A UNIVERSAL SERIAL BUS GAMEPAD DEVICE USING THE MC68HC05JB2

By Manish Bakshi and Tolly Smith

Introduction

This application note details how the Motorola USB Device Firmware library can be used to develop a sample universal serial bus (USB) application belonging to the human interface device (HID) class. It describes how to setup the USB device information, and integrate the library with custom external hardware controlling firmware.

The universal serial bus (USB) standard is an interconnection designed to support consumer, telephony and productivity peripherals for the PC. It offers a user friendly, dynamically attachable and bandwidth adjustable plug and play interface. The bus provides expansion of up to 127 peripherals, and operates at two physical transfer rates: 1.5 Mb/s or 12 Mb/s, making it an ideal candidate for use in such multimedia applications as audio, and compressed video. In addition, the USB standard is implemented in an open software architecture through a base class, and a series of horizontal device classes. This renders the standard expandable for future usability.

Supporting the USB base class and all of the other device classes can be a tedious procedure for an embedded developer. To reduce the work involved (such as writing USB firmware, testing and measuring USB device class' compliance), Motorola has developed this firmware library for the USB base and human interface device (HID) class on the MC68HC05JB2. This library contains an advantageous solution to developing and manufacturing USB conformant products in today's fast paced and competitive market place. This is its fully tested, optimized and documented functions written in the C programming language.

The MC68HC05JB2 is the first USB conformant microcontroller from Motorola. It is a low-cost, high-performance microcontroller belonging to the M68HC05 device family. The chip is available in a 20 pin PDIP or SOIC package, and incorporates a low-speed USB module. The JB2 CPU supports the same instruction set and is therefore code-compatible with other devices in the M68HC05 family. It has 128 bytes of user RAM and 2048 bytes of user ROM. It supports memory mapped input/output (I/O) registers, and 10 bidirectional I/O pins with software programmable pull-up or down capability for external hardware interfacing. The low speed USB module supports 3 endpoints. The first is a control endpoint and other two interrupt endpoints. This makes the MC68HC05JB2 ideally suited for use in low speed USB applications such as HID devices.

USB OVERVIEW

The USB standard is a data exchange interface between a host computer and up to 127 simultaneously accessible peripherals. The attached peripherals share USB bandwidth using a host scheduled token based protocol. The bus allows peripherals to be attached, configured, used, and detached while the host and other peripherals are in operation using a simple 4 pin connector shown in Figure 1. USB peripherals can be self-powered or rely totally on power from the USB cable. USB hosts and devices can also perform active power management meeting another requirement for some of today's power sensitive applications.

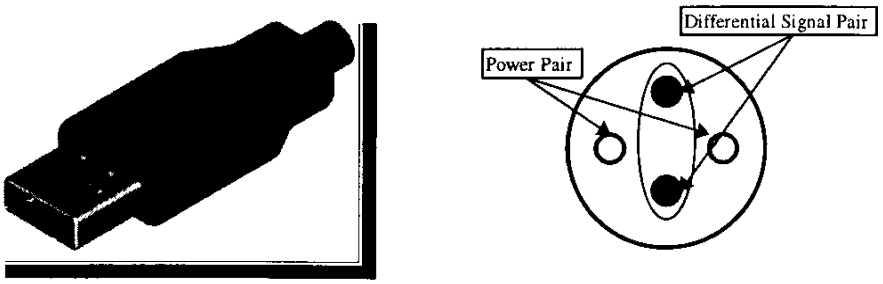

Figure 1: A USB connector and signal pair.

USB token based protocol transactions are initiated by the host on a scheduled basis. The host begins by sending four pieces of information in a USB packet. These are the message type, data transfer direction, the USB device address and the sub-channel or endpoint number it is communicating with in the USB device. The selected USB device decodes the token, and begins receiving/sending information. When all data has been communicated, the receiver responds with a final handshake packet indicating whether the transfer was successful. Each USB device is required to have one sub-channel available at all times called the control endpoint. It is used by the host at device installation to determine the capabilities and activate one device configuration. Apart from control endpoints, the USB standard also allows for three other endpoint types: bulk, interrupt and isochronous. Interrupt and isochronous endpoints provide guaranteed data delivery using dedicated bandwidth, while bulk endpoints do not. When multiple USB peripherals are connected to a USB host controller, devices are configured on the network only if their necessary bandwidth is available.

USB data transmitted on the serial bus use variable length messages. The USB standard uses three types of messages: token, data and handshake. Token messages have three subtypes called SETUP, IN and OUT. Data messages have two types – DATA0 and DATA1, and handshake messages have three types – ACK, NAK and STALL. The format of the three USB message types are shown in Figures 2, 3 and 4. As depicted, all messages begin with a synchronization (SYNC) field, which has the start of packet (SOP) field embedded in it. In a USB token packet, the SYNC field is followed by the packet identifier (PID), device address (ADDR), endpoint number (ENDP) and a cyclic redundancy check (CRC) field. A USB data packet consists of the SYNC field, the packet identifier (PID) field, zero or up to 1023 data bytes, and a CRC field. A USB handshake packet consists of the packet identifier (PID) field only. All messages end with an end of packet (EOP) field. For further information, consult the USB Specification document [1].

