Application Report
April 2008

DM299 Open Source Usage Guide

Tony Cave — *Multimedia Systems FAE*

Greg Hewes — *Applications/Systems Engineer*

ABSTRACT

The DM29X/DM50X family of devices are highly flexible HW accelerators for digital imaging and video applications with SW programmable engines. Before the device is able to perform the video and imaging acceleration required, these devices need the firmware for the SW engine downloaded to them. The host processor in the system is required to store and download firmware images to DM29X/DM50X when required.

This document is intended to give development engineers a thorough understanding of the boot sequence, and steps needed to accelerate imaging and video tasks on the DM29X/DM50X. For simplicity in this documentation, the DM29X/DM50X devices will be referred to collectively as DM299.

Note: This document is strictly for wireless/cellular software developers using DM299/DM50x Media Co-processors, which are not available for the broad market through authorized distributors.

Contents

Figures

Tables

1 Scope

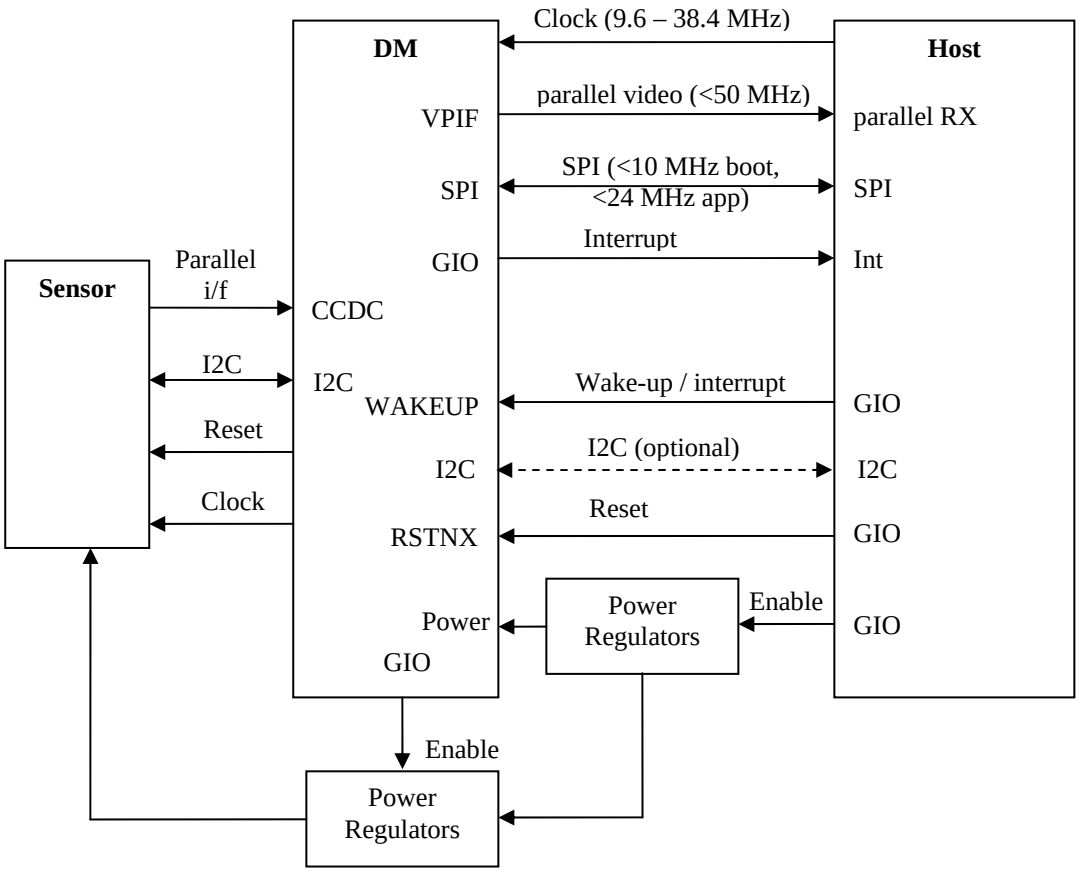
The scope of this document is to enable a HW/SW developer to understand the communication flow between Host and DM299 required to boot the device, at a bit and byte level. This understanding is crucial to anyone implementing a Host side DM299 device driver, and also for anybody wanting to test the device's connectivity with the host. There are 2 types of framework for DM based devices; Messaging based and Register based.

Loading an application to the DM299 is a 2 stage process. First you must load a boot loader; secondly you must load the application. The first chapter of this document describes the boot sequence and the bits and bytes that are sent over the SPI interface during a bootload. This does not follow the messaging protocol and has its own protocol that needs to be applied at this stage. The second section of the document describes how the messaging protocol is used to load the application.

2 Signal connections

The drawing below shows an example of how to connect the DM299 to the host.

Figure 1 – DM299/host Connections

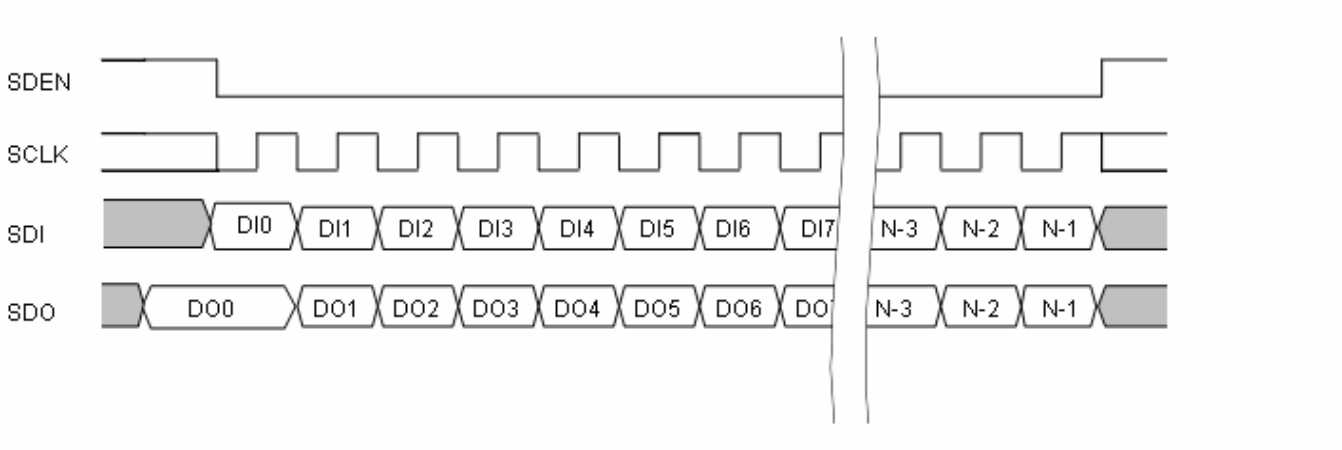


Note that the bootloader and application are built assuming a particular input frequency.

