AVR32708: AVR32 UC3A and UC3B Flash JTAG Programming Algorithms

Features
- Low level JTAG programming algorithms for UC3A and UC3B devices internal FLASH

1. Introduction
The aim of this application note is to provide 3rd party programmer vendors, the JTAG programming specification for AT32UC3x products.

The following algorithms assume that programmer has knowledge of the IEEE®1149.1 JTAG norm as well as the information in UC3 product datasheet concerning JTAG and Service Access Bus sections. See section 5. Related Documents.

2. Product signature
Each UC3 product is identified by a unique JTAG signature made a 32-bit value. This signature is mentioned for each product in the product datasheet.

The following public (standard-defined) JTAG instruction IDCODE is used to read the JTAG signature.

The IDCODE is the default instruction loaded into the JTAG on power-up or after a JTAG reset.

This selects the 32-bit JTAG IDCODE register.

If the flash controller is in reset the ID code will be undefined.

Note that if the flash controller is in reset, the security bit in the device is set and JTAG instructions using the Service Access Bus will be blocked.
3. Programming via JTAG interface

3.1 Constants & addresses:

3.1.1 Flash controller addresses

The following addresses are used by the Flash controller. All offsets are relative to the base address allocated to this Flash controller.

For AT32UC3A & AT32UC3B devices, Flash controller mapping has changed with latest revision of silicon:

For AT32UC3 devices, the JTAG ID code is the following 0x<revision><part number>03F.

On AT32UC3A device for revision less than 7 (Revision H) and on AT32UC3B device for revision less than 3, the flash controller mapping is the following:

<table>
<thead>
<tr>
<th>Register definition</th>
<th>Name</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flash Control Register</td>
<td>FCR</td>
<td>0x00</td>
</tr>
<tr>
<td>Flash Command Register</td>
<td>FCMD</td>
<td>0x04</td>
</tr>
<tr>
<td>Flash Status Register</td>
<td>FSR</td>
<td>0x08</td>
</tr>
<tr>
<td>Flash General Purpose Fuses Register</td>
<td>FGPFR</td>
<td>0x0C</td>
</tr>
</tbody>
</table>

On AT32UC3A devices with revision equal or above 7 (Revision H) and on AT32UC3B devices with revision equal or above 3 (Revision C), the flash controller mapping is the following:

<table>
<thead>
<tr>
<th>Register definition</th>
<th>Name</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flash Control Register</td>
<td>FCR</td>
<td>0x00</td>
</tr>
<tr>
<td>Flash Command Register</td>
<td>FCMD</td>
<td>0x04</td>
</tr>
<tr>
<td>Flash Status Register</td>
<td>FSR</td>
<td>0x08</td>
</tr>
<tr>
<td>Flash General Purpose Fuses Register High</td>
<td>FGPFRHI</td>
<td>0x0C</td>
</tr>
<tr>
<td>Flash General Purpose Fuses Register Low</td>
<td>FGPFRLO</td>
<td>0x10</td>
</tr>
</tbody>
</table>
### 3.1.2 Flash controller constants:

<table>
<thead>
<tr>
<th>Constant definition</th>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flash Status Ready Mask address</td>
<td>FSR_FRDY_MASK</td>
<td>0x00000001</td>
</tr>
<tr>
<td>Flash Status Ready Offset</td>
<td>FSR_FRDY_OFFSET</td>
<td>0</td>
</tr>
<tr>
<td>Flash Status Programming Enable Mask Address</td>
<td>FSR_PROGE_MASK</td>
<td>0x00000008</td>
</tr>
<tr>
<td>Flash Status Programming Enable Offset</td>
<td>FSR_PROGE_OFFSET</td>
<td>3</td>
</tr>
<tr>
<td>Flash Status Lock Enable Mask Address</td>
<td>FSR_LOCKE_MASK</td>
<td>0x00000004</td>
</tr>
<tr>
<td>Flash Status Lock Enable Offset</td>
<td>FSR_LOCKE_OFFSET</td>
<td>2</td>
</tr>
<tr>
<td>Flash Status Size Mask Address</td>
<td>FSR_FSZ_MASK</td>
<td>0x0000E000</td>
</tr>
<tr>
<td>Flash Status Size Offset</td>
<td>FSR_FSZ_OFFSET</td>
<td>13</td>
</tr>
<tr>
<td>Flash Command Mask Address</td>
<td>FCMD_FCMD_MASK</td>
<td>0x0000001F</td>
</tr>
<tr>
<td>Flash Command Offset</td>
<td>FCMD_FCMD_OFFSET</td>
<td>0</td>
</tr>
<tr>
<td>Flash Command Page Enable Mask Address</td>
<td>FCMD_PAGEN_MASK</td>
<td>0x00FFFF00</td>
</tr>
<tr>
<td>Flash Command Page Enable Offset</td>
<td>FCMD_PAGEN_OFFSET</td>
<td>8</td>
</tr>
<tr>
<td>Flash Command Key Mask Address</td>
<td>FCMD_KEY_MASK</td>
<td>0xFF000000</td>
</tr>
<tr>
<td>Flash Command Key Offset</td>
<td>FCMD_KEY_OFFSET</td>
<td>24</td>
</tr>
<tr>
<td>Flash GP Fuse Lock Mask Address</td>
<td>FGPFR_LOCK_MASK</td>
<td>0x0000FFFF</td>
</tr>
<tr>
<td>Flash GP Fuse Lock Offset</td>
<td>FGPFR_LOCK_OFFSET</td>
<td>0</td>
</tr>
<tr>
<td>Number of words in a FLASH page</td>
<td>WORDS_PER_PAGE</td>
<td>128</td>
</tr>
<tr>
<td>Number of bytes in a FLASH page</td>
<td>BYTES_PER_PAGE</td>
<td>WORDS_PER_PAGE * 4</td>
</tr>
<tr>
<td>User Page Offset</td>
<td>USER_PAGE_OFFSET</td>
<td>0x00800000</td>
</tr>
<tr>
<td>Write Protect command</td>
<td>WRITE_PROTECT_KEY</td>
<td>0x50000000</td>
</tr>
<tr>
<td>Write Page command</td>
<td>CMD_WRITE_PAGE</td>
<td>1</td>
</tr>
<tr>
<td>Erase Page command</td>
<td>CMD_ERASE_PAGE</td>
<td>2</td>
</tr>
<tr>
<td>Clear Page Buffer command</td>
<td>CMD_CLEAR_PAGE_BUFFER</td>
<td>3</td>
</tr>
<tr>
<td>Lock Region command</td>
<td>CMD_LOCK_REGION</td>
<td>4</td>
</tr>
<tr>
<td>Unlock Region command</td>
<td>CMD_UNLOCK_REGION</td>
<td>5</td>
</tr>
<tr>
<td>Erase All command</td>
<td>CMD_ERASE_ALL</td>
<td>6</td>
</tr>
<tr>
<td>Write General Purpose Fuse Bit command</td>
<td>CMD_WRITE_GP_FUSE_BIT</td>
<td>7</td>
</tr>
<tr>
<td>Erase General Purpose Fuse Bit command</td>
<td>CMD_ERASE_GP_FUSE_BIT</td>
<td>8</td>
</tr>
<tr>
<td>Set Security Bit command</td>
<td>CMD_SET_SECURITY_BIT</td>
<td>9</td>
</tr>
<tr>
<td>Program General Purpose Fuse Byte command</td>
<td>CMD_PROGRAM_GP_FUSE_BYTE</td>
<td>10</td>
</tr>
<tr>
<td>Write User Page command</td>
<td>CMD_WRITE_USER_PAGE</td>
<td>13</td>
</tr>
<tr>
<td>Erase User Page command</td>
<td>CMD_ERASE_USER_PAGE</td>
<td>14</td>
</tr>
</tbody>
</table>
3.1.3  **Low level Flash Memory functions:**

The read & write memory access are made thru the JTAG interface: **ReadMemoryWord** & **WriteMemoryWord**.

getRegister(address) is a ReadMemoryWord access to the "address" location.

setRegister(address, value) is a WriteMemoryWord access to the "address" location.

