AN10963

Reducing code size for LPC11XX with LPCXpresso

Rev. 1 — 2 August 2010 Application note

Document information

Info	Content
Keywords	Cortex, M0, Optimization, Size
Abstract	This application note will cover some basic techniques which can be used to reduce the size of your program's binary image. This is an important step in the design process when targeting LPC11XX parts with small memory footprints.

Revision history

Rev	Date	Description
1	20100802	Initial version.

Contact information

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

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

1. Introduction

The LPC11XX part family offers a wide granularity of flash densities which enable designers to reduce overall system costs by targeting optimally sized devices. When targeting parts with the lowest available memory sizes, it becomes critical that a program's binary image does not exceed the part's resources. Many strategies exist to reduce an embedded program's footprint, and this application note will detail a general purpose set of recommendations which if followed properly will allow designers to target lower density microcontrollers with minimal effort.

It should be noted that a typical case of development may involve the use of an evaluation board loaded with a microcontroller featuring more memory than the intended target hardware. Often larger part variants in a family will inherit the memory map of the lower end part variants. By adjusting the project's settings to target the lowest end part early in the development process, the designer will have a better feel for the constraints they will be working with in their final implementation.

2. Code size reduction strategies

Most embedded toolchains support configurable optimization settings, and LPCXpresso is no different. However, adjusting compiler optimization levels alone may not result in the smallest code size possible, and in general enabling optimization may not be a good starting point when attempting to reduce code size. Analyzing a program allows a developer to detect when additional functions have been unintentionally imported due to the use of libraries. Enabling compiler size optimization will not remove any superfluous symbols from a program.

The recommend strategy for reducing code size is outlined below:

- Avoid the use of high level library routines
- Target a larger (or "virtual") device in the case of link failure
- Configure linker options to remove unreferenced sections
- Replace generic routines with application specific ones
- Enable compiler optimization for code size

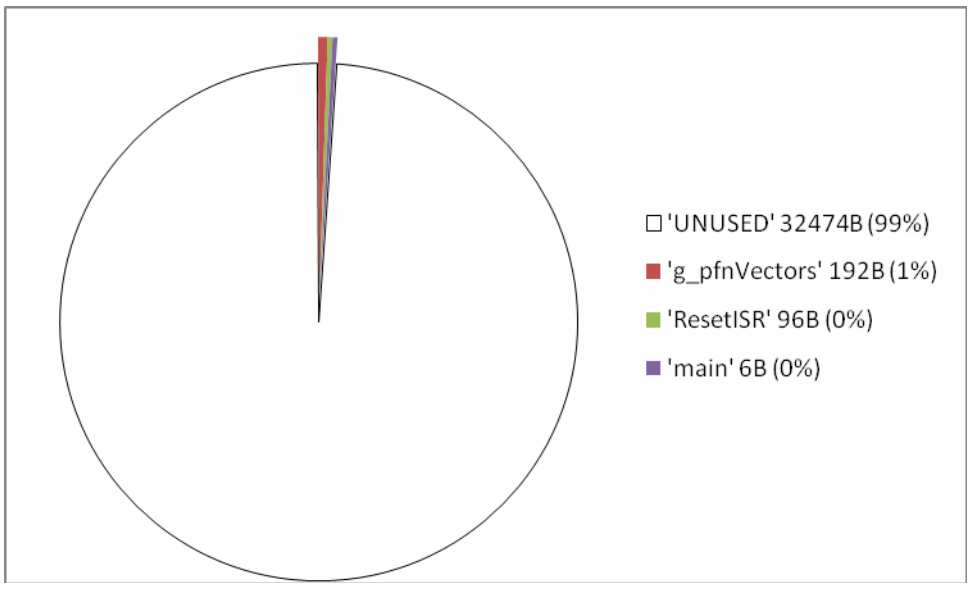
2.1 Avoid the use of high level library routines

As an example, a simple program with a perpetual loop was created. Project options were selected to enable semi-hosting. The program requires as little as 350 bytes. Due to the removal of extraneous startup routines there is little overhead in this program: the exception vector for the Cortex-M0, an application specific low level startup routine and the body of main().

Simply adding a call to printf("Hello World!") can lead to a program with over 200 functions being included. The specific functions included may be a surprise to many developers as they are not all related to I/O. Because the Cortex-M0 does not include a hardware divider, several math functions are harvested in order to enable output formatting. The inclusion of printf(…) also requires several double precision floating point routines, each of which can take up over 400 bytes. By removing unreferenced sections the number of functions is reduced to 57, but this is still surprisingly large and uses a substantial amount of the LPC1114's available flash as can be seen when Fig 1 is compared with Fig 2.