3 SPI Configuration

The DM299 is configured as a SPI slave, where the host will provide the clock and chip select for all SPI communication.

Figure 2 – Timing Diagram for Slave Mode Data Transfer



Notes:

1) N means the number of transfer bits.

2) SDEN is active low. When loading the downloading the bootloader using the SPI hardware boot protocol, SDEN must remain low throughout the duration of each section. During SPI messaging or application downloading, SDEN can toggle between bytes/words/sections.

3) SCLK is sampled on the rising edge. The clock can be inactive high or low. The maximum SCLK speed is 10 MHz for booting the device, and 24 MHz for messaging and application loading.

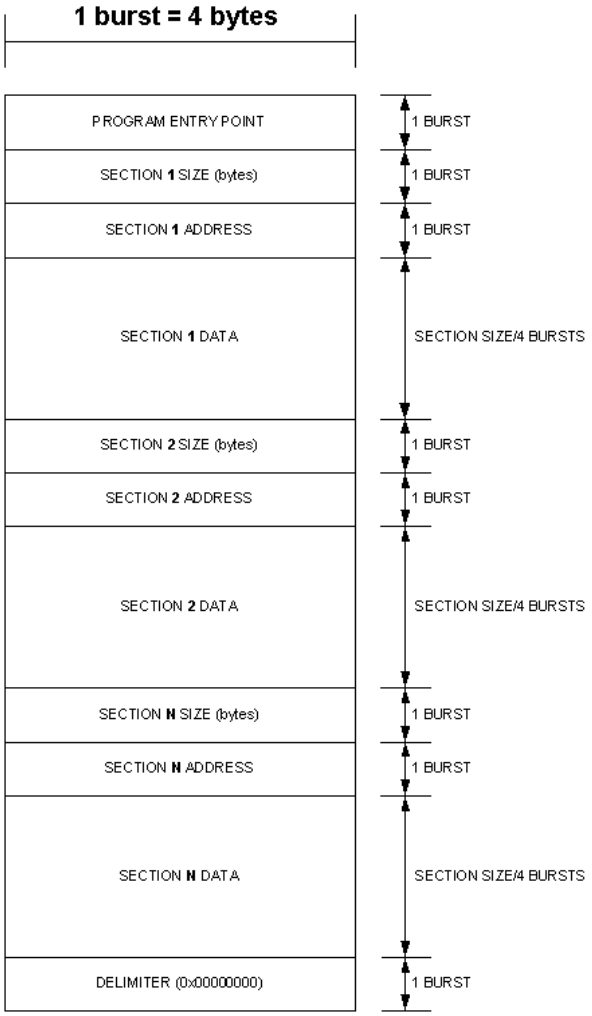
4 Downloading the boot loader

The DM299 supports two slave boot modes – I2C and SPI. This document covers the SPI bootload method supported by TI and other Third Party firmware. In this mode, the boot strap in the DM299 allows firmware to be downloaded directly into Arm internal memory (AIM) while the ARM is held in internal reset. The ARM will be released from RESET after the bootload is complete by pulsing the WAKEUP pin.

4.1 Format of the bin file

The boot coff file goes through a number of conversions to make it suitable for down loading from the Host. The .out (COFF) file is generated from the make file, this a file is then converted to a table of sections like that shown below:

Figure 3 - Boot file format



4.2 Boot sequence overview for booting over SPI

This document assumes that DM299 uses the SPI boot mode to download the boot image. In this mode the boot image is downloaded directly into ARM internal memory after the device reset is released but while the ARM is held in internal reset. The signal RSTNX is the external reset signal to the device. 16 SYSCLK cycles after the RSTNX signal is de-asserted, the device is out of reset and ready for the external host to begin the bootload process. It is important to note the ARM is still held in reset during this time. In this mode any activity on the SPI is parsed to find the AIM address and the data is written/read directly to/from AIM. The ARM is not released from reset until the WAKEUPX signal is strobed.

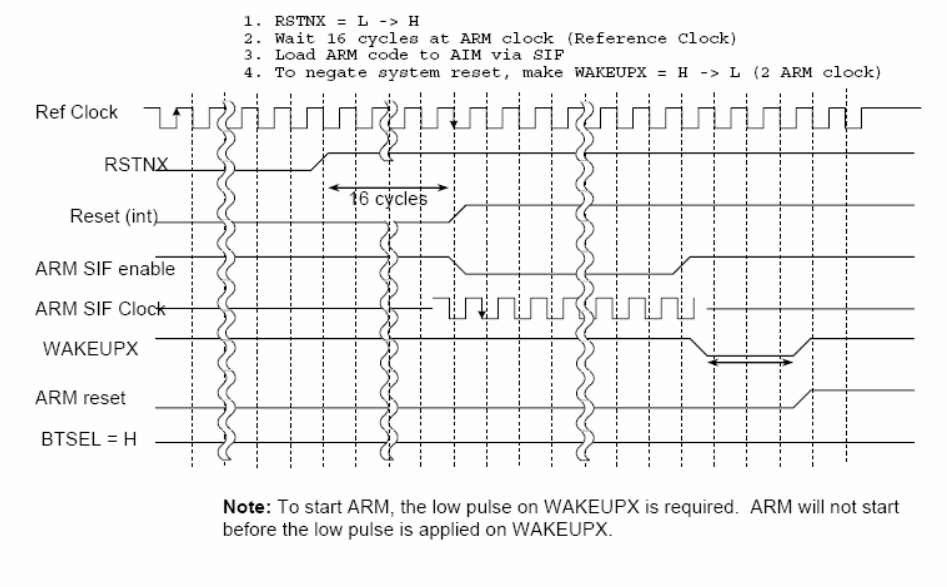


Figure 4 - Boot signals

The bootloader parsing done by the DM299 in the SPI boot mode is shown below. The AIM is being written to/read from while the device is out of reset but ARM is still in reset. The parsing starts from the first bit when the SDEN goes active. The first bit indicates a write (0) or read (1), the next 16b is the address of the access, and all following bits are data bytes for that section. The access will continue until the SDEN returns inactive. SDEN must remain active throughout each section, as this process will start again each time SDEN goes active.

