

Migrating from SDK to PE

Ed Camacho

1. Introduction

The 56800/E product family has long been supported by the CodeWarrior Development tool and the Freescale Embedded Software Development Kit (SDK). This environment has enabled a customer to quickly prototype his applications and to reduce development time by simplifying the processor peripheral learning curve. Processor Expert™ (PE) has taken the lessons learned from the development of the SDK and improved upon it to provide the next generation of CodeWarrior integrated development tools. PE targets the same development goals as SDK and provides a Graphical User Interface (GUI) that further simplifies the tools usage by providing a more user-friendly interface. It also offers an expert knowledge system that prevents the user from misconfiguring or misusing any of its software modules. PE and SDK support embedded software development with these common goals:

- Single integrated development environment (*CodeWarrior*)
- Faster time to market
- Reduce learning curve of processor-specific register programming
- Increased portability and reuse across MCU and 56800/E derivatives
- Fully tested, production-quality software modules
- Source code provided for customer optimization and customization
- High-Level Encapsulated API that promotes portability and processor migration
- Low-Level Register API for customization of High-Level
- Application-Specific Algorithm Libraries

This application note will show how these common goals are supported in SDK and PE, identify PE improvements, and demonstrate how to migrate an SDK project to PE.

Contents

1. Introduction	1
2. Common Goals	2
2.1 Single Integrated Development Environment	2
2.2 Shorten Application Development Time	4
2.3 Reduce Learning Curve	4
2.4 Increased Portability and Reuse Across MCU and 56800/E Derivatives	4
2.5 Fully Tested, Production-Quality Software Modules	4
2.6 Source Code Provided for Customer Optimization and Customization.....	4
2.7 High-Level Encapsulated API	4
2.8 Low-Level Register API	5
2.9 Application-Specific Algorithm Libraries	5
3. PE Improvements over SDK	7
3.1 Common Development Environment	7
3.2 PE Graphical User Interface	7
3.3 Improved Help System	8
3.4 Expert Error-Checking System.....	9
3.5 Object-Oriented Software Modules.....	9
3.6 Code Generation	10
3.7 Single Installation	10
4. Migrating SDK to PE	11
4.1 Migrating Application-Specific Algorithm Libraries.....	11
4.2 Migrating Low-Level Register Programming.....	13
4.3 Migrating High-Level Encapsulated Programming.....	14
5. Summary	22

2. Common Goals

SDK and PE both target the same customer goals, each achieving these goals within its own development environment. This section identifies how the two environments meet these goals.

2.1 Single Integrated Development Environment

CodeWarrior provides the framework into which both SDK and PE are integrated. Both SDK and PE provide stationery that automatically creates the development environment necessary to support the instantiation and configuration of the available software modules. This stationery makes it easy for a customer to concentrate on writing his application-specific software. A new SDK or PE project may be created by selecting *File / New* from within the CodeWarrior IDE menu to navigate to the window shown in [Figure 2-1](#). SDK stationery files are found under *Embedded SDK Stationery* and PE stationery files are found under *DSP56800E EABI Stationery*.

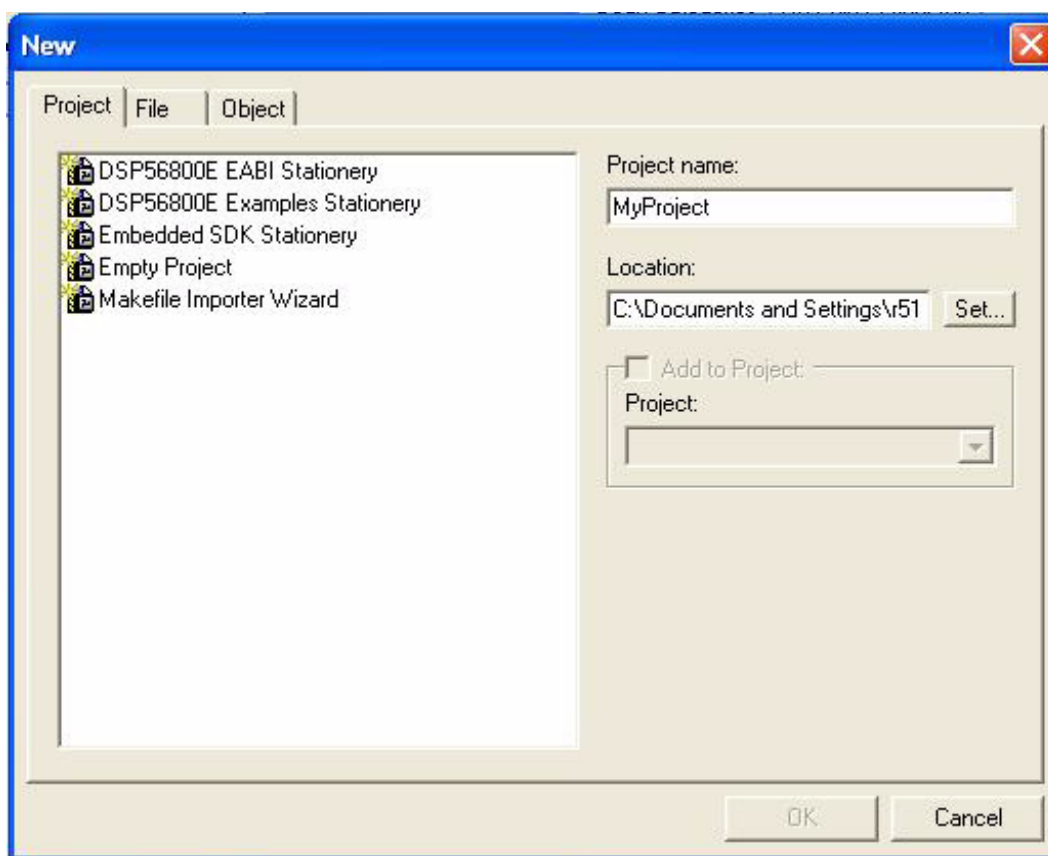


Figure 2-1. Creating a New Project

An SDK project may be created by selecting the appropriate 56800/E processor derivative and stationery, as shown in [Figure 2-2](#), then clicking OK.

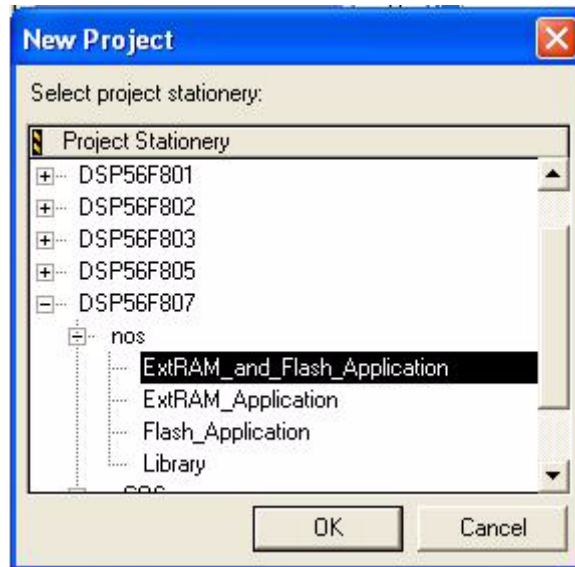


Figure 2-2. SDK Stationery

Similarly, a PE project may be created by selecting the appropriate 56800/E processor derivative and stationery, *C with Processor Expert*, as shown in [Figure 2-3](#).

