

Running a FIR Filter on the AXE Using the AXE Scheduler

by: Kalle Odenthal
Microcontroller Solutions Group, Guadalajara, Mexico

1 Introduction

1.1 Abstract

The MPC5121e, besides an e300 core and an MBX graphics coprocessor, has also the AXE coprocessor, which is optimized for digital signal processing. This enables the applications running on the e300 core to offload signal processing (or other) tasks to the AXE, to reduce its workload and to speed up the system. This application note describes how to use the AXE coprocessor to offload tasks from the e300 core using the Freescale Linux OS board support package (BSP) available at www.freescale.com (search MPC5121e). This application note shows:

- How to set up a development environment using CodeWarriorTM.
- How to load and run the AXE scheduler to facilitate task scheduling on the AXE and to provide an easy API to manage the AXE tasks for inter-processor communication (IPC).

Contents

1	Introduction	1
1.1	Abstract	1
1.2	Objective	2
2	Detailed Description	2
2.1	Requirements	2
2.2	Foreword	3
2.3	e300-AXE Software Architecture	3
2.4	Development Environment Setup	5
2.5	Creating an AXE-FIR Filter Task	7
2.6	The e300 Core Application for the AXE-FIR Filter Task	29
2.7	Running the System	50

Detailed Description

- How to include the PPC-AXE driver library in your application project. Such library provides an API for the IPC on the e300 core side.
- How to write an AXE task.
- How to write an e300 core application that communicates with an AXE task.

This application note is working with a small FIR filter as an example: the e300 core application loads a FIR filter task to the AXE, runs it with a set coefficients and input data, and prints the output.

1.2 Objective

At the end of this application note you will be able to:

- Create and test your own AXE tasks.
- Load and run an AXE task from the e300 core application.

This enables you to increase the system performance significantly.

2 Detailed Description

This section gives a detailed explanation on how to accomplish the objectives listed in the abstract. The different sections cover the following topics:

- Requirements needed to fulfill the documents objectives
- A foreword to the example source code
- The e300-AXE architecture
- How to properly set up your development environment
- An implementation guide to create a FIR filter running as an AXE scheduler task
- An implementation that runs the AXE scheduler task from the e300 core application
- How to run the whole system

2.1 Requirements

To work with this application note the user has to complete the following requirements:

- An ADS5121 board Rev. 3 or later. You may use CodeWarrior simulation capabilities if you don't have an ADS5121 board, but you will not be able to complete all the activities.
- Linux host (can be done using a virtual machine) with the latest Freescale LTIB installed. The .iso file with the installer is available at www.freescale.com (search for Linux BSP MPC5121EADS).
- Host machine for CodeWarrior. This can be the Linux host or a Windows host. This application note is focused on a Windows host for CodeWarrior. Screenshots and menu structures might vary slightly, if you are using CodeWarrior on Linux OS.
- Serial communication program installed in the host system.
- Serial cable and a serial port on the serial terminal host. If you don't have a serial port, a USB serial adapter can be used.

- CodeWarrior USB/ethernet TAP. This is optional. You'll need it for the AXE stand-alone task debugging (See [Section 2.5.2, “Creating an AXE Stand-Alone Task with CodeWarrior”](#)).
- CodeWarrior Development Studio for MobileGT v9.0 Build 71208 or higher, available at www.freescale.com
- An ethernet network to connect the Linux OS host, the target ADS board, and the CodeWarrior host.
- Basic knowledge of:
 - How to deploy a new package to LTIB. See www.bitshrine.org for help.
 - How to mount a network file system (NFS) for the on-board Linux OS. Browse the www.freescale.com webpage for help.
 - C programming language
 - Elementary knowledge of FIR filters

2.2 Foreword

This application note contains a lot of source code. This was considered as necessary. To avoid expanding this document unnecessarily, code comments were left out. However, this document comes along with a .zip file that contains all the source code, libraries, and project files. The code in the .zip file contains proper code comments. Whenever you need more information regarding the code, browse the .zip file content. The .zip file can be downloaded at www.freescale.com. Whenever the content of this .zip file is referred to, the prefix zip:\ is used. For example zip:\AXE scheduler refers to the folder “AXE scheduler” in the .zip file.

The source code was written to explain the necessary steps in an understandable way rather than to be optimized for performance. Most steps are sequentially written down in the e300 application main function, which, in real applications, might be encapsulated in different functions. Also the source is not always divided clearly into header and source files. It was considered that having certain information in just one file will increase clarity.

2.3 e300-AXE Software Architecture

This section gives an overview of the software architecture used in this application note to swap tasks from the e300 core to the AXE.

The e300 core and AXE are two independent cores that share common memory space. For inter-processor communication, there are assigned registers in the shared memory. Refer to the MPC5121e reference manual on the www.freescale.com webpage for more detailed information.

Freescale offers a framework that covers the software for the communication layers on both sides: the e300 and AXE. The AXE scheduler and the PPC-AXE driver are handled in the following sections.

2.3.1 AXE Scheduler

On the AXE side there is the AXE scheduler. The AXE scheduler is supposed to be shipped within the ADS5121e BSP. In the current version (ltib-mpc5121ads-20080528.iso) the scheduler is shipped only partially. How to obtain a complete version is explained in [Section 2.4.2, “LTIB and AXE”](#).

Besides the implementation of a message-based communication layer for inter-processor communication, the AXE scheduler offers some features of the OS-like functionality:

- It can handle various tasks at the same time and execute them according to their priority.
- It pauses and resumes tasks due to the requested resources' availability.
- It pauses tasks waiting for e300 core messages and resumes them respectively.
- It provides an inter-task message API .
- It provides a memory allocation API.

The scheduler is visible to a task by an API header file linked at an execution time. Tasks are loaded as binaries to the running scheduler. For detailed information on the AXE scheduler refer to further documentation.

2.3.2 PPC-AXE Driver

On the e300 core side, Freescale provides a software library called PPC-AXE driver. The PPC-AXE driver covers all inter-processor communication functions. That way it provides an easy-to-use API for the e300 core applications to load, run, and manage tasks on the AXE.

[Section 2.6.1, “Adding the PPC-AXE Driver to the e300 Core Project”](#) covers the integration of the PPC-AXE driver to your e300 core application.

2.3.3 e300 Core Application to AXE Task Interaction

[Figure 1](#) outlines the interaction between the e300 core application and an AXE task.

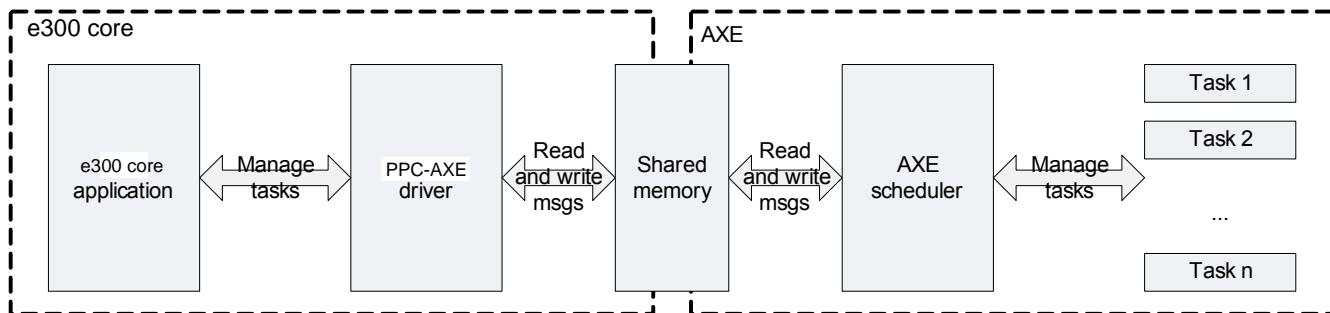


Figure 1. e300-AXE Interaction

- The PPC-AXE driver provides an API to the e300 core application, to manage the AXE tasks (load, start, send message, and so on).
- The PPC-AXE driver passes the e300 core application requests to the AXE scheduler using messages.
- The AXE scheduler manages the tasks.

NOTE

The PPC-AXE driver and AXE scheduler are given software. The scope of this application note lies in the creation of the e300 application and corresponding AXE tasks.

2.4 Development Environment Setup

This section guides you to setup the development environment properly. [Figure 2](#) and [Figure 3](#) show a topological and functional view of the development environment respectively. If you are using CodeWarrior for Linux OS, another serial terminal or a virtual machine, your topology will vary slightly, but four main entities can be named independent of their actual realization:

- The ADS5121 board
- A Linux OS host, containing the LTIB and the network file system (NFS)
- A CodeWarrior host (Windows or Linux)
- A serial terminal host (Windows or Linux)

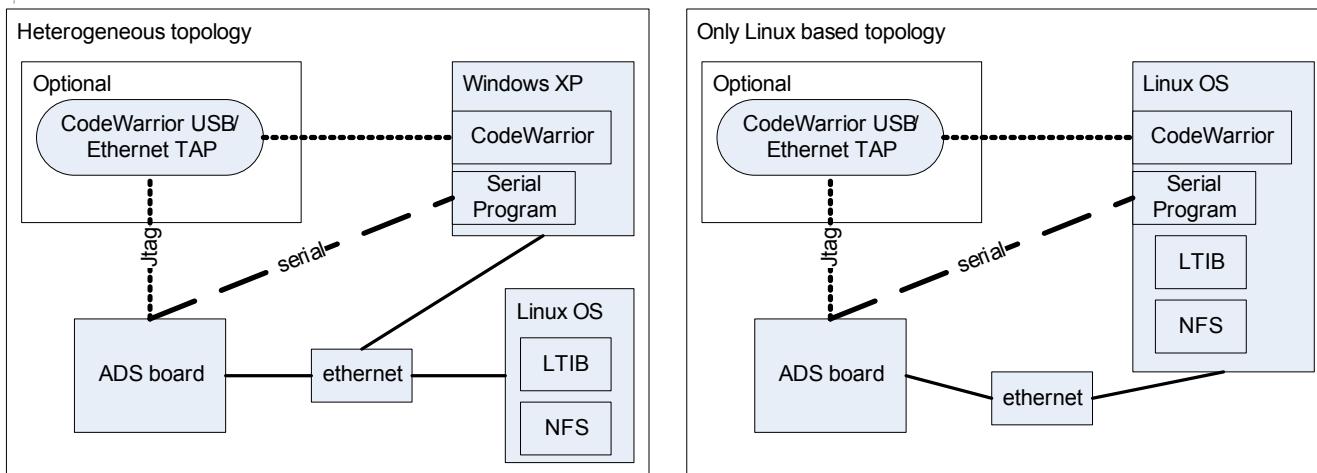


Figure 2. Two Possible Development Environment Topologies

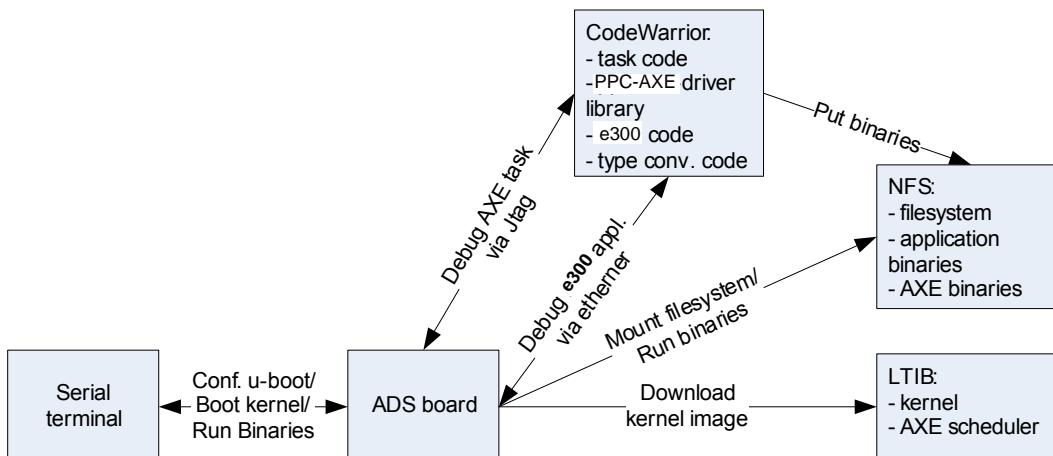


Figure 3. Functional View of the Development Environment

Detailed Description

Below, the setup of all components is discussed in detail.

2.4.1 Serial Terminal

Use the serial terminal you are used to use on your preferred platform. There is no restriction. Set the serial terminal configuration to:

- Baud rate: 115200
- Data: 8-bit
- Parity: none
- Stop: 1-bit
- Flow control: none

2.4.2 LTIB and AXE

At the time of writing this application note the AXE scheduler is not delivered with the latest LTIB distribution (ltib-mpc5121ads-20080528). Browse the Freescale webpage to find a newer version of the LTIB that provides the AXE scheduler as a package or install the scheduler manually as described below.

2.4.3 Installing and Running the AXE Scheduler on the ADS5121e

It is assumed that the LTIB is installed on the Linux OS host and Linux is running on the ADS5121e using the network file system, hosted on the Linux OS host in the folder <ltibInstallationFolder>/rootfs.

Follow the following steps to install the AXE scheduler on the network file system:

1. In the NFS root file system create a /usr/lib/axebins and a /lib/modules/2.6.24.6/misc folder if it doesn't exist already.
2. Copy the file scheduler.bin from zip:\ AXE scheduler\bin to ~rootfs/usr/lib/axebins.
3. Copy the file axe.ko from zip:\ AXE scheduler\bin to ~rootfs/lib/modules/2.6.24.6/misc.
4. Copy the file startaxeserver from zip:\ AXE scheduler\bin to ~rootfs/usr/bin.
5. Copy the file axed from zip:\ AXE scheduler\bin to ~rootfs/usr/bin.

NOTE

All those files have to be executable by users. Check the file's permissions (chmod) to ensure this.

2.4.4 CodeWarrior Development Studio for MobileGT V9.0

Within this application note, CodeWarrior Development Studio for MobileGT V9.0 (CodeWarrior) is used to:

- Write, compile, and build the AXE-task source code.
- Debug the AXE-task algorithms via JTAG.
- Deploy the AXE-task binaries to the NFS.
- Write, compile, build, and debug the e300 core application source code.

- Link the e300 application source code to the PPC-AXE driver.

Using CodeWarrior is not mandatory to complete the scope of this application note. Any step can be done outside the CodeWarrior using the accordant tools:

- A tool chain to compile and build the AXE task (Search the www.freescale.com webpage).
- An ethernet debugger (in other words the GNU project debugger GDB).
- A tool chain to compile, build, and link the e300 core application.

NOTE

Alternative tools to CodeWarrior are not within the scope of this application note.

2.4.4.1 CodeWarrior Installation

If you plan to use CodeWarrior on a Windows system, you have to previously install Cygwin. Cygwin is a Windows driver that simulates a Linux environment. This is necessary because some GNU and AXE CodeWarrior tools need a Linux environment. To install Cygwin, browse the Cygwin webpage (www.cygwin.com) and follow the given instructions.