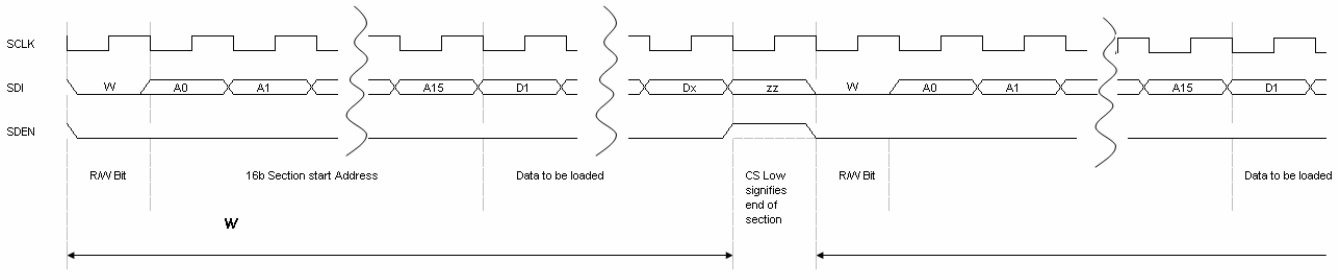


Figure 5 - Loading boot loader through SPI

During the loading of the boot loader the SPI clock should not exceed 10MHz.

CAUTION: During the period between the de-assertion of RESET and the assertion of WAKEUP, the DM29X/DM50X SPI SDO pin will be driven regardless of the SDEN pin. Please ensure that reads from other SPI slave devices on the bus do not occur during this initial boot period, as they will be corrupted.

4.2.1 RESET the device

The first step to initiate the bootload sequence is to reset the DM299 device, by asserting the RSTNX pin for at least 16 DM system clock (MCLK) cycles. Once RSTNX has been de-asserted, the device will boot over SPI or I2C, depending on whether the BOOTSEL pin is pulled high or low (respectively).

4.2.2 Host parsing

The binary supplied from TI consists of a number of memory sections. Each section needs to be loaded to a different address in AIM. Each section download will require the SPI SDEN to be made active and then inactive after the end of each section. SDEN going inactive then active will signify to the boot loader that a new section needs to be loaded (to a different address).

The host will need to parse the binary supplied by TI. The rest of this section describes what parsing is needed.

The bin file format is described in section 4.1. Below is a description of what happens during the file conversion process from the .out file to the format of the binary (boot.bin) that is supplied.

4.2.2.1 Bootload binary creation

When each bootloader/application is built, the compiler/linker generates coff files. These files must be processed through a TI-supplied script and be converted into loadable binaries. An example console output from the Coff2Table conversion utility is shown below:

```
string table complete
coff file contains 12 sections
memory usage: 6328/1175806775/1052, entry point 0x0000158c

section .debug_a (page 0) at 0x00000000-0x0000008c (0x0000008d bytes) .. skipped
section .debug_i (page 0) at 0x00000000-0x000004e4 (0x000004e5 bytes) .. skipped
section .debug_l (page 0) at 0x00000000-0x00000265 (0x00000266 bytes) .. skipped
section .vect    (page 0) at 0x00000000-0x0000003f (0x00000040 bytes) .. converted
section .spiexec (page 0) at 0x00000100-0x000001bb (0x000000bc bytes) .. converted
section .text    (page 0) at 0x00000200-0x00001ab7 (0x000018b8 bytes) .. converted
section .cinit   (page 0) at 0x00001ab8-0x00001af7 (0x00000040 bytes) .. converted
section .shared_ (page 0) at 0x010ffe00-0x010ffe07 (0x00000008 bytes) .. skipped
section .bss     (page 0) at 0x00008000-0x0000841b (0x0000041c bytes) .. skipped
section .stack   (page 0) at 0x0000841c-0x0000941b (0x00001000 bytes) .. skipped
section .mbuff_s (page 0) at 0x00080000-0x00080213 (0x00000214 bytes) .. skipped
section .mbuff_s (page 0) at 0x00080218-0x0008042b (0x00000214 bytes) .. skipped

boot.out converted
```

From the `Coff2Table` output file, you can see that 4 sections were converted to the table format required for the binary downloaded to the DM299.

A sample from the converted binary file is shown below. The first section shown below is the .vect section.

```
00000000h: 8C 15 00 00 40 00 00 00 00 00 00 00 18 F0 9F E5
00000010h: 18 F0 9F E5 18 F0 9F E5 18 F0 9F E5 18 F0 9F E5
00000020h: 18 F0 9F E5 18 F0 9F E5 18 F0 9F E5 8C 15 00 00
00000030h: 1C 16 00 00 2C 16 00 00 54 16 00 00 64 16 00 00
00000040h: 00 00 00 02 74 16 00 00 84 16 00 00

Code Entry point 0x158c (not part of section, only present at 1st 4 bytes of file)
.vect section size 0x40
.vect section start address 0x0000
.vect section data
```

The next section in the file is the .SPI section.

```
0000004ch: BC 00 00 00
00000050h: 00 01 00 00 AC 00 8F E2 04 60 90 E4 04 80 90 E4
00000060h: 40 70 A0 E3 21 00 00 EB 0A 90 A0 E1 1F 00 00 EB
....
000000f0h: B0 00 D8 E1 00 A0 D6 E5 07 00 10 E1 FB FF FF 1A
00000100h: B0 70 C8 E1 0E F0 A0 E1 82 03 03 00 08 05 03 00

.spiexec section size 0x00BC
.spiexec section start address 0x0100
.spiexec section data
```

The following sections can be decoded using the same principle.

