



Introduction

This document explains how to measure the code size of STxP70-4 applications. It describes two utilities (**stxp70-size** and **sxmeminfo**) delivered with the STxP70 toolset that measure the size of sections in the binary file.

In the early stages of firmware development, the size of C libraries is usually taken into account when measuring code size; however, in most embedded applications with tight memory constraints, the C libraries will not be part of the final memory footprint. [Chapter 4: Considering the size of libraries on page 9](#) describes a method of measuring the generated code size versus the library size, which can be used during the early stages of firmware development to get an idea of the real application code size.

[Chapter 6: Considering the object file size on page 12](#) describes how measuring the code size on object files leads to a bias in the measurement. This is because the STxP70-4 architecture has a variable length instruction set with 16-, 32- and 48-bit instructions. During the compile stage, all possible instructions are provisioned as 48-bit instructions. During the link stage, the linker relaxes the code to the real 16-, 32- or 48- bit instructions. Moreover, dead code elimination is performed at the post-link stage using the binary optimizer. As a consequence, measuring the code size on the final executable file is the only way to get the real code size.

Finally, [Chapter 7](#) provides some tips to reduce the code size on STxP70.

Contents

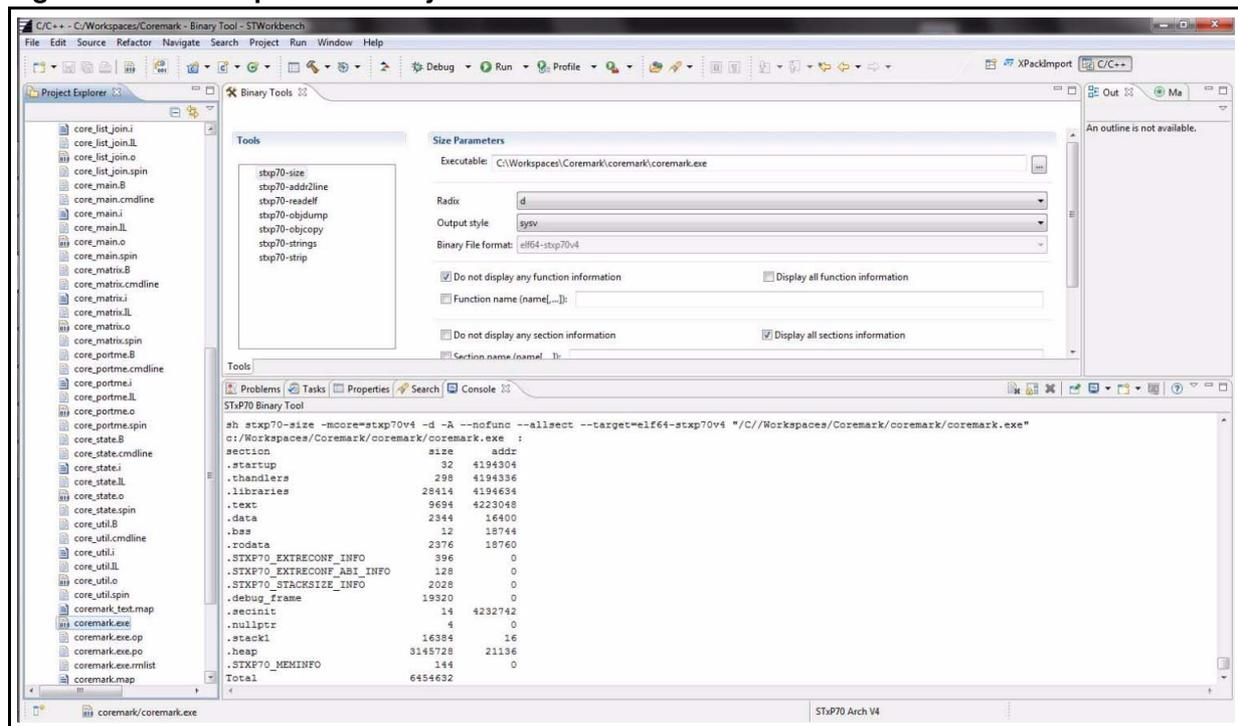
1	The stxp70-size utility	3
1.1	Code sections	3
1.2	Data sections	4
1.2.1	Read-only data sections	4
1.2.2	Read-write data sections	4
1.3	Other sections	5
2	The sxmeminfo utility	6
3	Sizing the program and data memories	8
3.1	Non-volatile program memory	8
3.2	Volatile program memory	8
4	Considering the size of libraries	9
4.1	Link script file	9
5	Code generation steps (reminder)	11
6	Considering the object file size	12
7	Tips and tricks to reduce code size	15
7.1	Optimize for size	15
7.2	Avoid using IPA	15
7.3	Use the SDA section	15
7.4	Write your own I/O functions	15
7.5	Use the floating point extension	15
8	Reference documentation	16
9	Revision history	17

1 The stxp70-size utility

The STxP70 toolset provides a utility called **stxp70-size** that computes the size of each section in an ELF file.

