Technical Notes on using Analog Devices' DSP components and development tools
Contact our technical support by phone: (800) ANALOG-D or e-mail: dsp.support@analog.com
Or visit our on-line resources http://www.analog.com/dsp and http://www.analog.com/dsp/EZAnswers

Implementing a Boot Manager for ADSP-218x Family DSPs

Contributed by Benno Kusstatscher *September 5, 2002*

Introduction

Conventional system design calls for a single program to boot at power-up. One may implement a more sophisticated approach to build an application comprised of separate programs, each booted into the DSP as needed, under the control of a boot manager. One benefit of such an approach is that the total memory consumption of the application may far exceed the on-chip memory resources provided by the chosen ADSP-218x family DSP.

A boot manager is a piece of control code that determines which program is booted after system reset (e.g. by sampling one of input flag pins). A boot manager may also help to boot a second program after the first one has terminated. The underlying technique remains the same.

If booted by a host processor through the IDMA interface, the ADSP-218x behaves like a slave and the host device has full control to reset and reboot the DSP with different programs any time.

This application note discusses the scenario where the ADSP-218x boots from an 8-bit EPROM using its BDMA capability. All programs are stored in the same EPROM the DSP boots from and need to be managed properly. It is obvious that the VisualDSP++ loader/splitter[1] utility elfspl21.exe plays a crucial role. The following explanations are based on VisualDSP++™ version 3.0 although the basic functionality was introduced with version 2.0.

[1] In this context the utility elfspl21.exe functions as a loader utility, only. The name "Splitter" has historical nature for usage on former ADSP-21xx devices that did not feature BDMA booting yet.

General Bootstrap Procedure

It is not an absolute must that one has an in-depth understanding of the BDMA boot process to get the boot manager up and running. Nevertheless it helps to adapt the procedure described below to slightly different requirements.

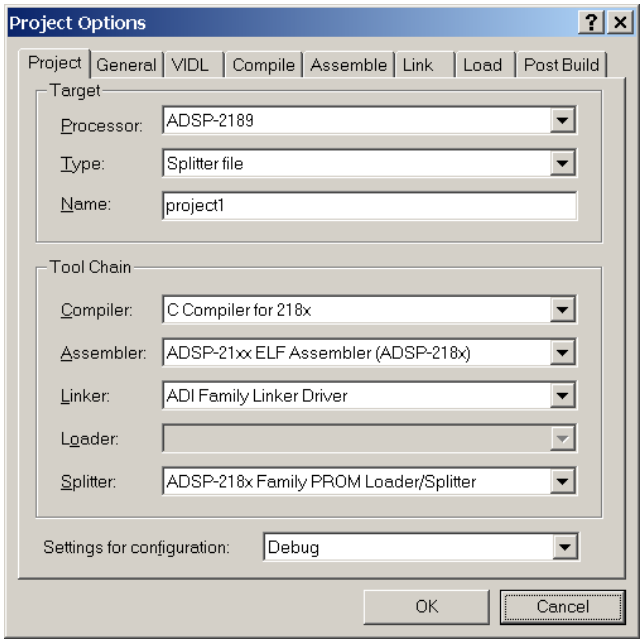

Figure 1: VisualDSP++ project property page

In order to boot any of the ADSP-218x family DSPs from an 8-bit parallel EPROM or Flash device, the VisualDSP++ Loader/Splitter utility

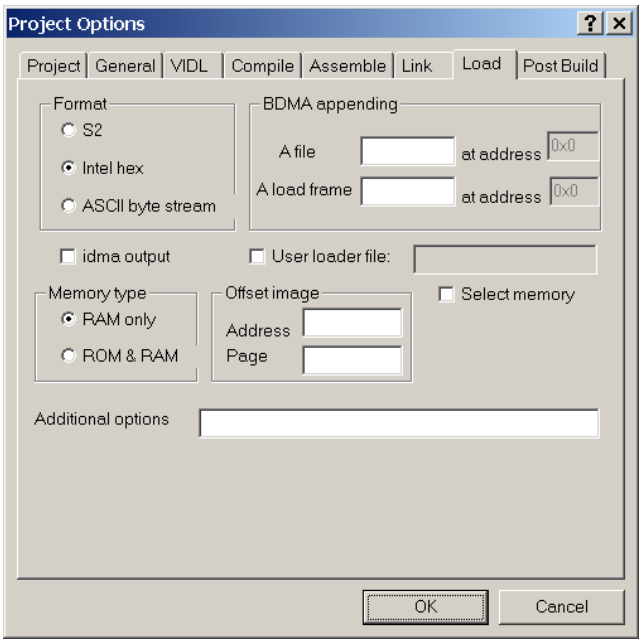

Figure 2: VisualDSP++ load property page

needs to be invoked as shown in Figure 1. The default settings within the load property page (Figure 2) force VisualDSP++ to invoke the splitter utility with the following command line switches:

```
elfspl21 project1.dxe project1
         -218x -loader -i
```

