Freescale Semiconductor
Application Note

Document Number: AN4755
Rev. 0, 06/2013

CAN Booting Procedure for Vybrid Controller Solutions

By: Biswanath Pal Bhowmick, Khushboo Gupta

Automotive and Industrial Solutions Group

Contents

1 Introduction

The Vybrid family features a heterogeneous dual-core solution that combines the ARM Cortex® -A5 and ARM Cortex® -M4 cores. The family also features dual USB 2.0 OTG controllers with integrated PHY, dual 10/100 Ethernet controllers with L2 switch, 1.5 MB of on-chip SRAM and a rich suite of communication, connectivity and human-machine interfaces (HMI). The VF6xx family is pin- and software-compatible with the VF[3,5,6] families. Vybrid VF6xx devices include multiple serial interfaces including UARTs with support for ISO7816 SIM/smart cards, SPI and I2C and dual CAN modules. VF6xx devices can interface to a variety of external peripherals and memories for system expansion and data storage. See the Vybrid factsheet located at www.freescale.com for complete feature list.

Vybrid family supports the downloading of boot image through various interfaces e.g., SD/MMC, FlexBus, SPI/I2C based EEPROM, QuadSPI, CAN, UART, USB. To facilitate this, a security feature enabled High Assurance Boot (HAB) code is implemented in the Vybrid boot ROM area. At power on reset, the program execution starts from the internal boot ROM area. Depending upon the boot mode and

configuration settings, it can boot through any one out of the aforementioned interfaces.

For automotive use cases, the CAN interface can be used to download a boot image from a host, having CAN channel connected to Vybrid through a CAN network. This is a suitable way to download a boot image in the OEM production house (particularly when other conventional serial boot interface like USB and UART are not available in an embedded system). Using serial CAN boot interface, a host can initially download and write the final boot image to an on-board resident NVM (non-volatile memory) like EEPROM connected through Quad-SPI interface. Once the final image is written to the EEPROM, further booting can occur directly from the EEPROM.

In this application note, the hardware setup and software algorithmic approach to download a boot image from the CAN-host to Vybrid CAN-device is explained. Various applications using CAN interface to download boot image can make use of this application note. The following section will describe the details and the steps for preparing the hardware setup and writing host application to download boot image through the Vybrid CAN interface.

Readers of this application note should have some basic knowledge of the FlexCAN module and the CAN standard ISO11898.

2 Hardware setup description

The simple hardware setup for downloading the boot image over Vybrid CAN interface is shown in Figure 1:

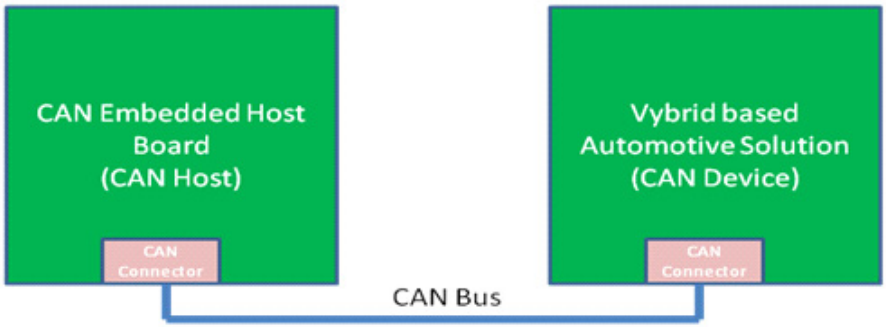

Figure 1. Hardware setup for downloading boot image through Vybrid CAN interface

A CAN-device (or simply “device”) is the board/target where the boot image needs to be downloaded. This can be done by any embedded CAN-host board (also simply “host”) containing any standard CAN controller.

In general, there is a very simple CAN hardware interfacing requirement. Vybrid has internal FlexCAN controller which is connected to a standard CAN PHY interface, typical schematic diagram is shown in Figure 2. The host may have similar hardware interface, allowing the communication of the host and the Vybrid device connected over CAN bus.