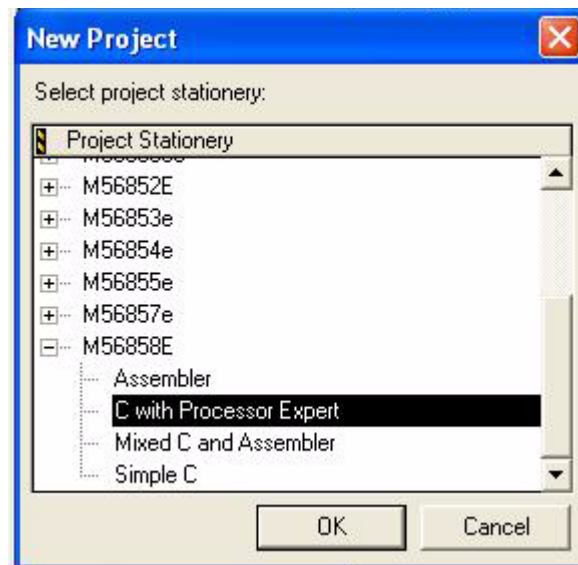


Figure 2-3. PE Stationery

2.2 Shorten Application Development Time

Both SDK and PE target a rapid prototyping environment that provides production-ready software modules, including peripheral drivers and application-specific algorithms. These software modules include APIs that are common across supported processor derivatives, increasing the level of portability. A customer is able to utilize these software module building blocks to quickly prototype his applications. Source code is provided to allow customers to further optimize and customize these prototypes to meet production performance and memory requirements.

2.3 Reduce Learning Curve

Both the SDK and PE tools provide easy-to-use APIs that make it simple for customers to begin using processor peripherals. It is no longer necessary to read through pages and pages of the chip's user manual before writing code. The encapsulated interfaces and configuration mechanisms of both SDK and PE hide the complicated, peripheral-specific control software behind the API. The customers can now focus his efforts on developing his applications, rather than learning processor-specific details.

2.4 Increased Portability and Reuse Across MCU and 56800/E Derivatives

Portability is increased by targeting common APIs across supported processor derivatives. While SDK supports only the 56800/E derivatives, PE has extended this support across Freescale's MCU 68HC08 and 68HCS12 families as well as the 56800/E devices.

"The demonstration of these integrated technologies furthers Metrowerks' goal to provide a seamless development environment for embedded applications, which brings major productivity benefits to our customers," - Metrowerks European Strategic Marketing Manager.

2.5 Fully Tested, Production-Quality Software Modules

All software modules delivered in the SDK and PE have been developed under industry-standard processes and have undergone extensive testing to ensure they are production quality. SDK software development follows industry-standard development procedures set forth in the SEI CMM and ISO 9000 standards. PE is also ISO 9000 certified in its development and testing procedures. Both tools are developed in close communication with processor design teams to ensure feature support and are constantly improved based on customer feedback. Further, PE's advanced object-oriented development methodology instantiates only the software required to perform the selected task, thereby providing the benefits of object-oriented software development without the overhead.

2.6 Source Code Provided for Customer Optimization and Customization

To meet the needs of the general development population, it is necessary to provide APIs and implementations that support all possible applications, which can sometimes lead to code growth and inefficient use of system resources. This issue is resolved in SDK and PE by providing source code for all non-proprietary software modules, giving a customer the freedom to optimize and customize these implementations to meet his specific needs.

2.7 High-Level Encapsulated API

This high-level API provides a customer with a means to select the processor that best meets the performance and cost requirements of his system. Both SDK and PE provide encapsulated interfaces to the peripherals that allow customers to easily migrate among supported processor derivatives. This ease of migration is critical when evaluating multiple processors. During the prototyping phase, it provides a developer a means to test his

prototypes on multiple processors, allowing a better understanding of the processor's performance characteristics. Once the application is functionally sound, it may be optimized to reduce the program and data footprints to the point it will fit into a smaller and more cost-effective device. SDK provides this interface via its POSIX API, while PE provides the Embedded Bean API. An example of how to migrate from SDK to PE at this level is provided in [Section 4.3](#).

2.8 Low-Level Register API

This low-level API is processor-specific and supports customers who want direct control and visibility into the register-level programming of peripherals. It mainly consists of C macro substitutions for setting and clearing bits within peripheral registers and is therefore very efficient, but at the cost of portability. It is intended to be used in conjunction with the Encapsulated API to take advantage of processor specific features. SDK provides this interface via its POSIX IO Control (*ioctl*) API, while PE provides the Processor Expert System Library (PESL) API. The PESL functions are derived from the *ioctl* command set and provide a high degree of compatibility between SDK and PE. An example of how to migrate from SDK to PE at this level is provided in [Section 4.2](#).

2.9 Application-Specific Algorithm Libraries

The SDK provides an extensive set of highly optimized, C-callable, application-specific algorithm libraries that target many of the application markets using the 56800/E processor family. These libraries provide building blocks for customer applications and can greatly shorten development cycle time, moving a product to market more quickly. All SDK libraries have been ported to PE and are available as Embedded Beans through the Bean Selector interface. The functionality, performance, and API are compatible between the SDK and PE. The available libraries can be selected from the PE Bean Selector window, as shown in [Figure 2-4](#). An example of how to migrate from SDK to PE at this level is provided in [Section 4](#).

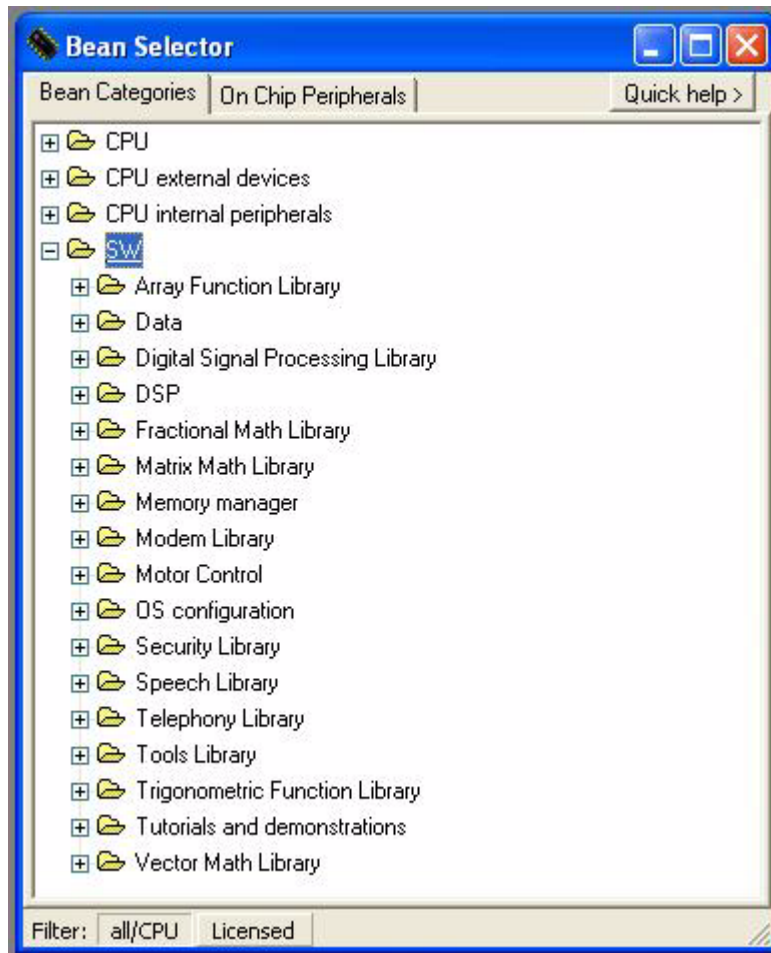


Figure 2-4. PE Bean Selector

3. PE Improvements over SDK

Freescale constantly focuses on improving our software development tools to achieve the best possible environment for our customers. In doing so, it was a natural progression to combine the best of SDK into PE to improve on the SDK environment's shortcomings, as well as to strengthen the PE environment. The merging of these tools has provided the customer with the following improvements over the SDK:

- Common development environment across Freescale's 8- and 16-bit products
- Graphical User Interface (GUI) for selecting and configuring software modules
- Code generation of selected software modules
- Expert system that checks for errors in peripheral configuration and usage
- Object-oriented software modules
- Highly integrated tool with a single installation mechanism

3.1 Common Development Environment

Processor Expert for the HC08 and HCS12 microcontroller families has been distributed with CodeWarrior and PE is now available for the 56800/E controller derivatives. Having a single common development environment improves the ease of migration among supported processors and assists in evaluation and application development. These processors provide a natural progression from the MCU application space to the hybrid MCU/DSP application space through the similarities of their peripheral sets, addressing modes, instruction sets, and, now, their development tools. PE further enhances this progression by providing common software interfaces across these processor derivatives. In the end, this common development environment improves development time by not forcing the developer to learn a new development tool. The developer is already familiar with the development environment, its design features, and its development philosophy.

3.2 PE Graphical User Interface

PE improves on the manual application configuration mechanism found in SDK by providing a GUI for selecting and configuring software modules. This improvement makes the tool much simpler to understand and use. After creating a PE project, the developer may parse through the Bean Selector, which contains all available software module APIs. These APIs are organized by Bean Category, as well as by peripheral, as shown in [Figure 3-1](#). The user may add any number of these software modules to the project by simply clicking on each. Both the Embedded Bean and PESL APIs are offered, providing the highest degree of programming flexibility to the developer.

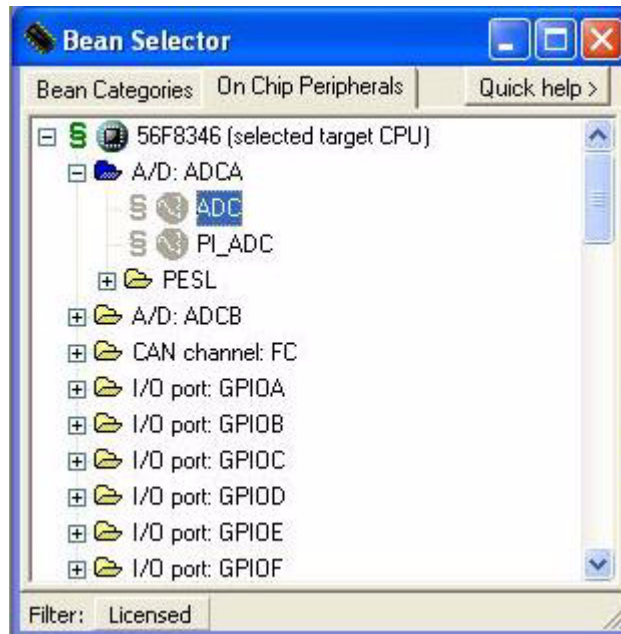


Figure 3-1. On-Chip Peripheral Embedded Bean and PESL APIs

3.3 Improved Help System

PE offers an extensive help system that provides both balloon help and online help. Balloon help appears whenever the cursor is placed over any PE module, providing quick reference information about the module’s purpose and parameters. More detailed help is available by right mouse clicking on the module and selecting *Help*. This will invoke the Processor Expert Help system, as shown in [Figure 3-2](#).

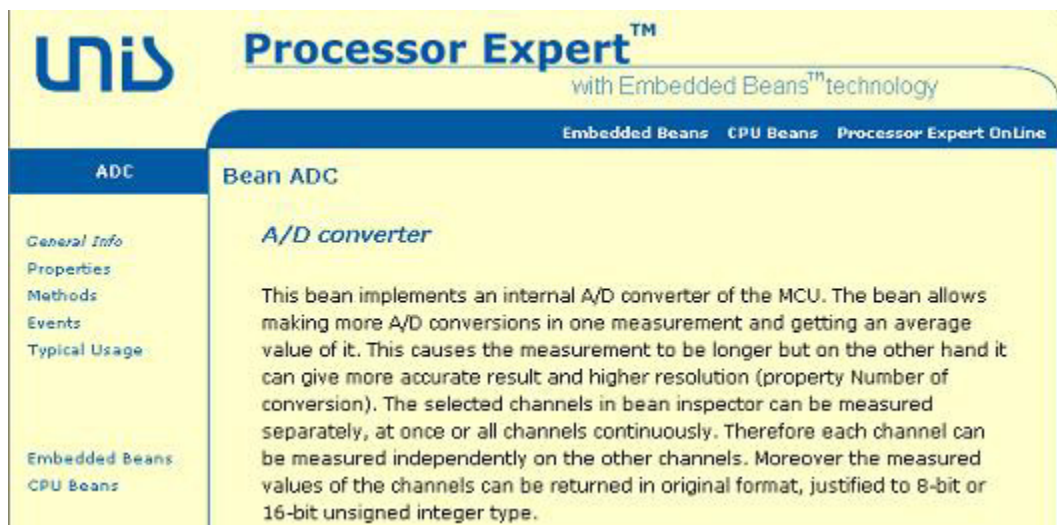


Figure 3-2. PE Help System

3.4 Expert Error-Checking System

As with many complex peripheral sets, only a finite set of register programming values are required to enable the peripheral's many features and modes. All other combinations will cause the peripheral to function improperly. PE Embedded Beans provide an expert error checking system that will flag these invalid combinations and force the user to correctly configure the peripheral before generating code. The developer is now freed from the tedious task of understanding the interaction between peripheral registers, and may instead concentrate on selecting the feature set that is required by the application. For example, [Figure 3-3](#) shows how the Expert Error Checking System notifies the developer that the ADC conversion time has not been specified, as indicated by the red exclamation point and Error window.