CodeWarrior Development Studio for MobileGT V9.0 is shipped with your ADS board. If you don't have an installation CD browse www.freescale.com to download the installer file. Follow the given instructions to install and register CodeWarrior.

2.5 Creating an AXE-FIR Filter Task

This section covers all the steps needed to create a task that runs with the AXE scheduler on the AXE coprocessor. By means of a simple FIR-filter example, the following topics are discussed:

- Fixed-point extension for the AXE.
- DSP-C: a C language specification designed for digital signal processing.
- How to create the FIR filter as a stand-alone task for the AXE and debug it.
- How to create an AXE scheduler task out of the stand-alone task.
- Utilities to debug the AXE scheduler task.

2.5.1 AXE Fixed-Point Extension

The AXE can work in two modes: fixed mode and integer mode. For each function of a task, the mode can be declared independently when, and only when, the code is compiled with fixed-point mode enabled. Fixed-point mode is by default enabled. To disable the fixed-point mode add `disable-fixed-point` to the AXE-compiler options. (Refer to further documentation). The fixed-point format is used to represent fractional numbers. Each fractional number is represented by an integer part and a fractional part, for example: number = 12.34, where 12 is the integer part and 0.34 the fractional part. In the fixed-point format the number of bits for the integer part and the fractional part is fixed. For example the type `_fixed` has zero integer bits and 31 bits for the rational part. This stands in contrast to the common double or floating types, where the number of integer bits and fractional-part bits is variable. Fixed-point arithmetic is normally easier to implement at hardware level and also faster than floating-point arithmetic. There are

Detailed Description

plenty of books and websites that further explain fixed and floating point. In 1998, the Associated Compiler Experts (ACE) released an extension to ISO/IEC IS 9899:1990 (C language specification) to enhance the C language for DSP-specific requirements. This extension is called DSP-C and can be downloaded at www.dsp-c.com. The fixed-point extension for the AXE-GCC compiler complies with the DSP-C specification. The syntax to define a function for fixed-point arithmetic is as follows:

In declaration:

```
<return type> <yourFunctionName>([parameters]) __attribute__((ifa));
```

In definition:

```
<return type> <yourFunctionName>([parameters]) __attribute__((ifa))
{
    Function body;
}
```

where IFA stands for instruction-flow-all mode. [Table 1](#) lists the fixed-point data types available for the AXE. The suffix is used to direct the compiler to treat a constant number as the data type determined by the suffix. For example:

```
short __fixed c = 0.89799999r + 0.029r;
```

Here, the addition of the summands is done as `__fixed` because of the suffix “r”. At the moment the result is assigned to `c`, it is converted to `short __fixed`. The `scanf/printf` specifier is used to direct the compiler to represent the variable in the `scanf/printf` functions in the suffixed manner. For example:

```
__fixed f = 0.1234r;
printf("This is a __fixed variable f = %r", f);
```

Refer to ‘*Using Fixed point Extension with GCC 4.1.1 -Rev 6.pdf*’ for more information about the AXE-GCC compiler-specific issues.

Table 1. AXE Fixed-Point Data Types

Data type	Sign bit	Number of integer bits	Number of fractional part bits	Suffix	scanf/printf Specifier
short __fixed	X	0	7	r, hr	%hr
__fixed	X	0	31	r	%r
long __fixed	X	0	39	R	%lr
short __accum	X	7	7	a, ah	%ha
__accum	X	7	31	a, ar	%a
long __accum	X	7	39	A, R	%la
unsigned short __fixed	-	0	7	ur, uhr	%hR
unsigned __fixed	-	0	31	ur	%R
unsigned long __fixed	-	0	39	UR, uR	%lR
unsigned short __accum	-	7	7	ua, uha	%hA
unsigned __accum	-	7	31	ua, ur	%A
unsigned long __accum	-	7	39	UA, uA, UR, uR	%lA

NOTE

All signed fixed-point data types are in 2's complement.

2.5.2 Creating an AXE Stand-Alone Task with CodeWarrior

Stand-alone tasks can be easily debugged on the AXE with CodeWarrior via the JTAG. However, tasks loaded to the AXE scheduler as binaries cannot be debugged in the common, convenient way. There are some utilities to facilitate debugging of those tasks, but they don't have the same power and usability as the conventional debugging tools.

So, before we create a task for the AXE scheduler, it makes sense to test our algorithm first with a stand-alone task. A stand-alone task is compiled directly for the AXE without using the scheduler and can be downloaded and debugged via the JTAG.

To create a stand-alone task for the AXE follow the following steps. Skip step one to three if you don't have a JTAG connector.

1. Connect the JTAG connector to the ADS board. See [Figure 4](#). Pay attention to the orientation of the JTAG connection. The way to connect the board might vary slightly due to different board revisions. Check the general board description to find the JTAG on your board.
2. Connect the JTAG connector to your network/USB. If you are using a CodeWarrior ethernet TAP check the instructions and configure it.
3. Connect the board to the power supply and push the reset button on the board.
4. Start CodeWarrior.
5. Over File — New select the AXE stationary project.
6. Enter a project name (for example axe-fir-so), choose the location where you want the project to be installed, and press the OK button.
7. Expand the MPC5121 node, choose AXE Sample on MPC5121, and press the OK button.
8. You have now created a project for an AXE stand-alone task.

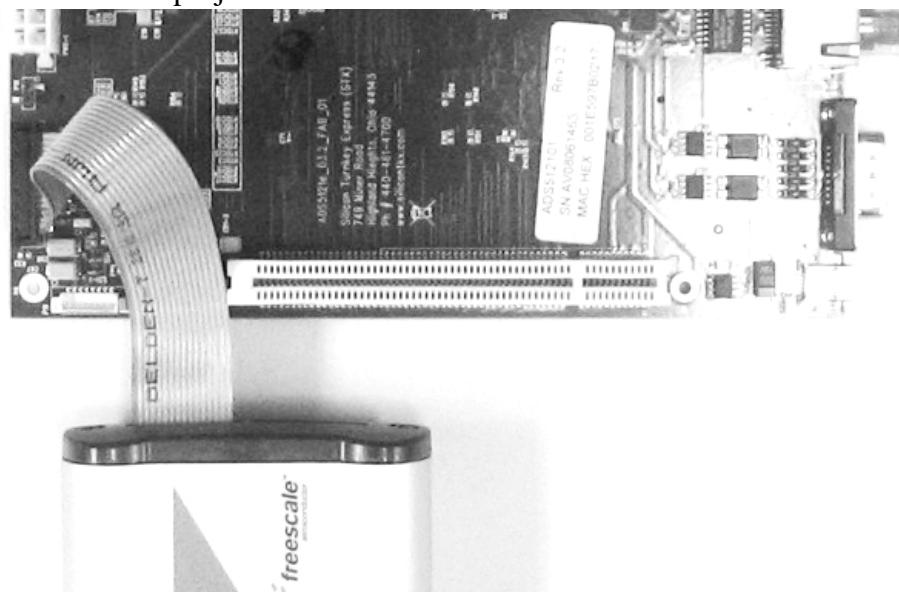


Figure 4. Connection of the JTAG Connector and ADS5121 Rev. 3.2 Board

Detailed Description

To enable the console printing and scanning, you have to include the system call library:

1. Open Edit — Debug Versions Settings.
2. Expand the Debugger node.
3. Choose System Call Service Settings.
4. Check Activate Support for System Services.
5. Now that we are here choose Remote Debugging on the left side and choose your connection type. If you don't have a JTAG connector choose the SNE simulator.
6. Press the OK button.
7. Open the Files tab in the CodeWarrior project, right click on Source and select Add Files to add a library to your project.
8. Browse in the explorer window to your CodeWarrior installation folder. Normally it would be something like C:\program files\freescale\CodeWarrior for MobileGT V9.0.
9. Open the folder SNE_ABI\SystemCallSupport\Lib.
10. Select axe_syscall.a and press the Open button.
11. Accept the default settings for Add Files.
12. You have now added a console to your project.

NOTE

The  button brings you directly to the Debug Versions Settings menu.

Now you are ready to build and debug your AXE stand-alone task:

1. Press F7 to make your project.
2. Press F5 to start the debugger.
3. Now you should be able to step through the code until you'll see the "Welcome to CodeWarrior" message in the console window.

NOTE

The printf() function will write a string into a buffer. When this buffer is read by the console it is not previewable. The "Welcome to CodeWarrior" message might appear some cycles later in the console than the actual printf() code was executed.

2.5.3 Creating an AXE Stand-Alone FIR Filter Task

Now that all of the requirements to create an AXE stand-alone task are fulfilled, it can be filled with the desired algorithm. In this example it will be an FIR filter.

The FIR filter filters the incoming signal by means of a set of coefficients and an inner-memory state. Depending on the choice of coefficients, the FIR filter behaves as low-, band-, or high-pass. The FIR filter responds to an impulse input signal with its coefficients as shown in [Figure 7](#). This is important to validate the implementation. [Figure 5](#) outlines the structure of a common FIR filter.

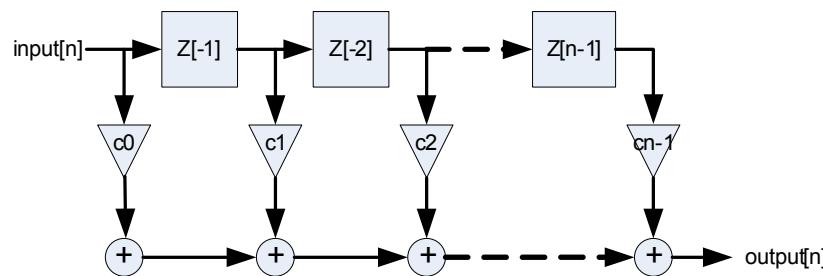


Figure 5. Digital FIR Filter Schematic

Four memory units can be derived from the schematic:

- Input: the algorithm's input
- State: internal state Z (here a shift register)
- Parameters: A set of parameters (the coefficients)
- Output: the algorithm's output

These four units are generic to the most of signal-processing algorithms. If you can determine them for your customized algorithm, you will be able to reuse most of the code presented in this application note. Refer to further documentation for generalization of the AXE task development.

Before the code for the algorithm is implemented, we have to define how to manage those four units. To validate the FIR filter code it will be sufficient to run it once with a fixed declared-data set. However, as a scheduler task it will be a bit more complicated. The calling application should have control over those four units (see [Section 2.6, “The e300 Core Application for the AXE-FIR Filter Task”](#)).

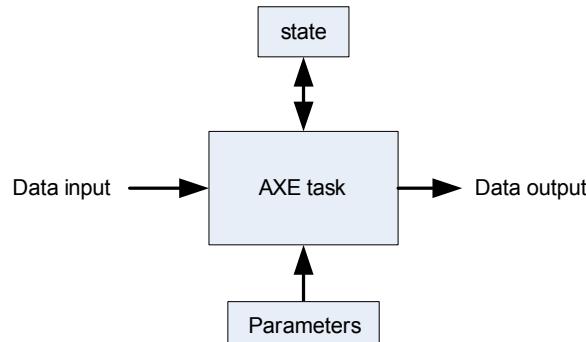


Figure 6. Memory Units

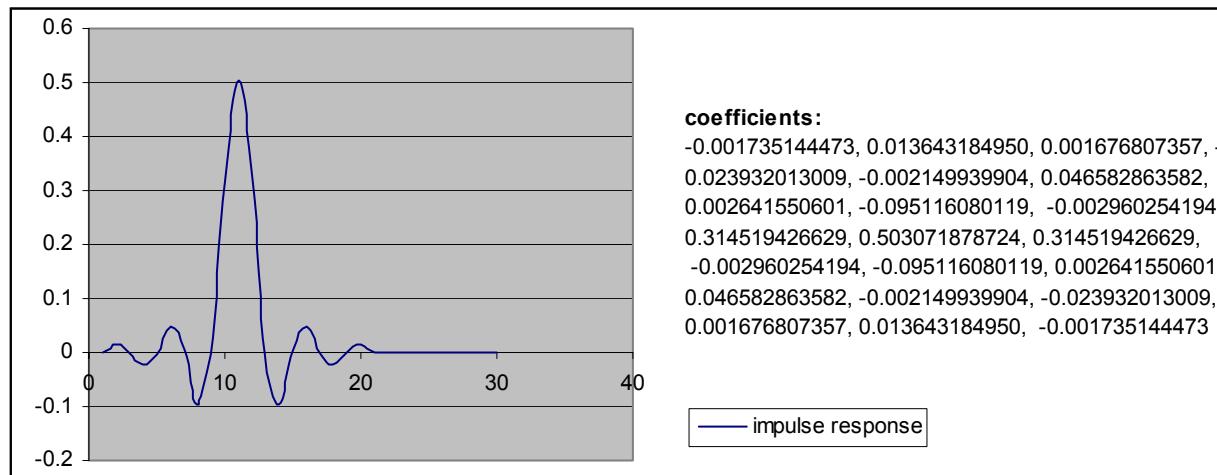


Figure 7. FIR-Filter Impulse Response

2.5.3.1 Defining Algorithm Units

The first step to do is to define the algorithm units. We will do this in the FIR header. To create the FIR header follow the following steps:

1. We assume that the CodeWarrior stand-alone task project is already created as explained in [Section 2.5.2, “Creating an AXE Stand-Alone Task with CodeWarrior.”](#)
2. Open File — New.
3. Select the File tab.
4. Select the text-file template and type fir.h in the File Name box.
5. Use Set to browse to your CodeWarrior AXE stand-alone project source folder.
6. Press Save.
7. Press OK.
8. In the CodeWarrior Project window choose the Files tab.
9. Right click on Source and choose Add Files.
10. Browse to your source folder and select the file fir.h.
11. Click Open.
12. Accept the default settings for Add Files by clicking OK.

Copy and paste the following code into that file:

```
//fir.h

#define __aacFirInput __accum*
#define __aacFirOutput __accum*

typedef struct
{
    AXEInt32 iFilterStages;
    AXEInt32 iCircBufOffset;
    __accum* aacZ;
```

```

} __sFirState;

typedef struct
{
    AXEInt32 iCoefCount;
    __fixed* afiCoef;
} __sFirParams;

```

Input and output are both defined as arrays of type `__accum`. The `__accum` type provides seven bits for the integer part and 32 bits for the fractional part. The filter's state is represented by an array of `__accum`, containing the shift-register entries. The integer `iFilterStages` defines the number of register stages. Due to the fact that the shift register will be implemented as a circular buffer, an offset integer that indicates the register's starting point is needed. The filters' parameters are an array of `__fixed` coefficients and an integer defining the number of coefficients.

NOTE

The syntax of the two underscores before a variable comes from the DSP-C syntax and should highlight that those types are for AXE use. This becomes more important when we create variables in the e300 core for the AXE (see [Section 2.6, “The e300 Core Application for the AXE-FIR Filter Task”](#)).