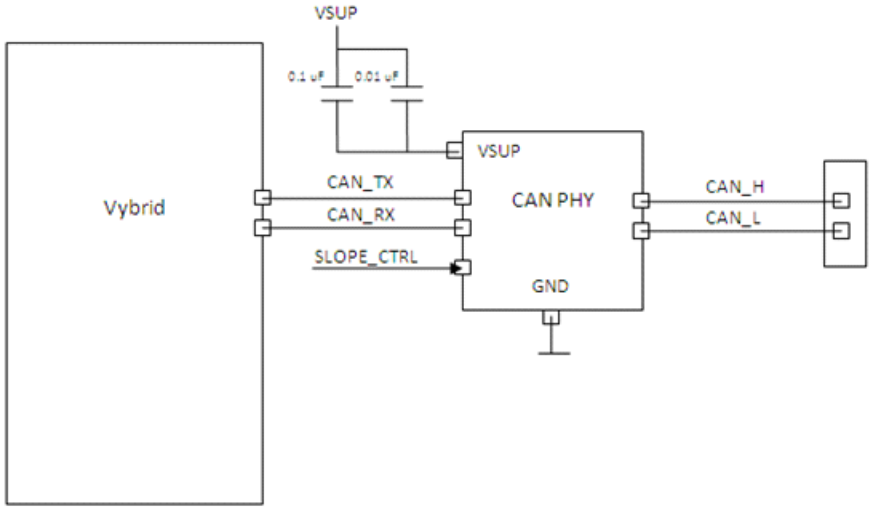

Figure 2. CAN hardware block schematic

3 Software design

3.1 Baud rate requirements and settings

The FlexCAN module is configured by the boot ROM to operate at 1 Mbps. Table 1 lists the bit timing parameters which are programmed in FlexCAN during boot process (as part of the boot ROM code).

Table 1. Bit timing parameters in FlexCAN

FlexCAN Parameter	Value (in Time Quanta)	CAN2.0 Parameter	Value (in Time Quanta)
SYNC_SEC	1	SYNC_SEC = FlexCAN SYNC_SEC	1
PROP_SEG	3	PROP_SEG = FlexCAN PROP_SEG+1	4
PSEG1	2	PHASE_SEG1 = FlexCXAN PSEG1+1	3
PSEG2	3	PHASE_SEG2 = FlexCAN PSEG2+1	4

The baud rate of host and device need to be the same such that their TSEG1:TSEG2 ratio is also as equal as is possible.

See Figure 3 for details on the parameters being used to configure the baud rate.

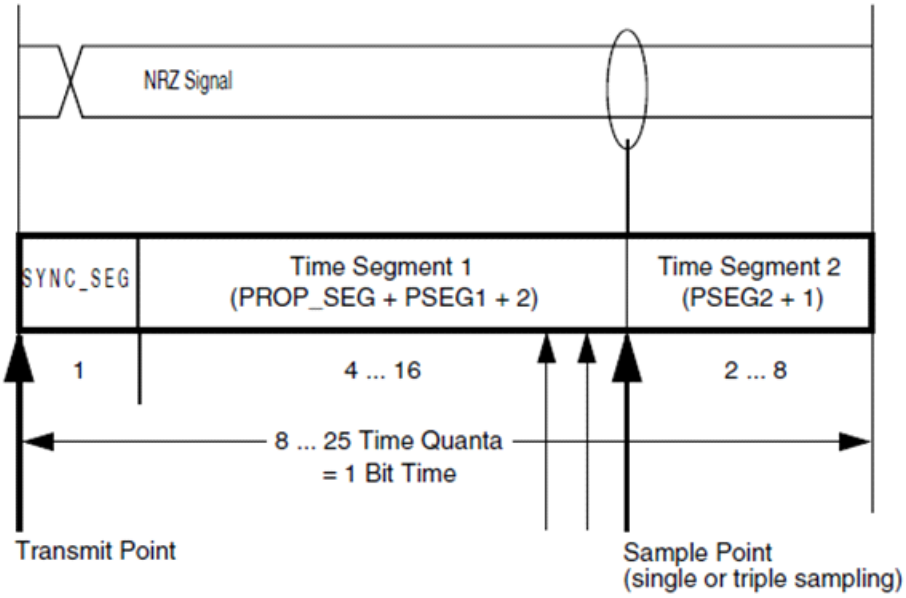

Figure 3. Segments within the Bit Time

Table 2 lists the CAN standard TSEG1 and TSEG2 settings.