In response to this command, the elfspl21.exe generates a boot stream from the linker's executable file project1.dxe that meets the ELF/DWARF-2 standard; the output file will be project1.bnm, encoded in the Intel Hex format. Let us discuss the BDMA boot feature of the ADSP-218x family first in order to understand the structure of the boot stream.

With BDMA booting enabled, after reset the ADSP-218x DSPs do not immediately start to fetch and execute instructions. First, the core is halted and the BDMA engine automatically loads the first 96 bytes from the BDMA space and copies them into the internal SRAM at PM address 0x0000. Once the BDMA transfer has finished the DSP core starts to execute the 32 instructions built by these 96 bytes.

Of course, most applications are much larger than 32 instructions. 32 instructions are not sufficient to boot real applications, either. Therefore, this first piece of program is used to implement a bootstrap scenario only. We shall refer to this as the "preloader". Listing 1 shows the preloader that is used by default.

```
/* standard preloader (32 instructions)                    address  opcodes          */
ax0 = 0x0060;  dm(0x3fe2) = ax0;          /* BEAD     */ /* 0x0000: 400600 93FE20      */
ax0 = 0x0020;  dm(0x3fe1) = ax0;          /* BIAD     */ /* 0x0002: 400200 93FE10      */
ax0 = 0x0000;  dm(0x3fe3) = ax0;          /* CTRL     */ /* 0x0004: 400000 93FE30      */
ax0 = 0x0087;  dm(0x3fe4) = ax0;          /* BWCOUNT  */ /* 0x0006: 400870 93FE40      */
ifc = 0x0008;  nop;                       /* BDMA IRQ */ /* 0x0008: 3C008C 000000      */
imask = 0x0008;                                          /* 0x000A: 3C0083             */
idle;                                                    /* 0x000B: 028000             */
jump 0x0020;  nop;  nop;  nop;                           /* 0x000C: 18020F 000000  ... */
nop;  nop;  nop;  nop;                                   /* 0x0010: 000000 000000  ... */
nop;  nop;  nop;  nop;                                   /* 0x0014: 000000 000000  ... */
nop;  nop;  nop;  nop;                                   /* 0x0018: 000000 000000  ... */
rti;  nop;  nop;  nop;                                   /* 0x001C: 0A001F 000000  ... */
```

Listing 1: Default preloader used by elfspl21.exe

Basically, the preloader just sets up another BDMA sequence that loads additional loader code from byte address 0x0060 to PM address 0x0020. An IDLE instruction stops the program execution until the BDMA has finished. The unmasked BDMA Interrupt brings the core out of the idle state afterward. Please note the RTI instruction at BDMA interrupt vector address 0x001C.

Once the BDMA transfer has finished, the code execution continues at the address subsequent to the IDLE instruction. The program jumps to 0x0020, where the recently loaded instructions have been placed. This is the first PM location after the BDMA interrupt vector.

This second part of the bootstrap loader is conventionally called "page loader". It boots the final application data. The elfspl21.exe tool generates this page loader content dynamically.

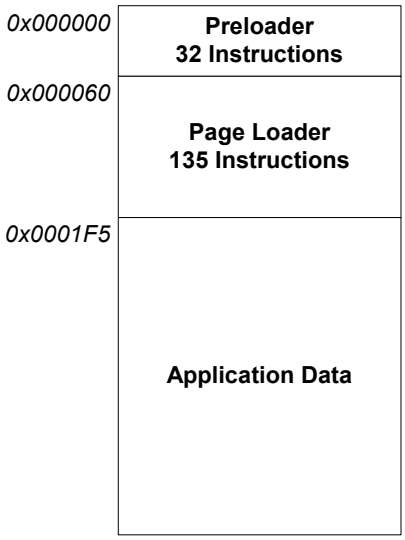

Figure 3: EPROM memory map example

Basically, the page loader consists of a set of BDMA setup sequences, one for every hardware page that needs to be booted. Such a setup sequence takes nine instructions; the last one is the IDLE instruction, required to halt program execution. Please note that the RTI instruction at address 0x001C is still required.