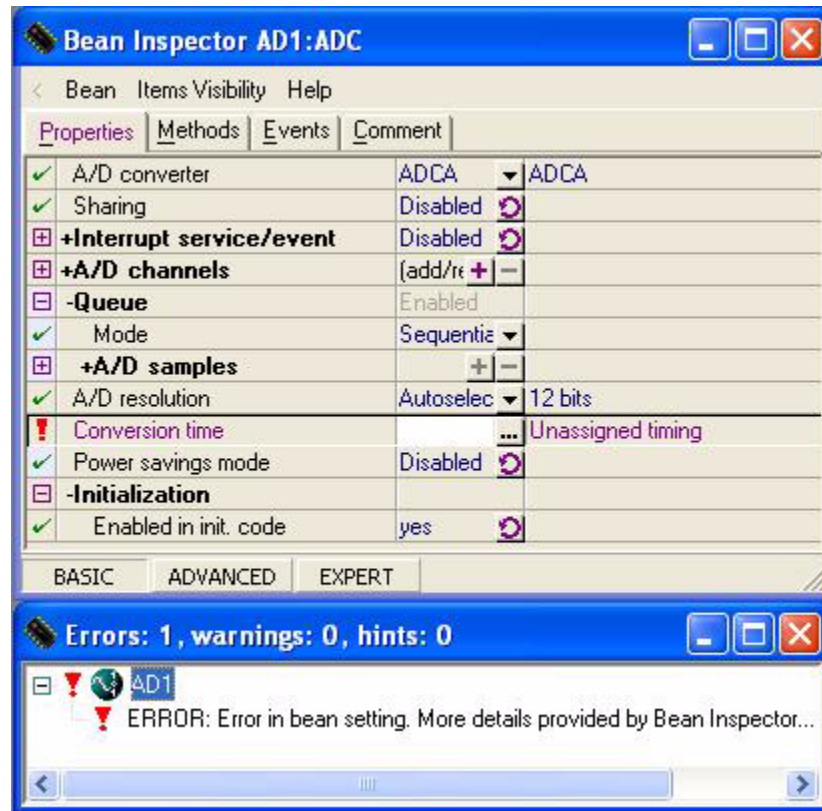


Figure 3-3. Configuration Error Checking

3.5 Object-Oriented Software Modules

PE makes it simple to instantiate multiple instances of a particular software module. Each time the Embedded Bean is selected from the Bean Selector, a new copy of the code is placed in the project with a new prefix for all methods associated with the Bean. This preserves name space and avoids any conflicts if the same function has been invoked incorrectly. A unique prefix is automatically selected by the tool and the developer has the option to redefine it. PE provides the benefits of object-oriented programming without the overhead.

3.6 Code Generation

Once the developer has selected and correctly configured all Embedded Beans required by the application, PE will automatically copy the code required to implement the selected functionality from its database into the developer's workspace and include it in the current project. This feature reduces the complexity that occurs when unneeded functions are included simply because they are part of a particular library. Even though extraneous functions can be dead-stripped by the linker from the project's executable, completely removing them from the project makes it easier for the developer to concentrate on understanding the content of only a few files which contain relevant information.

Methods associated with the selected Beans can be included into the application through PE's drag-and-drop feature, as shown in **Figure 3-4**.

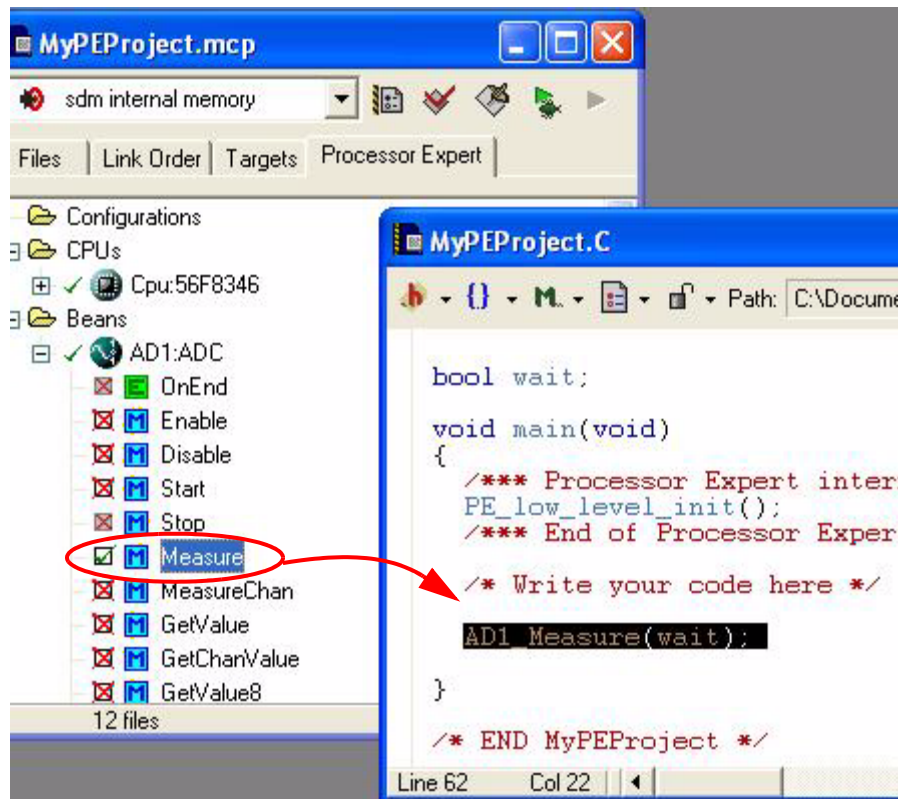


Figure 3-4. Drag-n-Drop Method Instantiation

3.7 Single Installation

The integration of CodeWarrior and PE has been improved over that of CodeWarrior and SDK via a single (combined) installation mechanism. Since CodeWarrior and SDK were installed separately, there was always the issue of maintaining compatible versions between the two packages. This issue has now been eliminated since CodeWarrior and PE are distributed as one tool.

4. Migrating SDK to PE

When migrating an SDK project to a PE project, there are three main API levels that will be encountered: Application-Specific Algorithm Library API, Low-Level Register Programming API, and High-Level Encapsulated API. This section will provide an example of how to port from SDK to PE.

4.1 Migrating Application-Specific Algorithm Libraries

The SDK Application-Specific Algorithm Libraries have all been ported to Processor Expert and provide a simple migration path between SDK and PE. Follow these steps when migrating applications that use SDK libraries to PE:

1. Create a PE project
2. Generate code to create the basic PE project structure
3. Copy the SDK application source code to PE source files
4. Select the Algorithm Library from the PE Bean Selector
5. Drag and drop appropriate methods into PE's *main.c*
6. Build the project and verify functionality

These steps are used in the following example to migrate a 56F801 SDK application to PE, which uses the Memory Manager and Array Math libraries.

1. Create a PE project using the 56F801 PE Stationery.

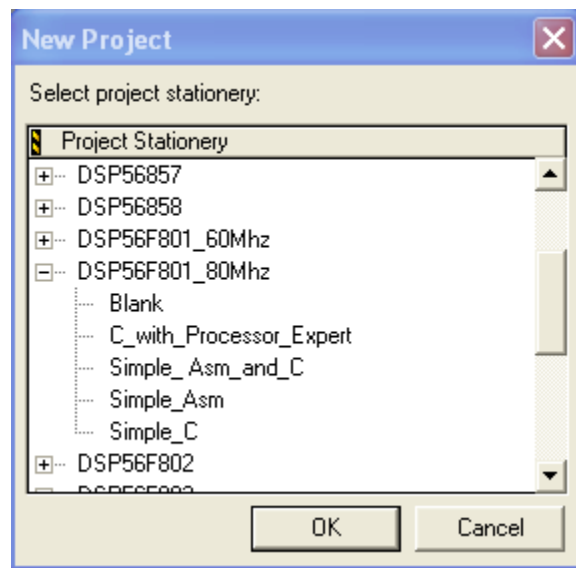


Figure 4-1. Create a 56F801 PE project

2. Generate code, which allows PE to create the basic project files associated with the 56F801 target system. **Figure 4-2** shows the PE menu selection that generates the base files; the base structure is also shown.
 - The *support* folder contains CodeWarrior basic libraries
 - The *Startup Code* folder contains source code that is run at processor reset to set-up the C programming environment; for example, setting up the software stack and initializing variables
 - The *Generated Code* folder contains all source code generated by PE and should generally not be modified
 - The *User Modules* folder contains all source code written by the user to implement the application-specific functionality
 - The *Doc* folder contains PE-generated documentation on the software selected