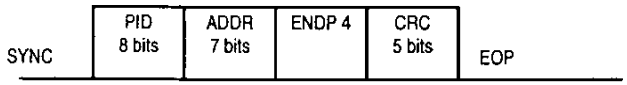

Figure 2. USB Token Packet Format

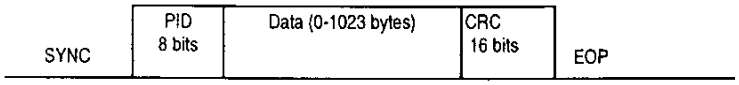

Figure 3. USB Data Packet Format

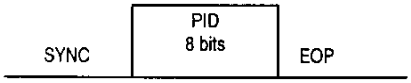

Figure 4. USB Handshake Packet Format

All USB devices belong to the Base class automatically, and can also belong to other classes such as HID and Audio. Each device class incorporates its own data transport and protocol requirements. For example, the USB Audio device class requires timely delivery of large data blocks, and uses isochronous endpoints. The HID device class, on the other hand, requires small reliable data transfers, and uses interrupt endpoints.

To support various device class types, USB devices may have one or more configurations, and each configuration may have a single or multiple device class interfaces. All information is provided in USB descriptors. For the base class, there are Device, Configuration, Interface, Endpoint and String descriptors. All other device classes are mapped to the USB base class Interface descriptor. For the HID device class, the Interface descriptor includes the HID descriptor, the HID report descriptor and the HID physical descriptors. All of these USB base and HID device class descriptors are fixed length except for the HID report descriptor. Its length depends on the number of items present. An item provides information about each physical control on an HID class USB device, and each item has a type, tag and size contained within it. The item type allows for global or local device controls, and the item tag identifies whether the data generated is absolute or relative. For further details on HID report descriptors, please consult the USB Device Class Definition for Human Interface Devices specification [2].

Motorola USB Device Firmware Library Overview

The Motorola USB Device firmware library implements the USB base and HID classes, and is available in four models - Tiny, Small, Medium and Large. This structure of the library provides maximum flexibility, and allow it to be used in various microcontroller code and memory sizes.

Each library model includes functions to detect and handle the MC6805JB2 USB module's interrupts, decode device requests and respond with either user customized firmware or USB descriptor information. These USB messages

provide the host with details on the device's capabilities, and allow the host machine to configure the device. The library provides to the host information contained in the Device, Configuration, Interface, String and Endpoint descriptors from the basic USB standard. From the support of the USB HID device class, the library model give information contained in the HID class descriptor, the HID Report descriptor and HID Physical descriptors. The current version of the Motorola USB Device Firmware library has HID physical descriptors on Interface 0 of the base class only.

Each Motorola library model provides different base and HID capabilities. USB base class support that varies across models includes string descriptor, multiple configurations, communicating in multiple USB packets on the interrupt endpoint, and notification of host changes on the USB device. HID device class support that varies across library models include support for Control (Feature), Output and Input reports.

The Tiny model supports one HID input report, string descriptors, and no HID output or HID feature reports. The Small model supports one HID input and output report, and idle input report capability. The Medium library supports one HID input report, one HID output report, reconfiguration of input reports, string descriptors, multiple packet messages and notification of host changes. The Large model supports more than one report of control, output and input type, and multiple configurations.

The Motorola USB device firmware library does not include the USB base and HID device class descriptors. These must be provided externally by the final user. The library also provides a mechanism to control and integrate external hardware using three predefined C-based functional hooks. These functions are the Init_Device_Interface, Suspend_Device_Interface and Poll_Gamepad_Interface.

Gamepad Specifications

The basic purpose of the Gamepad is to scan a directional pad which acts as a joystick, and a series of buttons. It must decode them and then transmit the information to the PC using a USB port. This information about the gamepad states must be transmitted continually as long as buttons are in use.

The gamepad hardware is designed to be bus or independently-powered. It must have the capability to request attention from the host and inform it of any new state changes. To minimize power usage, it must enter and exit STOP mode based on the host input requirements.

Gamepad

The Gamepad board (supplied in the Motorola USB Device Evaluation Kit) has eight buttons, four of which implement a directional pad, and the remaining are an ordinary button interface. All buttons are connected into the 68HC05JB2 which provides up to 10 general purpose software configurable I/0 pins. For this gamepad application, eight I/O lines are connected to the eight buttons on Port A. Port A is set to an input in its Data Direction Register and its default logic state is set to a low voltage using the internal pull-down register facility. External CPU interrupt capability is also available through four I/O pins on Port A (PA0, PA1, PA2, PA3).

The gamepad block diagram and printed circuit board are shown in Figures 5 and 6 respectively. The board has jumper JP1 to select either a bus-powered or self-powered USB operation mode, an on-board oscillator (U2), the ZIF socket for the JB2 and eight standard buttons. The ZIF socket can be used with a standard JB2 chip or interface directly with a JB2 MMDS module to provide in-circuit emulation and debugging capability from a PC environment.

