AN4738
Application note

Getting started tutorial for SPC560P50xx

Introduction

The aim of the getting started tutorial is to introduce the hardware and software tools delivered in SPC560PKIT144S starter kit for SPC560P50xx microcontrollers and to show how to use them. The information available in this tutorial, with the due slightly modifications, can be applied to all SPC560Pxx family.

The tutorial is based on two examples, the first one running from internal random access memory (RAM) and the second one running from internal Flash memory. The tutorial covers the first steps of creating a project in Green Hills® development environment, of writing the code of the application and the building process of the project. Then it describes the first steps on how to setup the hardware environment and how to debug the application.

With this tutorial the user is able to create his own functional examples and to prepare to move to more complex programs using wider range of peripherals of the microcontroller.

Contents

List of tables

List of figures

1 Prerequisites

This tutorial is based on the use of different hardware and software tools that need to be installed and used with their relative license.

Table 1. Prerequisites table

Item	Tool	Version	Part of starter kit CD	Description
1	SPC56XX EVB Motherboard	-	Yes	Motherboard for SPC56XX processor families
2	SPC560P 144LQFP Mini-module	-	Yes	Mini-Module for SPC560P50xx (144LQFP)
3	GHS MULTI IDE	5.0.3	No	Green Hills Software integrated development environment (IDE)
4	GHS scripts for SPC560P50xx processor	-	Yes	Scripts adding targets for SPC560P in GHS project manager
5	P&E ICDPPCNEXUS code debugger	1.16	Yes	64K Starter edition of Power Architecture® Nexus debug environment supporting SPC560P50xx devices (debugger comes with P&E's PROGPPCNEXUS flash programmer)
6	P&E USB Multilink JTAG pod	-	Yes	JTAG debug cable for USB connection
7	SPC560P50xx header files	1.0	Yes	Header files for SPC560P50xx processor registers (jdp_0100.h)

2 Glossary / Terminology

Table 2. Abbreviations and acroynms

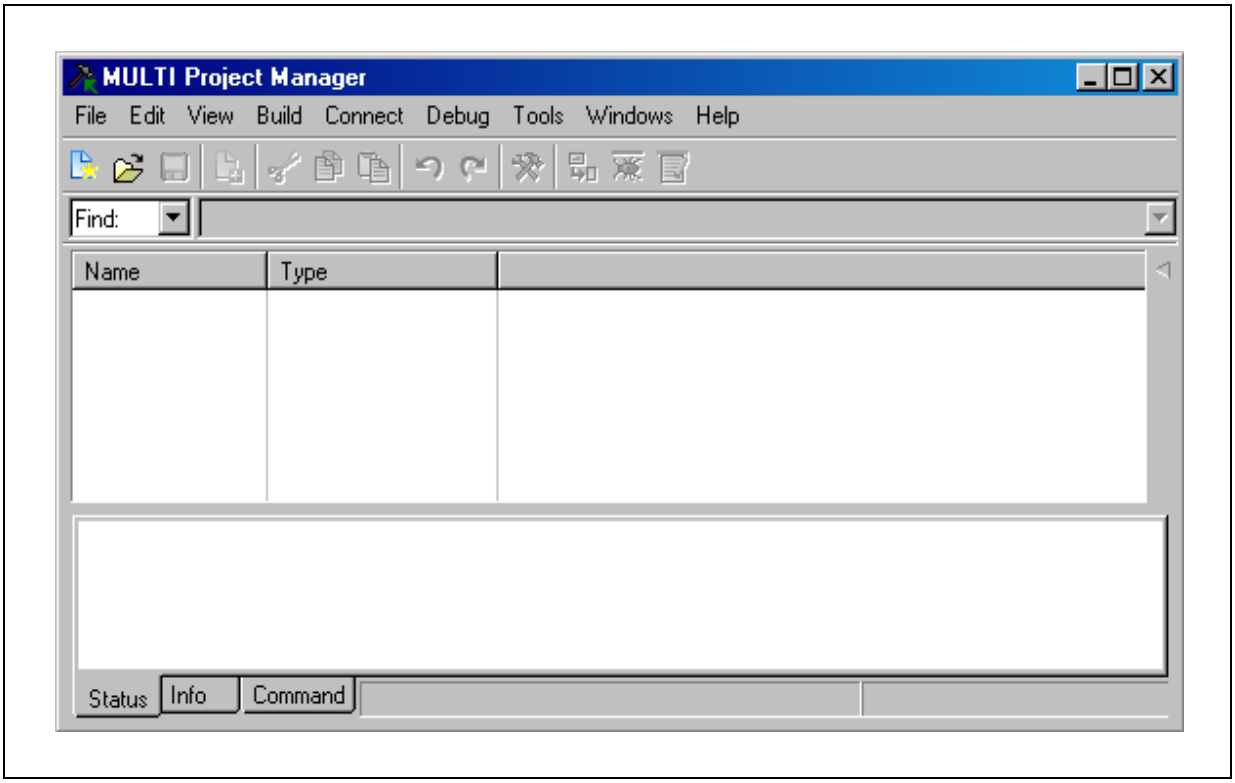
Abbreviation or acronym	Description / Definition
EVB	Evaluation board
GHS	Green Hills® Software
JTAG	Joint Test Action Group
LED	Light emitting diode
LQFP	Low-profile quad flat package
Nexus	Embedded processor debug interface standard (IEEE-ISTO 5001-2003)
P&E	P&E Microcomputer Systems
PIT	Periodic interrupt timer
RAM	Random access memory
SRAM	Static random access memory
USB	Universal serial bus

3 Creating a project

This section describes how to create a new project in the Green Hills environment. The project includes folder structure and all needed files. It is the starting point for both examples of getting started tutorial.

Run the multi.exe to open the Green Hills environment. The multi.exe file is available in the start menu under the Green Hills Software (GHS) branch or directly under the GHS installation directory. It opens the MULTI® Project Manager as shown in *Figure 1*. The MULTI Project Manager manages the project collections and settings and allows compiling and building operations.

Figure 1. MULTI Project Manager



3.1 Defining a project

The MULTI Project Manager tool is used to create a new project for the application. Select FileNew Top Project to make sure that a popup window “Project Wizard” containing an initial project setup appears, helping the user in creating a new project (see *Figure 2*).

The Project Wizard creates a project in three steps:

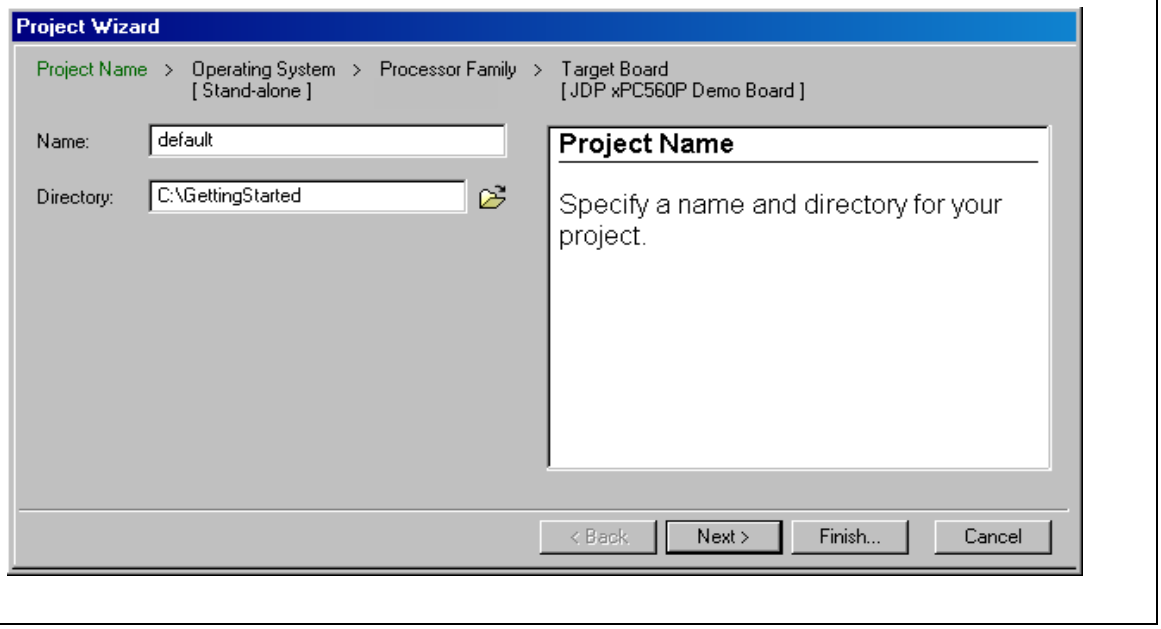
1. Project name: specify the name and directory for project
2. Operating system: selects the operating system the project runs on
3. Processor family or target board
   a) Processor family: it can select the processor family want to use.
   b) Target board: it can select the target board want to use.

Step 1 - Project name

Figure 2 shows the window in which the user can select the project name, the folder name and its path where the project configuration has to be stored.

In this case you select directory “**C:\GettingStarted**”, but it is up to the user to decide where project files should be placed. Refer to this directory as project default directory.

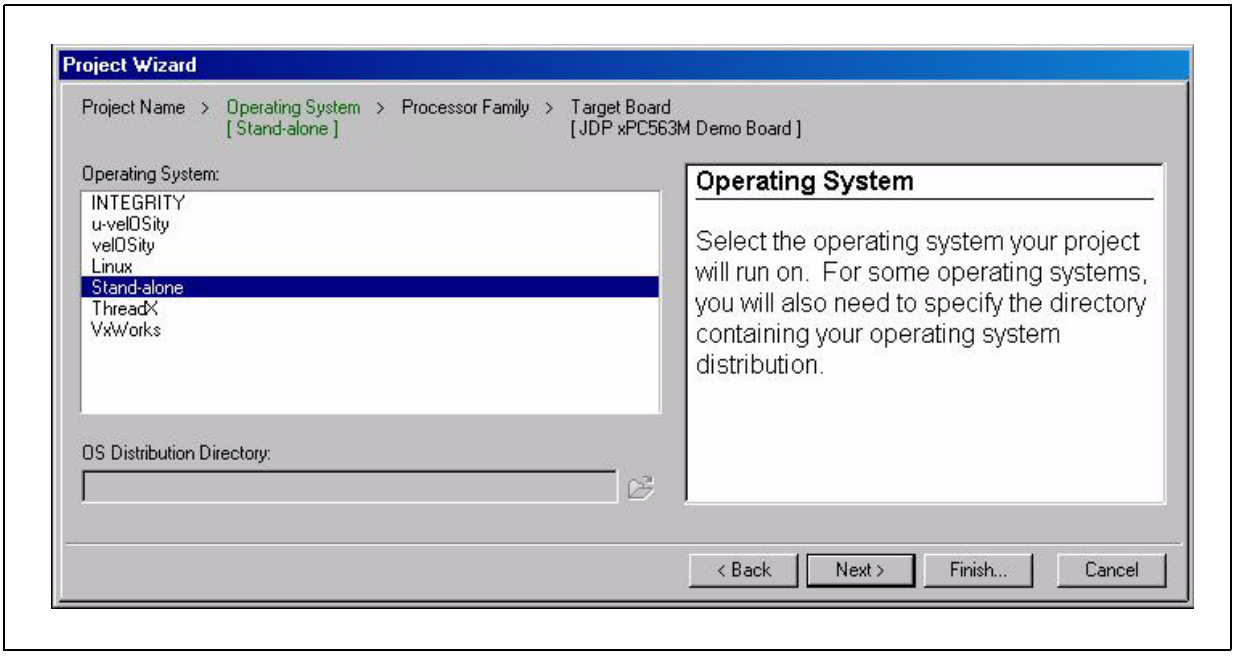
Figure 2. Project wizard – name selection



Step 2 - Operating system

This step, illustrated in *Figure 3*, offers options for the implementation of an operating system. In this case select **Stand-alone**, which means that the application does not need any operating system.

