AVIX Real Time Operating System

AVIX for PIC24-dsPIC Microchip MPLAB Port Guide

Product Version 5.0.0

AVIX-RT

AVIX
Advanced
Virtual
Integrated
eXecutive

AVIX-RT
Maïsveld 84
5236 VC 's-Hertogenbosch
The Netherlands

phone +31(0)6 15 28 51 77
e-mail info@avix-rt.com
www www.avix-rt.com

Table of Contents

Table of Figures

Table of Tables

Table of Code Samples

1 Introduction

This document describes how to use AVIX with a controller belonging to one of the PIC24F, PIC24H, PIC24E, dsPIC30F, dsPIC33E or dsPIC33F families of microcontrollers using the MPLAB development environment combined with the C30/XC16 compiler suite.

This document forms an extension to the AVIX User Guide and Reference Guide and should be considered an indivisible part of it. Terms and abbreviations defined in the AVIX User Guide and Reference Guide are also applicable to this Port Guide.

► *The full name of the AVIX port described in this document is AVIX for PIC24-dsPIC. Whenever using the name AVIX throughout this document, this refers to AVIX for PIC24-dsPIC.*

1.1 Reader Guidance

To find the information you are looking for as efficient as possible, here an overview is given of the different chapters this document contains.

The different chapters can be divided in three categories:

1. **Chapter 2** contains background information on the compiler toolsuite AVIX can be used with.

2. **Chapter 3** contains a compact overview how to create an AVIX based application. When familiar with AVIX, this is all that is needed to get up and running.

3. **Chapters 4, 5, 6 and 7** contain information how to; install AVIX, make required changes to the project settings, deal with linking large programs and configure AVIX. The information in these chapters is not pure background but essential to obtain a working project.

4. The remaining chapters can be considered to contain background information which is not essential to get up and running but do contain information about the way AVIX interacts with the application and with the underlying hardware. In more detail these are:

   - **Chapter 8, Stack usage and Interrupt Service Routines,** provides a description of how to declare ISR's to make use of the AVIX system stack.

   - **Chapter 9, Controller Specific Capabilities,** provides details about capabilities specific to the targeted controller families. Topics are use of the TBLPAG register for access to FLASH memory, use of the PSVPAG register for Program Space Visibility and use of DSP functionality[1].

   - **Chapter 10, Using memory in the Extended Data Space**, provides information how AVIX allows the memory in the Extended Data Space (EDS memory), to be used by your application.

   - **Chapter 11, Power Management**, provides information how to use the power management capabilities offered by the controller and how this is supported by AVIX.

   - **Chapter 12, Resource Usage**, description of the hardware resources used by AVIX and the rules to obey when these or related resources are required by the application.

[1] DSP functionality is only available on controllers belonging to the dsPIC30F and dsPIC33F families of microcontrollers.

2 Compiler selection and compatibility

AVIX can be used with applications build using either the C30 or the newer XC16 compiler suite. Both compiler suites can be selected to be used from within the legacy MPLAB8x development environment or from the new MPLABX development environment.

Throughout this document a number of dialogs are shown as they are used by the MPLAB development environment to make specific project settings. These dialogs are based on the C30 compiler suite. When using the XC16 compiler suite, similar dialogs are presented by MPLAB8x in essence offering the exact same entry fields. Some of the content of the XC16 based dialogs may be a little different from those of the C30 based dialogs.

► *In case of using MPLAB8x, when switching between compiler toolsuites (either from C30 to XC16 or from XC16 to C30), a number of project settings are restored to their default values. Since AVIX depends on a number of custom project settings, after switching between compiler toolsuites, it is essential for these settings to be restored. For details on the applicable settings, see chapter 5.1*

When using MPLABX, switching between compilers does not restore any other setting.

When referring to the compiler toolsuite in this document, use is made of the phrase C30/XC16 compiler suite to denote both compiler toolsuites may be used.

3 Getting Started

This chapter contains a quick guide to get up and running with AVIX and the steps needed to create an AVIX based project.

AVIX can be used with the legacy Microchip development environment MPLAB8x and the new Microchip development environment MPLABX, both with the C30/XC16 compiler suite. Both environments require a number of mandatory project settings. These settings are mentioned in the steps described below referring §5 where the applicable settings are described in more detail.

AVIX virtually works out of the box and requires the following steps to be up and running with a new project:

1	Install AVIX See §4 for details. The names and relative order of directories created by the setup utility must remain as specified since AVIX depends on this. The location where AVIX is installed is not important but advised is to choose a project related location.
2	Create a new project Create a new MPLAB project based on the C30/XC16 compiler suite. For MPLAB8x use ***Project – Project Wizard...***, for MPLABX use ***File – New Project…***
3	Create the projects main source file and add to the project Manually add a source file (.c) to the project containing function '`void avixMain(void)`'. An AVIX project does not contain a user provided function '`main`' which is implemented by AVIX. The application must offer '`avixMain`' instead which is the entry point of an AVIX based application.
4	Add AVIX configuration file (AVIXConfig.c) to the project This AVIX file must be built as part of the project like any other project source file.
5	Add AVIX library file (AVIX_PIC24-dsPIC_MICROCHIP_MPLAB_E_050000_FH.a [2, 3]) to the project. This AVIX file must be linked against the other project binaries
6	Add AVIX interface include path to project settings The location of the AVIX 'Interface' directory must be added to the project settings. See §5.1.1 for details on the MPLAB8x environment and §5.2.1 for details on the MPLABX environment.
7	Allow the compiler to pass 'C' structs by value AVIX is highly type safe, a feature requiring the compiler to pass 'C' structs by value. Change the project settings to allow this by adding compiler flag –fno-pcc-struct-return. See §5.1.2 for details on the MPLAB8x environment and §5.2.2 for details on the MPLABX environment. ***This setting is highly important. An AVIX based application build without this setting will not work.***
8	Set the project to generate binary files according the ELF format The AVIX library is distributed in the ELF format. For compatibility the project specific source files must be build according the same binary format. See §5.1.3 for details on the MPLAB8x environment and §5.2.3 for details on the MPLABX environment.

Table 1: Project Setup Steps

[2] The capital E in the file name denotes the library of an Extended distribution. Depending on the type of distribution, the filename may be different.
[3] Library AVIX_PIC24-dsPIC_MICROCHIP_MPLAB_E_050000_FH.a is for use with PIC24F, PIC24H, dsPIC30F and dsPIC33F controllers. When using PIC24E or dsPIC33E controllers, use library AVIX_PIC24-dsPIC_MICROCHIP_MPLAB_E_050000_EP.a instead.

The result of the above steps is a project that builds. At this point however, the application does not yet use any AVIX function resulting in the linker ignoring the AVIX library. This in turn results in the linker complaining function `'main'`, which is implemented in this library, cannot be found.

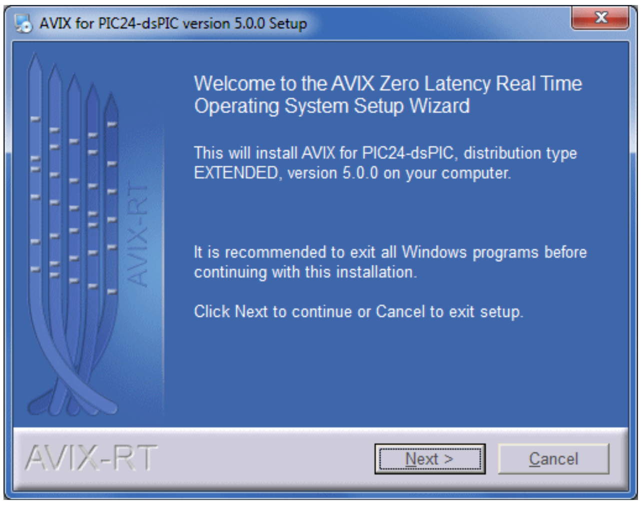
To solve this, AVIX functions must be called from application code, a step required anyway.

From this point on, application code will be added to the project in the form of threads, ISR's and DIH's. Threads are separate 'C' functions that are registered with AVIX so they will run as a thread. Thread functions are typically registered from '`avixMain`' using function `avixThread_Create`. See the AVIX User Guide and Reference Guide for details.

Function avixMain is executed with all interrupts disabled. For AVIX to work correctly it is essential during execution of avixMain interrupts remain disabled. Make sure not to enable interrupts during execution of avixMain, neither direct nor indirect by (third party) functions being called.

4 How to Install AVIX

AVIX is distributed in the form a Windows based setup utility, the opening screen of which is shown in Figure 1. Note that the content of this screen may vary based on the type of distribution you have acquired.



Welcome to the AVIX Zero Latency Real Time Operating System Setup Wizard

This will install AVIX for PIC24-dsPIC, distribution type EXTENDED, version 5.0.0 on your computer.

It is recommended to exit all Windows programs before continuing with this installation.

Click Next to continue or Cancel to exit setup.

Figure 1: AVIX Main Setup Screen

► *Installing AVIX is very straightforward and the only relevant information that is to be provided is the directory where AVIX is to be installed.*

When pressing Next, the install procedure will start. During the install process you will be asked a number of questions allowing you to install those parts of AVIX you need in the directory of your choice.

Note that the AVIX setup utility does not install any executables apart from an uninstall utility named uninstall.exe. This utility is installed in a directory named Uninstall which is created in the main installation directory that is selected as part of the install procedure. Besides this uninstall utility, directory Uninstall contains a number of additional files needed by the uninstall utility.

AVIX can be uninstalled by running this utility through “Add or Remove Programs” present in the Windows Control Panel or from the Windows Start Menu shortcut created as part of the install process. It is not possible to directly run uninstall.exe.

4.1 Directory structure and files

After installing AVIX, the following directory/file structure is created where the root, <Install Directory>, is the directory provided during the install procedure[4]:

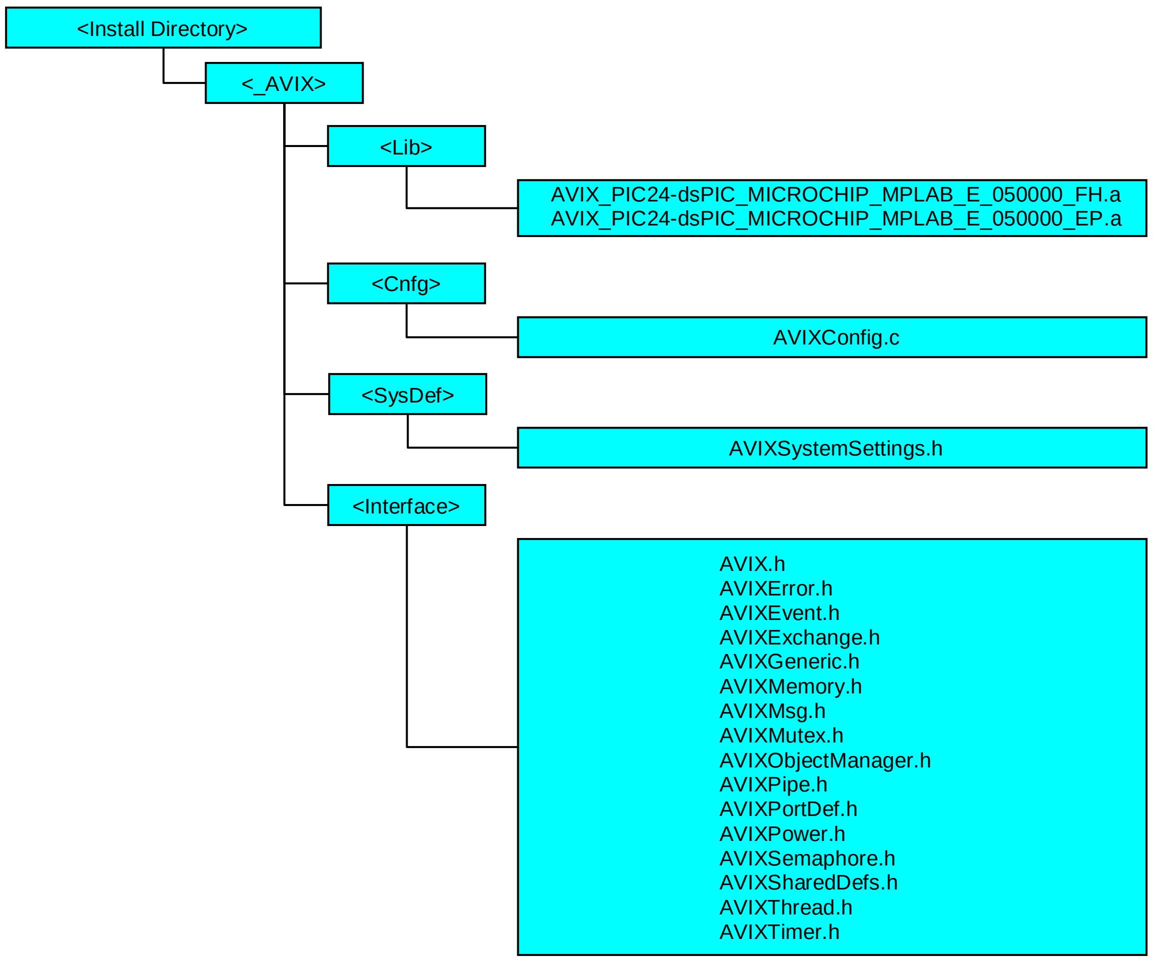


Figure 2: AVIX Directory Structure

The names and relative order of directories ‘Lib’, ‘Cnfg’, ‘SysDef’ and ‘Interface’ must remain as shown in Figure 2 since the build process depends on this.

