Freescale Semiconductor
Application Note

Document Number: AN3690
Rev. 0, 06/2008

CAN-to-USB Bridge

by: William Jiang & Daniel Uribe Rodriguez
MSG Applications Engineering
Asia Pacific & North America

1 Introduction

Universal Serial Bus (USB) is a low-cost, fast, bi-directional, isochronous, dynamically attachable serial interface that is consistent with the requirements of the PC platform of today and tomorrow. It is widely used in the PC connection world, as the term “universal” implies.

Freescale’s MCF51JM128 devices integrate a USB On-The-Go (OTG)-capable controller and Freescale’s Controller Area Network (MSCAN) module. Thanks to the USB OTG controller, these devices can be used as a USB host, a USB full-speed device, or a dual-role OTG peripheral. Because of the MSCAN module, these devices are widely used in industrial automation and automotive applications, as well as other areas that require real-time processing, reliable operation in the EMI environment, and wide bandwidth.

Meanwhile, Freescale offers the complementary CMX USB-Lite Stack software for these devices, which can be downloaded for free from www.freescale.com. This

Contents

USB solution makes MCF51JM128 devices easy to use, and enables products that use them to have a rapid approach to market.

This application note first gives an overview of the USB OTG module and the MSCAN module as well as the CMX USB-Lite Stack, then describes the CAN-to-USB bridge demo application. With the CAN-to-USB bridge, any CAN message can be transferred via USB to the PC and displayed on a PC graphical user interface (GUI) or logged to a file. A CAN message can also be sent from the PC GUI of the bridge to the CAN network, via USB. This makes it easy to view the CAN events and isolate any issues on the CAN network.

2 USB OTG Module

This section is an overview of the USB OTG module. For more detailed information, please refer to the Freescale document *MCF51JM128RM*, the MCF51JM128 reference manual.

The USB OTG module is a USB Dual-Mode (DM) controller. The OTG implementation in this module provides limited host functionality, as well as device FS solutions for implementing a USB 2.0 full-speed/low-speed compliant peripheral. The OTG implementation supports the On-the-Go (OTG) addendum to the USB 2.0 specification. Only one protocol can be active at any time. A negotiation protocol must be used to switch to USB host functionality from a USB device. This is known as the Host Negotiation Protocol (HNP).

The main features of the USB OTG module are:

- USB 1.1 and 2.0 compliant full-speed device controller
- Sixteen bi-directional endpoints
- DMA and FIFO data stream interfaces
- Low power consumption
- On-the-Go protocol logic

3 MSCAN Module

The MSCAN module is a communications controller implementing the CAN 2.0 A/B protocol defined in the Bosch specification dated September 1991.

MSCAN uses an advanced buffer arrangement, resulting in predictable real-time behavior and simplified application software.

The basic features of the MSCAN are:

- Implementation of the CAN protocol — version 2.0 A/B
  - — Standard and extended data frames
  - — Zero to eight bytes data length
  - — Programmable bit rate up to 1 Mbps
  - — Support for remote frames
- Five receive buffers with FIFO storage scheme
- Three transmit buffers with internal prioritization using a “local priority” concept

- Flexible maskable identifier filter supporting two full-size (32-bit) extended identifier filters, or four 16-bit filters, or eight 8-bit filters
- Programmable wakeup functionality with integrated low-pass filter
- Programmable loopback mode supports self-test operation
- Programmable listen-only mode for monitoring CAN bus
- Programmable bus-off recovery functionality
- Separate signalling and interrupt capabilities for all CAN receiver and transmitter error states (warning, error passive, bus-off)
- Programmable MSCAN clock source, either bus clock or oscillator clock
- Internal timer for time-stamping of received and transmitted messages
- Three low-power modes: sleep, power down, and MSCAN enable
- Global initialization of configuration registers

For more information on the MSCAN module, please refer to *MCF51JM128RM*, the MCF51JM128 reference manual.

4 CMX USB Stack

The CMX USB stack is developed by CMX Systems, Inc. It can be downloaded from the Freescale website (www.freescale.com/usb) with the search keyword “MCF51JM_USB_LITE_CMX.”

The CMX USB Stack includes three layers: the USB driver, the class drivers, and the USB applications.

The USB driver sits on the USB controller, manages the USB protocol and standard USB device requests, and will report to the upper-layer class drivers if an event occurs through callback routines that are defined by upper-layer class drivers. It also requests user-defined USB descriptors via callback routines.

The HID class driver manages the HID protocols. The CDC class driver manages the communication class protocols by implementing the Abstract Control Model Serial Emulation (refer to Section 3.6 of CDC specification version 1.1).

The USB applications use the services provided by the class drivers to implement application-specific functions based on the USB communication link. This stack provides some example applications to facilitate the use of all drivers. Typical examples are an HID mouse, a keyboard, and a USB-to-UART bridge.

The USB applications can also directly call the USB driver to implement the vendor-specific protocols, such as the CAN-to-USB bridge protocol.

For more information on the CMX USB Stack, please refer to Freescale document AN3492, “USB and Using the CMX USB Stack.” The Application Programming Interface functions and internal process of the CMX USB stack are similar to those described in AN3492.

5 CAN-to-USB Bridge

As shown in Figure 1, the CAN-to-USB bridge can be used as an event viewer or event logger in a wide variety of applications that require robust and cost-effective real-time processing.

This section describes the CAN-to-USB bridge architecture, CAN-to-USB bridge communication protocol, and firmware design.