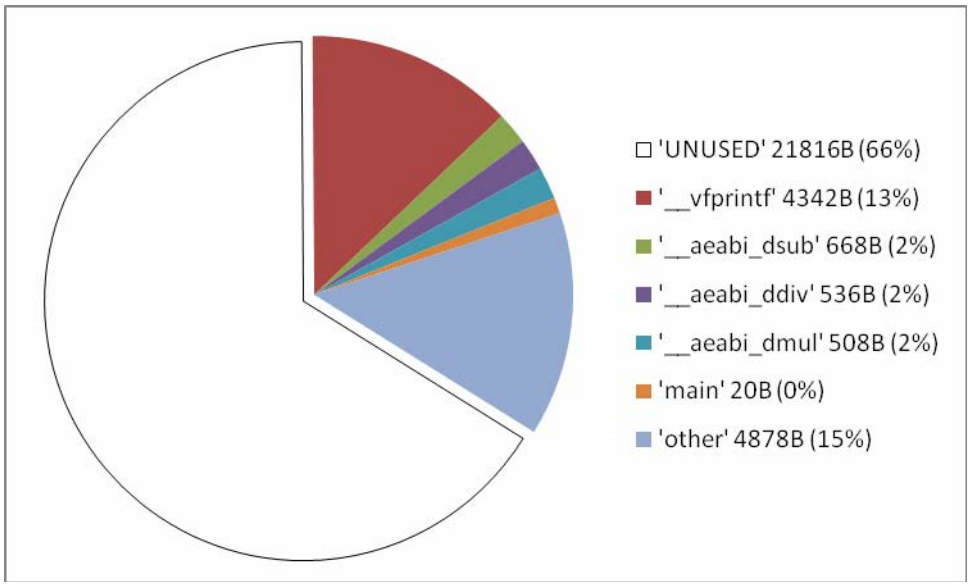
If the use of printf(…) cannot be avoided, consider defining the preprocessor symbol CR_INTEGER_PRINTF in LPCXpresso's compiler settings, to limit formatting when using the REDLIB libraries as it alleviates the need for double precision floating point routines, thus saving space.

This section has focused on printf(…) as it is a commonly used library function that will typically harvest many other functions, however it is not the only library routine that does this. A recommended practice is to always analyze a project's symbol table to ensure only those symbols which are required are included in the program image.



(1) Not drawn to scale

Fig 1. Flash usage by a simple program without printf(…) for LPC1114



(1) Not drawn to scale

Fig 2. Flash usage by a simple program with printf(…) for LPC1114

Coding style can also affect code size. For instance consider the following two routines to increment and bound a value:

Table 1. Coding style comparison

C Source	Note
count = (count+1) % 60;	Will include division routine
if (++count >= 60) count = 0;	No division required

2.2 Target a larger (or "virtual") device in the case of link failure

When a program's image exceeds the resources of the target device, the linker will stop and issue an error, preventing further analysis by the developer. Without a fully linked image, it is difficult to detect unintentionally included libraries. Analysis of a properly linked image enables developers to efficiently improve the largest sections of a project, rather than attempting to reduce the size of every single function. It is therefore recommended as a first step that developers work around linker failures in order to better reduce their code size, rather than enabling optimizations straight away.

LPCXpresso allows designers to easily adjust their target devices in the "MCU Settings" dialog. As an example, when a project targeting an LPC1111 fails, selecting the LPC1114 instead may be a sufficient means to link the project. Be aware that changing the target device may allow the build to complete, but you will not be able to debug the device in LPCXpresso as the device ID will not match.

Should the project be too large for even the LPC1114, by manually managing the linker's configuration file inside LPCXpresso, you can configure the toolchain into thinking your target device has more memory than it actually does. As the topic of generating custom linker scripts is slightly lengthy, it has been included later in the document as Appendix A. When using a custom linker script, just as when selecting a larger device debugging and execution will not be possible.

