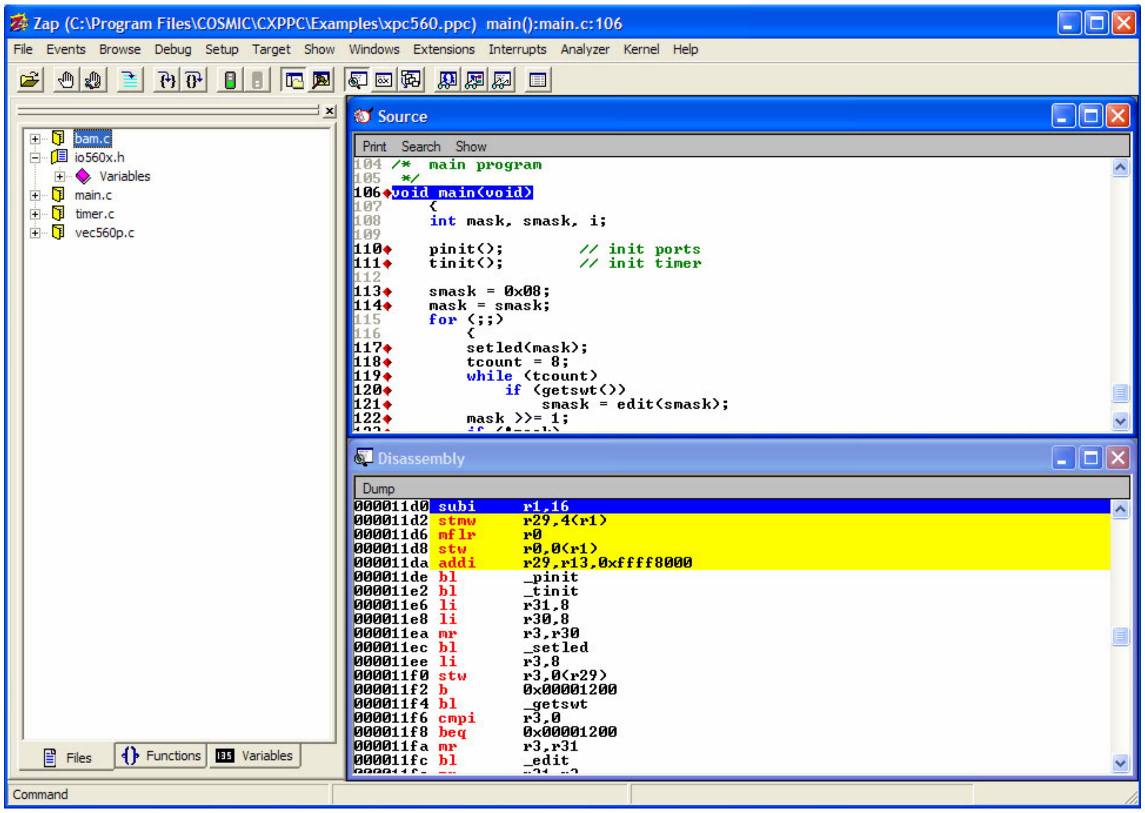
```
void tinit(void)
        {
        INTC.MCR = 0x0001;              // hardware interrupts
        STM.CR = 3;                     // setup STM with freeze
        STM.CMP0 = STM.CNT + 50000;     // first start
        STM.CCR0 = 1;                   // enable channel 0
        INTC.PSR[30] = 1;               // set priority
        IVPR = 0;                       // set vector base
        INTC.CPR = 0;                   // enable priorities
        _asm(" wrteei 1");              // enable interrupts
        }

@interrupt @vle void timer()
        {
        if (tcount)                     // count down
                --tcount;
        STM.CMP0 = STM.CNT + 50000;     // new compare
```

```
void main(void)
        {
        int mask, smask, i;
        pinit();                // init ports
        tinit();                // init timer
        smask = 0x08;
        mask = smask;
        for (;;)
                {
                setled(mask);
                tcount = 8;
                while (tcount)
                        if (getswt())
                                smask = edit(smask);
                mask >>= 1;
                if (!mask)
                        mask = smask;
```

The screenshot also shows the most important part of the application that we want to debug. This is an interrupt-based application (hardware interrupts) that uses the timer located at vector 30 as a time base in order to flash the LEDs. The main application does flash the LEDs according to a given mask until the buttons are pressed; when this happens, the mask is modified according to the buttons and then the main loop is resumed.
This specific example is highly interesting because it shows some aspects of an application that are typically compiler-dependent. For example, inside the file timer.c (partly showed in the screenshot) you will find an example of how to declare an interrupt function (that is linked to its own interrupt vector declared in file vec560p.c), how to initialize peripherals by accessing Peripheral Registers directly in C, and how to use simple inline assembler when necessary (example: enabling interrupts).