Figure 3. Project Wizard - operating system selection

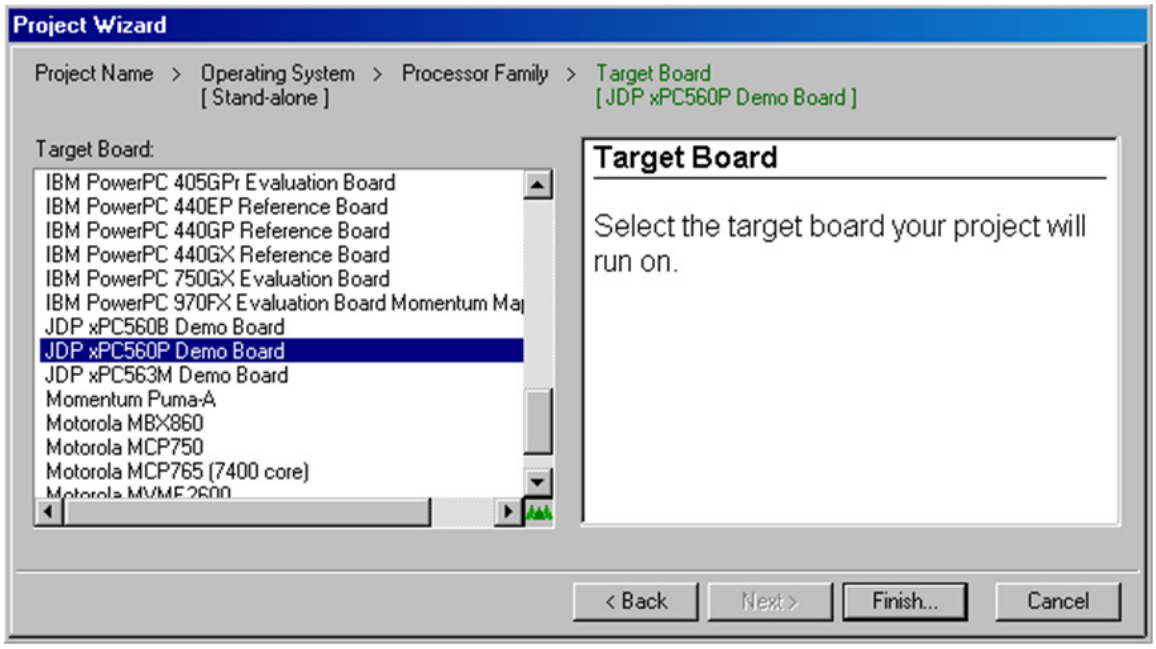


Step 3 - Processor family or target board

During this step the user can select the processor family or the target board to be used. According to the selection, Project Wizard configures parameters and selects appropriate files to include in the project structure. The Target Board list in *Figure 4* contains the large number of evaluation boards already supported.

Select **JDP SPC560P Demo Board** to keep alignment with the hardware. If the SPC560P *Demo Board* option is missing in the menu, please check **GHS\multi503\target\ppc** directory where it should be available directories called SPC560B, SPC560P and SPC563M. If not, copy these directories from the starter kit CD into the specified path. Then after restarting the MULTI environment you should see them in the target board list.

Figure 4. Project Wizard - processor and hardware selection



3.2 Defining a program content

At completion of the first phase of creating a new project, MULTI has created a project skeleton with predefined settings and target-specific information. Now the Project Wizard continues with the definition of the project content where it is needed to set the project type and desired build scheme.

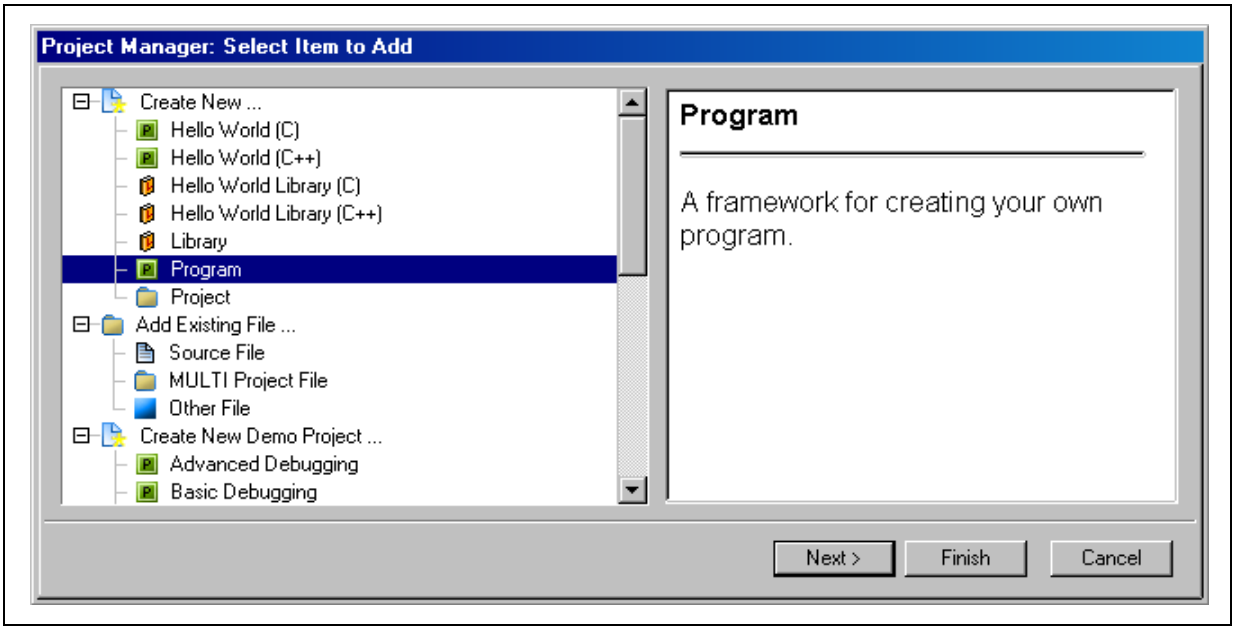
Project manager consists of three steps:

1. Selection of project type
2. Name and directory setting
3. Program layout

Step 1 - Selection of project type

First, specify the project type (see *Figure 5*). The option “Program”, blue highlighted in the *Figure 5*, allows to compile and build a particular application and provide its source files later on.

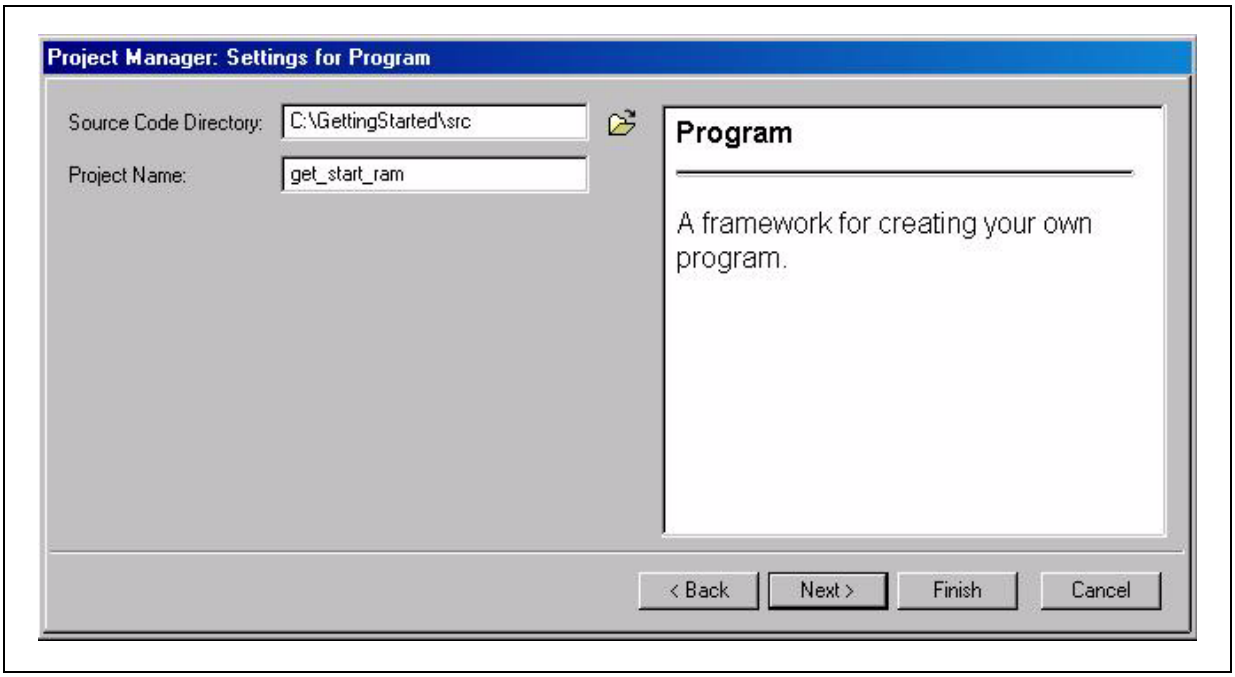
Figure 5. Project Manager - selection of project type



Step 2 - Name and directory setting

This step is used to define the source files location and the program name (see *Figure 6*). We leave the default setting for source file destination and set the name of the project to **get_start_ram**.

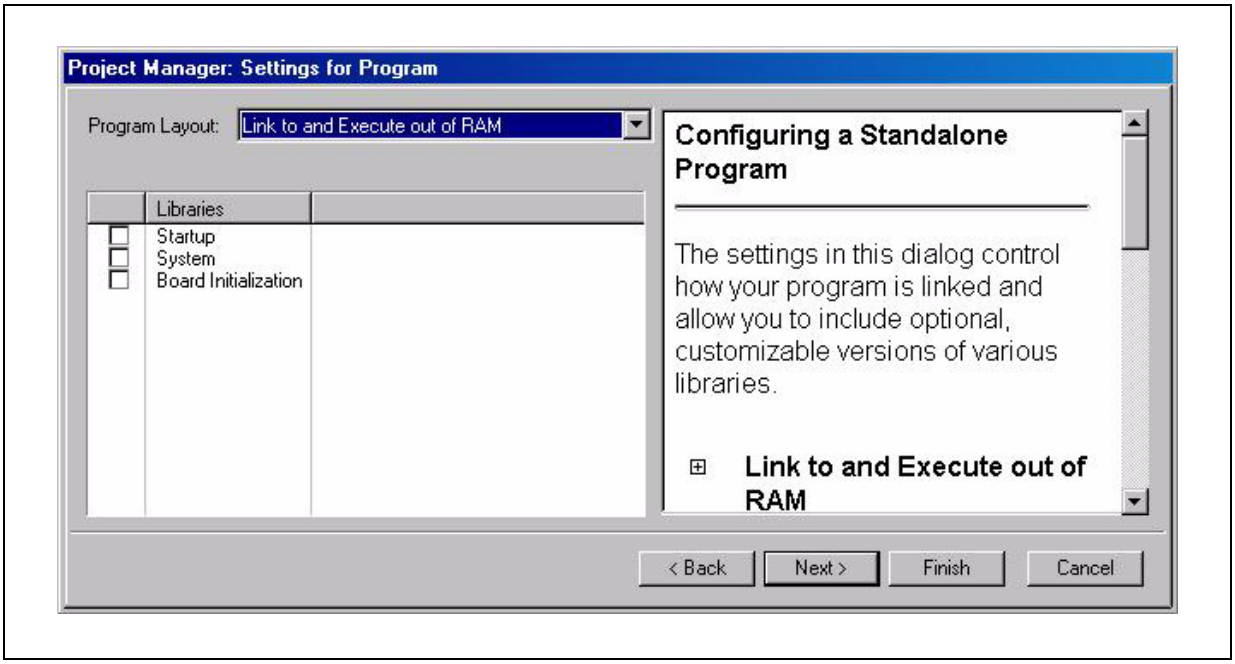
Figure 6. Project Manager - name and directory setting



Step 3 - Program layout

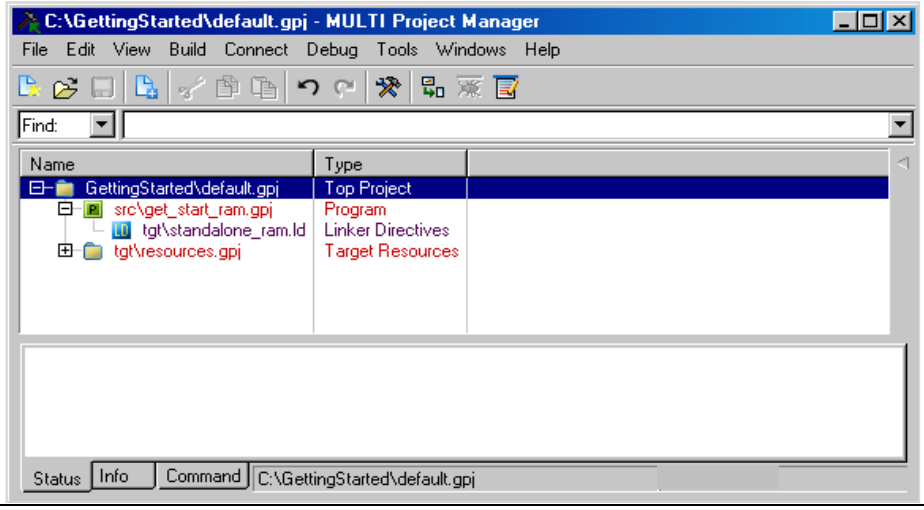
The configuration window, shown on *Figure 7*, allows to select the desired program layout. Here, for the first example of getting started tutorial, choose **Link to and Execute out of RAM** without checking any of the three options that can be seen on the left side of the *Figure 7* . Select layout enables to use the simple JTAG debugging environment without any Flash programming functionality.

Figure 7. Program Manager - program layout



When all previous steps are finished, the Project Wizard generates a skeleton of the project and fills it with appropriate files based on the previous selections. The final project skeleton should look like the one in the *Figure 8*.