4.2.2.2 Bootload binary 16_17conv conversion

From section 4.1 you can see that the DM299 SPI hardware bootloader expects a single R/W bit at the start of each section. To save complexity of the host parser, the R/W bit is inserted into the binary file. 1 bit is inserted at the beginning of each section (causing a 1bit shift in section address and data). 7 bits are then inserted into the end of the section to insure that the section is byte-aligned and conforming to the SPI transfer protocol. This 1-bit section shift is only required for boot.

This format is illustrated in the diagram below:

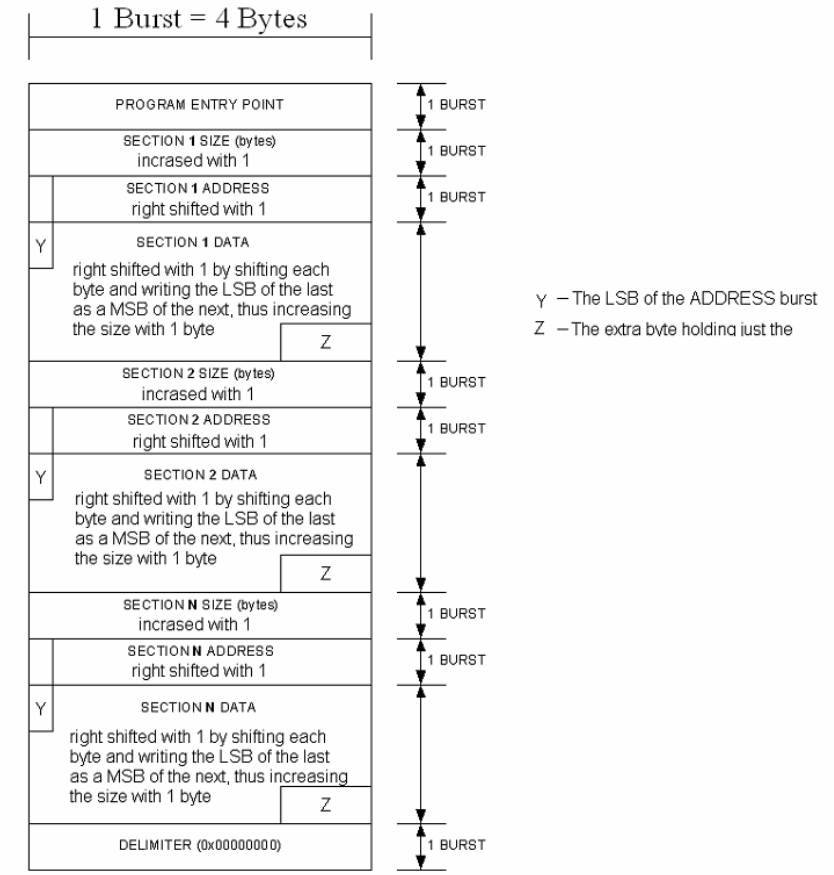


Figure 6 - 1bit shifted boot file

The sections in the file now look like the following, stating with .vect section:

```
00000000h: 8C 15 00 00 41 00 00 00 00 00 00 00 0C 78 4F F2
00000010h: 8C 78 4F F2 8C 78 4F F2 8C 78 4F F2 8C 78 4F F2
00000020h: 8C 78 4F F2 8C 78 4F F2 8C 78 4F F2 C6 0A 80 00
00000030h: 0E 0B 00 00 16 0B 00 00 2A 0B 00 00 32 0B 00 00
00000040h: 00 00 00 01 3A 0B 00 00 42 0B 00 00 00
```

```
Program Entry point 0x158c (not part of section), unaffected by the 1 bit shift.
.vect section size 0x41 (unaffected by shift).
vect section start 0 (bit shifted).
.vect section data (bit shifted).
7 bits inserted to make the section byte-aligned
```

Note that the extra 1 bit inserted after the size and 7 bits at the end of the section have increased the size from 0x40 to 0x41. Also, note that the Program Entry Point is not used during the boot sequence. After the internal ARM reset, the ARM always begins execution at AIM address 0x0000, so there is a hard-coded branch in the bootloader firmware.

The second section (.spi_exec) is shown next:

```
0000004dh: BD 00 00
00000050h: 00 80 00 00 00 56 00 47 F1 02 30 48 72 02 40 48
00000060h: 72 20 38 50 71 90 80 00 75 85 48 50 70 8F 80 00
....
000000f0h: 70 D8 00 6C 70 80 50 6B 72 83 80 08 70 FD FF FF
00000100h: 8D 58 38 64 70 87 78 50 70 C1 01 81 80 04 02 81
00000110h: 80 00

.vect section size 0xBD (unaffected by shift).
.vect section start 0x100 (bit shifted).
.vect section data (bit shifted).
7 bits inserted to make the section byte-aligned.
```

4.2.2.3 Host bootload over SPI

The host now needs to down load the binary to the DM299. As shown in Figure 5, the DM299 expects to receive the 1 bit read/write command followed by 16b address, followed by section data all within 1 SDEN period. The size is not sent to the DM299, as it is only used by the host to figure out where each section ends (and hence when the SPI_EN should be de-asserted). The host will need to parse the boot.bin, extract the size, and convert the 32b address to 16b. The load of the .spi_exec is shown below.

```
00 80 56 00 47 F1 02 30 48 72 02 40 48 72 20 38 50 71 90 80 00 75 85 48 50 70 8F 80 00 75 85
4A 44 F0 8E 80 00 75 85 4C 44 F0 8D 80 00 75 85 4E 44 F0 84 E8 50 70 8C 00 00 75 85 48 50 70
8B 00 00 75 85 4A 44 F0 8A 00 00 75 85 4C 44 F0 89 00 00 75 85 4E 44 F0 80 60 4C F1 07 00 00
05 07 00 00 75 85 48 50 70 86 00 00 75 85 4A 44 F0 85 00 00 75 85 4C 44 F0 84 00 00 75 85 4E
44 F0 84 D8 50 70 82 80 00 75 80 D0 65 F2 00 E0 2E 71 7D FF FF 8D 00 80 00 75 F3 FF FF F5 06
F8 50 70 D8 00 6C 70 80 50 6B 72 83 80 08 70 FD FF FF 8D 58 38 64 70 87 78 50 70 C1 01 81 80
04 02 81 80 00

.spi_exec 16b section start 0x100 (bit shifted).
.spi_exec section data (bit shifted).
7 bits inserted to make the section byte-aligned
```

4.2.3 Host data verification

If desired, the host can send a READ bit followed by a 16-bit address, in order to read the contents of the ARM internal memory. Instead of writing the data received from the SDI pin, the DM299 will output auto-incrementing data on the SDO pin until the SDEN is de-asserted.