The output of this utility depends on the application but the commonly used sections are described in this chapter.

Figure 1. The stxp70-size utility in STWorkbench



1.1 Code sections

The following sections are considered as code. In other words, the measurement of code size has to take these sections into account:

```
.startup      // Boot address code
.ivtable     // Interrupt vector table
.handlers    // Hardware and software TRAP handlers
.ihandlers   // Interrupt handlers
.text        // Code section
```

1.2 Data sections

Data sections can be split in two types:

- read-only sections
- read-write sections

1.2.1 Read-only data sections

```
.rodata      // Read only data
.tda_ro      // Read only data in tiny data area (deprecated section)
.sda_ro1     // 8-bit aligned read only data in small data area
.sda_ro2     // 16-bit aligned read only data in small data area
.sda_ro4     // 32-bit aligned read only data in small data area
.sda_ro8     // 64-bit aligned read only data in small data area
.da_ro1      // 8-bit aligned read only data in data area
.da_ro2      // 16-bit aligned read only data in data area
.da_ro4      // 32-bit aligned read only data in data area
.da_ro8      // 64-bit aligned read only data in data area
.secinit     // Memory regions initialization table
```

1.2.2 Read-write data sections

```
.data        // Static global data area
.bss         // Uninitialized global and static data area
.tda_data    // Static tiny data area (deprecated section)
.tda_bss     // Uninitialized global and static tiny data area (deprecated
// section)
.sda_data1   // 8-bit aligned static tiny data area
.sda_data2   // 16-bit aligned static tiny data area
.sda_data4   // 32-bit aligned static tiny data area
.sda_data8   // 64-bit aligned static tiny data area
.sda_bss1    // 8-bit aligned uninitialized global and static small data
// area
.sda_bss2    // 16-bit aligned uninitialized global and static small data
// area
.sda_bss4    // 32-bit aligned uninitialized global and static small data
// area
.sda_bss8    // 64-bit aligned uninitialized global and static small data
// area
.da_data1    // 8-bit aligned static data area
.da_data2    // 16-bit aligned static data area
.da_data4    // 32-bit aligned static data area
.da_data8    // 64-bit aligned static data area
.da_bss1     // 8-bit aligned uninitialized global and static data area
.da_bss2     // 16-bit aligned uninitialized global and static data area
.da_bss4     // 32-bit aligned uninitialized global and static data area
.da_bss8     // 64-bit aligned uninitialized global and static data area

.stack1      // Stack used by first context
.heap        // Dynamic data area
.ctors       // Constructors data
.dtors       // Destructors data
```

Note: *The .heap and .stack sections represent the memory size allocated for these sections in the link script file. They do not represent the amount of stack and heap actually used by the application.*

1.3 Other sections

Other sections are referenced in the ELF file but will not be part of the memory footprint on the device. This means debugging information can be kept for a program image residing in program memory without this information appearing in the program memory itself.