Below an overview is given of the installed files. Source files requiring AVIX functionality can include file AVIX.h. Optionally the specific header for the desired functionality can be included. For ease of use this is however not advised since AVIX.h in itself is always sufficient.

[4] Depending on options offered during installation, additional directories may be created under the <Project Directory> containing for instance utility software. These additional directories and the files contained in them do not belong to the core of AVIX but are intended to be used with AVIX. When applicable, these files are documented in separate documents.

- **AVIX_PIC24-dsPIC_MICROCHIP_MPLAB_E_050000_FH.a**: Contains the AVIX code. This is a library file that must be linked in the project. The capital ‘E’ in this filename denotes an extended distribution. Depending on the type of distribution used, another character may be present in the filename. (FH) denotes this library is for the F and H parts (PIC24F, PIC24H, dsPIC30F and dsPIC33F). When using a PIC24E or dsPIC33E controller, use library **AVIX_PIC24-dsPIC_MICROCHIP_MPLAB_E_050000_EP.a** instead.

- **AVIXSystemSettings.h**: File containing the configuration settings as described in §7.
  This file does never need to be explicitly included.

- **AVIXConfig.c**: Source file needed for configuring AVIX through the settings in AVIXSystemSettings.h.
  This file must be compiled and linked as part of the project under development.

- **AVIX.h**: Header file including all other header files.
  Including this header file is sufficient to use all AVIX functions without including one of the specific header files.

- **AVIXError.h**: Header file to include when the error facility is used.

- **AVIXEvent.h**: Header file to include when event group functionality is used.

- **AVIXExchange.h:** Header file to include when exchange functionality is used.

- **AVIXGeneric.h**: Header containing generic definitions used from the other header files. ***This file does never need to be explicitly included.***

- **AVIXMemory.h** Header containing definitions for memory pool functions.

- **AVIXMsg.h**: Header file to include when message functionality is used.

- **AVIXMutex.h**: Header file to include when mutex functionality is used.

- **AVIXObjectManager.h**: Header file containing generic definitions related to kernel object handling used for the other header files.
  This file does never need to be explicitly included.

- **AVIXPipe.h**: Header file to include when pipe functionality is used.

- **AVIXPortDef.h:** Header containing platform definitions, used by other headers.
  This file does never need to be explicitly included.

- **AVIXPower.h:** Header file to when power management functionality is used.

- **AVIXSemaphore.h**: Header file to include when semaphore functionality is used.

- **AVIXSharedDefs.h**: Header file with definitions shared between the configuration file and the AVIX library.
  This file does never need to be explicitly included.

- **AVIXThread.h**: Header file to include when thread functionality is used.

- **AVIXTimer.h**: Header file to include when timer functionality is used.

5 Development Environment Settings

AVIX can be used with the legacy Microchip development environment MPLAB8x and the new MPLABX, both with the C30/XC16 compiler suite. A description of these environments falls outside the scope of this document. To use AVIX from within these environments, a number of AVIX specific project settings are however required which are described in this chapter.

5.1 Settings for the MPLAB8x environment

This chapter contains mandatory projects settings when using AVIX with the legacy Microchip development environment MPLAB8x.

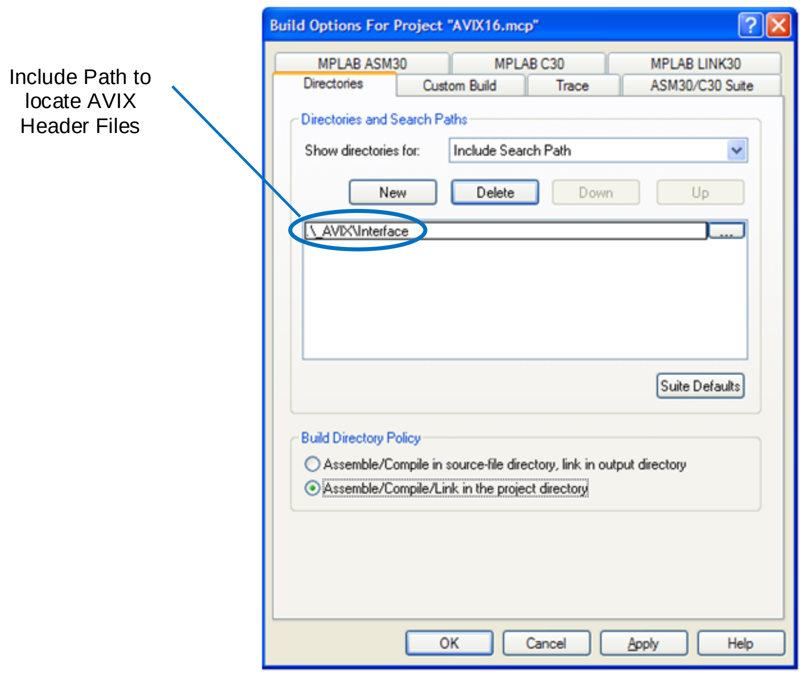
5.1.1 Specify AVIX Include Path

An AVIX based project must have access to the directory containing the AVIX interface files, this location must be added to the MPLAB8x project settings.

Open the applicable dialog through ***Project - Build Options… - Project – Directories Tab.*** Select 'Include Search Path' from the 'Show Directories for:' dropdown control.

The dialog is shown in Figure 3.

The include path entered depends on the location where AVIX is installed. The value shown is based on AVIX being installed in the project directory.



Build Options For Project "AVIX16.mcp"

Directories and Search Paths

Show directories for: Include Search Path

._AVIX\Interface

Build Directory Policy

( ) Assemble/Compile in source-file directory, link in output directory
(•) Assemble/Compile/Link in the project directory

Figure 3: Set AVIX Interface include path for MPLAB8x

5.1.2 Structure based function parameters

The functions offered on the AVIX API use ‘C’ structures. The C30/XC16 compiler suite allows for structures to be passed by value or by reference. To implement the high level of type safety the AVIX library is built based on passing structures by value. This implies that code using AVIX functions has to use the same mechanism in order to be compatible with the library.

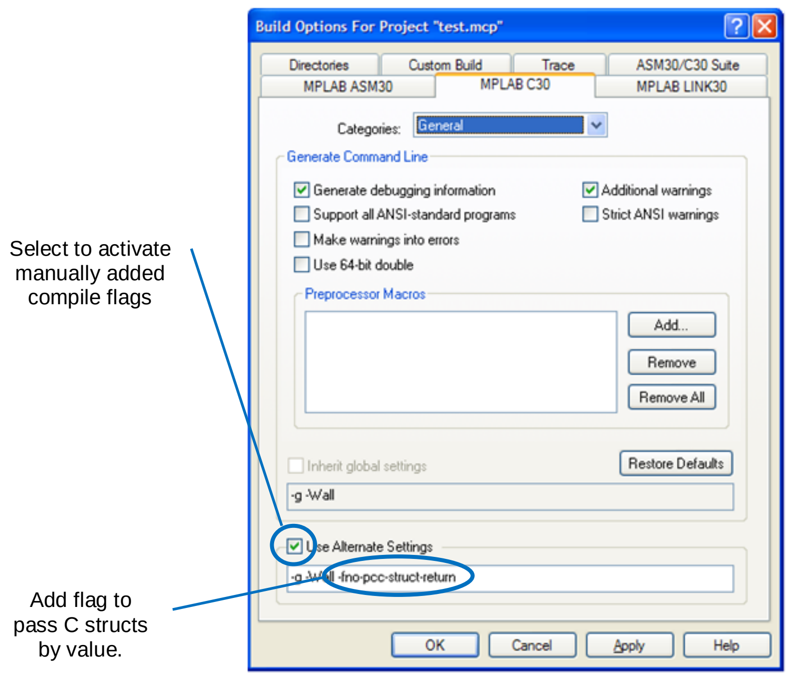
This is accomplished by using compiler flag **-fno-pcc-struct-return**.

Open the applicable dialog through ***Project - Build Options… - Project – MPLAB C30 Tab*** and add the required flag according Figure 4.

It is essential to select checkbox ‘Use Alternate Settings’ in order for the setting to be used.

A consequence of using ‘Alternate Settings’ is that settings made through one of the dialogs (like for instance the compiler optimization level) have to be copied manually to the Alternate Settings field in order to be effective. When for instance changing the compiler optimization level to O3 through the applicable entry, this value will be entered in the default dialog field for settings. After making the setting, manually copy this to the field also used to enter –fno-pcc-struct-return.

► *Setting flag **-fno-pcc-struct-return** requires use of alternate settings (checkbox “Use Alternate Settings” in the dialog shown in Figure 4. Activating alternate settings implies that changes to the project settings made in the regular way have to be manually copied to the same field where flag –fno-pcc-struct-return is contained in, in order to be persistent when closing this dialog.*



Categories: General

Generate Command Line

✓ Generate debugging information
☐ Support all ANSI-standard programs
☐ Make warnings into errors
☐ Use 64-bit double
✓ Additional warnings
☐ Strict ANSI warnings

☐ Inherit global settings

-g -Wall

✓ Use Alternate Settings

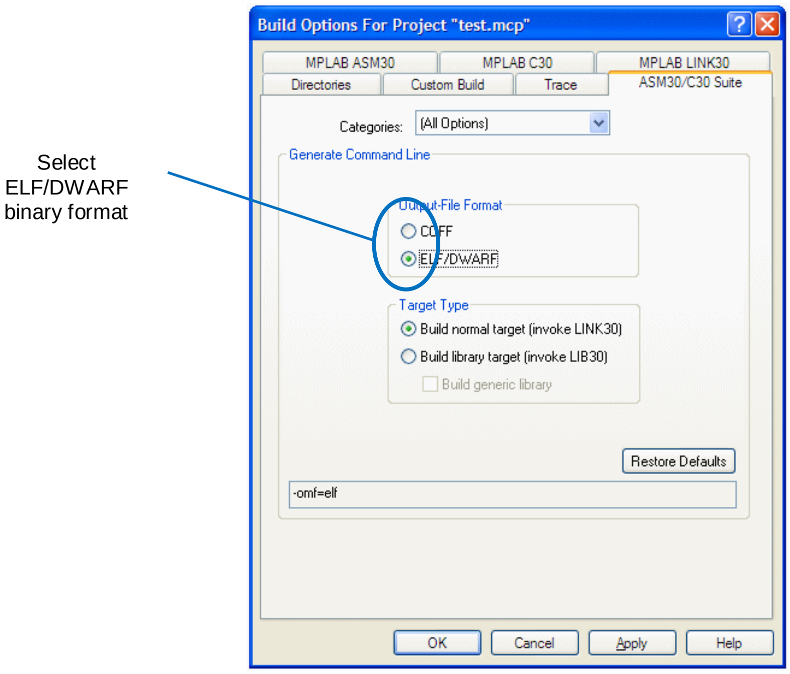
-g -Wall -fno-pcc-struct-return

Figure 4: Struct parameter project setting for MPLAB8x

5.1.3 ELF object file type

The AVIX library is distributed in the ELF format. For compatibility the project specific source files must be build according the same binary format. Selecting the ELF binary format is done through through the project settings dialog.

Open the applicable dialog through ***Project - Build Options… - Project – ASM30/C30 Suite Tab*** and select the desired option according Figure 5.



Categories: (All Options)

Generate Command Line

Output-File Format: COFF [✓ ELF/DWARF]

Target Type: [✓ Build normal target (invoke LINK30)] Build library target (invoke LIB30)

Build generic library

-omf=elf

Figure 5: ELF binary format project settings for MPLAB8x

5.2 Settings for the MPLABX environment

This chapter contains mandatory projects settings when using AVIX with the new Microchip development environment MPLABX.