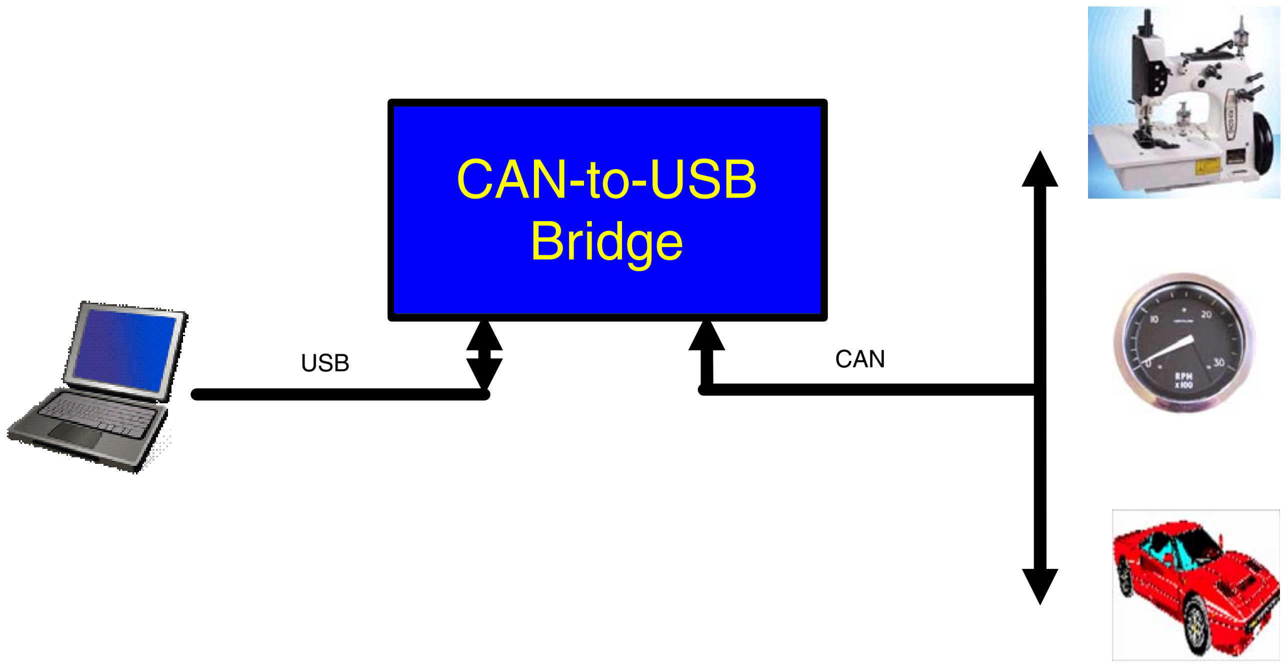


Figure 1. CAN-to-USB Bridge Application

5.1 CAN-to-USB Bridge Architecture

Figure 2 shows the architecture of the CAN-to-USB bridge. The bridge sits on top of the drivers. It receives commands from the USB host by calling the USB driver, then interprets and manages the commands. It also transmits the CAN message received from the CAN network by calling the MSCAN driver, and then bridges the message to the host via USB by calling the USB driver.

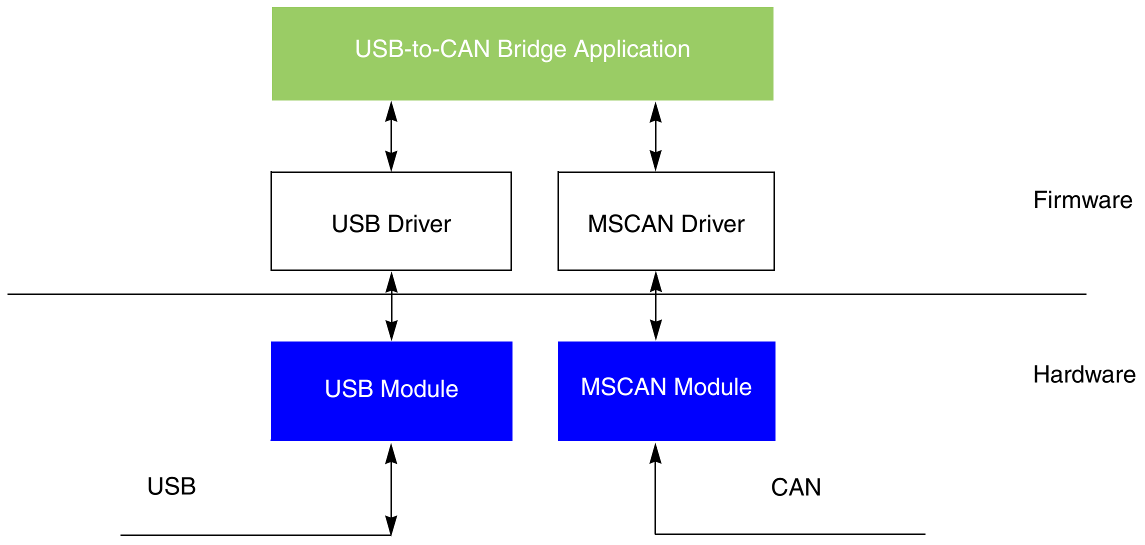


Figure 2. CAN-to-USB Bridge Architecture

5.2 CAN-to-USB Bridge Communication Protocol

The USB communication between host and device uses two types of endpoints (pipes): one is used to transfer data to the device (OUT endpoint) and the other is used to send data to the host (IN endpoint). Both types of endpoints support configuration and application data. Endpoint 0 is also used for USB enumeration and control as defined in the USB standard.

Endpoint 1 and endpoint 2 are 16-byte bulk endpoints. Endpoint 3 and endpoint 4 are 32-byte bulk endpoints.

Endpoint 1 and endpoint 3 are used for transferring commands and data to the device from the host. Endpoint 2 and endpoint 4 are used for transferring commands and handshake data to the host from the device. Figure 3 shows the pipes and their meanings.

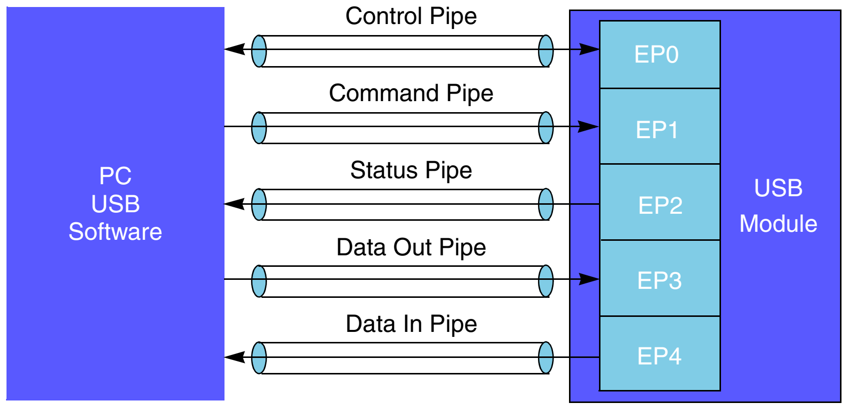


Figure 3. USB Pipes in CAN-to-USB Bridge

The bridge protocol packet in the pipe is organized into three fields, as shown in Figure 4.