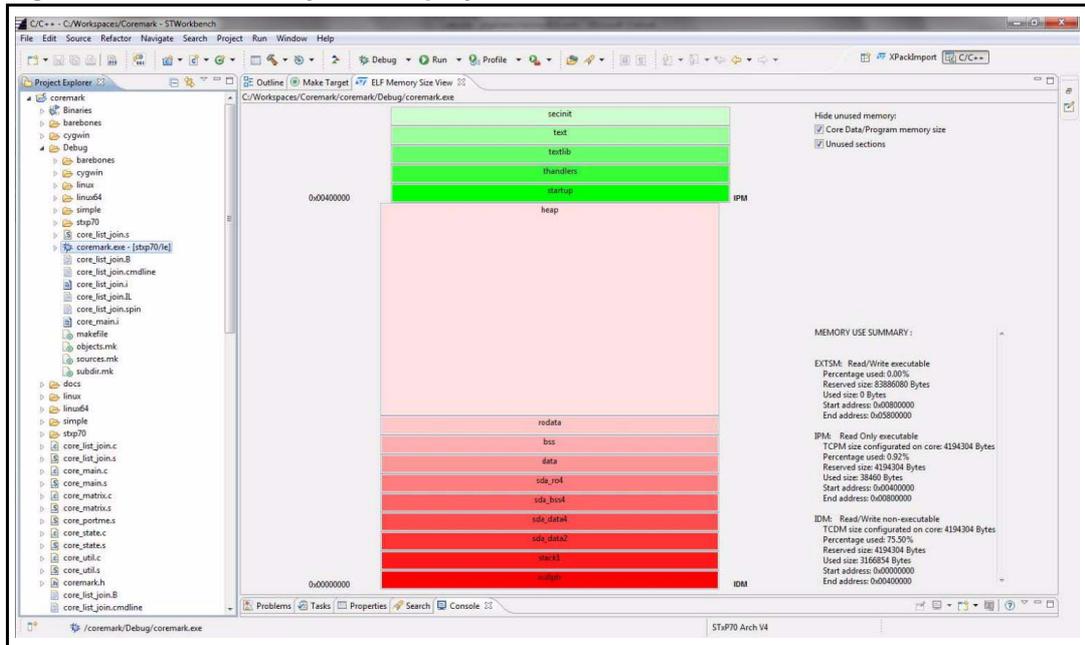
```
.debug_srcinfo
.debug_sfnames
.debug_aranges
.debug_pubnames
.debug_info
.debug_abbrev
.debug_line
.debug_frame
.debug_str
.debug_loc
.debug_macinfo
.debug_weaknames
.debug_funcnames
.debug_typenames
.debug_varnames

.STXP70_EXTRECONF_INFO
.STXP70_EXTRECONF_ABI_INFO
.STXP70_STACKSIZE_INFO
.STXP70_MEMINFO
```

2 The sxmeminfo utility

The functionality of the **sxmeminfo** utility is similar to the **stxp70-size** utility (see [Chapter 1](#)); however, it provides additional information such as the used code and data memory compared to the available memory sizes as defined in the link script file. It also provides a graphical interface.

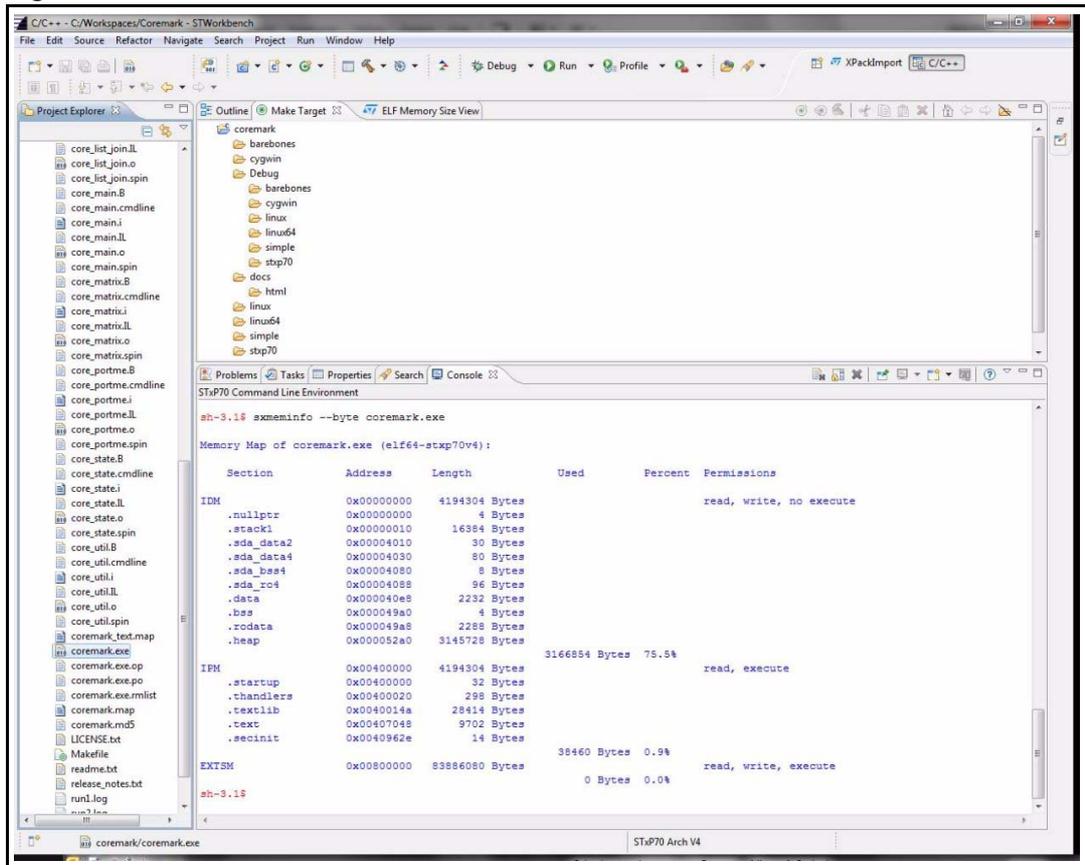
Figure 2. ELF memory size display in STWorkbench