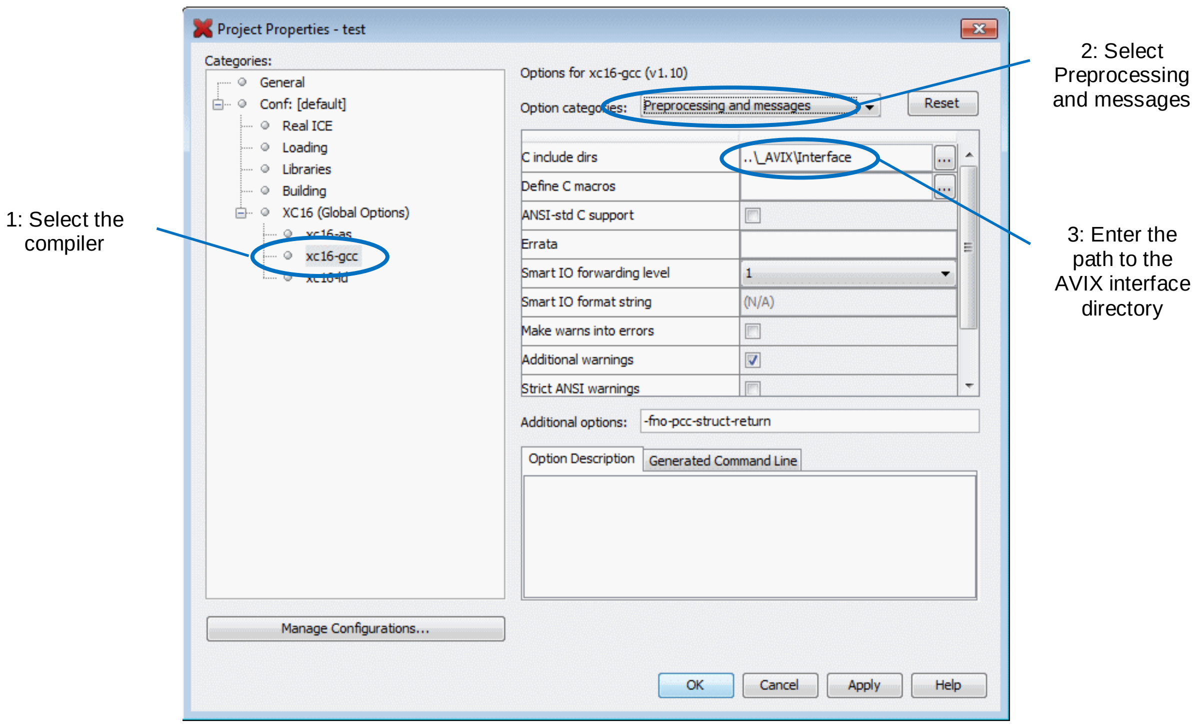
5.2.1 Specify AVIX Include Path

An AVIX based project must have access to the directory containing the AVIX interface files, this location must be added to the MPLABX project settings.

Open the applicable dialog by clicking the left mouse button on the project name and select entry ***Properties*** from the drop down menu.

The dialog is shown in Figure 6 where the selections to make in this dialog are highlighted.

The include path entered depends on the location where AVIX is installed. The value shown is based on AVIX being installed in the same directory where the MPLABX project files directory is located.



Options for xc16-gcc (v1.10)

Option categories: Preprocessing and messages

C include dirs	.._AVIX\Interface
Define C macros	
ANSI-std C support	☐
Errata	
Smart IO forwarding level	1
Smart IO format string	(N/A)
Make warns into errors	☐
Additional warnings	☑
Strict ANSI warnings	☐

Additional options: -fno-pcc-struct-return

Figure 6: Set AVIX Interface include path for MPLABX

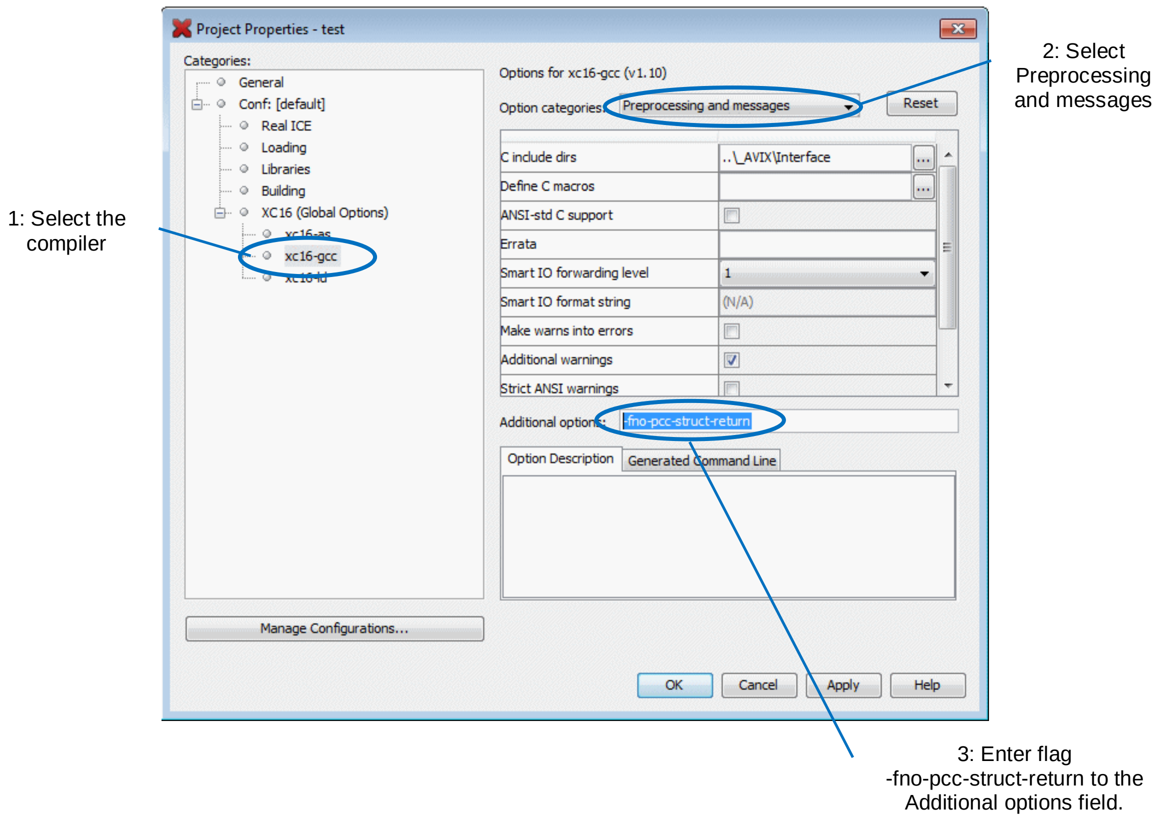
5.2.2 Structure based function parameters

The functions offered on the AVIX API use ‘C’ structures. The C30/XC16 compiler suite allows for structures to be passed by value or by reference. To implement the high level of type safety the AVIX library is built based on passing structures by value. This implies that code using AVIX functions has to use the same mechanism in order to be compatible with the library.

This is accomplished by using compiler flag **-fno-pcc-struct-return**.

Open the applicable dialog by clicking the left mouse button on the project name and select entry ***Properties*** from the drop down menu.

The dialog is shown in Figure 7 where the selections to make in this dialog are highlighted.



Options for xc16-gcc (v1.10)

Option categories: Preprocessing and messages

C include dirs	.._AVIX\Interface
Define C macros	
ANSI-std C support	☐
Errata	
Smart IO forwarding level	1
Smart IO format string	(N/A)
Make warns into errors	☐
Additional warnings	☑
Strict ANSI warnings	☐

Additional options: -fno-pcc-struct-return

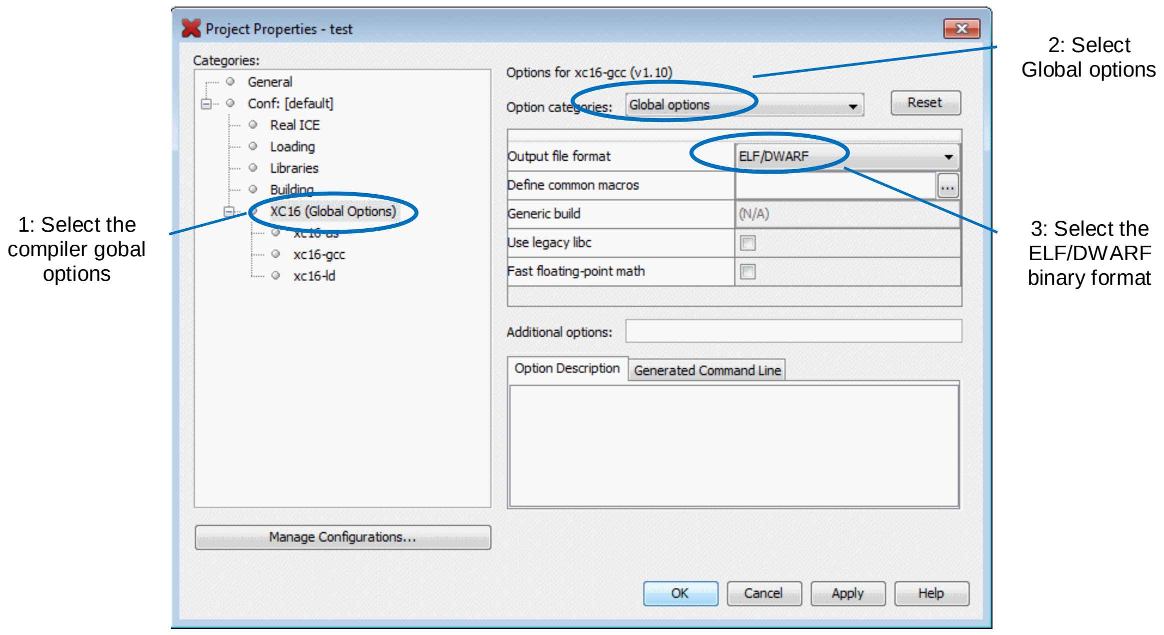
Figure 7: Struct parameter project setting for MPLABX

5.2.3 ELF object file type

The AVIX library is distributed in the ELF format. For compatibility the project specific source files must be build according the same binary format. Selecting the ELF binary format is done through through the project settings dialog.

Open the applicable dialog by clicking the left mouse button on the project name and select entry ***Properties*** from the drop down menu.

The dialog is shown in Figure 8 where the selections to make in this dialog are highlighted



Options for xc16-gcc (v1.10)

Option categories: Global options

Output file format	ELF/DWARF
Define common macros	
Generic build	(N/A)
Use legacy libc	☐
Fast floating-point math	☐

Figure 8: ELF binary format project settings for MPLABX

6 Memory Model and Linking

▶ *The content of this chapter is especially relevant when during building the application, the linker generates errors related to unreachable code suggesting the large code model.*

PIC24-dsPIC family members are 16 bit controllers. Branch instructions allow code within 32Kb from the branch instruction to be reached. Other instructions are offered in multiple formats. Calling a function for instance can be done using the `rcall` instruction or the `call` instruction. Using `rcall`, the same limitation as with branch instructions exist. Using `call`, allows any location in memory to be reached.

When compiling, either a small or a large memory model can be used. Using the small model, the compiler will use relative branches and calls with a 32Kb range. Using the large model the compiler uses instructions allowing the entire program memory to be reached. Which model is used depends on the size of the application. The disadvantage of the large model is that the compiler generated code is larger and slower.

AVIX is distributed in the form of a binary library. The AVIX library is build using the small memory model. Doing so, the generated code is smaller and faster. This model is compatible with application code regardless the model used when compiling the application. Based on the selected memory model, the application will contain ‘small’ or ‘large’ calls to the AVIX functions.

Functions present in the AVIX library do also use each other and since AVIX is compiled using the small memory model, the different AVIX functions must be no further apart from each other than 32Kb. Here a potential issue exists. The AVIX library contains multiple object files and in principle these individual files can be placed anywhere in program memory. The linker can place AVIX object files such that functions calling each other are out of reach, resulting in a linker error.

A solution for this would be to build the AVIX library using the large model. Since this does generate larger and slower code a different solution for this potential issue is chosen.

All compiler generated code lives in sections which can be given a name. For AVIX code, the section name ._avix_code_ is used. The linker allows all sections with the same name to be grouped. Since the total amount of AVIX code is something like 12Kb, when grouping all AVIX code sections together, all functions are guaranteed to be no further than 32Kb from each other, preventing the potential issue.

To accomplish this, the linker description file for the used controller must be added to the project. For every controller, such a linker description file exists (.gld file). Suggested is to copy the applicable file to the project directory and make the suggested changes to this file.

All linker description files contain a SECTION statement where directives can be added to control the application memory layout. To group all AVIX code together, the text shown in brown in Code sample 1 must be added to the linker description file SECTION statement.

```
SECTIONS             // Statement present in all .gld files
{
  …;
  ._avix_code_ :     /* Note: between ._avix_code_ and the colon a blank is present */
  {
    *(._avix_code_) /* Group all AVIX code                                     */
  }
  …;
}
```

Code sample 1: Group AVIX Code Sections to prevent Linker Errors

7 Configuring AVIX

AVIX is highly configurable. All configuration settings are made by manipulating the content of file AVIXSystemSettings.h. Details are provided in §7.1.

Besides the configuration settings, AVIXSystemSettings.h contains include statements for the global header files of the different controller families supported by this port. Based on the selected controller (see §5.1), the correct include file is automatically used. The applicable include files are shown in Table 2.

Controller family	Controller family include file
PIC24F	p24Fxxxx.h
PIC24E	p24Exxxx.h
PIC24H	p24Hxxxx.h
dsPIC30F	p30Fxxxx.h
dsPIC33E	p33Exxxx.h
dsPIC33F	p33Fxxxx.h

Table 2: Controller Family Header Files

When including one of the AVIX header files (preferably AVIX.h, see §4.1 for more details), AVIXSystemSettings.h is automatically included and as a result the applicable controller family include file is included. Therefore all definitions of the applicable controller family include file are available to the application and there is no need for the application to explicitly include one of these files.

