Freescale QUICC Engine Firmware Uploading

Revision Information

November 30, 2007: Rev 1.0 - Initial version

I - Software License for Firmware

Each firmware file comes with its own software license. For information on the particular license, please see the license text that is distributed with the firmware.

II - Microcode Availability

Firmware files are distributed through various channels. Some are available on http://opensource.freescale.com. For other firmware files, please contact your Freescale representative or your operating system vendor.

III - Description and Terminology

In this document, the term 'microcode' refers to the sequence of 32-bit integers that compose the actual QE microcode.

The term 'firmware' refers to a binary blob that contains the microcode as well as other data that

1. describes the microcode's purpose
2. describes how and where to upload the microcode
3. specifies the values of various registers
4. includes additional data for use by specific device drivers

Firmware files are binary files that contain only a firmware.

IV - Microcode Programming Details

The QE architecture allows for only one microcode present in I-RAM for each RISC processor. To replace any current microcode, a full QE reset (which disables the microcode) must be performed first.

QE microcode is uploaded using the following procedure:

1. The microcode is placed into I-RAM at a specific location, using the IRAM.IADD and IRAM.IDATA registers.
2. The CERCR.CIR bit is set to 0 or 1, depending on whether the firmware needs split I-RAM. Split I-RAM is only meaningful for SOCs that have QEs with multiple RISC processors, such as the 8360. Splitting the I-RAM allows each processor to run a different microcode, effectively creating an asymmetric multiprocessing (AMP) system.
3. The TIBCR trap registers are loaded with the addresses of the trap handlers in the microcode.
4. The RSP.ECCR register is programmed with the value provided.
5. If necessary, device drivers that need the virtual traps and extended mode data will use them.

Virtual Microcode Traps

These virtual traps are conditional branches in the microcode. These are "soft" provisional introduced in the ROMcode in order to enable higher flexibility and save h/w traps If new features are activated or an issue is being fixed in the RAM package utilizing they should be activated. This data structure signals the microcode which of these virtual traps is active.

This structure contains 6 words that the application should copy to some specific been defined. This table describes the structure:

Offset in array	Protocol	Destination Offset within PRAM	Size of Operand
0	Ethernet interworking	0xF8	4 bytes
4	ATM interworking	0xF8	4 bytes
8	PPP interworking	0xF8	4 bytes
12	Ethernet RX Distributor Page	0x22	1 byte

```
|     16    | ATM Globtal      |      0x28          | 1 byte  |
|           | Params Table     |                    |         |
-------------------------------------------------------------
|     20    | Insert Frame     |      0xF8          | 4 bytes |
-------------------------------------------------------------
```

Extended Modes

This is a double word bit array (64 bits) that defines special functionality which has an impact on the software drivers. Each bit has its own impact and has special instructions for the s/w associated with it. This structure is described in this table:

```
-----------------------------------------------------------------------
| Bit #  |     Name     |   Description                                |
-----------------------------------------------------------------------
|   0    | General      | Indicates that prior to each host command    |
|        | push command | given by the application, the software must  |
|        |              | assert a special host command (push command)|
|        |              | CECDR = 0x00800000.                          |
|        |              | CECR = 0x01c1000f.                           |
-----------------------------------------------------------------------
|   1    | UCC ATM      | Indicates that after issuing ATM RX INIT     |
|        | RX INIT      | command, the host must issue another special|
|        | push command | command (push command) and immediately       |
|        |              | following that re-issue the ATM RX INIT      |
|        |              | command. (This makes the sequence of         |
|        |              | initializing the ATM receiver a sequence of  |
|        |              | three host commands)                         |
|        |              | CECDR = 0x00800000.                          |
|        |              | CECR = 0x01c1000f.                           |
-----------------------------------------------------------------------
|   2    | Add/remove   | Indicates that following the specific host   |
|        | command      | command: "Add/Remove entry in Hash Lookup    |
|        | validation   | Table" used in Interworking setup, the user  |
|        |              | must issue another command.                  |
|        |              | CECDR = 0xce000003.                          |
|        |              | CECR = 0x01c10f58.                           |
-----------------------------------------------------------------------
|   3    | General push | Indicates that the s/w has to initialize     |
|        | command      | some pointers in the Ethernet thread pages   |
|        |              | which are used when Header Compression is    |
|        |              | activated.  The full details of these        |
|        |              | pointers is located in the software drivers.|
-----------------------------------------------------------------------
|   4    | General push | Indicates that after issuing Ethernet TX     |
|        | command      | INIT command, user must issue this command   |
|        |              | for each SNUM of Ethernet TX thread.         |
|        |              | CECDR = 0x00800003.                          |
|        |              | CECR = 0x7'b{0}, 8'b{Enet TX thread SNUM},   |
|        |              |        1'b{1}, 12'b{0}, 4'b{1}               |
-----------------------------------------------------------------------
| 5 - 31 |     N/A      | Reserved, set to zero.                       |
-----------------------------------------------------------------------
```

V - Firmware Structure Layout

QE microcode from Freescale is typically provided as a header file. This header file contains macros that define the microcode binary itself as well as some other data used in uploading that microcode. The format of these files do not lend themselves to simple inclusion into other code. Hence, the need for a more portable format. This section defines that format.