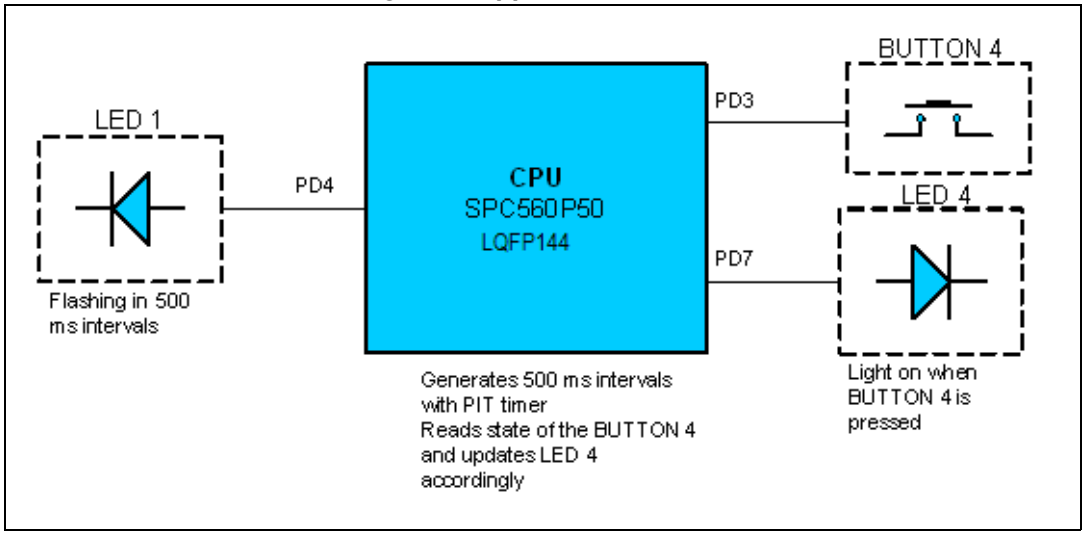
Figure 8. Final project structure



4 Creating an application

This section describes an application code of the tutorial example whose block diagram of the functionalities is shown in *Figure 9*. Each function of the application is considered. The whole code of the application is listed in *Appendix A: appTop.c*.

Figure 9. Application scheme



The built-in editor of the GHS MULTI Project Manager allows to edit an application code. It is possible to open the editor selecting the **Tools Editor** item in the toolbox menu. Subsequently, an open file dialog appears, where you can select the file to edit or to create. In the example presented here, a new file named **appTop.c** has been created **(**please refer to *Section Appendix A*). Once confirmed, an editor is opened with a blank sheet. Of course any editor can be used for creating the source code. Application functionality is controlled by **main()** function which calls event handlers in an endless loop. Before entering in the endless loop, the function **Init()** is run.

```
void main(void)
{
    // init peripherals
    Init();

    // infinitive loop
    // - call TIMER0 event handler
    // - call BUTTON4 event handler
    while (1)
    {
        TIMER0_Event();
        BUTTON4_Event();
    }
}
```

The initialization of processor peripherals and modules is covered by the **Init()** function. It configures the following steps:

- Prepare configuration for Periodic Interrupt Timer (PIT) peripheral in RUN0 mode
- Switch operating mode from DRUN to RUN0 to enable used PIT peripheral
- Configure and start PIT peripheral
- Configure general IO pins used for LEDs and button
- Disable software watchdog module

```
void Init(void)
{
    // mode & peripheral enable
    ME.RUNPC[1].B.RUN0 = 1;       // prepare configuration word
    ME.PCTL[92].B.RUN_CFG = 1;    // and use it for PIT enable in
RUN0 mode

    // mode transition request
    ME.MCTL.R = 0x40005AF0;
    ME.MCTL.R = 0x4000A50F;
    while (ME.GS.B.S_CURRENTMODE != 4) ;  // wait for DRUN->RUN0
transition

    // PIT timer configuration
    PIT.PITMCR.R = 1;             // enable periph. & stop timer
during debug
    PIT.CH[0].LDVAL.R = 0x7A1200; // reload value
    PIT.CH[0].TFLG.B.TIF = 1;     // clear potential flag
    PIT.CH[0].TCTRL.B.TEN = 1;    // enable timer

    // GPIO configuration
    SIU.PCR[52].R = 0x0200;       // enable output for LED1
    SIU.PCR[51].R = 0x0100;       // enable input for BUTTON4
    SIU.PCR[55].R = 0x0200;       // enable output for LED4

// disable SWT
SWT.SR.B.WSC = 0xC520; // clear soft lock sequence
SWT.SR.B.WSC = 0xD928;
SWT.CR.B.
N = 0; // disable watchdog}
```

All peripherals are accessible through symbolic names which are defined in the predefined header file. You need to include the **jdp_0100.h** file in the source file. It is recommended to keep the files in one place, as the header file represents the top of header file structure for SPC560P50xx devices and includes another header file inside. For that reason it is recommended to copy **jdp_0100.h** and **typedefs.h** header files from the starter kit CD to **spcHeaders** directory created in the project default directory **C:\GettingStarted\spcHeaders**

```
#include "..\spcHeaders\jdp_0100.h"
```

The example code contains two handlers which are called periodically from main() function.

The first handler, **TIMER0_Event()**, asserts periodic events generated by the PIT timer. Each time it finds the timer flag set, it toggles LED1 and clears the flag.

The second handler, **BUTTON4_Event()**, polls status of the button and lights LED4 on when button is pressed. The following code snippets show handler implementation.

```
void TIMER0_Event(void)
{
    // wait for timer flag signalling expiration
    // then toggle with pin and clear flag by writing '1'
    if (PIT.CH[0].TFLG.B.TIF)
    {
        SIU.GPDO[52].B.PDO = ~SIU.GPDO[52].B.PDO;
        PIT.CH[0].TFLG.B.TIF = 1;
    }
}
void BUTTON4_Event(void)
{
    if (SIU.GPDI[51].B.PDI == 0)
        SIU.GPDO[55].B.PDO = 0;
    else
        SIU.GPDO[55].B.PDO = 1;
}
```

The last extract of code, shown in the following, is a **__ghs_board_memory_init()** function, which represents a specific functionality that has to be executed after power on when running from Flash memory. It initializes the SRAM and its ECC module because after power on, there are random values that can generate system events when not properly initialized. As seen from the code, RAM initialization is done by means of assembly instructions. It can be noticed, there is a preprocessor macro called **BUILD_TO_FLASH** providing an option to remove the function when building the image to the RAM when it is undesired, because it would overwrite the code loaded to the RAM. In this case, memory initialization is provided by a debugger script before loading the program to the RAM. In the first example the get_start_ram macro BUILD_TO_FLASH is not defined, therefore the function is not present in the image. You use it when you build the image to Flash memory in the second example shown in this tutorial.

```
#ifdef BUILD_TO_FLASH

void __ghs_board_memory_init(void)
{
    // RAM memory initialization
    #pragma asm
        e_lis  r6, 0x4000
        e_or2i r6, 0x0000
        e_lis  r7, 0x4000
        e_or2i r7, 0x9FFF

    init_ram_loop:
        e_stmw  r0,0(r6)
        e_addi  r6,r6,128
        se_cmp  r6,r7
        e_blt   cr0,init_ram_loop

    #pragma endasm
}
#endif //BUILD_TO_FLASH
```

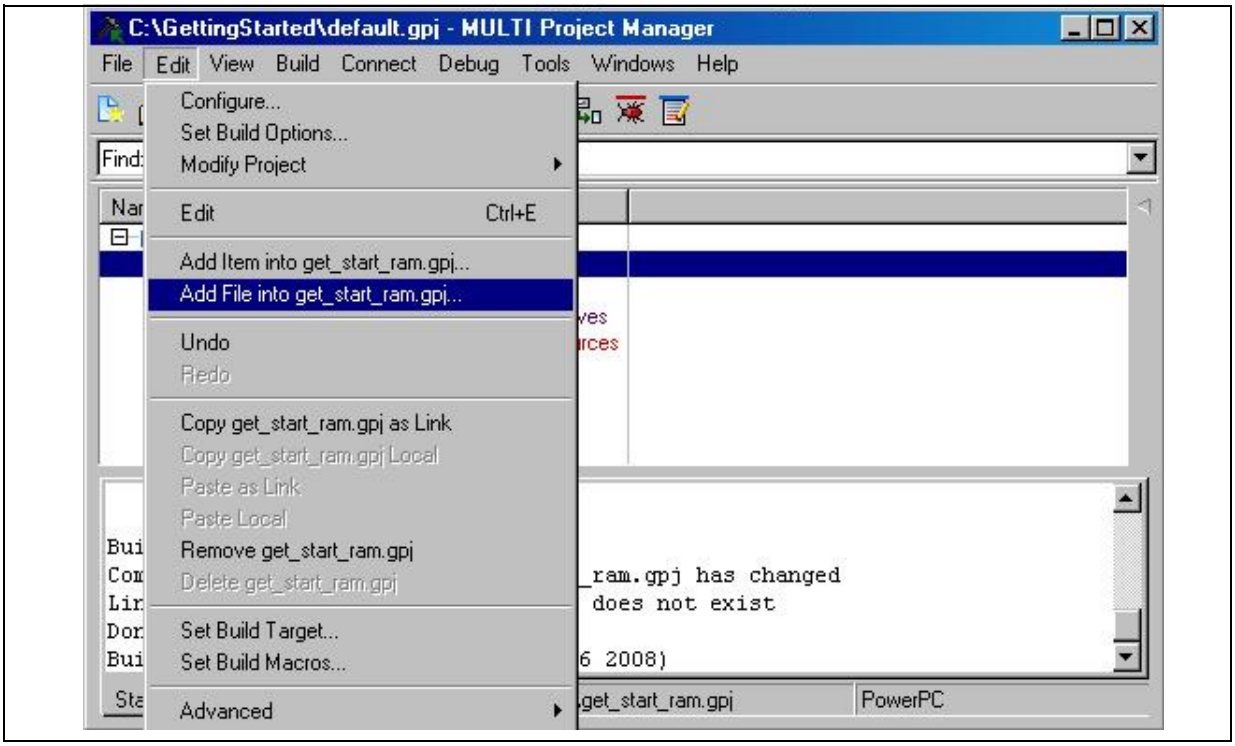
The previous pages described the application source code. The whole code is listed in the *Appendix A: appTop.c*. Next step is to add the source file to the project, compile it and build an image that can be downloaded to the processor.

5 Building the project

5.1 Adding new files to the project

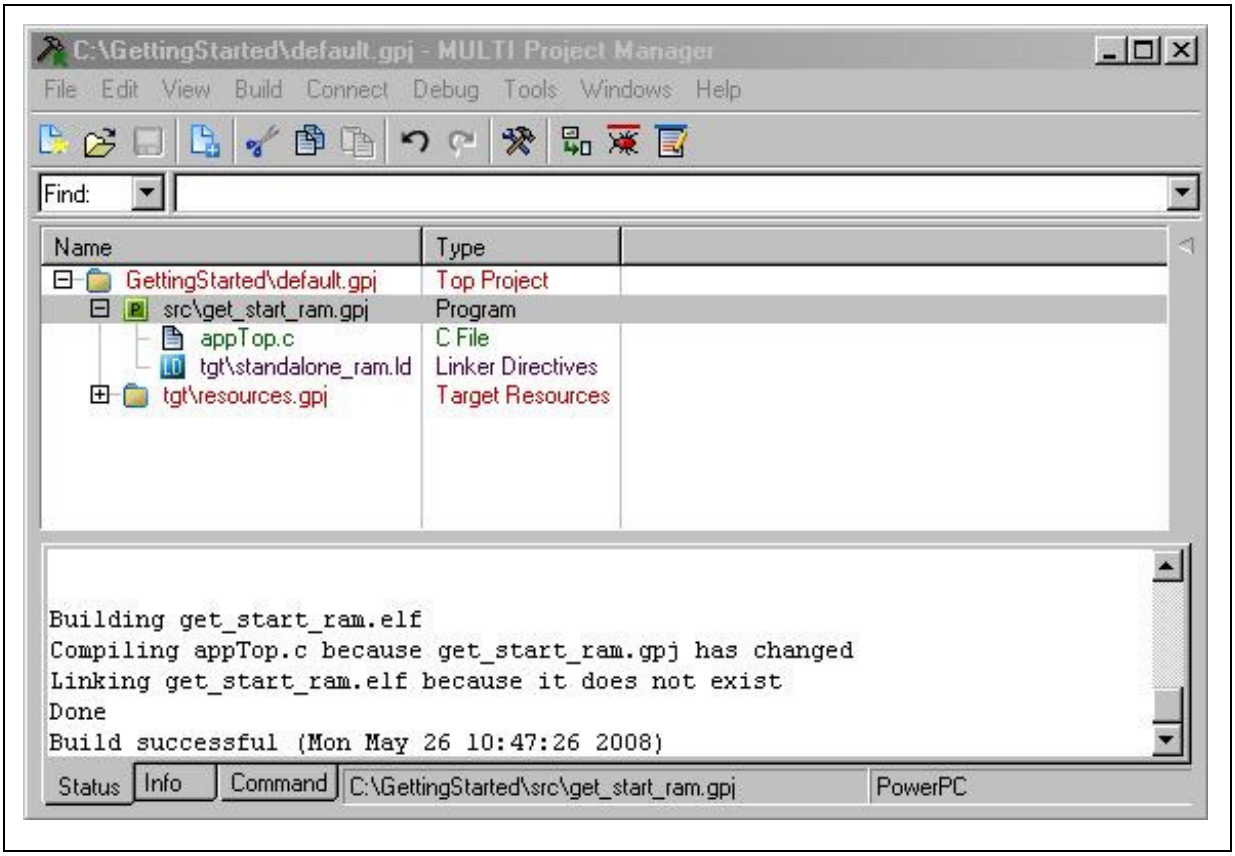
The source file appTop.c has to be added into the project. Select src\get_Start_ram.gpj in the project manager and then click on **EditAdd File into get_start_ram.gpj** item (see *Figure 10*).