7.1 Configuration Parameters

Configuring AVIX is done by manually editing the content of file AVIXSystemSettings.h. The following settings are present[5]:

avix_DEVICE_CLOCKhz

The controller's core speed and device speed are to be set by the application. Based on these settings, the speed at which the AVIX used hardware timer is running must be configured through this parameter. This parameter specifies the frequency in Hertz used for clocking the hardware timer configured through configuration parameter `avix_SYSTMR`.

From this value the applicable timing related settings used by AVIX are derived. An incorrect value for this parameter will lead to imprecise timing of timers and round robin time slice duration.

The value specified here is related to configuration parameter `avix_TIMER_CLK_SECONDARY`:

- When using an internal clock for the hardware timer, the value for `avix_DEVICE_CLOCKhz` must equal the instruction frequency the controller core is operating at.

- When using an external clock for the hardware timer, the value for `avix_DEVICE_CLOCKhz` must equal the frequency of this external clock which most of the time is 32,768Hz.

[5] Depending on the type of distribution, some configuration parameters are fixed. The applicable parameters are clearly marked in AVIXSystemSettings.h. Details are found in Table 3.

avix_TIMER_CLK_SECONDARY
Specify whether the hardware timer is clocked from an internal or an external clock. 0: Hardware timer is clocked from internal processor instruction clock. 1: Hardware timer is clocked from external clock. When using AVIX power management, certain power reduction modes require the system timer to be clocked from an external clock in order for the AVIX timing functions to continue operating. For more details see §11.
avix_ROUND_ROBIN_CYCLEus
The time in microseconds a thread is allowed to run before another thread at the same priority will be activated. A typical value for this parameter is 10,000 (10 ms). The value for this parameter must be at least five (5) times the value you choose for `avix_SYS_CLOCK_TICKus`. Optionally this parameter can be given the value 0 effectively disabling round robin scheduling. When doing so and using multiple threads at the same priority, such a thread must explicitly call `avixThread_Relinquish` to allow the next thread at the same priority to be activated.
avix_SYS_CLOCK_TICKus
Hardware timer period in microseconds. This parameter specifies the period of the AVIX system tick. This value must be at least five (5) times lower than the value chosen for `avix_ROUND_ROBIN_CYCLEus`. This parameter specifies the accuracy available for all timing related functionality. A typical value is between 100µs and 1000µs where the lower the number, the more accurate application timing can be obtained. When using AVIX power management, to place the controller in SLEEP mode, the timer period configured through this parameter may need to be substantially longer than the times mentioned above. For more details see §11. Based on the resolution of the clock used for the hardware timer, the actual period may differ from the period configured through this parameter. The actual hardware timer period is available through symbol `AVIX_SYS_CLOCK_ACTUAL_PERIOD`. When the application needs the hardware timer period, make sure to use this symbol i.s.o. `avix_SYS_CLOCK_TICKus`.
avix_MEM_POOL_COUNT
Maximum number of memory pools that can be created by the application. For every memory pool that can be created, two bytes are reserved. As such the value of this parameter has a marginal effect on RAM usage.

avix_MSG_POOL_NR_MESSAGES

Number of messages in the message pool available to the application.

When this parameter is given a value of zero (0), no message pool is created.

Based on this parameter, during initialization, a memory pool is created from which messages are allocated. Make sure to configure this parameter to the lowest possible value for the lowest memory consumption.

avix_MSG_BODY_NR_BYTES

The message mechanism is based on messages with a fixed size of the message data section. This parameter specifies the size in bytes of a message data section.

A value of zero (0) is allowed meaning no data can be placed in messages and only the message type field can be used to differentiate between messages.

avix_SYSTEM_STACK_SIZE

AVIX implements a system stack for use by Interrupt Service Routines. Using this system stack for interrupts means that stacks of individual threads can be smaller. This configuration parameter specifies the size in bytes allocated for the system stack. The configured size is scaled such that the stack size adheres to the stack alignment requirements for the applied microcontroller.

Besides ISR’s, this system stack is used by some AVIX internal functions.

A typical value for this parameter is 200 to 400 bytes.

avix_MAX_PRIORITY

The maximum thread priority an application can use. A thread can be given any priority from 1 up to and including `avix_MAX_PRIORITY`.

The maximum value for this configuration parameter is 254.

Adapt the value of this parameter to the actual priorities used by the application. A lower value for this parameter has a positive effect on both memory usage and performance.

avix_TRACING
Control Thread Activation Tracing. This parameter may have one of the following values: 0: Tracing **disabled**, trace code **executed**. On every context switch trace code is executed but no trace port assigned to any thread will be asserted. Instead of this a dummy memory location is asserted. This allows disabling tracing retaining the same performance as using value 1. 1: Tracing **enabled**, trace code **executed**. On every context switch, trace code is executed. For threads having a trace port assigned this port will be asserted. For threads not having a trace port assigned, a dummy memory location is asserted. This allows enabling tracing retaining the same performance as using value 0. 2: Tracing **disabled**, trace code **not executed**. Using this value, AVIX will not execute the trace code leading to optimal performance. Using this value AVIX executes a different instruction sequence than with either value 0 or 1. Values 0 and 1 exist to allow having a build with or without tracing having the same performance. These modes are intended to be used during development where switching between these mode enable/disable tracing without influencing performance. This forms the basis of the non-intrusiveness of tracing. Regardless if value 0 or 1 is used, the exact same instruction sequence will be executed. Regardless the value of this configuration parameter, trace ports can be assigned to a thread using function `avixThread_SetTracePort`. The value of this configuration parameter only influences how these ports are used by AVIX.
avix_EXTENDED_MEMORY
Number of bytes in the Extended Data Space (EDS) to allocate AVIX memory pools in. The value of this parameter must be in the range 0...65,532. More details can be found in §10. Note when using memory pools allocated in EDS RAM from multiple threads, this implies multi-threaded usage of the DSRPAG and DSWPAG registers. In this case these registers must be saved as part of the thread context and as a consequence flag `avix_MULTI_THREAD_PSV` must be set.

avix_MULTI_THREAD_PSV
Most controllers belonging to the PIC24-dsPIC families allow a FLASH page to be mapped to the upper data area (address 0x8000 to 0xffff) and contain a PSVPAG register to facilitate this. Some members of the PIC24F family and the PIC24E and dsPIC33E families support EDS RAM and allow either a FLASH page or a RAM page to be mapped to this upper data area. These controllers do not have a PSVPAG register but a pair of registers named DSRPAG and DSWPAG instead. AVIX is compatible with both types of controllers. Although the name of the configuration flag contains the term PSV, in practice this flag controls either PSVPAG or the DSRPAG/DSWPAG pair. When a controller is equipped with PSVPAG, the register is said to control PSV pages. When a controller is equipped with DSRPAG/DSWPAG, these registers are said to control EDS (Extended Data Space) pages. Control usage of multiple PSV/EDS pages from multiple threads. 0: PSV/EDS usage from multiple threads is not allowed. 1: PSV/EDS usage from multiple threads is allowed When using value 1, the PSVPAG register or the DSRPAG/DSWPAG register pair is saved as part of the thread context. As a consequence, two or four more bytes of stack space are used per thread and thread context switch time will be a few instructions longer. For this reason, when not using PSV/EDS pages from multiple threads it is advised to set this parameter to value 0. For more details see §9.1. Note when using EDS RAM based memory pools from multiple threads, this implies usage of the DSRPAG and DSWPAG registers and flag `avix_MULTI_THREAD_PSV` must be set.
avix_MULTI_THREAD_TBLPAG
Control usage of the table access instructions from multiple threads. 0: TBLPAG usage from multiple threads is not allowed. 1: TBLPAG usage from multiple threads is allowed When using value 1, the TBLPAG register is saved as part of the thread context. As a consequence, two more bytes of stack space are used per thread and thread context switch time will be a few instructions longer. For this reason, when not using table page access from multiple threads it is advised to set this parameter to value 0. For more details see §9.2.

avix_DSP_ENABLED
Control usage of DSP functionality. 0: DSP functionality may not be used 1: DSP functionality may be used by a single thread When using value 1, an additional register is saved as part of the thread context. As a consequence, two more bytes stack space are used per thread and thread context switch time will be a few instructions longer. For this reason, when not using DSP functionality it is advised to set this parameter to value 0. This parameter only has effect when using AVIX with a member of either the dsPIC30F, the dsPIC33E or the dsPIC33F family. When using controllers of the PIC24F, PIC24E or PIC24H families, this parameter is ignored. For more details see §9.3.
avix_SWI **avix_SWI_IRQ_REG** **avix_SWI_IRQ_CTRREG** **avix_SWI_IRQ_LVLREG**
Specify the internal interrupt and related registers used by AVIX. Any available interrupt can be chosen as long as it is not used by the application. For more details, see §7.2
avix_SYSTMR **avix_SYSTMR_IRQ_REG** **avix_SYSTMR_IRQ_CTRREG** **avix_SYSTMR_IRQ_LVLREG**
Specify the hardware timer and the related interrupt registers used by AVIX. Any available timer can be chosen as long as it is not needed by the application. For more details, see §7.2

Table 3: AVIX Configuration Parameters

7.2 Interrupt and Timer

Special care must be taken when configuring the interrupt and hardware timer used by AVIX.

For the interrupt this concerns the following configuration parameters:

- **`avix_SWI:`** The name of the selected interrupt.
- **`avix_SWI_IRQ_REG:`** The interrupt register belonging to the selected interrupt.
- **`avix_SWI_IRQ_CTRREG:`** The interrupt control register belonging to the selected interrupt.
- **`avix_SWI_LVLREG:`** The interrupt priority register belonging to the selected interrupt.

The hardware timer is configured by specifying its number. Internally the hardware timer is used on interrupt basis so for the hardware timer also the applicable interrupt registers must be configured. The following configuration parameters are applicable:

- **avix_SYSTMR:** The number of the selected hardware timer.
- **avix_SYSTMR_IRQ_REG:** The interrupt register belonging to the selected timer.
- **avix_SYSTMR_IRQ_CTRREG:** The interrupt control register belonging to the selected timer.
- **avix_SYSTMR_LVLREG:** The interrupt priority register belonging to the selected timer.

The PIC24-dsPIC interrupt structure is based on using three different registers for every interrupt:

- **Interrupt register (IF)**: Register containing the actual interrupt flag.
- **Interrupt control register (IE)**: Register containing a flag to enable or disable the related interrupt.
- **Interrupt priority register (IP)**: Register containing a three bit value specifying the priority of the interrupt.

When selecting an interrupt for use by AVIX, first select the name of the desired interrupt and assign this name to configuration parameter `avix_SWI`. Interrupt names can be found in the reference manual of the controller being used.

For example, using DMA2 as the interrupt this is done by the following entry in AVIXSystemSettings.h:

```
#define avix_SWI                  DMA2
```

Next the three related configuration parameters must be given a value such that they represent the interrupt register name, the interrupt control register name and the interrupt priority register name for the selected interrupt, DMA2 in this example.

This information can be found in the reference manual of the selected controller. Figure 9 shows this information for the dsPIC33FJ256GP710. Based on the selection of DMA2 as the interrupt to be used by AVIX, locate DMA2IF, DMA2IE and DMA2IP in the table which are the bit-fields belonging to the selected interrupts. In the table these values are marked red.

The registers to use for the configuration parameters are found in the leftmost column of this table in the row where the bit field is found. These register names are used for the three additional configuration parameters related to the selected interrupt. For DMA2IF, the register is IFS1, for DMA2IE the register is IEC1 and for DMA2IP, the register is IPC6.

This results in the following entries in AVIXSystemSettings.h:

```
#define avix_SWI_IRQ_REG       IFS1
#define avix_SWI_IRQ_CTRREG    IEC1
#define avix_SWI_IRQ_LVLREG    IPC6
```

Specifying these four entries conclude configuring AVIX to use DMA2 as its internal interrupt.