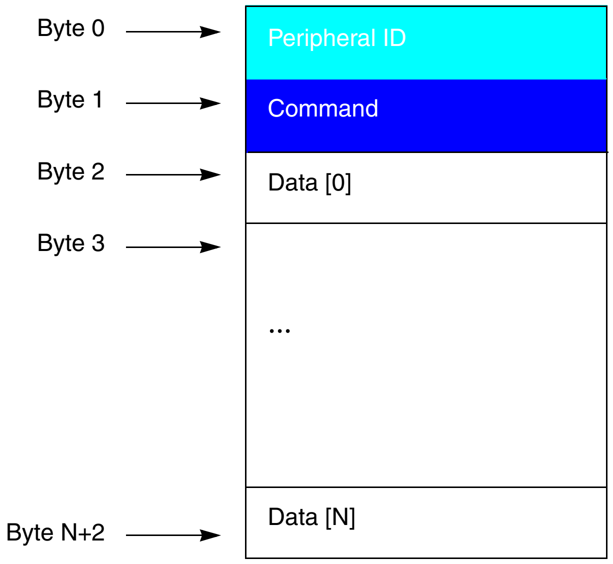


Figure 4. Protocol Packet Format

The first two bytes, containing both peripheral ID and command field, are called the protocol head. Table 1 describes each field of the protocol packet:

Table 1. Protocol Packet Fields

Protocol fields	Description
Peripheral ID	A unique identifier for the peripheral to be configured — for MSCAN, it is 8.
Command	The command code defines the action to be executed by the device or peripheral.
Data	This can be either the command parameters or the formatted data sent in response to a command. The size of the data varies depending on the command.

Table 2. Commands for Host-to-Device Communication

Command	Description	Data
0x00	CAN message to the device	Format: ID (4 bytes) + length (1 byte) + data (0 to 8 bytes) + priority (1 byte) ID format is binary and in Big Endian byte order mode: • Bit 31— ID flag (0 — standard ID, 1 — extended ID) • Bit 30 — remote frame flag (0 — data frame, 1 — remote frame) • Bit 29 — reserved — 0 • Bit 28:0 — ID (ID28–ID0 or ID10–ID0) For the detailed format of the CAN ID, see Figure 5. Length format: • Bit 7:4 — ignored • Bit 3:0 — byte count
0x01	Set CAN baud rate If this command from host has not been received, the device will run at the default baud rate, 1 Mbps.	Baud rate in Hz. It is a 32-bit binary value in Big Endian byte order mode. For example, if a 1 MHz baud rate is desired, the value will be 1,000,000 in decimal or 0x000F4240 in hexadecimal. It will be transmitted as 00 0F 42 40 in hexadecimal. Currently supported baud rates are: 1M, 500K, 400K, 250K, 100K, 50K, 25K, 10K.
0x02	Set CAN message filter to define which messages to receive If this command is not received, the device will receive all CAN messages.	Message filter format: Message mask code (32 bits) + message acceptance code (32 bits) Both message acceptance code and message mask code correspond to the ID on a bit-by-bit basis. The message mask code is defined as: 0 — Match the corresponding message acceptance code bit 1 — Ignore the corresponding message acceptance code bit Both codes are in hexadecimal format and ordered in Big Endian byte mode. Refer to MCF51JM128 Reference Manual figure 11-38, *32-Bit Maskable Identifier Acceptance Filter*, for the detailed bit format.
0x03	Abort the transmission of the pending CAN messages	None
0x04	Report CAN status code	None

Table 3. Commands for Device-to-Host Communication

Command	Description	Data
0x00	CAN message to the host	Format: ID (4 bytes) + length (1 byte) + data (0 to 8 bytes) + time stamp (16 bits — read directly from the time stamp registers in the device)
0x01	Baud rate configuration ACK	0xFF – configuration accepted 0x00 – configuration not accepted
0x02	CAN message filter configuration ACK	0xFF – configuration accepted 0x00 – configuration not accepted
0x03	Ack to abort the transmission of the pending CAN message	0xFF – configuration accepted 0x00 – configuration not accepted
0x04	Report CAN status code	CAN status code (32 bits): 0 (1 byte) + TX Error counter (1 byte) + RX Error counter(1 byte) + MSCAN Receiver Flag register content (1 byte) For more information about the bit details please refer to Chapter 11, “MSCAN,” in *MCF51JM128RM.*

This table shows the protocol packet content for command 0 from host to device. The protocol packet content for other commands can be derived from this table.

Table 4. Command 0 Packet: CAN Message to the Device

Byte Offset	Size in Bytes	Contents	Description
0	1	08	CAN peripheral ID
1	1	00	Command code
2	4	CAN ID	CAN ID
6	1	N	N stands for the length of the CAN data in bytes, and can be any value from 0 to 8.
7	N	CAN Data	CAN data
7+N	1	Priority	The priority of the CAN message can be 0x00 to 0xFF.

For example, if a CAN message will be sent to the device with a CAN ID of 0x0713, two data bytes with values of 0x11 and 0x22, and a priority of 0x86, then the protocol packet on endpoint 3 looks like this:

Byte Order	0	1	2	3	4	5	6	7	8	9
Packet	08	00	00	00	07	13	02	11	22	86

The CAN ID bit format is shown in Figure 5.