Figure 10. Building a project – adding source file



A dialog box appears where you find and select **appTop.c** file. When the file is added, you should see a project structure like the one shown on the *Figure 11*.

Figure 11. Building a project – structure after adding source file

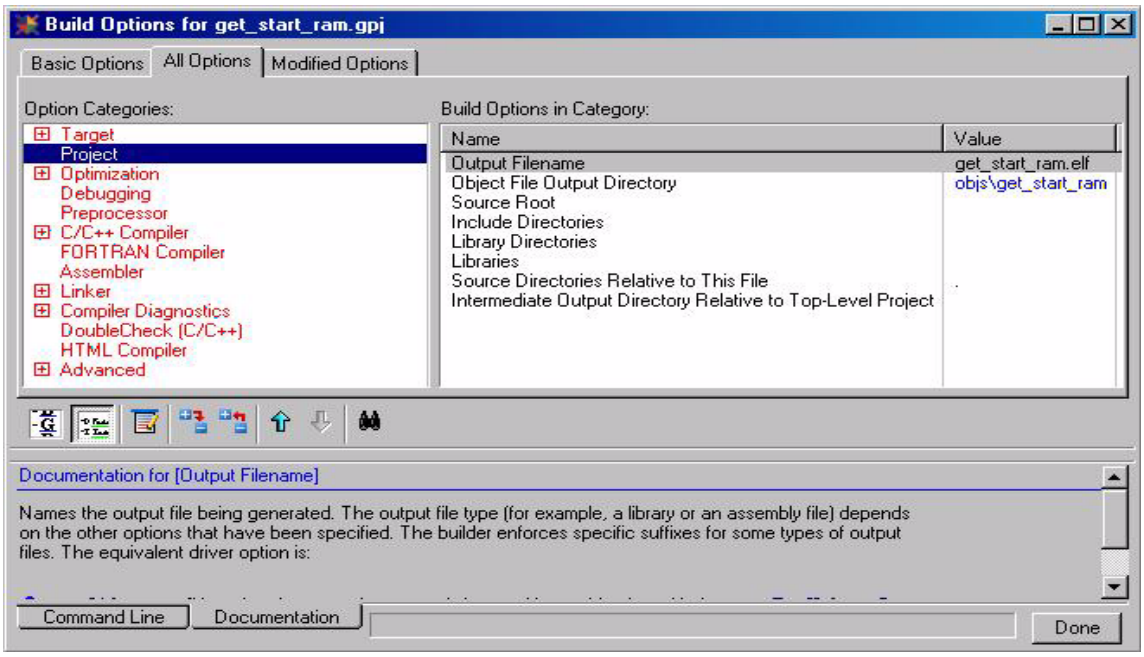


Once the source file is inside the project, compile it. But before performing compile operation, it is recommended to check and eventually set some build parameters.

5.2 Build options

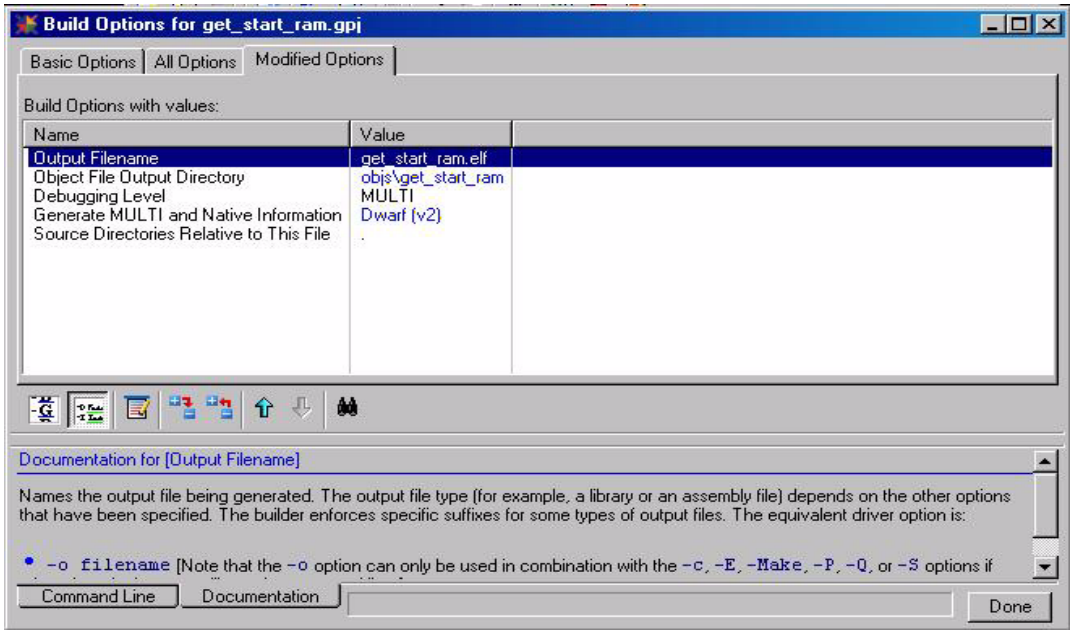
To open the build setup dialog, keep **src\get_start_ram.gpj** line selected and choose **EditSet Build Options**. A dialog window appears with three tabs. Keep the All Options tab selected (see *Figure 12*).

Figure 12. Building a project – build options



Select “Project” from the listed Option Categories. Change the name of the build result. Double click on **Output Filename** item, fill in the target name, and click OK. In this case the name is **get_start_ram.elf**. Another option that has to be modified is located under group **AdvancedDebugging OptionsNative Debugging Information**. There is a **Generate MULTI and Native Information** element which has to be set to **Dwarf (v2)**, in order to see mixed source code and assembly in the debugger. These two options are enough for now. Selected and modified options for this project can be checked by selecting the **Modified Options** tab (see *Figure 13*) or by opening a **get_start_ram.gpj** file in an editor program.

Figure 13. Building a project – Modified Options tab



Once all options are set, compile source file. Select appTop.c file in project manager and click on **BuildCompile appTop.c** item. If compilation process doesn't run into problems, build successful information appears in the status window. Otherwise build details window is opened where more details about the problem appear by double clicking on the error or warning message. Output files of compilation process are put under **objs\get_start_ram** directory.

5.3 Linker directive file

Before starting a build operation, you check the configuration of linker file used by linker to determine memory layout. So, double click on the **tgt\standalone_ram.ld** file in the project manager window to open the file in the editor.

Basic linker file has three sections:

1. Memory definition
2. Constant definition
3. Partitioning of code and data segments

Memory definition tells the linker which memories are present on the hardware, their start addresses and size. If needed it is necessary to align it with used hardware. For the SPC560P50xx microcontroller it should look like the following line codes, 512 KB of Flash memory and 40 KB of internal RAM.

```
MEMORY {
//  512K Internal Flash
    flash_rsvd1   : ORIGIN = 0x00000000, LENGTH = 0
    flash_reset   : ORIGIN = .,    LENGTH = 8
```

```
    flash_rsvd2    : ORIGIN = .,     LENGTH = 0
    flash_memory : ORIGIN = .,           LENGTH = 512K-8
    flash_rsvd3   : ORIGIN = .,          LENGTH = 0

//  40KB of internal SRAM starting at 0x40000000
    dram_rsvd1    : ORIGIN = 0x40000000, LENGTH = 0
    dram_memory   : ORIGIN = .,     LENGTH = 40K
}
```

Next section represents a definition of the useful constant. By default it defines the size of stack and heap regions.

```
DEFAULTS {
    stack_reserve = 4K
    heap_reserve = 2K
}
```

The last section of the linker file is the partitioning of code and data segments in memory. The only things to check are target memories for code segments (.text, etc.) and data (.bss, .data, etc.). To build everything to RAM, all partitions should be placed to RAM memories, **dram_memory** in this example.

```
SECTIONS
{
    .PPC.EMB.sdata0                      ABS : > dram_memory
    .PPC.EMB.sbss0                             CLEAR ABS : > .

    .text               : > dram_memory
    .vletext      : > .
    .syscall      : > .
    .resetvectorNOCHECKSUM: > .
    .secinfo      : > .
    .rodata      : > .
    .sdata2      : > .
    .fixaddr     : > .
    .fixtype   : > .

    .sdabase                ALIGN(16) : > dram_memory
    .sdata     : > .
    .sbss              : > .
    .data              : > .
    .bss               : > .
    .heap                   ALIGN(16) PAD(heap_reserve)   : > .
    .stack                  ALIGN(16) PAD(stack_reserve) : > .

// These special symbols mark the bounds of RAM and ROM memory.
// They are used by the MULTI debugger.
//
    __ghs_ramstart   = MEMADDR(dram_rsvd1);
    __ghs_ramend     = MEMENDADDR(dram_memory);
    __ghs_romstart   = MEMADDR(flash_rsvd1);
    __ghs_romend     = MEMENDADDR(flash_rsvd2);
}
```

5.4 Compile and build

If the compile operation is completed and the memory layout matches the requirements, try to build the image to debug on simulator or use a real hardware. Select src\get_start_ram.gpj line in the project manager and click on **BuildBuild Program get_start_ram.elf** item. Again status and result of build program operation are in the status window. Any error or warning opens build details window with more pieces of information. Output files of build phase are put under project default directory where default.gpj file is placed. By default result of build operation consists of more files, such as memory map file, etc. The most important file in this case is **get_start_ram.elf**. This file is used in the debugging phase.

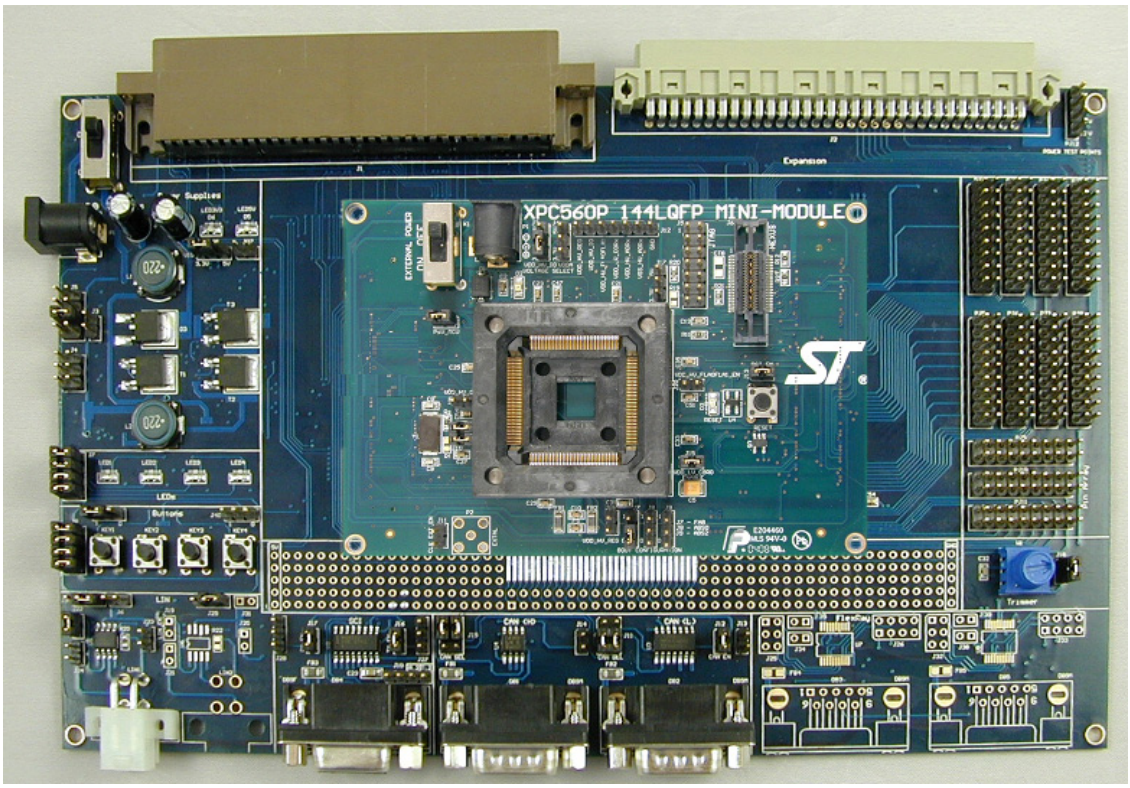
6 Hardware setup

Before going into the real debugging, it is right time to check hardware environment and its setting. Tutorial goes through some important points of the hardware configuration needed by getting started example. SPC560P50xx evaluation board, xPC56XX EVB, consists of two parts of hardware:

- Motherboard (see *Figure 14*)—Motherboard provides common functionality used in most application like serial communication interface, CAN transceivers, power supply, buttons, LEDs and so on.
- Mini-module to provide a minimum setup for the microcontroller, such as socket for the processor, crystal oscillator, debug interface and so on.

Figure 14 provides an overview of the system. In this figure, both the motherboard and the SPC560P50xx mini-module are visible.