▶ *The selected interrupt is for exclusive use by AVIX and may not be used by the application. The device belonging to the selected interrupt is not used by AVIX and thus available to the application. In this case however it is important to guarantee the device itself does not generate interrupts. Failure to do so leads to an instable application.*

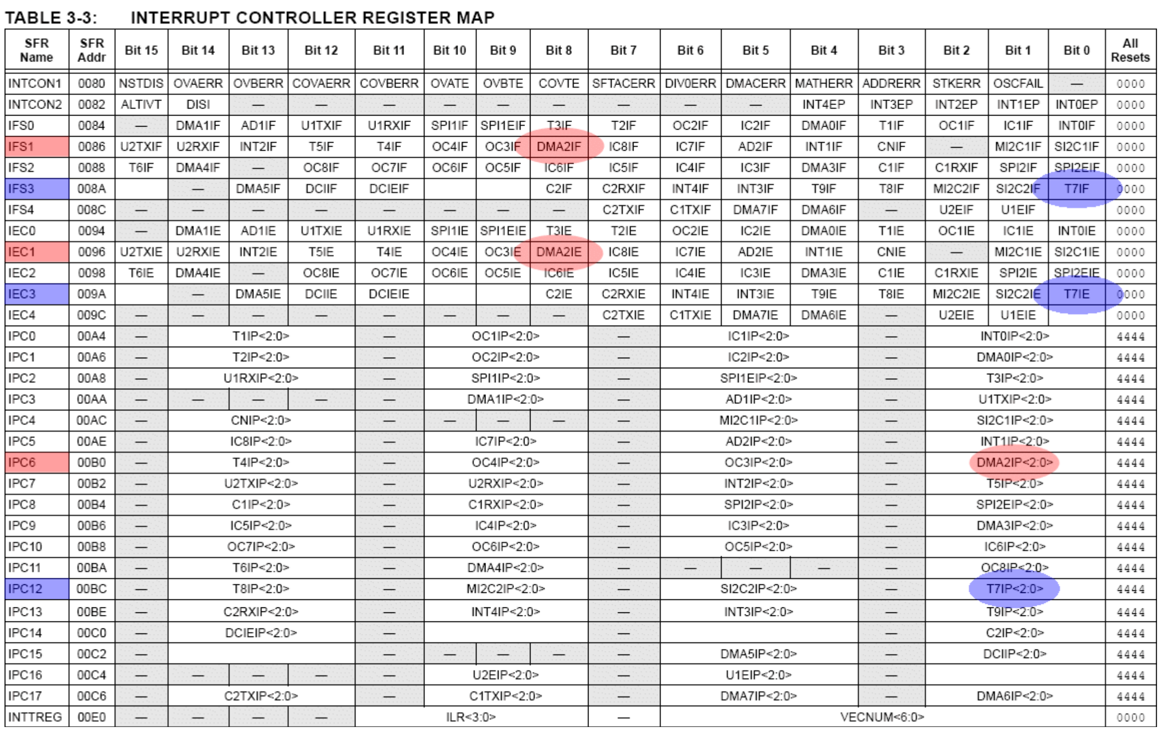


TABLE 3-3: INTERRUPT CONTROLLER REGISTER MAP

SFR Name	SFR Addr	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	All Resets
INTCON1	0080	NSTDIS	OVAERR	OVBERR	COVAERR	COVBERR	OVATE	OVBTE	COVTE	SFTACERR	DIV0ERR	DMACERR	MATHERR	ADDRERR	STKERR	OSCFAIL	—	0000
INTCON2	0082	ALTIVT	DISI	—	—	—	—	—	—	—	—	—	INT4EP	INT3EP	INT2EP	INT1EP	INT0EP	0000
IFS0	0084	—	DMA1IF	AD1IF	U1TXIF	U1RXIF	SPI1IF	SPI1EIF	T3IF	T2IF	OC2IF	IC2IF	DMA0IF	T1IF	OC1IF	IC1IF	INT0IF	0000
IFS1	0086	U2TXIF	U2RXIF	INT2IF	T5IF	T4IF	OC4IF	OC3IF	DMA2IF	IC8IF	IC7IF	AD2IF	INT1IF	CNIF	—	MI2C1IF	SI2C1IF	0000
IFS2	0088	T6IF	DMA4IF	—	OC8IF	OC7IF	OC6IF	OC5IF	IC6IF	IC5IF	IC4IF	IC3IF	DMA3IF	C1IF	C1RXIF	SPI2IF	SPI2EIF	0000
IFS3	008A		—	DMA5IF	DCIIF	DCIEIF			C2IF	C2RXIF	INT4IF	INT3IF	T9IF	T8IF	MI2C2IF	SI2C2IF	T7IF	0000
IFS4	008C	—	—	—	—	—	—	—	—	C2TXIF	C1TXIF	DMA7IF	DMA6IF	—	U2EIF	U1EIF		0000
IEC0	0094	—	DMA1IE	AD1IE	U1TXIE	U1RXIE	SPI1IE	SPI1EIE	T3IE	T2IE	OC2IE	IC2IE	DMA0IE	T1IE	OC1IE	IC1IE	INT0IE	0000
IEC1	0096	U2TXIE	U2RXIE	INT2IE	T5IE	T4IE	OC4IE	OC3IE	DMA2IE	IC8IE	IC7IE	AD2IE	INT1IE	CNIE	—	MI2C1IE	SI2C1IE	0000
IEC2	0098	T6IE	DMA4IE	—	OC8IE	OC7IE	OC6IE	OC5IE	IC6IE	IC5IE	IC4IE	IC3IE	DMA3IE	C1IE	C1RXIE	SPI2IE	SPI2EIE	0000
IEC3	009A		—	DMA5IE	DCIIE	DCIEIE			C2IE	C2RXIE	INT4IE	INT3IE	T9IE	T8IE	MI2C2IE	SI2C2IE	T7IE	0000
IEC4	009C	—	—	—	—	—	—	—	—	C2TXIE	C1TXIE	DMA7IE	DMA6IE	—	U2EIE	U1EIE		0000
IPC0	00A4	—	T1IP<2:0>			—	OC1IP<2:0>			—	IC1IP<2:0>			—	INT0IP<2:0>			4444
IPC1	00A6	—	T2IP<2:0>			—	OC2IP<2:0>			—	IC2IP<2:0>			—	DMA0IP<2:0>			4444
IPC2	00A8	—	U1RXIP<2:0>			—	SPI1IP<2:0>			—	SPI1EIP<2:0>			—	T3IP<2:0>			4444
IPC3	00AA	—	—	—	—	—	DMA1IP<2:0>			—	AD1IP<2:0>			—	U1TXIP<2:0>			4444
IPC4	00AC	—	CNIP<2:0>			—	—	—	—	—	MI2C1IP<2:0>			—	SI2C1IP<2:0>			4444
IPC5	00AE	—	IC8IP<2:0>			—	IC7IP<2:0>			—	AD2IP<2:0>			—	INT1IP<2:0>			4444
IPC6	00B0	—	T4IP<2:0>			—	OC4IP<2:0>			—	OC3IP<2:0>			—	DMA2IP<2:0>			4444
IPC7	00B2	—	U2TXIP<2:0>			—	U2RXIP<2:0>			—	INT2IP<2:0>			—	T5IP<2:0>			4444
IPC8	00B4	—	C1IP<2:0>			—	C1RXIP<2:0>			—	SPI2IP<2:0>			—	SPI2EIP<2:0>			4444
IPC9	00B6	—	IC5IP<2:0>			—	IC4IP<2:0>			—	IC3IP<2:0>			—	DMA3IP<2:0>			4444
IPC10	00B8	—	OC7IP<2:0>			—	OC6IP<2:0>			—	OC5IP<2:0>			—	IC6IP<2:0>			4444
IPC11	00BA	—	T6IP<2:0>			—	DMA4IP<2:0>			—	—	—	—	—	OC8IP<2:0>			4444
IPC12	00BC	—	T8IP<2:0>			—	MI2C2IP<2:0>			—	SI2C2IP<2:0>			—	T7IP<2:0>			4444
IPC13	00BE	—	C2RXIP<2:0>			—	INT4IP<2:0>			—	INT3IP<2:0>			—	T9IP<2:0>			4444
IPC14	00C0	—	DCIEIP<2:0>			—				—				—	C2IP<2:0>			4444
IPC15	00C2	—				—	—	—	—	—	DMA5IP<2:0>			—	DCIIP<2:0>			4444
IPC16	00C4	—	—	—	—	—	U2EIP<2:0>			—	U1EIP<2:0>			—				4444
IPC17	00C6	—	C2TXIP<2:0>			—	C1TXIP<2:0>			—	DMA7IP<2:0>			—	DMA6IP<2:0>			4444
INTTREG	00E0	—	—	—	—	ILR<3:0>				—	VECNUM<6:0>							0000

Figure 9: AVIX Sample Interrupt Controller Register Map

▶ *The table shown in Figure 9 contains the registers for the dsPIC33FJ256GP710. Make sure to use the table belonging to the controller you are using since the applicable registers differ per controller type.*

The same process must be followed for the hardware timer required by AVIX. First you choose the number of the desired timer and make this number the value of `#define avix_SYSTMR`. ***Note that this is just a number without a T***. When using timer 7, the configuration parameter looks like:

```
#define avix_SYSTMR                 7
```

For every timer, there is a TxIF, TxIE and TxIP register, respectively containing the interrupt register, the interrupt control register and the interrupt priority register for that timer. Since in this example timer 7 is used, locate T7IF, T7IE and T7IP in the Interrupt Controller Register table.

In the table above these entries are marked blue. Next the corresponding registers found in the leftmost column are entered in AVIXSystemSettings.h. This leads to:

```
#define avix_SYSTMR_IRQ_REG       IFS3
#define avix_SYSTMR_IRQ_CTRREG    IEC3
#define avix_SYSTMR_IRQ_LVLREG    IPC12
```

▶

▶ *Correctly specifying these values is essential for AVIX to operate correctly. Make sure after editing the selected values in AVIXSystemSettings.h to check them once more.*

7.3 Distributions

AVIX is available in three different retail distributions. These distributions differ in user configurable parameters and the number of kernel resources (thread, mutex, etc.) that can be used. Every distribution comes with an AVIXSystemsSettings.h file, containing all configuration parameters mentioned in §7. Based on the type of distribution, a configuration parameter is either fixed (as determined by the build process of that distribution) or user changeable. Fixed configuration parameters are clearly denoted as such in AVIXSystemsSettings.h.

Table 4 shows a summary of the fixed configuration parameters and their values together with the maximum number of kernel objects that can be used for each type of distribution.

Distribution	BASIC	STANDARD	EXTENDED
Controller Type	Configurable		
Device Speed	Configurable		
Round Robin Cycle Time	Configurable		
System Clock Period	Configurable		
Internal Interrupt	Configurable		
Hardware Timer	Configurable		
System Stack size	🟨 200	🟨 300	Configurable
Maximum Priority	🟨 8	🟨 16	Configurable
Message Body Size	🟨 4	🟨 4	Configurable
Nr. Messages in Message Pool	🟨 10	🟨 20	Configurable
Maximum number of Memory Pools	🟨 2	🟨 4	Configurable
Maximum number of Threads	🟦 8	🟦 16	Memory dependent
Maximum number of Mutexes	🟦 4	🟦 8	Memory dependent
Maximum number of Semaphores	🟦 4	🟦 8	Memory dependent
Maximum number of Event Groups	🟦 4	🟦 8	Memory dependent
Maximum number of Pipes	🟦 4	🟦 8	Memory dependent
Maximum number of Timers	🟦 4	🟦 8	Memory dependent
Maximum number of Exchange Objects	🟦 4	🟦 8	Memory dependent

🟨 Fixed configuration parameter. Value present in AVIXSystemSettings.h for reference purposes. Changing has no effect.
🟦 Fixed value in Distribution Library code

Table 4: AVIX Distribution Capabilities

The retail version of AVIX is distributed as a binary library. This library is controller neutral and can be used with any controller of the families targeted by AVIX.

8 Stack usage and Interrupt Service Routines

Controllers belonging to the PIC24F, PIC24E, PIC24H, dsPIC30F, dsPIC33E and dsPIC33F families do not implement a hardware system stack for use by Interrupt Service Routines (ISR's).

To reduce memory usage, AVIX does implement a software system stack

ISR's can be declared using the standard compiler mechanism. Doing so however, the system stack is not used and interrupts use the stack of the then active thread.

An example of a basic ISR declaration is shown in Code sample 2.

```c
// ISR declaration for timer 4 based on the regular C30/XC16 compiler suite mechanism
//
void __attribute__((__interrupt__, no_auto_psv)) _T4Interrupt()
{
  …;                     // Application specific ISR code
  …;

  IFS1bits.T4IF = 0;    // Reset the interrupt flag.
}
```

Code sample 2: How to declare an ISR using the compiler syntax

Alternatively ISR's can be declared using AVIX provided macros. ISR's using these macros make minimal use of the stack of the active thread and will mainly use the software system stack.

Using the software system stack for ISR's leads to a significant reduction of RAM usage, at the cost of five additional instruction cycles consumed by the ISR.

ISR's based on the regular compiler mechanism and ISR's based on the AVIX provided macros may be used together. So for every individual ISR a choice can be made whether reduction of RAM usage or ultimate performance is the most important.

The macros to declare an ISR using the AVIX system stack are shown below.

`avixDeclareISR(isrName, psvUsage)`
This macro defines an ISR that will use the AVIX system stack. As a result, the ISR only uses six bytes of the stack of the interrupted thread. Local variables and other stack usage of the ISR are placed on the system stack. The macro must be followed by 'C' style curly brackets {}, in between which the code of the ISR is present. Effectively this defines a 'C' style function for the ISR. The psvUsage parameter defines how the ISR deals with PSV. The value of this parameter is either auto_psv or no_auto_psv. The semantics of this parameter are identical to the semantics for these values as described in the C30/XC16 compiler suite Users Guide.