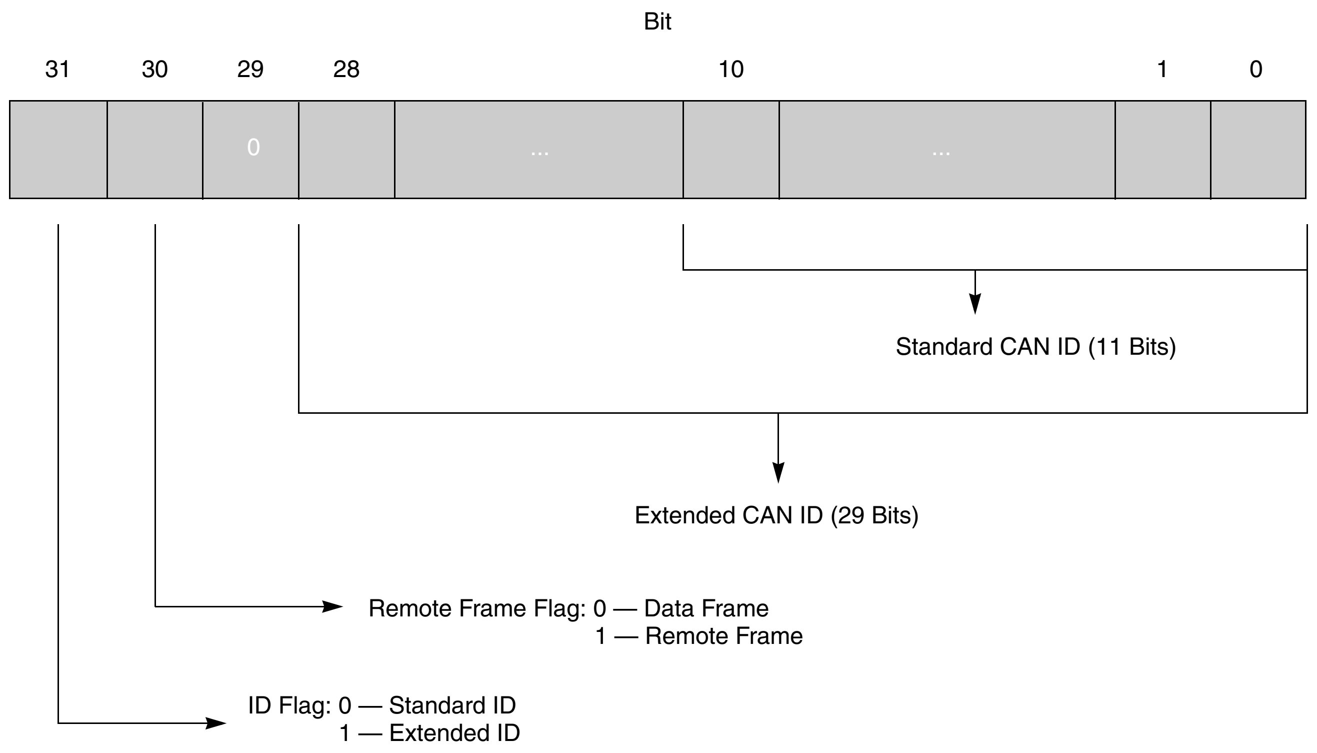

Bit
31
30
29
28
10
1
0
0
...
...
Standard CAN ID (11 Bits)
Extended CAN ID (29 Bits)
Remote Frame Flag: 0 — Data Frame
1 — Remote Frame
ID Flag: 0 — Standard ID
1 — Extended ID

Figure 5. CAN ID Format

These tables show the detailed endpoint usage.

Table 5. MSCAN Host-to-Device Endpoint Use

Pipe	Command	Description	Data
Endpoint 3	0x00	CAN message to the device	Format: ID (4 bytes) + length (1 byte) + data (0 to 8 bytes) + priority (1 byte)
Endpoint 1	0x01	Set CAN baud rate	Baud rate in Hz — 32-bit binary value in Big Endian byte order mode
Endpoint 1	0x02	CAN message filter configuration	Message mask code (32 bits) + message acceptance code (32 bits)
Endpoint 1	0x03	Abort the transmission of the pending CAN messages	None
Endpoint 1	0x04	Report CAN status code	None

Table 6. MSCAN: Device-to-Host Endpoint Use

Pipe	Command	Description	Data
Endpoint 4	0x00	CAN message to the host	Format: ID (4 bytes) + length (1 byte) + data (0 to 8 bytes) + time stamp (16 bits)
Endpoint 2	0x00	Acknowledge to the CAN message to the device command	0xFF – command accepted 0x00 – command not accepted
Endpoint 2	0x01	Set CAN baud rate ACK	0xFF – command accepted 0x00 – command not accepted
Endpoint 2	0x02	CAN message filter configuration ACK	0xFF – command accepted 0x00 – command not accepted
Endpoint 2	0x03	Abort the transmission of the pending CAN messages ACK	0xFF – command accepted 0x00 – command not accepted
Endpoint 2	0x04	Report CAN status code	CAN status code

5.3 CAN-to-USB Bridge Firmware Design

The firmware contains three main parts: the USB driver, the MSCAN driver, and the bridge application. It was developed with CodeWarrior 6.1.

The USB driver reuses the existing CMX USB Stack/USB driver to manage the USB protocol. The USB device enumeration is done by the USB driver through endpoint 0. Only one USB configuration is defined — it contains one interface with four endpoints. The direction of the endpoints and their usage are described in Section 5.2, “CAN-to-USB Bridge Communication Protocol.”

Because the bridge defines its specific protocol as described in Section 5.2, “CAN-to-USB Bridge Communication Protocol,” the bDeviceClass, bDeviceSubClass, and bDeviceProtocol fields of the USB device descriptor must all be set to 0xFF, and the bInterfaceClass, bInterfaceSubClass, and bInterfaceProtocol fields of the interface descriptor must also be set to 0xFF.

Because the device can be self-powered, the D6 bit in *bmAttributes* field of the USB configuration descriptor must be set.

Endpoint 4 will carry the real-time CAN message received on the CAN bus, so the polling interval for this endpoint must be the minimum value allowed by the USB specification. Therefore the polling interval is 1 ms.

All other endpoints have a longer polling interval value in their endpoint descriptors than endpoint 4 has in its descriptor. This is done to reduce the communications load on the USB bus.

Here are the definitions of USB descriptors for the bridge:

```
const hcc_u8 CANUSB_device_descriptor[] = {
  USB_FILL_DEV_DESC(0x0200,0xFF,0xFF,0xFF, EP0_PACKET_SIZE, CANUSB_VENDOR_ID,
CANUSB_PRODUCT_ID
      , CANUSB_DEVICE_REL_NUM, 1, 2, 3, 1)
};
const hcc_u8 CANUSB_config_descriptor[] = {
  USB_FILL_CFG_DESC(9+3+9+7*4, 1, 1, 4, (CFGD_ATTR_BUS_PWR|CFGD_ATTR_SELF_PWR), 50),
  USB_FILL_OTG_DESC(1, 1),
  USB_FILL_IFC_DESC(CANUSB_IFC_INDEX, 0, 4, 0xFF, 0xFF, 0xFF, 5), /* */
  USB_FILL_EP_DESC(0x1, DIR_OUT, TT_BULK, EP1_PACKET_SIZE, 0x10),      /* OUT endpoint */
  USB_FILL_EP_DESC(0x2, DIR_IN, TT_BULK, EP2_PACKET_SIZE, 0x10),
  USB_FILL_EP_DESC(0x3, DIR_OUT, TT_BULK, EP3_PACKET_SIZE, 0x10),      /* OUT endpoint */
  USB_FILL_EP_DESC(0x4, DIR_IN, TT_BULK, EP4_PACKET_SIZE, 0x1),
};
```

Where macros are defined as below:

```
#define USB_FILL_DEV_DESC(usb_ver, dclass, dsubclass, dproto, psize, vid,\
                          pid, relno, mstr, pstr, sstr, ncfg) \
  (hcc_u8)0x12u, STDD_DEVICE, (hcc_u8)(usb_ver), (hcc_u8)((usb_ver) >> 8)\
  , (hcc_u8)(dclass), (hcc_u8)(dsubclass), (hcc_u8)(dproto), (hcc_u8)(psize)\
  , (hcc_u8)(vid), (hcc_u8)((vid) >> 8), (hcc_u8)(pid), (hcc_u8)((pid) >> 8)\
  , (hcc_u8)(relno), (hcc_u8)((relno) >> 8), (hcc_u8)(mstr), (hcc_u8)(pstr)\
  , (hcc_u8)(sstr), (hcc_u8)(ncfg)
```