4.2.4 Bootload complete

After the download is complete the Host should strobe the WAKEUP signal. This will release the DM299 ARM Core from reset and allow it to execute the downloaded binary.

4.2.5 Verifying successful download

After the bootloader has successfully begun execution and initialization, the DM299 will interrupt the Host. The interrupt will signify that the DM299 has a 532-byte or 2016-byte SPI message for the Host to read. Note that the message size is determined at build-time, depending on the requirements. The message will be a CAM_SEND_STATUS_CHANGE message, with a CAM_APP_READY value in the status field. This will indicate that the DM299 is ready for the host to load the application using the SPI messaging protocol. Downloading the application using the SPI messaging protocol is described in the following chapter.

5 Downloading the application

This section describes the steps to download the application using the SPI messaging protocol.

After successfully downloading the DM299 boot loader, the boot loader will execute and enable download of the application firmware. As the DM299 is operating as an SPI slave, it must interrupt the host when it has a message to send. All messages are 532-bytes in length (512B payload + 20B header) or 2016-bytes in length (1996B payload + 20B header), depending on the customer/application requirement. The message size is determined at build-time.

During the application load, the SPI clock can operate up to 24MHz. The application binary is loaded byte-by-byte to the DM299 (little endian). The application will automatically execute once the last byte of the application binary file has been transferred.

5.1 Reading the CAM_APP_READY message

On successfully loading the boot loader and executing, the DM299 should interrupt the host to indicate it is ready for firmware download. The host should then read 532/2016-bytes over the SPI by sending a dummy message to the DM299 (the host cannot send a new command to the DM299 until after it has received the initial status message without error). To ensure that a message from the host is not erroneously received and processed, the CAM_HEADER is the transmitted with a sequence number of 0 or unchanged from the previous message sent.

The message read should decode similar to the one shown below. A typical 532B CAM_APP_READY message is shown below.

01 00 00 00 12 80 04 00 06 00 00 00 3C 00 00 00 BA AE A7 AD 80 40 06 00 01 00 00 … 00

The message decodes to the following message:

```
typedef struct
{
    UData32     message_seq_num;
    UData16     message_id;
    UData16     message_status;
    UData32     message_size;
    UData32     message_time;
    UData32     message_checksum;
} CAM_HEADER;

Message_seq_num: 01.
Message_id: 0x12 = CAM_SEND_STATUS_CHANGE. Ack bit (bit 15) is also set.
Message_status: 0x0004 = CAM_APP_READY.
Message_size: 06 = For CAM_SEND_STATUS_CHANGE there are 6 bytes of payload.
message_time: Unused for this message.
Message_checksum: 0xBAAEA7AD. Checksum is always set to this value and is unused.
Message payload: Defined 6bytes, remainder undefined.
```

5.2 Sending the application load message

Before the application can be downloaded, the host must send a message to prepare the DM299 to accept the application download. This is done with a CAM_SET_CAMERA_VERSION message. To send a message to the DM299, the host must interrupt the DM299, and wait for an acknowledge interrupt from the DM299 before clocking entire 532/2016-byte message over the SPI interface. The CAM_SET_CAMERA_VERSION version message sent by the host should look like the following:

01 00 00 00 03 00 00 00 02 00 00 00 00 00 00 00 00 00 00 00 05 00 00 … 00

```
Message_seq_num: 01.
Message_id: CAM_SET_CAMERA_VERSION = 0x0003.
Message_status: CAM_APP_READY = 0x0000. Unused for this message.
Message_size: 02 = For CAM_SET_CAMERA_VERSION there is 2 bytes payload (see below).
message_time: Unused for this message.
Message_checksum: 0x00000000. Checksum is unused.
Message payload: 0x0005 (2bytes), remainder of 512/1996-byte payload is undefined.
```

This message has a 2-Byte payload, the message is defined below and shows that the payload is an application ID number.

```
typedef struct
{
CAM_HEADER;
UData16           app_number;
} CAM_SET_CAMERA_VERSION_TYPE;
```

It can be seen in the example above that the Application ID is 0x05 (image encoder). The application ID will depend on the application type being downloaded. The below table describes the current list of application ID's.

0x0000	APP_NUMBERS_CAM_APP_HOST	Application will be downloaded from the host
0x0001	APP_NUMBERS_CAM_JOTF	On the fly JPEG encoder
0x0003	APP_NUMBERS_CAM_SLEEP	Sleep application
0x0004	APP_NUMBERS_CAM_VIDEO_ENCODER	MPEG4 encoder
0x0005	APP_NUMBERS_CAM_IMAGE_ENCODER	JPEG encoder
0x0007	APP_NUMBERS_CAM_VIDEO_DECODER	MPEG4 decoder
0x000B	APP_NUMBERS_CAM_SELF_TEST	Self test application
0x000D	APP_NUMBERS_CAM_H264_DECODER	H.264 decoder
0x0011	APP_NUMBERS_CAM_H264_ENCODER	H.264 encoder
0x0012	APP_NUMBERS_CAM_VC1_DECODER	VC1 decoder
0x0013	APP_NUMBERS_CAM_VID_DECODER	Video Decoder Application

Table 1 -Application ID's

After sending the CAM_SET_CAMERA_VERSION Message, the host must wait for an interrupt back from the DM299. This will signify that the DM299 has a message for the Host. The host should then read entire 532/2016-byte message over the SPI.