`avixDeclareISRShadow(isrName, psvUsage)`
This macro defines an ISR that will use the AVIX system stack. As a result, the ISR only uses six bytes of the stack of the interrupted thread. Local variables and other stack usage of the ISR are placed on the system stack. The macro must be followed by 'C' style curly brackets {}, in between which the code of the ISR is present. The ISR declared by this macro uses the controllers shadow register set. The psvUsage parameter defines how the ISR deals with PSV. The value of this parameter is either auto_psv or no_auto_psv. The semantics of this parameter are identical to the semantics for these values as described in the C30/XC16 compiler suite Users Guide.

Multiple ISR's can be declared with this macro. In this case it is important these ISR's all have the same priority since the underlying hardware platform only offers one set of shadow registers. Failure to do so will lead to an unstable application.

▶ *Make sure when using* `avixDeclareISRShadow` *for multiple ISR's, each of the interrupts these ISR's belong to have the same priority Failure to do so will lead to an unstable application.*

An example of an AVIX macro based ISR declaration is shown in Code sample 2.

```
// Interrupt Service Routine declaration for timer 4 based on the AVIX mechanism
//
avixDeclareISR(_T4Interrupt, no_auto_psv)
{
  …;                     // Application specific ISR code
  …;

  IFS1bits.T4IF = 0;    // Reset the interrupt flag.
}
```

Code sample 3: How to declare an ISR using the AVIX software system stack syntax

An ISR declared using one of the AVIX macros uses six (6) bytes of the stack of the interrupted thread before switching to the software system stack. For the remaining ISR stack requirement, use is made of the software system stack. The hardware platform supports nested interrupts with seven (7) possible priority levels. As a result, the worst case thread stack usage is six times seven is 42 bytes in case all possible ISR priority levels are used.

When specifying the threads stack size, make sure to reserve the above mentioned ISR stack requirement. This figure is only applicable when all ISR's are based on the AVIX provided mechanism. When also using ISR's based on the standard compiler mechanism, the thread stack usage cannot be predicted and the above figure is not applicable.

How much RAM is preserved now when using the software system stack?

An ISR saves a minimum of 26 bytes on the stack of the interrupted thread. Using all possible interrupt priorities, this implies each thread has to preserve 26 * 7 equaling 182 bytes. For an average application using 15 threads, this equals 2,730 bytes.

Using the AVIX mechanism, only 6 of these 26 bytes are placed on the stack of the interrupted thread. Multiplied with 7 priority levels and 15 threads, this equals ~630 bytes. The remaining ISR stack requirement (20 bytes) is placed on the software system stack. This figure times 7 priority levels equals 140 bytes. In total, when using the AVIX software system stack, 770 bytes are used.

This implies that using the AVIX software system stack mechanism, RAM usage is lowered by as much as ~2,000 bytes for this particular example.

In practice more RAM is preserved since the above calculation is based on the minimal ISR stack requirement. When using ISR local variables and/or allow ISR's to make function calls, stack requirement will increase and thus the positive effect of using the AVIX software system stack will be even higher.

9 Controller Specific Capabilities

9.1 Multi-Thread PSV/Extended Data Space usage

▶ *This section contains information on compatibility between AVIX and usage of the controllers PSV or DSRPAG/DSWPAG registers. The content of this section is important when these registers are used from more than one thread.*

Controllers belonging to the PIC24-dsPIC families allow a 32KB page of the FLASH memory to be mapped to the upper 32KB of the data area. This page is used to access constant data values. The number of the page is stored in the PSVPAG register. This register is given an initial value by the runtime linked with your application. For more details on how to use this mechanism see the applicable Microchip documentation.

Some members of the PIC24F family and controllers belonging to the PIC24E or dsPIC33E families, not only allow a FLASH page to be mapped to the upper 32KB of the data area but also offer the choice of mapping a RAM page. These controllers do not have a PSVPAG register but a pair of registers named DSRPAG and DSWPAG instead. When mapping a RAM page to the upper 32KB data area, these registers are used to read (DSRPAG) or write (DSWPAG). When mapping a FLASH page to the upper 32KB data area, it shall be obvious this can only be read which in this case is accomplished through DSRPAG. For this mechanism too, more details can be found in the applicable Microchip documentation.

AVIX is compatible with both types of controllers.

Selecting what type of memory and which page of this memory is mapped to the upper 32KB data area is the responsibility of the application. The application does so by manipulating either PSVPAG or DSRPAG/DSWPAG, depending on the applied microcontroller.

AVIX allows these registers to be used by multiple threads. In which case, each thread needs a private copy of the content of these registers. This can be accomplished by setting configuration parameter `avix_MULTI_THREAD_PSV` to the value 1. This value instructs AVIX to save the mentioned register(s) as part of the threads context. When the thread is pre-empted by another thread, these registers are saved and when the thread is activated again, the content of these registers is restored. Effectively the thread will not notice that during the time it was pre-empted, these registers have been used by another thread.

In case these registers are used from a single thread only, no other thread will change their content and the mentioned registers do not need to be saved as part of the threads context. This is accomplished by setting configuration parameter `avix_MULTI_THREAD_PSV` to the value 0.

The initial value of PSVPAG or DSRPAG/DSWPAG is determined by the C30/XC16 compiler suite runtime. When `avix_MULTI_THREAD_PSV` is set to 1 and a thread starts, AVIX takes care these registers contain this initial value. So when one thread has been running and manipulated the content of these registers, when another thread starts, this thread will receive as the initial value of these registers the value determined by the C30/XC16 compiler suite runtime.

The value of `avix_MULTI_THREAD_PSV` has no effect on the behavior of ISR's. For ISR's, a C30/XC16 compiler suite supplied mechanism must be used. ISR's can be declared with a flag to specify whether to save these registers (`auto_psv`) or not save these registers (`no_auto_psv`) as part of the ISR saved context. An ISR only needs to save the content of these registers if the ISR itself manipulates their content. So when using these registers from multiple threads, it is not said that ISR's must use `auto_psv`.

▶ *When using multiple PSV or EDS pages, be very careful when passing addresses (pointers) of variables from one thread to another using a mechanism like messages or pipes. When sending a 'plain' 16-bit pointer. The thread receiving such a pointer does not know which PSV or EDS page this pointer belongs to. The receiving thread is likely to have its own, local value of PSVPAG or DSRPAG/DSWPAG and using such a pointer will add the pointer as an offset to the thread local value of these registers which is not the same as the value used to obtain the pointer in the sending thread. This will lead to incorrect results. A means to prevent such problems is to use Managed Pointers or Extended Data Space pointers. See the C30/XC16 compiler suite reference manual for details. Alternatively, instead of passing pointers, the desired values can be passed by value by not writing the pointer in the message or the pipe but the value pointed to.*

9.2 Table Page (TBLPAG) usage

Controllers targeted by AVIX allow programmed access to FLASH memory through the TBL… instructions which use the TBLPAG register to form the upper 8 bits of a 24 bit address. The TBLPAG register is explicitly manipulated by the software and every thread requiring access to FLASH memory through this mechanism will have to do so.

If multiple threads use this mechanism and a thread is preempted while using TBLPAG, once the thread is resumed, the value in the TBLPAG register can be changed by another thread. Under this condition, the value in the TBLPAG register can be said to belong to a thread and thus must be part of the context which is saved on thread preemption and restored on thread reactivation.

Setting `avix_MULTI_THREAD_TBLPAG` to 1 takes care of this. The result of this setting is that on preemption, the content of the TBLPAG register is saved as part of the preempted threads context and when resumed, the saved value is written back to the TBLPAG register. As a result, a thread will not notice it has been preempted and on resumption can continue as if nothing has changed (which is indeed the case).

When this mechanism is used from a single thread only, no other thread will manipulate the TBLPAG register and even when the thread is preempted, the value of this register will not change. In this case `avix_MULTI_THREAD_TBLPAG` may be set 0.

Setting `avix_MULTI_THREAD_TBLPAG` to the value 1 will add two bytes to the stack usage of each thread and add two cycles to the context switch time.

9.3 DSP Functionality

The 16-bit microcontrollers targeted by AVIX are divided in two sub-families, the PIC24 range, being a pure microcontroller family and the dsPIC3x range, besides all functionality of the PIC24 range, in addition offering DSP functionality. From a machine instruction point of view, the PIC24 range is a pure subset of the dsPIC3x range. All software functioning on a PIC24 can potentially operate on the dsPIC3x range, provided the hardware configuration is adapted to the specific type of controller.

The major difference is that the dsPIC3x range offers DSP capabilities which are not found on the PIC24 range. This DSP functionality consists of additional controller internal registers, instructions and addressing modes. AVIX is compatible with the DSP functionality of the dsPIC3x range of microcontrollers. This chapter gives an overview how to use DSP functionality in your AVIX based application.

▶ *The content of this section is only applicable to controllers belonging to the dsPIC3x range. The applicable configuration parameter has no effect when using the PIC24 range.*

9.3.1 How to make AVIX DSP aware

When using DSP functionality, configuration parameter `avix_DSP_ENABLED` must be set to value 1. When not using DSP functionality, parameter `avix_DSP_ENABLED` must be set to value 0.

► *Using DSP functionality with configuration setting `avix_DSP_ENABLED` having value 0 will lead to a system crash.*

Setting `avix_DSP_ENABLED` to the value 1 will add two bytes to the stack usage of each thread and add four cycles to the context switch time.

9.3.2 DSP Details

► *When a multi-threaded AVIX based application uses DSP functionality, this functionality may be used from a single thread only.*

DSP functionality may be used from a single thread only. AVIX cannot guard this and therefore it is the user's responsibility to assure this. Violating this rule will lead to system failure. This rule is imposed by hardware restrictions present in dsPIC30F and dsPIC33F controllers. A number of bits in the DSP specific registers are read-only and therefore it is not possible to save and restore these registers as part of the thread context[6].

As long as DSP functionality and the related registers are used from a single thread only, no other thread will corrupt the contents of those registers and the 'DSP' thread can be assured the content of the registers will be as expected.

► *The fact that DSP functionality is allowed to be used from one thread only must be taken literally, implying that it is neither allowed to use DSP functionality from Interrupt Service Routines or Deferred Interrupt Handlers.*

In practice this does not pose a problem since most applications will contain a single DSP based algorithm making it natural to implement this in a single thread. Under the rare circumstance a design would ask for multiple threads to use DSP functionality the solution for this is easy. Just add a DSP server thread to the application, a thread responsible for the different DSP calculations. This thread can be considered a DSP server where other threads request it to perform the desired operations by sending a message with the data the DSP thread should operate on. This data can for instance be stored in a memory block, referenced from a message block id in the message or by writing the data to an Exchange object monitored by the thread containing the DSP functionality.

The advantage of this approach is that AVIX with or without using DSP functionality is almost equally fast. Besides the aforementioned additional four cycles added to the context switch time, configuring AVIX to allow the use of DSP functionality has a very small impact on interrupt handler latency which is caused by the following;

The DSP functionality allows modulo and bit reversed indirect register addressing. For details see the applicable controller reference manual. These types of addressing use the standard controllers registers in a DSP specific way not compatible with the regular addressing these registers are normally used for. AVIX is a preemptive operating system meaning it is unpredictable when a thread is interrupted. Likewise when using plain interrupts, being one of the key features of AVIX.

[6] The mentioned hardware restriction does not exist for dsPIC33E microcontrollers. For this family it is possible to save and restore all DSP specific registers allowing DSP functionality to be used by multiple threads. Doing so would however have severe impact on the thread context switch time for all threads, regardless whether they use DSP functionality or not. For this reason, AVIX does not implement this and when using a dsPIC33E microcontroller, still DSP functionality may be used from a single thread only.

As a result, when an Interrupt Service Routine starts, be it AVIX internal or in your application this can happen while the DSP thread has modulo or bit reverse addressing active. Without precautions this would lead to a crash of the interrupt code. To deal with this AVIX takes the approach of disabling those DSP addressing modes when entering an Interrupt Service Routine. When the Interrupt Service Routine is ready, the state of these addressing modes is restored to the state when the Interrupt Service Routine was activated.

With AVIX, when creating Interrupt Service Routines, you can chose to use the AVIX supplied macros, `avixDeclareISR` or `avixDeclareISRShadow` described in §8 or the regular C30/XC16 compiler suite interrupt syntax. These AVIX macros take care of the above automatically. When using the plain C30/XC16 compiler suite syntax, no support for the above is present and you have to provide this yourself.

► *Using the native C30/XC16 compiler suite style interrupt syntax, an interrupt occurring while modulo or bit reverse addressing is active might lead to a system crash since default C30/XC16 compiler suite interrupt handlers are DSP unaware. Use the AVIX supplied interrupt declaration macros to prevent this.*

10 Using memory in the Extended Data Space

For microcontrollers offering additional RAM in the form of Extended Data Space (EDS) RAM, AVIX allows memory pools to be allocated in this RAM. This chapter contains details on the applicable mechanisms and how to use them.

► *The content of this chapter is only relevant when using a controller with EDS RAM above the 32K boundary, and you want to use (part of) this RAM for memory pools managed by AVIX. If so, you might want to do the following:*

1. Reserve the desired amount of EDS RAM to be used by AVIX for memory pools by assigning the desired amount of bytes to configuration parameter `avix_EXTENDED_MEMORY`.

