AN3505
Application note

STxP70 - how to build and test embedded software

Abstract

An embedded application is an application that runs standalone without depending upon the support of a debugger or a runner.

The goal of this application note is to explain how to produce and to test such an application. It covers the following tasks:

- how to initialize the STxP70 processor
- how a C program is initialized (the startup routine)
- what to do in a link script file so that the startup can perform the initialization
- how to build and to test the binary of an application with the debugger
- putting all the elements together

The *STxP70-4.4.0 core and instruction set architecture reference manual* (Doc ID 023738) and the *STxP70 compiler user manual* (8027948) may be of interest for obtaining a better understanding this application note.

This application note explains in detail the actions performed prior to the execution of an application, and the relationships that exist between the startup routines and the link file.

The examples are based on the STxP70-4 utilities. For more information see *STxP70-4 utilities reference manual* (8210925).

Associated documents

- *STxP70-4.4.0 core and instruction set architecture reference manual* (Doc ID 023738)
- *STxP70 compiler user manual* (8027948)
- *STxP70 professional toolset user manual* (7833754)
- *STxP70-4 utilities reference manual* (8210925)

Contents

1 Initializing the STxP70 processor

Some hardware resources of the STxP70 must be explicitly initialized by software following a hardware reset (this includes the stack pointer SP and the enable status of the hardware loops). The file responsible of this initialization is called `crt0.s` (located in the directory `stxp70cc/version-stxp70cc/src/lib/newlib/newlib/libc/sys/stxp70`).

Note: Depending upon the STxP70 core configuration, this file is called `crt0-nohwloop.s` *when hardware loops are not implemented in the STxP70 core.*

This file is a good starting point from which to write a dedicated routine that performs the initialization routine of a complete system. The only point where particular care should be taken is the initialization of the stack pointer. *Figure 1* is an excerpt from `crt0.s` that shows how it normally deals with stack pointer initialization:

Figure 1. Excerpt from crt0.s

```
...snip...

// ***************************
// Stack pointer initialization
// ***************************

// __syscallbkp1 is handled by a potential debugger if connected.
// nop is replaced by a bkp. When the debugger reaches bkp associated
// to __syscallbkp1, it sets r0 to argc value.
g7? MAKECST    r0, 0  ;;  // resets potential value of argc
__syscallbkp1:
    nop          ;;
    cmpeq      g0, r0, 0   ;;
g0? MAKEABS    sp, __stm_end_stack1  ;;

...snip...
```

The stack pointer is conditionally initialized by the application. The reason for this is that when the application is run from the debugger (or from the runner), arguments may be passed to the `main()` function, which means that the debugger (or runner) may have to modify the value of the stack pointer. In the contrary case, when the application runs standalone, the stack pointer MUST be initialized by the application.

When the condition is managed by the debugger (or by the runner) automatically, it is at a cost of three extra instructions (these instructions are saved in the final application). In the code in *Figure 1*, when a debugger (or a runner) is connected, it automatically looks for a `__syscallbkp1` symbol within the application. If it finds it, it replaces the associated instruction (a `nop` in this case) with a `bkp` that it handles by returning 1 in `r0`. Therefore, the comparison that follows clears `g0`, ensuring that the two instructions following are not executed.

In the case where no debugger (or runner) is connected, the `nop` instruction following `__syscallbkp1` label is left unchanged and so the two `sp` initialization instructions are executed.

If code size is at a premium, savings can be obtained by removing the conditional part of the code and initializing SP explicitly. The code shown in *Figure 1* can consequently be replaced by the code in *Figure 2*.

Note: However, if the application is run from the debugger, any arguments (`argc` *or* `argv`*) passed to* `main` *are lost.*

Figure 2. Excerpt from crt0.s (modified)

```
...snip...

        // ***************************
        // Stack pointer initialization
        // ***************************
       MAKEABS    sp, __stm_end_stack1  ;;

...snip...
```

2 Initialization of a C program

An embedded application is an application that runs without depending upon the support of a debugger or a runner. This means that initialization of the C program must be performed by the application itself and cannot be done by the debugger.

In this context, initialization means the implicit initialization of global variables. When a debugger runs an application, then the debugger itself initializes these variables (thus avoiding the need to insert extra cycles into the startup routine). This application note describes the following:

- how the debugger performs this initialization
- how to build an application that performs the same initialization by itself
- how to test such an executable with the debugger.