The message received will be a CAM_SET_CAMERA_VERSION acknowledgement. This message is shown below:

02 00 00 00 03 80 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 … 00

```
Message_seq_num: 02.
Message_id: 0x8003 = CAM_SET_CAMERA_VERSION_ACK.
Message_status: 0x0000 = CAM_SUCCESS.
Message_size: 00 = No payload for this message.
message_time: Unused for this message.
Message_checksum: 0x00000000. Checksum is unused.
Message payload: 512/1996 byte payload is undefined.
```

After this message is received, the Host can send the application binary.

5.3 Sending the application binary

The application binary can now be sent as a binary bitstream. The binary still contains the COF sections as described in section 4.2.2.1 (without the 16_17conv bit-shift), but the parsing will occur on the fly from the DM299. Because of this, there is no restriction on SPI burst size and the SPI SDEN can optionally toggle between bytes.

The host does not need to send any message to identify that the download is complete. When the DM299 receives a 0 length section which is embedded at the end of each application binary, it will automatically execute the application that has just been downloaded.

To signify the successful download the first thing the application will do is interrupt the host to send a CAM_SEND_STATUS_CHANGE message with status CAM_APP_READY, as described in section 4.2.5.

6 SPI Messaging

The SPI messages provide the host the ability to:

- download application code to the DM299
- configure the DM299 (encode resolution/frame-rate/quality/parameters, decode output size, YUV/RGB/JPEG format, etc)
- read version information from the DM299
- change the state of the DM299 (run, stop)
- control the 3A function
- debug the DM299 (limited)
- send the DM299 encoded data (for decode application)

The messages provide the DM299 the ability to

- inform the host of a status change or error condition
- send encoded video data to the host

6.1 SPI Messaging Rules

The following rules apply to the SPI messaging protocol between the host and DM299.

1. Each message begins with a message sequence number which is incremented each time a new message is constructed. It is this message sequence number that is used to determine when a new message has been constructed for transmission and when a new message has been received. The message sequence counter will reset to zero after boot or whenever a new application is downloaded. The host must also reset its message sequence number when a new application is downloaded to the DM299. Therefore, the first valid message sent from both the host and DM299 will be of sequence number 1. The message sequence number simply overflows to zero when it reaches the maximum value.

2. If a message sequence number is the same as the previous message, the receiver interprets this as a repeated message and ignores the new message. Only the message sequence number needs to be checked by the receiver to determine whether the remainder of the buffer needs to be processed. There is no need to reset or clear a message buffer before it is retransmitted. It can simply be retransmitted and the message is ignored after the first time it is sent.

3. The DM299 and host transmitters use independent message sequence numbers.

4. The DM299 and host transmitters and receivers must be in sync, such that the message sequence number is always the first word in a buffer, both during transmission and reception.

6.2 SPI Handshake Protocol

The flow diagram below shows the flow of message data and interrupts between the host and the DM299.

1. Either the DM299 or the host can initiate the message transfer by interrupting the other, but only the host can generate the SPI clocks to actually execute the transfer.
2. Some of the message and buffer preparations can be performed in parallel.
3. A race condition is created when the host and the DM299 interrupt each other simultaneously. This condition results in messages being sent simultaneously by both parties the next time the host turns on the SPI clock. Depending on the order of the interrupts and latencies, the DM299 may request the host to turn on the SPI clocks one additional time. During this time either party will need to repeat the previously sent message or essentially invalidate the message by not incrementing the sequence number. The DM299 message buffers are always protected because the host turns on the SPI clocks only upon receipt of an interrupt from the DM299 and the DM299 will always have a receive buffer ready before sending the interrupt.