The final BDMA transfer loads the fixed PM page (PM 0x0000 to 0x1FFF). It overwrites both, preloader and page loader. Therefore the final BDMA setup does not use the IDLE instruction, but sets the BCR (BDMA Context Reset) bit of the BDMA control register. This forces the DSP to halt the program execution until the BDMA transfer has finished and to jump to PM address 0x0000 afterward. Then, it immediately starts to execute the booted application.

VisualDSP++™ 2.0 versus 3.0

Beside minor bug fixes the elfspl21.exe utility of VisualDSP++ 3.0 is the same as the one of the 2.0 release – with one exception: VisualDSP++ 3.0 supports booting of off-chip DM and PM overlay pages, while VisualDSP++ 2.0 does not. This new feature results in some impacts we need to be aware of.

VisualDSP++ 2.0 fixes the number of page loader instructions to 135, providing space for up to 15 BDMA setup sequences. This is sufficient to boot all on-chip overlay pages of ADSP-2188 devices. Unused page loader instructions are simply filled with NOP instructions. The preloader's BWCOUNT value is always 135 (=87h) as shown in Listing 1 and Figure 3.

VisualDSP++ 3.0 can boot off-chip SRAM. Since the hardware does not support BDMA transfers directly to off-chip DM and PM memories, additional instructions are required to copy boot data from intermediate internal storage to off-chip overlay memory. Therefore the page loader of VisualDSP++ 3.0 may exceed these 135 instructions. The number of instructions may vary and the default preloader's BWCOUNT value is calculated dynamically by elfspl21.exe. If the –uload switch is used to replace default preloader, the user has no access to the real length of the page loader. The BWCOUNT register needs to be filled with a constant value that mirrors the maximal possible page loader length. This worst-case length depends on the application. Actually, the maximal length

generated by the VisualDSP++ 3.0 splitter is 658 instructions if off-chip pages are also booted, but this may change in future releases.

Simulating the boot procedure

The entire boot procedure can be simulated with the VisualDSP++ debugger. Please follow these steps:

- Invoke the integrated VisualDSP++ environment and select an ADSP-218x simulator session
- Set *Settings➔Simulator➔Boot Mode* to "Boot from EPROM"
- Load the generated BNM file by *Settings➔Simulator➔Load ROM File*
- Load the preloader by clicking on *Debug➔Reset*
- Set a breakpoint to address 0x0020 and let the Debugger run (press F5)
- Set another breakpoint at address 0x0000 and press F5 again

Grouping more programs into a single EPROM image

Former splitter versions already featured the command-line switch –bdma file.dxe offset, where offset is any decimal or hexadecimal number (0x….). It allows additional executable files to be included into one EPROM image, but the splitter does not generate extra boot information. Therefore, the VisualDSP++ splitter features an additional switch called –bdmaload file.dxe offset that includes the required boot loader. The only restriction is that the second parameter offset has to be a BDMA page boundary. In other words: it must be a multiple of 0x4000. The complete syntax may look like

```
elfspl21 project1.dxe project1
         -i -218x -loader
         -bdmaload project2.dxe 0x4000
```

Then, the splitter places a second preloader at address 0x4000 that looks just like the first one except that the BMPAGE field within the BDMA control register is set to one this time. The preloader is followed by another page loader and finally the application data.

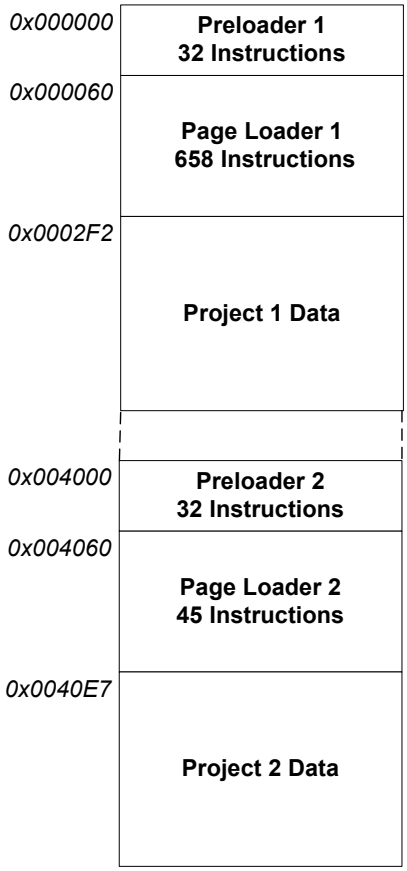

Figure 4: EPROM memory map (example)