Instead of distributing a header file, the microcode and related data are embedded into a binary blob. This blob is passed to the qe_upload_firmware() function, which parses the blob and performs everything necessary to upload the microcode.

All integers are big-endian. See the comments for function qe_upload_firmware() for up-to-date implementation information.

This structure supports versioning, where the version of the structure is embedded into the structure itself. To ensure forward and backwards compatibility, all versions of the structure must use the same 'qe_header' structure at the beginning.

'header' (type: struct qe_header):

> The 'length' field is the size, in bytes, of the entire structure, including all the microcode embedded in it, as well as the CRC (if present).
>
> The 'magic' field is an array of three bytes that contains the letters 'Q', 'E', and 'F'. This is an identifier that indicates that this structure is a QE Firmware structure.
>
> The 'version' field is a single byte that indicates the version of this structure. If the layout of the structure should ever need to be changed to add support for additional types of microcode, then the version number should also be changed.

The 'id' field is a null-terminated string(suitable for printing) that identifies the firmware.

The 'count' field indicates the number of 'microcode' structures. There must be one and only one 'microcode' structure for each RISC

processor. Therefore, this field also represents the number of RISC processors for this SOC.

The 'soc' structure contains the SOC numbers and revisions used to match the microcode to the SOC itself. Normally, the microcode loader should check the data in this structure with the SOC number and revisions, and only upload the microcode if there's a match. However, this check is not made on all platforms.

Although it is not recommended, you can specify '0' in the soc.model field to skip matching SOCs altogether.

The 'model' field is a 16-bit number that matches the actual SOC. The 'major' and 'minor' fields are the major and minor revision numbers, respectively, of the SOC.

For example, to match the 8323, revision 1.0:

```
soc.model = 8323
soc.major = 1
soc.minor = 0
```

'padding' is necessary for structure alignment. This field ensures that the 'extended_modes' field is aligned on a 64-bit boundary.

'extended_modes' is a bitfield that defines special functionality which has an impact on the device drivers. Each bit has its own impact and has special instructions for the driver associated with it. This field is stored in the QE library and available to any driver that calles qe_get_firmware_info().

'vtraps' is an array of 8 words that contain virtual trap values for each virtual traps. As with 'extended_modes', this field is stored in the QE library and available to any driver that calles qe_get_firmware_info().

'microcode' (type: struct qe_microcode):

> For each RISC processor there is one 'microcode' structure. The first 'microcode' structure is for the first RISC, and so on.
>
> The 'id' field is a null-terminated string suitable for printing that identifies this particular microcode.
>
> 'traps' is an array of 16 words that contain hardware trap values for each of the 16 traps. If trap[i] is 0, then this particular trap is to be ignored (i.e. not written to TIBCR[i]). The entire value is written as-is to the TIBCR[i] register, so be sure to set the EN and T_IBP bits if necessary.
>
> 'eccr' is the value to program into the ECCR register.
>
> 'iram_offset' is the offset into IRAM to start writing the microcode.
>
> 'count' is the number of 32-bit words in the microcode.
>
> 'code_offset' is the offset, in bytes, from the beginning of this structure where the microcode itself can be found. The first microcode binary should be located immediately after the 'microcode' array.
>
> 'major', 'minor', and 'revision' are the major, minor, and revision version numbers, respectively, of the microcode. If all values are 0, then these fields are ignored.
>
> 'reserved' is necessary for structure alignment. Since 'microcode' is an array, the 64-bit 'extended_modes' field needs to be aligned on a 64-bit boundary, and this can only happen if the size of 'microcode' is a multiple of 8 bytes. To ensure that, we add 'reserved'.

After the last microcode is a 32-bit CRC. It can be calculated using this algorithm:

```
u32 crc32(const u8 *p, unsigned int len)
{
      unsigned int i;
      u32 crc = 0;

      while (len--) {
         crc ^= *p++;
         for (i = 0; i < 8; i++)
                 crc = (crc >> 1) ^ ((crc & 1) ? 0xedb88320 : 0);
      }
      return crc;
}
```

VI - Sample Code for Creating Firmware Files

A Python program that creates firmware binaries from the header files normally distributed by Freescale can be found on http://opensource.freescale.com.