NOTE

`AXEInt32` is an AXE integer type. It is defined in `axetypes.h`. It is chosen over the integer type, concerning that the AXE stand-alone task will be an AXE scheduler task. We'll come to this later.

2.5.3.2 Creating the Test Data

With knowledge about the algorithm's units, the test data can now be easily created. Due to the fact that the AXE stand-alone task is used to debug and validate the algorithm, the test data can be put in a header file and it is easier than implementing dynamic data-loading procedures. The only thing that has to be considered is that the algorithm's units are defined as dynamic data, while in the header we can only define static data. So the dynamic data has to be created from the static one. Create and include a new header file to your project as you did before, but name it `firdata.h`. In `firdata.h` you can specify all static objects and then convert them into the dynamic entities. In the .zip file corresponding to this application note, there is a set of coefficients. However, there are plenty of applets on the internet that can be used to calculate a set of coefficients corresponding to a request.

Below you can see the `firdata.h`. Add the test data at your will.

```

//firdata.h

//define buffer sizes
#define BUFFER_COUNT ???
#define NUMBER_COEF ???
#define NUMBER_STAGES (NUMBER_COEF - 1)

//define static arrays

```

Detailed Description

```

__accum myStaticInput[BUFFER_COUNT] = { ... };
__accum myStaticOutput[BUFFER_COUNT] = { ... };
__fixed myStaticCoefficients[NUMBER_COEF] = { ... };
__accum myStaticRegister[NUMBER_STAGES] = { ... };

//create the dynamic algorithm entities
__aacFirInput myDynInput = myStaticInput;
__aacFirInput myDynOutput = myStaticOutput;
__sFirState myDynState = { NUMBER_STAGES, 0, myStaticRegister };
__sFirParams myDynParams = { NUMBER_COEF, myStaticCoefficients };

```

2.5.3.3 Implementation of the FIR Algorithm

The algorithm will be implemented in a function called ifa_fir, where the suffix ifa denotes execution in the fixed-point mode (see [Section 2.5.1, “AXE Fixed-Point Extension”](#)). Additionally, the compiler has to be informed to compile the function in that way. This is done in the function declaration by using the suffix __attribute__((ifa)). Besides the algorithm entities, the ifa_fir function takes an integer value that indicates how many cycles the function should calculate. In our case we take the size of our input buffer for this value to ensure that all of our input is processed. An exact explanation of the algorithm is out of the scope of this application note. For the purpose of this application note it will be sufficient to sketch it.

Declaration:

```

void ifa_fir(__aacFirInput aacInput, __sFirParams *psFirParams
            , __sFirState *psFirState, __aacFirOutput aacOutput
            , int cycles) __attribute__((ifa));

```

Definition:

```

void ifa_fir(__aacFirInput aacInput, __sFirParams *psFirParams
, __sFirState *psFirState, __aacFirOutput aacOutput
, int cycles)
{

    //creating some shorter handles
    int *state_z = &(psFirState->iCircBufOffset);
    int stages = psFirState->iFilterStages;
    __accum *aacZ = psFirState->aacZ;
    int iCoefCount = psFirParams->iCoefCount;
    __fixed *afiCoef = psFirParams->afiCoef;

    int i = 0, ii = 0;//loop counters..

    // loop through the input buffer
    for (i = 0; i<cycles; i++){
        *(aacOutput + i) = (*(afiCoef)) * (*(aacInput+i));
        //summation loop
        for (ii = 1; ii < iCoefCount - 1; ii++)
        {
            *(aacOutput + i) += *(afiCoef+ii)
                * *(aacZ + *state_z);

            if (++(*state_z) >= stages)
            {
                *state_z = 0;
            }
        }
    }
}

```

```

        }

        *(aacOutput + i) += *(afiCoef+ii) * *(aacZ + *state_z);
        *(aacZ + *state_z) = *(aacInput+i);
    }

    //print out results
    for(i=0;i<cycles;i++)
    {
        printf("%a\n",*(aacOutput + i));
    }
}

```

2.5.3.4 Aggregation to an AXE Stand-Alone Task

The last step in obtaining a working AXE stand-alone FIR filter is to aggregate the work we have done before and to run the FIR filter from the main() function. Follow the following steps to run the FIR filter:

1. Add the ifa_fir function declaration to your main.c file of the opened CodeWarrior AXE stand-alone task project.
2. Add the ifa_fir function definition to your main.c file.
3. Include the fir.h header to the main.c file.
4. Include the firdata.h header to the main.c file.
5. Put the function call for the ifa_fir function into the main function.
6. Define the AXEInt32 integer type, since we still don't have the axetypes.h header of the AXE scheduler.

Below it is shown how the main.c file should now look like:

```

//main.c

typedef int AXEInt32;

#include "fir.h"
#include "firdata.h"

#include <stdio.h>
#include <string.h>

void ifa_fir(__aacFirInput aacInput, __sFirParams *psFirParams
            , __sFirState *psFirState, __aacFirOutput aacOutput
            , int cycles) __attribute__((ifa));

void ifa_fir(__aacFirInput aacInput, __sFirParams *psFirParams, __sFirState *psFirState,
            __accum *aacOutput, int cycles)
{
    ...
}

int main()
{

```

Detailed Description

```

//call the fir filter
ifa_fir(myDynInput,&myDynParams,&myDynState,myDynOutput,BUFFER_COUNT);
while (1) {} // loop forever
return 0;
}

```

With F5 you can now run and debug the filter. Congratulations, you have just achieved the first milestone!

NOTE

In having the declaration in a header file, it is recommended to create a new header file for this purpose. Don't put the ifa_fir declaration in the fir.h header because this header will be called also by the e300 core application, resulting in a linker error caused by a missing function definition. If you insist in having the declaration in a header file, it is recommended to create a new header file for this purpose.

2.5.4 Creating the AXE Scheduler FIR Filter Task

Once the algorithm is working correctly you can start creating the AXE scheduler task out of it. There are mainly two basic approaches:

- Building the scheduler task from a scratch and copying the algorithm code into it.
- Reusing the AXE stand-alone code and differentiating sections that are particular for either one of the two targets by using #ifdef's.

Although our sample algorithm is very easy and to copy and paste it to the AXE scheduler task code would be the fastest and easiest way to get started, it is highly recommended to go for the second approach. The reasons are:

- You usually have to change or improve the algorithm while developing it. It would be troublesome to always copy the code from one project into the other to test it, especially if the algorithm's complexity and length really outperforms our sample.
- By having two projects with two times the code parts, persistence problems are pre-programmed.

In this application note, the second approach was chosen. We will create a new project that compiles the code as an AXE scheduler task. In this project we'll include all of the source files from the AXE stand-alone project and add some header files necessary for the AXE scheduler. A configuration header file is created for both projects that define what parts of the source code have to be compiled for each project.

2.5.4.1 The AXE Scheduler

This section gives a small introduction to the AXE scheduler. The AXE scheduler is a code running on the AXE. The script startaxeserver, which was installed during the system setup (see [Section 2.4.2, “LTIB and AXE”](#)), starts the scheduler. To avoid name confusion, it should be mentioned that the terms AXE server and AXE scheduler are used indistinctively. Just the point of view of it differs from one to another. AXE scheduler is the actual name of the software. The term scheduler refers to what in fact the software is doing — it schedules different tasks. On the other hand the term AXE server refers to how an e300 core application (or the AXE client) sees the AXE scheduler — as a server, where due to determined messages

concrete services are executed. The AXE server describes an external view on the scheduler, just as the AXE client describes an external view on the PPC-AXE driver. The AXE scheduler can run, stop, resume, and interrupt tasks. Tasks within the scheduler can have different priorities that determine the scheduling behavior. It also offers mechanisms for semaphores and internal message communication. As a server, it takes care of the correct message handling from the AXE client messages. For a detailed description refer to further documentation.

In this application note the description of the scheduler is restricted to some essential points that are crucial to follow the sample code:

- Loading a task to the scheduler means to load the executable task .bin file to the shared memory and instruct the scheduler to load the code.
- Waiting for a resource or for a message puts a task automatically into the suspend state. In the suspend state, the task consumes no processor load.
- If the resource gets available or a message arrives, the scheduler resumes the task as soon as no higher-priority task is running.
- The scheduler has no direct access to a console. The console printing is handled to the e300 core by means of messages (see CPrint() in [Section 2.5.4.5.3, “Creating the Message Handler”](#)).
- Scheduler function calls are linked while loading the task. This has the disadvantage that wrong typed or used functions are not detected at a build time, but at the moment the scheduler loads the task.

2.5.4.2 Creating an AXE Scheduler Task Project in CodeWarrior

To create an AXE scheduler task project follow the following steps:

1. Start CodeWarrior.
2. Under File — New select AXE stationary project.
3. Enter a project name (for example axe-fir-sched), choose the location where you want the project to be installed, and press the OK button.
4. Expand the MPC5121 node, choose the AXE Task Sample and press the OK button.
5. Copy the task-crt0.S and lcf.x files from the folder zip:\AXE scheduler task setup files to the AXE scheduler task project source folder. You'll have to overwrite the existing lcf.x.
6. Right click on the Source folder in the Files tab of the CodeWarrior window and select Add Files.
7. Browse to your source folder, select the task-crt0.S and press Open.
8. Accept the default Add Files settings.
9. You have now created a project for the AXE scheduler task.

NOTE

Note that lcf.x is the linker compiler file needed to link the output correctly for the AXE scheduler. The file task-crt0.S is the startup code for any AXE task.

2.5.4.3 Merging the AXE Stand-Alone Project to the AXE Scheduler Task

To integrate the source of the stand-alone task follow the following steps:

1. First we have to remove the default main.c file. To do so, open the Files tab of the AXE scheduler project and expand the Source node. Remove the main.c file by right clicking on it and choosing Remove. This action will not delete the file, but just remove it from the project. To delete it, you have to do so manually.
2. To add the AXE stand-alone code to the AXE scheduler task project right click the Source folder in the Files tab and choose Add Files.
3. Browse to your AXE stand-alone task folder and choose both the main.c file and the fir.h header file.
4. Press Open.

Now the necessary files from the AXE stand-alone task project are included. However, the project would not compile due to the printf() call in the ifa_fir function, which is not implemented for an AXE scheduler task. So we have to start to differentiate cases in our source by adding a #ifdef statement:

```
#ifdef STAND_ALONE
    //print out results
    for(i=0;i<cycles;i++)
    {
        printf("%a\n",*(aacOutput + i));
    }
#endif //STAND_ALONE
```

To enable this code for the AXE stand-alone task, STAND_ALONE has to be defined there:

1. Open your AXE stand-alone task project.
2. Right click on the Source folder in the Files tab.
3. Select Add Files.
4. Browse to your AXE stand-alone project folder.
5. Click the  button in the Select Files to add popup to create a new folder.
6. Name the folder stand-alone config.
7. Open that folder.
8. Right click on the empty space, choose New — Text Document (see [Figure 8](#)).
9. Rename the new document from New Text Document.txt to taskcfg.h.
10. Accept the change of the extension.
11. Press Open and accept the default Add Files options.
12. Add #define STAND_ALONE to the taskcfg.h file.
13. Repeat all steps with the AXE scheduler task project, but instead of stand-alone config, name the folder scheduler config. And instead of #define STAND_ALONE, put #define AXE_SCHEDULER_TASK into the file.
14. Add #include “taskcfg.h” at the beginning of the main.c file.

Figure 9 illustrates the resulting structure. Now we can ad libitum define the source-code section just for either one of the projects.

NOTE

It is important to create the configuration file in a new folder since the AXE scheduler task project has the root path of the AXE stand-alone project already as a source path. This happened when we included its source. It has to be ensured that each task project includes its own taskcfg.h.

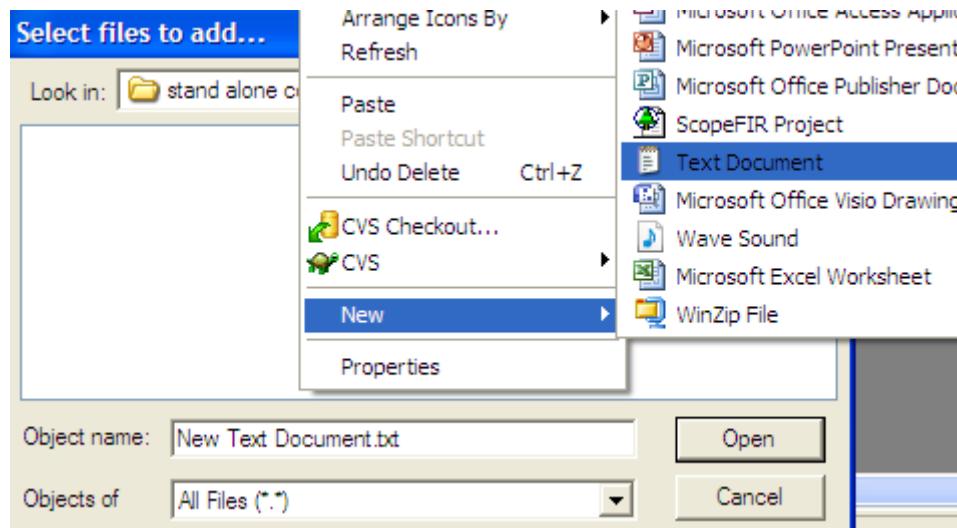


Figure 8.

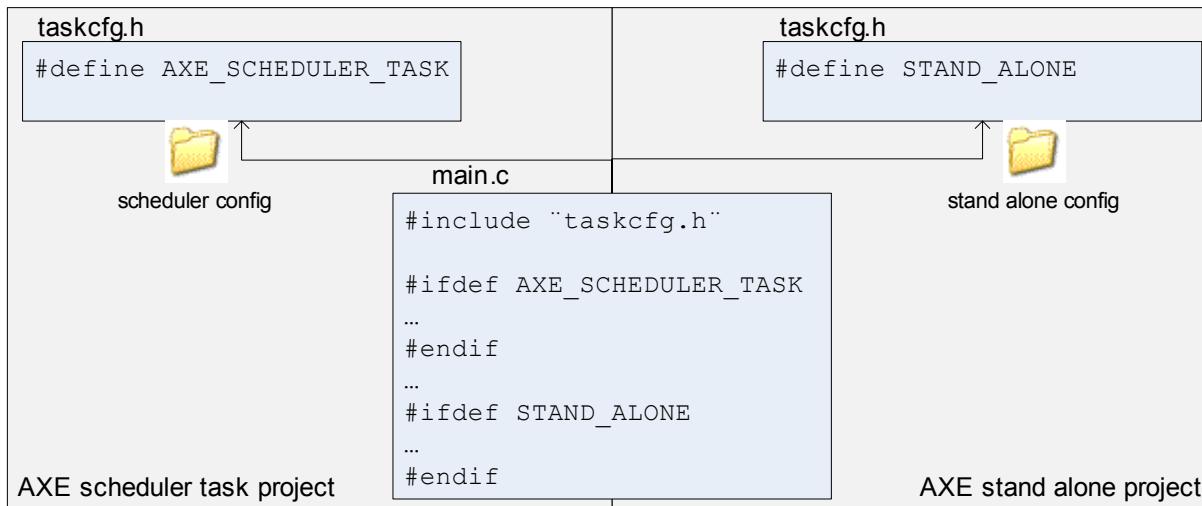


Figure 9. File Distribution

Now this code can run on the AXE scheduler, although the result would be quite disappointing, due to lack of output to the console. Also the static data declaration might not fulfill normal requirements. So in the next step it is shown how to prepare the AXE scheduler task to dynamically load data, execute on

Detailed Description

AXE client's demand, and return the output to the AXE client. To understand the implementation well, a small introduction to the inter-processor communication is given in the next section.