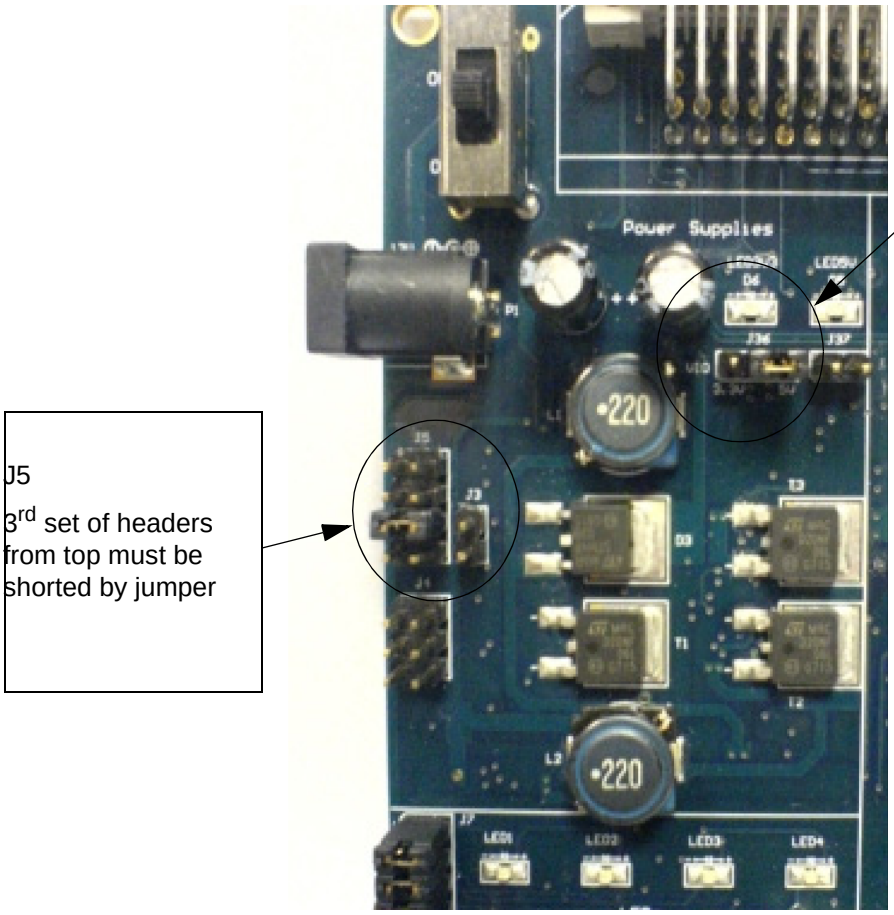
Figure 14. Hardware - SPC560P50xx evaluation board



6.1 Power setup

As first step you can power supply block setting, connection of jumper J5 and VIO as it is showed in *Figure 15*.

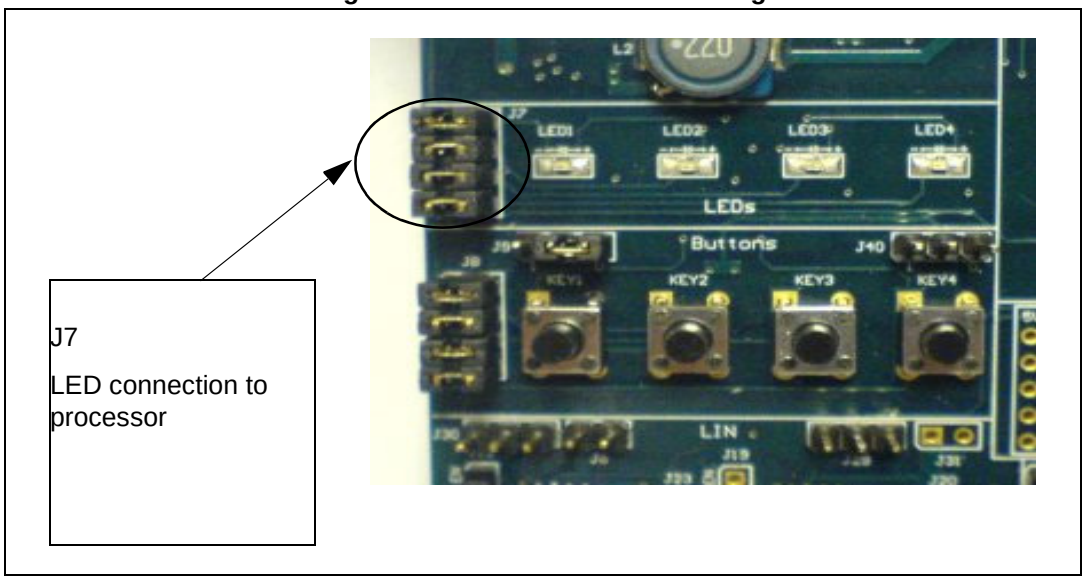
Figure 15. Hardware - power block setting



6.2 LED setup

In the example shown *Figure 16* in LED1 and LED4 are used. To connect them to the processor, headers 1–2 and 7–8 of the J7 jumper array have to be shorted (see *Figure 16*).

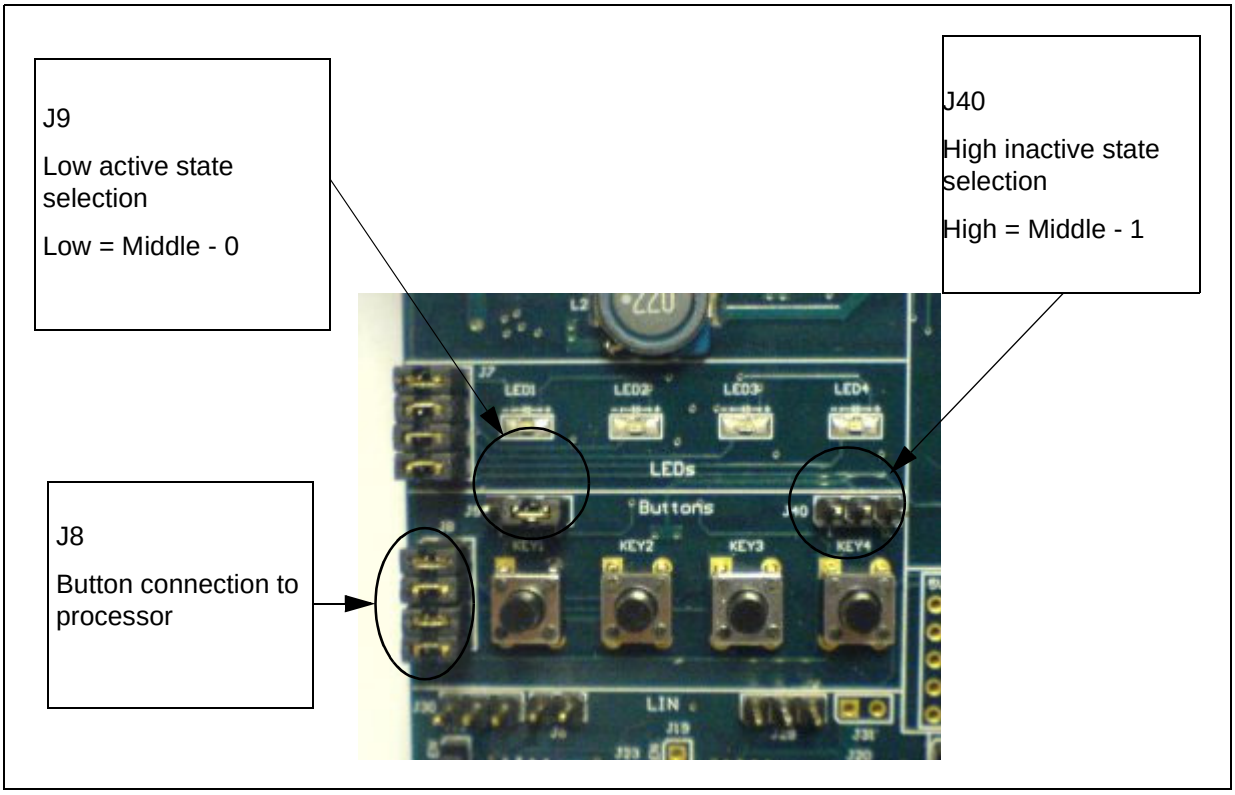
Figure 16. Hardware – LEDs setting



6.3 Button setup

Check the buttons settings on the motherboard. In this example, the active state of button 4 must be set to low active state level. For such purpose header 7–8 of J8 jumper area should be shorted, J9 connection shorted between middle and “0” header and J40 shorted between middle and “1“ header. *Figure 17* shows the right settings.

Figure 17. Hardware – buttons setting



6.4 Mini-module setup

The SPC560P mini-module has connectors and several configuration jumpers. *Figure 18* shows their layout and *Table 3* shows the right settings and functions.

Figure 18. SPC560P50xx mini-module setting

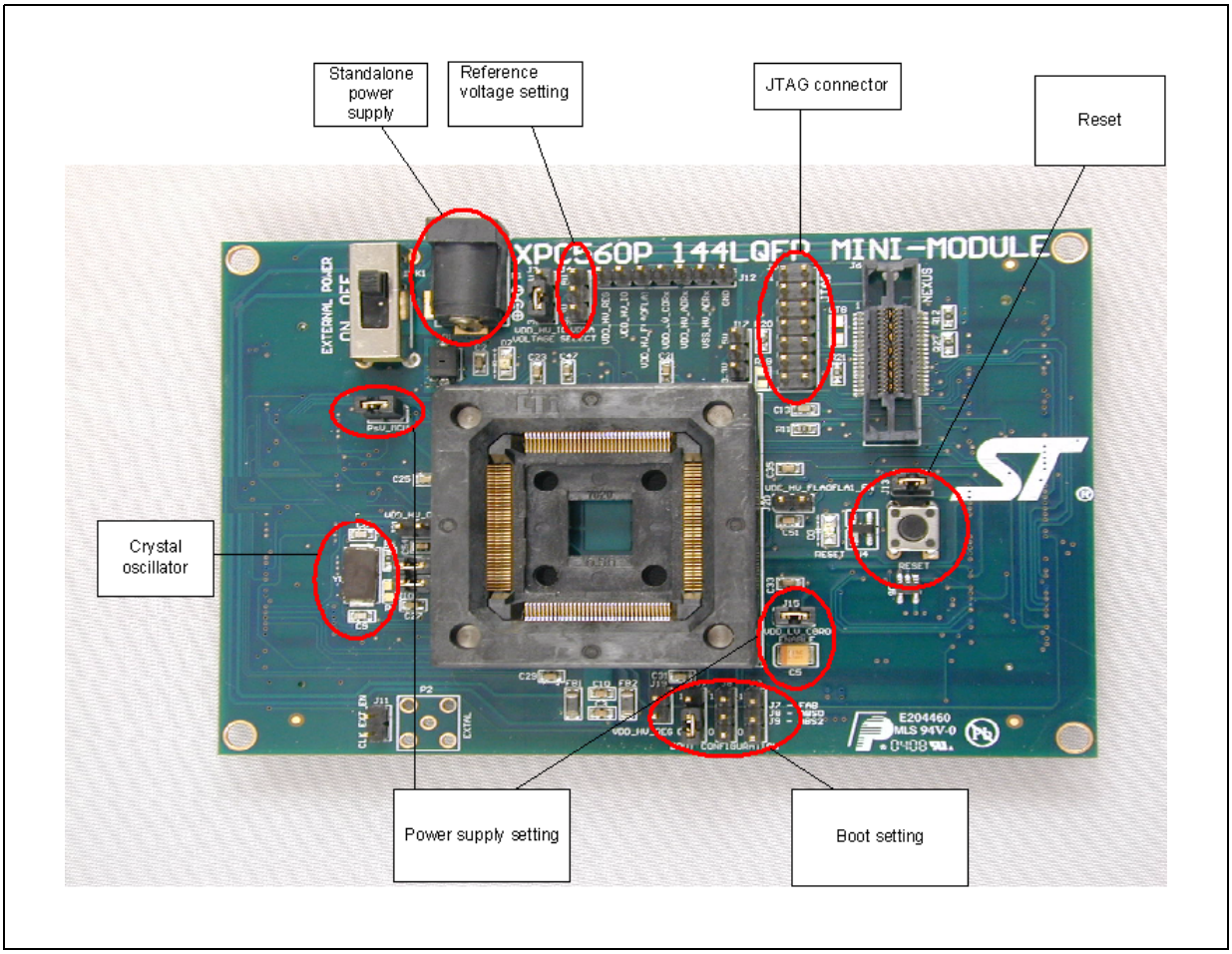


Table 3. SPC560P50xx mini-module setting

Block	Jumper	Setting
Standalone power supply	-	Connector for power supply
Reference voltage setting	J3	5V
	J4	5V
Power supply setting	J14	Connected
	J15	Connected
Boot settings	J7	Connected middle-0
	J8	Connected middle-0
	J9	Connected middle-0
Crystal oscillator	J10	Connected 1–2 and 3–4
Reset	J13	Connected to enable reset circuit
JTAG connector	-	Connector for debugger
V_Debug	J17	Connected middle-3.3V

Table 3. SPC560P50xx mini-module setting (continued)

Block	Jumper	Setting
V_{DD}_HV_REG_0	J19	Closed
V_{DD}_HV_FLA0FLA1_EN	J20	Closed
V_{DD}_HV_OSC0	J21	Closed

Note: For more details on the connection of the evaluation board, please see SPC560PX EVB User Manual v.1.02.pdf.