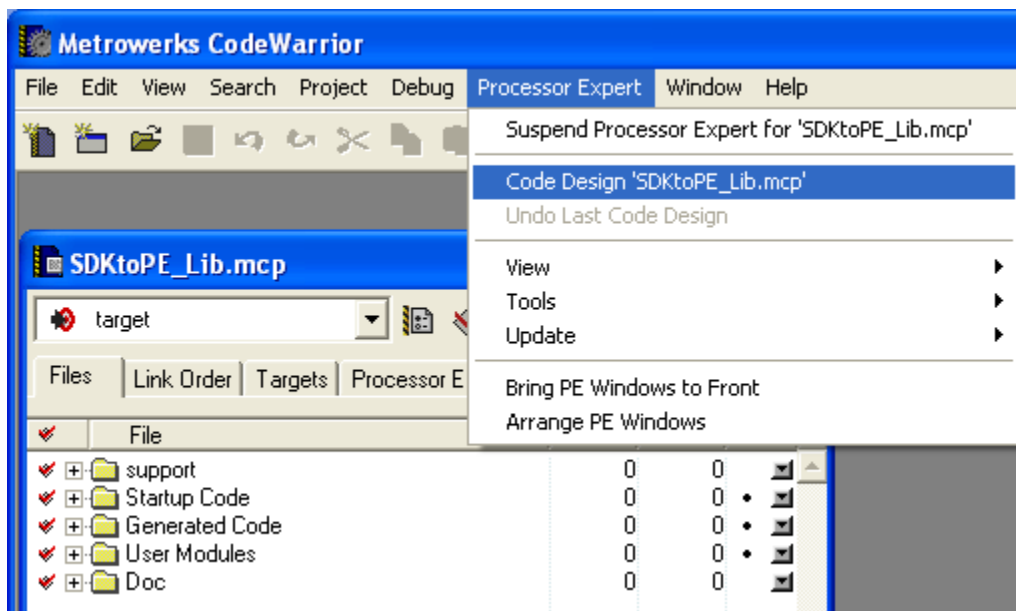


Figure 4-2. Generate PE Base Files

3. Copy the SDK main function code to the PE main function. The PE main source file (*SDKtoPE_Lib.c*) is named after the project name chosen when the project was created and may be found in the *User Modules* folder. Any other application-specific source files, like *appconst.c*, should also be added to the *User Modules* folder. When copying the SDK main function code to the PE main function, do not remove the PE initialization code located at the top of the main function; this code is needed to initialize the PE Beans selected.

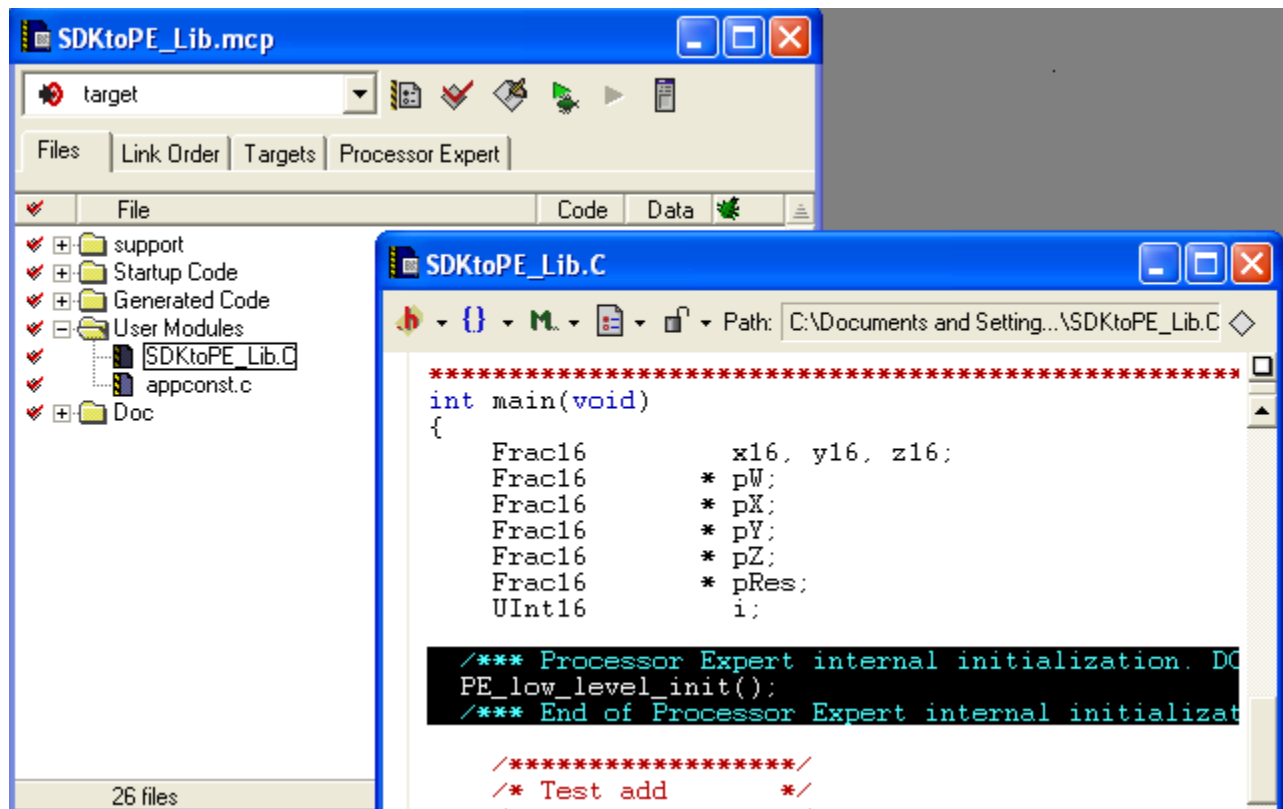


Figure 4-3. Copy SDK Application Source to PE

4. Add the Memory Manager and Array Math libraries to the PE project from the Bean Selector
5. Inspect the code being ported to determine which methods are required, then drag and drop the appropriate methods from the Bean libraries.
6. Build the project by selecting the *Make* icon. This will cause PE to generate code for the selected Bean methods; compile; assemble; and link the project. PE will automatically check for Bean configuration errors and notify the user by halting the build process and displaying the errors in the *Error* window.

The migration process is now complete and the project may be tested for functionality.

4.2 Migrating Low-Level Register Programming

The SDK provides the lowest-level programming API in its peripheral memory access interface. This interface has been ported to PE to support migration at this register programming level. The following functions define the API:

- *UWord16 periphMemRead(UWord16 *pSrc)*
- *void periphMemWrite(UWord16 Data, UWord16 *pDest)*
- *void periphBitSet(Mask, Addr)*
- *void periphBitClear(Mask, Addr)*
- *bool periphBitTest(Mask, Addr)*
- *void periphBitChange(Mask, Addr)*

The SDK maps the entire peripheral register set to a C-type defined structure, named *arch_sIO*, in the architecture definition file, *arch.h*. PE provides a similar mapping of the peripheral registers in the I/O Mapping file, *IO_Map.h*. Both SDK and PE declare a global variable, called *ArchIO*; its base address is defined in the linker command file. This variable is used to provide the peripheral memory addresses required by the peripheral memory access APIs.