2.5.4.4 e300-AXE Inter-Processor Communication

The inter-processor communication covers all the communication between an application and the PPC-AXE driver on one side, and the AXE scheduler and its tasks on the other. All the communication is message-based. Although the structure of a message is quite flexible, data is not supposed to be sent within a message. This is due to the fact that the message memory should be allocated for a defined number of messages at the beginning of the e300 core application. This accelerates the communication, since the allocation is normally slow. Hence, adding data to messages would require a huge amount of preliminary-allocated memory. So the data memory should be allocated separately and the message should contain just a pointer to that memory.

The principle structure of a message consists of:

- A recipient ID. This is an unique identifier that identifies the AXE task or the e300 core process which is supposed to receive the message. The recipient ID of an AXE task is created by the AXE scheduler at the moment a task is loaded. The e300 core process IDs have to be given by the programmer.
- A sender ID. This is a unique identifier that identifies the sending AXE task or e300 core process.
- A message type. This identifies the type of the message. The receiving task or process will react to the message according to the message type. The message types are defined by the programmer.
- The message size. This identifies the size of the message in bytes including the message body.
- The message body. The message body can be zero or can contain whatever information that should be transferred to the recipient.

There is a small set of system messages:

- The load-task message is a message that is sent to the AXE scheduler to instruct it to load a binary AXE task. The response to this message contains the assigned AXE task ID.
- The start-task message is sent to the scheduler to start a loaded task.
- A message reply is sent by a task to the e300 core process in response to a foregoing message. The message reply contains a pointer to the message, sent by the e300 core process. It indicates the e300 core process that the message has been processed and that the e300 core process can free the message memory.

NOTE

The memory for the load-task message and the start-task message is allocated by the PPC-AXE driver.

This introduction does not cover all the inter-processor communication aspects and messages but it is sufficient for this application note. For more information reffer to further documentation.

Figure 10 shows a typical inter-processor communication flow chart of a task's lifecycle:

1. The e300 core process first allocates a defined number of memory slots for messages.

2. Then it loads the binary task.bin to the shared memory and sends a load-task message to the scheduler. For system messages no memory allocation is needed.
3. Now the e300 core process can start the task with a start-task message.
4. The e300 core process allocates input and output memory for the task's algorithm.
5. The e300 core process requests a free message slot.
6. The e300 core process sends a message to the task to process the data.
7. When the data is processed, the e300 core process can free the memory slot, the input data, and read the output for further use.
8. The task can now be unloaded. However, it can stay in the scheduler for further use too.

Detailed Description

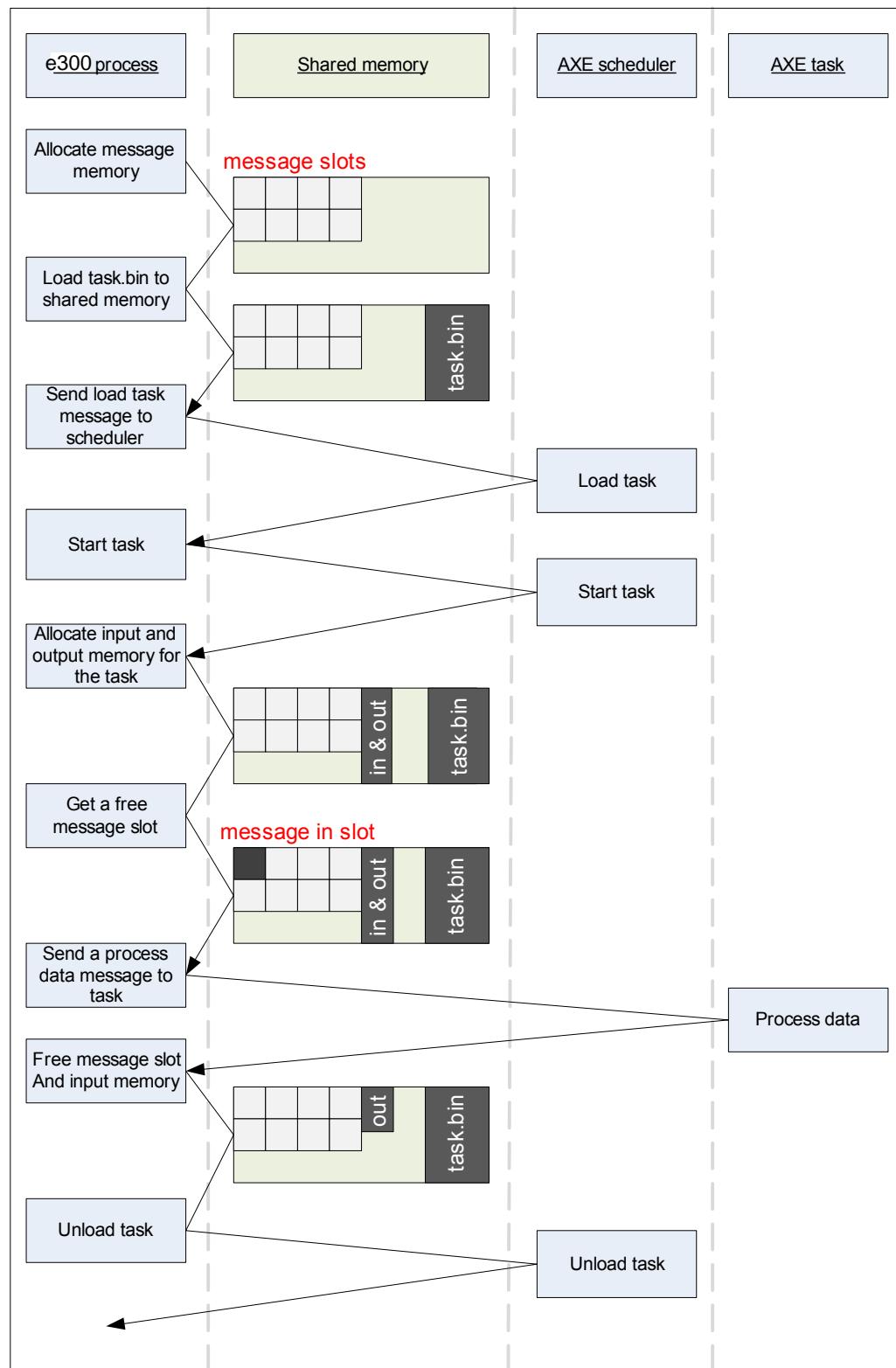


Figure 10.

2.5.4.5 Preparing the AXE Scheduler Task for the IPC

The example FIR task is restricted to receive just one kind of message that passes to the task, the four algorithm inputs, and indicates the start of the calculation. However, the proposed architecture will make it easy for any developer to enhance the task's function.

To prepare the AXE scheduler task for the IPC, three extensions from the AXE stand-alone task are needed, and are discussed in this section:

- A shared header file that defines the valid message formats.
- An endless loop waiting for the incoming messages.
- A message handler that decides the further procedure of the messages' content.

2.5.4.5.1 Creating a Shared Header for the IPC

In the first step a new header file is created. It defines the messages that the task can handle. In this example this header will be called fir_ipc.h. Due to the fact that fir_ipc.h will use types defined in fir.h, it must be included below fir.h.

To define the message needed, the basic AXE message type (MsgType) is used in the message's header. MsgType is defined in the axetypes.h header, shipped with the scheduler. This file is also in the folder zip:\AXE scheduler files\include. The file axetype.h makes use of the header axecfg.h, which specifies the configuration values for the scheduler. To include those files in the AXE scheduler FIR filter task project follow the following steps:

1. Copy the files axetypes.h and axecfg.h from the .zip file to the AXE scheduler task source folder.
2. Right click the folder Sources in the Files tab.
3. Choose Add Files.
4. Browse to the Sources folder.
5. Choose the axetypes.h and axecfg.h files and click Open.

Create the fir_ipc.h header file as described in [Section 2.5.4.3, “Merging the AXE Stand-Alone Project to the AXE Scheduler Task”](#) and include it together with the axetypes.h in the main.c file. Don't include the axetypes.h header directly to the fir_ipc.h header, since this file is shared with the e300 core application, which will use a slightly different header to define MsgType. The file axetypes.h defines the type AXEInt32, already defined in the main.c. This definition is needed for the AXE stand-alone task, so it can't be deleted. It will be defined just for the AXE stand-alone task. On the other hand, MsgType and other IPC-related codes are not relevant to the AXE stand-alone task, so the resulting main.c declaration sector should look like this:

```
//main.c
#include "taskcfg.h"

#ifndef STAND_ALONE
typedef int AXEInt32;
#endif /* STAND_ALONE */

#ifndef AXE_SCHEDULER_TASK
#include "axetypes.h"
#endif /* AXE_SCHEDULER_TASK */
```

Detailed Description

```
#include "fir.h"

#ifndef AXE_SCHEDULER_TASK
#include "fir_ipc.h"
#endif /* AXE_SCHEDULER_TASK */

#ifndef STAND_ALONE
#include "firdata.h"
#endif /* STAND_ALONE */

#include <stdio.h>
#include <string.h>
```

Now, add a structure to fir_ipc.h to define the message that will initiate the calculation:

```
//fir_ipc.h
#define CALCMMSG (200 | (1 << 31))
typedef struct
{
    MsgType header;      /* the header */
    int iBufferCount;   /* count of entries in the input buffer.*/
    __aacFirInput aacInput; /* pointer to the input buffer*/
    __aacFirOutput aacOutput; /* pointer to the output buffer*/
    __sFirState *psFirState; /* pointer to the state struct*/
    __sFirParams *psFirParams; /* pointer to the parameter struct*/
} sCalcMessage;
```

MsgType is defined in axetypes.h and looks like this:

```
typedef struct
{
    TaskType recipientID; /* message recipient */
    TaskType senderID;   /* message sender */
    Uint32 type;         /* message type */
    Uint32 size;         /* size of remain message part (in bytes) */
} MsgType;
```

NOTE

All user messages have a leading one in the message type ID. The IDs with leading zeros are reserved for the system messages. The codec API uses message IDs smaller than 200 | (1 << 31), so start numbering your messages starting with 200 to avoid collisions with the codec API.

2.5.4.5.2 Creating an Endless Loop to Wait for Messages

The task's activity is message-driven only. This means the task waits until a message arrives, processes the message, and goes back into the waiting state.

NOTE

The scheduler puts the task into suspended mode while it is waiting for a message and awakes it when a message arrives. This way a waiting task consumes no processor load.

This waiting loop is implemented by an endless while loop.

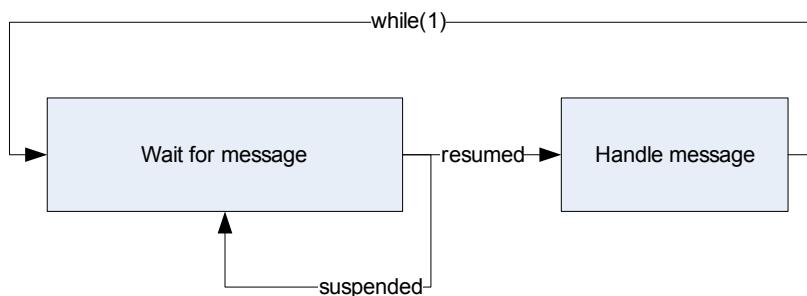


Figure 11. Endless Loop of the AXE Scheduler Task

The loop runs in the main() function. Remember that no memory for psMsg has to be allocated, since it is allocated by the calling e300 core application.

```

...
#ifndef STAND_ALONE
ifa_fir(myDynInput, &myDynParams, &myDynState
       , myDynOutput, BUFFER_COUNT);
while (1) {} // loop forever
#endif //STAND_ALONE
...
#ifndef AXE_SCHEDULER_TASK
MsgType *psMsg;

while (1)
{
    ReceiveMessage(&psMsg);
    handleIncomingMessage(psMsg);
}
#endif /* AXE_SCHEDULER_TASK */
  
```

Again, this loop has to be put in a #ifdef clause, since it is only relevant to the AXE scheduler task. The same it is for the ifa_fir() function call, since it is just called from the main function in the case of an AXE stand-alone task. The function ReceiveMessage(MsgType** refMsg) is a part of the AXE scheduler API and handleIncomingMessage(MsgType * refMsg) still has to be defined.

To enable the AXE scheduler API calls, axeapi.h has to be included. You'll find that file in the folder zip:\AXE scheduler files\include. Include axeapi.h to the AXE scheduler task project, as done before with axetypes.h (see [Section 2.5.4.5.1, “Creating a Shared Header for the IPC”](#)), and include it in the main.c file:

```

...
#ifndef AXE_SCHEDULER_TASK
#include "axetypes.h"
#include "axeapi.h"
#endif /* AXE_SCHEDULER_TASK */
...
  
```

NOTE

In axeapi.h the type RefMsgType is a pointer to the MsgType. The ReceiveMessage() is defined for RefMsgType. However, in this application note just MsgType and MsgType* are used, since RefMsgType and MsgType* can be used equivalently.

2.5.4.5.3 Creating the Message Handler

As mentioned before, no message handler is needed, since there is just one message. But to provide scalability a structure is needed, where adding new messages would be very easy. The message handler takes a message as an input. The output is void.

The function declaration is added after the include statements. In this occasion a function called handleError() is declared too. We'll come to that later.

```
//main.c
...
#ifndef AXE_SCHEDULER_TASK
void handleIncomingMessage(MsgType* psMsg);
void handleError(char* err);
#endif /* AXE_SCHEDULER_TASK */
```

The function body has three main responsibilities:

- To delegate messages to handling functions.
- To respond to the calling e300 core application that the message was handled.
- To detect errors due to wrong messages.

The first job is done by a switch statement that switches due to the message type, sent within the message header. If the message type is CALCMMSG, the ifa_fir() function is called. The second one is done by the AXE scheduler API function call MessageIsProcessed(...). This method sends a response to an incoming message to the e300 core application. The argument of the function is the sent message. This way the e300 core application knows that the message was processed. This means for the FIR example that all of the input values have been filtered.