7 Debugging the application

Once the evaluation board is properly configured, the microcontroller is inside SPC560P 144LQFP minimodule socket and the sw application is successfully built, the debugging process can start.

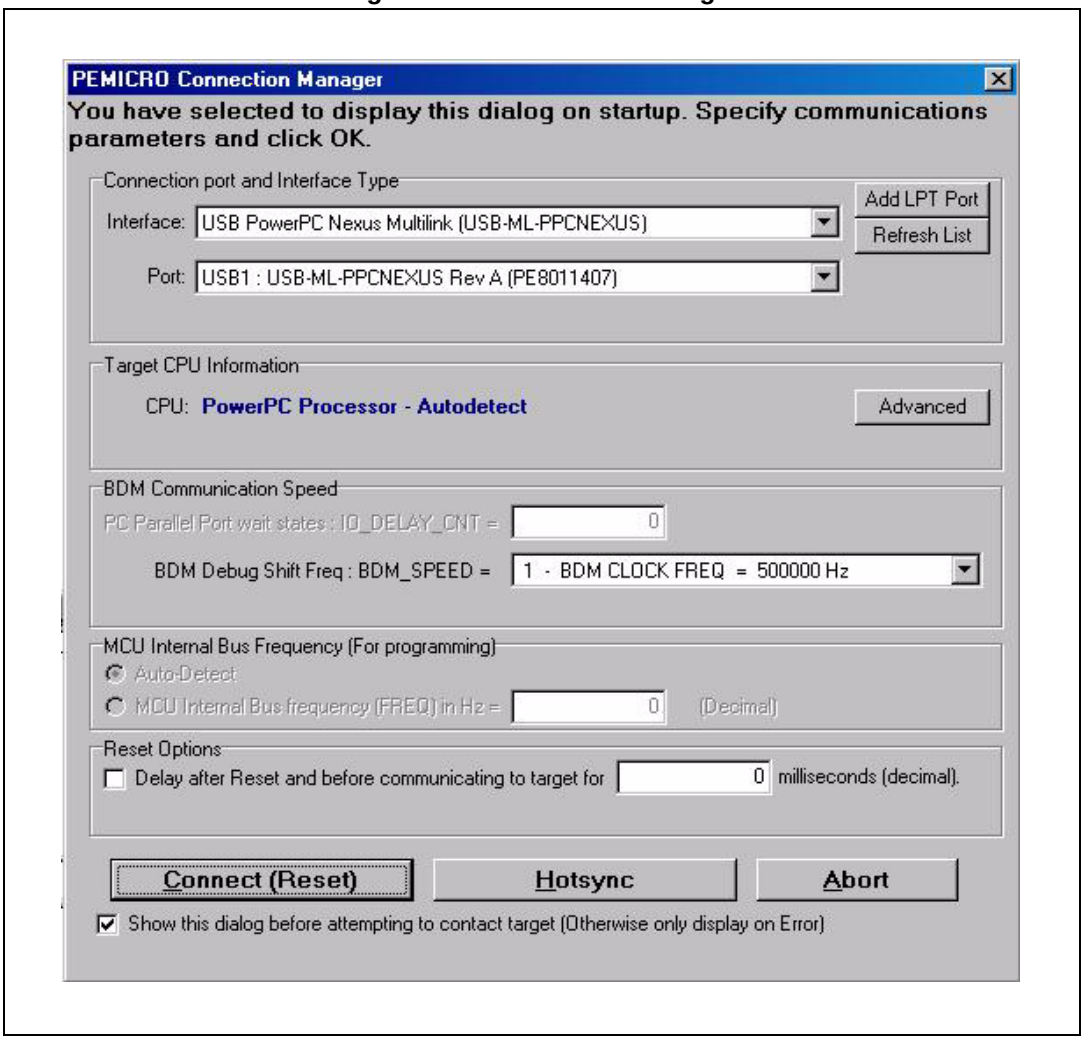
7.1 Debug connection

First, connect a debug tool, in this case P&E's USB multilink, to the JTAG connector placed on the SPC560P50xx mini-module. The red stripe on the cable corresponds to pin 1 on the connector, which is in the top-left position on the header.

Caution: The power supply must be off before connecting debug tools to the board.

When the debug tool is connected, switch the power supply on using switch K1 in the top-left corner on the motherboard. The green LED on the P7E debug tool should be on. Then start ICDPPCNEXUS debugging software on PC. There is a link to executable file in start menu under P7E Nexus starter kit item or directly in P&E micro-install directory. At first connection the manager dialog box opens, where a connection method and its parameters are asked (see *Figure 19*).

Figure 19. Connection manager

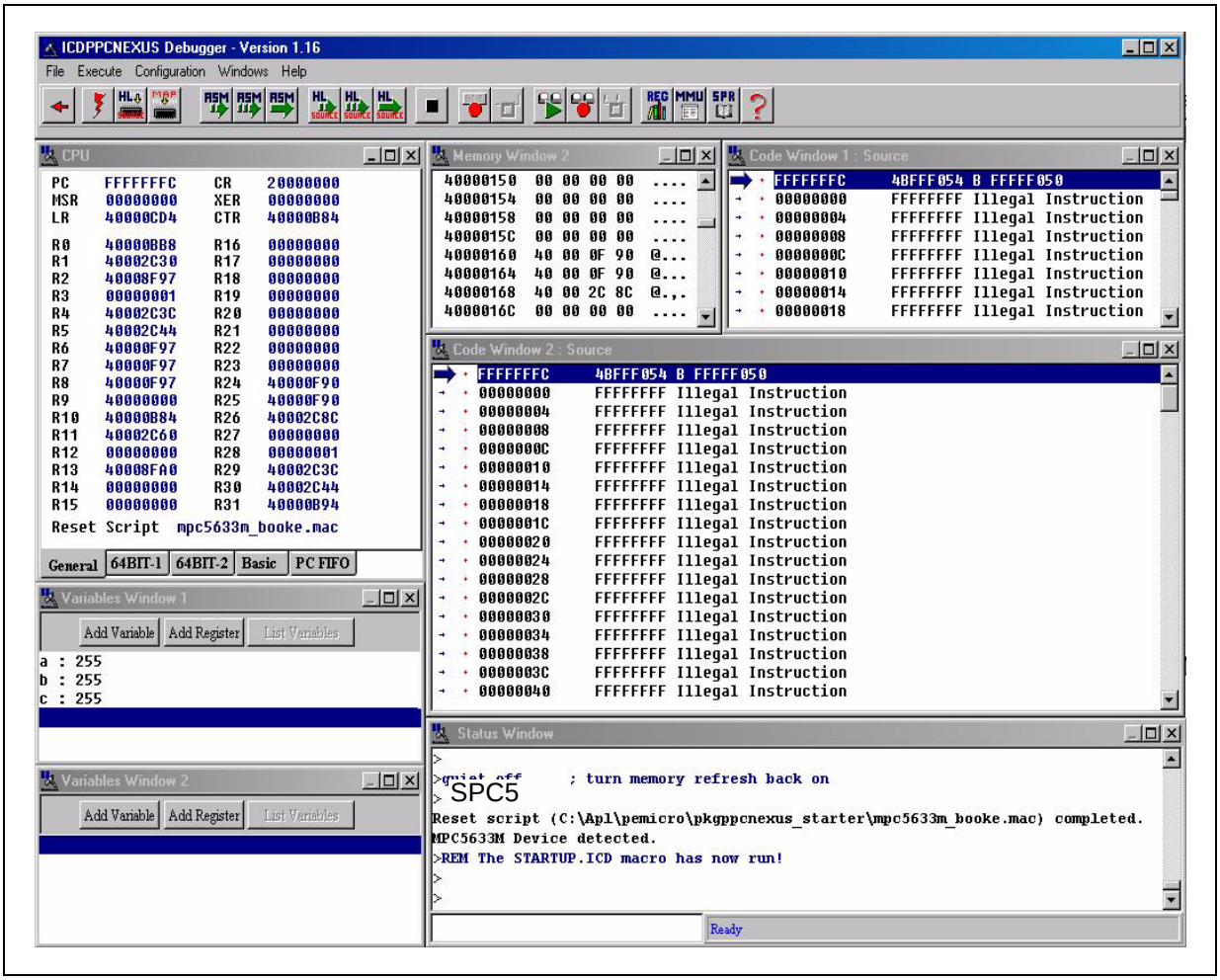


The proper interface and port, in this case the USB PowerPC Nexus multilink, must be specified. The rest of selection boxes remain unchanged. Then you can connect to the microcontroller. Click on connect (reset) button. Debugger tries to enter debug mode in the microcontroller.

7.2 Debug IDE default layout

When the debug mode is established, it executes an initializing configuration default script. It serves to emulate boot assist mode (BAM) sequence, which is skipped when booting in debug mode. Regardless of script success, program starts with default windows layout, see *Figure 20*.

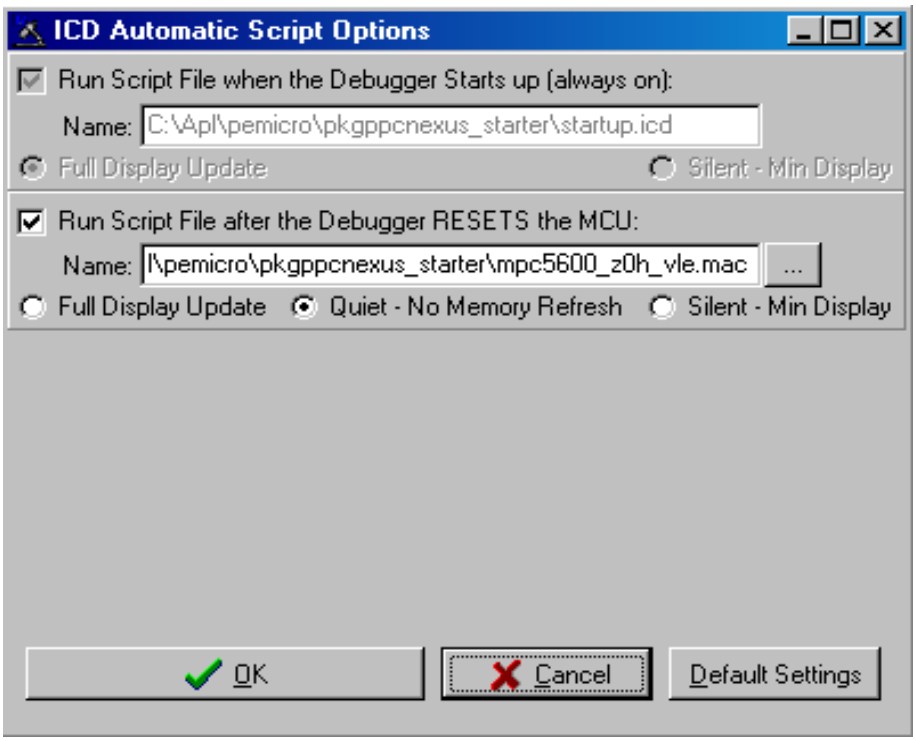
Figure 20. Default layout



7.3 Script file setting

When running the P&E debugger for the first time, it is necessary to check and eventually change the default script file run after reset. Click on **ConfigurationAutomated script options** item to open a dialog box where you can change the initialization script in the box **Run Script after the Debugger RESETS the MCU**, *Figure 21*. Choose **mpc5600_z0h_vle.mac** script file located under installation directory. It should be immediately visible in a file selection dialog box.

Figure 21. Automatic script options dialog box

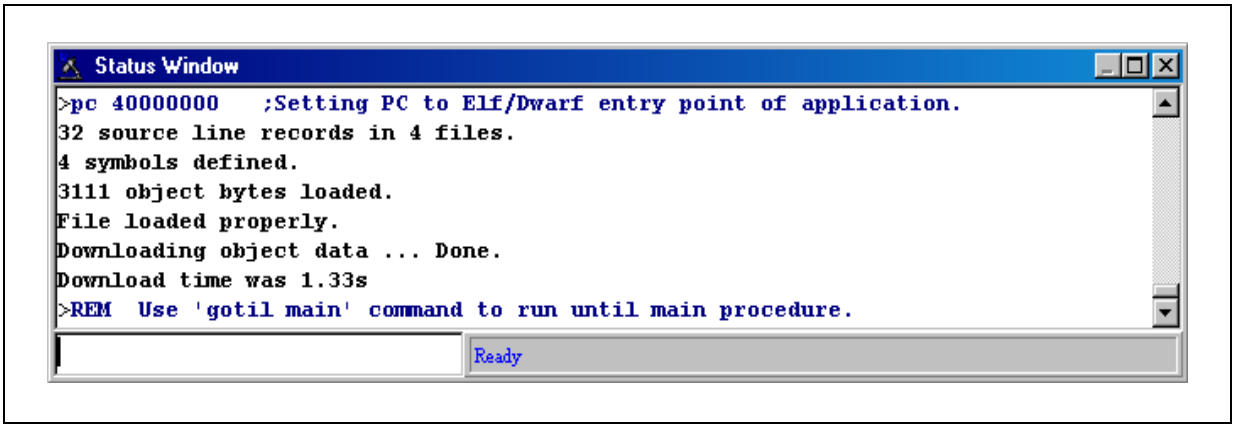


After selection of script file, you run the execution of reset command which is automatically followed by processing of the selected script. Reset command can be issued from menu ExecuteReset Processor or by clicking on reset button on the toolbar. You can verify the functional debug state by changing the content of one byte in the internal memory area. Click with right button of the mouse in the memory window 2 area, select Set Base Address item from popup menu and fill it with a value 0x40000000. Window should be refreshed with the memory content of that area. Now click on whatever byte inside memory window 2 area and change the value. Data byte should change the value accordingly.

