PROGRAMMING AND COMPILING BOOTABLE CODE FOR THE LH77790

Scott Robertson, Senior Software Engineer

INTRODUCTION

The ARM Software Development Toolkit (ARMTools) supports a couple different methods for producing bootable code images. Some of these methods are fully supported by the EmbeddedICE or Angel Debug Monitor, others do not provide full feature debugging.

The LH77790 'system-on-chip' microcontroller has a very powerful memory management capability, allowing a couple different ways to change which memory code is executing from. This application note will describe the different ways to create bootable code using the ARMTools, how to initialize the LH77790 (790 for short) memory map, and how to move memory regions and change which memory code is executing from.

START-UP CODE FOR BASIC 790 INITIALIZATION

There are two ways to generate a bootable program using the ARMTools. The easier method, which only works for assembly language programs, is to let ARMTools map your program code itself and set-up its own jump to the start of your code. This method works well for programming assembly code for bootable EPROMs, but it does not allow for hardcoded interrupt vectors or C code. The other more involved method accommodates C code and hardcoded vectors, so code could remain in EPROM or FLASH if desired. The following discussion on how to initialize the memory management registers is common to both methods for generating bootable code.

790 Memory Initialization

As described in Chapter 5 'Memory and Peripheral Interface' in the 790 User's Guide, there are a number of registers which must be programmed to take advantage of the built-in chip enable signals. For non-DRAM memory and peripherals, there are four types of registers which must be initialized for each region of memory: a Bank Control Register, a Segment Descriptor Register, a Start Register, and a Stop Register. The order of programming these registers is important due to the way the 790 determines if a memory address is mapped to a chip enable. Please refer to chapter 5 in the 790 User's Guide for a full discussion on the registers and how the 790 uses them, as well as for using DRAM. The examples presented here only use SRAM and ROM.

Upon power-up/reset, the 790 starts fetching instructions from location 0x0 (x8 or x16 memory interface is determined by the state of the Byte Boot pin on boot-up). Memory configuration is typically programmed very early in the boot-up code so that read/write memory can be used (boot-up memory is typically not read/write). The following assembly code sample can be assembled, burned into an EPROM, and run on the 790 Evaluation Board. To do so, the linker options must be set for a base at 0x0, as stated in the next section. The code configures a 16-bit wide 512KB SRAM memory region for Supervisor and User read/write privileges, one wait state, non-cacheable, using chip enable 4 (the chip enable used on the 790 Evaluation Board for accessing external SRAM). Any aspect of this code can be easily modified for specific applications - please refer to the 790 User's Guide for bit definitions. After configuring external memory, it initializes part of the Parallel Peripheral Interface (PPI) and sits in a loop blinking the LED on the 790 Evaluation Board.

```
    AREA        TopTest, CODE          ; name this block of code, as required by ARM Tools.

; DEFINES
SRAM_START1         *  &00600000
SRAM_END1           *  SRAM_START1 + &00080000
WAIT_VALUE          *  &F000

    ENTRY                              ; Start of code declaration required by ARM Tools.

; Code boots from EPROM at 0x0, using the default segment.
;   The following code fragment configures SRAM at 0x00600000 - 0x00680000.
;    It uses Bank Configuration Reg 2, Segment Descriptor Reg 2, Start 2, and Stop 2,
;      though others could be used as long as care is taken regarding the segment that code
;      is executing from.
;    SDR2 is programmed to select BCR2, though any bank other than the default could be used.

init_sram           ; Set up External SRAM for
                    ;  location 0x00600000 - 0x00680000,
                    ;  using START2,STOP2,SDR2,BCR2
   LDR    r0,=0xffffa108          ; Addr of Bank Cfg Reg 2
   LDR    r1,=0x00009300          ; 16-bit I/F, use CE4 for Hi&Low byte
   STR    r1,[r0]                 ; Write value to BCR2

   LDR    r0,=0xffffa048          ; Addr of Segment Descriptor Reg 2
   LDR    r1,=0x00007804          ; S/U R/W,non-cacheable,32-bit mode,Bank2
   STR    r1,[r0]                 ; Write value to SDR2 Reg

   LDR    r0,=0xffffa008          ; Addr of START2 reg.
   LDR    r1,=SRAM_START1         ; Start addr of SRAM
   STR    r1,[r0]                 ; Write start addr to START2 Reg

   LDR    r0,=0xffffa028          ; Addr of STOP2 reg.
   LDR    r1,=SRAM_END1           ; Start end addr of SRAM
   STR    r1,[r0]                 ; Write start addr to START2 Reg

; ppi_reg_init                     ; Setup for blinking the LED
   MOV    r11,#0x80               ; Value to init PPI to all outputs
    LDR    r12,=0xffff1c00         ; PPI Base Addr
   STR    r11,[r12,#&C]           ; Set all ports to outputs

wait4ever
   LDR    r8,=WAIT_VALUE          ; wait value
   MOV    r11,#&00                ; Set up to turn LED ON
   STR    r11,[r12,#8]            ; PPI port C=> 00000000
wait4ever2
   SUBS    r8,r8,#1               ; Decrement & set flags
   BNE    wait4ever2              ; loop to wait #1.
   MOV    r11,#&FF                ; Set up to turn LED OFF
   STR    r11,[r12,#8]            ; PPI port C=> 11111111
   LDR    r8,=WAIT_VALUE          ; wait value
wait4ever3
   SUBS    r8,r8,#1               ; Decrement & set flags
   BNE    wait4ever3              ; loop to wait #2.
   B      wait4ever               ; Branch Always - endless loop

    END                            ; Declare the end of the program.
```