/* This macro will evaluate to an array initializer list with values of a configuration descriptor. */

```
#define USB_FILL_CFG_DESC(size, nifc, cfg_id, str_ndx, attrib, pow) \
  (hcc_u8)0x09u, STDD_CONFIG, (hcc_u8)(size), (hcc_u8)((size) >> 8)\
, (hcc_u8)(nifc), (hcc_u8)(cfg_id), (hcc_u8)(str_ndx), (hcc_u8)(attrib), (hcc_u8)(pow)
#define USB_FILL_OTG_DESC(hnp, srp) \
  (hcc_u8)3, 0x9, 0

/* Values for the attrib field of the configuration descriptor. */
/* Device is BUS powered. */
#define CFGD_ATTR_BUS_PWR  (1u<<7)
/* Device is self powered. */
#define CFGD_ATTR_SELF_PWR (1u<<6)
/* This macro will evaluate to an array initializer list with values of a
   interface descriptor. */
#define USB_FILL_IFC_DESC(ifc_id, alt_set, no_ep, iclass, isubclass, iproto, strndx) \
  (hcc_u8)0x09u, STDD_INTERFACE, (hcc_u8)(ifc_id), (hcc_u8)(alt_set), (hcc_u8)(no_ep)\
  , (hcc_u8)(iclass), (hcc_u8)(isubclass), (hcc_u8)(iproto), (hcc_u8)(strndx)
```

/* This macro will evaluate to an array initializer list with values of a endpoint descriptor. */

```
#define USB_FILL_EP_DESC(addr, dir, attrib, psize, interval) \
  (hcc_u8)0x07u, STDD_ENDPOINT, (hcc_u8)((addr)&0x7f) | (((hcc_u8)(dir))<<0x7)\
  , (hcc_u8)(attrib), (hcc_u8)((psize) & 0xff), (hcc_u8)(((psize) >> 8) & 0xff)\
  , (interval)
/* Endpoint Types
 */
#define TT_ISO    0x01
#define TT_BULK   0x02
#define TT_INTRP  0x03

/* Endpoint Direction
 */
#define DIR_OUT   0
#define DIR_IN   1
```

The next sections describe the USB driver, the MSCAN driver, and the bridge application firmware design.

5.3.1 USB Driver

The USB driver:

- Manages the USB OTG controller
- Sets up the USB buffer descriptor table (BDT) in the RAM space for transfers
- Monitors USB packets
- Encodes and decodes USB packets based on USB transactions and transfers
- Manages the standard USB device requests and USB device enumeration

This section describes some useful USB driver application programming interface routines (APIs) that will be called by the bridge application.

5.3.1.1 Prototype: hcc_u8 usb_init(void);

This function initializes the internal data structures of the USB driver, and also enables USB interrupts and the USB module. It returns 0 if successful, and 1 if unsuccessful.

5.3.1.2 Prototype: void usb_send(hcc_u8 ep, usb_callback_t f, hcc_u8* data, hcc_u32 tr_length, hcc_u32 req_length);

This function sets up a TX (IN) transfer on the given endpoint (ep) to send the specified number of data (tr_length) in byes to the host. The data to be transmitted is pointed to by the data parameter.

It will not wait for the transfer to be completed. The data will be transferred the next time the host requests data from the endpoint.

Because all packet transmissions on USB are started by the host, it needs to know how many bytes can be transferred during a transfer. Because of this, the host needs to tell the device how many bytes it can receive. This is specified by the req_length parameter. The device may have less data to be transmitted (tr_length).

If a transaction is complete and hence the endpoint buffer is empty, the USB driver will notify the user by calling the user callback routine passed by the parameter f. A user can set up this callback routine to flag the completion of a transaction, set the report state, and prepare more data to be sent. In general, it can remain as NULL.

5.3.1.3 Prototype: void usb_receive(hcc_u8 ep, usb_callback_t f, hcc_u8* data, hcc_u32 tr_length);

This function sets up an RX (OUT) transfer on the endpoint (ep) to receive the specified amount of data (tr_length) in bytes. The user must define a buffer to store the data received, which is pointed to by the data parameter. The size in bytes of the user-defined buffer is given by tr_length parameter, which must be the same amount as the host wants to send.

After the specified number of bytes is received from host by the USB driver, the callback function f is called.

5.3.1.4 Prototype: hcc_u8 usb_ep_is_busy(hcc_u8 ep);

This function checks the endpoint status for endpoint (ep). It returns nonzero if the endpoint is busy (a transfer is ongoing), and 0 otherwise.

5.3.1.5 Prototype: hcc_u8 usb_get_state(void);

This function returns the current USB state. See USBST_xxx in usb.h.

5.3.2 MSCAN Driver

The MSCAN driver manages the CAN protocol. It initializes and configures the MSCAN module, then transmits and receives CAN messages. It uses the interrupt method for receiving and the polling method for transmitting.

It provides these application programming interface (API) functions:

```
void can_config_bit_timing(s_can_config *p_config);  /* configures the default CAN baud rate */
void can_init(s_can_config *config);
/* initializes MSCAN module according to values in configuration structure */
int can_rxstat(void);
/* gets status of the internal receive buffer: 0-empty, 1-at least one packet ready in buffer */

Word8 can_tx(Word8 buffer, s_can_frame *data);

/* Transmits a CAN message frame. */
/* buffer: 0, 1, 2 for buffer1, 2 or 3; other value for first empty buffer */
/* returns the buffer number used for transmission */

Word16 can_rx(s_can_frame *data);
/* returns: –1 if no message available in the internal receive buffer or 0 if a message is
stored in the internal receive buffer */

short    can_rxflag(void);
/* returns Receiver Flags in the Receiver Flag Register */
void can_clr_rxflag(short flag);
/* clears the RX flag */

Word8 void can_abort_transmission();
/* Aborts the current pending tx requests. */

UWord32 can_get_status();
/* Returns the status of MSCAN. The status code is defined as aforementioned. */
```

The s_can_frame structure stands for the CAN frame either received or to be transmitted, and is defined according to MSCAN *Programmer's Model of Message Storage* as below:

```
typedef struct can_frame {
        UWord8 id[4];              //  Contents of Identify Register 0 to 3
        UWord8 data[8];            //  Contents of Data Segment Register 0 to 7
        UWord8 length;             //  Content of Data Length Register
        UWord8 priority;           //  Content of Transmit Buffer Priority Register1
        UWord8 timeh;              //  Content of Time Stamp Register High
        UWord8 timel;              //  Content of Time Stamp Register Low
}s_can_frame;
```