While SDK uses the *ArchIO* variable to reference individual registers, PE provides a set of defined constants that can be used. The PE register definitions are named after the register names found in the chip's user's manual. Here is an example of the type of substitution that can be expected.

SDK:

```
periphMemWrite(0xbf00, &ArchIO.PwmA.OutputControlReg);
                /* ENABLE SW CONTROL */

periphMemWrite(0x0000, &ArchIO.PwmA.DisableMapping1Reg);
                /* DISABLE FAULT BITS */

periphMemWrite(0x0000, &ArchIO.PwmA.DisableMapping2Reg);
                /* DISABLE FAULT BITS */

periphMemWrite(0x000e, &ArchIO.PwmA.ConfigReg);
                /* MAKE INDEPENDENT PWMs */
```

PE:

```
periphMemWrite(0xbf00, &PWMA_PMOUT);
                /* ENABLE SW CONTROL */

periphMemWrite(0x0000, &PWMA_PMDISMAP1);
                /* DISABLE FAULT BITS */

periphMemWrite(0x0000, &PWMA_PMDISMAP2);
                /* DISABLE FAULT BITS */

periphMemWrite(0x000e, &PWMA_PMCFG);
                /* MAKE INDEPENDENT PWMs */
```

The transition between the SDK and PE for this register-level programming interface can be further understood by comparing the *arch.h* and *IO_Map.h* files.

4.3 Migrating High-Level Encapsulated Programming

To migrate an SDK project to PE at this level, a high-level overview of the functionality provided by the PE Embedded Beans is necessary. This can be understood by spending a few minutes reviewing the available Embedded Beans found in the PE Help System at *Help / Processor Expert / Embedded Beans Categories*. Understanding these interfaces will help the user select the appropriate Bean to replace the SDK API. Follow these steps when migrating SDK applications to PE:

1. Create a PE project
2. Generate code to create the basic PE project structure
3. Copy the SDK application source code to PE source files
4. Select the appropriate Embedded Bean(s) from the PE Bean Selector
5. Drag and drop appropriate methods into PE's *main.c*
6. Build the project and verify functionality

These steps are used in the following example to migrate a 56F800 SDK application to a 56F8323 PE application. This SDK application communicates with an application running on a PC via the Serial Communications Interface (SCI) to dump Program and Data memory information to the PC's display. **Figure 4-7** illustrates the GUI for this application at the end of the migration. The SDK and PC applications referred to here are included in all SDK installations, and may be found at these locations for a typical installation:

C:\Program Files\Freescale\Embedded SDK\src\dsp5680xevm\nos\applications\sci

C:\Program Files\Freescale\Embedded SDK\src\x86\win32\applications\serial\serial.exe

The first three steps in this migration are nearly identical to the process already covered in **Section 4.1**.

1. Follow Step 1 in **Section 4.1** to create the project. The PE project name used here is `8323_sci`, since this example migrates a 56F80x SDK project to a 56F8323 PE project.
2. Follow Step 2 in **Section 4.1** to generate code for the basic PE project structure
3. Follow Step 3 in **Section 4.1** to copy the SDK application source code
4. Determine which PE Beans are required to migrate the SDK application. By studying the SDK application code shown in **Code Example 4-1**, it is possible to determine that the SCI is being used as a standard DCE device running at 28.8Kbps, 8 data bits, 1 stop bit, no parity. The application polls the SCI receiver to get the data space and address requests from the PC application. If Program memory is requested, the SDK memory interface is used to read data from Program memory to access P space from C.

Example 4-1. SDK Application Code

```

*****
*
* FILE NAME: sci.c
*
*****/

#include "bsp.h"
#include "io.h"
#include "fcntl.h"
#include "mempx.h"

#include "assert.h"

#include "sci.h"

#define X_MEMORY 'X'
#define P_MEMORY 'P'

#define MEMORY_SIZE 0x101

#define BUFFER_SIZE 10

UWord16 Buffer[MEMORY_SIZE];
    
```

```

/*****/
int main()
{
    UWord16      StartLoc;
    UWord16      OneWord;
    UWord16      *  pData;
    UWord16      MemoryType;
    UWord16      I;
    int          SciFD;
    sci_sConfig   SciConfig;

    for ( I = 0; I < MEMORY_SIZE; I++)
    {
        Buffer[I] = I;
    }

    SciConfig.SciCntl   =  SCI_CNTL_WORD_8BIT | SCI_CNTL_PARITY_NONE;
    SciConfig.SciHiBit  =  SCI_HIBIT_0;
    SciConfig.BaudRate  =  SCI_BAUD_28800;

    SciFD = open(BSP_DEVICE_NAME_SCI_0, O_RDWR, &(SciConfig)); /* open device in Blocking
mode */

    if ( SciFD == -1 )
    {
        assert(!" Open /sci0 device failed.");
    }

    while(true)
    {
        ioctl( SciFD, SCI_DATAFORMAT_EIGHTBITCHARS, NULL );

        MemoryType = 0;
        while ((MemoryType != X_MEMORY) && (MemoryType != P_MEMORY))
        {
            read(SciFD, &MemoryType, 1);
        }


        ioctl( SciFD, SCI_DATAFORMAT_RAW, NULL );
        read(SciFD, &StartLoc, sizeof(StartLoc));

        if(MemoryType == X_MEMORY)
        {
            write( SciFD, (UWord16 *)StartLoc, MEMORY_SIZE);
        }
        else if(MemoryType == P_MEMORY)
        {
            pData = (UWord16 *)StartLoc;

            for(I = 0; I < MEMORY_SIZE; I++, pData++)
            {
                OneWord = memReadP16(pData);
                write(SciFD, &OneWord, sizeof(OneWord));
            }
        }
    }
}

```


After reviewing the available serial communication Beans in the PE Help System (see [Figure 4-4](#)), it is determined that the *AsynchroSerial* Bean is the perfect match to the SDK's SCI functionality. The SDK program memory read function is available in PE's *Memory manager* Bean.



Processor Expert™

with Embedded Beans™

Embedded Beans CPU Beans

Bean AsynchroSerial

Asynchronous serial communication

This bean implements an asynchronous serial communication. If you don't use Attention mode only two devices can communicate. Several communication features can be set up, such as number of information bits, number of stop bits, parity, and handshaking. The two devices must have the same settings. For use Attention mode see property [Attention mode](#).