Bootable Assembly Code

ARMTools, by default, locates program code starting at address 0x8000 (8000 Hex). The actual program code starts at offset 0x80, since the tools add a little overhead that is executed prior to the program code. In order to generate bootable *assembly* code, which must support the initial processor fetches from 0x0, the following linker option can be added:

-RO-base 0x0

This will put the base of Read Only memory at 0x0, so the ARMTools will place code starting at 0x0 instead of 0x8000 (user code still starts at offset 0x80). To add this linker option in the Windows-based ARMTools (the ARM Project Manager), select 'Options' from the top menu bar, then select 'Linker' from the pull down menu, type in the desired options, click 'OK', and build the code.

Unless otherwise specified, the output of the above process will be an executable file with the same file name as the source code but with no extension. This file can be burned into an EPROM and run on the LH77790 Evaluation Board. The code will run from the default memory segment and will have to initialize any RAM or peripherals it wants to use.

Bootable Code and 'C'

Generating bootable code is described in ARM's 'ARM Software Development Toolkit Programming Techniques' book in Chapter 9 'Writing Code for ROM'. This book is supplied with ARMTools. The following example is based on that chapter. It expects ROM to reside at address 0x0 upon power-on/reset.

The assembly code fragment below initializes parts of the 790 from power-on/reset and sets up for executing code written in C. The routine sets up the necessary definitions for compiled C code and at the end branches to a C program (not included here). Since the keyword 'Entry' in the assembly code defines the start of code execution, a name other than 'main' should be used in the C code. The assembly code fragment below uses the function name 'C_Entry' for the entry point in C code. The following paragraphs will describe the operation of the assembly below.

Upon power-on/reset, the first instruction fetched is the reset vector, which branches to the Reset_Handler. The intervening instructions map to the ARM's interrupt vectors and are at the correct location for programming a ROM (provided the correct linker options are specified, which will be detailed below). The other interrupt handlers are also placed prior to the Reset_Handler so that code execution bypasses them during a reset. Users can add their own program code for the Undefined, SWI, Prefetch, Abort, IRQ, and FIQ interrupt handlers (they are currently just infinite loops).

The Reset_Handler first ensures that interrupts are disabled and sets up an interrupt stack at the top of RW memory, then sets up a supervisor stack immediately under it. Note that the appropriate mode must be entered to initialize its stack pointer. Next the 790 internal registers are configured for accessing memory. This example programs the chip enables as they are used on the Evaluation Board. EPROM is assigned to a segment rather than leaving it in the default segment, and the default segment is mapped to the unused chip enable 5. This way CE5 will go active if the code tries to access an undefined memory region. SRAM and DRAM regions are then configured, again in accordance with the Evaluation Board wiring for the chip enables.

Finally, interrupts are enabled, memory is initialized, the cache is enabled (this may not be desirable for early debugging of code since cache bus cycles will not be seen externally), and the code will branch to the user's 'C_Entry' routine in C code.