Once you have built the application, hit the ZAP button and ZAP will start, in a new window, looking like this:


Zap (C:\Program Files\COSMIC\CXPPC\Examples\xpc560.ppc) main():main.c:106
File Events Browse Debug Setup Target Show Windows Extensions Interrupts Analyzer Kernel Help
bam.c
io560x.h
Variables
main.c
timer.c
vec560p.c
Files Functions Variables
Source
Print Search Show
Disassembly
Dump
Command

```
/*  main program
 */
void main(void)
    {
    int mask, smask, i;

    pinit();            // init ports
    tinit();            // init timer

    smask = 0x08;
    mask = smask;
    for (;;)
        {
        setled(mask);
        tcount = 8;
        while (tcount)
            if (getswt())
                smask = edit(smask);
        mask >>= 1;
```

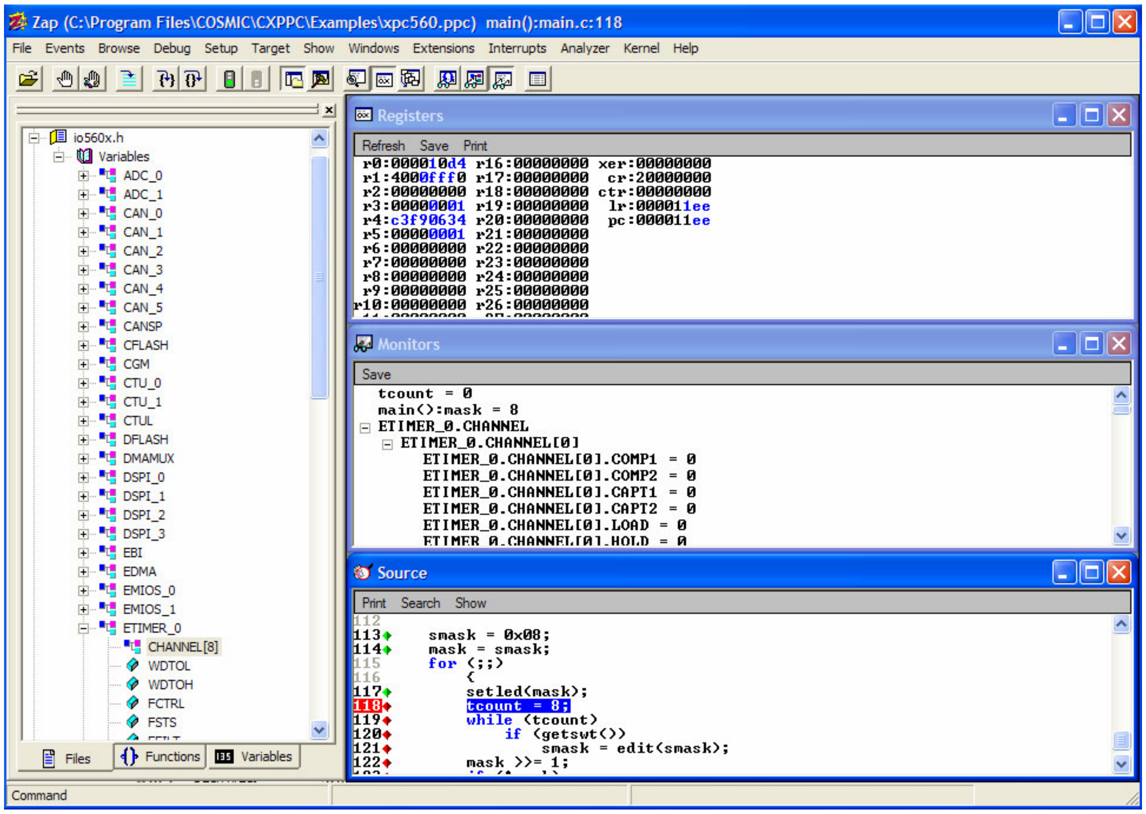
```
000011d0 subi     r1,16
000011d2 stmw     r29,4(r1)
000011d6 mflr     r0
000011d8 stw      r0,0(r1)
000011da addi     r29,r13,0xffff8000
000011de bl       _pinit
000011e2 bl       _tinit
000011e6 li       r31,8
000011e8 li       r30,8
000011ea mr       r3,r30
000011ec bl       _setled
000011ee li       r3,8
000011f0 stw      r3,0(r29)
000011f2 b        0x00001200
000011f4 bl       _getswt
000011f6 cmpi     r3,0
000011f8 beq      0x00001200
000011fa mr       r3,r31
000011fc bl       _edit
```

