AN11074

Using LPC122x ROM division routines

Rev. 1 — 1 May 2011 **Application note**

Document information

Info	Content
Keywords	LPC1227, LPC1226, LPC1225, LPC1224, LPC122x Cortex-M0 Division ROM
Abstract	LPC122x devices are equipped with constant-runtime integer division routines stored in ROM. These routines operate independently of tool chain and because they are stored in ROM, using them requires very little flash memory. Because the runtime of these routines is not affected by the numerator and divisor being used, they are well suited for high reliability and real time applications.

Revision history

Rev	Date	Description
1	20110501	Initial version.

Contact information

For more information, please visit: http://www.nxp.com

For sales office addresses, please send an email to: salesaddresses@nxp.com

1. Introduction

In order to fully appreciate the usefulness of the LPC122x ROM based constant runtime integer division routines, a thorough background of how division is commonly implemented on Cortex-M0 devices is required. The topics covered include the differences between M0 and other devices (such as M3), the ARM EABI, and design trade-offs for division algorithms. Once these topics are covered, developers will have all requisite knowledge required to implement direct calls to the division routines stored in ROM, and subsequently implement wrapper routines which 'overload' the EABI functions.

2. Cortex-M0 vs Cortex-M3/M4

There are a host of differences between the Cortex-M0 architecture and the Cortex-M3/M4 architecture, but a commonly overlooked difference is that the Cortex-M0 does not feature hardware dividers of any sort. Thus, in order to perform division, software routines must be used. The implementation of these routines is commonly implemented through the inclusion of precompiled libraries contained in a developer's tool-chain. The same C language program when compiled on various tool chains can vary in regard to these division routines, or they can even vary version to version of the same tool chain.

3. ARM Enhanced Application Binary Interface (EABI)

To enable interoperability between tool chains ARM has defined a standard calling convention for integer division on Cortex-M0. This convention is detailed in the ARM EABI documentation, available from ARM directly. The signatures for the four functions defined in the conventions are detailed in Fig 1. Please note that the directive *__value_in_regs* is implemented by several ARM tool-chains, but is not part of ANSI C and may not be available in all development environments. In the event that it is not supported, care must be taken to ensure that return results are contained in registers rather than passed back via the stack.

```
typedef struct { int quot, int rem; } idiv_return;
typedef struct { unsigned quot, unsigned rem; } uidiv_return;

int __aeabi_idiv(int numerator, int denominator);
unsigned __aeabi_uidiv(unsigned numerator, unsigned denominator);
__value_in_regs idiv_return __aeabi_idivmod(  int numerator,
                                             int denominator);
__value_in_regs uidiv_return __aeabi_uidivmod(  unsigned numerator,
                                               unsigned denominator);
```

Fig 1. Prototypes for ARM EABI 32-bit integer division operations

As an example, take the following C function:

```
unsigned int simpleDiv (unsigned int a, unsigned int b)
{
    unsigned int x;
    x = a / b;
    return x;
}
```

Fig 2. C language routine using '/' division operator

Inspecting the disassembly of this clearly shows how the EABI convention is used:

```
0x00000146:    b570        p.      PUSH     {r4-r6,lr}
0x00000148:    4604        .F      MOV      r4,r0
0x0000014a:    460d        .F      MOV      r5,r1
0x0000014c:    4629        )F      MOV      r1,r5
0x0000014e:    4620         F      MOV      r0,r4
0x00000150:    f000f8c0    ....    BL       __aeabi_uidivmod ;
0x2d4
0x00000154:    4606        .F      MOV      r6,r0
0x00000156:    4630        0F      MOV      r0,r6
0x00000158:    bd70        p.      POP      {r4-r6,pc}
```

Fig 3. ARM Thumb2 assembly of C routine

4. Runtime of division routines

In real time and high reliability applications the consistency of an algorithm can be critically important. In other words, it may be disadvantageous to use a routine which is performance optimized if this routine behaves differently based on the value of the arguments passed to it.

Note that the performance measurements contained in this section make use of CPU cycles rather than real time units. Doing so removes the dependency of operating frequency, making it easier to conceptually compare results.