2.1 Initialization of global variables by the debugger

Consider the code fragment called `embedded.c` in *Figure 3*:

Figure 3. Example code to demonstrate initialization of global variables

```
#include <stdio.h>
int A[10]={0,1,2,3,4,5,6,7,8,9};

int main(void) {
          int i;
          for(i=0;i<10;i++) {
               printf("A[%d]=%d\n",i,A[i]);
          }
          return 0;
}
```

The simplest way to compile this code is with the following command line:

```
stxp70cc -mcore=stxp70v4 -g -o embedded.exe embedded.c
```

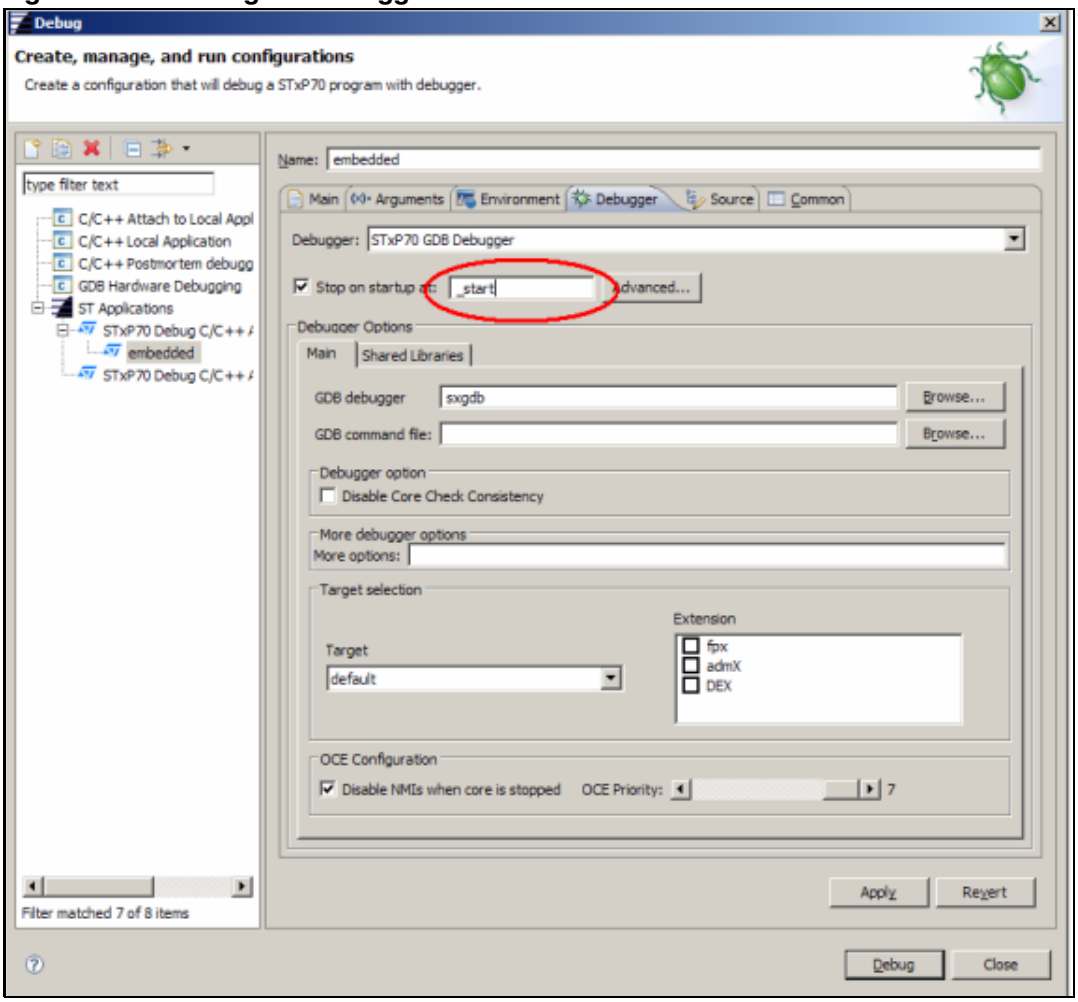
Note: By default, the link script used is `sx_valid.ld`*, located at* *Toolset_Root/stxp70cc/4.3/stxp70v4/lib/ldscript.*

Next, invoke the application with the debugger. This is done with the following command line:

```
sxgdb -target=default embedded.exe
```

Alternatively, the debugger can be invoked from within STWorkbench. The default for the **Stop on startup at:** option is for the debugger to stop when it reaches `main`; modify this so that the debugger stops when it reaches `_start` instead. See *Figure 4*.