As mentioned earlier, certain assembler, compiler, and linker options must be set in the ARM Project Manager so that the executable code produced will all work together and start at address 0x0. Options can be set by selecting 'Options' on the top menu bar in the ARM Project Manager, and then selecting Assembler, Compiler, and Linker as desired to set options for each. The 'Project Options' should be set to Little Endian, ARM 6/7 Target Processor, and ARMCC/ARMASM. The following options will work with the above assembly boot code and user supplied C code:

Assembler Options:

-apcs 3/noswst

Compiler Options:

-list -fc -apcs 3/noswst/nofp

Linker Options:
-info sizes -LIST graphics.lst -Xref -Symbols - -o graphics -Bin -RO-base 0 -RW-base 0x00600000 -First init_gra.o(Init) -Remove -NoZeroPad -Map

The -Bin option specifies a binary output file instead of an ARM Image Format. This code is intended for burning in an EPROM, not running under the ARMulator. Please refer to the ARM manuals for explanations of the above options. This is one example of how to program, compile, and link bootable C code, others may exist.

```
; The AREA must have the attribute READONLY, otherwise the linker will not place it in ROM.
;
; The AREA must have the attribute CODE, otherwise the assembler will not
; let us put any code in this AREA
;
; Note the '|' character is used to surround any symbols which contain
; non standard characters like '!'.

        AREA   Init, CODE, READONLY
      OPT    1
      OPT    64
      OPT    256
      OPT    1024
      OPT    4096

; Now some standard definitions...

Mode_IRQ      EQU    0x12
Mode_SVC      EQU    0x13

I_Bit         EQU    0x80
F_Bit         EQU    0x40

SWI_Exit      EQU    0x11

; Locations of various things in our memory system

RAM_Base      EQU    0x600000          ; 512k RAM at this base
RAM_Limit     EQU    0x680000

IRQ_Stack     EQU    RAM_Limit         ; 1K IRQ stack at top of memory
SVC_Stack     EQU    RAM_Limit-1024    ; followed by SVC stack

; 790 EQUATES
ROM_BASE          *  &00000000
ROM_END           *  &00100000
INT_SRAM          *  &60000000
SRAM_START        *  &00600000
SRAM_END          *  SRAM_START + &00080000
SRAM_TEST         *  &00650000
DRAM1START        *  &00700000
DRAM1END          *  DRAM1START + &00100000
DRAM2START        *  &00800000
DRAM2END          *  DRAM2START + &00100000
CACHE_CTRL        *  &FFFFA400
WAIT_VALUE        *  &5000

; --- Set the entry point
    ENTRY

; --- Setup interrupt / exception vectors
; The ROM is expected to be at address 0, so this is just a sequence of branches
    B     Reset_Handler         ; This branch is taken at power-up/reset.
    B     Undefined_Handler
    B     SWI_Handler
    B     Prefetch_Handler
    B     Abort_Handler
```

```
    NOP                             ; Reserved vector
    B     IRQ_Handler
    B     FIQ_Handler

; The following handlers do not do anything useful in this example.
;
Undefined_Handler
    B     Undefined_Handler
SWI_Handler
    B     SWI_Handler
Prefetch_Handler
    B     Prefetch_Handler
Abort_Handler
    B     Abort_Handler
IRQ_Handler
    B     IRQ_Handler
FIQ_Handler
    B     FIQ_Handler

; ************** The RESET entry point **************
Reset_Handler

; --- Initialise stack pointer registers
; Enter IRQ mode and set up the IRQ stack pointer
    MOV         R0, #Mode_IRQ:OR:I_Bit:OR:F_Bit       ; No interrupts
    MSR         CPSR, R0
    LDR         R13, =IRQ_Stack

; Set up other stack pointers if necessary
    ; ...

; Set up the SVC stack pointer last and return to SVC mode
    MOV         R0, #Mode_SVC:OR:I_Bit:OR:F_Bit       ; No interrupts
    MSR         CPSR, R0
    LDR         R13, =SVC_Stack

; --- Initialise memory system
; Do 790 initializations.
 ; NOTES:
 ;   -  r11, r12 are used in this example for controlling the LED.
 ;   -  When programming Memory Configurations, first set
 ;         BCR & SDR, then START, and do STOP last.

ppi_reg_init                        ; Setup 1st for calls that blink the LED
    MOV     r11,#0x80               ; Value to init PPI to all outputs
    LDR     r12,=0xffff1c00         ; PPI Base Addr
    STR      r11,[r12,#&C]          ; Set all ports to outputs

init_rom                            ; Set up ROM for
                                    ;  location 0x00000000 - 0x00100000,
                                    ;  using START1,STOP1,SDR1,BCR1
   LDR     r0,=0xffffa104           ; Addr of Bank Cfg Reg 1
   LDR     r1,=0x00001003           ; 8-bit I/F, 1 wait, CE0 for Hi&Low byte
   STR     r1,[r0]                  ; Write value to BCR1

   LDR     r0,=0xffffa044           ; Addr of Segment Descriptor Reg 1
   LDR     r1,=0x00007C02           ; S/U R/W,cacheable,32-bit mode,Bank1
   STR     r1,[r0]                  ; Write value to SDR1 Reg
```

```
   LDR    r0,=0xffffa004          ; Addr of START1 reg.
   LDR    r1,=ROM_BASE            ; Start addr of ROM
   STR    r1,[r0]                 ; Write start addr to START0 Reg

   LDR    r0,=0xffffa024          ; Addr of STOP1 reg.
   LDR    r1,=ROM_END             ; End addr of ROM
   STR    r1,[r0]                 ; Write start addr to START0 Reg

chg_default_seg
   LDR    r0,=0xffffa100          ; Addr of Bank Cfg Reg 0
   LDR    r1,=0x00008C00          ; 16-bit I/F, use CE5 for Hi&Low byte
   STR    r1,[r0]                 ; Write value to BCR0

   LDR    r0,=0xffffa060          ; Addr of Segment Descriptor Reg 8
   LDR    r1,=0x00007801          ; S/U R/W,non-cache,32-bit mode,Bank0
   STR    r1,[r0]                 ; Write value to SDR1 Reg
   ; SDR8, the default seg, does not have a START and STOP register.

init_sram                          ; Set up External SRAM for
                                   ;  location 0x00600000 - 0x00680000,
                                   ;  using START2,STOP2,SDR2,BCR1
   LDR    r0,=0xffffa108          ; Addr of Bank Cfg Reg 2
   LDR    r1,=0x00009300          ; 16-bit I/F, use CE4 for Hi&Low byte
   STR    r1,[r0]                 ; Write value to BCR1

   LDR    r0,=0xffffa048          ; Addr of Segment Descriptor Reg 2
   LDR    r1,=0x00007804          ; S/U R/W,non-cache,32-bit mode,Bank2
   STR    r1,[r0]                 ; Write value to SDR0 Reg

   LDR    r0,=0xffffa008          ; Addr of START2 reg.
   LDR    r1,=SRAM_START          ; Start addr of SRAM
   STR    r1,[r0]                 ; Write start addr to START2 Reg

   LDR    r0,=0xffffa028          ; Addr of STOP2 reg.
   LDR    r1,=SRAM_END            ; Start end addr of SRAM
   STR    r1,[r0]                 ; Write start addr to START2 Reg

init_dram                          ; Set up CE2 External DRAM for location
                                   ;  0x00700000 - 0x00800000, (1MB)
                                   ;  using START3,STOP3,SDR6,BCR6
;   LDR    r0,=0xffffa118          ; Addr of Bank Cfg Reg 6a
;   LDR    r1,=0x00009030          ; 8-bit I/F, use CE2 for Hi&Low byte
;   STR    r1,[r0]                 ; Write value to BCR6a
;
;   LDR    r0,=0xffffa120          ; Addr of Bank Cfg Reg 6b
;   LDR    r1,=0x00000013          ; Active Refresh, 1meg, PageMode
;   STR    r1,[r0]                 ; Write value to BCR6a
;
;   LDR    r0,=0xffffa04C          ; Addr of Segment Descriptor Reg 3
;   LDR    r1,=0x00007840          ; S/U R/W,non-cache,32-bit mode,Bank6
;   STR    r1,[r0]                 ; Write value to SDR0 Reg
;
;   LDR    r0,=0xffffa00C          ; Addr of START3 reg.
;   LDR    r1,=DRAM1START          ; Start addr of SRAM
;   STR    r1,[r0]                 ; Write start addr to START3 Reg