To display a graph equivalent to the one in [Figure 2](#), right click on the STxP70 executable (`coremark.exe` in this example) and select **ELF Memory Size Display**.

The **sxmeminfo** utility is also available on the command line as shown in [Figure 3](#).

Figure 3. sxmefinfo command line in STWorkbench



3 Sizing the program and data memories

This chapter differentiates between two kinds of usage:

- non-volatile memory (the processor is stand-alone and boots from non-volatile memory)
- volatile memory (both the program and data memories are volatile and loaded at boot time by an external processor)

3.1 Non-volatile program memory

In this case, all the code sections as well as the read-only data sections have to be put in non-volatile program memory. In other words, all the sections described in [Section 1.1 on page 3](#) and [Section 1.2.1 on page 4](#) have to be put in non-volatile program memory.

All the remaining data sections (described in [Section 1.2.2 on page 4](#)) have to be placed in volatile data memory.

Note: If you are trying to minimize the memory footprint, you can leave the `.rodata` sections inside the non-volatile program memory during execution and avoid a copy in RAM. But for performance reasons, it may be wise to copy the `.rodata` sections described in [Section 1.2.1 on page 4](#) (except the `.secinit` section) in data RAM as the processor is less efficient when accessing data from program memory.

3.2 Volatile program memory

If a host places the STxP70 code and data in its memory at start-up, all code sections have to be put in volatile program memory. In fact, only the tightly coupled program and external memories are executable by the STxP70 processor.

All data sections can be put in either volatile program memory or volatile data memory but code execution will always be more efficient if data is placed in tightly coupled data memory. An example of section placement is shown in [Figure 4](#).

Figure 4. Section placement with volatile program memory

```
sh stxp70-readelf -mcore=stxp70v4 -l -W "/C//Workspaces/Coremark/coremark/coremark.exe"

Elf file type is EXEC (Executable file)
Entry point 0x400000
There are 4 program headers, starting at offset 64

Program Headers:
Type           Offset  VirtAddr           PhysAddr           FileSiz  MemSiz   Flg  Align
LOAD           0x001000 0x0000000000000000 0x0000000000000000 0x000004 0x004010 RW   0x1000
LOAD           0x001010 0x00000000000004010 0x00000000000004010 0x000990 0x000994 RW   0x1000
LOAD           0x0019a8 0x000000000000049a8 0x000000000000049a8 0x0008f0 0x3008f8 RW   0x1000
LOAD           0x003000 0x000000000000400000 0x000000000000400000 0x00963c 0x00963c RWE  0x1000

Section to Segment mapping:
Segment Sections...
00  .nullptr .stack1
01  .sda_data2 .sda_data4 .sda_bss4 .sda_ro4 .data .bss
02  .rodata .heap
03  .startup .handlers .textlib .text .secinit
```