Following picture shows example of connection between two DCE devices (RS232F interface) with possible handshake (dashed line):

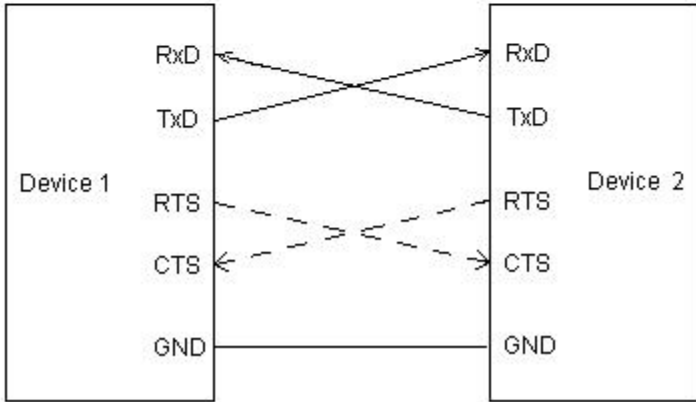


Figure 4-4. *AsynchroSerial* Bean Help

Now that it's been determined which beans are required, these Beans may be selected from the PE Bean Selector and configured appropriately. **Figure 4-5** shows the Bean Inspector window after selecting the *AsynchroSerial* Bean. As noted by the red exclamation point, PE has flagged an error notifying the user that a baud rate has not yet been specified.

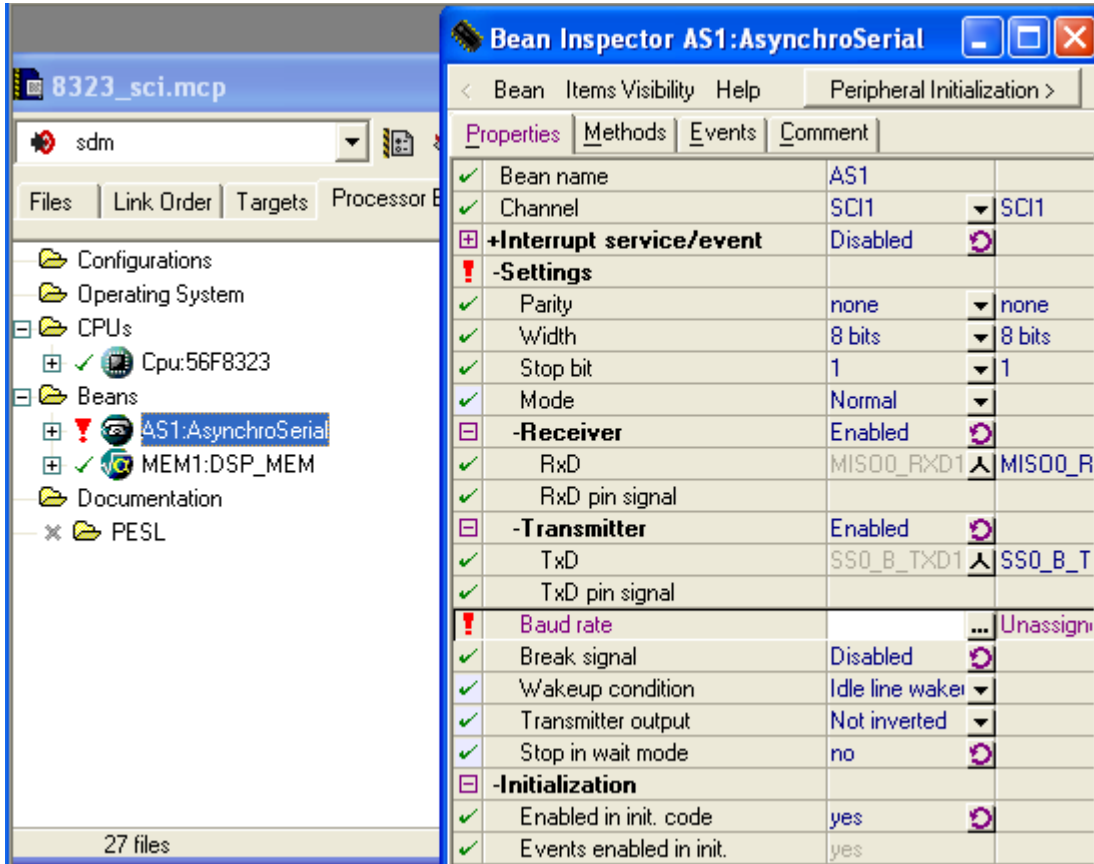


Figure 4-5. Configuring AsynchroSerial Bean

By clicking the three periods button next to the *Baud rate* property, the window shown in **Figure 4-6** will appear and allow the user to specify the requested baud rate. Note that PE will automatically calculate the exact baud rate that may be generated, based on the current rate of the chip. This value is shown in the lower left-hand corner of the configuration window, next to *Closest values*. The user may determine if this rate falls within the requirements of the system and make the appropriate adjustments. In this case, the 28800 rate is specified and PE automatically selected the closest value of 28846.154 baud.

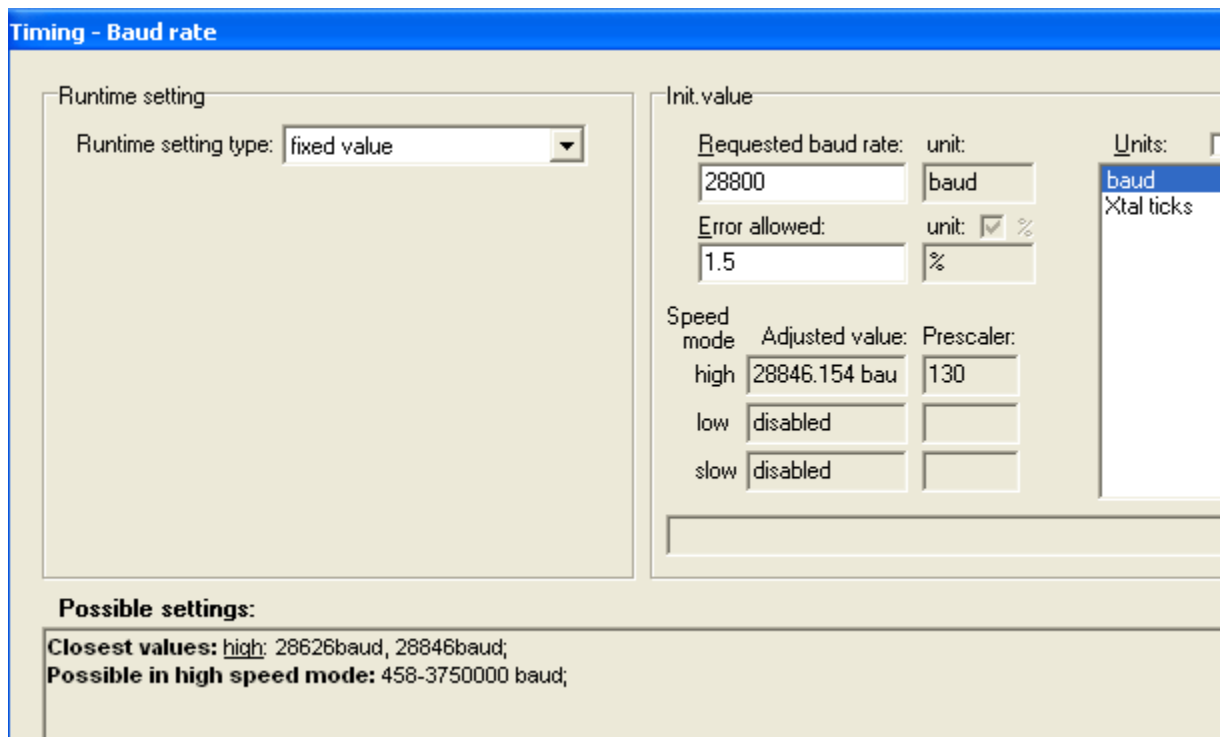


Figure 4-6. Specify Baud Rate

Now that this rate has been specified, the Red exclamation point is removed, and can now proceed to the next step.

