Using elftosb to generate HAB enabled boot streams

Introduction

HAB4 was added to the boot ROM on the MX28 chip. By default, with no OTP bits burned, HAB is enabled in Open mode. All that is required to boot with the chip in this configuration is the addition of an IVT (Image Vector Table) to your boot stream. Because the MX28 is intended to always have HAB enabled, adding the IVT must be very easy.

elftosb versions 2.5 and above can output the boot commands necessary to use HAB4 on the MX28. Versions 2.6 and above add the ability to generate the IVT solely from a description provided in the boot descriptor (.bd) file. This note will describe how to add the IVT and make your application bootable with HAB enabled in Open mode. The process to add support for HAB in Closed mode is much more in-depth and will be covered elsewhere.

This note assumes you understand elftosb terminology and boot descriptor file syntax.

The IVT

The IVT is a small, 32-byte structure that the ROM passes to the HAB4 routines every time a call is made into user code. The most important piece of data it contains is the entry point address.

Field	elftosb name	Description
Header	n/a	Tag-Length-Version field
Entry point	`entry`	Start address for the image in cold boot
Boot data	`boot_data`	Address of arbitrary information for the bootloader (ROM, etc).
IVT	`self`	Pointer to self.
DCD	`dcd`	DCD pointer.
CSF	`csf`	CSF pointer.

The "elftosb name" column in the above table is the keyword used in the elftosb IVT syntax, described in detail below.

The only fields required out of those listed above when HAB is in Open mode are the header, entry point, and self pointer.

IVT syntax

IVT structure generation in elftosb uses a new syntax to specify the IVT contents.

```
ivt (
    entry = 0,
    dcd = 0,
    boot_data = 0,
    self = 0,
    csf = 0
)
```

All fields of the IVT are optional. Each field can be assigned to any constant expression, including symbol references like elffile:_start, etc.

An IVT definition can be used anywhere you can specify a data source. Most usefully, in a load command.

```
load ivt (entry=myfile:_start) > 0x100;
```

The IVT definition is fairly smart about the self pointer. If the self pointer is not specified, it will be assigned the value of the load address. So in the above example, self would be set to 0x100 automatically. If the self pointer is explicitly set in the definition, then the IVT has a natural address (equal to the self pointer) and can be loaded without having to give a target address. elftosb will report an error if you try to load an IVT without providing either an explicit value for the self pointer or a load target address. The example below is equivalent to the one above:

```
load ivt (entry=myfile:_start, self=0x100);
```

To invoke the IVT and call the entry point address, you need to use the hab call/jump command. To invoke the IVT loaded by one of the above examples, you could use:

```
hab jump 0x100;
```

The address specified in the call/jump statement is not the entry point itself, but the address of the IVT structure in memory.

One argument may be specified in the call/jump statement, just as with normal call/jump statements. For instance, passing 1 to the entry point:

```
hab jump 0x100 (1);
```

The only difficult part about adding the IVT is finding a spot in memory for 32 bytes. In the future, it should be possible to have elftosb help with this. But for now it is entirely manual. You just have to use objdump or similar to examine where things are loaded in memory, and then search for a small hole. (And then hopefully you don't change your app in a way that puts something important where you put the IVT!)

 One important note is that you must set the chip family to MX28 on the elftosb command line. Otherwise elftosb will default to the MX23 family, which doesn't support HAB, thus causing errors when you use HAB related syntax in the .bd file. Do this by adding `--chip-family=mx28` to the command line (the family name is not case sensitive).

Converting an existing .bd file

This section uses the conversion of the uboot .bd file as an example. This is the imx-bootlets-src-10.05.02/uboot.db (all .bd files in the BSP have the incorrect extension of .db!) file from the Linux BSP. The original uboot.bd file looks like this:

```
// STMP378x ROM command script to load and run U-Boot

options {
    // turn on command printing by the rom
flags = 1;
}

sources {
    power_prep="./power_prep/power_prep";
    sdram_prep="./boot_prep/boot_prep";
    image="/home/creed/projects/mx28/ltib/rootfs/boot/u-boot";
}

section (0) {
    //-------------------------------------------------------
// Power Supply initialization
//-------------------------------------------------------

    load power_prep;
    call power_prep:_start;

    //-------------------------------------------------------
// SDRAM initialization
//-------------------------------------------------------

    load sdram_prep;
    call sdram_prep:_start;

    //-------------------------------------------------------
//  Load and call u_boot - ELF ARM image
//-------------------------------------------------------

    load image;
    call image:_start;
}
```

For each call or jump statement in your .bd file, you need to add an IVT that contains the entry point address that would normally be specified in the call/jump command itself. Instead, the call/jump specifies the IVT address.

The first step is to figure out where you're going to put the IVT. Examine the map file produced during the build and find an unused 32 byte region somewhere in memory. If you don't have a map file handy, you can use objdump or even elftosb or sbtool output. Remember that SDRAM probably is not available until late in the boot process. In the case of u-boot, SDRAM only becomes available after sdram_prep runs. So the IVT for sdram_prep and the bootlet that runs before it must reside in OCRAM. Fortunately, the IVT only has to exist in memory from the time it is loaded until the call/jump command starts being processed by the ROM. This means you don't have to worry about your application overwriting the IVT once it starts executing.

To convert u-boot, I just used the elftosb output. When you look at the sequence of load commands, it's pretty obvious that most of OCRAM is unused by either the 2 bootlets or u-boot itself. The bootlets load into low OCRAM, not more than about 10KB. U-boot loads entirely into SDRAM. So I picked the address 0x8000, in the middle of OCRAM. Be aware that the top 16KB of OCRAM is a keep out area used by the ROM for as long as it is running.

Here's a close up example from the uboot .bd file. The first bootlet to run is power_prep. The original section of the .bd file for power_prep:

```
load power_prep;
call power_prep;
```

Pretty straightforward. This is changed to the following to add HAB support:

```
load power_prep;
load ivt (entry = power_prep:_start) > 0x8000;
hab call 0x8000;
```

As you can see, an IVT with the entry point set to the `_start` symbol of power_prep is loaded at 0x8000. Then the call command is changed into a HAB call (by adding the 'hab' keyword) that points at the IVT we just loaded. You can define a constant for the IVT load address, as is done in the uboothab.bd shown below.

Note that unlike with the call/jump statement, you must explicitly specify the symbol name when setting the IVT entry point. i.e., `entry=power_prep:_start` and not `entry=power_prep`.

That's all there is to it! Here is the converted uboot.bd.

```
// MX28 ROM command script to load and run U-Boot

options {
    // turn on command printing by the rom
flags = 1;
}

sources {
     power_prep="./power_prep/power_prep";
     sdram_prep="./boot_prep/boot_prep";
     image="/home/creed/projects/mx28/ltib/rootfs/boot/u-boot";
}

constants {
    // Nice place to put the IVT.
IVT_ADDR = 0x8000;
}

section (0) {
    //-------------------------------------------------------
// Power Supply initialization
//-------------------------------------------------------

     load power_prep;
     load ivt (entry = power_prep:_start) > IVT_ADDR;
     hab call IVT_ADDR;

    //-------------------------------------------------------
// SDRAM initialization
//-------------------------------------------------------

     load sdram_prep;
     load ivt (entry = sdram_prep:_start) > IVT_ADDR;
     hab call IVT_ADDR;

    //-------------------------------------------------------
//  Load and call u_boot - ELF ARM image
//-------------------------------------------------------

     load image;
     load ivt (entry = image:_start) > IVT_ADDR;
     hab jump IVT_ADDR;
}
```