3.1.3.1  **WriteCommand(Command):**

```c
WriteCommand(Command):
{
    WriteMemoryWord(FCMD, Command);
}
```

3.1.3.2  **WaitFlashReady:**

```c
WaitFlashReady:
{
    Timeout=1 Second
    while (Not Timeout)
    {
        uint32_t fsrReg = getRegister(FSR);
        // If LOCKE bit in FSR set
        if((fsrReg & FSR_LOCKE_MASK) >> FSR_LOCKE_OFFSET)
            Error: Programming of at least one locked lock region has happened since
            the last read of FSR.
        // If PROGE bit in FSR set
        if((fsrReg & FSR_PROGE_MASK) >> FSR_PROGE_OFFSET)
            Error: An invalid command and/or bad keywords were written in the Flash
            Command Register.
        // Read FRDY bit in FSR
        if ((fsrReg & FSR_FRDY_MASK) >> FSR_FRDY_OFFSET)
            return; // FLASH ready for next operation
    }
}
```
3.1.3.3 \textit{ClearPageBuffer}

\textbf{ClearPageBuffer:}

\begin{verbatim}
uint32_t command = WRITE_PROTECT_KEY | CMD_CLEAR_PAGE_BUFFER;
WaitFlashReady
WriteCommand(Command);
WaitFlashReady
\end{verbatim}

3.1.3.4 \textit{GetInternalFlashSize}

\textbf{Unsigned int GetInternalFlashSize:}

\begin{verbatim}
uint32_t fsrReg = getRegister(FSR);
unsigned int fsz = (fsrReg & FSR_FSZ_MASK) >> FSR_FSZ_OFFSET;
unsigned int size = 0;
switch (fsz)
{
    case 0:  size = 32 * 1024;
             break;
    case 1:  size = 64 * 1024;
             break;
    case 2:  size = 128 * 1024;
             break;
    case 3:  size = 256 * 1024;
             break;
    case 4:  size = 384 * 1024;
             break;
    case 5:  size = 512 * 1024;
             break;
    case 6:  size = 768 * 1024;
             break;
    case 7:  size = 1024 * 1024;
             break;
    default: Error: Unknown flash size
             break;
}
return size;
\end{verbatim}
3.1.4 Unlock Region

UnlockRegion(unsigned int Offset, unsigned int Size)
{
    if (Offset >= USER_PAGE_OFFSET
        && Offset < USER_PAGE_OFFSET + BYTES_PER_PAGE)
        return; // the user page doesn't need unlocking
    if (Offset >= mDeviceSize || Offset + size > mDeviceSize)
        Error: Region to be unlock lies outside flash address space.
    int lastpagetounlock = ((Offset + Size) / BYTES_PER_PAGE)
    // compute start offset of page to write to
    uint32_t page = Offset & ~(BYTES_PER_PAGE - 1);
    int pagenr = ((Offset) / BYTES_PER_PAGE)
    while (pagenr <= lastpagetounlock)
    {
        uint32_t command = WRITE_PROTECT_KEY | CMD_UNLOCK_REGION;
        // include the correct page number in the command
        command |= ((pagenr << FCMD_PAGEN_OFFSET) & FCMD_PAGEN_MASK);
        // Unlocking page: pagenr
        WaitFlashReady
        WriteCommand(command); // execute unlock page command
        WaitFlashReady
        page += BYTES_PER_PAGE;
        Offset = page;
        pagenr = ((Offset) / BYTES_PER_PAGE)
    }
}