4 Considering the size of libraries

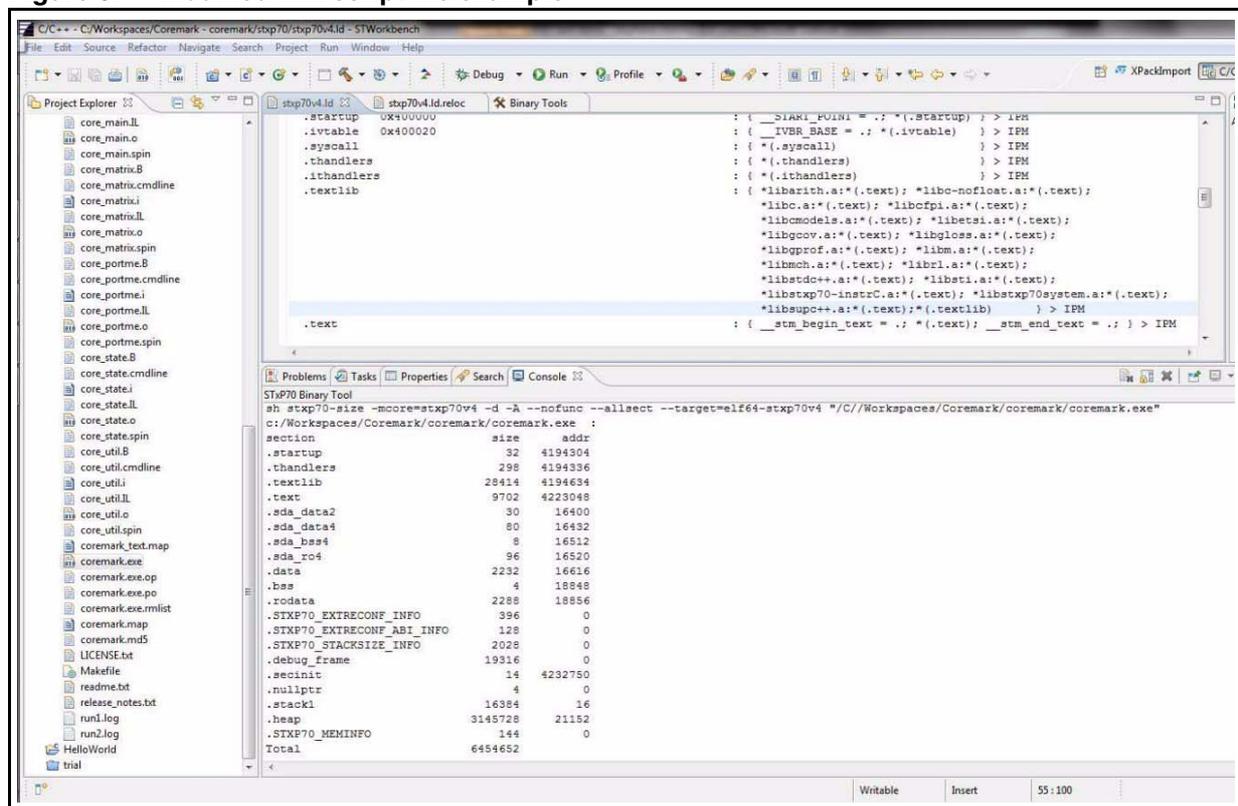
In the early stages of firmware development, the size of C libraries is usually taken into account when measuring code size; however, in most embedded applications with tight memory constraints, the C libraries will not be part of the final memory footprint (refer to [Chapter 7: Tips and tricks to reduce code size on page 15](#) for some hints on achieving this).

This chapter describes a method of measuring the generated code size versus the library size, which can be used during the early stages of firmware development to get an idea of the real application code size.

4.1 Link script file

Redirect the libraries to a specific `.text` section called `.textlib` by modifying the link script file as shown in [Figure 5](#).

Figure 5. Modified link script file example



The `stxp70v4.ld.reloc` file also has to be modified to redirect the libraries to this specific section at high optimization levels (Os, O2, O3 and O4), see [Figure 6](#).

Figure 6. Relocatable file

```

SECTIONS
{
/* Read-only sections, merged into text segment: */
.startup      0x0 : { *(.startup)          }
.ivtable      0x0 : { *(.ivtable)         }
.syscall      0x0 : { *(.syscall)         }
.thandlers    0x0 : { *(.thandlers)       }
.ithandlers   0x0 : { *(.ithandlers)      }
.textlib      0x0 : { *libarith.a:*(.text); *libc-nofloat.a:*(.text); *libc.a:*(.text);
                  *libcfpi.a:*(.text); *libcmmodels.a:*(.text); *libetsi.a:*(.text);
                  *libgcov.a:*(.text); *libgloss.a:*(.text); *libgprof.a:*(.text);
                  *libm.a:*(.text); *libmch.a:*(.text); *libl.a:*(.text);
                  *libstdc++.a:*(.text); *libsti.a:*(.text); *libstxp70-instrC.a:*(.text);
                  *libstxp70system.a:*(.text); *libsupc++.a:*(.text); }
.text         0x0 : { *(.text)           }
    
```

The .text section is now split in two sections: the application code itself in the .text section and the C libraries in the .textlib section. These modifications to the .ld and .ld.reloc files are implemented by default in the STxP70 toolset R2012.2 and later versions.