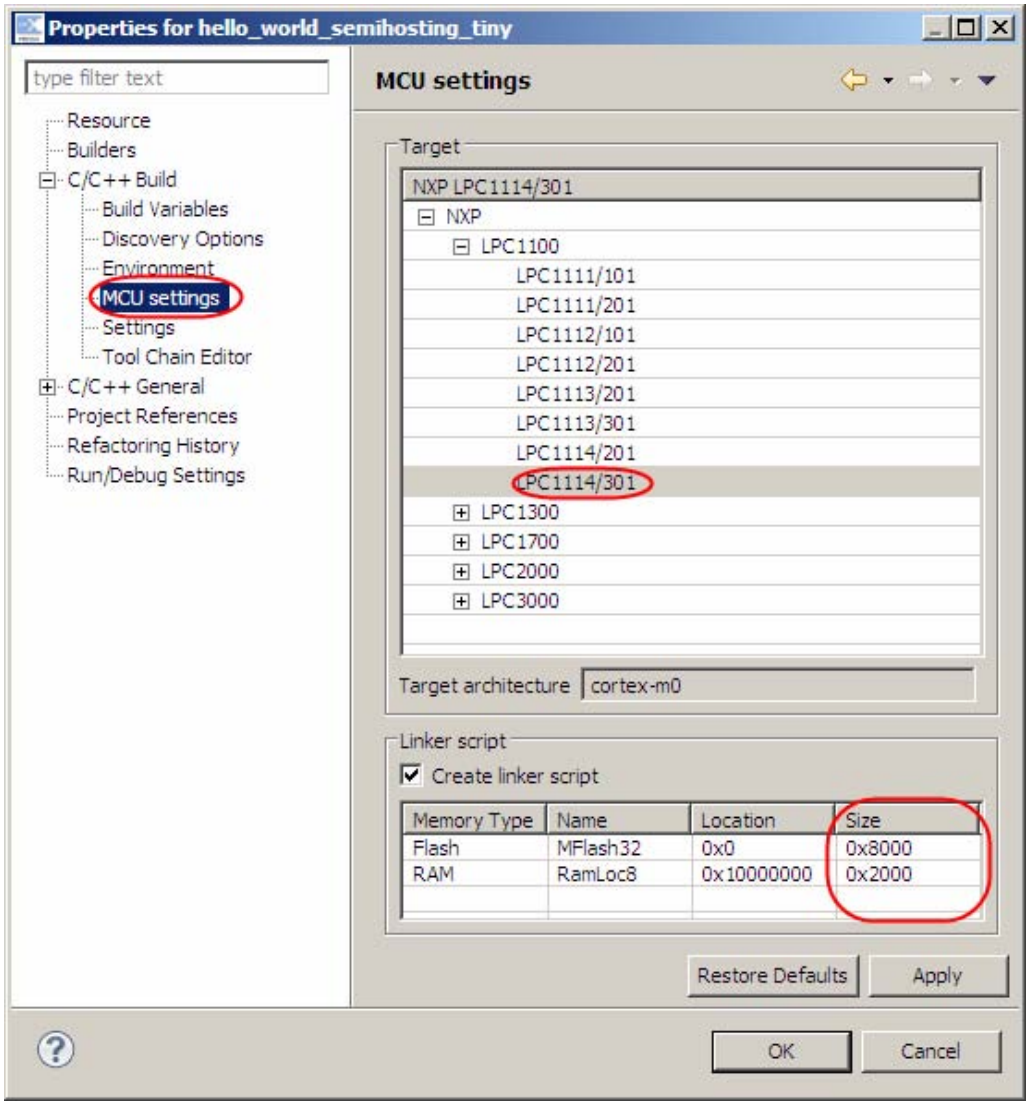


Fig 3. LPCXpresso: MCU settings

2.3 Configure linker options to remove unreferenced sections

Most toolchains make use of a linker with configurable options. Typically these will include a parameter to check for any unreferenced symbols, and in the event that any are found they will be removed from the linked program image.

Because LPCXpresso is based on the GNU toolchain users should ensure that their project has specific options enabled to ensure that unreferenced sections are properly removed at link time. The compiler options apply not only to project source code, but any libraries as well.

Table 2. LPCXpresso recommended toolchain options

Flag	Tool	Description
-ffunction-sections	Compiler	Create function specific sections in each module
-fdata-sections	Compiler	Create data specific sections in each module
--gc-sections	Linker	Remove unreferenced sections
--cref	Linker	Generate cross reference report in map file