The –bdmaload switch can be specified multiple times. For example, to group three projects use

```
elfspl21 project1.dxe project1
         -i -218x -loader
         -bdmaload project2.dxe 0x4000
         -bdmaload project3.dxe 0x8000
```

Within the load property dialog specify the A load frame field to obtain such a command line. If you want to specify a third or even more "load frames" you need to type them into the additional options box, like illustrated in Figure 5.

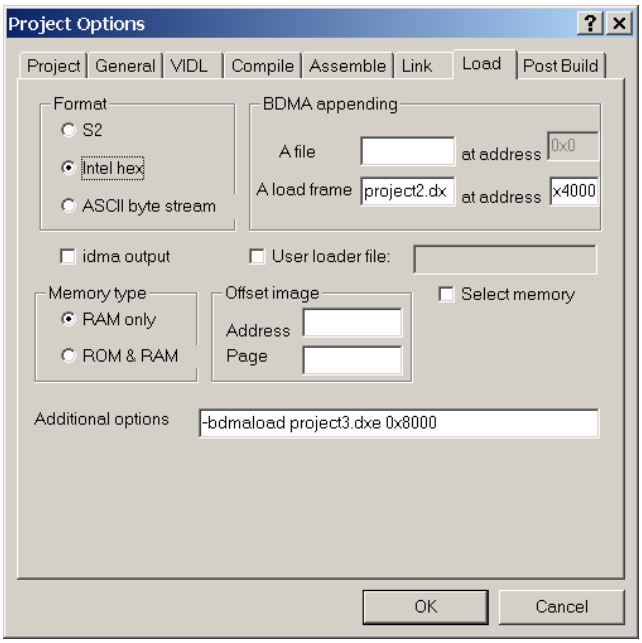

Figure 5: Load frame example settings

Complex Boot Manager as a separate application

The user can easily implement a boot manager as a separate application by taking advantage of the –bdmaload switch. A newly created project may contain all the code required to handle the selective booting. First of all, this boot manager determines the index number of the application to be booted. Regardless whether this information is received on the SPORT or is determined by flag pins, finally the index number will be available and might be stored in the AR register as required by the example below.

A simple relationship between this index number and the corresponding byte address needs to be established. The BEAD register holds the lower 14 bits of the byte address. Since the –bdmaload switch enables offsets multiple to 0x4000 only, the BEAD register is always set to zero. The BMPAGE bit field in the BDMA control register holds the upper 8 bits. Figure 6 shows the BDMA control register. The 8 LSBs are always zero for our purpose.

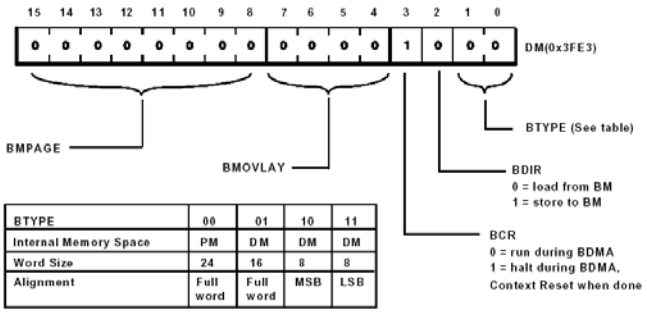

Figure 6: BDMA Control Register

In the simplified case that every single application takes less than 0x4000 bytes and therefore application number 1 starts at byte address 0x4000 and application number 2 at 0x8000 etc. the following procedure can be used.