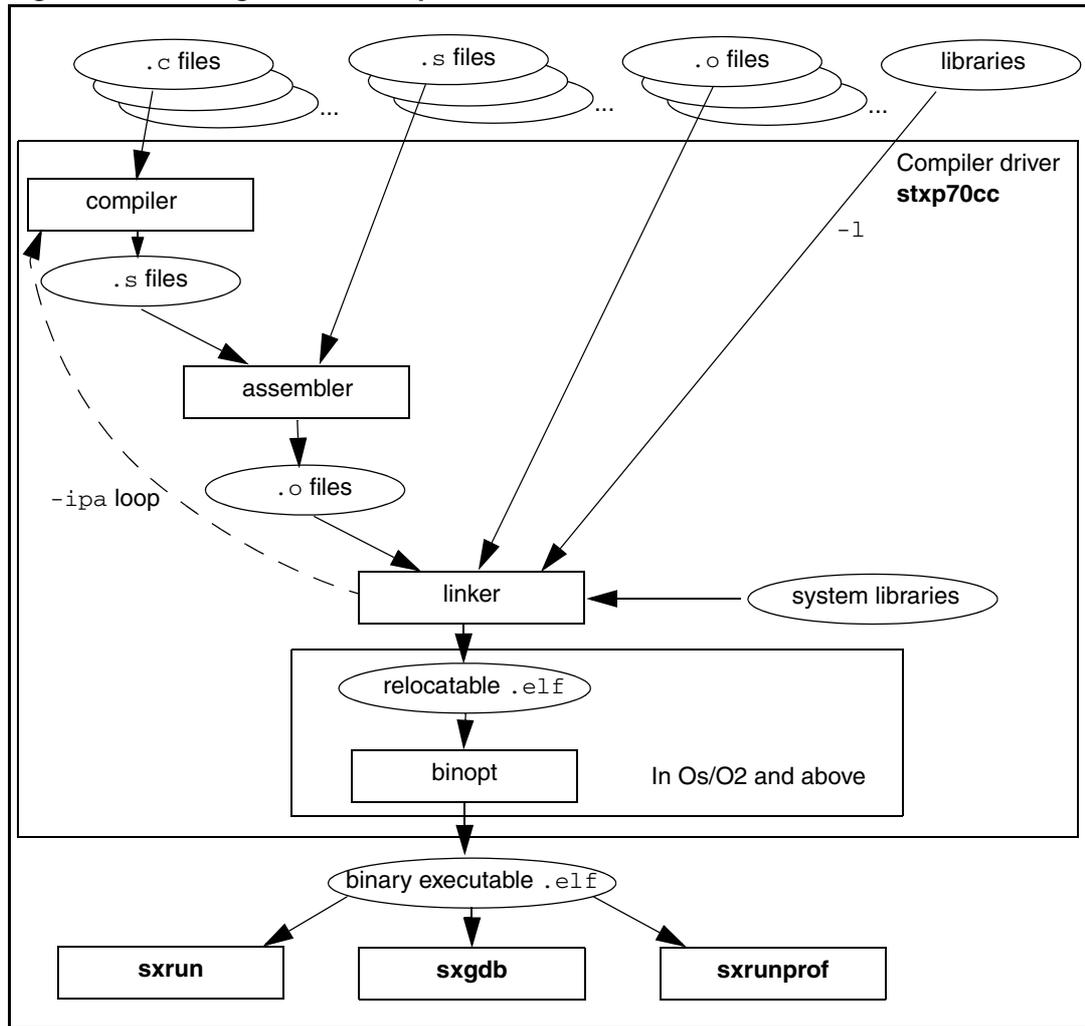
Note: As shown in [Figure 5](#), the library size is much bigger than the application size itself. This is often the case for small applications so only reference library functions that are required.

5 Code generation steps (reminder)

This chapter provides a reminder of the steps taken during code generation. [Figure 7](#) illustrates the complete steps.

1. The `.c` source files are individually compiled into generated `.s` assembly files.
2. The generated assembly files and `.s` source files are individually assembled to generate `.o` object files.
3. All `.o` objects files are linked with the libraries to generate the final binary file in ELF format.