Figure 4. Invoking the debugger from STWorkbench



Examining the global variable `A` at this point shows that it is already initialized. The debugger has performed this initialization itself.

To do this, the debugger copied all the sections described in the ELF file to their respective memory locations. The global data variable `A` is located into the `.data` section. This section has an identical value for both its Load Memory Address (or LMA, also known as Physical Address, that is, the address where this section is physically loaded) and for its Virtual Memory Address or VMA (the runtime address).

This can be seen with the **stxp70-readelf** utility provided in the STxP70 Toolset. The output of the utility is shown in *Figure 5*.

Figure 5. stxp70-readelf output of embedded.exe

```
$ stxp70-readelf -mcore=stxp70v4 -l embedded.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              Flags  Align
  LOAD           0x0000000000001000 0x0000000000000000 0x0000000000000000
                 0x0000000000000004 0x0000000000004010  RW     1000
  LOAD           0x0000000000001010 0x0000000000004010 0x0000000000004010
                 0x0000000000000910 0x0000000000000914  RW     1000
  LOAD           0x0000000000001928 0x0000000000004928 0x0000000000004928
                 0x0000000000000570 0x0000000000300578  RW     1000
  LOAD           0x0000000000002000 0x0000000000400000 0x0000000000400000
                 0x0000000000009ba4 0x0000000000009ba4  RWE    1000

 Section to Segment mapping:
  Segment Sections...
   00     .nullptr .stack1
   01     .data .bss
   02     .rodata .heap
   03     .startup .thandlers .textlib .text .secinit
$
```

2.2 Initialization of global variable by the application

An embedded application cannot use the debugger to initialize its global variables. Instead, modify the link script, `sx_valid.ld`, to place the LMA in ROM or Flash Memory. This means that the LMA has a different value to the VMA.

2.2.1 Link script

Use the following procedure to create a modified version of the link script.

1. Copy the default `sx_valid.ld` file to the working directory and rename it `embedded.ld`.
2. Modify this file to correspond with the listing in *Figure 6*. The modified lines are in **bold** text.

Figure 6. embedded.ld

```
MEMORY
   {
   IDM    (rw!x): org = 0x000000, len = 512K    /* Internal Data Memory      */
   IPM    (rx)  : org = 0x400000, len = 512K    /* Internal Program Memory   */
   IROM   (r!wx): org = 0x480000, len = 32K     /* Internal ROM              */
   EXTSM  (rwx) : org = 0x800000, len = 80M     /* External System Memory    */
   }
   ENTRY(__START_POINT)
   SECTIONS
   {
   ...
   .text     : { __stm_begin_text = .; *(.text); __stm_end_text = .; } > IPM
   ...
   .secinit : { CREATE_SECINIT_TABLE ; __IROM = . ; } > IROM
   ...
   .data     : AT (__IROM) { *(.data) } > IDM
   __IROM = ABSOLUTE(_IROM_BASE) + SIZEOF(.data);
   ...
   }
```

This script now creates a new memory region called `IROM` that starts at `0x480000` and is 32 Kbytes in length. The section contribution of `.secinit` has been moved up and augmented with the `__IROM` symbol definition. The link script uses the symbol `__IROM` as the next free location in the `IROM` memory region.

As with the standard link script `sx_valid.ld`, this section contains the `CREATE_SECINIT_TABLE` macro contribution. This macro contribution is the generation of a table that contains two lists of secinit triplets (see *Section 2.2.2* for details).

The script places the `.data` section LMA at `__IROM` (using the `AT` key word), which is within the `IROM` memory region. The `.data` section VMA remains in the `IDM` memory region.

*Note: The toolset also provides a link script file **sx_valid_embedded.ld** that defines an IROM memory region and performs all necessary LMA locations. An application linked with this script is ready for standalone startup on silicon.*

2.2.2 ind_crt0.S file structure

The file `ind_crt0.S` uses the information in the link script to initialize the `.data` section.