When comparing the surfaces seen in Fig 4 and Fig 5, notices how Vendor A's performance is on average faster, but in cases where the numerator is greater than the divisor, the runtime is drastically faster. The magnitude of this inconsistency may vary from tool-chain to tool-chain, and may even vary between release versions of a given tool-chain.

Not only does the runtime of the routines stored in the LPC122x family out perform both of the plots below, it also has a much lower variability[1]. Because the routines are stored in the physical device itself they are not affected by tool chain used, making the LPC122x parts very flexible while maintaining their consistent runtime.

While it is outside the scope of this application note, it should be mentioned that the particular implementation used by a given tool chain may also affect a program's image size, as typically a trade-off is made between performance (typically larger) and code density (typically slower). As it will be shown later in this application note, by using wrapper functions to implement EABI compliant routines using the ROM library, code size can be reduced while typically improving performance and simultaneously maintaining a high degree of runtime consistency.

1. Testing resulted in an average of 122 cycles per division operation with a coefficient of variation (CV) for runtime of 0.33 % across the set of data tested.

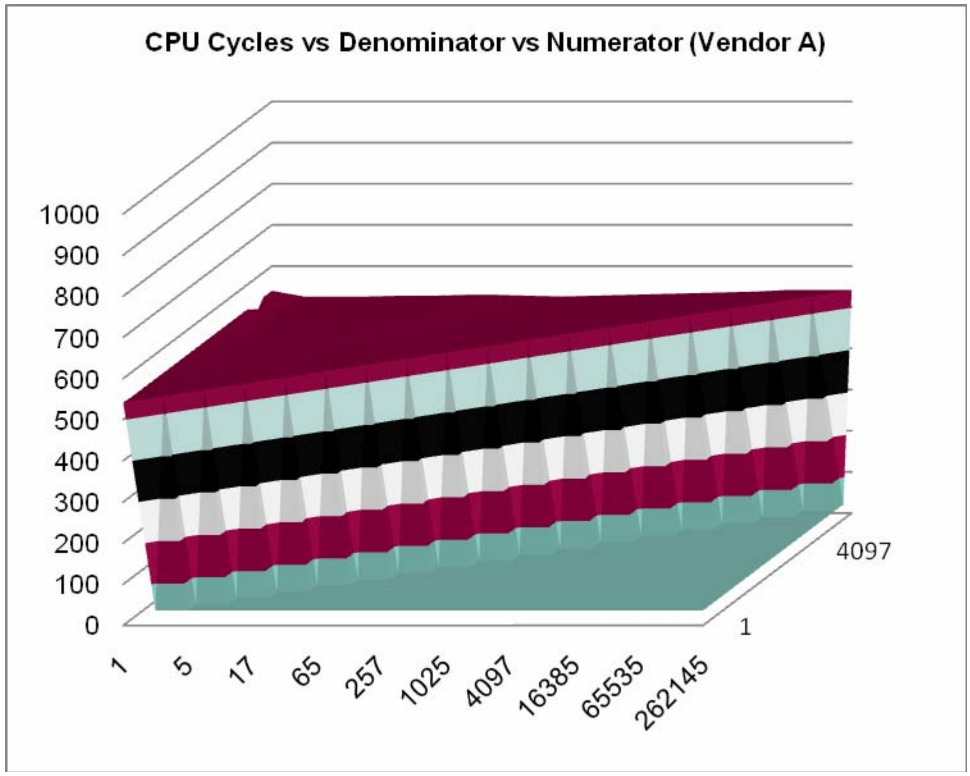

Fig 4. A particular tool-chain's division runtime

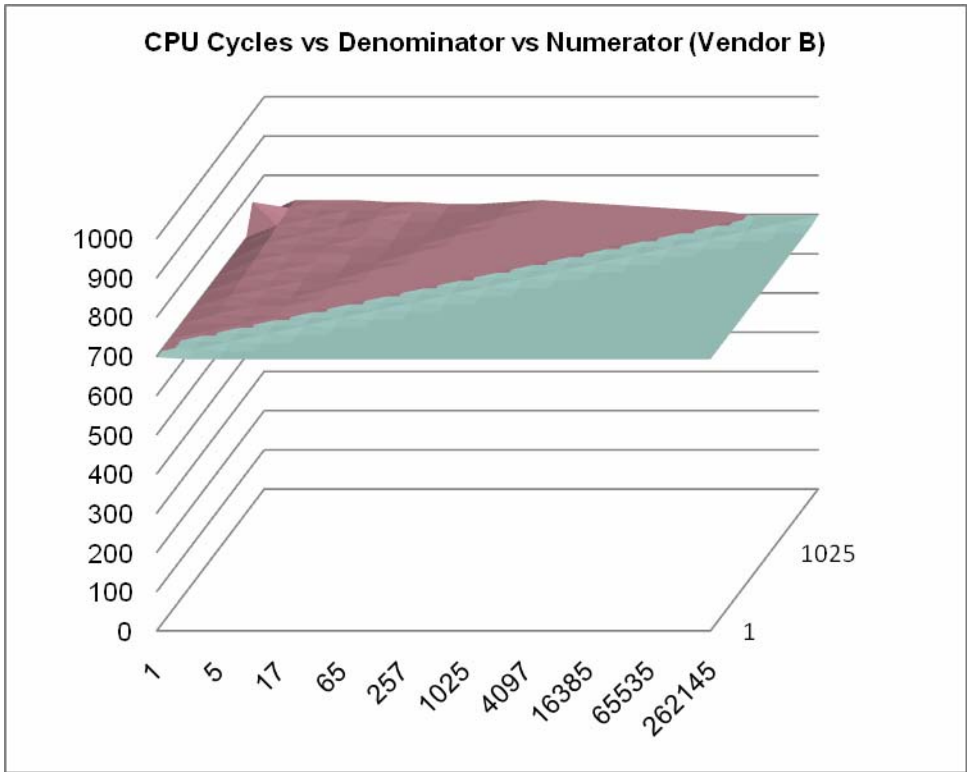

Fig 5. An alternate tool-chain's division runtime

5. Invoking ROM division routines directly

Invoking the division library in software is straight forward. There is a defined structure which implements a table of function pointers to the various division operations. This is seen in Fig 6.