Table 2. CAN standard compliant bit time settings

Time Segment 1 (TSEG1)	Time Segment 2 (TSEG2)	Resynchronization Jump Width (RJW)
5 .. 10	2	1 .. 2
4 .. 11	3	1 .. 3
5 .. 12	4	1 .. 4
6 .. 13	5	1 .. 4
7 .. 14	6	1 .. 4
8 .. 15	7	1 .. 4
9 .. 16	8	1 .. 4

Vybrid's FlexCAN module works as a device connected to an external CAN-host. The application software running in the external CAN-host should follow layered software architecture as described in Figure 4.

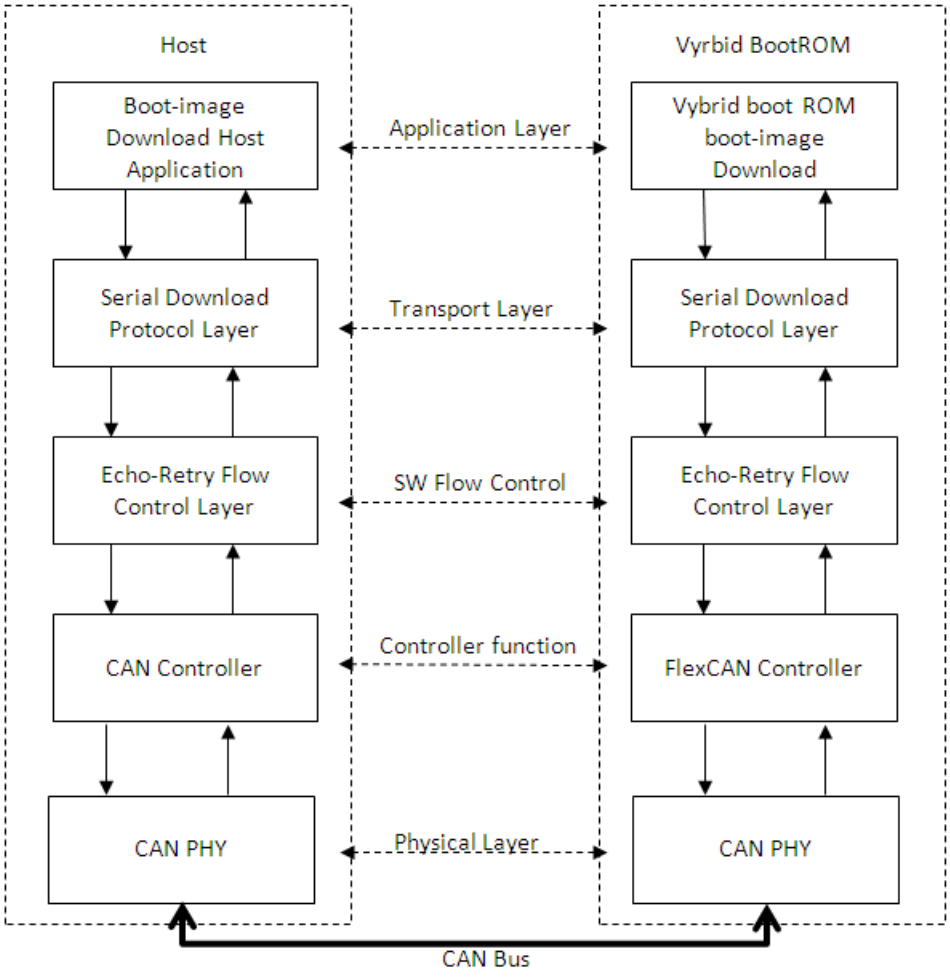

Figure 4. FlexCAN boot process layered architecture

From the bottom up, the CAN boot architecture is similar to a subset of OSI layer architecture consisting of physical, data link/flow control, transport and application layers only.

3.2 Serial download protocol

In Vybrid, boot image is downloaded through CAN interface using Serial Download Protocol (SDP). Using this protocol, the commands are sent from the CAN-host to CAN-device, which are responded by the device, with (optional) data and response field.

Table 3 lists all the commands used to facilitate serial download over CAN bus.