The code in `ind_crt0.S` uses lists of **secinit** triplets to transfer the contents of memory from the LMA to the VMA. The code uses two types of secinit triplets:

- **info_copy** triplets have the pattern {*VMA, LMA, size*} and copy *size* bytes from *LMA* to *VMA*
- **info_clear** triplets have the pattern {*VMA, value, size*} and initialize *size* bytes starting at *VMA* with *value*

The **secinit** table is located through two symbols. These are managed by the linker:

- `__stm_binfo_copy` for **info_copy**
- `__stm_binfo_clear` for **info_clear**

The code responsible for the LMA to VMA copy is located in the `ind_crt0c.S` file and is listed in *Figure 7*.

Figure 7. Code for copying LMA to VMA

```
// Equivalent C source code is in comments...

    // extern struct info_copy {
    //    void * VMA;
    //    void * LMA;
    //    size_t size;
    // } * __stm_binfo_copy, * __stm_einfo_copy;

    // cp = &__stm_binfo_copy;
     makeabs r3, __stm_binfo_copy
     makeabs r4, __stm_einfo_copy

    // while( cp < &__stm_einfo_copy ) {
.while_test:
    cmpge      g0, r3, r4
g0? jr         .while_end

    //    bcopy(cp->VMA,cp->LMA,cp->size);
    lw         r0, @(r3!+4)
    lw         r1, @(r3!+4)
    lw         r2, @(r3!+4)
    cmplt      g0, r2, 1
g0? jrgtudec   g0, r2, .bcopy_end
.bcopy_start:
    lb         r5, @(r1!+1)
    sb         @(r0!+1), r5
g0? jrgtudec   g0, r2, .bcopy_start
.bcopy_end:

    //    cp++;
    // already done with lw above //

    // }
    jr         .while_test
.while_end:
```

The code responsible for clearing the VMA is also located in the ind_crt0c.S file and is listed in *Figure 8*.

Figure 8. Code for clearing VMA

```
 // Equivalent C source code is in comments...

    // extern struct info_clear {
    //    void * VMA;
    //    int    value;
    //    size_t size;
    // } * __stm_binfo_clear, * __stm_einfo_clear;

    // cl = &__stm_binfo_clear;
    makeabs r3, __stm_binfo_clear
    makeabs r4, __stm_einfo_clear

    // while ( Cl < &__stm_einfo_clear ) {
.while_test:
    cmpge      g0, r3, r4
g0? jr         .while_end

    //    memset(cl->VMA,vl->value,cl->size);
    lw         r0, @(r3!+4)
    lw         r1, @(r3!+4)
    lw         r2, @(r3!+4)
    cmplt      g0, r2, 1
g0? jrgtudec   g0, r2, .memset_end
.memset_start:
    sb         @(r0!+1), r1
g0? jrgtudec   g0, r2, .memset_start
.memset_end:

    //    cl++;
    // already done with lw above //

    // }
    jr         .while_test
.while_end:
```

When crt0.s, ind_crt0.S or both have been tuned for your application, build the executable with the following command-line:

```
stxp70cc -mcore=stxp70v4 -Mnostartup -g -o embedded.exe \
embedded.c your_crt0.s your_ind_crt0.S embedded.ld
```

where your_crt0.s and your_ind_crt.s are the customized versions of the two start up files, and embedded.ld is the customized linker file.

In addition to providing the **info_copy** and **info_clear** routines, ind_crt0.S can be customized for embedded applications that never return from the main routine. To do this, add -D __EMBEDDED_END__=1 to the ind_crt0.S file compile line to indicate that the application never returns. (The default for this symbol is 0, indicating that the application does return.) If, when this symbol is set, the application does return to ind_crt0.S, it is

simply placed into an endless loop. Setting __EMBEDDED_END__ to 1 significantly reduces the application's memory footprint.

2.2.3 Checking the LMA and VMA

Use the utility **stxp70-readelf** to check the memory addresses of the newly built `embedded.exe`. *Figure 9* gives an example of the output.