If no valid message type is delivered, the switch will go to the default case, to execute a well-defined error handling. In this example the error handling is restricted to a simple-console output, realized by the AXE scheduler API function CPrint(). The CPrint() is a message-based, restricted function, alike to printf() function. Refer to further documentation for more information.

```
#ifndef AXE_SCHEDULER_TASK
void handleIncomingMessage(MsgType* psMsg)
{
    switch (psMsg->type)
    {
        case CALCMMSG:
            ifa_fir(
                (sCalcMessage*) psMsg)->aacInput,
                (sCalcMessage*) psMsg)->psFirParams,
                (sCalcMessage*) psMsg)->psFirState,
                (sCalcMessage*) psMsg)->aacOutput,
                (sCalcMessage*) psMsg)->iBufferCount
            );
    }
}
```

```

        break;
    default:
        handleError("No valid message type");

}

MessageIsProcessed(psMsg);

}

void handleError(char* err)
{
    CPrint(err);
}
#endif /* AXE_SCHEDULER_TASK */

```

Now the code can be compiled into the AXE stand-alone project just like in the AXE scheduler task project. Congratulations! You have just created a working AXE scheduler FIR filter task.

2.5.4.6 Debugging Utilities

Debugging of an AXE scheduler task must be done indirectly. The algorithm itself can be debugged using the AXE stand-alone task, but the IPC functionality cannot be debugged in the usual way. However, the CPrint() function gives at least the opportunity to send strings to a console. This is helpful, since problems with the IPC can always occur.

Since algorithm issues have been resolved using the AXE stand-alone task, debugging an AXE scheduler task is mostly related with correct data transmission and interpretation. Usual errors might be wrongly passed pointers, badly initialized buffers, wrongly passed values, and so on. Here the debugging tools library (libdebtools.a) is very helpful. This section shows how to include and use that small library.

To include the debugging tools into the library, copy the files deb_tools.h and libdebtools.a from the folders zip:\AXE debugging tools\include and zip:\AXE debugging tool\lib respectively to your AXE scheduler task source folder. Include them with a right click on Source in the Files tab as you did in [Section 2.5.4.5.1, “Creating a Shared Header for the IPC”](#) with the axeapi.h and axetypes.h.

The library provides these two functions:

- printmem:

Syntax	void printmem(void* <pInput>, int <length>)
Parameters (In)	<pInput> - pointer to memory to be printed <length> - number of bytes to be printed
Parameters (Out)	None
Description	Prints the content of the specified memory locations, in hex format, to the console
Return Value	None

- inttostr:

Syntax	char* inttostr(int I, char* buf, int <length>)
--------	--

Detailed Description

Parameters (In)	<i> - signed integer <buf> - a pointer to a buffer where the string will be written to <length> - the size of the buffer in bytes
Parameters (Out)	none
Description	Converts a signed integer to a zero terminated string
Return Value	A pointer to the first char of the zero-terminated string representing the integer

Example of usage:

```
int i = 100;
char myString[5];
__accum myAccumArray[2] = {1.0, 10.0};

CPrint("i: ");
CPrint(inttostr(i, myString, 5));
CPrint("\nmyAccumArray: ");
printmem(myAccumArray, 2*sizeof(__accum));

output:
i: 100
myAccumArray: ??010000000?????0A0000000????
```

NOTE

The __accum is neither defined for the first eight bits nor for the last 17 bits of a 64-bit array (see [Section 2.5.1, “AXE Fixed-Point Extension”](#)). So the ? signs can't be previewed and depend on what was in the memory before.

2.5.4.7 Deploying an AXE Scheduler Task to the AXE Scheduler

The output file CodeWarrior creates is an .elf file, but the AXE scheduler needs a .bin file. To convert the output file copy the AXEPostLinker.exe from the related .zip file to your AXE scheduler task project folder.

The syntax for the AXEPostLinker.exe for the MS command prompt is:

```
...\\<AXESchedulerTaskProjectFolder> AXEPostLinker.exe debug.elf
```

where debug.elf is the output file of the CodeWarrior project. The post-linker will create the task's binary debug.bin. This file has to be copied to the network file system of the ADS5121 on user's Linux OS host.

For example:

```
.../ltib-mpc5121ads-20080528/rootfs/tmp
```

NOTE

Use the CodeWarrior BatchRunner Postlinker to make the post linking and file copying automatic with a batch script. Use the Linux OS samba server to enable direct file writing to the NFS folder.

2.6 The e300 Core Application for the AXE-FIR Filter Task

This section describes how to write and debug the e300 core application with CodeWarrior that runs on the Linux OS BSP. It is shown how to use the PPC-AXE driver to run the AXE scheduler FIR-filter task from the e300 core application.

Furthermore, it is discussed how to convert from floating-point format to fixed-point using a provided library.

Creating a Linux OS Application with CodeWarrior

This section guides you step by step through the process of creating a CodeWarrior project for an application that runs on Linux OS on the ADS5121.

NOTE

It is required that you have Linux running on your ADS5121 with a network file system (NFS) on your Linux OS host. The Linux OS host, the CodeWarrior host, and the ADS5121 have to be in the same network as described in [Section 2.4, “Development Environment Setup.”](#)

CodeWarrior uses the ethernet network to debug the application. To enable ethernet debugging, the program Apptrk has to be running on the target. The Apptrk is a package that is deployed with the LTIB by default. However, if for some reason you have disabled the Apptrk, go to the LTIB configuration menu, select the Apptrk from the package list, and rebuild it.

On the target console type the following command to enable the Apptrk on port 1000:

```
~# apptrk :1000 &
```

Follow the following steps to create a Linux OS application project with CodeWarrior:

1. Open CodeWarrior.
2. Select File — New.
3. Select EPPC New Project Wizard in the Project tab.
4. Type the project name, for example PPC-AXE task loader.
5. Set your preferred location for the project folder.
6. Press OK.
7. Select the EPPC Linux GNU Linker and press Next.
8. Select Application as Application and C as the Language and press Next.
9. Select the gcc-3.3.2-glibc-2.3.2 toolchain and press Next.
10. Select the 52xx family as the Processor Type and press Next.
11. Select the EPPC Linux CodeWarrior TRK connection.
12. Set the hostname to the target IP followed by a colon and the Apptrk’s port number you’ve set before. For example 192.168.0.123:1000.
13. Press Next.
14. Accept the /tmp folder as the download location on the Linux target by pressing Finish.
15. Press F5 to build and debug the project.

Detailed Description

Congratulations! You are running an application on the ADS5121 and you are debugging it with CodeWarrior via the ethernet.

2.6.1 Adding the PPC-AXE Driver to the e300 Core Project

To load and run the AXE scheduler task on the AXE scheduler the PPC-AXE driver is needed. The PPC-AXE driver is freeware and it is available in the BSP. In this application note a static library is used to provide the PPC-AXE driver. This way it is much easier to integrate it to the PPC-AXE task loader project. Check the folder zip:\PPC AXE driver\ to find the necessary files:

PPC-AXE driver header files are:

- os_depend.h
- ppc-axe-driver.h
- axe-ipc.h
- axecfg.h

PPC-AXE driver library is the file libaxe.a.

Copy the header files to the PPC-AXE task loader project source folder and include it as done before.

To include the libaxe.a library follow the following steps:

1. Copy libaxe.a to the PPC-AXE task loader project source folder.
2. Open the PPC-AXE task loader project.
3. Select Edit — Application Debug Settings.
4. Expand the Linker node in the Target Settings panel.
5. Select GNU Linker.

Type the following to the Libraries text field to include the PPC-AXE driver library:

“<yourProjectSourceFolder>\libaxe.a”

The file libaxe.a uses both the pthread and the rt library. To include the pthread and the rt libraries to the project, add the following to the Libraries text field:

-lrt -lpthread

[Figure 11](#) shows the example configuration for the GNU Linker. Building the project should not result in any errors.

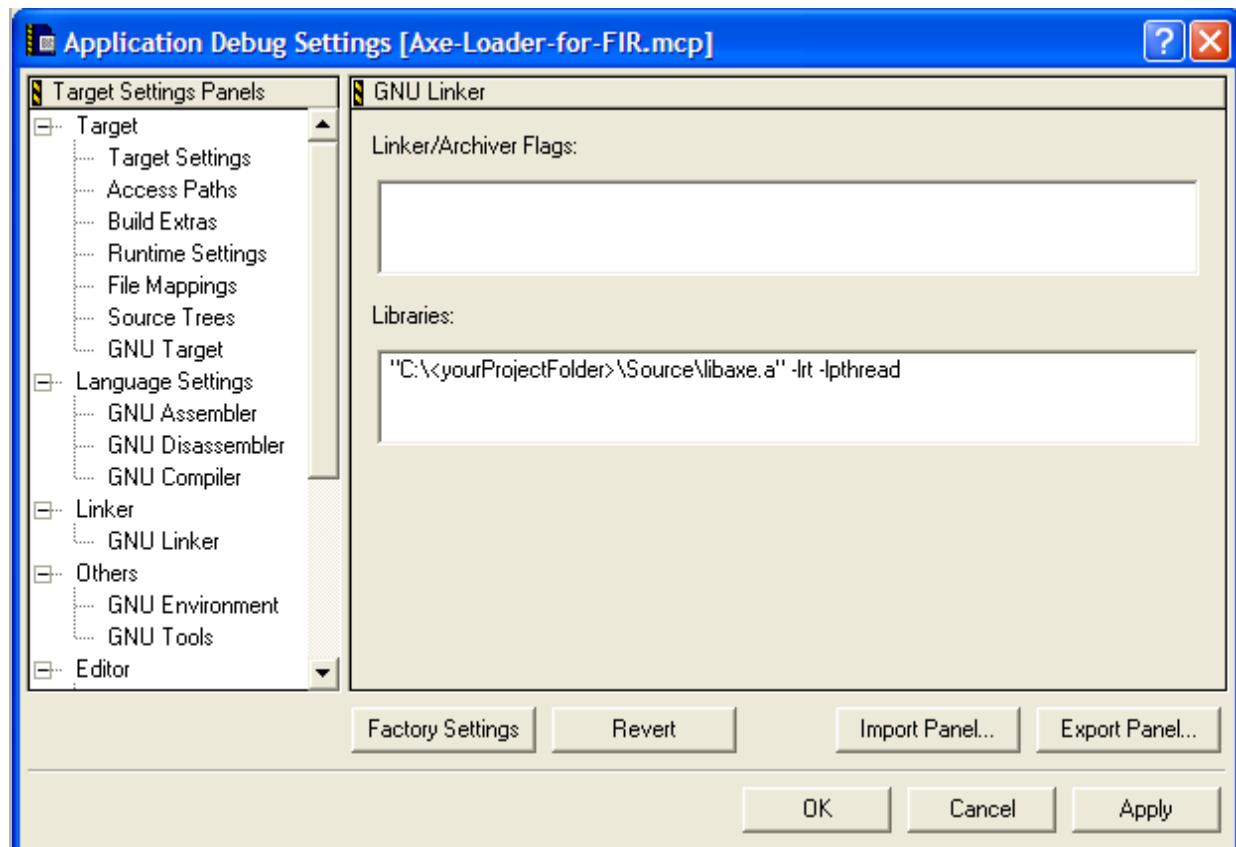


Figure 12. GNU Linker Configuration

Include the header files into the main.c file:

```
//main.c
...
#include "os_depend.h"
```

NOTE

The explanation of the PPC-AXE driver API will be given in further documentation. However, the following sections will explain enough to understand the used functions. You can also browse the PPC-AXE driver header files to find the function documentation.

2.6.2 Including the Shared Headers from the AXE Scheduler Task

Besides the PPC-AXE driver, the PPC-AXE task loader needs the header files of the AXE scheduler FIR filter task. Those header files are:

- fir.h for the algorithms entities
- fir-ipc.h for the IPC message declarations

To include the fir.h file, right click on the Source folder in the File tab, select Add File, and browse to the AXE stand-alone FIR filter task source folder. Select fir.h and press Open. Repeat this for the fir_ipc.h

Detailed Description

header file, which is located in the AXE scheduler task source folder. Also the header file commontypes.h is needed. It can be found in zip:\PPC additional files\ . The commontypes.h header is the e300 core-side counterpart to the axetypes.h header. Copy this header to the PPC-AXE task loader source folder and include it as usual.

```
//main.c
...
#include "os_depend.h"
#include "commontypes.h"

#include "fir.h"
#include "fir_ipc.h"
```

NOTE

The project won't compile now, since __accum and __fixed used in the fir.h header haven't been defined yet. These types will be defined in the next step.

2.6.3 Including the Data-Type Conversion Library

The e300 core application has a floating-point unit, while the AXE uses a fixed-point number. There are various strategies to overcome this problem. In this application note, type conversion is done on the e300 core side using the library libaxetypconv.a, which can be found in zip:\type conversion library\PPC\lib. This library can be included just like the libaxe.a library (see [Section 2.6.1, “Adding the PPC-AXE Driver to the e300 Core Project”](#)).

However, to increase the variety of possibilities, a different way to include libraries is presented now. Instead of including the library directly to the project, we'll make it available to the GNU Linker. Then the GNU Linker is instructed to link the project with the according library.

Follow the following steps:

1. Browse to your CodeWarrior installation folder. Normally it would be C:\Program Files\Freescale\CodeWarrior for MobileGT V9.0\.
2. From here on, browse to ...\\Cross_Tools\\binary\\Metrowerks\\ppc\\tools\\gcc-3.3.2-glibc-2.3.2\\powerpc-linux\.
3. Copy the file axedatatype-conv.h to the ...\\include folder. You'll find this file in zip:\\type conversion library\\PPC\\include.
4. Copy the file libaxetypconv.a to the ...\\lib folder. You'll find this file in zip:\\type conversion library\\PPC\\lib.
5. Add -laxetypconv to the GNU Linker Libraries textbox as previously done in [Section 2.6.1, “Adding the PPC-AXE Driver to the e300 Core Project.”](#)
6. Include the axedatatype-conv.h header to main.c, above the fir.h header:

```
//main.c
...
#include "os_depend.h"
#include "commontypes.h"
#include "axedatatype-conv.h"
#include "fir.h"
#include "fir_ipc.h"
```

NOTE

The suffix `-laxetypconv` instructs the linker to search in its lib folder for the library file `libaxetypeconv.a`. As seen, the suffix ‘lib’ is replaced by ‘l’. So, any library included this way needs the prefix ‘lib’.

In `axedatatype-conv.h` the datatypes `_float`, `_double`, `_fixed`, and `_accum` are defined on the basis of integers to provide compatibility, since the source of the library can be compiled for the e300 core and AXE. So the conversion from a known type to an unknown type is always done on the basis of integer manipulation.