Table 3. 16-Byte SDP Command Structure

BYTE Offset	Size	Name	Description
0	2	COMMAND TYPE	Commands supported: • 0x0101 READ_REGISTER • 0x0202 WRITE_REGISTER • 0x0404 WRITE_FILE • 0x0505 ERROR_STATUS • 0x0A0A DCD_WRITE • 0x0B0B JUMP_ADDRESS
2	4	ADDRESS	Only relevant for: READ_REGISTER, WRITE_REGISTER, WRITE_FILE, DCD_WRITE, and JUMP_ADDRESS commands.[1]
6	1	FORMAT	Access format, 0x8 for 8-bit access, 0x10 for 16-bit and 0x20 for 32-bit access. Only relevant for READ_REGISTER and WRITE_REGISTER commands.
7	4	DATA COUNT	Size of data to read or write. Only relevant for WRITE_FILE, READ_REGISTER, WRITE_REGISTER and DCD_WRITE command. For WRITE_FILE and DCD_WRITE commands DATA COUNT is in byte units.
11	4	DATA	Value to write. Only relevant for WRITE_REGISTER command.
15	1	RESERVED	Not used in Vybrid ROM

[1] For READ_REGISTER and WRITE_REGISTER commands, this field is address to a register. For WRITE_FILE and JUMP_ADDRESS commands, this field is an address to internal or external memory address.

3.3 Flow control protocol

On top of FlexCAN module, there is an Echo-Retry Flow Control Layer which implements a software echo-retry flow control mechanism. This mechanism has a concept of echoing back the received data by the device which has been sent by a host. In this architecture, host sends the data as an originator coming from the upper layer, which is SDP here, and the SDP responses are sent back by device. This echo back mechanism ensures that the host doesn't send the next data to the device until it has received the echo of the data already transmitted over CAN bus.

Only Echo back mechanism is not enough for stable flow control, as SDP protocol has a command-response model, i.e. data can flow from host to device and vice-versa. Therefore, to facilitate the switching between sending and receiving functions without any data/response being missed by either entity, a Retry mechanism is implemented.

3.4 Host application

The host application to downloading boot image through serial CAN bus should be developed as per the layered model describe in Figure 4. The following is the sequence to be followed in the host application.

1. Initialize CAN-host interface as per the settings listed in Table 1.

2. Set the interface speed to 1 Mbps.
3. Send a packet with pattern 0x67898967 to the CAN-device. While sending any packet to the device, ensure the host ID is always set as 0x14. After sending the packet, configure the receive message buffer to receive a response packet from the CAN-device, ensuring that the device ID is always set to 0x04. Implement the echo-retry flow control mechanism for a system defined retry count.
4. Once the host receives back the same pattern 0x67898967 from the device, the host reaches a state called “associated”.
5. At this point, host can start boot image download by sending the SDP commands as stated in Table 3.
6. Optionally, the host can send a DCD_WRITE SDP command to the device. This is useful if the host wants to send DCD words to device in order to configure, for example, the SDRAM[1] before the boot image can be written to SDRAM memory in the device/target board. DCD_WRITE command will send the SDRAM initialization word sequences as per the SDRAM device in the target board, i.e., the one containing device.
7. Next, the host should send a WRITE_FILE SDP command to the device. The parameters of the WRITE_FILE command are:
   - COMMAND TYPE = 0x0404
   - ADDRESS=<an address of SDRAM or internal SRAM>
   - FORMAT=0x00
   - DATA COUNT=<size of boot image>
   - DATA=0x00000000
   - RESERVED=0x00

In the following pseudo code, the host application prepares a receive message buffer before preparing and sending a transmit message buffer (calling function config_rx_buf). This is done in order to keep the host ready to receive an echo response of WRITE_FILE packet, guaranteeing the reception of echo without fail. It’s possible that the device will fail to receive WRITE_FILE packet at the first attempt (because of its own execution context or delay), so a retry mechanism is helpful to take care of the device missing packet. Instead of an indefinite number of retries, a retry count can be utilized.

```
step("Configure RX buffer and send 16 bytes of WRITE_FILE command and check for echo.");

/*Transmitting 1st part of the WRITE_FILE command*/

/*Configure and keep ready a RX buffer before transmission is done*/

config_rx_buf(rx_buf, rcv_message_id);

/* Write the sdp_packet (0x0404 3F040000 00 00), for WRITE_FILE packet */

sdp_packet[0] = 0x04; sdp_packet[1] = 0x04;                    /* COMMAND TYPE */

sdp_packet[2] = 0x3F; sdp_packet[3] = 0x04; sdp_packet[4] = 0x00;  sdp_packet[5] = 0x00;
/* ADDRESS */

sdp_packet[6] = 0x00;  /* FORMAT */

sdp_packet[7] = 0x0;   /* DATA COUNT - 1st byte */
```