The s_can_config structure contains the CAN module configuration parameters for bit timing, message filtering, CAN clock source, and CAN clock frequency. It is defined as:

```
typedef struct can_config {
        UWord16 timing;   /* higher bits for BTR1, lower bits for BTR0 */
        union {
                struct {
                        UWord32 filter0;
                } f32b;
                struct {
                        UWord16 filter0;
                        UWord16 filter1;
                } f16b;
                struct {
                UWord16 filter0: 8;
                        UWord16 filter1: 8;
                UWord16 filter2: 8;
                        UWord16 filter3: 8;
                } f8b;
        } acceptance0,mask0, acceptance1, mask1;
        UWord16 filters;  /* 0 - 2x 32bit filters, 1 - 4x 16 bit, 2 - 8x 8 bit, 3 - filter 
closed (no rx) */
  UWord8  clks:1;             /* clock source: 0 for osc clock, 1 for ip bus clock */
  UWord8  oscMHz:7;     /* oscillator clock/ip bus clock in MHz */
} s_can_config;
```

MSCAN module initialization is performed by these steps:

1. Enable MSCAN module by asserting CANE bit in CAN Control Register 1;.
2. Check if the MSCAN is in the initialization mode. If not, force it to sleep mode, then to initialization mode.
3. Change CANCTL1, CANBTR0, CANBTR1, CANIDAC, CANIDAR0-7, and CANIDMR0-7.
4. Exit initialization mode.
5. Enable RX interrupts.
6. If MSCAN remains in sleep mode, exit the sleep mode.

The message is transmitted according to Figure 6.

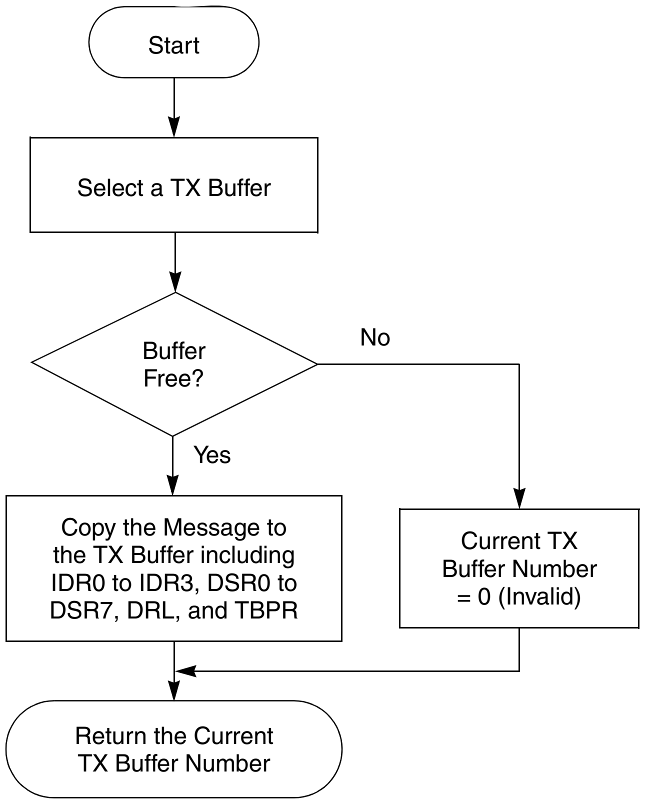

Start
Select a TX Buffer
Buffer Free?
No
Yes
Copy the Message to the TX Buffer including IDR0 to IDR3, DSR0 to DSR7, DRL, and TBPR
Current TX Buffer Number = 0 (Invalid)
Return the Current TX Buffer Number

Figure 6. Message Transmit

The message is received according to these steps:

1. Copy the content of CAN IDR from the internal RX buffer to the id field of the s_can_frame structure.
2. Copy the length field from the internal RX buffer to the length of the s_can_frame structure.
3. Copy the message body from the internal RX buffer to the data field of the s_can_frame structure.
4. Copy the time stamp register contents from the internal RX buffer to the timeh and timel fields of the s_can_frame structure.

5.3.3 Bridge Application Firmware

The bridge application firmware initializes the USB driver and MSCAN driver, and prepares the USB pipes for receiving the command from the USB host. It then performs the main task — to check the command received from the USB host — and manages the command accordingly.

Preparation of USB pipes for receiving the command from the host is done by calling USB driver API routine usb_receive() on both pipe 1 and pipe 3. Before the preparation or use of USB pipes, the device must be enumerated and in its configured state, as required by the USB specification. That is, the USB driver API routine usb_get_state() must return USBST_CONFIGURED before using any USB pipes other than the default pipe.

The prototype of the main task is:

```
  if (wait_for_commands()) {
       handle_commands();
}
if(can_rxstat()){
    send_msg_to_host();
}
```

The wait_for_commands routine first checks both USB pipe 1 and pipe 3 to see if they are busy, by calling USB driver API routine usb_ep_is_busy(). If a USB pipe is not busy, it retrieves the command code and the peripheral ID from the internal endpoint buffer.

The handle_commands routine calls one of the command handler routines, as defined below, to process the command.

```
typedef void (* T_CMD_HANDLER) (void);
const T_CMD_HANDLER cmd_handler[] = {
  can_rx_msg_from_host,
  can_set_baudr,
  can_set_msg_filter,
  can_abort_tx,
  can_report_status,
  NULL
};
```

A CAN message will be sent to the host via the send_msg_to_host() routine. This routine calls can_rx() to retrieve the received CAN message from the MSCAN driver. It waits until USB pipe 4 is not busy, then it transmits the CAN message to the host by calling the USB driver API routine usb_send().

The command handler routine can_set_baudr configures the MSCAN bit timing parameters according to the user-selected baud rate from the PC side GUI. If there are no problems, it will return 0xFF in the data field of the protocol packet over endpoint 2 to acknowledge the command. Otherwise, it will return 0x00 in the data field of the protocol packet.