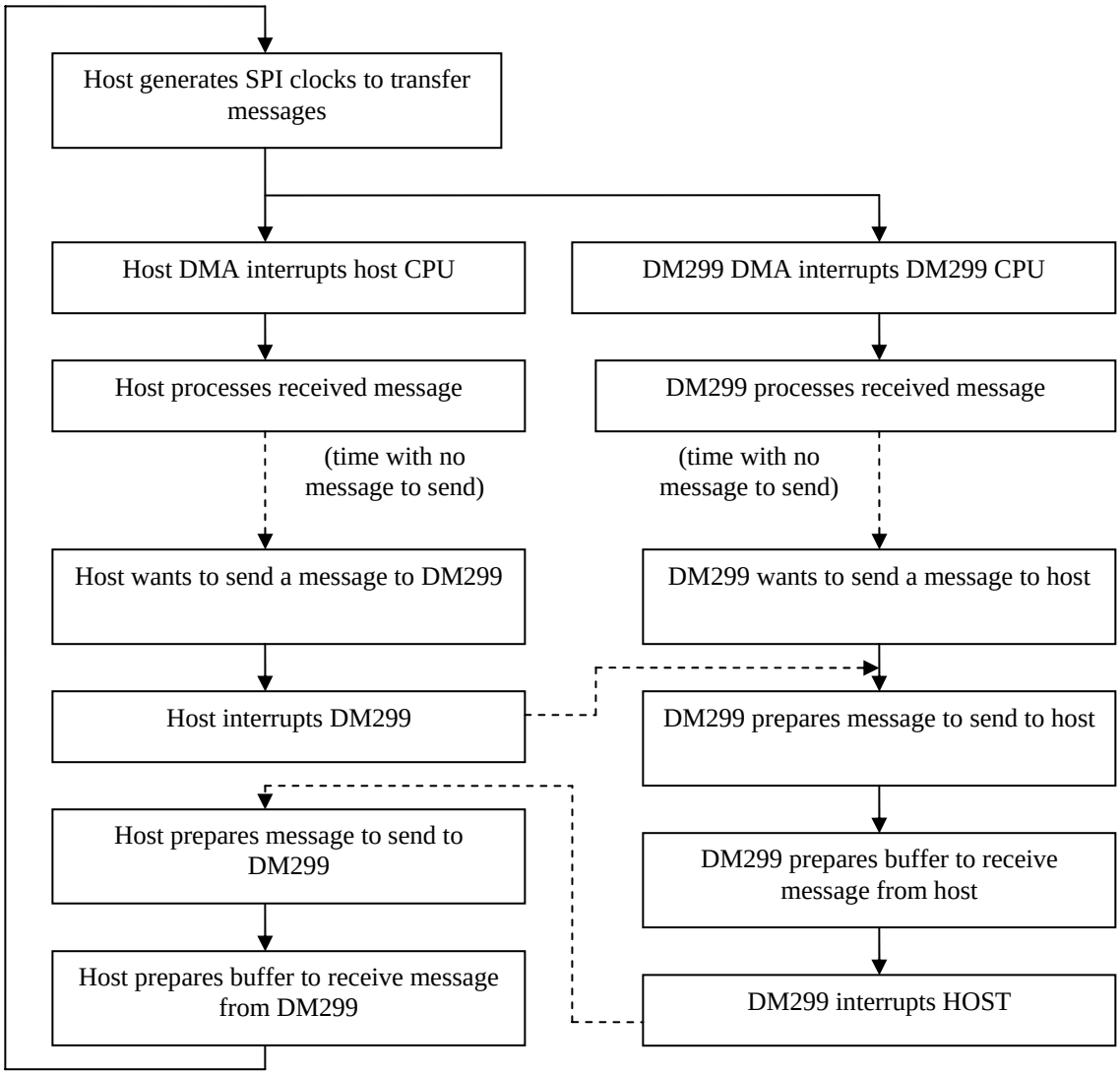


Figure 7 – SPI messaging flow diagram

6.3 Data and Command Channels

Messages are classified into command and data messages. Data messages are messages used to transmit or receive bulk data. All messages other than data messages are command messages. Acknowledgements to command and data messages are called command and data acknowledgements respectively. Command messages are used to initiate action, change configuration, stop an action, etc. There is only one command channel and it is open at all times. There can be none or up to two data channels open at any time**. All logical channels use the same SPI physical channel but each message received through SPI is assigned to a logical channel based on the message ID and sender.** (See Figure 8 below)

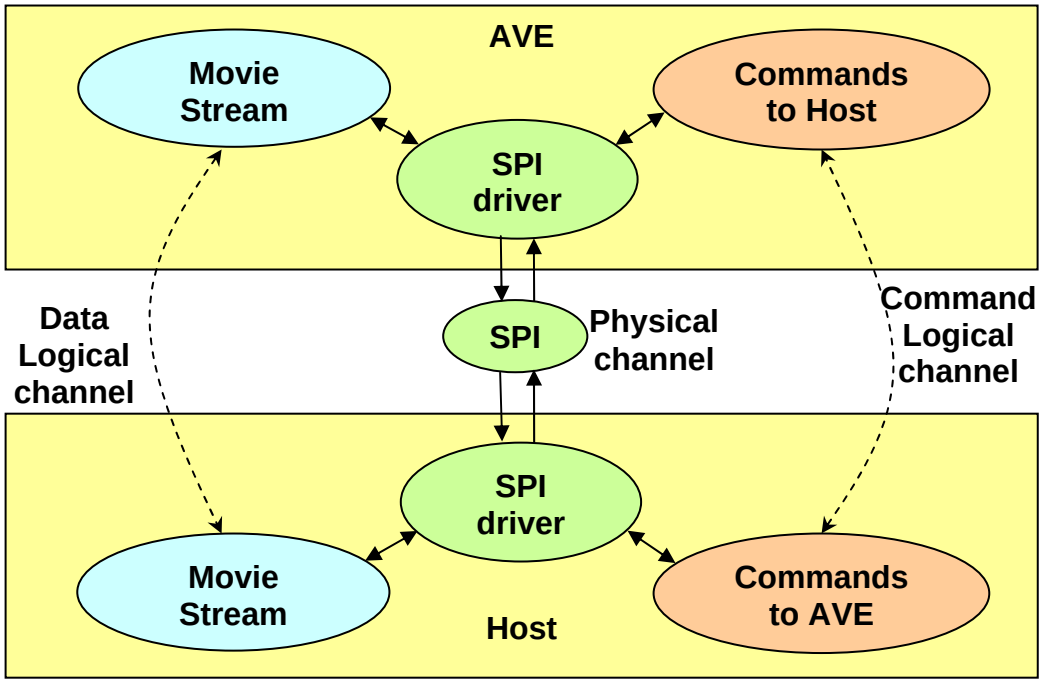


Figure 8 – SPI logical channel

6.3.1 Command Logical Channel

There is no message queue in the command channel. Each command message is processed and acknowledged before another command message can be sent. A new command message cannot be sent by a party unless that party is in receipt of the acknowledgement to the previous command message.

Note that command logical channel is always open. Logical data channels are opened/closed implicitly using command channel. For example, the Host sends command to the DM299 to start encoding. This opens Data Logical channel that transfers encoded data from the DM299 to the Host. When the Host commands the DM299 to stop encoding and the DM299 acknowledges, the data logical channel is closed. The same scenario is true for decoder application.

6.3.2 Data Logical Channel

A data channel is established implicitly when a command from the host necessitates a high throughput data transfer mechanism. For instance, the command from host that starts video encoding on the DM299 will necessitate a data channel that carries encoded data from the DM299 to the host. A data channel is closed implicitly when a command from the host obviates the need for a high throughput data transfer mechanism.

The 'commander' in a channel – that is to say the side which issues CAM_SEND_DATA or CAM_REQUEST_DATA – will act according to these rules:

- It has the right to send message even if the previous, but only the previous, message hasn't been acknowledged;
- When one unacknowledged message is out, the next one will be sent in a transfer that is requested by the other side. Acknowledgers of CAM_SEND_DATA will keep sending one acknowledge for each CAM_SEND_DATA they receive, but they must be ready to accept two consecutive CAM_SEND_DATA's.

A data channel has a message queue of length two. This means that the message originator can send up to two data messages without receiving an acknowledgement. This means both parties must have at least two transmit and two receive buffers exclusively for this channel, which increases the throughput of the data channel.

Acknowledgers of CAM_REQUEST_DATA will keep sending one acknowledge for each CAM_REQUEST_DATA they receive until data channel is closed thru command channel. Even when the Host has no more data (end-of-stream), it should continue to acknowledge CAM_REQUEST_DATA with zero length packets.

Typically the 'commander' is DM299, because it is presumed that it has low resources. Please note that Host should be prepared to accept, in worst case, three unacknowledged packets – two from data logical channel and one from command logical channel.