Notes:

- If ZAP asks something about using “Hardware Breakpoints” always answer in such a way that hardware breakpoints are used. The other option is only useful if you need more than 4 breakpoints, but it is not real-time and should be used only in special cases
- You must hit the “step into” button (just below the Debug menu item) once in order to execute the startup and stop on the main() function
- The number and position of the windows in the right pane could be different than showed
- In some cases, ZAP might not be able to find the source files of your application (typically when the project has been built with relative file names and the source are not in the working directory). If you see a message along the line of “Cannot find source file, would you like to change the PATH?”, answer YES and then select the path where the source files are and append it to the list of paths where ZAP looks for

the source files. <u>Note an important side effect of this</u>: if you are working on several projets, and more than one of them contain a file named (for example) main.c, ZAP might try to read the source for a main.c that is not the same that was compiled into the projet you are trying to debug: if this happens the source lines will be completely out of phase with the executable and you will not be able to set breakpoints or to follow the execution step by step. In order to correct this, change the list of paths so that the correct main.c is loaded.

You are now ready to debug, so here is a list of the basic debugging functions you will need.



```
 r0:000010d4 r16:00000000 xer:00000000
 r1:4000fff0 r17:00000000  cr:20000000
 r2:00000000 r18:00000000 ctr:00000000
 r3:00000001 r19:00000000  lr:000011ee
 r4:c3f90634 r20:00000000  pc:000011ee
 r5:00000001 r21:00000000
 r6:00000000 r22:00000000
 r7:00000000 r23:00000000
 r8:00000000 r24:00000000
 r9:00000000 r25:00000000
r10:00000000 r26:00000000
```

```
  tcount = 0
  main():mask = 8
⊟ ETIMER_0.CHANNEL
  ⊟ ETIMER_0.CHANNEL[0]
      ETIMER_0.CHANNEL[0].COMP1 = 0
      ETIMER_0.CHANNEL[0].COMP2 = 0
      ETIMER_0.CHANNEL[0].CAPT1 = 0
      ETIMER_0.CHANNEL[0].CAPT2 = 0
      ETIMER_0.CHANNEL[0].LOAD = 0
      ETIMER_0.CHANNEL[0].HOLD = 0
```

```
112
113◆    smask = 0x08;
114◆    mask = smask;
115     for (;;)
116         {
117◆        setled(mask);
118◆        tcount = 8;
119◆        while (tcount)
120◆            if (getswt())
121◆                smask = edit(smask);
122◆        mask >>= 1;
```

The lower window on the right shows the source code: you can single step through it (commands Step Into or Step Over) or you can set a breakpoint (by double clicking on the line number; it will become red like “118” in the picture) and then hit GO (green semaphore button). If you need to watch a variable, just select it with the mouse, then right click and choose “Monitor”: the variable will be added in the “Monitors” window (showed in the middle of the right pane in the picture) and will be updated every time the program is stopped. As you can see, complex variables are shown with their structure, which provides a useful feature: if a complete header file was included, all the peripherals can be accessed symbolically by just selecting their name in the list.

For additional debug features please refer to the ZAP user manual, but we would like to highlight one noteworthy point:

- debugging optimized applications can be tricky: some variables might appear as non existent (optimized out or moved to registers without the debugger being aware of it) or the code might have been factorized in a way that it does not look similar to the original anymore. If you have such problems try to rebuild your application disabling the optimizations (add "-no" in the user flags for the compiler and rebuild)

Common issues / FAQ

This chapter deals with some common issues that have been reported by new users of the compiler: it is a bit more detailed than the previous chapter but still far from being complete for the subjects it covers; please refer to the user manual or other application notes for more details.

The subjets that we will cover are:

- porting the example to another chip
- using external debuggers or running standalone
- sharing projects with your Customers / Colleagues
- compiler versions

Porting the example to another chip

As mentioned before, the example used in this application note was developed for a P&E board with a Pictus minimodule.

Modifying the example is straightforward, as long as you remain within the specifications of the memory model and compiler options used (less than 64k of data, less than 32k of constants). Just add your own files and functions and rebuild (if you add a new file, don't forget to add it also in the list of objects to be linked, inside the .lkf file).