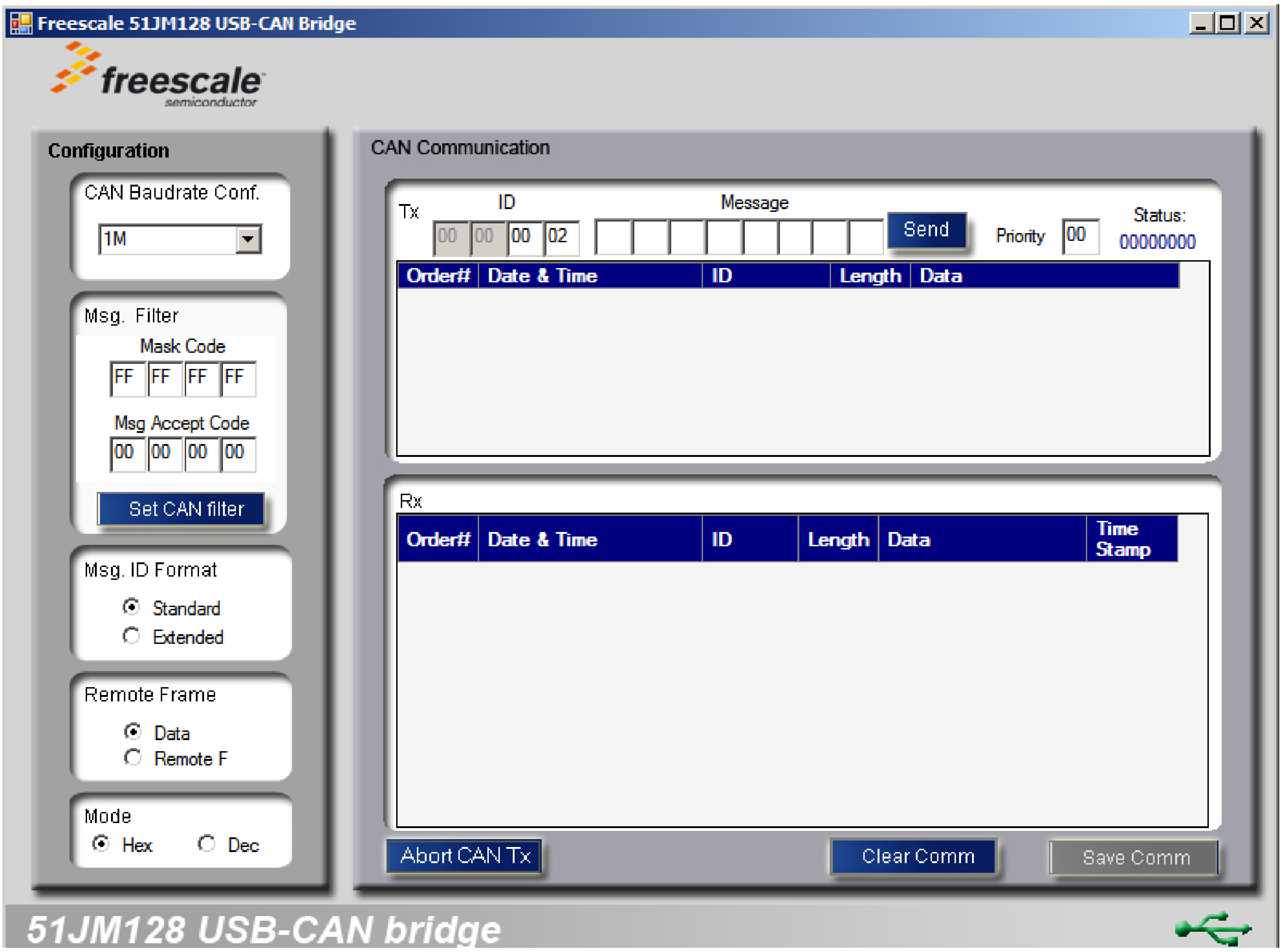
The can_set_msg_filter routine sets the MSCAN identifier mask registers and the identifier acceptance registers according to the user-selected mask code and the acceptance code from the PC side GUI. It also returns 0xFF in the data field of the protocol packet over endpoint 2 to acknowledge the command.

The can_abort_tx routine aborts all current pending MSCAN TX requests and returns 0xFF in the data field of the protocol packet over endpoint 2 as an acknowledgement of the command.

The can_report_status routine returns the status code of the CAN in the data field of the protocol packet over endpoint 2.

5.4 PC-Side GUI

This picture shows the bridge demo GUI. It was developed with Microsoft Visual Studio 2005 Visual Basic 2005. The driver is developed with Visual C 2005 under the Microsoft Driver Foundation using WinUSB technology.



CAN Baudrate Conf.: 1M

Msg. Filter
Mask Code: FF FF FF FF
Msg Accept Code: 00 00 00 00

Msg. ID Format: [✓ Standard] Extended

Remote Frame: [✓ Data] Remote F

Mode: [✓ Hex] Dec

Tx ID: 00 00 00 02
Priority: 00
Status: 00000000

The bridge GUI is divided into four parts: Configuration, CAN Communication, File Operation, and Connection State.

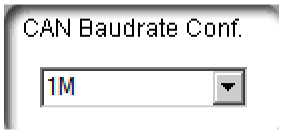
The Configuration pane includes options for CAN Baudrate Conf, Msg. filter, Msg. ID Format, and Remote Frame.

The CAN Communication pane includes TX information, RX information, and an Abort transmission (Abort CAN TX) button. Here there are also two file operation buttons:

Clear Comm

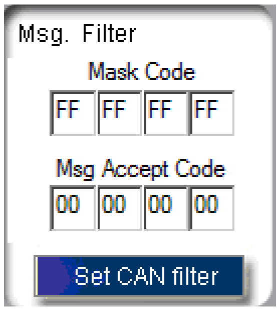
Save Comm

The Connection State shows the USB connection state of the device. If the bridge device is connected to the PC and recognized by the GUI, then the USB logo state is highlighted with green — otherwise it is gray.



CAN Baudrate Conf.: 1M

The CAN Baudrate Conf. drop-down list is used to select the baud rate. Please refer to Section 5.2, “CAN-to-USB Bridge Communication Protocol,” for a list of supported baud rates.

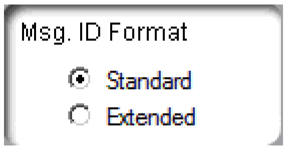


Msg. Filter

Mask Code: FF FF FF FF

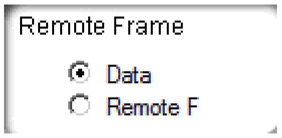
Msg Accept Code: 00 00 00 00

The Mask Code and Msg Accept Code in the Message Filter (Msg. Filter) group are used to configure the identifier mask registers and the identifier acceptance registers. After the Set CAN Filter button is clicked, these registers are configured with the value specified.



Msg. ID Format: [✓ Standard] Extended

The Msg. ID Format group specifies whether the message to be sent uses a standard ID or an extended ID. It is used when sending a CAN frame. The ID field of the TX group in the communications item group is qualified by the Msg. Format group.



Remote Frame: [✓ Data] Remote F

The Remote Frame group specifies whether the message to be sent is a data frame or a remote frame.

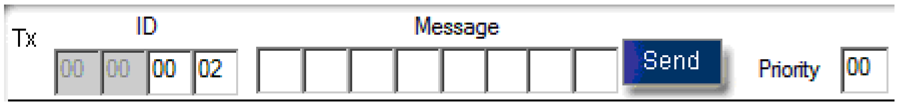
The Abort CAN Tx button is used to abort all the current pending TX requests.

The maximum number of pending TX requests is 3 because of the 3 TX buffers in MSCAN. If further messages can not be sent Abort CAN Tx button must be clicked to abort the pending TX requests.