Below, the type-conversion API is listed:

- `_doubleToAccum`:

Syntax	<code>int _doubleToAccum(_double *pD, _accum *pA)</code>
Parameters (In)	<code><pD></code> - pointer to the <code>_double</code> that has to be converted
Parameters (Out)	<code><pA></code> - pointer to an <code>_accum</code> where the result is stored
Description	Converts a <code>double</code> cast to <code>_double</code> to a <code>_accum</code> This method is just without data loss for $0 \leq \text{exponent}(\text{double}) < 128$ if $\text{exponent} \geq 127$ it fails and returns -1 if $\text{exponent} < 0$ the mantissa is cut by $ \text{exponent} $ bits
Return Value	-1 if double is out of range, 0 else

- `_accumToDouble`:

Syntax	<code>int _accumToDouble (_accum *pA , _double *pD)</code>
Parameters (In)	<code><pA></code> - pointer to the <code>_accum</code> that has to be converted
Parameters (Out)	<code><pD></code> - pointer to an <code>_double</code> where the result is stored
Description	Converts a <code>_accum</code> to a <code>_double</code>
Return Value	none

- `_doubleToFixed`:

Syntax	<code>int _doubleToFixed (_double *pD , _fixed *pF)</code>
Parameters (In)	<code><pD></code> - pointer to the <code>_double</code> that has to be converted
Parameters (Out)	<code><pF></code> - pointer to an <code>_fixed</code> where the result is stored
Description	Converts a <code>_double</code> to a <code>_fixed</code> This method is without data loss for exponent of double = -1 It returns a failure for exponent of double > -1 It suffers loss of accuracy for exponent of double < -1
Return Value	-1 if double is out of range, 0 else

Example of usage:

```
double d = 0.5;
__accum __a;

__doubleToAccum( (__double*) &d, &__a);
...
__accumToDouble(&__a, ( __double*) &d);
```

Detailed Description

```
printf("double: %1.2f", d);
```

Output:

```
double: 0.50
```

2.6.4 PPC-AXE Task Loader Application Structure

Now everything is prepared to start with the implementation of the PPC-AXE task loader. The upper [Section 2.6.4, “PPC-AXE Task Loader Application Structure”](#) illustrates the core structure of the application. It is kept very simple:

1. The PPC-AXE driver is initialized.
2. The AXE-scheduler FIR-filter task is loaded into the AXE scheduler.
3. The task is started. It will now wait for a message.
4. The CALCMSG (see [Section 2.5.4, “Creating the AXE Scheduler FIR Filter Task”](#)) is sent to the task to run the algorithm.
5. When the task has finished, the result is printed to a console.
6. The task is unloaded.

The lower part of [Figure 13](#) shows the steps that have to be followed in order to enable the steps of the core structure:

1. The path of the task binary is needed to load the task.
2. Memory for the task has to be allocated.
3. Memory for the task’s memory units has to be allocated and initialized and test data has to be created.
4. Memory for the CALCMSG message has to be allocated and the message has to be initialized.

In the following section, the PPC-AXE task loader implementation will be explained, based on the previously described structure.

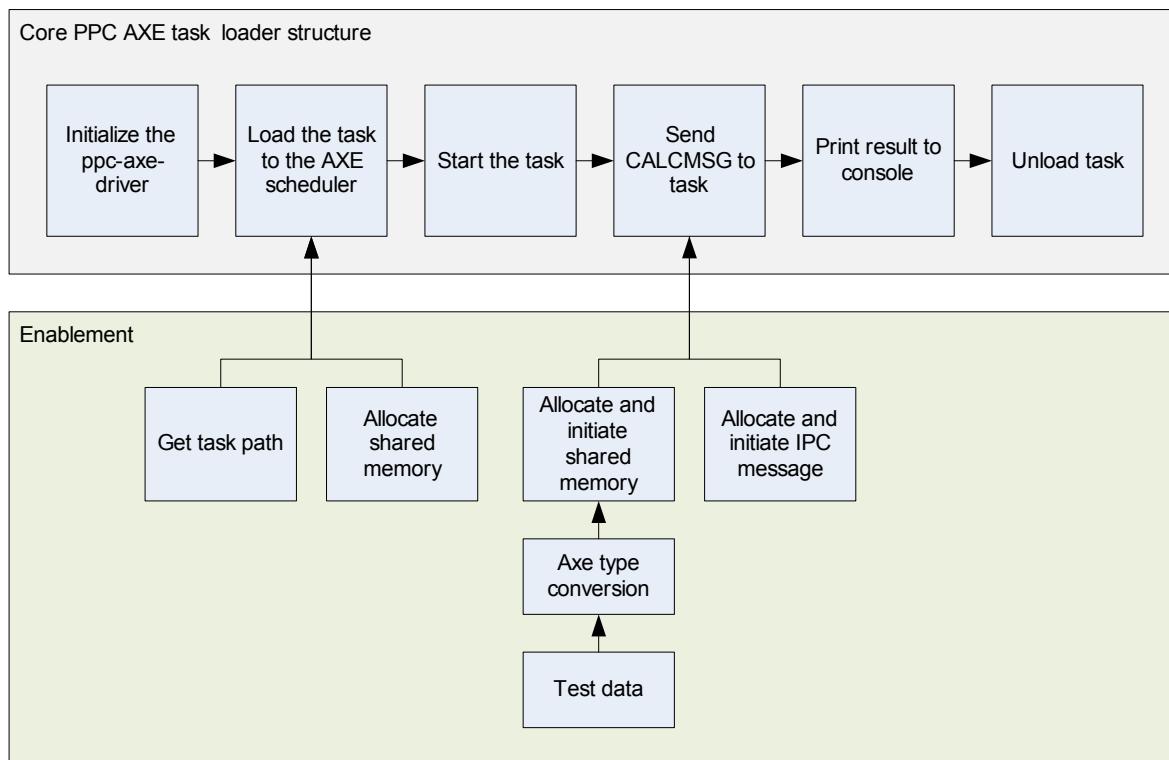


Figure 13. Core PPC-AXE Task Loader Structure

2.6.5 Administration of the Shared Memory

It is needed to clean up all allocated shared memory manually. If you don't do so, the memory stays marked as allocated after the application ends. This is a common source of segmentation errors. Shared memory also has to be freed when an error occurs and the application ends in an unexpected way. The number of the shared memory objects can easily grow high within one application. To let this application exit on an error at any time without leaving the memory allocated in the stack, it is indispensable to trace the shared-memory allocations globally. In the provided solution, shared-memory allocation is done by a function that allocates the shared memory and registers it to a global array. This way, all shared memory can be freed at any time by iterating through the array.

In this application, four functions are introduced to provide an easy API to that problem:

- Function `initAxeMemObjects` — initializes a global array to trace the shared memory.
- Function `exitAxeMemObjects` — frees all traced shared-memory objects.
- Function `addAxeMemObject` — allocates the shared memory and registers it to the global array.
- Function `removeAxeMemObject` — frees the shared memory and removes it from the global array.

The usage of those functions is explained during the course of this document. Add the following code to your `main.c` file of your PPC-AXE task loader project to provide the global array:

```
//main.c
...
#define MAX_AXE_MEM_OBJ 40 // maximal number of shared memory objects
```

Detailed Description

```
/*
 * a struct for a shared memory object
 * pMem: pointer to the memory
 * size: size in bytes
 * region: external or internal memory
 * name: an identifier name
 */
typedef struct
{
    void* pMem;
    int size;
    int region;
    char *name;
} sAxeMemObject;

sAxeMemObject* gapAxeMemObjects[MAX_AXE_MEM_OBJ]; //the global array
```

To implement the functions (read the zip:\CW projects\ppc-axe-task-loader\Source\main.c file for code documentation):

```
//main.c
...
void initAxeMemObjects()
{
    int i = 0;
    for(i=0; i<MAX_AXE_MEM_OBJ; i++)
    {
        gapAxeMemObjects[i] = NULL;
    }

    return;
}

void exitAxeMemObjects()
{
    int i = 0;
    for(i=0; i<MAX_AXE_MEM_OBJ; i++)
    {
        if(gapAxeMemObjects[i] != NULL)
        {
            AxeFree(gapAxeMemObjects[i]->pMem
                    ,gapAxeMemObjects[i]->size
                    ,gapAxeMemObjects[i]->region);

            printf(gapAxeMemObjects[i]->name);
            printf("\n");
            gapAxeMemObjects[i] = NULL;
        }
    }

    return;
}

int addAxeMemObject(void **ptr, unsigned size, int region, char *name)
{
    int i = 0;
    int status;
    for(i = 0; i<MAX_AXE_MEM_OBJ; i++)
```

```

{
    if(gapAxeMemObjects[i] == NULL)
    {
        status = AxeMAlloc(ptr, size, region);
        gapAxeMemObjects[i] = (sAxeMemObject*)
            malloc(sizeof(sAxeMemObject));
        gapAxeMemObjects[i]->pMem = *ptr;
        gapAxeMemObjects[i]->size = size;
        gapAxeMemObjects[i]->region = region;
        gapAxeMemObjects[i]->name = name;

        printf("added at %d: %s \n", i, name);
        return status;
    }
}
return -1;
}

int removeAxeMemoObject(void* pMem)
{
    int i = 0;
    for(i = 0; i<MAX_AXE_MEM_OBJ; i++)
    {
        if(gapAxeMemObjects[i] !=NULL
           && gapAxeMemObjects[i]->pMem == pMem)
        {

            AxeFree(gapAxeMemObjects[i]->pMem
                   ,gapAxeMemObjects[i]->size
                   ,gapAxeMemObjects[i]->region);

            gapAxeMemObjects[i] = NULL;
            return i;
        }
    }
    return -1;
}

```

Now add initAxeMemObjects() to the main function:

```

//main.c
...
int main(){
    initAxeMemObjects();
    //printf("Welcome to CodeWarrior!\r\n");← delete that default statement
    return 0;
}

```

NOTE

AxeMAlloc and AxeFree are calls of the PPC-AXE driver API to allocate and free the shared memory. Refer to further documentation for more details or browse the os_depend.h header file.

2.6.6 Initializing the PPC-AXE Driver

The PPC-AXE driver has to be initialized. This is done by calling the function of the PPC-AXE driver API:

```
//main.c  
...  
int main(){  
    AxeClientInit(); // initializes the ppc-axe-driver  
    initAxeMemObjects();  
    return 0;  
}
```

2.6.7 Loading a Task to the AXE Scheduler

This section shows how to load the AXE-scheduler FIR-filter task to the AXE scheduler:

- How to get the path of the task's binary with CodeWarrior.
- How to allocate the shared memory for the task.
- How to finally load the task to the scheduler.

2.6.7.1 Getting the Path of the Task's Binary with CodeWarrior

To load a task, its path and filename are needed. In this application note main(int argc, char *argv[]) is used to pass the path and filename. The argument can easily be added in the CodeWarrior using the Runtime Settings:

1. Open the Application Debug Settings as done before.
2. Expand the Target node.
3. Select Runtime Settings.
4. Type the filename and path within your NFS of your AXE-scheduler FIR-filter task in the Program Arguments text file (see [Figure 14](#)). For example: /tmp/debug.bin.

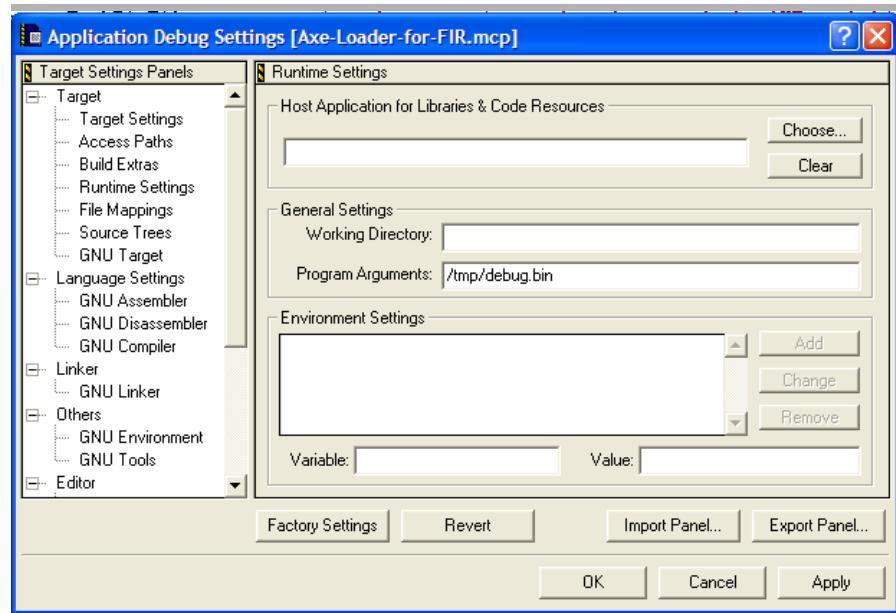


Figure 14. Setting the Program Arguments

Add the following code to your main.c file to make the path accessible for the rest of the program and to detect whether an argument was passed or not:

```
//main.c
...
int main(int argc,  char *argv[])
{
    ...
    char *pcTaskBinFile;// string that contains path and name of the
    if (argc>0) // GET PASSED ARGUMENTS
    {
        pcTaskBinFile = argv[1];
    }
    else
    {
        printf("not sufficient arguments");
        return 0;
    }
    initAxeMemObjects ();
    ...
}
```

2.6.7.2 Allocating the Memory for the AXE Scheduler Task

To allocate the memory for the AXE scheduler task, the addAxeMemObject function is used. This function will call the PPC-AXE driver function AxeMalloc. The AxeMalloc takes three arguments. The first argument is a reference to a pointer that should be set to the allocated memory. The second argument specifies the size of memory that should be allocated in bytes. The third argument indicates the region, where the memory should be allocated:

Detailed Description

```
//axe-ipc.h
...
#define AXE_MALLOC_ANY      0
#define AXE_MALLOC_INTERNAL1
#define AXE_MALLOC_EXTERNAL2
#define AXE_MALLOC_IRAM      3
```

In addition to the memory allocation, addAxeMemobject registers the memory object to the global memory-tracing array. To allocate the right size of memory, the binary-task file has to be opened first, to get the necessary file information. This is done in a function. The following function takes a path, opens the indicated file, allocates the shared memory, and copies the file to the allocated shared memory. It returns the pointer to the shared memory. Add the code to your main.c file:

```
//main.c
...
/* libraries to handle files */
#include <fcntl.h>
#include <sys/stat.h>
...
void *readBinFile(char *path)
{
    int fd;
    struct stat st;
    int rv = 0;
    int size;
    void *buf;

    printf("loading axe binary from %s\n", path);

    fd = open(path, O_RDONLY);
    if (fd < 0) {
        return NULL;
    }

    if (fstat(fd, &st) !=0) {
        close(fd);
        return NULL;
    }
    if (!S_ISREG(st.st_mode)) {
        close(fd);
        return NULL;
    }
    size = st.st_size;
    printf("buffersize: %d\n", size);
    addAxeMemObject((void **) &buf, size, AXE_MALLOC_EXTERNAL, path);
    read(fd, buf, size);

    close(fd);

    return buf;
}
```

2.6.7.3 Loading the Task

Loading the task to the scheduler is now easy. The readBinFile function is used to read the binary task file and to write it into the shared memory. The return value of the function — the pointer to that memory segment — is now passed to the PPC-AXE driver API function AXELoadTask. Besides the pointer, AXELoadTask takes an integer value that indicates the priority of the task. In this application note the priority is set by default to one. AXELoadTask loads the task into the AXE scheduler. The return value of AXELoadTask is the unique task ID, which is assigned to the task by the AXE scheduler.