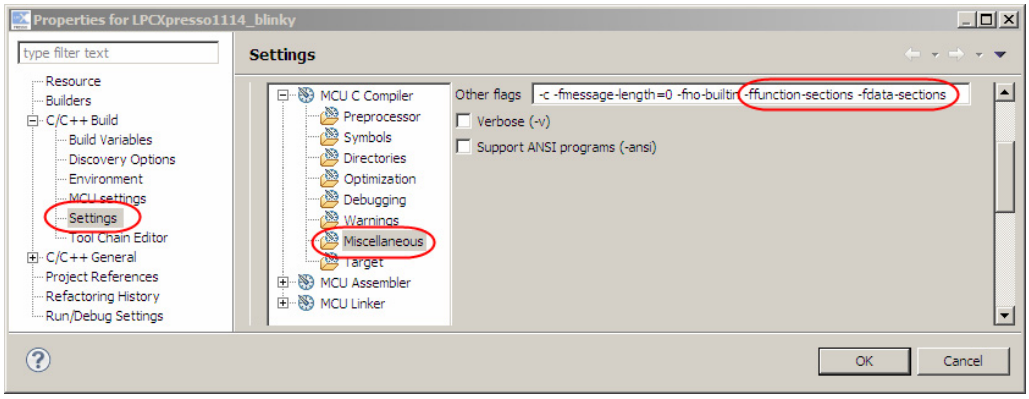


Fig 4. Compiler options

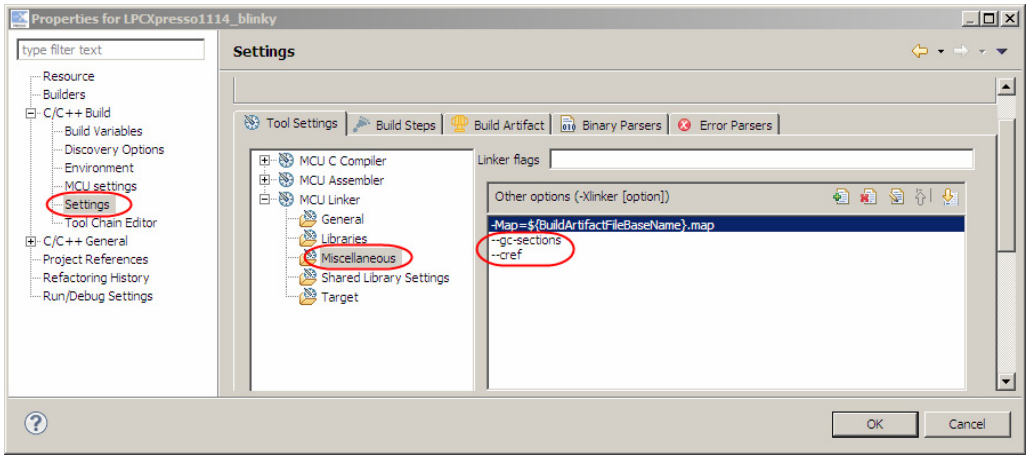


Fig 5. Linker options

2.4 Replace generic routines with application specific ones

While the use of general purpose routines can aid in rapidly developing an application, the increased flexibility typically comes at a price: increased code size. A recommendation in achieving the leanest program is to replace generic routines with application specific ones. Example LPCXpresso projects will typically include several

routines which, if deemed to be too large for the functionality they offer, can either be replaced or removed.

Take the function “SystemInit()” as an example. Without compiler optimization enabled, this function requires up to 224 bytes of “.text” (as well as 4 bytes of RAM for the SystemCoreClock variable). If your application runs at a fixed clock rate (which is common in many applications) you should consider replacing this routine with a focused initialization routine to configure your clock(s) as required.

The example project LPC1114_Blinky makes use of a routine GPIOSetDir(). By including a post build action “arm-none-eabi-nm --size-sort ${BuildArtifactFileName}” it can be seen that GPIOSetDir() occupies over 500 bytes. Some applications do not need to adjust the directionality of peripheral pins at runtime because they are dedicated on the PCB. In these scenarios substantial amounts of code can be recovered by using fixed pin input/output routines at the start of execution.

Another example involves the SYSAHBCLKCTRL register. Many of the routines included with LPCXpresso will set or clear bits in SYSAHBCLKCTRL, but in some applications (typically those without power consumption constraints) peripherals are powered up at system startup and remain enabled indefinitely. In such a situation, it may make more sense to simply configure SYSAHBCLKCTRL once at start up by setting multiple bits in the mask. Doing this will prevent additional code associated with inline read/modify/write cycles being included.

There is no universal rule as to which library functions should be replaced, therefore doing so is recommended only in the event that implementing the application specific routine can be accomplished with relative ease and in cases where prior analysis shows that space savings warrant the development of application specific routines.

2.5 Enable compiler optimization for code size

GCC supports a multitude of compiler options, which at times may overwhelm a developer. As a general rule, the “-Os” meta-option to optimize for size is a good starting point. This option will enable the majority of optimizations included in “-O2”, while disabling those which generally increase code size.

Many optimization levels can reorder program execution complicating the task of debugging. In cases where a program simply will not link without optimizations enabled, yet debugging is required, the developer may have an option to make life easier: LPCXpresso allows users to enable optimizations on a file by file basis. This makes it possible to selectively disable optimization on a specific module in order to make program flow more intuitive while operating under a debugger.

Despite the fact that optimization can make debugging more challenging, it should also be noted that optimizing code can also affect the behavior of a program. This typically manifests when memory mapped devices and/or peripherals are not declared with the volatile attribute,[1] and when variables are used to enforce timing. As such it is generally recommended to enable optimizations as soon as a module of code has stabilized, so that any issues (usually timing related) introduced by optimization are detected as early as possible.

It is also recommended that in addition to an application’s code being optimized, any libraries (for which source is available) also be compiled with optimizations enabled.

1. http://www.nxp.com/redirect/en.wikipedia.org/wiki/Volatile_variable

Once an application's development has stabilized, testing with further optimization options can occur. When code size savings can be had due to reducing library bloat and removing unused sections, the designer is free to experiment at their discretion with further optimizations levels such as "-O3" or focusing optimizations on run time performance rather than code size reduction, as determined by their application's requirements.

3. Advanced topics

This section will cover several advanced topics. They are:

- Utilities
- Structure alignment
- Disassembly
- Vector table modification

3.1 Utilities

LPCXpresso offers several tools to assess a program's code size right out of the box. A command prompt with an automatically configured environment can be quickly opened by CTRL+Clicking on the active project's name in the LPCXpresso status bar. Additionally, these utilities can be added to pre/post build actions, as seen in some of the included example code.

The first of these utilities is the "size" program. The arm-none-eabi-size tool is part of the GNU toolchain included with LPCXpresso. It can be accessed via an ELF binary's context menu (Binary Utilities->Size) or can be invoked as a part of a custom build script or manually from the command line. In Fig 6 it can be seen that the example "Blinky" project when built without any optimization takes up nearly the entire 8 kB of flash offered by an LPC1111.