Make sure that

- The BDMA interrupt vector address 0x001C contains a RTI instruction
- All the hardware stacks are empty
- No interrupts are pending
- The function loadapplication is located at any PM address higher than 0x20

Then, jump to the label loadapplication which loads the preloader of the wanted program and executes it afterward in order to boot the corresponding program.

```
/* AR holds index of the program that needs
 * to be booted. This example assumes that
 * every program occupies less than 16384
 * byte memory to keep the relationship
 * between AR and the start byte address as
 * simple as possible.
 *
 * EPROM map
 * 0x0000: boot manager
 * 0x4000: program 1
 * 0x8000: program 2
 *     ...
 *
 */

loadapplication:


/* clear and unmask BDMA interrupt     */

  ifc = 0x0008;  nop;
  imask = 0x0008;

/* BEAD (lower 14-bit) is 0x0000       */
```

```
  ax0 = 0x0000;  dm(0x3fe2) = ax0;
/* BIAD is PM 0x0000                    */
  ax0 = 0x0000;  dm(0x3fe1) = ax0;
/* BDMA Control (BM Page = AR+1)        */
  ar = ar + 1;
  sr = lshift ar by 8 (lo);
  dm(0x3fe3) = sr0;
/* BWCOUNT is 32 x 24 bit               */
  ax0 = 0x0020;  dm(0x3fe4) = ax0;
  idle;
  jump 0x0000;
```

Listing 2: Load another (next) program

Finally, use the ELF splitter to group the boot manager and the various applications together in a single BNM file.

```
elfspl21 bootman.dxe bootman
         -i -218x -loader
         -bdmaload project1.dxe 0x4000
         -bdmaload project2.dxe 0x8000
```

Of course, the code of Listing 2 may or may not be part of an explicit boot manager. It may also be part of a regular application project and can be called once the program has finished in order to boot the next one. Then the instruction `ar=ar+1;` should be removed from Listing 2.

If individual boot images don't fit into a single BM page

The previous example assumed the individual boot images fit into 0x4000 bytes. In real applications, this is usually not the case.

In case of an ADSP-2185 device with 16k PM words and 16k DM words on-chip a realistic boot image may take six BM pages (0x18000 bytes). Then the elfspl21.exe utility could be invoked by

```
elfspl21 bootman.dxe bootman
         -i -218x -loader
         -bdmaload project1.dxe 0x04000
         -bdmaload project2.dxe 0x1C000
          . . .
```

And the relationship between the program number (stored in AR) and the corresponding BMPAGE value could be calculated by

```
/* AR holds index of the program that needs
 * to be booted.
 *
 * EPROM map
 * 0x000000: boot manager
 * 0x004000: program 1 (BMPAGE =  1)
 * 0x01C000: program 2 (BMPAGE =  7)
 * 0x034000: program 3 (BMPAGE = 13)
 *      ...
 *
 */
  dis m_mode;
  my0 = 0x0300;
  mr0 = 0x0100;
  mr = mr + ar * my0 (SS);
  dm(0x3fe3) = mr0;
```

Listing 3: Alternative AR to Byte Address Relationship

Simple Boot Manager replaces preloader

The procedure described above is very flexible, but there exists an alternative and even simpler scenario. If the boot manager requires a few instructions only (e.g. determining the application to boot by testing an input flag pin) the preloader can be exchanged.

The ELF splitter elfspl21.exe features an additional command line switch –uload objectfile.doj. This forces the splitter to use a user-defined preloader instead of the standard one. This is typically used to speed up the booting by reducing wait-states.

```
elfspl21 project1.dxe project1
         -i -218x -loader
         -bdmaload project2.dxe 0x4000
         -uload myloader.doj
```

Within the load property page you can specify your own preloader file by activating the user loader file check box.

Please note: since the –uload switch expects a DOJ file as input, the user defined preloader must not contain symbols and labels. Labels are resolved by the linker, but DOJs are just assembler output files. The following example illustrates how the FI pin might be checked in order to select either program 0 or program 1.

```
/* example of an user-defined preloader
 * (32 instructions, no labels)
 * Input pin FI selects program to be
 * booted
 */
/* start at address 0x0000              */
.section / pm interrupts;
/* BEAD                                 */
  ax0 = 0x0060;  dm(0x3fe2) = ax0;
/* BIAD                                 */
  ax0 = 0x0020;  dm(0x3fe1) = ax0;
/* set BM page to 0                     */
  ax0 = 0x0000;
/* test FI pin                          */
  if flag_in jump 0x0007;
/* set BM page to 1                     */
    ax0 = 0x0100;
/* at address 0x0007: BDMA Control      */
  dm(0x3fe3) = ax0;
/* BWCOUNT (worst case is 658 words)    */
  ax0 = 0x0292;  dm(0x3fe4) = ax0;
/* clear and unmask BDMA interrupt      */
  ifc = 0x0008;  nop;
  imask = 0x0008;
/* halt core until IRQ                  */
  idle;
/* jump to page loader                  */
  jump 0x0020;   nop;
  nop; nop; nop; nop;
  nop; nop; nop; nop;
  nop; nop; nop; nop;
/* rti still at 0x001C !!!              */
  rti; nop; nop; nop;
/* next address is 0x0020               */
```