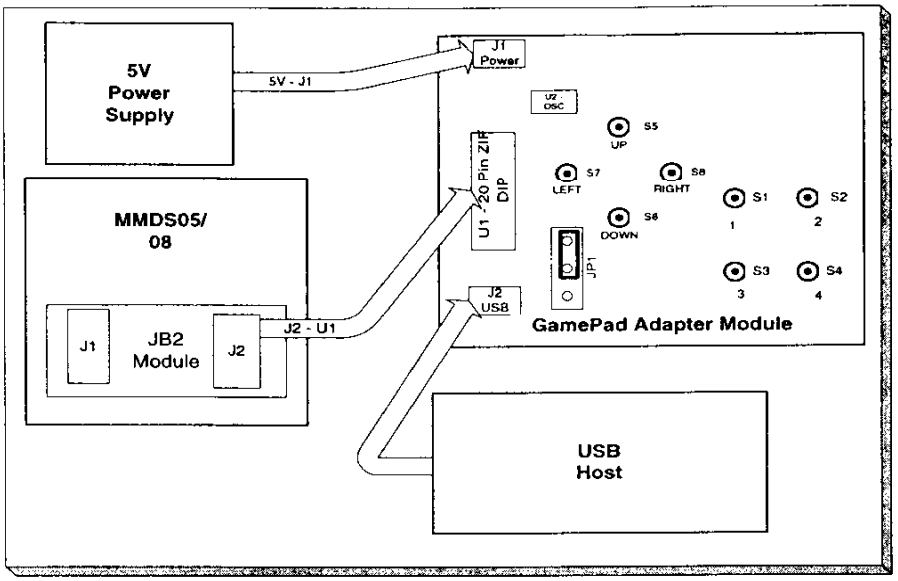

Figure 5. Motorola USB Gamepad Adapter Block Diagram

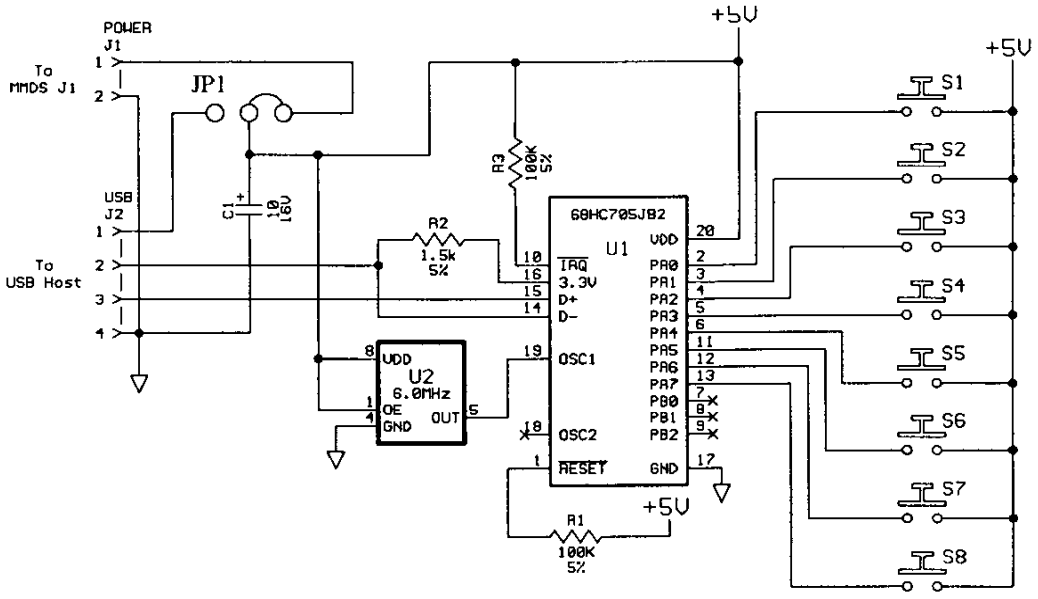

Figure 6 . Schematic of the Motorola USB Gamepad Adapter

Implementation

The software architecture of a USB device implemented with the Motorola USB library is shown in Figure 7. The figure illustrates all of the firmware blocks that must be adjusted (highlighted), and the data flows between them. At device installation, USB messages sent back and forth between the host and device, select an I/O device interface to activate. This activated interface reads and writes to JB2 external hardware and provides this information to the User Application Firmware block. This block then relays this information to the Motorola USB library, which finally sends it over in a USB formatted packet to the host computer.

The Motorola libraries allow easy integration of the user application firmware block, and device interface firmware block into one software unit using two procedures. The first procedure involves the placement of device and interface information into USB device descriptors. The second merges the user application firmware with library functions to read, write and control external hardware.

For the sample Gamepad interface, the Tiny model includes enough functionality to meet the requirements. The Tiny model has support for a single USB device configuration, a single interface, and one HID INPUT report.

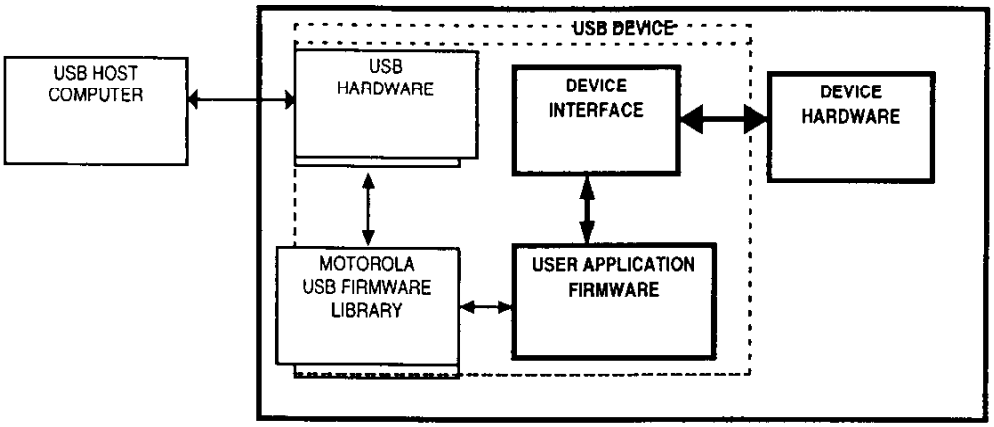

Figure 7: A USB Device's Software Architecture.

