Document ID: ETPUPORTCODE
Rev. 0.1, 08/2010

Porting eTPU Code to the Freescale eTPU Build Tools

Contents

1 Overview

This guide takes you through the process of porting your application to the new Freescale eTPU compiler (FC).

The FC is as backward compatible with the old tools as possible. However, in cases where the behavior of the old tools is ambiguous or does not comply with a standard in a way that harms quality, the FC implements a different behavior that corrects the situation. For this reason, and because of lack of support by old tools and incomplete or ambiguous features, programmers may have written non-portable code. This document covers the topics that must be considered while porting code to the FC.

1.1 Converting Inline Assembly Language to the New Syntax

The new assembly language syntax is mnemonic-based, which is the de facto standard in the semiconductor industry. The first step in porting your code is converting your files to this new assembly language syntax.

© Freescale Semiconductor, Inc., 2010. All rights reserved.

The easiest way to do this is to put all your source code files into a single folder and issue the command:

```
etpu_asm_converter.exe -c <folder-name>
```

The converter converts all C language files in the folder and renames each file into `<`*`filename`*`>.converted.c`.

For more information regarding the converter tool, please refer to `etpu_asm_converter.doc`. Make sure you read the Limitations section of this document, and make any changes required.

2 Non-Standard Syntax

The Freescale eTPU compiler is built on a standard Freescale compiler infrastructure. Further, the compiler conforms to industry standards and therefore generates all mandated warning and error messages. In some cases, non-compliant code that compiled using the old tools will not compile using the FC and therefore requires minor changes. These changes are easy to make because the FC issues an error in such cases and identifies the location of the problem code.

After you fix all non-standard code, the FC should be able to compile your files.

3 Functional Incompatibilities

Once the FC can compile your application, the next step is to ensure that the program still does what is supposed to. This section covers the main topics that should be considered when porting your code. There are two main reasons for incompatibilities: 1) the old tools generated code that does not comply with the standards and 2) code was written in an unsafe way and is therefore sometimes ambiguous or incorrect. This document suggests the best way to fix such ambiguity and to write safer code that is not implementation dependent.

3.1 C99 and TR18037 Compliance

3.1.1 Types

- `int` and `fract` interaction: applications that use `fract` arithmetic should be reviewed carefully. `TR18037` specifically says that a conversion from `fract` to `int` implies rounding towards 0. While porting several functions, we noticed that some programmers use `int` and `fract` variables together, as if no conversion occurs between them — this is the ByteCraft tool's behavior. If you want no conversion to occur, use the `_int_from_fract` intrinsic function. Notice that the type of a variable determines the code generated for it. For example, if you multiply two `fract` variables, the compiler generates a `fract` multiplication instruction; however, if the variables are of type `int`, the compiler generates an `int` multiplication instruction. As result, you should review such code carefully.
- Specifying a register should be considered as a type *modifier* and not as a type by itself. If the type is not specified, then the variable is made the default type — `int24`. For example:

```
register_diob x;        // x is of type int24 and is stored in diob
register_diob fract8 x; // x is of type fract8 and is stored in diob
```

© Freescale Semiconductor, Inc., 2010. All rights reserved.

- Division of signed variables is correctly handled by the Freescale compiler, while the ByteCraft compiler generated an unsigned division. If you want an unsigned division, then change the types of the variables involved or do a type caste before the division.

NOTE

- Signed division is much more expensive (in terms of code size *and* cycles); as a result, unless you really need signed variables, stick to unsigned variables.
- Signed bitfields are handled correctly by Freescale compiler; the ByteCraft compiler treated them as unsigned.

3.2 Inline Assembly Language Topics

3.2.1 %hex <opcode>

While porting several applications, we noticed that in some cases, programmers inserted `hex` directives in their code. Although the converter successfully converts this directive to a `.word` directive and the code compiles, this is a non-portable feature that relies on specific code generation. For example, if the opcode in question were a load from address 9 in memory in which the user expected that a specific variable was allocated, it might be that it will not be allocated to this location by the FC. If the opcode in question were a function call, then it encodes the address that was given to the function by the old tools and will not work with the new ones. Therefore, in most cases, programmers should avoid writing such code and should use specific inline assembly instructions instead.

3.2.2 C-Inline Assembly Language Interaction

The programmer must not make any assumptions regarding issues such as register allocation, variable storage location, or code location.

All interaction between C source code and inline assembly language should be done symbolically.

The following code example is a sure way to get into trouble:

```
tmp = accel_tbl[tmp];
#asm( alu a = d;  ram p <- start_period. )
```

Here, the programmer assumes (based on the old tool's behavior) that `tmp` will reside in register `d`.