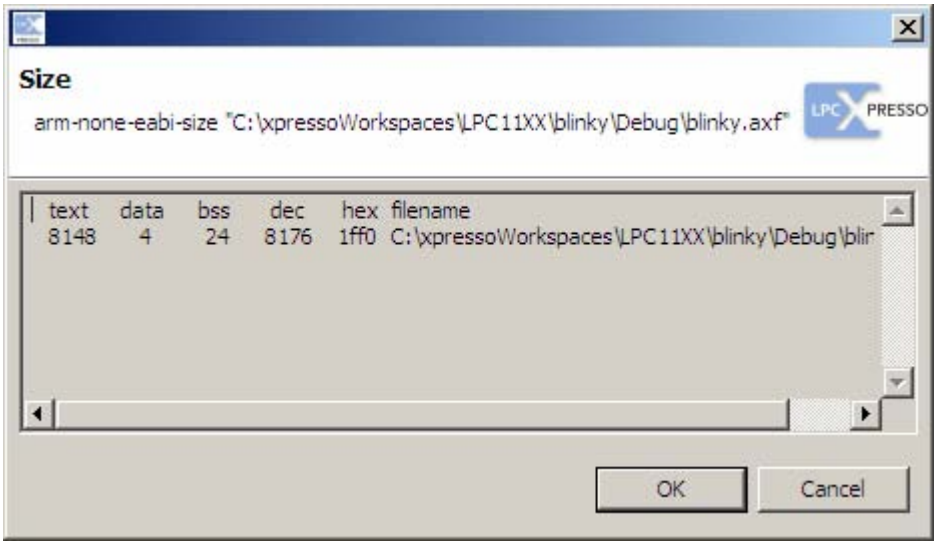


Fig 6. LPCXpresso code size example

Analyzing a list of a program's symbols can be useful in determining whether libraries have unintentionally included unnecessary functions. In addition to the list of symbols,

knowing which symbols are the largest will aid in reducing the size of code without wasting effort on optimizing insignificant routines.

This can be assessed by reading the toolchain's map report. Adding the command line switch "--cref" will append a "Cross Reference Table" to the map report which alphabetically lists all symbols and the modules in which they are referenced.

Sometimes it can be more convenient to sort the list of symbols by their size rather than alphabetically. The GNU toolchain included in LPCXpresso comes with a copy of the utility "arm-none-eabi-nm" which lists symbols from ELF object files. Invoking the command with the "--size-sort" option will result in output being sorted size. Including the following line in a list of post build actions will display the size of all symbols after each build: "arm-none-eabi-nm --size-sort ${BuildArtifactFileName};"

3.2 Structure alignment

There is one particular case in which structure member ordering can be a significant cause for program bloat: when there are large arrays of structures being used. Because the Cortex-M0 cannot transfer arbitrarily sized data from every memory location, the compiler must insert padding to ensure that all members are aligned naturally in memory. This padding is illustrated in Fig 8. Padding misaligned structures can impact code size whenever the array is not zero-initialized, regardless of whether or not it was declared with the const qualifier. It should also be noted that the same padding is required if the array is stored in RAM. In the case of the LPC11XX family of parts, there is less on chip RAM than flash, and as such the padding will be underutilizing a higher percentage of the part's resources.

Take the structures in Fig 7 as an example. Notice that they have the exact same type of members, however they are ordered differently. While the size improvement seen by properly ordering the structure (four bytes) may seem insignificant, the loss of memory from having a table of 100 poorly structured elements can be significant when using parts with lean resources. For instance, 400 padding bytes on a program targeting the LPC1111 results in nearly 5 % wasted code storage (and 10 % RAM).

A useful rule to follow is to order a structure's members from smallest sized to largest. That is to say, declare all 8-bit members prior to 16-bit ones, followed by any 32-bit sized members such as integers and floating point variables. This way the compiler will be able to minimize the number of padding bytes that might be added, and more efficiently access members via the 16-bit Thumb instruction LDRB.

Be aware that using a pragmatic directive to pack structures may reduce the RAM footprint of a program, but additional code must be generated to properly access a packed structure's members. Because members will be aligned to potentially unnatural boundaries, the compiler will automatically generate additional operations to access the data to be manipulated in order to be accessed. This will not only degrade runtime performance, but will also increase a program's size. Unless there is a specific need (such as packetized networking), it is therefore not recommended to use packed structures when optimizing for code size.

```
typedef struct { //naieve member ordering
  char c;
  short s1;
  long l;
  char arr_c[3];
  short s2;
}misaligned;

typedef struct { //intentional 32-bit aligned ordering
  char c;
  char arr_c[3];
  short s1;
  short s2;
  long l;
}aligned;

#pragma pack(1)
typedef struct { // naieve (packed on byte bounadaries)
  char c;
  short s1;
  long l;
  char arr_c[3];
  short s2;
}packed;
#pragma pack
```