USB Base and HID device class descriptors.

The USB base and HID device class descriptors must be provided external to the library. USB base class descriptors are the Device, Configuration, Interface, Endpoint and String descriptors. USB HID descriptors are the HID class, the HID Report descriptor, and an optional HID Physical Descriptor.

The following tables present the descriptor attributes, and indicate whether they are fixed or variable when used with the Tiny model of the Motorola USB Device Firmware library. Strings used in the Device, Configuration and Interface descriptors are referenced by their indices, but only defined in the String Descriptor section.

1. USB Device descriptor

   This descriptor is shown from line 93 to line 109 of the software listing. The C-based structure Tdevice_descriptor, is present in the library for defining Device descriptors, and the settings of individual attributes for the Gamepad are shown in the table below. The USB Vendor ID and Product ID have been set to the Motorola USB and Gamepad identification tags respectively. The device release number, and the number of configurations have both been set to 1.

Attribute	Gamepad	Fixed	Variable
Length	18	✔	
Descriptor Type	DEVICE	✔	
USB Standard release number	100	✔	
USB Device Class	0		✔
USB Device Subclass	0		✔
USB Device Protocol	0		✔
Maximum packet size on Control Endpoint	8 bytes	✔	
USB Vendor ID	1063		✔
USB Product ID	2560		✔
USB Device Release number	1		✔
Manufacturer String Descriptor Index	1		✔
Product String Descriptor Index	2		✔
Serial Number String Descriptor Index	0 (unused)		✔
Number of Configurations in Device	1	✔	

2. Configuration descriptor

   A base class configuration consists of one or many Interface, Endpoint and HID class descriptors. In the Motorola USB Device Firmware library, each individual configuration is referenced with the Tconfiguration_descriptor variable. For the single configuration on the Gamepad, this descriptor is defined from line number 1502 to 163 and it's attributes are as follows:

Attribute	Gamepad	Fixed	Variable
Descriptor Length	9	✔	
Descriptor Type	CONFIG	✔	
Total length (Conf., Interf., Endpt, HID)	37		✔
Number of Interfaces in this Configuration	1	✔	
Value to use to select this Configuration	1	✔	
Configuration string descriptor index	0 (unused)		✔
Characteristics: • Bus-powered • Self-powered • Remote wakeup • Reserved	0		✔
Maximum power consumption	50 mW		✔

3. Interface Descriptor

An Interface descriptor in the library is defined by the Tinterface_descriptor variable. This descriptor contains the HID class, HID sub-class and HID protocol attributes, and the alternate settings and the number of endpoints used by this interface. For the Gamepad, this descriptor appears from line 164 to 174 of the Software listing. The descriptor attributes and their settings for the Tiny model are as follows:

Attribute	Gamepad	Fixed	Variable
Descriptor Length	9	✔	
Descriptor Type	INTERFACE	✔	
Number to select this interface	0	✔	
Alternate Setting of Interface	0	✔	
Number of Endpoints used by interface	1		✔
Interface Class Code	3 (HID)		✔
Interface Subclass Code	1		✔
Interface Protocol Code	2		✔
Interface String descriptor index	3		✔

4. Endpoint descriptor

This is defined in the Tendpoint_descriptor in the Motorola USB library, shown from line numbers 175 to 182 of the Software listing and has the following attributes:

Attribute	Gamepad	Fixed	Variable
Descriptor Length	7	✔	
Descriptor Type	ENDPOINT	✔	
Address of endpoint and direction of transfers	1, IN	✔	
Endpoint type	3	✔	
Maximum Packet size	8 bytes		✔
Polling interval	10 ms		✔

5. String descriptors

These must be defined for all base class descriptors that have non-zero string indices . For this Gamepad application, these are the Device and Interface descriptors. Each string must be encoded in the desired language using the UNICODE International Character standard. Strings are defined in the Motorola USB libraries using the Tstring_descriptor C-based structure. The Gamepad application uses three String descriptors to describe the Manufacturer, Product and Serial number in the Device descriptor, and one String descriptor to describe the Interface descriptor above. These four descriptors are shown from line numbers 36 to 77 of the Software listing.

6. HID Class Descriptor

An HID class descriptor has a target country attribute, an HID class compliance attribute, the number of HID descriptors (HID Physical and HID Report only) used by the HID interface, and the length of all HID descriptors. The gamepad application has no physical preferences and requires no HID physical descriptors. It uses one HID report descriptor to describe the physical controls on the Gamepad. The attributes are shown from line numbers 183 to 191, of the Software listing and are:

Attribute	Gamepad	Fixed	Variable
Descriptor length	9	✔	
Descriptor type	21	✔	
HID class specification release number	100	✔	
Hardware Target Country	0		✔
Number of HID class descriptors to follow	1	✔	
Type of HID class descriptor included	0x22		✔
Total length of Report Descriptor	53		✔