Listing 4: Example of a user-defined preloader that evaluates the FI input pin

The preloader in Listing 4 always loads 658 page loader instructions. This is the maximal number of page loader instructions generated by VisualDSP++ 3.0, but this number may differ in future releases. If an application does not require booting overlay pages a BWCOUNT value of 135 instructions may be sufficient.

The EZ-KIT Lite of the ADSP-2189M connects a push button to the memory-mapped flag input PF4. An alternative preloader that evaluates the state of PF4 could look like the following code example.

```
/* example of an user-defined preloader
 * (32 instructions)
 * Input pin PF4 selects program
 * to be booted
 */
/* start at address 0x0000              */
.section / pm interrupts;
/* BEAD                                 */
  ax0 = 0x0060; dm(0x3fe2) = ax0;
/* BIAD                                 */
  ax0 = 0x0020; dm(0x3fe1) = ax0;
/* read PFDATA and test PF4             */
  ar = dm(0x3fe5);
  ar = tstbit 4 of ar;
/* result is either 0 or 0x0010         */
/* if 0x0010 change to 0x0100           */
  if ne ar = pass 0x0100;
/* BDMA Control                         */
  dm(0x3fe3) = ar;
/* BWCOUNT                              */
  ax0 = 0x0292; dm(0x3fe4) = ax0;
/* clear and unmask BDMA interrupt      */
  ifc = 0x0008; nop;
  imask = 0x0008;
/* halt core until IRQ                  */
  idle;
/* jump to page loader                  */
  jump 0x0020;   nop;
  nop; nop; nop; nop;
  nop; nop; nop; nop;
  nop; nop; nop; nop;
/* rti still at 0x001C !!!              */
  rti; nop; nop; nop;
/* next address is 0x0020               */
```

Listing 5: Example of a user-defined preloader that evaluates the PF4 input pin

Again, these examples assume that the single applications do not take more than 0x4000 bytes

and that the page loader consists of 658 instructions.

More choices required?

Sampling a single input flag pin like the previous examples does limit the number of program options to two. Of course, this number can be increased by evaluating multiple PFx pins. Application note EE-125 used an alternative technique that might be of general interest:

Many applications use the external memory bus only for booting; some connect 16-bit devices only. In case the eight lower data pins D0..D7 are unused, they can be connected by jumpers to a little network consisting of pull-up and pull-down resistors. Depending on the jumper settings individual pins read '0' or '1'. Just make sure that these pins are never shorted to a power supply or a buffer output.

```
    i4 = 0x2000;
    m4 = 0;

    pmovlay = 1;
    ar = pm(i4,m4);
    pmovlay = 0;
    ar = px;

    jump loadapplication;

/* sr = lshift ar by 8 (lo);
 * dm(0x3fe3) = sr0;
 */
```

Listing 6: Read D0..D7 into AR

The little piece of code in Listing 6 reads the 8 LSBs and stores the result into AR like required by Listing 2. It may also replace the PF4 handling in Listing 5. Then an additional left shift is required in order to copy AR into the BMPAGE bit field properly.

References

- ADSP-218x DSP Hardware Reference
- ADSP-218x DSP Instruction Set Reference
- VisualDSP++ 3.0 Online Help
- VisualDSP++ 2.0 Linker and Utilities Manual for ADSP-21xx DSPs
- VisualDSP 6.1 Release Notes
- ADSP-218x Embedded system software management and In-System Programming (EE-125)
- ADSP-2106x: Storing multiple applications in a single boot EPROM (EE-108)

Document History

Version	Description
Sep. 5, 02	Discussion of VisualDSP++ 3.0 and general rewording. Chapters "More choices required?" and "If individual boot images don't fit into a single BM page" added.
Sep. 20, 01	Initial version based on VisualDSP++ 2.0