```
typedef struct { int quot; int rem; } idiv_return;
typedef struct { unsigned quot; unsigned rem; } uidiv_return;

typedef struct{
    /* Signed integer division */
    int         (*sidiv)    (int numerator, int denominator);
    /* Unsigned integer division */
    unsigned    (*uidiv)    (unsigned numerator, unsigned
denominator);
    /* Signed integer division with remainder */
    idiv_return (*sidivmod) (int numerator, int denominator);
    /* Unsigned integer division with remainder */
    uidiv_return(*uidivmod) (unsigned numerator, unsigned
denominator);
} LPC_ROM_DIV_STRUCT;
```

Fig 6. Signatures of the division API functions

There is an API table at a fixed address in ROM. Fig 9 illustrates the organization of API tables in ROM. The first element of this table points to the division API table. In the example code shown in Fig 7 this is stored in the pointer *pDivAPI*. Once the table has been located, all that remains to be done is to call the desired member function, in this case *uidiv*.

```
int main(void)
{
  unsigned int result;
  //Entry to ROM API Table - fixed location
  const void**const pROMTable = (const void**) 0x1FFC0000;

  //Entry to Division API - location may vary across ROM versions
  LPC_ROM_DIV_STRUCT*const pDivAPI = (LPC_ROM_DIV_STRUCT*) pROMTable[0];

  result = pDivAPI->uidiv(500,321);

  return 0;
}
```

Fig 7. Calling *uidiv*

6. "Overloading" EABI division

There are several reasons why it is desirable to overload the '/' and '%' operators with the ROM based division routines on LPC122x. Most obviously, directly invoking the ROM calls can be cumbersome, and results in code that isn't as readable as standard C.