;
;   LDR    r0,=0xffffa02C          ; Addr of STOP3 reg.
```

```
;    LDR     r1,=DRAM1END        ; Start end addr of SRAM
 ;   STR     r1,[r0]             ; Write start addr to START3 Reg
 ;
 ;                   ; Set up CE3 External DRAM for location
 ;                   ;  0x00800000 - 0x00900000, (1MB)
 ;                   ;  using START4,STOP4,SDR7,BCR7
 ;   LDR     r0,=0xffffa11C      ; Addr of Bank Cfg Reg 7a
 ;   LDR     r1,=0x000090C0      ; 8-bit I/F, use CE3 for Hi&Low byte
 ;   STR     r1,[r0]             ; Write value to BCR7a
 ;
 ;   LDR     r0,=0xffffa124      ; Addr of Bank Cfg Reg 7b
 ;   LDR     r1,=0x00000013      ; Active Refresh, 1meg, PageMode
 ;   STR     r1,[r0]             ; Write value to BCR6a
 ;
 ;   LDR     r0,=0xffffa04C      ; Addr of Segment Descriptor Reg 4
 ;   LDR     r1,=0x00007880      ; S/U R/W,non-cache,32-bit mode,Bank7
  ;  STR     r1,[r0]             ; Write value to SDR0 Reg
  ;
  ;  LDR     r0,=0xffffa010      ; Addr of START4 reg.
  ;  LDR     r1,=DRAM2START      ; Start addr of SRAM
  ;  STR     r1,[r0]             ; Write start addr to START4 Reg
;
;   LDR     r0,=0xffffa030      ; Addr of STOP4 reg.
;   LDR     r1,=DRAM2END        ; Start end addr of SRAM
;   STR     r1,[r0]             ; Write start addr to START4 Reg

dram                            ; Test DRAM
;   LDR     r3,=DRAM1START      ; Set r3 to point to DRAM1
;   LDR     r4,=&12345678       ; Value to write
;   STR     r4,[r3]             ; Write value to DRAM
;   LDR     r5,[r3]             ; Read DRAM
;   ADD     r3,r3,#0x4          ; Increment addrss to next word (0C)
;   STR     r5,[r3]             ; Write value to DRAM
;   SUBS    r4,r4,r5            ; r4=r4-r5 (chk readback)
;
 ;  LDR     r3,=DRAM2START      ; Set r3 to point to DRAM2
 ;  LDR     r4,=&BADDFADE       ; Value to write
 ;  STR     r4,[r3]             ; Write value to DRAM
 ;  LDR     r5,[r3]             ; Read DRAM
 ;  ADD     r3,r3,#0x4          ; Increment addrss to next word (0C)
 ;  STR     r5,[r3]             ; Write value to DRAM
 ;  SUBS    r4,r4,r5            ; r4=r4-r5 (chk readback)


; --- Initialise critical IO devices

; --- Initialise interrupt system variables here
    ; ...

; --- Enable interrupts
; Now safe to enable interrupts, so do this and remain in SVC mode
    MOV         R0, #Mode_SVC:OR:F_Bit      ; Only IRQ enabled
    MSR         CPSR, R0

; --- Initialise memory required by C code

    IMPORT      |Image$$RO$$Limit|          ; End of ROM code (=start of ROM data)
    IMPORT      |Image$$RW$$Base|          ; Base of RAM to initialise
```

```
    IMPORT      |Image$$ZI$$Base|             ; Base and limit of area
    IMPORT      |Image$$ZI$$Limit|            ; to zero initialise

    LDR         r0, =|Image$$RO$$Limit|       ; Get pointer to ROM data
    LDR         r1, =|Image$$RW$$Base|        ; and RAM copy
    LDR         r3, =|Image$$ZI$$Base|        ; Zero init base => top of initialised data
    CMP         r0, r1                        ; Check that they are different
    BEQ         %1
0   CMP         r1, r3                        ; Copy init data
    LDRCC       r2, [r0], #4
    STRCC       r2, [r1], #4
    BCC         %0
1   LDR         r1, =|Image$$ZI$$Limit| ; Top of zero init segment
    MOV         r2, #0
2   CMP         r3, r1                        ; Zero init
    STRCC       r2, [r3], #4
    BCC         %2


; --- Now we enter the C code

    IMPORT      C_Entry

; Enable Cache
cache_enable
    LDR         r0,=CACHE_CTRL                ; Addr of Cache Control Reg.
    LDR         r1,=&01                       ; Value to write - Enable Cache
    STR         r1,[r0]                       ; Write value to Cache Ctrl Reg.

    B           C_Entry
; The application is not expected to return.

        END
```