1. DCD is not limited to only SDRAM set-up.

```
tx_done=0;
while(tx_done==0) {  /*Keep polling until the packet is echoed*
     /* Configure MB to transmit to BOOT device (DUT)
     config_tx_buf(tx_buf, send_message_id, sdp_packet, 8);
     transmit_and_wait_for_tx_flag(tx_buf, 8);

     /*Wait to finish the ECHOED RX transfer*/
     if(wait_for_rx_flag_and_chk_rcvd_data(rx_buf, sdp_packet, 8))
     tx_done=1;
           else
                      printf("Retransmitting first part of WRITE FILE command!");
     }


/*Transmitting 2nd part of the WRITE_FILE command*/
/*Configure RX buffer again*/
config_rx_buf(rx_buf, rcv_message_id);
/* Write the data (0x001000 00000000 00), for WRITE_FILE packet */
sdp_packet[0] = 0x00;  sdp_packet[1] = 0x10; sdp_packet[2] = 0x00;  /*DATA COUNT - 2nd
to 4th bytes*/
sdp_packet[3] = 0x00;  sdp_packet[4] = 0x00; sdp_packet[5] = 0x00;  sdp_packet[6] = 0x00;
/*DATA*/
sdp_packet[7] = 0x00;  /* RESERVED */
tx_done=0;
while(tx_done==0) {  /*Keep polling untill the packet is echoed*/
     /* Configure MB to transmit to BOOT device (DUT)*/
           config_tx_buf(tx_buf, send_message_id, sdp_packet, 8);
           transmit_and_wait_for_tx_flag(tx_buf, 8);
           /*Wait to finish the ECHOED RX transfer*/
           if(wait_for_rx_flag_and_chk_rcvd_data(rx_buf, sdp_packet, 8))
                                                                  tx_done=1;
           else
                     printf("Retransmitting second part of WRITE FILE command!");
     }
```

Immediately following the WRITE_FILE command, the host should send the entire image data to the Vybrid device (keeping echo-retry flow control mechanism intact as describe above). In the below case, the image size is known to be 4 Kbytes only.

```
step("Send boot image (of size 4Kbytes in this case) data");
for(loop=0; loop<4096; loop=loop+8) {
```

```
/* Write the boot image file data to CAN */
/* file_data is the pointer where boot image is preloaded */
      data[0] = file_data[0+loop];  data[1] = file_data[1+loop];
      data[2] = file_data[2+loop];  data[3] = file_data[3+loop];
      data[4] = file_data[4+loop];  data[5] = file_data[5+loop];
      data[6] = file_data[6+loop];  data[7] = file_data[7+loop];

          /*Configure RX buffer again*/
          config_rx_buf(rx_buf, rcv_message_id);
          tx_done=0;
          while(tx_done==0) {  /*Keep polling untill the packet is echoed*/
             /* Configure MB to transmit to BOOT device (DUT) */
                config_tx_buf(tx_buf, send_message_id, data, 8);
                transmit_and_wait_for_tx_flag(tx_buf, 8);

                /*Wait to finish the ECHOED RX transfer*/
                if(wait_for_rx_flag_and_chk_rcvd_data(rx_buf, data, 8))
                             tx_done=1;
                else
             printf("Retransmitting WRITE FILE image data!");
            }
        }
printf("4K data SENT");
```

After the entire boot image data is successfully sent to the Vybrid device (and being ensured by the underlying flow control mechanism), host then expects to receive the response packets from the device.