3.1.5 Unlock entire FLASH

Call the UnlockRegion function with Offset: 0 and total size of the device
UnlockEntireFlash:
{
    DeviceSize = GetInternalFlashSize;
    UnlockRegion(0, DeviceSize);
}
3.1.6 Erase sequence:

```c
EraseSequence:
{
    WaitForFlashReady
    Command = WRITE_PROTECT_KEY | CMD_ERASE_ALL
    WriteCommand(Command):
    WaitForFlashReady
}
```

3.1.7 Erase User Page sequence:

```c
EraseUserPage:
{
    uint32_t command = WRITE_PROTECT_KEY | CMD_ERASE_USER_PAGE;
    WaitForFlashReady
    WriteCommand(command); // execute user page erase command
    WaitForFlashReady
}
```

3.1.8 Erase region sequence:

```c
EraseRegionSequence(Offset, Size):
{
    if (Offset >= USER_PAGE_OFFSET
        && Offset < USER_PAGE_OFFSET + BYTES_PER_PAGE)
        EraseUserPage

    lastpagetoerase = (Offset + Size) / BYTES_PER_PAGE
    page = Offset & ~(BYTES_PER_PAGE - 1) // compute start offset of page to write to
   pagenr = (Offset) / BYTES_PER_PAGE
    while (pagenr <= lastpagetoerase)
    {
        Command = WRITE_PROTECT_KEY | CMD_ERASE_PAGE
        Command |= ((pagenr << FCMD_PAGEN_OFFSET) & FCMD_PAGEN_MASK)
        // include the correct page number in the command
        WaitForFlashReady
        WriteCommand(Command) // execute page erase command
        WaitForFlashReady

        page += BYTES_PER_PAGE;
        Offset = page;
        pagenr = (Offset) / BYTES_PER_PAGE
    }
}
```
3.1.9 Programming sequences:

3.1.9.1 Program User Page sequence:

```c
ProgramUserPage(Offset - USER_PAGE_OFFSET, DataBuffer):
{
    if (Offset >= BYTES_PER_PAGE || Offset + Length(DataBuffer) > BYTES_PER_PAGE)
        Error: Tried to program past user page boundary

    // Packet bufferPacket(BYTES_PER_PAGE) define a buffer packet
    // to manipulate the data
    // If the packet to be written is smaller than the user page we fill the
    // remaining space with existing data
    if (Offset > 0 || Length(DataBuffer) < BYTES_PER_PAGE)
        ReadMemory(mBaseAddress + USER_PAGE_OFFSET, bufferPacket, 0);
    // Must clear the page buffer before writing to it.
    ClearPageBuffer();

    int bytesLeftInPacket = Length(DataBuffer);
    int i = 0;  // data packet index
    // Fill buffer packet
    while (bytesLeftInPacket > 0)
    {
        bufferPacket.writeSingleByte(Offset++, ReadSingleByte(DataBuffer)
            i++;
        bytesLeftInPacket--;
    }
    // Write page buffer
    WriteMemory(mBaseAddress + USER_PAGE_OFFSET, bufferPacket);
    uint32_t command = WRITE_PROTECT_KEY | CMD_WRITE_USER_PAGE;
    WaitFlashReady
    WriteCommand(command);  // execute user page write command
    WaitFlashReady
}```
### 3.1.9.2 Program sequence:

**ProgramSequence**(Offset, DataBuffer):  

```c
if (Offset >= USER_PAGE_OFFSET  
    && Offset < USER_PAGE_OFFSET + BYTES_PER_PAGE)  
    ProgramUserPage(Offset - USER_PAGE_OFFSET, DataBuffer)
```

```c
if (Offset >= mDeviceSize || Offset+Length(DataBuffer) > mDeviceSize)  
    Error: Region to be programmed lies outside flash address space.
```

```c
// compute start offset of page to write to  
uint32_t page = Offset & ~(BYTES_PER_PAGE - 1);  
unsigned int bytesLeft = (Length(DataBuffer));  
int dataOffset = 0; // current offset in the data packet  
Packet bufferPacket(BYTES_PER_PAGE); // we write one page at a time
```