Fig 7. Structure alignment examples

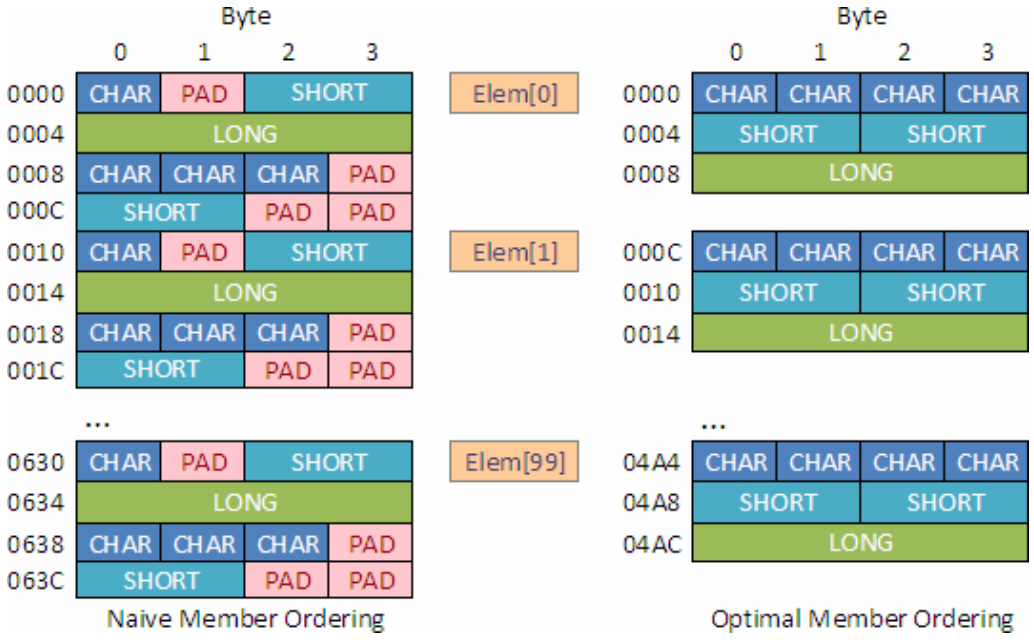
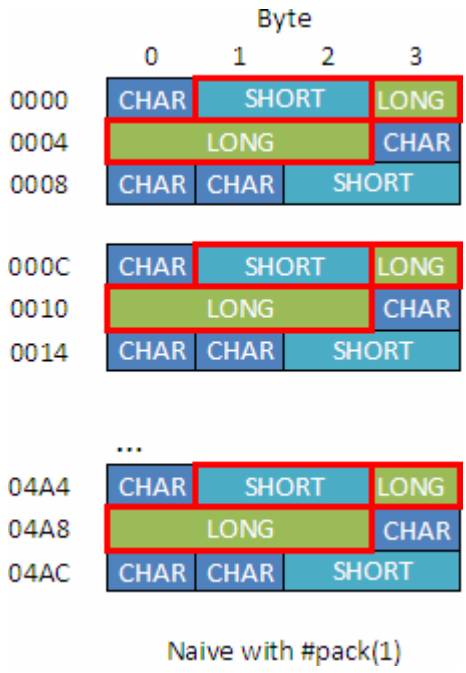


Fig 8. Memory layout (differing member ordering)



(1) Red outline indicates unnatural alignment.

Fig 9. Memory layout (misaligned, packed structure)

3.3 Disassembly

Oftentimes it is useful to inspect the inner workings of a function which is suspiciously large. This is easily done with the GNU objdump utility. By invoking objdump with the "--source" and "--syms" switches will display the final object file's symbol table as well as disassembly intermixed with original C source. Note that LPCXpresso has an option to view disassembly, but it does not invoke the inline source option. Below is a portion of the disassembly of an application specific clock initialization routine.

```
static void InitPLLIRC48(void)
{
  c0:  b580       push {r7, lr}
  c2:  af00       add  r7, sp, #0
    // Set PLL frequency= x4 = 48 MHz from 12 MHz IRC
    LPC_SYSCON->SYSPLLCTRL = 0x3 + (0x2<<5);
  c4:  4a0f       ldr  r2, [pc, #60](104 <InitPLLIRC48+0x44>)
  c6:  2343       movs r3, #67
  c8:  6093       str  r3, [r2, #8]
    // Turn on PLL
    LPC_SYSCON->PDRUNCFG &= ~(1<<7);
  ca:  490e       ldr  r1, [pc, #56](104 <InitPLLIRC48+0x44>)
  cc:  4a0d       ldr  r2, [pc, #52](104 <InitPLLIRC48+0x44>)
  ce:  238e       movs r3, #142
  d0:  009b       lsls r3, r3, #2
  d2:  58d2       ldr  r2, [r2, r3]
  d4:  2380       movs r3, #128
  d6:  439a       bics r2, r3
  d8:  238e       movs r3, #142
  da:  009b       lsls r3, r3, #2
  dc:  50ca       str  r2, [r1, r3]
  …
```

(1) Bold text indicates C language

Fig 10. Example disassembly with inline C source

3.4 Vector table modification

The Cortex-M0 core uses the vector table stored at 0x00000000 at startup to enter low level startup routines and eventually call main(). The vector table also contains the addresses of any interrupt or exception handlers. Be aware that this strategy is recommended only when no other means of space reduction are possible, and in applications with either very specific or no interrupt requirements. If your design requires use of lower power modes, ensure that any required wakeup functionality will continue to operate after any modification to the vector table. Because an incorrectly configured vector table can prevent a system from booting, refer to the “Criterion for Valid User Code” section of your selected device’s Users Manual before attempting these changes. It is also recommend that developers inspect their program images prior to flashing the device, to ensure that the vector table has not been disturbed due to the placement of code and/or constant data in the unused vector locations.