2. Use function `avixMemPool_CreateExt` to create memory pools in EDS ram. To create memory pools in the regular AVIX Free Store, continue to use `avixMemPool_Create`. The type of memory to use can be decided per memory pool and both types of pool can be used together.

3. Use macro `AVIX_MEM_BLOCK_PTR_EXT` to convert memory blocks allocated from an EDS RAM based memory pool to a 'C' pointer. Memory blocks created from regular pools using `avixMemPool_Create` continue to be converted to a 'C' pointer using macro `AVIX_MEM_BLOCK_PTR`.

4. Tag 'C' pointers used for access to memory blocks from an EDS RAM based pool with the C30/XC16 compiler suite provided `__eds__` attribute or use macro `AVIX_EDS` instead. More details about this macro are found in §10.2.

All other AVIX memory pool functions work with pools allocated either in the regular AVIX Free Store or allocated in EDS RAM. Allocating a memory block from a pool is always done using function `avixMemPool_Allocate`. This function receives an id of the memory pool and AVIX knows whether this id refers a memory pool allocated in the AVIX free store or in EDS RAM. Likewise returning a memory block to its pool for this block to be reused is done using function `avixMemPool_Free`, again regardless whether the memory block id passed to this function is the id of a memory block allocated in the AVIX Free store or in EDS RAM.

Although AVIX goes a long way in hiding the differences between using memory blocks allocated from the AVIX Free Store or from EDS RAM, there is one noticeable difference that is impossible to hide, a difference you, as a user will be aware of and must manage carefully. This difference is that pointers to blocks allocated from the AVIX Free Store are 16-bit in size and pointers to blocks allocated in EDS RAM are 32-bit in size. This difference is caused by the fact that the PIC24-dsPIC architecture is 16 bit.

Another difference is related to performance. Access to blocks allocated from EDS RAM is substantially slower than access to blocks allocated from the AVIX Free Store. Again this is caused by the fact that PIC24-dsPIC is a 16-bit architecture and the C30/XC16 compiler suite generates more code to manipulate pointers to EDS RAM.

Still AVIX, together with the C30/XC16 compiler suite, goes a long way in hiding the functional differences and the 'C' sources are highly portable between controllers that do and controllers that don't offer EDS RAM. More details about this are found in §10.2.

There are two reasons you might want to use the AVIX EDS RAM support. First, since you just want to use the extra RAM. Second, since you might want your code to be prepared for porting to a controller offering the extra RAM.

10.1 Basic Usage

Like in earlier versions of AVIX, memory pools still can be allocated in the AVIX Free Store using function `avixMemPool_Create`. Optionally, memory pools can be allocated in EDS RAM using function `avixMemPool_CreateExt`. Code sample 4 below shows the two alternative methods to create a memory pool, allocate a memory block from this pool and finally convert the memory block id to a pointer that can be used to directly access the RAM.

The differences between the two approaches are colored brown. The most noticeable difference is shown on line 15, where the pointer used to access the ram is tagged with the `__eds__` attribute making it a 32-bit pointer.

```
 1 // Code fragment showing how to create a regular memory pool and use one of its blocks
 2 //
 3 tavixMemPoolId  regMemPool;
 4 tavixMemBlockId regMemBlock;
 5 char*           pRegBlock;
 6 …;
 7 regMemPool  = avixMemPool_Create(NULL, 10, 20);
 8 regMemBlock = avixMemPool_Allocate(regMemPool);
 9 pRegBlock   = AVIX_MEM_BLOCK_PTR(regMemBlock);
10
11 // Code fragment showing how to create an EDS memory pool and use one of its blocks
12 //
13 tavixMemPoolId  extMemPool;
14 tavixMemBlockId extMemBlock;
15 __eds__ char*   pExtBlock;
16 …;
17 extMemPool  = avixMemPool_CreateExt(NULL, 10, 20);
18 extMemBlock = avixMemPool_Allocate(extMemPool);
19 pExtBlock   = AVIX_MEM_BLOCK_PTR_EXT(extMemBlock);
```

Code sample 4: Using a memory pool in base or EDS memory

10.2 Portability

AVIX goes a long way in offering a portable mechanism that builds on all supported controllers, regardless whether the controller does or does not contain EDS RAM.

In order for EDS RAM to be used the following three conditions must be met:

1. Use must be made of version 3.20 or later[7] of the C30/XC16 compiler suite when using a PIC24F microcontroller with EDS RAM or version 3.25 or later of the C30/XC16 compiler suite when using a PIC24E or dsPIC33E microcontroller.

2. And... Use must be made of a controller implementing EDS RAM. EDS RAM is offered by a small number of PIC24F controllers and by the PIC24E and dsPIC33E controllers.

3. And..., a section of EDS RAM must be reserved for use by AVIX: Reserving a section of EDS RAM is done through configuration parameter `avix_EXTENDED_MEMORY`. This parameter specifies the number of bytes reserved by AVIX.

If not all three conditions are met, function `avixMemPool_CreateExt` behaves the same as `avixMemPool_Create`, implying the Memory Pool is created in the AVIX Free Store. Also macro `AVIX_MEM_BLOCK_PTR_EXT` behaves the same as `AVIX_MEM_BLOCK_PTR`, implying it returns a regular 16-bit pointer to the memory block.

[7] The mentioned version number refers to the C30 compiler toolsuite. The version number of its successor, the XC16 compiler toolsuite starts with version 1.0 again. Any version of the XC16 compiler toolsuite is implicitly more recent than any version of the C30 compiler toolsuite.

Finally there is the issue of pointers that are tagged with the `__eds__` attribute. Pointers tagged with this attribute will also work on controllers not containing EDS RAM. The pointer is still 32-bit in size and access is slower but the C30/XC16 compiler suite will hide the differences and the code will compile and run. For this reason AVIX offers a pointer tagging macro that can be used instead of the plain `__eds__` attribute. This macro, `AVIX_EDS`, will translate to __eds__ when all three conditions are met and it will be empty when one or more of these conditions is not met. The advantage of using this macro instead of the plain `__eds__` attribute actually is twofold: First, when not using EDS RAM, the size of a pointer tagged with this macro will be 16-bit again, leading to an optimal performance. Second, code using this macro will also build using versions of the C30/XC16 compiler suite older than version 3.20[8].

Use of this macro is shown on line 5 in Code sample 5.

```
1 // Code fragment showing how to tag an EDS pointer in a portable fashion
2 //
3 tavixMemPoolId  extMemPool;
4 tavixMemBlockId extMemBlock;
5 AVIX_EDS char*  pExtBlock;
6 …;
7 extMemPool   = avixMemPool_CreateExt(NULL, 10, 20);
8 extMemBlock  = avixMemPool_Allocate(extMemPool);
9 pExtBlock    = AVIX_MEM_BLOCK_PTR_EXT(extMemBlock);
```

Code sample 5: Portable EDS pointer tagging

As a user of EDS RAM you must be aware what type of RAM pointers refer and tag these pointers accordingly.

Tagging a pointer to regular RAM will not fail. In this case, a 32-bit pointer is used where a 16-bit pointer would be sufficient, performance will not be optimal but everything works as expected.

Forgetting to tag a pointer which will be used for EDS RAM however is more dramatic. In this case a 16-bit pointer will be used where a 32-bit pointer is required and this will result in access to wrong memory locations and probably a failing system. Luckally, this situation is detected by the C30/XC16 compiler suite which will issue the following warning:

warning: assignment discards qualifiers from pointer target type

When this warning is issued, do not ignore it but solve the problem by applying the correct pointer tagging.

10.3 Extended Data Space background

Controllers belonging to the PIC24-dsPIC families implement a 16-bit data address. In principle this allows for a maximum of 64KB of data to be directly accessible. The lower 32KB (addresses 0x0000 to 0x7FFF) contains the SFR's and fixed RAM. The upper 32KB (addresses 0x8000 to 0xFFFF) contains a variable page of FLASH or RAM which is mapped to this area.

Most controllers belonging to the PIC24-dsPIC families offer an amount of RAM that fits entirely in the lower 32KB address range. This RAM is accessed using addresses in the range 0x0000-0x7FFF. When allocating AVIX memory pools through function `avixMemPool_Create`, these pools are allocated in this RAM.

[8] The mentioned version number refers to the C30 compiler toolsuite. The version number of its successor, the XC16 compiler toolsuite starts with version 1.0 again. Any version of the XC16 compiler toolsuite is implicitly more recent than any version of the C30 compiler toolsuite.

Controllers offering EDS RAM allow RAM to be mapped to the upper 32KB address range. Although the total amount of RAM can be much more than 32KB, the controllers mapping mechanism allows a maximum of 32KB of this RAM to be ‘visible’ in the upper 32KB data area. For this purpose, these controllers implement two registers, DSRPAG and DSWPAG containing the page number of the mapped EDS RAM page. More details can be found in the controller’s reference manual.

Although EDS is based on using the mentioned paging mechanism, code generated by the C30/XC16 compiler suite fully hides this and presents the total amount of RAM as one contiguous area. Access to EDS requires a pointer qualified with the `__eds__` attribute. Such a pointer requires 32 bit storage compared to a basic data pointer which requires 16 bit.

When defining EDS variables, these are allocated starting at the highest available physical RAM address, growing to lower addresses. These variables are allowed to cross page boundaries and may even grow into page 0, the lower 32KB of RAM. So the boundary between base RAM and EDS RAM does not depend on the physical location of the RAM but on the amount of EDS variables defined only.

This is illustrated by controllers that do offer the EDS mechanism but still only offer a total amount of physical RAM that lies well below the 32KB boundary. For these controllers too, EDS variables may be defined although such variables will always be located below the 32KB boundary. Since for these controllers the RAM can also be accessed using a 16-bit pointer, defining EDS variables does not make much sense.

► *When using EDS RAM, make sure the total amount is such that the related physical memory lies above the 0x7FFF address boundary. Allocating such an amount that part of the address range of 0x0000 to 0x7FFF is also used has two drawbacks. First, access to memory below address 0x7FFF will be slower since it is accessed using 32 bit pointers. Second, less RAM is available for regular variables and AVIX Free Store.*

11 Power Management

This section specifies the AVIX power management implementation aspects specific to the PIC24E/F/H, dsPIC30F and dsPIC33E/F controllers.

► *When using AVIX power management, detailed knowledge of the controller specific power reduction features is required. Although this section provides some controller specific power reduction information, this manual is no substitute for the controller's data sheet.*

11.1 Power mode mapping

The AVIX generic power mode constants have the following mapping to the controller specific power modes:

- `AVIX_POWER_REDUCTION_NONE`: No mapping to a controller specific power mode. When using this mode, no controller power reduction mode is used.

- `AVIX_POWER_REDUCTION_LOW`: Maps to the controller IDLE mode. For a description of the IDLE mode consult the controller's data sheet.

- `AVIX_POWER_REDUCTION_HIGH`: Maps to the controller SLEEP mode. For a description of the SLEEP mode consult the controller's data sheet.

In the remainder of this section, the terms IDLE and SLEEP are used.

► *Be aware that with waking up from the SLEEP mode significant delays can be involved influencing the timing of your application. These delays depend, amongst others, on the controller's oscillator configuration. Details can be found in the controller's data sheet.*

11.2 Power modes and the system timer

The configured system timer is initialized by AVIX to continue operation during IDLE mode. This implies that when using IDLE mode, any desired hardware timer can be configured to be used as the system timer.

When using SLEEP mode the controller's core oscillator is stopped meaning that hardware timers clocked from this oscillator also stop. As a result, AVIX timing related functionality will no longer function. This might be acceptable when waking up the application from SLEEP mode does not depend on AVIX timers, and when AVIX timers that are in use are allowed to stop during SLEEP. Note that also round robin scheduling depends on the system timer being operational.

When AVIX timing functionality must continue to operate, even during SLEEP, the system timer must be clocked from an external oscillator. This is accomplished by setting configuration parameter `avix_TIMER_CLK_SECONDARY` to value 1. In this situation not all available hardware timers can be configured to be used as the system timer. Typically hardware timer 1 is the timer that can be clocked from the external oscillator and thus it is also required to configure timer 1 as the AVIX system timer.

When configuring the system timer to be clocked from an external oscillator, AVIX configures the hardware timer accordingly. Also AVIX takes care of setting the applicable bit in the OSCCON register to activate the external oscillator.

11.3 Power modes and the system timer period

The system timer period specifies the period with which the system timer will generate an interrupt. When using IDLE or SLEEP, these modes are aborted by interrupts and thus also by the system timer interrupt provided the system timer is active (see above).

Waking up from IDLE virtually happens instantaneous. This implies that any allowed system timer period can be configured.

Waking up from SLEEP may take longer depending on the system clock configuration. When using a PLL based clock, it can take up to 2ms for the controller to wake up with a locked oscillator.

It shall be clear that in this case it makes no sense for the system clock to expire faster than this 2ms period without losing accuracy. Therefore when using SLEEP it is advised to select a system timer period substantially longer than this 2ms wake up time. Advised is to set the system timer period 2 to 5 times longer than the SLEEP wake up time. For details about the SLEEP wake up time which is related to your oscillator configuration consult the controller’s data sheet.