This code can be easily fixed:

```
asm{move a,tmp@Rn; ldm p,start_period};
```

For details regarding inline assembly language usage, refer to the *Inline Assembly* chapter of *eTPU_Build_Tools_Reference.pdf*.

3.2.3 Inline Assembly Language Functions

Many users have written functions that are fully implemented using inline assembly.

A few changes might be needed for such functions.

© Freescale Semiconductor, Inc., 2010. All rights reserved.

Let's take as an example this code sample (which is an extract from Freescale set4) and see how to change it to standard, safe code:

```
/*
fract24 mc_ctrl_pid( fract24 error,
                     mc_ctrl_pid_t *p_pid)
*/
#asm
MC_CTRL_PID:
  /* Inputs:                                 */
  /* register a ...... error */
  /* register diob ... p_pid */
  /* Limit error to range <MIN24, MAX24>     */
  if V == 0 then goto MC_CTRL_PID_I, flush.
  if N == 0 then goto MC_CTRL_PID_I, no_flush.
…
#endasm
```

1. All inline assembly language code should reside inside a function, so we should first uncomment the function's prototype. We should also make sure that the function's name is the name used to call this function. Notice that in many cases, programmers call a label at the beginning of the inline assembly language; however, since label names are local, this will not work, so we must name our function using the label name. The label itself might now be redundant (if it is not referenced from within the function), in which case we can remove it or comment it out:

```
fract24 MC_CTRL_PID( fract24 error,
                     mc_ctrl_pid_t *p_pid)
{
#asm
//MC_CTRL_PID:
  /* Inputs:                                 */
  /* register a ...... error */
  /* register diob ... p_pid */

  /* Limit error to range <MIN24, MAX24>     */
  if V == 0 then goto MC_CTRL_PID_I, flush.
  if N == 0 then goto MC_CTRL_PID_I, no_flush.
…
#endasm
}
```

2. The function's prototype is a very important feature that was disregarded by the old tools. The prototype contains information that is used by the calling function to create correct and safe code that does not destroy local variables used in the calling function. This is why it is very important

© Freescale Semiconductor, Inc., 2010. All rights reserved.

to create a full prototype for the function. In this function, we can see that the function expects the arguments to be passed through registers `a` and `diob`. This information should be in the prototype. Now, the comments that describe this are also redundant, so we remove them:

```
fract24 MC_CTRL_PID( register_a fract24 error,
                     register_diob mc_ctrl_pid_t *p_pid)
{
#asm

  /* Limit error to range <MIN24, MAX24>              */
  if V == 0 then goto MC_CTRL_PID_I, flush.
  if N == 0 then goto MC_CTRL_PID_I, no_flush.
…
#endasm
}
```

3. Since this is a pure assembly language function, and we do not want the compiler to generate a prologue and epilogue for the function and do not perform any optimizations inside it, we should add `pure_assembly` as an attribute to the function. If the function does not contain an `rtn` instruction at its end, and you want the compiler to generate this instruction, you can use attribute `no_save_registers` instead of `pure_assembly`.
4. Since the function has a hidden argument — the flags `V` and `N` in the first two lines of the function — you should include this information in the function's prototype, so the optimizer knows that flags are important to this function and does not optimize away flags generation in the calling function:

```
__attribute__((expects_flags)) __attribute__((pure_assembly)) fract24 MC_CTRL_PID(
register_a fract24 error, register_diob mc_ctrl_pid_t *p_pid)
{
#asm

  /* Limit error to range <MIN24, MAX24>              */
  if V == 0 then goto MC_CTRL_PID_I, flush.
  if N == 0 then goto MC_CTRL_PID_I, no_flush.
…
#endasm
}
```

5. If your function contains inline assembly `rtn` instructions, you must add the `-inline off` argument to the compiler's command line, so this function is not inlined into a calling function. Such a function cannot be inlined since it contains explicit use of the `ret` instruction which, if inlined, causes undesired results.

© Freescale Semiconductor, Inc., 2010. All rights reserved.

NOTE

The correct and standard way to write a function that is fully implemented in assembly language is to put the function in an assembly language file (`.asm`), so the function is assembled by the assembler. This is the recommended way of writing assembly language functions.

© Freescale Semiconductor, Inc., 2010. All rights reserved.

© Freescale Semiconductor, Inc., 2010. All rights reserved.

How to Reach Us:

Home Page:
www.freescale.com

email:
support@freescale.com

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
(800) 521-6274
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
+81 2666 8080
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
(800) 441-2447
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 which 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.

© Freescale Semiconductor, Inc., 2010.

ETPUPORTCODE
Rev. 0.1
08/2010