Figure 7. Code generation steps



6 Considering the object file size

Measuring the code size on object files leads to a bias in the measurement. The STxP70-4 architecture has a variable length instruction set with 16-, 32- and 48-bit instructions. During the compile stage, all possible instructions are provisioned as 48-bit instructions. During the link stage, the linker relaxes the code to the real 16-, 32- or 48- bit instructions. Moreover, dead code elimination is performed at the post-link stage using the binary optimizer. As a consequence, measuring the code size on the final executable file is the only way to get the real code size.

In the example in [Figure 8](#), adding the code size of the different `.o` files (that is, the sum of the `.text` section sizes excluding libraries) will lead to 17526 bytes. But the real application size is 10318 bytes as shown in [Figure 9](#). As a conclusion, we can say that measuring the code size on object files is not a valid measurement.

Figure 8. Measuring the code size on object files

```

$ stxp70v4-size *.o
core_list_join.o :
section                size  addr
.text                  4208  0
.data                   0      0
.bss                    0      0
.debug_frame           816     0
.STXP70_EXTRECONF_INFO 396     0
.STXP70_EXTRECONF_ABI_INFO 128    0
.STXP70_STACKSIZE_INFO 144     0
Total                  11384

core_main.o :
section                size  addr
.text                  2762  0
.data                   44     0
.bss                    0      0
.rodata                 1240  0
.debug_frame           300     0
.STXP70_EXTRECONF_INFO 396     0
.STXP70_EXTRECONF_ABI_INFO 128    0
.STXP70_STACKSIZE_INFO  24     0
Total                  9788

core_matrix.o :
section                size  addr
.text                  4888  0
.data                   0      0
.bss                    0      0
.debug_frame           808     0
.STXP70_EXTRECONF_INFO 396     0
.STXP70_EXTRECONF_ABI_INFO 128    0
.STXP70_STACKSIZE_INFO 108     0
Total                  12656

core_portme.o :
section                size  addr
.text                   272     0
.data                    4      0
.bss                     8      0
.debug_frame            244     0
.STXP70_EXTRECONF_INFO 396     0
.STXP70_EXTRECONF_ABI_INFO 128    0
.STXP70_STACKSIZE_INFO  96     0
Total                   2296

core_state.o :
section                size  addr
.text                  3672  0
.data                   64     0
.bss                    0      0
.rodata                 304     0
.debug_frame            268     0
.STXP70_EXTRECONF_INFO 396     0
.STXP70_EXTRECONF_ABI_INFO 128    0
.STXP70_STACKSIZE_INFO  36     0
Total                   9736

core_util.o :
section                size  addr
.text                  1724  0
.data                   0      0
.bss                    0      0
.debug_frame            228     0
.STXP70_EXTRECONF_INFO 396     0
.STXP70_EXTRECONF_ABI_INFO 128    0
.STXP70_STACKSIZE_INFO  84     0
Total                   5120

```

Figure 9. Measuring the code size on the final executable

```
$ stxp70v4-size coremark.exe
coremark.exe :
section          size      addr
.startup         32      4194304
.thandlers       298      4194336
.textlib        28414     4194634
.text           10318     4223048
.data           2344      16400
.bss            12       18744
.rodata         2384     18760
.STXP70_EXTRECONF_INFO      396      0
.STXP70_EXTRECONF_ABI_INFO  128      0
.STXP70_STACKSIZE_INFO     2028     0
.debug_frame    19508     0
.secinit        14     4233366
.nullptr        4         0
.stack1        16384     16
.heap          3145728    21152
.STXP70_MEMINFO      144      0
Total          6456272
```

Note: Observe again the relatively large size of the libraries (28414 bytes) compared with the generated code size (10318 bytes) and refer to [Chapter 7](#) which includes tips on reducing the size of these libraries.

7 Tips and tricks to reduce code size

7.1 Optimize for size

The C compiler provides efficient optimization options `-Os` and `-O2`. Use them on all files and as linker options as well. Refer to the *STxP70 compiler user manual (8027948)* for more details on how to use these options.

7.2 Avoid using IPA

Interprocedural analysis (IPA) is mainly a speed oriented optimization that generally impacts code size. Avoid using the `-ipa` option if your goal is to compile for size.

7.3 Use the SDA section

On some applications, using the small data area (SDA) instead of the standard data area (DA) reduces code size. Data accessed in this section only uses one instruction instead of two. Depending on the size of each of those instructions, this may help reduce code size.

To place all variables in the SDA memory area, use the `-Msda=all` compiler option.