7. HID report descriptor

This descriptor details the input/output capabilities, and contains information placed in HID items. Each HID item informs the USB host about all the data to be transmitted by a USB device. Report descriptors can be created using an HID report descriptor generator. For further details on specifying HID report descriptors, please consult the USB Device Class Definition for Human Interface Devices document [2]. The HID report descriptor is on lines 111 to 140 of the software listing. The listing in Figure 8 gives the individual byte codes for the Gamepad HID report descriptor, and lists each HID item for the corresponding byte code.

```
0x05,0x01,          // Usage Page (Generic Desktop)
  0x09,0x04,          // Usage (Joystick)
  0xA1,0x01,          // Collection (Application)
  0x09,0x01,          // Usage (Pointer)
  0xA1,0x00,          // Collection (Physical)
  0x05,0x01,          // Usage Page (Generic Desktop)
  0x09,0x30,          // Usage (X)
  0x09,0x31,          // Usage (Y)
  0x15,0x00,          // Logical Minimum (-127)
  0x25,0x7F,          // Logical Maximum (127)
  0x66,0x00,0x00, // Unit (None (2 bytes))
  0x75,0x08,          // Report Size (8) (bits)
  0x95,0x02,          // Report Count (2) (fields)
  0x81,0x02,          // Input (Data, Variable, Absolute)
  0x05,0x09,          // Usage page (buttons)
  0x19,0x01,          // Usage minimum (1)
  0x29,0x04,          // Usage maximum (4)
  0x15,0x00,          // Logical Minimum (0)
  0x25,0x01,          // Logical Maximum (1)
  0x75,0x01,          // Report Size (1)
  0x95,0x04,          // Report Count (4)
  0x81,0x02,          // Input (Data, Variable, Absolute)
  0x95,0x01,          // Report Size (1)
  0x75,0x04,          // Report Count (4)
  0x81,0x01,          // input (4 bit padding)
  0xC0,                  // End Collection
  0xC0                   // End Collection
```

Figure 8. USB HID Report Descriptor

This report descriptor defines four input items sent from the gamepad: two 8-bit items for the X-axis, and Y-axis on the directional pad, a 4-bit item for each on/off button and one four bit dummy input item.

8. The Motorola USB device library can support the above descriptors in one or multiple quantities based on the USB firmware library model selected. The following descriptor lists must be adjusted based on choice of library model:
   i. MUSB_CONFIG_DESCRIPTOR_LIST (shown on lines 194-197)
   ii. MUSB_STRING_DESCRIPTOR_LIST (shown on lines 199-202)
   iii. MUSB_HID_REPORT_DESCRIPTOR_LIST (shown on lines 203-206)
   iv. MUSB_HID_PHYSICAL_DESCRIPTOR_LIST (shown on lines 207-210)
   v. MUSB_HID_INPUT_REPORT_LIST (shown on lines 26-32)
   vi. MUSB_HID_OUPUT_REPORT_LIST (used in the Tiny model).
   vii. MUSB_HID_FEATURE_REPORT_LIST (used in the Tiny model).
9. The Motorola USB device library requires some memory for internal data storage, which varies with library model selected. The following variables must be adjusted with USB device class descriptors :

i. MUSB_INTERFACE_STATE: This array must have as many elements as the number of Interface descriptors. For the Gamepad, only one element is required as shown on line number 213.
ii. MUSB_ENDPOINT_STATE : This array must have as many elements as used in the Gamepad application, and cannot exceed the JB2 limit of three elements. The number of elements has been set to two for the Gamepad.
iii. MUSB_ENDPOINT_STATE_LIST: This has two elements for the two endpoints used in the Gamepad application.
iv. MUSB_RECEIVE_BUFFER_0 : This ties into the maximum packet size on endpoint 0 defined in the USB device descriptor.

Interfacing with the library

The Motorola libraries have function hooks to provide easy installation, configuration and use of MC68HC705 chip connected hardware. These three functions are Init_Device_Interface for installation and configuration, Poll_Device_Interface to determine the states of any custom hardware and transmit them to the host, and the Suspend_Device_Interface function.
For the Gamepad application, it is necessary to store button states scanned from the HID device interface, and transmit them to the host in an HID INPUT report format. A C-based structure has been defined to store these input states called TGamepad_Report. This structure contains three byte-sized variables that store the directional pad's X coordinate value, Y coordinate value, and the binary states of the four remaining buttons.
The Init_Device_Interface function sets up the I/O port by writing to the Port A data direction register, and resets the USB device and endpoint state information.
Reading of gamepad button states and transmission of data is done on a polled basis by the Poll_Device_Interface function. For the Gamepad implementation, this function reads the I/O signal lines, and assembles the state information into the C-based variable Tgamepad_Report. Transmission of the data is done by invoking the Motorola library function Musb_HID_Write_INPUT_Report. This sends the scanned button state information to the host at the polling rate described in the Endpoint descriptor.
A USB host can suspend the USB port on which a device is connected. To detect this condition and perform hardware adjustments, the Suspend_Device_Interface has been provided. This is used to put the gamepad into a STOP mode, until further USB activity is detected

Conclusion

This application note has detailed how the Motorola USB Device Firmware library lends itself to quick development of an HID conformant application. With some minor modifications, it is possible to produce a USB solution of any HID complexity with the Tiny, Small, Medium and Large USB library models. The library also allows incorporation of end-user customized firmware for controlling custom external hardware. This represents an advantage for those Motorola customers already using the MC6805 device family, but now requiring a USB conformant product in today's consumer driven marketplace.
Motorola remains dedicated to supporting the USB standard, and continues to develop USB conformant microcontrollers and microprocessors. For the latest information, contact your nearest Motorola sales representative.