A secondary side effect is that many standard C libraries use division and will invoke EABI division, and at link time the libraries included with tool-chain will be imported into the program image. This would result in fragmented division performance, as well as duplicate functionality and increased code size. These issues may not affect all applications, but it is likely that a majority of developers would prefer to use overloaded routines to alleviate the issues stated above. Fortunately, accomplishing this is relatively easy; in the case of the LPC122x many major tool chains already support this out of the box.

Modern tool chains have smart linkers which will use local implementations of library routines without any additional configuration required. This is the case in KEIL MDK and IAR Embedded Workbench, and may apply to other embedded development platforms.

GCC does not automatically behave this way, but use of the linker flag *--allow-multiple-definition* can be used to enable this feature. In the case of LPCXpresso (an Eclipse distribution using GCC) LPC122x projects should automatically include support for overloading division with the ROM routines.

As an example, by defining a function with an EABI compliant signature, the ROM call can be wrapped in a function which will overload the desired functionality. This can be seen in Fig 8.

```
int __aeabi_idiv(int numerator, int denominator)
{
  //Entry to ROM API Table - fixed location
  const void**const pROMTable = (const void**) LPC_122x_DIVROM_LOC;

  //Entry to Division API - location may vary across ROM versions
  LPC_ROM_DIV_STRUCT*const pDivAPI = (LPC_ROM_DIV_STRUCT*) pROMTable[0];

  return pDivAPI->sidiv(num,div);
}
```

Fig 8. An EABI compliant signed integer division wrapper function

7. Performance enhancement: caching ROM entries at startup

While all LPC122x devices feature division libraries stored in ROM, future revisions of the LPC122x ROM may not store these routines at the same location. Because of this, the use of the ROM library must look up function locations at runtime to ensure operation on future device versions.

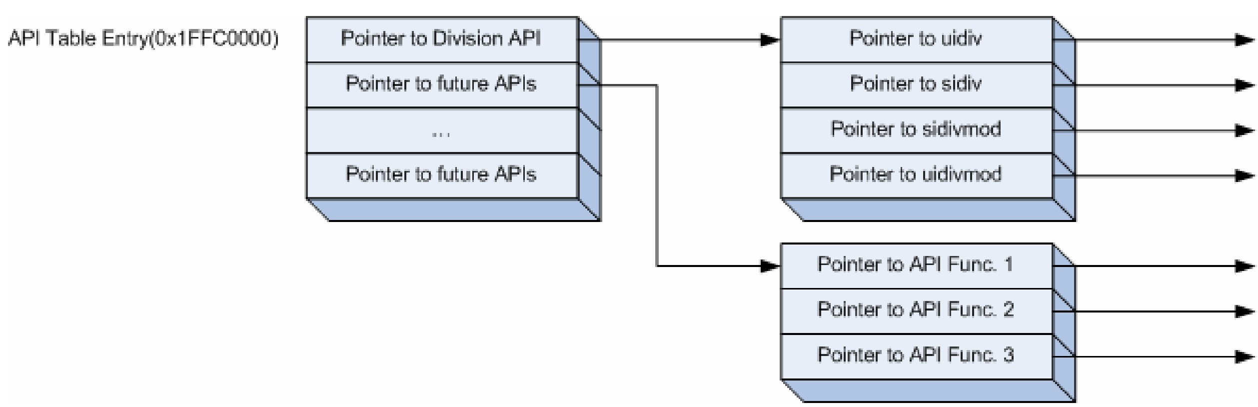

Fig 9. Conceptual diagram illustrating how division routines are stored in ROM

Notice how the code contained in Fig 8 will dereference the symbol *LPC_122x_DIVROM_LOC* twice, each time the division routines are invoked. By storing these locations in a table (or cache) in RAM during system startup, performance can be further improved. It is necessary that this caching occur very early in execution in case CMSIS compliant routines such as *SystemInit*(void) perform any division operations. While it is outside the scope of this application note, example code is provided in the included source archive which illustrates this caching strategy.

8. Legal information

8.1 Definitions

Draft — The document is a draft version only. The content is still under internal review and subject to formal approval, which may result in modifications or additions. NXP Semiconductors does not give any representations or warranties as to the accuracy or completeness of information included herein and shall have no liability for the consequences of use of such information.