In the event that a program does not use any (or many) interrupts (including power mode wake up events), this vector table is acting as underutilized memory. This table can potentially waste up to 128 bytes of memory, and can be reclaimed relatively easily.

In the rare case that an application does not require interrupts, the declaration of the vector table can simply be truncated and the default linker options will subsequently start the “.text” segment after the vector table.

```
__attribute__ ((section(".isr_vector")))
void (* const g_pfnVectors[])(void) =
{
    &_vStackTop,          // The initial stack pointer
    ResetISR,             // The reset handler
    NMI_Handler,          // The NMI handler
    HardFault_Handler,    // The hard fault handler
    0,                    // Reserved
    0,                    // Reserved
    0,                    // Reserved
    0,                    // Reserved
    0,                    // Reserved
    0,                    // Reserved
    0,                    // Reserved
    SVCall_Handler,       // SVCall handler
    0,                    // Reserved
    0,                    // Reserved
    PendSV_Handler,       // The PendSV handler
    SysTick_Handler       // The SysTick handler

};
```

Fig 11. Redefined vector table (no peripheral interrupts)

If peripheral interrupts are required for an application, manually configuring the linker as seen in Appendix A will allow a developer to reclaim some of the space normally occupied by the vector table as well. The general process of doing this first requires function and constant-data size analysis to make sure there are symbols which will fit contiguously in the space to be freed. In the case of the included example project “LPCXpresso1114_blinky_tiny” there is a look up table of 0x30 bytes used for GPIO routines. After determining that savings can be had, two additional sections must be added to the linker. The example uses “.reset_vector” and “.vector_const” as the names. The vector table is split into two arrays, one of M0 core related handlers, the other of peripheral handlers; the middle section of wakeup handlers is removed.

Complications in modifying the exception vector can be difficult to debug, as such the recommendation is that developers first allocate space for additional functions and variables by splitting the vector array into two parts and test their applications. After the design is verified with the previous change define functions and constants with attributes to place them in the “.vector_const” section.

```
  .text :
  {
    KEEP(*(.reset_vector))   //Portion of vector table related
to M0
    KEEP(*(.vector_const))   //Small functions and constants

    . = 0x7C;
    KEEP(*(.isr_vector))   //Peripheral interrupt handlers

    . = 0xC0;
    *(.text*)
    *(.rodata*)

  } > MFlash32
```

Fig 12. Example linker configuration adding two new sections

4. Example software projects

Included with this application note is an LPCXpresso workspace archive which contains four projects and their required libraries. The workspace contains two “stock” projects and two “tiny” projects. The intent is to show a direct comparison as to the gains that can be attained by following the guidelines in this application note.

One projects pair use the semi-hosting features of LPCXpresso to display a running count to the user. Table 3 shows a comparison in resulting image file sizes. Note that the standard project is 3X the size of the LPC1111’s available flash memory.

Table 3. Comparison of “hello_world_semihosting” and “hello_world_semihosting_tiny”

Type	Bytes	LPC1111 Usage	LPC1114 Usage
Stock	25,772	314%	79%
Tiny	2,870	35%	9%

The second project is even more simplistic, it simply blinks an on board LED. The tiny version of this project makes use of several advanced concepts covered in this document, please be aware that it has custom linker configuration and startup routines.

Table 4. Comparison of “LPCXpresso1114_blinky” and “LPCXpresso1114_blinky_tiny”

Type	Bytes	LPC111 Usage	LPC1114 Usage
Stock	1,448	18%	4%
Tiny	516	6%	2%

5. Conclusion

The strategies and tools covered in this application note should give developers the ability to ensure their programs will fit into their target hardware with minimal effort. By analyzing their code, they will be better suited to meet product requirements using devices with limited program storage. While this application note has been LPC11XX centric, the recommendations made here can largely be applied to any NXP device.

6. Appendix A

Several steps are required to specify a custom linker script set.[2] First make a new folder in the project named “custom_ld”. This will store the customized script files. Next copy the automatically generated linker scripts with “.ld” extensions from the active target folder (typically “Debug” or “Release”) into the “custom_ld” folder. Fig 13 illustrates where the automatically generated scripts are stored in the project. Rename the file as per Table 5. Before you modify the content of the scripts, you need to ensure that the toolchain is configured to use them as seen in Fig 14. To redefine the memory map of the ‘virtual’ target device, manually edit Custom_mem.ld to suit your needs; in the example shown in Fig 15 the flash section has been expanded to 1 MB. Finally update Custom.ld to point to the Custom_mem.ld and Custom_lib.ld scripts in the newly created custom_ld folder. An example of this is shown in Fig 16.