References

1. *Universal Serial Bus Specification*, Revision 1.0, January 15, 1996 (http://www.usb.org).
2. *Universal Serial Bus Device Class Definition for Human Interface Devices (HID),* Version 1.0#4, December 1996 (http://www.usb.org).
3. *Universal Serial Bus HID Usage Table*, Draft 0.7f, December 1996 (http://www.usb.org).
4. *MC68HC(7)05JB2 Technical Data*, 1997.
5. *Motorola USB Device Firmware Library Manual*, 1997.

Glossary

ACK	A type of USB handshake message.	EOP	End of Packet.
		HID	Human Interface Device.
ADDR	USB device address in a packet.	IN	A type of USB token message.
CRC	Cyclic Redundancy Check in a USB packet.	JB2	MC68705JB2.
		MMDS	Motorola Modular Development System.
ENDP	Endpoint number in a USB packet.		

NAK	A type of USB handshake message.
OUT	A type of USB token message.
PDIP	Plastic Dual In-Line Package.
PID	Packet Identifier.
RAM	Random Access Memory.
ROM	Read Only Memory.
SETUP	A type of USB token message.
SOIC	Surface-mount Small Outline Integrated Circuit.
SOP	Start of Packet.
STALL	A type of USB handshake message.
SYNC	Synchronization Field of a USB packet
USB	Universal Serial Bus.
ZIF	Zero Insertion Force.