5. Replace the SDK functions by dragging and dropping the appropriate Bean methods. **Code Example 4-2** shows the converted application code. The basic SDK application structure is still intact.

Example 4-2. Application Converted from SDK to PE

```

/* MODULE _8323_sci */

/* Including used modules for compilling procedure */
#include "Cpu.h"
#include "AS1.h"
#include "MEM1.h"
/* Include shared modules, which are used for whole project */
    
```

```

#include "PE_Types.h"
#include "PE_Error.h"
#include "PE_Const.h"
#include "IO_Map.h"

#define X_MEMORY 'X'
#define P_MEMORY 'P'

#define MEMORY_SIZE 0x101

void main(void)
{
    UWord16      StartLoc;
    UWord16      OneWord;
    AS1_TComData * pData;
    AS1_TComData MemoryType, Data[2];
    UWord16      I;
    byte error;

    /*** Processor Expert internal initialization. DON'T REMOVE THIS CODE!!! ***/
    PE_low_level_init();
    /*** End of Processor Expert internal initialization.          ***/

    /* Write your code here */

    for(;;)
    {
        MemoryType = 0;
        while ((MemoryType != X_MEMORY) && (MemoryType != P_MEMORY))
        {
            AS1_RecvChar(&MemoryType);
        }

        while(AS1_GetCharsInRxBuf() == 0);
        AS1_RecvChar(&Data[0]);
        while(AS1_GetCharsInRxBuf() == 0);
        AS1_RecvChar(&Data[1]);
        StartLoc = Data[0]<<8 | Data[1]&0x00FF;
        pData = (AS1_TComData *) (StartLoc<<1);

        if(MemoryType == X_MEMORY)
        {
            for(I=0;I<MEMORY_SIZE; I++, pData+=2)
            {
                while(AS1_SendChar(*(pData+1))==ERR_TXFULL);
                while(AS1_SendChar(*pData)==ERR_TXFULL);
            }
        }
        else if(MemoryType == P_MEMORY)
        {
            for(I = 0; I < MEMORY_SIZE; I++, pData+=2)
            {
                OneWord = MEM1_memReadP16((UWord16 *)pData);
                while(AS1_SendChar(((unsigned char *) &OneWord) +
1))==ERR_TXFULL);
                while(AS1_SendChar(((unsigned char *) &OneWord))==ERR_TXFULL);
            }
        }
    }
}
/* END_8323_sci */

```

6. At this point, the migration process is complete and the application may be functionally tested. The 56F8323EVM serial port may be connected to a PC running the serial application mentioned at the beginning of this section. The PE project may be built, downloaded to the EVM and run. At this point, you can specify on the PC Application, the address and memory space from which to read data from the 56F8323 and it will display information as shown in [Figure 4-7](#).

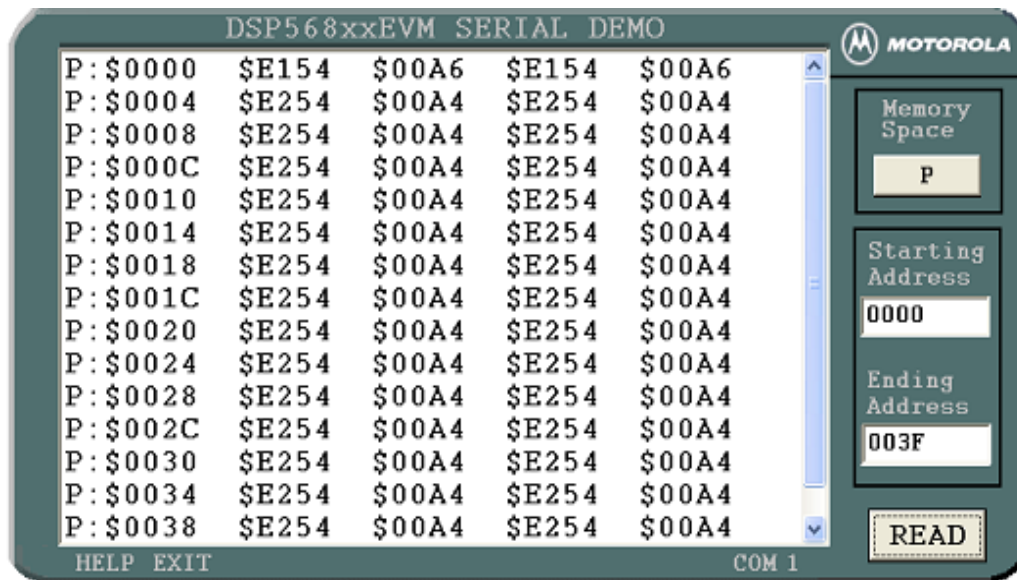


Figure 4-7. PC Serial Application

5. Summary

It is easy to understand how PE can be considered an extension or upgrade to the SDK. It shares the same SDK development concepts, as well as many of the SDK software modules. It offers the next logical development step for the SDK by providing a more user-friendly environment that even notifies the developer if he requests a configuration conflict. PE also provides a common development environment across Freescale's 8- and 16-bit product line: HC08, HCS12, 56800E.

A developer who wants to migrate an existing application from the SDK to PE will first have to become familiar with the PE development environment, its similarities to the SDK, and its enhancements. The most critical comparisons are summarized below:

- Application-Specific Libraries
 - PE includes SDK libraries without modification to APIs (for example, DSP Function, Motor Control, Tools, etc.)
- High-Level Encapsulated API
 - PE provides a GUI for configuration vs. the SDK's `appconfig.h`
 - PE and SDK APIs are not compatible
- Low-Level Register API
 - SDK's `ioctl` commands are fully supported by PESL
 - SDK peripheral macro library is available in PESL

There is no "secret decoder ring" that will allow an SDK-based application to be transformed into a PE-based application but the direct porting of all SDK libraries and the Low-Level Register command support provide a common thread for the developer to begin transitioning the application. Combined with the following PE enhancements, makes the transition even simpler.

- Migration between 8- and 16-bit MCU and DSP derivatives
- Graphical User Interface for selection and configuring software modules
- Expert knowledge system automatically prevents invalid configuration of software modules
- Drag and drop coding for object methods
- Expert on-line help system, including balloon help
- Object-oriented methodology
- Resource meter reports peripheral allocation
- Target CPU view identifies pin package definitions
- Software is tested according to ISO-certified development procedures

Finally, Metrowerks and Freescale help line support services are available 24 hours per day, 7 days a week. These services will assist in answering any SDK or CodeWarrior/PE questions. Find support information at these links:

Metrowerks: www.metrowerks.com

Freescale: www.freescale.com



How to Reach Us:

Home Page:

www.freescale.com

E-mail:

support@freescale.com

USA/Europe or Locations Not Listed:

Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
+1-800-521-6274 or +1-480-768-2130
support@freescale.com

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)
support@freescale.com

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 Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T., Hong Kong
+800 2666 8080
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

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.



Freescale™ and the Freescale logo are trademarks of Freescale Semiconductor, Inc. All other product or service names are the property of their respective owners. This product incorporates SuperFlash® technology licensed from SST.

© Freescale Semiconductor, Inc. 2005. All rights reserved.