AN3348
Application note

Porting eTPU code to eTPU compiler build tools guides

Introduction

This guide would take you through the process of porting your application to the new eTPU Compiler (EC). EC was designed to be as much backward compatible to the old tools as possible. However, in cases where the old tools' behavior was ambiguous or non standard compliant in a way that damaged the quality of the tools, EC implemented a different behavior which corrected the situation. For this reason, and because of lack of support by old tools and incomplete or ambiguous features, programmers might have written a non portable code. This document details the topics which need to be checked during the porting process.

Contents

1 Converting inline assembly to new syntax

The new assembly syntax is mnemonic based which is the standard the facto in the semiconductor industry. The first stage of porting your code is converting your files to the new assembly syntax.

The easiest way to do that is to put all your source files into a single folder and run:

etpu_asm_converter.exe -c <folder-name>

The converter would convert all c files in the folder and rename each file into <filename>.converted.c.

For more information regarding the converter tool, please look at *eTPU assembly converter* application note.

Make sure you also read it's *Limitation* chapter and make appropriate changes if needed.

2 Non standard syntax

The eTPU compiler is built on a standard eTPU compiler infrastructure which is used also for other architectures. It is conforming to the industry standards and therefore is generating all necessary warnings and error messages. In some cases, non compliant code which compiled using the old tools won't compile using EC and would need minor changes which are easy to do because the compiler would simply issue an error and indicate its specific location.

After making these changes, your files should be able to compile using EC.

3 Functional incompatibilities

After your application builds with EC, it is time to make sure it also does what it meant to do. This section describes the main topics which should be considered when porting the code. There are two main reasons for these incompatibilities: previous tools generated code which do not comply with the standards and code which is written in an unsafe way which is in times ambiguous and in times incorrect. This document would suggest the best way for fixing the ambiguity and for writing a safer code which is not implementation dependent.

3.1 C99 and TR18037 compliance

3.1.1 Types

- `int` and `fract` interaction: applications which are using `fract` arithmetic should be reviewed carefully. `TR18037` specifically says that a conversion from `fract` to `int` implies rounding towards 0. During porting of several functions we've noticed that programmers used `int` and `fract` variables together as if no conversion occurs between them (this was the Bytecraft tools behavior). If you want no conversion to occur use the `_int_from_fract` intrinsic functions. Notice that the type of a variable determines the code generated for it, for example if we multiply two `fract` variables the compiler would used a `fract` multiplication instruction and if the variables are of type `int` an `int` multiplication would be generated so a lot of care should be used when reviewing such code.
- 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 would be of 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
```

- Division of signed variables is correctly handled by the eTPU compiler whereas with Bytecraft an unsigned division was generated. If the user meant an unsigned division then he should change the types of variables or use casting before the division.

Note: Signed division is much more expensive (both code size and cycles) so if you don't really need it, make sure you use unsigned variables.

Signed bitfields are handled correctly by eTPU compiler whereas with Bytecraft it is treated as unsigned.

3.2 Inline assembly topics

3.2.1 %hex <opcode>

During porting several applications we've noticed that in some cases users have inserted `hex` directives in their code. Although the converter has successfully converted this to a `.word` directive and the code compiles, this is a non portable feature which relies on specific code generation. For example, if this opcode was a load from address 9 in memory in which the user expected that a specific variable was allocated, it might be that it won't be allocated in this location with EC. If this was a call to a function, then it encodes the address

that was given to the function by the old tools and it won't work with the new ones. Therefore, in normal cases users should avoid writing such a code and should change it to specific inline assembly instructions.

3.2.2 C-inline assembly interaction

The user mustn't make any kind of assumptions regarding issues as register allocation, variable storage location and code location.

All the interaction between the C and inline assembly should be done symbolically.

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

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

Here the user assumed (based on old tools behavior) that `tmp` would reside in register `d`.

This code can be easily fixed to

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

For more details regarding inline assembly usage please look at the *Inline Assembly* chapter in *eTPU_Build_Tools_Reference.pdf*.

3.2.3 Inline assembly functions

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

A few changes might be needed for these functions.

Let's take as an example this code sample and see how we change it into a standard and 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 code should reside inside a function, so first we should uncomment the function's prototype. We should also make sure that the function's name is the name which is used in order to call this function. Notice that in many cases users used to call a label at the beginning of the inline assembly but since label names are local this won't work so now we'll have to call our function with the same name of this label. The label itself might be redundant now if it is not referenced from within the function so we can remove or comment it:

```
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 quite disregarded by the old tools. The prototype contains information regarding the function which is used by the caller function in order to create a correct and safe code which would not destroy local variables which are used in the caller function. That is why it is very important 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 which describe that are also redundant and we can 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 function and we don't want the compiler to generate any prologue and epilogue to it and do not perform any optimizations inside it, we should add it as an attribute to the function. If the function does not contain an `rtn` instruction at it's end and you want the compiler to generate it you can use attribute `no_save_registers` instead of `pure_assembly`.
4. Since the function is having also a hidden argument which is the flags (`V` and `N` at the first two lines of the function) we should add also this information into the function's prototype so that the optimizer would know that flags are important to this function so flags generation in the caller function should not be optimized away:

```
__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 to the compiler's command line the `-inline off` so that this function would not get inlined into a caller function. It cannot be inlined since it contains explicit use of `ret` which would get inlined and cause an undesired effect.

Note: The correct and standard way to write a function which is fully implemented in assembly is to put this function in an assembly file (`.asm`) which is only assembled by the assembler. This is the recommended way of writing assembly functions.

4 Revision history

Table 1. Document revision history

Date	Revision	Changes
02-Mar-2011	1	Initial release.
18-Sep-2013	2	Updated Disclaimer.