To send a CAN frame, these fields must be provided:

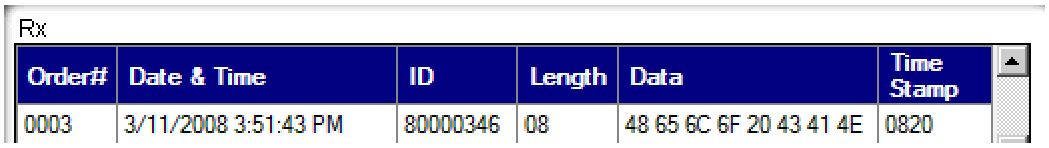
- ID
- Length of message data
- Message data
- Priority

The fields must be specified in the appropriate boxes by the user: the ID field in the ID box, the message data field in the Message box, and the priority field in the Priority box. The length field is automatically calculated when the user enters the data in the message box. All values in these boxes are input in hexadecimal format.



Tx	ID				Message								Priority
	00	00	00	02									00

The TX Communication area has a Send button. When the Send button is clicked, the message specified in the above fields is sent to the device, which will bridge it to the CAN bus. This message will then be logged in the TX list window.


Rx

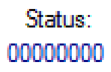
Order#	Date & Time	ID	Length	Data	Time Stamp
0003	3/11/2008 3:51:43 PM	80000346	08	48 65 6C 6F 20 43 41 4E	0820

The RX Communication area has an RX list window. If a CAN message has been received from the device, it will appear in the RX list area. The columns of the RX list and TX list are described below:

- The “Order #” column displays the order of the CAN message received from the device or transmitted to the device.
- The “Date & Time” column shows the local date and time retrieved from the PC if this message was received or transmitted.
- The “ID” column displays the message ID of this message.
- The “Length” column displays the number of data bytes for this message.
- The “Data” column shows the actual data bytes of this message.
- The “Time Stamp” column shows the time stamp of the free-running counter on the device at the time it received the message.

When the Save Comm button is clicked, all messages in the RX list window will be saved to an Excel file.

To clear the RX list window, click on the Clear Comm button.


Status:
00000000

The Status item displays the status code of the CAN module or the status string of the current GUI operation. The status of the CAN module will be polled in an interval as specified in the endpoint 2 descriptor.

If the abort operation is successful, the Status item will show “Abort Ok.”

If the CAN baud rate is configured successfully, the Status item will show “BR Ok.”

If the Set CAN filter operation is successful, the Status item will show “Filt Ok.”

5.5 How to Run the Demo Application

5.5.1 Install Windows Software

First make sure that Windows XP (or newer) is running on the PC.

Run Setup.exe to open the MCF51JM128 USB Driver Installer. The installer will detect if the Microsoft .NET Framework 2.0 is installed on the PC. If not, the installer will ask you whether or not to download and install it from the Microsoft website. After clicking on the Accept button, the installer will automatically download and install the .NET software.

Now follow the directions of the installer all the way to the end of the installation procedure.

5.5.2 Prepare the DEMOJM Board

Before applying the power to the DEMOJM board, make sure the jumper settings are correct. The external power supply must not be plugged into P1. The 51JM128 daughter card must be installed on the DEMOJM board.

Here is a list of correct jumper settings for the DEMOJM board Rev. D.

Note: settings listed indicate the on (or installed) position.

Jumper	Installed settings
J3	3 & 4
J4	1 & 2, 3 & 4
J6	2 & 3
J7	1 & 2
J8	1 & 2, 3 & 4
J11	1 & 2
J12	1 & 2
J17	All on
J18	2 & 3
J19	2 & 3

Jumper	Installed settings
J20	2 & 3
J21	1 & 2, 3 & 4, 5 & 6
J24	1 & 2
J27	All on
J28	1 & 2, 3 & 4
J29	1 & 2, 3 & 4
J30	1 & 2
J31	1 & 2, 3 & 4
J32	1 & 2, 3 & 4
J33	1 & 2

Use one USB cable to connect the board and the host PC. Use another USB cable with one mini-AB connector to connect the mini-AB plug on the board to the host PC.

Now connect the MCF51JM128 to the CAN bus by connecting CANH and CANL on J5 to the corresponding connectors of the correct CAN node. For example, you can use another MCF51JM128 DEMOJM as a remote CAN node, and interconnect the corresponding pins on J5 of both DEMOJM boards. In this case, both DEMOJM boards must be connected to their respective PCs via USB cables and prepared as described. In addition, the firmware must be downloaded on both boards and the PC-side GUI and Windows drivers must also be installed on both PCs. The firmware download and Windows driver installation are described in the next section.

5.5.3 Run the DEMOJM Board

Turn on the power by placing switch K1 to the on position.

Download the code with CodeWarrior 6.1 (or newer) to the device and run it.

The host will recognize the new USB device and request to install the driver. Obey the directions of the driver installation wizard to browse to the folder where the driver is installed and then click on Next. The driver will be installed correctly.

Now click on the "USB-CAN Bridge" application on the desktop. It will display the GUI depicted in Section 5.4, "PC-Side GUI." The GUI will automatically set the CAN baud rate to 1 Mbps. If a different baud rate is needed, select one from the CAN Baudrate Conf. drop-down list until the status item displays "BR Ok."

By default, the bridge will receive all messages. If you are interested in only some of the messages, then the Mask Code & Accept Code must be specified by clicking the Set CAN Filter button. To make sure that the Set CAN Filter operation is performed successfully check the Status item. Click on the Set CAN Filter button until the string "Filt Ok" is displayed in the Status item.

If the baud rate is correct, the bridge will send all received CAN messages to the GUI, which will be listed in the RX list window.

All the received messages in the RX list window can be saved to an Excel file by clicking the Save Comm button.

The RX list window can also be emptied by clicking on the Clear Comm button.

If a CAN message is sent to the bridge device, you must enter the values into the ID box, Message box, and Priority box in the TX group and then click the Send button. The extended CAN message can also be sent to the bridge device by checking the Extended box in the Msg. Format group before entering values in the ID box.

Up to three messages may be sent to the bridge device at the same time. If any of the message is not sent successfully by the bridge device, it is because the remote CAN node has failed to receive these messages. The messages can be aborted by clicking on the Abort CAN Tx button.

If the abort operation is successful, the Status item will show "Abort Ok."

5.6 Resource Usage

With the compiler optimizations selected, the bridge occupies a very small amount of memory in bytes as listed in Table 7. Flash memory contains code, initialized data, and constant variables such as USB descriptors. RAM contains uninitialized data, initialized data, heap, stack, and the BDT (the USB buffer descriptor table).

Table 7. CAN-to-USB Bridge Memory Usage

Memory	CAN-to-USB Bridge
Flash	10948
RAM (Total)	3096
Stack	336
BDT	512

Document Number: AN3690
Rev. 0
06/2008