GAMEPAD APPLICATION SOFTWARE

```
1       /*****************************************************************************/
2       /*  All of Gamepad Information stored in C structure : Gamepad Report
  */
3       /*    Member x   :  stores the value of the x-axis joystick control and controls 2 buttons.
  */
4       /*    Member y   :   stores the value of the y-axis joystick control and controls 2 buttons.
        */
5       /*    Member buttons : Stores the status of four buttons in ON/OFF manner
  */
6       /*****************************************************************************/
7
8       #define                 VENDOR_ID               0x0427          /* USB Vendor ID for Motorola
*/
9       #define                 PRODUCT_ID              0xA000          /* USB Product ID for Gamepad
*/
10      #define                 RELEASE_NUM     0x0001          /* USB Device Release  for Gamepad */
11
12      #define                 SW (Word)                ( ( ( Word & 0xFF) << 8) | (Word >> 8) )
13      typedef struct
14          {
15          uchar8 X ;
16          uchar8 Y ;
17          uchar8 buttons ;
18          } TGamepad_Report ;
19
20      struct
21          {
22          TINPUT_report_state state ;
23         uchar8 buffer[sizeof(TGamepad_Report)] ;
24         } INPUT_Report_GAMEPAD ;
25
26      CONST TINPUT_report_list MUSB_HID_INPUT_REPORT_LIST[ ] =
27         {
28          &INPUT_Report_GAMEPAD.state,            /* Current state of the Gamepad Buttons */
29          sizeof(INPUT_Report_GAMEPAD.buffer),    /* Size of data to be transferred */
30          1                                                   /* USB Endpoint to be used in
transfers */
31          };
32
33      CONST uchar8 MUSB_HID_N_INPUT_REPORTS =  /*Number of INPUT reports */
34          nelem(MUSB_HID_INPUT_REPORT_LIST);          /* present in the INPUT report list.*/
35
36      CONST struct
37          {
38          Tstring_descriptor desc ;                   /* String descriptor is of a 1 byte
  */
39          uchar8 string[2];                           /* length, and contains the serial*/
40          } String_Language_List_LE =                         /* number of the USB device
  */
41          {                                           /* encoded in the UNICODE  */
42          { sizeof(String_Language_List_LE ),                 /* format.                 */
43          DESCRIPTOR_STRING },
44          {  0, 9 }
45          };
46
47      CONST struct
48          {
49          Tstring_descriptor desc ;                   /* String descriptor is an 8 byte */
50          uchar8 string[2*8] ;                                /* field, and stores the
Manufacturer */
51          } String_Manufacturer_LE =                          /* string in the UNICODE format*/
52          {
53          {sizeof( String_Manufacturer_LE ),
54          DESCRIPTOR_STRING },
55          { 0,'M',0,'O',0,'T',0,'O',0,'R',0,'O',0,'L',0,'A' }
56          };
57      CONST struct
58          {
59          Tstring_descriptor desc ;                   /* This string descriptor stores the */
60          uchar8 string[2*7] ;                                /* Product name in a UNICODED */
61          } String_Product_LE =                               /* formatted string of length 7
bytes */
62          {
63          { sizeof( String_Product_LE ),
64          DESCRIPTOR_STRING },
65          { 0,'G',0,'a',0,'m',0,'e',0,'p',0,'a',0,'d' }
66          };
67
68      CONST struct
69          {
70          Tstring_descriptor desc ;                   /* This string descriptor stores the */
71          uchar8 string[2*4] ;                                /* UNICODED encoded string
*/
```

```
72          } String_Interface1_LE =                    /* describing the Interface descriptor. */
73          {
74          {sizeof(String_Interface1_LE),
75          DESCRIPTOR_STRING },
76          { 0,'D',0,'E',0,'M',0,'O' }
77          };
78
79      #define ISTR_MANUFACTURER     1       /* Set the manufacturer string index to 1 */
80      #define ISTR_PRODUCT                  2       /* Set the product string index to 2 */
81      #define ISTR_SERIAL_NUMBER    0       /* Set the serial number string index to 0 */
82      #define ISTR_CONFIGURATION1   0       /* Set the configuration string index to 0*/
83      #define ISTR_INTERFACE1               3       /* Set the Interface string index to 3 */
84
85      Tstring_descriptor const MUSB_STRING_DESCRIPTOR_LIST[] =
86           {
87           &String_Language_List_LE.desc,                 /*  Index 0 */
88           &String_Manufacturer_LE.desc,                  /*  Index 1 */
89           &String_Product_LE.desc,                           /*  Index 2 */
90           &String_Interface1_LE.desc                         /*  Index 3 */
91      };
92
93      CONST Tdevice_descriptor MUSB_DEVICE_DESCRIPTOR_LE =
94           {
95           sizeof( MUSB_DEVICE_DESCRIPTOR_LE ),           /* Length in bytes of descriptor */
96           DESCRIPTOR_DEVICE,                             /* Descriptor type is set to USB device
*/
97           SW(0x100),                                          /* USB standard the device
conforms to */
98           0,                                                       /* USB Device Class
number */
99           0,                                                       /* USB Device Subclass
number */
100          0,                                                       /* USB Device Protocol
number */
101          MAX_PACKET_SIZE,                               /* Maximum Packet size on Endpoint 0
*/
102          SW(VENDOR_ID),                                      /* USB Vendor ID number */
103          SW(PRODUCT_ID),                                     /* USB Product ID number */
104          SW(RELEASE_NUM),                               /* USB release number  */
105          ISTR_MANUFACTURER,                             /* String index of manufacturer */
106          ISTR_PRODUCT,                                       /* String index of product */
107          ISTR_SERIAL_NUMBER,                            /* String index of USB serial number */
108          1                                                        /* Number of USB
Configurations on device */
109     };
110
111     CONST uchar8 Report_Descr1_LE[] =
112     {
113          0x05,0x01,                                     /* Usage Page (Generic Desktop) */
114          0x09,0x04,                                     /* Usage (Joystick) */
115          0xA1,0x01,                                     /* Collection (Application) */
116          0x09,0x01,                                     /* Usage (Pointer) */
117          0xA1,0x00,                                     /* Collection (Physical) */
118          0x05,0x01,                                     /* Usage Page (Generic Desktop) --HID
tag item  */
119          0x09,0x30,                                     /* Usage (X) - HID tag item */
120          0x09,0x31,                                     /* Usage (Y) - HID tag item */
121          0x15,0x00,                                     /* Logical Minimum (-127) - HID tag
item */
122          0x25,0x7F,                                     /* Logical Maximum (127) - HID tag item
*/
123          0x66,0x00,0x00,                                /* Unit (None (2 bytes)) - HID tag item
*/
124          0x75,0x08,                                     /* Report Size (8) (bits) - HID tag
item */
125          0x95,0x02,                                     /* Report Count (2) (fields) - HIS tag
item */
126          0x81,0x02,                                     /* Input (Data, Variable, Absolute) -
HID tag item */
127          0x05,0x09,                                     /* Usage page (buttons) - HID tag item
*/
128          0x19,0x01,                                     /* Usage minimum (1) - HID tag item */
129          0x29,0x04,                                     /* Usage maximum (4)  -- HID tag item
*/
130          0x15,0x00,                                     /* Logical Minimum (0) - HID tag item
*/
131          0x25,0x01,                                     /* Logical Maximum (1) - HID tag item
*/
132          0x75,0x01,                                     /* Report Size (1) - HID tag item */
133          0x95,0x04,                                     /* Report Count (4) - HID tag item */
134          0x81,0x02,                                     /* Input (Data, Variable, Absolute) -
HID tag item */
135          0x95,0x01,                                     /* Report Size (1) - HID tag item */