8.2 Disclaimers

Limited warranty and liability — Information in this document is believed to be accurate and reliable. However, NXP Semiconductors does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and shall have no liability for the consequences of use of such information.

In no event shall NXP Semiconductors be liable for any indirect, incidental, punitive, special or consequential damages (including - without limitation - lost profits, lost savings, business interruption, costs related to the removal or replacement of any products or rework charges) whether or not such damages are based on tort (including negligence), warranty, breach of contract or any other legal theory.

Notwithstanding any damages that customer might incur for any reason whatsoever, NXP Semiconductors' aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms and conditions of commercial sale of NXP Semiconductors.

Right to make changes — NXP Semiconductors reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

Suitability for use — NXP Semiconductors products are not designed, authorized or warranted to be suitable for use in life support, life-critical or safety-critical systems or equipment, nor in applications where failure or malfunction of an NXP Semiconductors product can reasonably be expected to result in personal injury, death or severe property or environmental damage. NXP Semiconductors accepts no liability for inclusion and/or use of NXP Semiconductors products in such equipment or applications and therefore such inclusion and/or use is at the customer's own risk.

Applications — Applications that are described herein for any of these products are for illustrative purposes only. NXP Semiconductors makes no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

Customers are responsible for the design and operation of their applications and products using NXP Semiconductors products, and NXP Semiconductors accepts no liability for any assistance with applications or customer product design. It is customer's sole responsibility to determine whether the NXP Semiconductors product is suitable and fit for the customer's applications and products planned, as well as for the planned application and use of customer's third party customer(s). Customers should provide appropriate design and operating safeguards to minimize the risks associated with their applications and products.

NXP Semiconductors does not accept any liability related to any default, damage, costs or problem which is based on any weakness or default in the customer's applications or products, or the application or use by customer's third party customer(s). Customer is responsible for doing all necessary testing for the customer's applications and products using NXP Semiconductors products in order to avoid a default of the applications and the products or of the application or use by customer's third party customer(s). NXP does not accept any liability in this respect.

Export control — This document as well as the item(s) described herein may be subject to export control regulations. Export might require a prior authorization from national authorities.

Evaluation products — This product is provided on an "as is" and "with all faults" basis for evaluation purposes only. NXP Semiconductors, its affiliates and their suppliers expressly disclaim all warranties, whether express, implied or statutory, including but not limited to the implied warranties of non-infringement, merchantability and fitness for a particular purpose. The entire risk as to the quality, or arising out of the use or performance, of this product remains with customer.

In no event shall NXP Semiconductors, its affiliates or their suppliers be liable to customer for any special, indirect, consequential, punitive or incidental damages (including without limitation damages for loss of business, business interruption, loss of use, loss of data or information, and the like) arising out the use of or inability to use the product, whether or not based on tort (including negligence), strict liability, breach of contract, breach of warranty or any other theory, even if advised of the possibility of such damages.

Notwithstanding any damages that customer might incur for any reason whatsoever (including without limitation, all damages referenced above and all direct or general damages), the entire liability of NXP Semiconductors, its affiliates and their suppliers and customer's exclusive remedy for all of the foregoing shall be limited to actual damages incurred by customer based on reasonable reliance up to the greater of the amount actually paid by customer for the product or five dollars (US$5.00). The foregoing limitations, exclusions and disclaimers shall apply to the maximum extent permitted by applicable law, even if any remedy fails of its essential purpose.

8.3 Trademarks

Notice: All referenced brands, product names, service names and trademarks are property of their respective owners.

9. Contents

1. Introduction 3
2. Cortex-M0 vs Cortex-M3/M4 3
3. ARM Enhanced Application Binary Interface (EABI) 3
4. Runtime of division routines 4
5. Invoking ROM division routines directly 6
6. “Overloading” EABI division 7
7. Performance enhancement: caching ROM entries at startup 8
8. Legal information 9
8.1 Definitions 9
8.2 Disclaimers 9
8.3 Trademarks 9
9. Contents 10

Date of release: 1 May 2011
Document identifier: AN11074