7.4 Loading application to RAM

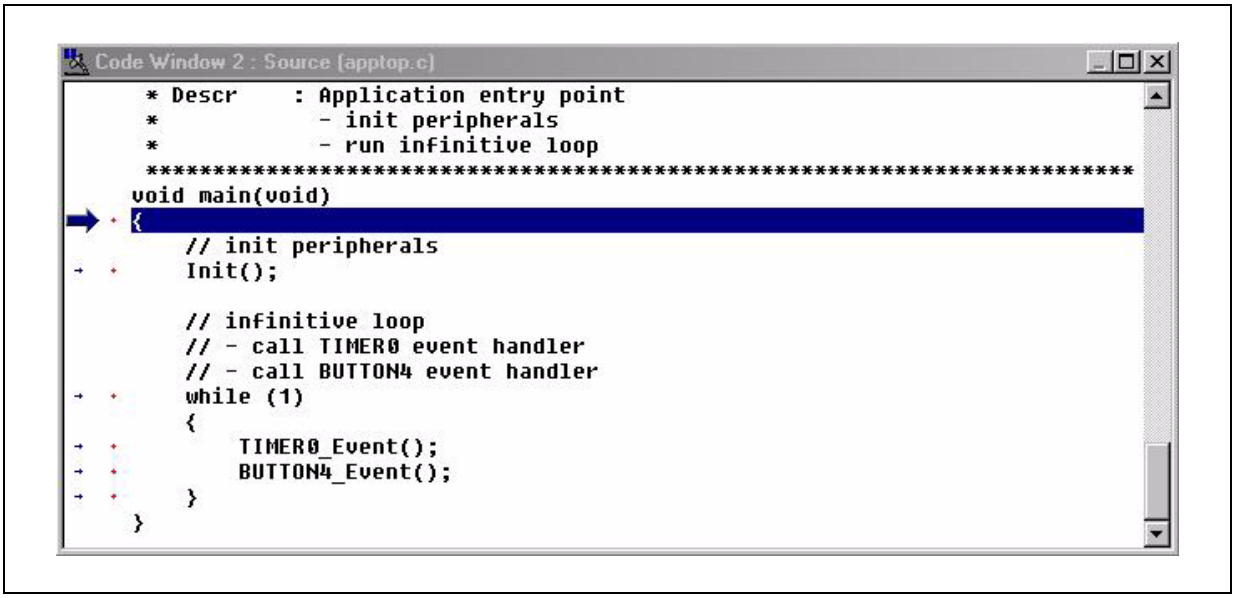
When the steps above are functional, it is time to load the application image to the processor. Select from the menu FileLoad Object/Debug file. A file selection dialog opens where you choose target image file, in this case get_start_ram.elf file. If it is confirmed, the image is downloaded to microprocessor. The download result is written in the status window.

Figure 22. Image download status



If the download operation has been successful, you should be able to execute the code. Write down gotil main command in status window and enter. Debugger runs the code until the entry of the main() function where it stops. If the image is built with debugging information in dwarf 2 format, you should see high level C language instructions in code window 2 area, see *Figure 23*. If the source code is not visible, check the setting by right click in the code window 2 and select Show Source/DisassemblyShow Source/Disassembly item.

Figure 23. Debugger – source code



Now you should be able to step instructions, place breakpoints, run the program and do basic debugging operations. To obtain more information about available operations and settings, see P&E debugger help (key F1). To verify the correct functionality of the application, click on HL button in order to let the application run without debugger intervention. LED1 should start flashing in 500 ms intervals. LED4 follows a state of button 4 where LED4 is on when the button 4 is pressed otherwise it is off. Now you are prepared for your own trials. You are free to modify the application functionality for example time period for LED1 or add new features you wish and test it directly on hardware running the program from RAM memory.

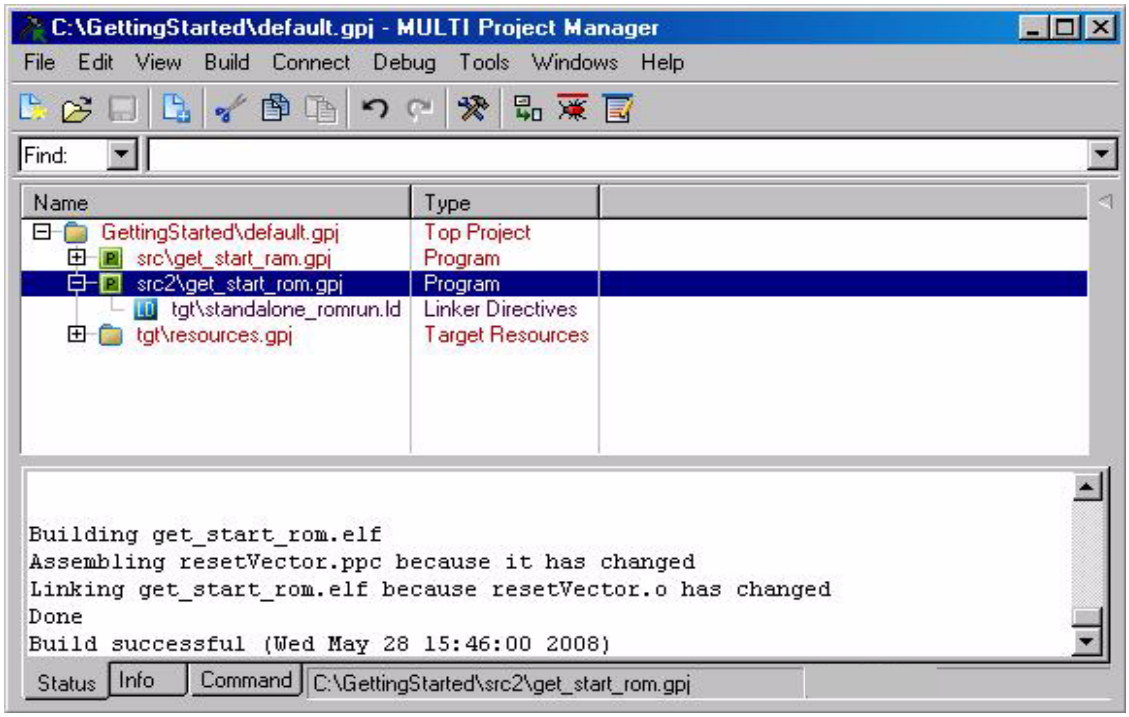
8 Preparing Flash image

In this section you give a look at how to prepare an image that runs from Flash memory. For that purpose you create a new project. Most of the steps you already know from previous examples. Now you extend your knowledge with a few new things which are necessary for a Flash image creation.

8.1 Creating get_start_rom project

Right click on GettingStarted\default.gpj in project manager and select Add Item into a default.gpj. Select New Program, give it a name get_start_rom and select Link to and Execute out of ROM program layout offered on consecutive pages. Based on this selection, project manager creates new program group under default.gpj project, see *Figure 24*.

Figure 24. GHS – program group for Flash image



8.2 Adding application file

The next action is to add source code for the application. You use the same application as for example in previous get_start_ram project. Right click on src2\get_start_rom.gpj line and select Add File into get_start_rom.gpj. In File selection dialog box go to the src directory and pick appTop.c file.

8.3 Adding ResetVector file

The ResetVector file is a new item needed in the project in order to run the application from Flash memory. The standard startup process of the SPC560P50xx processor consists of executing the special boot code which among other things reads specific addresses in a Flash memory where reset configuration half word is stored together with boot reset vector pointing to first valid instruction of the code (see the BAM chapter in the device reference manual).

Table 4. Boot address occupation

Address	Field name	Description
0x0000_0000	RCHW	Reset configuration half word
0x0000_0004	Reset vector	Address of the first code instruction

Table 5 shows the bit definition inside the RCHW configuration word.

Table 5. RCHW bit definition

Byte 0								Byte 1							
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	BOOTID							

8.3.1 BOOTID – Boot identifier (fixed 0x5A value)

Due to the BAM startup sequence it is necessary to fill the two lowest addresses in the Flash memory with correct values. For such purpose create a new assembly file resetVector.ppc. About the content of the file see the code below.

```
.section ".resetvector", "vax"
   .align 2# alignment to WORDS (4 bytes)

   .long 0x005A0000 # startup fixed code looked for by BAM
   .long 0x08# startup address (set to following address line)
```

The first line defines a name of the section that is used in linker directive file. The second line specifies alignment directive and the rest of the code puts right values to addresses 0x0 and 0x4. Then it adds the resetVector.ppc file into the get_start_rom project. Right click on **src2\get_start_rom.gpj**, select **Add File into get_start_rom** and choose **resetVector.ppc**.

8.4 Linker file standalone_romrun.ld

Last step before building operation is to check memory layout setting. Comparing this file to previous linker file used in **get_start_ram** project, you can see two main differences: change of code sections placement and a presence of one new section called.resetvector. An extract from standalone_romrun.ld file is below.

```
MEMORY {
//  512K Internal Flash
    flash_rsvd1   : ORIGIN = 0x00000000, LENGTH = 0
```

```
    flash_reset   : ORIGIN = .,          LENGTH = 8
    flash_rsvd2   : ORIGIN = .,    LENGTH = 0
    flash_memory  : ORIGIN = .,          LENGTH = 512K-8
    flash_rsvd3   : ORIGIN = .,          LENGTH = 0

//  40KB of internal SRAM starting at 0x40000000
    dram_rsvd1    : ORIGIN = 0x40000000, LENGTH = 0
    dram_memory   : ORIGIN = .,    LENGTH = 40K
}
SECTIONS
{
// RAM SECTIONS
    .PPC.EMB.sdata0                  ABS : > dram_memory
    .PPC.EMB.sbss0                CLEAR ABS : > .

    .sdabase             ALIGN(16): > dram_memory
    .sdata     : > .
    .sbss      : > .
    .data      : > .
    .bss       : > .
    .heap                  ALIGN(16) PAD(heap_reserve)   : > .
    .stack                 ALIGN(16) PAD(stack_reserve) : > .

// ROM SECTIONS
    .resetvector  NOCHECKSUM: > flash_reset
    .text      : > flash_memory
    .vletext: > .
    .syscall: > .
    .rodata : > .
    .sdata2 : > .
    .secinfo : > .
    .fixaddr : > .
    .fixtype : > .

    .CROM.PPC.EMB.sdata0   CROM(.PPC.EMB.sdata0) : > .
    .CROM.sdata                 CROM(.sdata) : > .
    .CROM.data                 CROM(.data) : > .
```

8.5 Building the Flash image

Before starting the build process, set the following project parameters:

1. A name of the image **get_start_rom.elf**
2. The support for debug symbols in **dwarf 2** format
3. Preprocessor constant for compilation **BUILD_TO_FLASH**

The set of the parameters can be done via build options dialog opened from the menu **EditSet Build Options** when the **src2\get_start_rom.gpj** line inside project manager window is highlighted. The image name is modified in ProjectOutput Filename section. Dwarf 2 support option is set in **AdvancedDebugging OptionsNative Debugging Options** section and preprocessor constant is set in PreprocessorDefine Preprocessor Symbol.

Preprocessor constant **BUILD_TO_FLASH** controls including of **__ghs_board_memory_init()** function into the image which takes care of internal RAM and its companion ECC module initialization. The SRAM initialization sequence consists of multiple 32-bit writes to SRAM which initialize error correction unit with proper values. Once the project is properly set up, start the building process by pressing the F7 key. The status of the process is shown in status window present below inside MULTI environment. Everything should be ok but if for any reason it is not, read error messages and fix errors accordingly. The result of successive build operation is the generation of image file **get_start_rom.elf** under default project directory.

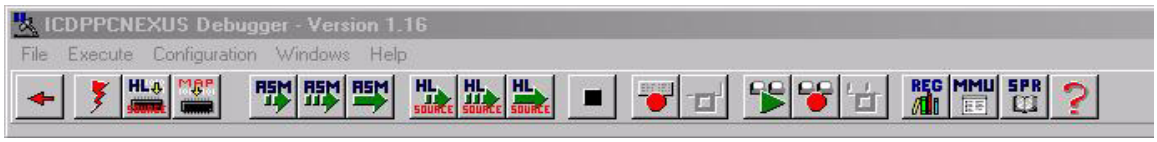
9 Programming Flash image

Last step of **get_start_rom** example is to program it into the internal Flash memory of the processor. P&E's debugger tool provides such a possibility. At first it needs to establish a debug connection by starting **ICDPPCNEXUS** program, selecting the right connection method and clicking on connect button.