Add the following code to your main.c file below the readBinFile function:

```
//main.c
...
void *readBinFile(char *path)
{
    ...
}

void loadTaskBinToAxe(char *path, int *pid)
{
    int status;
    void *taskfilebuf;

    taskfilebuf = readBinFile(path);
    status = AXELoadTask((const void *)taskfilebuf, 1);
    printf("AXELoadTask returned status code %d\n", status);

    if (status < AXE_E_OK) {
        printf("failure in loadTaskBinToAxe");
    }

    *pid = status;
}
```

Now call this function from the main function to load the task. To do so, the integer taskID1 has to be declared first:

```
int main(int argc, char *argv[])
{
    int taskID1;
    ...
    AxeClientInit();
    initAxeMemObjects();

    // LOAD AXE TASK BINARIES
    loadTaskBinToAxe(pcTaskBinFile, &taskID1);

    /* controlled exit on error */
    if(taskID1 < AXE_E_OK)
    {
        exitAxeMemObjects();
        return 0;
    }
    ...
}
```

2.6.8 Starting the AXE Scheduler Task

Once the task is loaded, starting the task is very easy. The PPC-AXE driver API provides a function called AXEStartTask which takes the task's ID to run it. If a task is started, it runs its main function. In this example the AXE scheduler FIR filter task will run until the ReceiveMessage statement is reached, then it's put to the suspend state until a message is received. If a task is started successfully, the AXEStartTask returns to zero.

To start the task, add the following code to the main function:

```
//main.c
...
int main(int argc,  char *argv[])
{
    int status;
    ...
    if(taskID1 < AXE_E_OK)
    {
        exitAxeMemObjects();
        return 0;
    }

    if(status = AXEStartTask(taskID1))
    {
        printf("AXEStartTask returned error number %d\n", status);
        exitAxeMemObjects();
        return 0;
    }
    ...
}
```

2.6.9 Sending CALCMSG to the AXE Scheduler Task

The AXE scheduler FIR filter task is now in the suspended state and waiting for the CALCMSG message. On the e300 core side, this message should now be sent, but before this could be done, some preparatory work has to be completed:

1. Test data has to be created.
2. Shared memory has to be allocated for the input and output buffer, FIR state, and FIR parameters.
3. The test data has to be type-converted and stored into the shared memory.
4. The message has to be created in the shared memory.

These steps are covered in the following section.

2.6.9.1 Creating the Test Data

To keep this example as simple as possible, the test data is created within a header file. In this example the same coefficients are taken as before in the AXE stand-alone FIR filter task, to make the results comparable. As the input data, an impulse is sent. As mentioned in [Section 2.5.3, “Creating an AXE Stand-Alone FIR Filter Task,”](#) the expected FIR-filtered responses to an impulse are the coefficients. This way it is very easy to prove the algorithm is working well. However, those values can be changed without

constraint as long as COEF_SIZE and INPUT_SIZE indicate the correct array size of aCoef and aInput1 respectively.

Create and include a header file called data.h as done in [Section 2.5.4.3, “Merging the AXE Stand-Alone Project to the AXE Scheduler Task.”](#) Add the data to the data.h file:

```
//data.h
#define COEF_SIZE 21
double aCoef[] =
{
    -0.001735144473, 0.013643184950, 0.001676807357, -0.023932013009,
    -0.002149939904, 0.046582863582, 0.002641550601, -0.095116080119,
    -0.002960254194, 0.314519426629, 0.503071878724, 0.314519426629,
    -0.002960254194, -0.095116080119, 0.002641550601, 0.046582863582,
    -0.002149939904, -0.023932013009, 0.001676807357, 0.013643184950,
    -0.001735144473
};

#define INPUT_SIZE 40
double aInput1[] =
{
    1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
};
```

Include the data.h header to the main.c file:

```
//main.c
...
#include "data.h"
...
```

2.6.9.2 Allocating the Memory

Memory has to be allocated for the input and output buffers, FIR parameters, and FIR state. The coefficients-array memory for the FIR parameters has to be allocated, since the allocation of the FIR parameters structure memory just allocates the memory for a pointer to the actual array. The same applies for the FIR state structure and the actual state array.

To allocate the memory, the variables have to be declared first in the main function. The names for the variables are:

- pInput1 — a pointer to an array of __accums. Since the type conversion is done on the e300 core side, the input values are passed directly as __accums to the filter.
- pOutput1 — a pointer to an array of __accums. Here applies the same as for the input. The type conversion is done on the e300 core side, so the output of the algorithm is of __accum type.
- psFirState1 — a pointer to a FIR state structure
- pState1 — a pointer to an array of __accums. Actually the state type doesn't matter, since the e300 core application, excluding its initialization, never reads or writes to the states. This is done exclusively by the FIR filter. However, the FIR filter internally works with __accum type for the states, so it makes the code more readable. Other algorithms might have more complex state types. As long as the accurate state array size is known, the inner structure can be totally opaque to the e300 core application.

Detailed Description

- `psFirParams1` — a pointer to the FIR parameters structure
- `pCoefs1` — a pointer to an array of `_fixed`. Due to the fact that the FIR coefficients always are in the range of $-1 < \text{coefficient} < +1$, the `_fixed` type is sufficient. The type conversion is done on the e300 core side again.

Declare the variables at the beginning of the main function:

```
//main.c
...
int main(int argc, char *argv[]) {
    ...
    __accum *pInput1,
            *pOutput1;

    __sFirState *psFirState1;
    __accum *pState1;

    __sFirParams *psFirParams1;
    __fixed *pCoefs1;
    ...
}
```

To allocate the memory, the size is needed, and also the decision whether external or internal memory will be used. The sizes are determined either by the `sizeof()` function for structures or by the defined sizes in the `data.h` header. The size for the output memory hasn't been defined yet, but it makes sense to make it as big as the input memory size. Within this application note, the external memory is chosen only.

NOTE

The size of the state array is determined by the size of the coefficients minus one. See [Figure 5](#).

To allocate the memory, the `addAxeMemObject` function is used as before, but we'll summarize its output to a variable called `status`. If the status at the end of all allocations doesn't equal zero, an allocation error has occurred. This summation avoids the status to be checked six times.

Add the following code to the main function to allocate the memory properly:

```
//main()
...
int status = 0;
...
status += addAxeMemObject(
    (void*) &pInput1
    , sizeof(__accum)*INPUT_SIZE
    , AXE_MALLOC_EXTERNAL, "input1"
);
status += addAxeMemObject(
    (void*) &pOutput1
    , sizeof(__accum)*INPUT_SIZE
    , AXE_MALLOC_EXTERNAL, "output1"
);
status += addAxeMemObject(
    (void*) &psFirParams1
    , sizeof(__sFirParams)
    , AXE_MALLOC_EXTERNAL, "firparams1"
);
```

```

status += addAxeMemObject(
(void*) &pCoefs1
, sizeof(__fixed)*COEF_SIZE
, AXE_MALLOC_EXTERNAL, "coefs1"
);
status += addAxeMemObject(
(void*) &psFirState1, sizeof(__sFirState)
, AXE_MALLOC_EXTERNAL
, "firstate1"
);
status += addAxeMemObject(
(void*) &pState1
, sizeof(__accum)*(COEF_SIZE-1)
, AXE_MALLOC_EXTERNAL, "states1"
);
if(status!=0)
{
    printf("Allocation error! Can't allocate memory\n");
    exitAxeMemObjects();
    return 0;
}
...

```

2.6.9.3 Type Conversion and Memory Initialization

This section shows how to initialize the shared memory. On one hand the coefficients and the input data have to be type-converted and written to the shared memory. On the other hand the output buffer and the state array are set to zero and the parameter and state structures' values are set. Input data and coefficients are written in loops, where each test data value is separately converted to __accum and __fixed respectively. For the conversion, the libaxatypeconv.a library is used (see [Section 2.6.3, “Including the Data-Type Conversion Library”](#)). Therefore, the pointer to the double of the test-data array has to be cast to a pointer to __double and __fixed respectively. For the for-loop the integer i has to be declared. Add this code to the main function:

```

//main()
...
int taskID1, i;
...
for(i=0; i<INPUT_SIZE; i++)
{
    __doubleToAccum((__double*) (aInput1 + i) , (pInput1 + i));
}

for(i=0;i<COEF_SIZE;i++)
{
    __doubleToFixed((__double*) (aCoef + i),(pCoefs1 + i));
}
The memset function is used to set both the output buffer and the state array to zero:

```

```

...
memset(pOutput1, 0x00, sizeof(__accum)*INPUT_SIZE);
memset(pState1, 0x00, sizeof(__accum)*(COEF_SIZE-1));
...
```

Now the values of the parameter and the state structure can be set:

Detailed Description

```

...
psFirParams1->iCoefCount = COEF_SIZE;
psFirParams1->afiCoef = AxeMapVirtToPhys(pCoefs1);

psFirState1->iCircBufOffset = 0;
psFirState1->iFilterStages = COEF_SIZE - 1;
psFirState1->aacZ = AxeMapVirtToPhys(pState1);
...

```

2.6.9.4 Creating a Message in the Shared Memory

The allocation of memory in general is very time consuming. As mentioned before in [Section 2.6.5, “Administration of the Shared Memory,”](#) in this application note it is proposed to allocate the memory for a set of messages preliminarily. During a process a lot of messages can be sent, so having the memory allocated in advance really increases the performance. By means of the function initMsgAlloc() at the beginning of the application, a defined number of message slots with a defined size is allocated. The maximum number is defined by MAX_MSGS, the maximum message size by MAX_MSG_SIZE. To assign and free a slot for a message, the functions allocMsg and freeMsg are used.

Copy this code to the main.c file to enable preliminary message-memory allocation:

```

//main.c
...
#define MAX_MSG_SIZE 36
#define MAX_MSGS 8
...
MsgRefType gapMsgFreeList[MAX_MSGS];
...
MsgRefType allocMsg (void)
{
    MsgRefType pMsg = NULL;
    int i;

    for (i = 0; i < MAX_MSGS; i++)
    {
        pMsg = gapMsgFreeList[i];
        if (pMsg != NULL)
        {
            gapMsgFreeList[i] = NULL;
            break;
        }
    }
    return pMsg;
}

void freeMsg (MsgRefType pMsgIn)
{
    MsgRefType pMsg = NULL;
    int i;

    for (i = 0; i < MAX_MSGS; i++)
    {
        pMsg = gapMsgFreeList[i];
        if (pMsg == NULL)
        {

```

```

        gapMsgFreeList[i] = pMsgIn;
        break;
    }
}
return;
}

void initMsgAlloc (void)
{
    int i;
    for (i = 0; i < MAX_MSGS; i++)
    {
        int status;
        void *ptr;

        status = addAxeMemObject(&ptr, MAX_MSG_SIZE
                               , AXE_MALLOC_EXTERNAL, "msg");

        if ((status != AXE_E_OK) || (ptr == NULL))
            printf("failed to allocate memory to message array\n");
        gapMsgFreeList[i] = (MsgRefType)ptr;
    }
}

```

Add the initMsgAlloc function to the beginning of the main function and allocate a slot for the calculation message:

```

//main()
...
sCalcMessage *psCalcMsg1;
...
initAxeMemObjects();
initMsgAlloc();
...
psCalcMsg1 = (sCalcMessage*) allocMsg();

```

2.6.9.5 Filling an Allocated Message

Now that the message memory is allocated, it can be filled with content. In this example case there are some pointers to memory that should be sent. The e300 core application uses a virtual memory through the Linux OS, while the AXE uses the physical addresses. So, there is a need to map from the virtual to the physical memory and vice versa. The PPC-AXE driver provides these functions:

- AxeMapPhysToVirt(...) — takes a pointer to physical memory and returns a pointer to virtual memory.
- AxeMapVirtToPhys(...) — takes a pointer to the virtual memory and returns a pointer to the physical memory.

Detailed Description

With these functions, filling the message with its content becomes quite easy and all the steps below should be clear to the reader:

```
//main()
...
psCalcMsg1->header.recipientID = taskID1;
psCalcMsg1->header.senderID = 100;
psCalcMsg1->header.type      = (unsigned int) CALCMSG;
psCalcMsg1->header.size     = sizeof(sCalcMessage);
psCalcMsg1->iBufferCount   = INPUT_SIZE;
psCalcMsg1->aacInput        = AxeMapVirtToPhys(pInput1);
psCalcMsg1->aacOutput       = AxeMapVirtToPhys(pOutput1);
psCalcMsg1->psFirState     = AxeMapVirtToPhys(psFirState1);
psCalcMsg1->psFirParams    = AxeMapVirtToPhys(psFirParams1);
...
...
```

2.6.10 Sending the CALCMSG Message and Receiving a Response

Now the message can be sent. The AXE scheduler will receive the message and extract the recipient ID. Due to the recipient ID, the AXE scheduler resumes the AXE scheduler FIR filter task, which was suspended and left waiting for a message. The task resumes in the main function in the `ReceiveMessage(&psMsg)` statement. The AXE scheduler sets the pointer of the received message to `psMsg`. The task takes the message, and its handler will decide due to the message type (CALCMSG) that it should execute the `ifa_fir` function to run the filter.

While the AXE task filters the input, the e300 core application goes into `AXEReceiveMessageReply(...)`. This function is a part of the PPC-AXE driver. It puts the e300 core application for a defined time to an inactive state, waiting for the response of a sent message. If the time expires, the function returns an error. The reception of the message response indicates that the AXE task finished the filtering. The `freeMsg(...)` function is then used to free the message slot, since the message is no longer used. [Figure 15](#) illustrates this process.

NOTE

It is absolutely not necessary to call the `AXEReceiveMessageReply(...)` directly after sending a message. The message reply can be fetched whenever wanted. Even if another message has been sent in between and its response has been fetched. This is very important since normally the e300 core application might do other calculations while the AXE task is running. So use `AXEReceiveMessageReply(...)` just for synchronization concerns.

Add this code to the main function to send the message and receive the response:

```
//main()
...
AXESendMessage((void*) psCalcMsg1);

if(AXEReceiveMessageReply(psCalcMsg1, 5000) == 0)
{
    freeMsg((MsgType*) psCalcMsg1);
}
else
{
```

```

printf("message 0x%8x (type) from task %d reply failed"
      ,psCalcMsg1->header.type
      ,psCalcMsg1->header.recipientID );
exitAxeMemObjects();
return 0;
}...