Figure 9. Output from stxp70-readelf

```
$ stxp70-readelf -mcore=stxp70v4 -l -W embedded.exe

Elf file type is EXEC (Executable file)
Entry point 0x400000
There are 8 program headers, starting at offset 64

Program Headers:
  Type           Offset   VirtAddr           PhysAddr           FileSiz  MemSiz   Flg Align
  LOAD           0x001000 0x0000000000400000 0x0000000000400000 0x009b96 0x009b96 R E 0x1000
  LOAD           0x00b000 0x0000000000480000 0x0000000000480000 0x000030 0x000030 RW  0x1000
  LOAD           0x00c000 0x0000000000000000 0x0000000000480030 0x000004 0x000004 RW  0x1000
  LOAD           0x00c210 0x0000000000000210 0x0000000000480034 0x000910 0x000910 RW  0x1000
  LOAD           0x00d010 0x0000000000000010 0x0000000000480040 0x000000 0x000200 RW  0x1000
  LOAD           0x00cb20 0x0000000000000b20 0x0000000000480944 0x000000 0x000004 RW  0x1000
  LOAD           0x00cb28 0x0000000000000b28 0x0000000000480944 0x000570 0x000570 R   0x1000
  LOAD           0x00e000 0x0000000000800000 0x0000000000800000 0x000000 0x004000 RW  0x1000


 Section to Segment mapping:
  Segment Sections...
   00     .startup .thandlers .textlib .text
   01     .secinit
   02     .nullptr
   03     .data
   04     .stack1 .stack2 .stack3 .stack4
   05     .bss
   06     .rodata
   07     .heap
$
```

This report shows that:

- the `.startup .thandlers .textlib .text` section segment starts at address `0x400000`
- the LMA of the `.data` section (section #3) is placed in the IROM memory region at address `0x480034`
- the run-time location (VMA) of the `.data` section is `0x4010` (that is, in the IDM memory region, immediately following the stack defined by the `.stack1` section)
- the **secinit** tables (section #1) are located at `0x480000`, and both LMA and VMA of **secinit** are the same.

This new executable file can be debugged using the simulator. In this case, however, the debugger still initializes the VMA section and leaves the LMA empty. The reason for this is that the LMA content is not related to an ELF section, so the debugger is unable to fill in the corresponding memory areas by itself.

3 Debugging an embedded application

At this point, you have produced an ELF file designed to perform its own initialization; the final step is to check whether or not initialization has been properly achieved.

As stated in *Section 2.1 on page 5*, by default, the STxP70 runner (**sxrun**), and the STxP70 debugger (**sxgdb**) copy ELF sections directly in VMA, which means that an application built to be embedded still runs successfully using the debugger or runner. This situation does not reflect the circumstances under which a purely embedded application is designed to be initialized. Also, in this case, copying bytes from LMA to VMA is deactivated in the startup code. This is done to achieve greater efficiency.

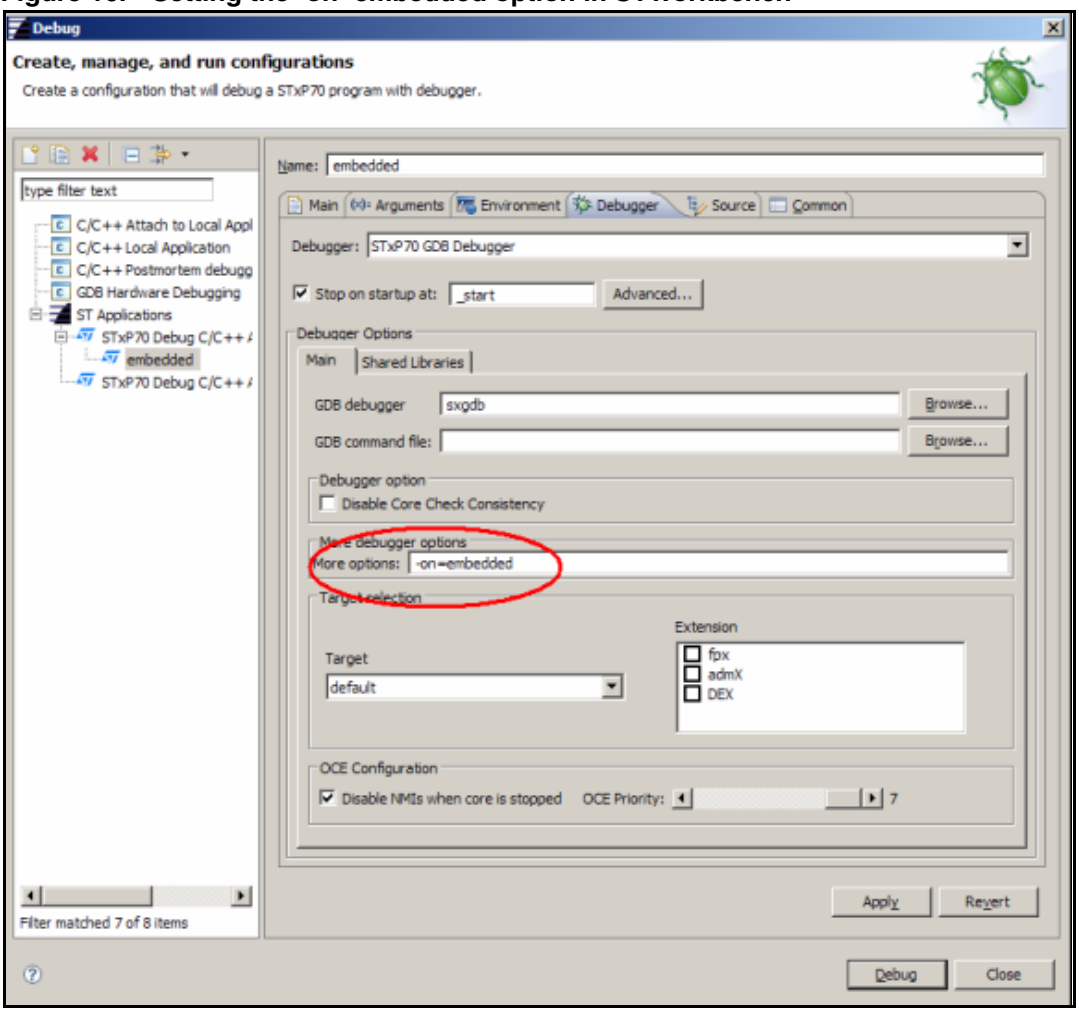
When a debugger or runner is connected to the application, use the command line switch `-on=embedded` to enable the LMA to VMA copying at startup. This switch instructs the debugger to copy the data at LMA and activate the startup code to copy that data to VMA.