Each data channel has a message originator and a data originator. The message originator is the party that initiates the data message. The data originator is the party that supplies the data. The message originator and the data originator are the same party when data is pushed or sent, and they are different when data is requested.

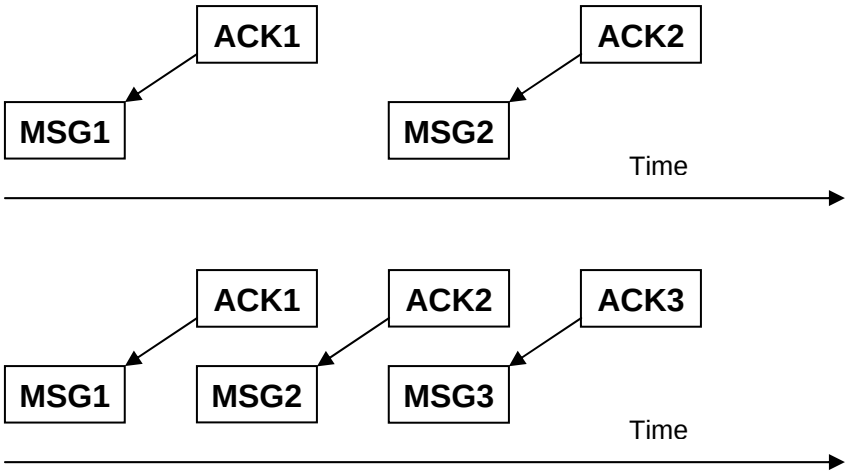


Figure 9 – Control Message Ack Flow (top) versus Data Message Ack Flow (bottom)

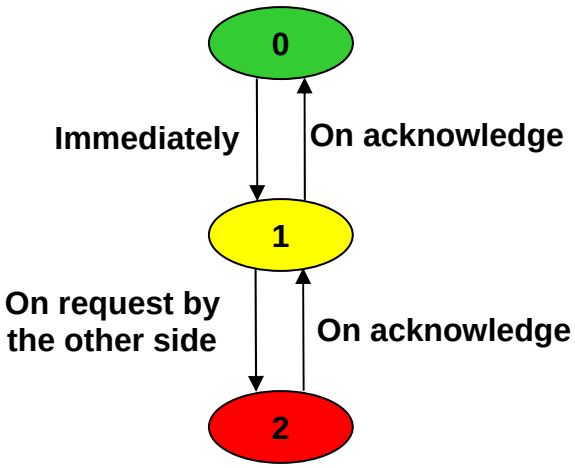


Figure 10 – Behavior of data commanders state diagram

The states '0', '1' and '2' mean 0, 1 or 2 unacknowledged messages are out, respectively.

Once again, the state of a channel is affected only by messages from that particular channel. This means that each logical channel has mechanism to regulate speed of data flow independently form other logical channels. Also each side can regulate data speed by delaying messages/acknowledges related to particular logical channel.

6.4 Message Content

The message can be broken into the header and payload sections.

6.4.1 Message Header

Each message includes the following header

```
typedef struct
{
        UData32     message_seq_num;
        UData16     message_id;
        UData16     message_status;
        UData32     message_size;
        UData32     message_time;
        UData32     message_checksum;
} CAM_HEADER;
```

- *message_seq_num* is used to uniquely identify incoming and outgoing messages
    - See SPI Messaging Rules in section 6.1 for more details on the sequence number field
- *message_id* indicates the type of message
    - Each message type has a unique *message_id*
    - An acknowledgement has a different *message_id* than the message being acknowledged, probably different only by the MSB being set for acknowledge
- *message_status* indicates the status of the message processing in the acknowledgment message.
    - This field can remain uninitialized in the original message.
    - If an acknowledgement does not use this field, no return value is indicated for that message in the message description
- *message_size* indicates the size, in bytes, of the entire message
    - For a particular *message_id,* the *message_size* is usually fixed but can vary such as when the DM299 is sending encoded data to the host in variable block sizes
- *message_time* indicates the clock value when this message is being sent to the host
    - Not all applications will need to support this feature
        - Unless support of this feature is indicated in the application-specific documentation, the application does NOT support this feature

        - If an application does not support this feature, it must send 0x00000000 for this value
    - If an application does support this feature
        - The time interval per tick of this value is application dependent
        - The application must update the clock value at the specified interval
        - The application must send the clock value with each message and acknowledgement that it sends to the host
    - The host is not expected to provide a value in this field and can leave it un-initialized
    - This feature can be used to profile, benchmark, and/or verify performance
- *message_checksum could be used in future releases for error detection. This field is currently unused*
    - The DM299 will ignore any checksum provided in a message from the host
    - The DM299 will provide a checksum of 0x00000000 in the bootloader, and a checksum of 0xBAAEA7AD in each application

6.4.1.1 Message ID Structure

To achieve best performance and to simplify implementation a new structure for message IDs is proposed.

Message ID is 16-bit wide with the following fields:

A	I	Reserved				IDX		Message ID							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

A [1]	15	One bit wide. Indicate whether the message is acknowledge: 0 – not acknowledge message; 1 – acknowledge message. Used for messages with payload. This bit has same functionality as the previous messaging protocol.
I [1]	14	One bit wide. Indicate whether the message is external (SPI) or internal (framework). It will be 0 for all external messages.
IDX [2]	9-8	Two bit wide. Logical channel number. (4 channels max or 3 data

		channels max)
Message ID [8]	7-0	Eight bit wide. Contains the message ID. This field has same functionality as the previous messaging protocol.

6.4.2 Message Payload

Any data that is specific to the *message_id* is appended to the header. The complete form of each message is as shown.

```
typedef struct
{
        struct          CAM_HEADER;
        UData8          message_content[];
} [insert message name here]_TYPE.;
```

- *message_content[]* contains any *message_id*-specific data
    - A checksum can be embedded within the data in *message_content[]* if desired
    - *message_content[]* can also be defined as a structure or a number of structure elements for clarity. Care must be exercised in structure definition that elements greater in length than UData8 are properly aligned for proper addressing.
    - The *message_content[]* array size is nominally 512 or 1996 bytes. The payload size is determined at build time, and should be confirmed in the host prior to integration of the host with the DM299.