- 4 bytes HAB mode indicating Production/Development part
  - — 0x56787856 for a development level Vybrid part
  - — 0x12343412 for a production level Vybrid part
- 4 bytes command complete code 0x88888888

```
/*Configure RX buffer*/
      config_rx_buf(rx_buf, rcv_message_id);
      step("Expect 16 bytes response to WRITE_FILE and check for echo.");
wait_for_rx_flag_and_chk_response(rx_buf, tx_buf, data, send_message_id, rcv_message_id,
0x56787856, 0x12343412, 0x88888888, 4);
```

Next, the host should send a JUMP SDP command to the device. The parameters of the WRITE_FILE command are:

— COMMAND TYPE = 0x0B0B

- ADDRESS=<same address specified in previous WRITE_FILE command>
- FORMAT=0x00
- DATA COUNT=0x00
- DATA=0x00000000
- RESERVED=0x00

```
printf("Sending JUMP SDP command...");

printf("Sending 1st part of JUMP SDP command...");

step("Configure RX buffer and send 16 bytes of JUMP command and check for echo.");

/*(Configure RX buffer*/

config_rx_buf(rx_buf, rcv_message_id);

/* Write the data (0x0B0B 3F040000 00 00), for a JUMP SDP packet */

sdp_packet[0] = 0x0B; sdp_packet[1] = 0x0B; /* COMMAND */

sdp_packet[2] = 0x3F; sdp_packet[3] = 0x04; sdp_packet[4] = 0x00;  sdp_packet[5] = 0x00;
/* ADDRESS */

sdp_packet[6] = 0x00;  /* FORMAT */

sdp_packet[7] = 0x0;   /* RESERVED */

tx_done=0;

while(tx_done==0) {  /*Keep polling untill the packet is echoed*/
    /* Configure MB to transmit to BOOT device (DUT)*/
          config_tx_buf(tx_buf, send_message_id, sdp_packet, 8);
          transmit_and_wait_for_tx_flag(tx_buf, 8);
          /*Wait to finish the ECHOED RX transfer*/
          if(wait_for_rx_flag_and_chk_rcvd_data(rx_buf, data, 8))
                                                                  tx_done=1;
          else
                        printf("Retransmitting first part of JUMP command!");
    }


printf("Sending 2nd part of JUMP SDP command...");

/*Configure RX buffer*/

config_rx_buf(rx_buf, rcv_message_id);

/* Write the data (0x00000000 00000000), if the frame is a data frame */

sdp_packet[0] = 0x00;  sdp_packet[1] = 0x00;  sdp_packet[2] = 0x00; /*DATA COUNT - 2nd
to 4th bytes*/

sdp_packet[3] = 0x00;  sdp_packet[4] = 0x00;  sdp_packet[5] = 0x0;

sdp_packet[6] = 0x0;  /* DATA */

sdp_packet[7] = 0x0;  /* RESERVED */
```

```
tx_done=0;
while(tx_done==0) {  /*Keep polling untill the packet is echoed*/
    /* Configure MB to transmit to BOOT device (DUT)*/
          config_tx_buf(tx_buf, send_message_id, data, 8);
          transmit_and_wait_for_tx_flag(tx_buf, 8);
          /*Wait to finish the ECHOED RX transfer*/
          if(wait_for_rx_flag_and_chk_rcvd_data(rx_buf, data, 8))
                                                                    tx_done=1;
          else
                        printf("Retransmitting first part of JUMP command!");
    }
```

After receiving the JUMP command correctly, the device sends back a response of command received with the 4 byte part type:

- 0x56787856 for a development level Vybrid part
- 0x12343412 for a production level Vybrid part

The device then authenticates the image previously received through WRITE_FILE command.

If the authentication is successful, it branches the program execution to the address mentioned in ADDRESS field.

If the authentication is unsuccessful, device sends back the 4 byte error code to host which comprises 1 byte reserved field, 1 byte context, 1 byte reason and 1 byte of status codes for Vybrid boot ROM code.

```
config_rx_buf(rx_buf, rcv_message_id);
step("Expect 16 bytes response to JUMP and check for echo.");
wait_for_rx_flag_and_chk_response(rx_buf, tx_buf, data, send_message_id, rcv_message_id,
0x56787856, 0x12343412, 0x0, 4);
printf("FLEXCAN BOOT COMPLETE");
```

This eventually completes the download and execution of boot image through FlexCAN interface.

4 References

1. *Vybrid Reference Manual* (VYBRIDRM)

How to Reach Us:

Home Page:
freescale.com

Web Support:
freescale.com/support

Document Number: AN4755
Rev. 0
06/2013