9.1 High level load

Once the debug mode is activated, click on the high level load icon in the toolbar that opens File Load Dialog (*Figure 26*) with several options and input boxes

Figure 25. Flashing – high level load

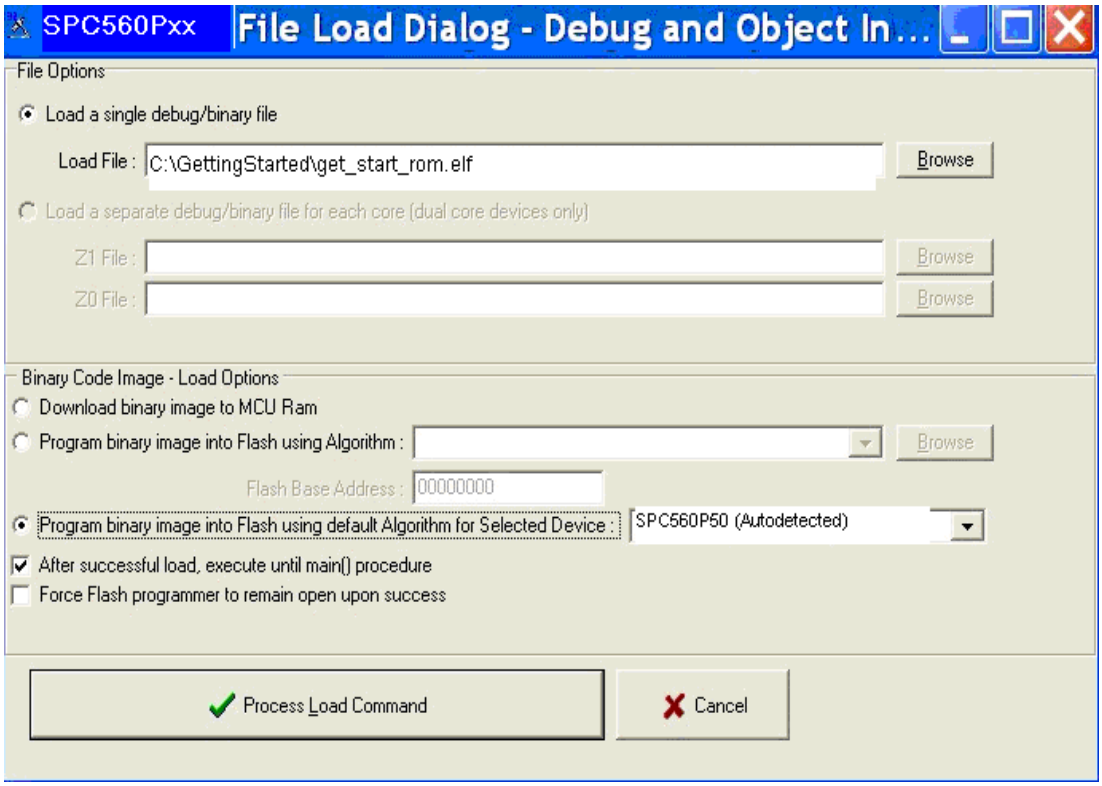


Here select an image file to program and be sure to select the "Program binary image into Flash using default Algorithm for Selected Device" check box. This option selects the default program algorithm.

Caution: If the program algorithm is selected manually, choosing the wrong programming algorithm can damage the Flash and censorship state of the microcontroller.

Then load **get_start_rom.elf** file.

Figure 26. Flashing – load dialog



9.2 Programming an image

Check that the base address of the Flash memory input box is set to 0x0. Then the Flash memory starts programming. The programming operation consists of two steps:

1. Memory erase
2. Program operation

Both operations are provided by a separate application, P&E programmer, called automatically from the P&E debugger. When the programming operation is finished the P&E debugger executes reset followed by the execution of configuration script. Programming is stopped on the first instruction of the code, here an address 0x8. Now issue a command gotil in the main status window which executes the program from Flash memory until entry of main() function where it stops. Now the program placed in Flash memory can run even without a debugger. The simplest test is to disconnect the debugger and press the reset button on the SPC560P50xx mini-module board. After that the program starts to execute from the internal Flash memory and you should see LED1 flashing in predefined time interval and change of LED4 state when button 4 is pressed and released.

10 Summary

This tutorial is issued to prepare projects, to define which files are necessary to write the simplest application, how to build it into an image that you can debug or run independently. It shows what is different between preparing an image to be executed from RAM and an image to be run from Flash memory. The last argument treated is how to program data into internal Flash memory of the microcontroller by means of P&E debug tool. From now on, you should be able to prepare your own program for SPC560P50xx microcontroller that you can debug directly on the hardware evaluation board.

Appendix A appTop.c

```
/*****************************************************************
**********
 * FILE          : appTop.c
 * DESCRIPTION   : GettingStarted example without interuppts
 * VERSION       : 1.0
 * RESOURCES     : - INTERRUTPS = no interrupts
 *                 - CLK = crystal in bypass mode (12 MHz)
 * HISTORY       : first version
*****************************************************************
*********/

#include "..\spcHeaders\jdp_0100.h"


/*****************************************************************
**********
 * Name      : __ghs_board_memory_init
 * Descr     : Called from GHS startup routine to enable user setup
of important
 *             resources before execution of startup operations
*****************************************************************
********/
#ifdef BUILD_TO_FLASH

void __ghs_board_memory_init(void)
{
    // RAM memory initialization
    #pragma asm
        e_lis  r6, 0x4000
        e_or2i r6, 0x0000
        e_lis  r7, 0x4000
        e_or2i r7, 0x9FFF

    init_ram_loop:
        e_stmw  r0,0(r6)
        e_addi  r6,r6,128
        se_cmp  r6,r7
        e_blt   cr0,init_ram_loop
    #pragma endasm
}

#endif //BUILD_TO_FLASH

/*****************************************************************
**********
 * Name      : Init
 * Descr     : Initialization of CPU peripherals
*****************************************************************
*********/
```

```
void Init(void)
{
    // mode & peripheral enable
    ME.RUNPC[1].B.RUN0 = 1;         // prepare configuration word
    ME.PCTL[92].B.RUN_CFG = 1;      // and use it for PIT enable in
RUN0 mode

    // mode transition request
    ME.MCTL.R = 0x40005AF0;
    ME.MCTL.R = 0x4000A50F;
    while (ME.GS.B.S_CURRENTMODE != 4) ;  // wait for DRUN->RUN0
transition

    // PIT timer configuration
    PIT.PITMCR.R = 1;             // enable periph. & stop timer
during debug
    PIT.CH[0].LDVAL.R = 0x7A1200; // reload value
    PIT.CH[0].TFLG.B.TIF = 1;     // clear potential flag
    PIT.CH[0].TCTRL.B.TEN = 1;    // enable timer

    // GPIO configuration
    SIU.PCR[52].R = 0x0200;          // enable output for LED1
    SIU.PCR[51].R = 0x0100;          // enable input for BUTTON4
    SIU.PCR[55].R = 0x0200;          // enable output for LED4

    // disable SWT
    SWT.SR.B.WSC = 0xC520;           // clear soft lock sequence
    SWT.SR.B.WSC = 0xD928;
    SWT.CR.B.WEN = 0;                // disable watchdog
}

/*****************************************************************
**********
 * Name     : TIMER0_Event
 * Descr    : Action linked to PIT Channel[0] event
 *            - Toggle with LED1 (pin 58)
*****************************************************************
*********/
void TIMER0_Event(void)
{
    // wait for timer flag signalling expiration
    // then toggle with pin and clear flag by writing '1'
    if (PIT.CH[0].TFLG.B.TIF)
    {
        SIU.GPDO[52].B.PDO = ~SIU.GPDO[52].B.PDO;
        PIT.CH[0].TFLG.B.TIF = 1;
    }
}

/*****************************************************************
**********
 * Name     : BUTTON4_Event
```

```
 * Descr    : Action linked to Input pin from Button4
 *            - Light on LED4 when button4 is pressed
*****************************************************************
*********/
void BUTTON4_Event(void)
{
    if (SIU.GPDI[51].B.PDI == 0)
        SIU.GPDO[55].B.PDO = 0;
    else
        SIU.GPDO[55].B.PDO = 1;
}

/*****************************************************************
**********
 * Name     : main
 * Descr    : Application entry point
 *            - init peripherals
 *            - run infinitive loop
*****************************************************************
*********/
void main(void)
{
    // init peripherals
    Init();

    // infinitive loop
    // - call TIMER0 event handler
    // - call BUTTON4 event handler
    while (1)
    {
        TIMER0_Event();
        BUTTON4_Event();
    }
}
```

Appendix B resetVector.ppc

```
/*****************************************************************
************
** FILE: STARTUP.PPC
**
** DESCRIPTION:
**  reset vector constants used by BAM
**
*****************************************************************
*********/

/* CODE section aligned to words (4 bytes) */

    .section ".resetvector", "vax"
   .align 2  # alignment to WORDS (4 bytes)

   .long 0x0000005A    # startup fixed code looked for by BAM
   .long 0x08# startup address (set to following address line)
```

Appendix C standalone_ram.ld

```
MEMORY {
//  512K Internal Flash
    flash_rsvd1  : ORIGIN = 0x00000000, LENGTH = 0
    flash_reset   : ORIGIN = .,         LENGTH = 8
    flash_rsvd2   : ORIGIN = ., LENGTH = 0
    flash_memory : ORIGIN = .,          LENGTH = 512K-8
    flash_rsvd3  : ORIGIN = .,          LENGTH = 0

//  40KB of internal SRAM starting at 0x40000000
    dram_rsvd1   : ORIGIN = 0x40000000, LENGTH = 0
    dram_memory  : ORIGIN = .,  LENGTH = 40K
}
DEFAULTS {
    stack_reserve = 4K
    heap_reserve = 2K
}
//
// Program layout for running out of RAM.
//

SECTIONS
{
    .PPC.EMB.sdata0                     ABS : > dram_memory
    .PPC.EMB.sbss0                            CLEAR ABS : > .

    .text               : > dram_memory
    .vletext     : > .
    .syscall     : > .
    .resetvectorNOCHECKSUM       : > .
    .secinfo     : > .
    .rodata     : > .
    .sdata2     : > .
    .fixaddr    : > .
    .fixtype   : > .

    .sdabase               ALIGN(16) : > dram_memory
    .sdata     : > .
    .sbss      : > .
    .data      : > .
    .bss       : > .
    .heap                  ALIGN(16) PAD(heap_reserve)   : > .
    .stack                 ALIGN(16) PAD(stack_reserve) : > .

// These special symbols mark the bounds of RAM and ROM memory.
// They are used by the MULTI debugger.
//
    __ghs_ramstart  = MEMADDR(dram_rsvd1);
    __ghs_ramend    = MEMENDADDR(dram_memory);
```

```
    __ghs_romstart  = MEMADDR(flash_rsvd1);
    __ghs_romend    = MEMENDADDR(flash_rsvd2);
}
Appendix D - standalone_romrun.ld

MEMORY {

//  512K Internal Flash
    flash_rsvd1  : ORIGIN = 0x00000000, LENGTH = 0
    flash_reset   : ORIGIN = .,        LENGTH = 8
    flash_rsvd2   : ORIGIN = ., LENGTH = 0
    flash_memory : ORIGIN = .,          LENGTH = 512K-8
    flash_rsvd3  : ORIGIN = .,          LENGTH = 0


//  40KB of internal SRAM starting at 0x40000000
    dram_rsvd1   : ORIGIN = 0x40000000, LENGTH = 0
    dram_memory  : ORIGIN = .,  LENGTH = 40K

}
DEFAULTS {

    stack_reserve = 4K
    heap_reserve = 2K

}
//
// Program layout for starting in ROM, copying data to RAM,
// and continuing to execute out of ROM.
//


SECTIONS
{

//
// RAM SECTIONS
//
    .PPC.EMB.sdata0             ABS : > dram_memory
    .PPC.EMB.sbss0               CLEAR ABS : > .

    .sdabase            ALIGN(16): > dram_memory
    .sdata  : > .
    .sbss     : > .
    .data     : > .
    .bss      : > .
    .heap      ALIGN(16) PAD(heap_reserve)  : > .
    .stack     ALIGN(16) PAD(stack_reserve) : > .

//
// ROM SECTIONS
//
```

```
    .resetvectorNOCHECKSUM: > flash_reset
    .text      : > flash_memory
    .vletext: > .
    .syscall: > .

    .rodata : > .
    .sdata2 : > .

    .secinfo : > .
    .fixaddr : > .
    .fixtype : > .

    .CROM.PPC.EMB.sdata0   CROM(.PPC.EMB.sdata0) : > .
    .CROM.sdata               CROM(.sdata) : > .
    .CROM.data             CROM(.data) : > .

//
// These special symbols mark the bounds of RAM and ROM memory.
// They are used by the MULTI debugger.
//
    __ghs_ramstart  = MEMADDR(dram_rsvd1);
    __ghs_ramend    = MEMENDADDR(dram_memory);
    __ghs_romstart  = MEMADDR(flash_rsvd1);
    __ghs_romend    = MEMENDADDR(flash_rsvd2);

//
// These special symbols mark the bounds of RAM and ROM images of
boot code.
// They are used by the GHS startup code (_start and
__ghs_ind_crt0).
//
    __ghs_rambootcodestart  = 0;
    __ghs_rambootcodeend    = 0;
    __ghs_rombootcodestart  = ADDR(.text);
    __ghs_rombootcodeend    = ENDADDR(.fixtype);
}
```

Revision history

Table 6. Document revision history

Date	Revision	Changes
09-Jul-2015	1	Initial release