```

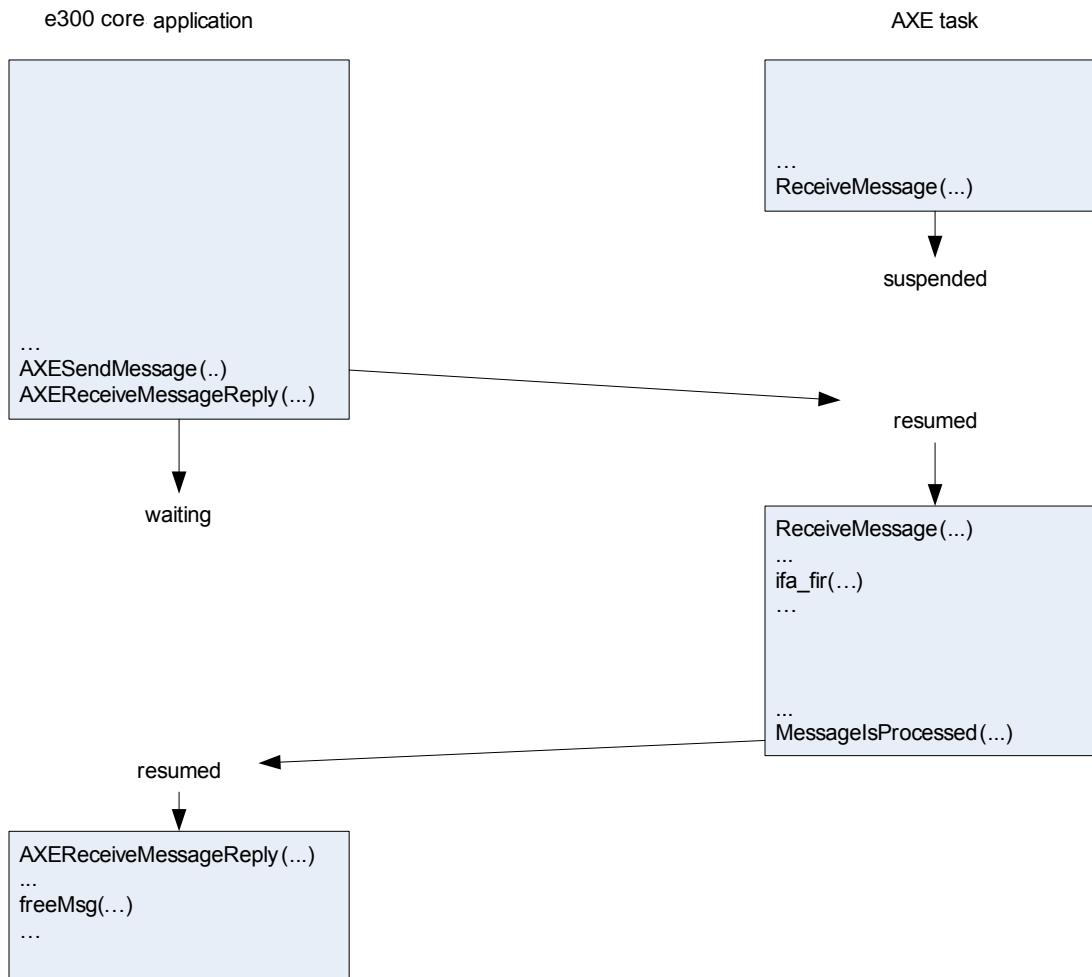


Figure 15. FIR-Message Process

2.6.11 Printing the Results and Cleaning Up

Assuming that everything went fine, we are now capable of printing out the results, unloading the task, and cleaning up the memory. The memory cleanup is easily done with the `exitAxeMemObjects()`, since all of the shared memory had been registered to the global memory array. Unloading the task is done in the same way as loading the task, passing the task ID to be unloaded to the AXE scheduler. This is done by the PPC-AXE driver function `AXEUnloadTask(...)`. To print out the result, the output array can be accessed directly. The result of the AXE scheduler FIR filter task is of the type `_accum`. So the output entries have to be reconverted to a double type. This is done by the `libaxetypconv.a` library function `_accum.ToDouble(...)`.

Detailed Description

Add this code to the main function:

```
//main()
...
double d = 0;
...
for(i=0; i<INPUT_SIZE ; i++)
{
    __accumToDouble(((__accum*) pOutput1) + i, (__double*) &d);
    printf("%4d:\t", i);
    printf("%2.5f\n", d );
}

if(AXEUnloadTask(taskID1))printf("error unloading task 1");

exitAxeMemObjects();
...
```

Congratulations! You have now finished the implementation.

2.7 Running the System

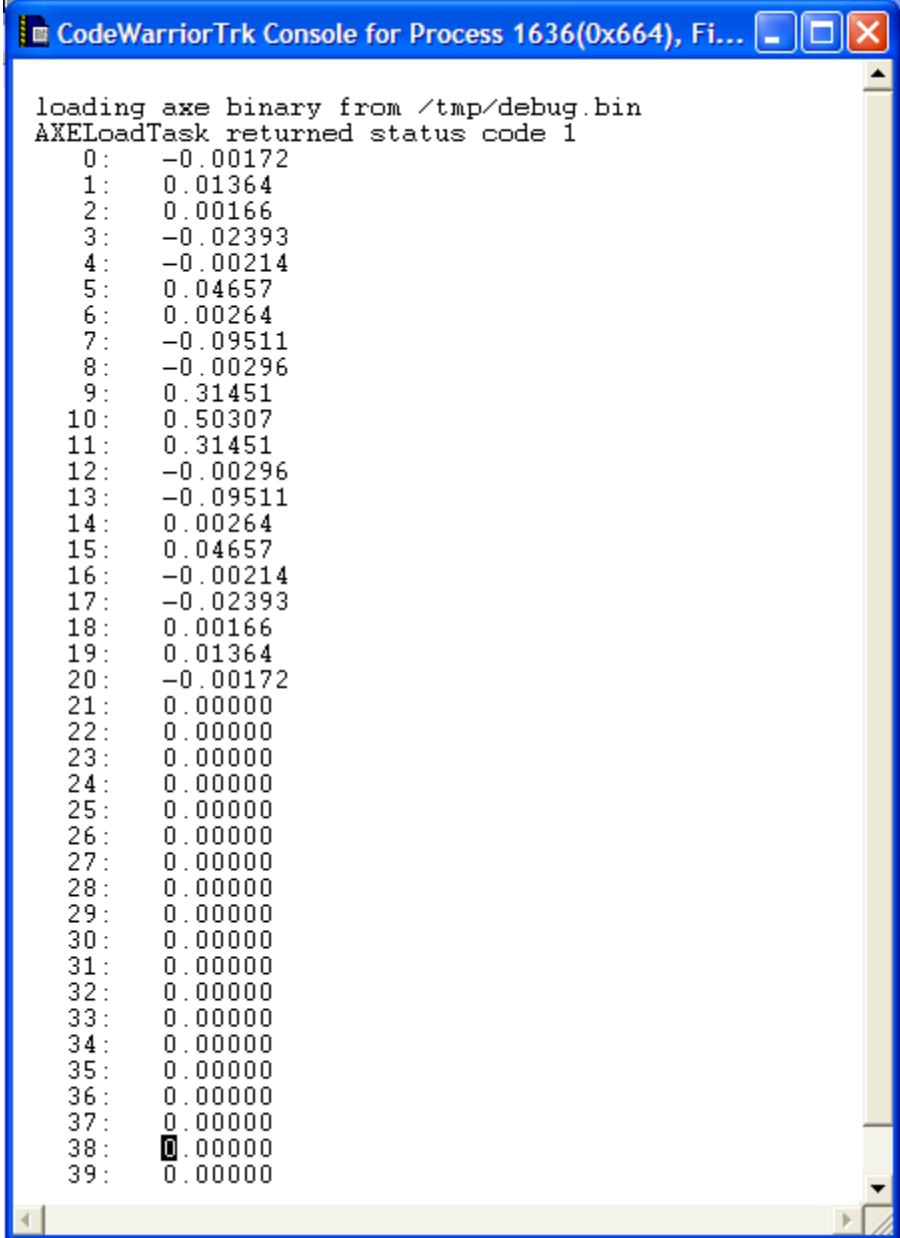
Before you run the system, make sure that:

- Linux is running on the target board.
- The AXE scheduler is installed (see [Section 2.4.2, “LTIB and AXE”](#)).
- The AXE scheduler FIR task binary (debug.bin) is in the tmp folder of the NFS filesystem.
- The IP address in the CodeWarrior Remote Debugging Settings (Connection Settings) matches the IP address of the target board (use ifconfig to check it).

To run the system follow the following steps:

1. Start the Apptrk client on the target by typing # apptrk :1000 &.
2. Start the AXE server on the target by typing # startaxeserver.
3. Start the e300 core application by pressing the F5 button.

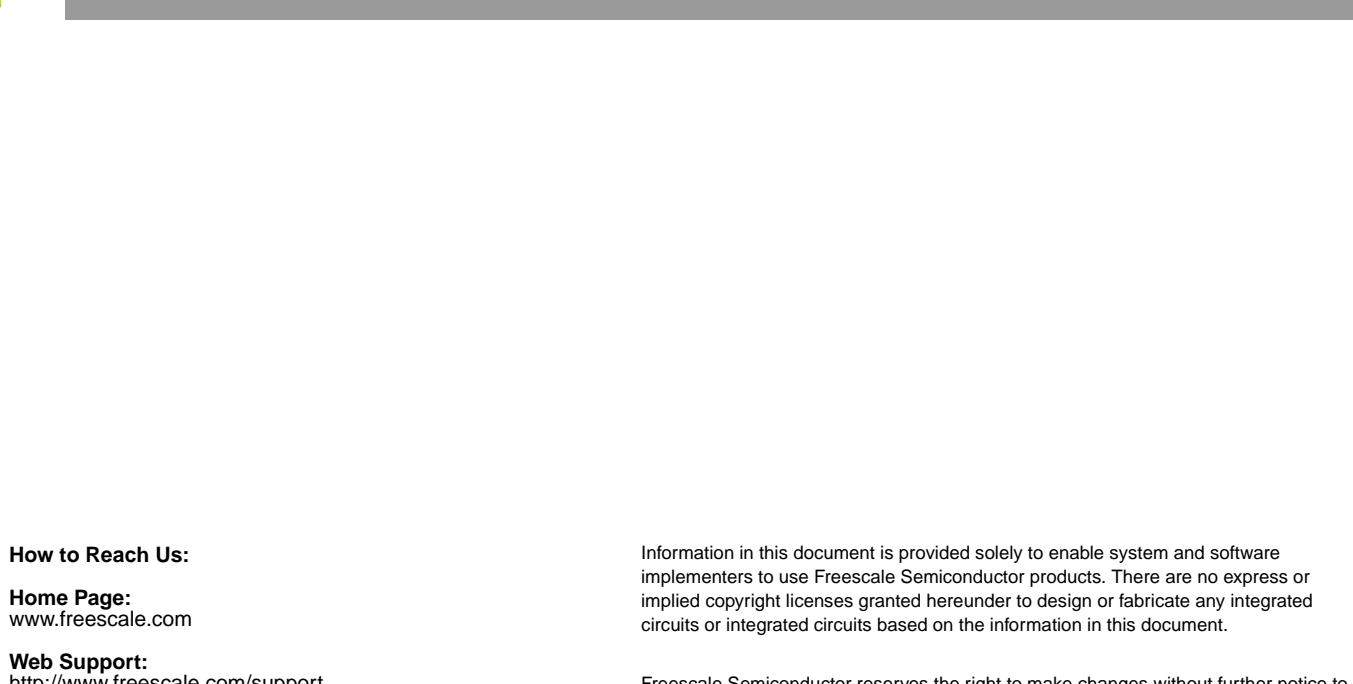
Depending on your parameters and input, you should now see something like what is shown in [Figure 16](#):



The screenshot shows a terminal window titled "CodeWarriorTrk Console for Process 1636(0x664), Fi...". The console displays the output of an FIR filter running on the AXE. The output consists of 40 lines of data, each containing a number representing the filter coefficient at index 0 to 39. The first few lines of output are:

```
loading axe binary from /tmp/debug.bin
AXELoadTask returned status code 1
0: -0.00172
1: 0.01364
2: 0.00166
3: -0.02393
4: -0.00214
5: 0.04657
6: 0.00264
7: -0.09511
8: -0.00296
9: 0.31451
10: 0.50307
11: 0.31451
12: -0.00296
13: -0.09511
14: 0.00264
15: 0.04657
16: -0.00214
17: -0.02393
18: 0.00166
19: 0.01364
20: -0.00172
21: 0.00000
22: 0.00000
23: 0.00000
24: 0.00000
25: 0.00000
26: 0.00000
27: 0.00000
28: 0.00000
29: 0.00000
30: 0.00000
31: 0.00000
32: 0.00000
33: 0.00000
34: 0.00000
35: 0.00000
36: 0.00000
37: 0.00000
38: 0.00000
39: 0.00000
```

Figure 16. FIR Output in CodeWarrior

**How to Reach Us:**

Home Page:
www.freescale.com

Web Support:
<http://www.freescale.com/support>

USA/Europe or Locations Not Listed:
Freescale Semiconductor, Inc.
Technical Information Center, EL516
2100 East Elliot Road
Tempe, Arizona 85284
+1-800-521-6274 or +1-480-768-2130
www.freescale.com/support

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
www.freescale.com/support

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064
Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor China Ltd.
Exchange Building 23F
No. 118 Jianguo Road
Chaoyang District
Beijing 100022
China
+86 10 5879 8000
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

Document Number: AN3763
Rev. 0
12/2008

Information in this document is provided solely to enable system and software implementers to use Freescale Semiconductor products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits or integrated circuits based on the information in this document.

Freescale Semiconductor reserves the right to make changes without further notice to any products herein. Freescale Semiconductor makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale Semiconductor assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. "Typical" parameters that may be provided in Freescale Semiconductor data sheets and/or specifications can and do vary in different applications and actual performance may vary over time. All operating parameters, including "Typicals", must be validated for each customer application by customer's technical experts. Freescale Semiconductor does not convey any license under its patent rights nor the rights of others. Freescale Semiconductor products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Freescale Semiconductor product could create a situation where personal injury or death may occur. Should Buyer purchase or use Freescale Semiconductor products for any such unintended or unauthorized application, Buyer shall indemnify and hold Freescale Semiconductor and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Freescale Semiconductor was negligent regarding the design or manufacture of the part.

RoHS-compliant and/or Pb-free versions of Freescale products have the functionality and electrical characteristics as their non-RoHS-compliant and/or non-Pb-free counterparts. For further information, see <http://www.freescale.com> or contact your Freescale sales representative.

For information on Freescale's Environmental Products program, go to <http://www.freescale.com/epp>.

Freescale™ and the Freescale logo are trademarks of Freescale Semiconductor, Inc. All other product or service names are the property of their respective owners. © Freescale Semiconductor, Inc. 2008. All rights reserved.