Porting the example to another chip is not very difficut under the following conditions:

- you port to another VLE only chip, that has no MMU and requires no specific hardware initialization
- you know the chip you are porting to well enough to change some peripheral addresses if needed. For example, the instruction
  SIU.PCR[SWT1] = 0x100;
  requires SWT1 to be equal to 48 for Pictus, equal to 64 for Bolero and probably equal to other values for other chips (possibly even inside the same family) depending on how the IOs are mapped.

Under these conditions, the main things you have to look for when porting to another chip are:

- include the IO file that is specific to that chip (make sure you do not forget the +rev option if the .h file is not provided by Cosmic)

- adjust any hardware related address that is in the source (see the example above, but also consider the interrupt vector table and any specific hardware initialization that might be required)
- adjust the linker file according to the RAM and FLASH sizes of the new chip: the main parameters to modify are shown below (please read the App Note "PPC_Compiler_Options" for a full explanation)

The main parameters you need to change in the linker file (.lkf) when you change chip are:

```
+seg .vtext -a const -m 0x100000 -n vtext -it      # program follow constants
```

modify the constant after "-m" so that it is equal to the flash size – 32kb

```
+def __stack=0x40010000                            # stack pointer initial value
```

modify the constant so that it is equal to the RAM highest address

Using external debuggers or running stand alone

Let's consider first the case where your application is running ok with the debugger, but not standalone: considering that the debugger does initialize some hardware for its own needs, and based on our experience, two things to check are:

- that the bam is correct for the chip you are using (some chips require the RCHW at address 0, some other at address 2).
- that the startup you are using does initialize the whole RAM before using it, so that ECC does not generate any error (all the Cosmic-provided startup file since v4.2.6 do this, but there might be some older versions / debug versions around)

Let's now consider what happens when you want to debug with an external debugger: first of all you must make sure that you generate the ELF file (this is not necessary for ZAP) by using the "cvdwarf" converter provided with the compiler (there is an option in the builder section of IDEA to launch this converter automatically at every build).
Once you have a .elf file, with most debuggers you just load it and start debugging, but, in some cases, the external debugger might complain that it does not find the source files. The problem is similar to what has been explained for ZAP in the previous chapter: if the files were compiled with no absolute path (example: `cxppc +debug +rev +modv main .c` as opposed to

cxppc +debug +rev +modv "c:\examples\main .c") which is the defaut for IDEA if you put your sources in the working directory, the debugger will only know the file by its name (no path) and therefore will only find them if they are in a list of known directories (typically the same directory as the .elf file). If they are not, you must add a new directory to the list, in a way that is specific to every debugger.

The obvious solution to this would be to always compile by giving the compiler full paths: although this has some drawbacks (see the section about Sharing Projects), if you want to work in this way you have two options:

- call the compiler from the command line: create a batch file that calls the compiler (you can use the file IDEABLD.BAT, that is created automatically by IDEA at every build, as an example) and make sure you pass full names to it
- make sure the IDEA option Customize->Force Absolute Names is checked before adding new files to a project: in this way the new files will be always compiled with their full names

Sharing projects

When you share projects with other people, you can share the source, the objects (including the final executable) or a mix of the two (you may provide some libraries in object only form that the Customer will link with their application).

Whatever you are sharing, if the directory structures are the same (the projet base directory, but also the compiler installation directory and other external directories that might be used) there should be no problem at all.

If the directories are different, several problems may appear. Let's consider a typical case where you send a project to a colleague and he/she will try to put it in a different directory. First when the receiving party will open the IDEA project, IDEA will not find most of the files and therefore it will show them with a cross. To solve this changing the IDEA working directory should be all that is needed. Once this is done rebuilding the whole project should allow you to start working.

Let's now consider the case where you want to share executables. If your source files were compiled with full path names (see the previous section) there's probably no way that the receiving party can debug the resulting file. You should therefore use relative names (although this implies specifying search paths to the debugger): if your building environment (IDEA, batch file, make file) does not provide this feature, or its use is inconvenient, you can use the “-pxp” compiler option to strip all the path information from the file name in the debug information.

Compiler versions

It is often important to make sure you are using the same compiler version as your Customers or Colleagues. If you want to check which compiler version you are using open the file version.txt in the compiler installation directory: it contains, in text form, the version numbers for the compiler and all utilities that come with it. Remember that compiler version, IDEA version and ZAP version are not related to each other.

If you installed a patch, you must execute version.bat in order to re-generate the file above with the latest information.

Finally, if you receive an objet file (or an executable) it is possible to know which version of the compiler generated it, and with what parameters (command line): look for the “.info” section in the user manual of the compiler to see how this works.