```c
// Loop until all bytes in data has been written  
while (bytesLeft > 0)  
```

```c
{
    bufferPacket.clear(0xff);
```

```c
    // Must clear the page buffer before writing to it.  
    ClearPageBuffer();
```

```c
    /* Keeps track of how many bytes to write to the bufferPacket.  
    * If the start offset is not aligned on a page boundary, we will not fill  
    * the bufferPacket completely. This is also the case when the number of  
    * bytes left to write is less than the size of a page. If the bufferPacket  
    * is not filled completely we first read the current flash content into  
    * the packet. This way we will always preserve existing flash data  
    * adjacent to the new data we wish to write.  
    */  
    int bytesLeftInPacket = min((page+BYTES_PER_PAGE-Offset), bytesLeft);  
    int bufferOffset = Offset % BYTES_PER_PAGE;  
    if (bufferOffset != 0 || bytesLeftInPacket != (BYTES_PER_PAGE))  
```

```c
        ReadMemory(mBaseAddress + page, bufferPacket, 0);
```

```c
    }  
    for (int i = 0; i < bytesLeftInPacket; ++i)  
```

```c
    {
        bufferPacket.writeSingleByte(bufferOffset++,  
            ReadSingleByte(DataBuffer));  
        Offset++;
    }
```
WriteMemory(mBaseAddress + page, bufferPacket);
intpagenr = ((Offset) / BYTES_PER_PAGE);
uint32_tcommand = WRITE_PROTECT_KEY | CMD_WRITE_PAGE;
// include the correct page number in the command
command |= pagenr << FCMD_PAGEN_OFFSET;
WaitFlashReady
WriteCommand(command); // execute page write command
WaitFlashReady
page += BYTES_PER_PAGE;
Offset = page;
bytesLeft -= bytesLeftInPacket;
}
3.1.10 Fuses management sequences:

3.1.10.1 Get General Purpose Fuse bits

```c
bool GetGeneralPurposeFuseBit(int Index)
{
    if (Index > 31 || Index < 0)
        Error: Index out of range

    uint32_t fgprReg = getRegister(FGPFR);
    return fgprReg & (1 << Index);
}
```

Note: FGPFR register becomes FGPFRHI or FGPFRLO for new silicon revisions, see 3.1.1 Flash controller addresses chapter.

3.1.10.2 Set General Purpose Fuse bits

```c
SetGeneralPurposeFuseBit(int Index, bool Value):
{
    if (Index > 31 || Index < 0)
        Error: Index out of range.

    uint32_t command = WRITE_PROTECT_KEY | (index << FCMD_PAGEN_OFFSET);
    if (Value)  // erase bit
        command |= CMD_ERASE_GP_FUSE_BIT;
    else        // program bit
        command |= CMD_WRITE_GP_FUSE_BIT;

    WaitFlashReady
    WriteCommand(command);
    WaitFlashReady
}
```
3.1.10.3 Set General Purpose Fuse Byte

```c
SetGeneralPurposeFuseByte(int Index, unsigned char Value):
{
    if (Index > 3 || Index < 0)
        Error: Index out of range

    uint32_t command = WRITE_PROTECT_KEY | CMD_PROGRAM_GP_FUSE_BYTE
                      | Index << FCMD_PAGEN_OFFSET
                      | Value << (FCMD_PAGEN_OFFSET + 2);

    WaitFlashReady
    WriteCommand(command);
    WaitFlashReady
}
```

3.1.11 Security management sequences:

```c
SetSecurityBit()
{
    uint32_t command = WRITE_PROTECT_KEY | CMD_SET_SECURITY_BIT;
    WaitFlashReady
    WriteCommand(command)
    // Wait at least 2 seconds
    Wait(2s)
}
```

4. Limitations

None

5. Related Documents

- IEEE1149.1 JTAG Norm
- AT32UC3A and AT32UC3B product datasheets: JTAG & Service Access Bus chapter