► *Under all circumstances, AVIX configures the system timer without prescaler for the highest possible accuracy. This implies that when the system timer is clocked from the core oscillator, for a 40MIPS part, the maximum period is ~1.6ms. As a result system timer periods exceeding this value are only possible when using an external oscillator which most of the time is running with a frequency of 32,768Hz.*

11.4 Other power reducing capabilities

Besides the power management mechanisms used by AVIX, additional power management capabilities are offered by the controllers in the form of clock switching and DOZE mode. Both mechanisms allow the controller to operate at a lower frequency. Since the controllers power consumption is proportional to its clock frequency, a lower operating frequency results in a reduction of power consumption. The lower clock frequency does however also result in an increase in processing time proportional to the clock frequency. As a result a lower clock frequency leaves less time available for the idle thread to run and thus less time the controller can be set in IDLE or SLEEP. Clock frequency based power regulation is intended to be used by applications that do not work event driven making it difficult to use IDLE or SLEEP mode. When using AVIX as the basis for power management these power management capabilities do not offer any added value and there is no need to make use of them.

12 Resource Usage

This chapter describes the controller's resources used by AVIX.

- **Section 12.1, Overview**, provides a global overview of the resources used by AVIX.
- **Section 12.2, Registers used by AVIX**, provides a detailed description of the controllers registers used by AVIX and guidelines how to manipulate bit fields in registers only partly used by AVIX.

12.1 Overview

Internally AVIX makes use of one of the controller's (hardware) interrupts, one of the controller's hardware timers and a small amount of RAM. This section provides information on these resources and interrupt configuration in general, which is important to know when building an application with AVIX.

12.1.1 Interrupts

Internal activation of the AVIX scheduler core is based on an interrupt. This interrupt is for exclusive use by AVIX and thus cannot be used by the application. The specific interrupt being used is configurable, allowing an interrupt to be selected which is not required by the application. Details on how to configure this interrupt are found in §7.

When the internal scheduler is active, effectively this runs as a hardware priority 1 interrupt.

A second interrupt used by AVIX is related to the configured hardware timer. The hardware timer interrupt handler runs at hardware priority 2.

Because of these two interrupts being used, the AVIX zero latency feature is available for hardware interrupt levels 3 up to and including 7.

AVIX assumes the controller to be initialized to use prioritized nested interrupt handling. AVIX never disables interrupts and expects your application to do so neither. Under no circumstance should you disable all interrupts since this prevents the AVIX scheduler and system timer from working. Because of the rich set of functions AVIX offers for interrupt handler-thread integration it is very unlikely all interrupts ever need to be disabled.

It is however allowed to disable specific interrupts by setting the applicable bit in the control register for that specific interrupt to zero (0) might your application require this.

► *Under no circumstance should the controllers global interrupt level be manipulated. Doing so will stop the AVIX internal timer and/or the scheduler from working for the time interrupts are disabled. In case you do need to block interrupts for a specific period of time, you should do so by disabling that specific interrupt by setting its IE bit to zero.*

12.1.2 Timer

The second hardware resource AVIX uses is a timer. Here too, the specific timer being used is configurable and a timer must be configured not needed by the rest of the application. For the timer also, information on how to configure this is found in §7.

Some of the hardware timers are related to other hardware devices in a dedicated fashion so when selecting a hardware timer for use by AVIX make sure this that it is not needed for some other hardware device which is required by the application[9].

Since AVIX offers a number of software timers limited by the amount of memory only, using one of the hardware timers for AVIX will be no problem since most of the applications timing will be based on these software timers anyway.

[9] The ADC device present in a number of controllers can use hardware timer 3 for starting the conversion. Configuring AVIX to use timer 3 as the base for its internal timing, prevents this timer from being used with the ADC device.

12.1.3 Memory

For its internal bookkeeping, AVIX uses a small amount of RAM which is reserved. Furthermore, some RAM is needed for each of the AVIX kernel objects created by the application. The memory layout of an AVIX based application is shown in Figure 10. This memory map occupies RAM below the 32K boundary, even if the controller supports EDS RAM above this boundary. In the latter case this EDS RAM can be used for AVIX Extended Memory, see §10 for details on EDS RAM.

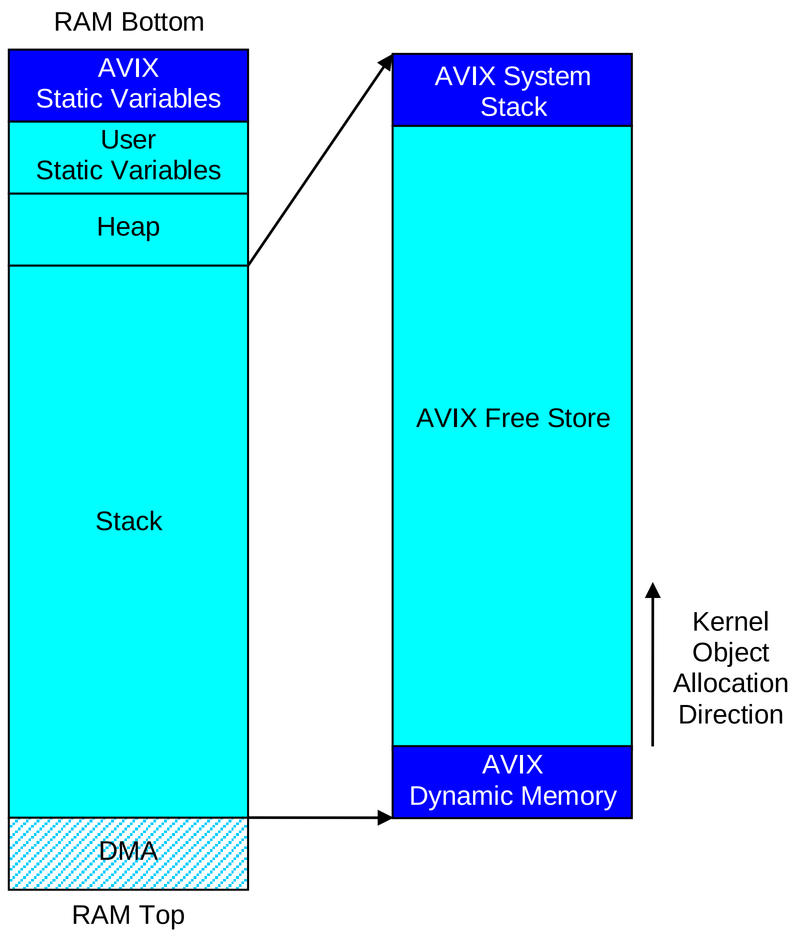


Figure 10: AVIX based application memory layout

When building an AVIX based application use is made of the default linker scripts as installed with the C30/XC16 compiler suite. AVIX requires no modifications to these linker scripts resulting in a standard memory layout which corresponds to the left part of Figure 10.

The lower addresses of RAM contain static variables used by AVIX and the application. Next there is some memory reserved for the heap. A heap only needs to be present when the application makes use of malloc and free functions, as offered by the standard runtime. Since AVIX offers its own memory management, advised is to define the heap to have a size of zero bytes. In case the application uses the heap, precautions must be taken since the heap is not ‘thread safe’.

When building a C30/XC16 compiler suite based application, al memory left when subtracting static variables and the heap is allocated to the stack. Note this is not the AVIX stack for which reason it is called C30/XC16 compiler suite stack.

This C30/XC16 compiler suite stack memory is entirely claimed by AVIX and used for three purposes. At the lower addresses a section is used for the system stack, the size of which is configurable again. At the upper addresses, some memory is reserved for internal bookkeeping purposes. All RAM left now is called the ‘AVIX Free Store’. This section is used for kernel objects that are created by the application. Whenever a …`_Create` service is called, the required amount of RAM for the bookkeeping related to the specific kernel object is claimed from this ‘AVIX Free Store’. This is a zero overhead implementation since AVIX kernel objects cannot be deleted implying no management information is required in order to be able to return memory to this ‘AVIX Free Store’.

For memory to be allocated and freed for reuse again by the application, AVIX offers a memory pool mechanism allowing fixed size memory blocks to be allocated and freed again.

12.2 Registers used by AVIX

Table 5 contains an overview of the registers (partly) used by AVIX. Partly, because for some registers AVIX only uses a subset of the bits of these registers while the other bits are available to the application.

With ‘used by AVIX’, what is meant is exclusive write access. Under no circumstance may the application write the bits in the registers mentioned in Table 5. Also, when of a specific register AVIX only uses a subset of the available bits and the application writes the other bits, the application must guarantee such writes are atomic. Under no circumstance should the application manipulate the content of a register partly used by AVIX using separate Read-Modify-Write operations since this will lead to unpredictable behavior and application failure.[10]

Note that using the ‘C’ language under no circumstance operations are guaranteed to be atomic.

The application is allowed read access to all registers since this will not change the content of the register. Table 5 contains the following columns:

- **Register**; name of the register as defined in the applicable Microchip documentation and header files. Note that not all registers are known on forehand since some depend on configuration parameters. In this case a placeholder character is used in the register name and an explanation is given what the concrete value of this placeholder character depends on.

- **Field**; name of the bit field in the register as defined in the applicable Microchip documentation and header files. Note that not all fields are known on forehand since some depend on configuration parameters. In this case a placeholder character is used in the field name where an explanation is given what the concrete value of this placeholder character depends on.

  When this column contains value <all>, this means AVIX uses all bits of that register and the application may not write a single bit of such a register.

- **Description**; textual explanation of the register and the field.

[10] Manipulating bits in a SFR is not exclusive to the use of AVIX. Also in a non-AVIX based application, atomic manipulation of bitfields in registers shared between main code and interrupt handlers or between multiple interrupt handlers is required. In this situation also it is the applications responsibility to guarantee this.

Register	Field	Description
SR	IPL	Status register interrupt priority bits.
IECx	<i>IE	AVIX uses one the controllers interrupt sources. The specific interrupt source used is user configurable. An interrupt source can be enabled through a bit in one of the IEC registers. The interrupt enable bit for the configured interrupt source is exclusively used by AVIX. The specific IEC register is the same as configured through configuration parameter `avix_SWI_IRQ_CTRREG`. The bit field is <i>IE where <i> is equal to the value of configuration parameter `avix_SWI`.
IFSx	<i>IF	AVIX uses one the controllers interrupt sources. The specific interrupt source used is user configurable. An interrupt being active is identified by a bit in one of the IFS registers. The interrupt bit for the configured interrupt source is exclusively used by AVIX. The specific IFS register is the same as configured through configuration parameter `avix_SWI_IRQ_REG`. The bit field is <i>IF where <i> is equal to the value of configuration parameter `avix_SWI`.
IPCx	<i>IP	AVIX uses one the controllers interrupt sources. The specific interrupt source used is user configurable. For each interrupt the priority can be set in a bit field in one of the IPC registers. The priority bit field for the configured interrupt source is exclusively used by AVIX. The specific IPC register is the same as configured through configuration parameter `avix_SWI_IRQ_LVLREG`. The bit field is <i>IP where <i> is equal to the value of configuration parameter `avix_SWI`.
IECy	T<t>IE	AVIX uses one the controller's hardware timers. Each hardware timer can act as an interrupt source. An interrupt source can be enabled through a bit in one of the IEC registers. The interrupt enable bit for the configured hardware timer is exclusively used by AVIX. The specific IEC register is the same as configured through configuration parameter `avix_SYSTMR_IRQ_CTRREG`. The bit field is T<t>IE where <t> is equal to the value of configuration parameter `avix_SYSTMR`.
IFSy	T<t>IF	AVIX uses one the controller's hardware timers. Each hardware timer can act as an interrupt source. An interrupt source being active is identified by a bit in one of the IFS registers. The interrupt bit for the configured hardware timer is exclusively used by AVIX. The specific IFS register is the same as configured through configuration parameter `avix_SYSTMR_IRQ_REG`. The bit field is T<t>IF where <t> is equal to the value of configuration parameter `avix_SYSTMR`.
IPCy	T<t>IP	AVIX uses one the controller's hardware timers. Each hardware timer can act as an interrupt source. For each interrupt the priority can be set in a bit field in one of the IPC registers. The priority bitfield for the configured hardware timer is exclusively used by AVIX. The specific IPC register is the same as configured through configuration parameter `avix_SYSTMR_IRQ_LVLREG`. The bit field is T<t>IP where <t> is equal to the value of configuration parameter `avix_SYSTMR`.
TMRy	<all>	Timer count register for the configured hardware timer where y is the timer number specified by configuration parameter `avix_SYSTMR`.

Register	Field	Description
PRy	<all>	Timer period register for the configured hardware timer where y is the timer number specified by configuration parameter `avix_SYSTMR`.
TyCON	<all>	Timer control register for the configured hardware timer where y is the timer number specified by configuration parameter avix_SYSTMR.
OSCCON	LPOSCEN	Used to manage the controllers low power mode

Table 5: Registers (partly) used by AVIX