The following command line invokes the debugger and enables “embedded” behavior to copy the startup code from the LMA to VMA even though the debugger is connected.

```
sxgdb -on=embedded -target=default embedded.exe
```

In STWorkbench, add the `-on=embedded` option in the **Debugger configuration** tab, **More debugger options** section, **More options** field. See *Figure 10*.

Figure 10. Setting the -on=embedded option in STWorkbench



The debugger stops the the `_start` label. Inspecting the memory at variable location `A` shows that the memory content is 0, that is, the variable is not initialized.

Set a breakpoint on `main` and then run the application. The memory at variable location `A` now contains an array with values from 0 to 9.

To examine this process in more detail, restart the application at set a breakpoint at `__stm_ind_crt0`. This symbol is the entry point of the `ind_crt0.S` file. Stepping through the code in this file demonstrates the **info_copy** and **info_clear** copying operations.

4 Creating binary executable image in formats other than ELF

The toolset provides the utility **stxp70-objcopy** for converting files from ELF format to other formats such as S-REC, I-HEX or BIN.

- S-REC: Motorola S-Record format. Files in this format usually have the extension `.s19`. Use the following command line to convert ELF format to S-REC format:

  ```
  stxp70-objcopy -O srec embedded.exe embedded.s19
  ```

- I-HEX: Intel Hex format. Files in this format usually have the extension `.hex`. Use the following command line to convert ELF format to I-HEX format:

  ```
  stxp70-objcopy -O ihex embedded.exe embedded.hex
  ```

- BIN: Binary format. Files in this format usually have the extension `.bin`. Use the following command line to convert ELF format to BIN format:

  ```
  stxp70-objcopy -O bin embedded.exe embedded.bin
  ```

5 Putting all the elements together

To embed a software application:

- customize the link file in case the application needs to initialize itself (see *Section 2.2.1 on page 8*)
- customize the `crt0.s` file so that the processor is properly initialized (see *Section 2.2.2 on page 9*)
- create the final image of the overall application, using S-REC, IHEX or BIN format (see *Chapter 4 on page 15*)

If the application running on the STxP70 processor must be downloaded from an host processor (for example, an ARM processor), it is possible that some of these file manipulations cannot be carried out. To overcome this limitation, create a loader on the host processor that copies all the ELF sections of the STxP70 application to their correct destinations. This makes the software running on the host a little more complex, but avoids the need to modify the link script file.

The user is responsible for deciding which method to use to build the application.

6 Acknowledgements

Intel® is a trademark of Intel Corporation in the U.S. and other countries.

MOTOROLA® is a registered trademark of Motorola Inc in the U.S.

ARM® is a registered trademark of ARM Limited in the EU and other countries.

7 Revision history

Table 1. Document revision history

Date	Revision	Changes
07-Oct-2009	A	Initial release.
31-Jan-2013	2	Update for Toolset 2012.2