Table 5. Renamed linker scripts

Original name	Renamed copy
*PROJECTNAME*_Debug.ld	Custom.ld
*PROJECTNAME*_Debug_mem.ld	Custom_mem.ld
*PROJECTNAME*_Debug_lib.ld	Custom_lib.ld

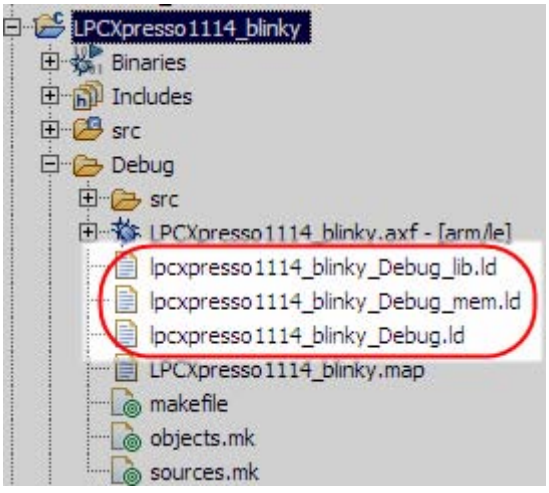


Fig 13. Automatically generated linker scripts

2. http://www.nxp.com/redirect/lpcxpresso.code-red-tech.com/LPCXpresso/node/31

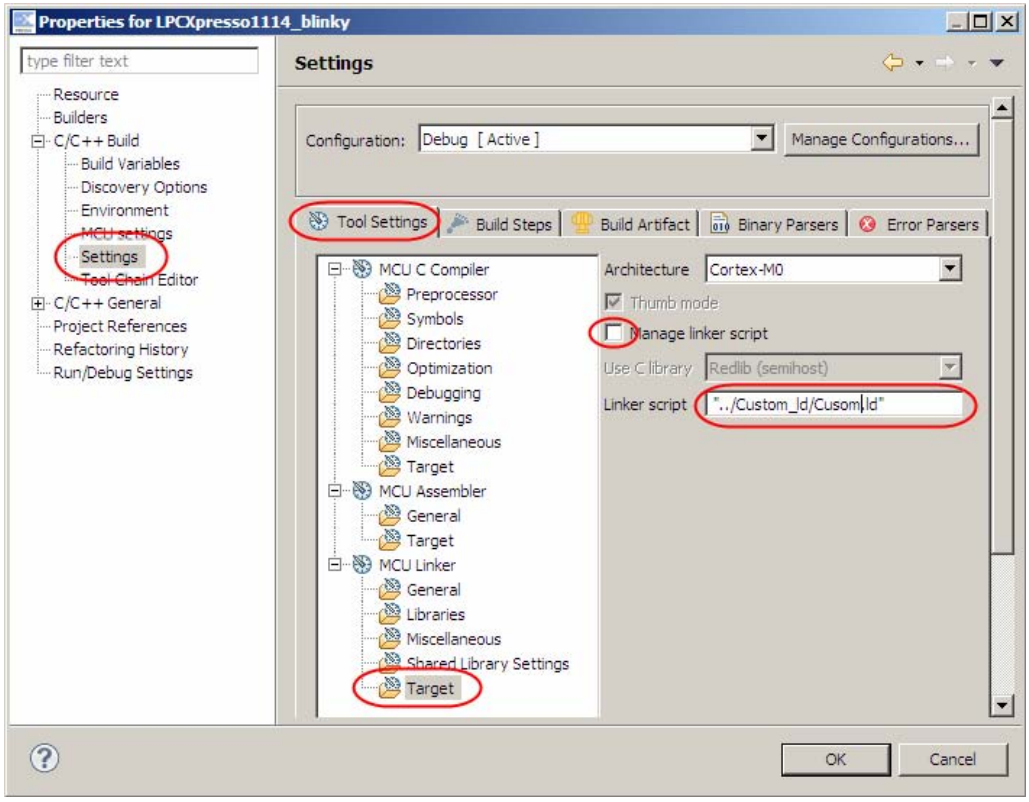


Fig 14. Specifying custom linker file

```
MEMORY
{
  /* Define each memory region */
  MFlash32 (rx) : ORIGIN = 0x0, LENGTH = 0x100000 /* 1M */
  RamLoc8 (rwx) : ORIGIN = 0x10000000, LENGTH = 0x2000 /* 8k */
}
  /* Define a symbol for the top of each memory region */
  __top_MFlash32 = 0x0 + 0x100000;
  __top_RamLoc8 = 0x10000000 + 0x2000;
```

Underline indicates changes from original file

Fig 15. Artificially large memory definition

```
/*
 * GENERATED FILE - DO NOT EDIT
 * (C) Code Red Technologies Ltd,
 * Generated C linker script file for LPC1114
*/
INCLUDE "../custom_ld / Custom_lib.ld "
INCLUDE "../custom_ld /Custom_mem.ld "

ENTRY(ResetISR) …
```

Underline indicates changes from original file

Fig 16. Modifications to “Custom.ld” linker script

7. Legal information

7.1 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.

7.2 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.

7.3 Trademarks

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

8. List of figures

9. List of tables

10. Contents

For more information, please visit: http://www.nxp.com
For sales office addresses, please send an please send an email to: salesaddresses@nxp.com

Date of release: 2 August 2010
Document identifier: AN10963