If the variables do not all fit in this memory area, place only the variables with a given alignment (`-Msda=1`, `-Msda=2`, `-Msda=4` or `-Msda=8`) in these memory areas.

Individual variables can also be placed in these memory areas using attributes (refer to the *STxP70 compiler user manual (8027948)* for details).

7.4 Write your own I/O functions

`printf`, `scanf` and some derivative library functions are very powerful but very large functions. They support different variable types that may not be required if the required output is a fixed string or a fixed number of values of known types.

Instead, `puts/putchar` can be used to deal with constant strings or characters. You may also write your own I/O functions if you know the number and type of the variables you need to display.

7.5 Use the floating point extension

If the application code contains floating point data, it is better to use the floating point extension (FPx). Otherwise, the C compiler embeds the necessary floating point emulation library functions that are usually code consuming.

The FPx is a single precision floating point extension and therefore the `-fshort-double` option should be added to avoid embedding the 64-bit floating point emulation library.

```
$ stxp70cc -mcore=stxp70v4 -Mextension=fpx -fshort-double -Os  
-corecfg1=0xc00 -corecfg=0x7390149c MyFloatingPointApplication.c
```

8 Reference documentation

- *STxP70-4.4.0 core and instruction set architecture manual* (Doc ID 023738)
- *STxP70-4C processor supercore datasheet* (Doc ID 023687)
- *STxP70 toolset user manual* (8204323)
- *STxP70 compiler user manual* (8027948)
- *STxP70 utilities reference manual* (8210925)
- *STxP70 compiler guidelines* (available in the toolset documentation portal)

9 Revision history

Table 1. Document revision history

Date	Revision	Changes
30-Jan-2013	1	Initial release.

Please Read Carefully:

Information in this document is provided solely in connection with ST products. STMicroelectronics NV and its subsidiaries ("ST") reserve the right to make changes, corrections, modifications or improvements, to this document, and the products and services described herein at any time, without notice.

All ST products are sold pursuant to ST's terms and conditions of sale.

Purchasers are solely responsible for the choice, selection and use of the ST products and services described herein, and ST assumes no liability whatsoever relating to the choice, selection or use of the ST products and services described herein.

No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted under this document. If any part of this document refers to any third party products or services it shall not be deemed a license grant by ST for the use of such third party products or services, or any intellectual property contained therein or considered as a warranty covering the use in any manner whatsoever of such third party products or services or any intellectual property contained therein.

UNLESS OTHERWISE SET FORTH IN ST'S TERMS AND CONDITIONS OF SALE ST DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY WITH RESPECT TO THE USE AND/OR SALE OF ST PRODUCTS INCLUDING WITHOUT LIMITATION IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE (AND THEIR EQUIVALENTS UNDER THE LAWS OF ANY JURISDICTION), OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

UNLESS EXPRESSLY APPROVED IN WRITING BY TWO AUTHORIZED ST REPRESENTATIVES, ST PRODUCTS ARE NOT RECOMMENDED, AUTHORIZED OR WARRANTED FOR USE IN MILITARY, AIR CRAFT, SPACE, LIFE SAVING, OR LIFE SUSTAINING APPLICATIONS, NOR IN PRODUCTS OR SYSTEMS WHERE FAILURE OR MALFUNCTION MAY RESULT IN PERSONAL INJURY, DEATH, OR SEVERE PROPERTY OR ENVIRONMENTAL DAMAGE. ST PRODUCTS WHICH ARE NOT SPECIFIED AS "AUTOMOTIVE GRADE" MAY ONLY BE USED IN AUTOMOTIVE APPLICATIONS AT USER'S OWN RISK.

Resale of ST products with provisions different from the statements and/or technical features set forth in this document shall immediately void any warranty granted by ST for the ST product or service described herein and shall not create or extend in any manner whatsoever, any liability of ST.

ST and the ST logo are trademarks or registered trademarks of ST in various countries.

Information in this document supersedes and replaces all information previously supplied.

The ST logo is a registered trademark of STMicroelectronics. All other names are the property of their respective owners.

© 2013 STMicroelectronics - All rights reserved

STMicroelectronics group of companies

Australia - Belgium - Brazil - Canada - China - Czech Republic - Finland - France - Germany - Hong Kong - India - Israel - Italy - Japan - Malaysia - Malta - Morocco - Philippines - Singapore - Spain - Sweden - Switzerland - United Kingdom - United States of America

www.st.com