136          0x75,0x04,                                     /* Report Count (4) - HID tag item */
```

```
137           0x81,0x01,                                        /* input (4 bit padding) - HID tag item
*/
138           0xC0,                                             /* End Collection - HID tag item */
139           0xC0                                              /* End Collection - HID tag item */
140       };
141
142       CONST struct
143     {
144           Tconfiguration_descriptor Conf_Descr ;
145           Tinterface_descriptor Intr_Descr1 ;
146           Tendpoint_descriptor Endpoint_Descr1 ;
147           THID_descriptor HID_Descr1 ;
148           } Conf_Descr_Grp1_LE =
149           {
150           {                  /* USB configuration and its encompassing descriptors */
151           sizeof(Tconfiguration_descriptor),                /* Length in bytes of descriptor */
152           DESCRIPTOR_CONFIGURATION,                 /* Descriptor type set to Configuration */
153           sizeof(Conf_Descr_Grp1_LE),                       /* Lower byte of the total length of
Configuration */
154           sizeof(Conf_Descr_Grp1_LE)>>8,                    /* Higher byte of the total length of
Configuration */
155           1,                                                      /* Number of Interfaces
supported in Configuration */
156           1,                                                      /* Value of this Configuration
*/
157           ISTR_CONFIGURATION1,                              /* String index for this Configuration;
*/
158           0,                                                      /* Reserved for future use and
set to 0 */
159           0,                                                      /* Configuration's Remote
Wakeup capability flag*/
160           0,                                                      /* Configuration's Self powered
capability flag */
161           1,                                                      /* Configuration's Bus Powered
capability flag */
162           0x32,                                                   /* Maximum Power in mW for this
configuration */
163       },
164           {                           /* Tinterface_descriptor  */
165           sizeof(Tinterface_descriptor),                          /* Length in bytes of interface
descriptor */
166           DESCRIPTOR_INTERFACE,                             /* Descriptor type set to Interface */
167           0,                                                            /* Interface Number for
this interface */
168           0,                                                            /* Value used to select
alternate setting */
169           1,                                                            /* Number of Endpoints
*/
170           DEVICE_CLASS_HID,                                 /* USB Class Code - Set to HID */
171           1,                                                            /* USB Interface
Subclass Code in Class Code  */
172           2,                                                            /* USB Interface
Protocol for the Class Code */
173           ISTR_INTERFACE1,                                  /* Index of Interface string descriptor
*/
174           },
175           {                  /* Tendpoint_descriptor  */
176           sizeof(Tendpoint_descriptor),                     /* Length of endpoint descriptor in
bytes */
177           DESCRIPTOR_ENDPOINT,                        /* Endpoint descriptor Type; */
178           0x81,                                                   /* Endpoint address */
179           3,                                                      /* Endpoint attributes such as
type and direction */
180           SW(3),                                                  /* Maximum packet size from
this endpoint */
181           10,                                                     /* Polling Interval for this
endpoint */
182           },
183           {                  /* THID_descriptor  */
184           sizeof(THID_descriptor),                          /* Length of HID descriptor in bytes */
185           DESCRIPTOR_HID,                                   /* Descriptor type set to HID */
186           SW(0x0100),                                             /* USB HID release number for
the descriptor */
187           HID_COUNTRY_CODE_NOT_SUPPORTED,     /* USB Hardware target country */
188           1,                                                      /* USB HID class descriptors to
follow */
189           DESCRIPTOR_REPORT,                                /* USB HID Report Descriptor type */
190           SW(sizeof(Report_Descr1_LE))                      /* Total length of USB HID Descriptor
*/
191           },
192       };
193
194           Tconfiguration_descriptor const MUSB_CONFIG_DESCRIPTOR_LIST[] =
195           {
```

```
196           &Conf_Descr_Grp1_LE.Conf_Descr          /* References the ONE Configuration descriptor
*/
197           } ;
198
199           THID_descriptor const MUSB_HID_HID_DESCRIPTOR_LIST[] =
200           {
201           &Conf_Descr_Grp1_LE.HID_Descr1
202           };
203           CONST MUSB_HID_REPORT_DESCRIPTOR_LIST[] =
204           {
205           { Report_Descr1_LE, sizeof(Report_Descr1_LE)    }
206           };
207           CONST TFar_List MUSB_HID_PHYSICAL_DESCRIPTOR_LIST[] =
208           {
209           { NULL, 0 }                             /* NULL, not used */
210           };
211
212      Tdevice_state  MUSB_DEVICE_STATE ;    /* Stores USB device state - See USB Lib manual */
213      Tinterface_state       MUSB_INTERFACE_STATE[1] ;      /* Stores USB interface data for this
interface */
214      Tendpoint_state        MUSB_ENDPOINT_STATE[2] ;       /* Stores the status of  Endpt 1 and
Control endpt*/
215
216      CONST TNear_endpoint_state_ptr MUSB_ENDPOINT_STATE_LIST[2] =
217                                                  /* array of pointers to endpoint state
structures  */
218           {
219           &MUSB_ENDPOINT_STATE[0],         /*  Control endpoint */
220           &MUSB_ENDPOINT_STATE[1],         /* Gamepad Interrupt endpoint */
221           };
222
223      uchar8 MUSB_RECEIVE_BUFFER_0[MAX_PACKET_SIZE*2] ;    /* Buffer to use for USB commun */
224
225      CONST struct receive_buffer_list_element MUSB_RECEIVE_BUFFER_LIST[1] =
226           {
227           {
228           MUSB_RECEIVE_BUFFER_0,                          /* USB message buffer  to use; */
229           sizeof(MUSB_RECEIVE_BUFFER_0)                   /* Size of the USB buffer
=2*MAX_PACKET_SIZE*/
230           }
231           } ;
232
233      /*****************************************************************************/
234      /*  Device Interface Section.
*/
235      /*  This section provides the interface between the application and the specific device.
It is the */
236      /*  code that will usually be supplied by the user.
*/
237      /*
*/
238      /*  The user is required to provide 3 device interface functions:
*/
239      /*            1.  Init_Device_Interface
*/
240      /*            2.  Poll_Device_Interface
*/
241      /*            3.  Suspend_Device_Interface
           */
242      /*   If the device is to wake up the host after being suspended, an interrupt  service
routine       */
243      /*   will be required.
*/
244      /*****************************************************************************/
245
246      /*****************************************************************************/
247      /*  TGamePort is the union of  individual bit field variables and one byte to define
*/
248      /*  the port pins on the gamepad.  The union of bit variables is written in the same order
*/
249      /* as the buttons are connected to the JB2 hardware Port A pins.
*/
250      /*****************************************************************************/
251      typedef union    {
252           struct
253           {
254           BITFIELD bfButtonLU:1 ;                 /* This stores the state of S1 from Figure 5.
*/
255           BITFIELD bfButtonRU:1 ;                 /* This stores the state of S2 from Figure 5.
*/
256           BITFIELD bfButtonLD:1 ;                 /* This stores the state of S3 from Figure 5.
*/
257           BITFIELD bfButtonRD:1 ;                 /* This stores the state of S4 from Figure 5.
*/
```

```
258           BITFIELD bfUp:1 ;                          /* This stores the state of S5 from Figure 5.
*/
259           BITFIELD bfDown:1 ;                              /* This stores the state of S6 from
Figure 5. */
260           BITFIELD bfLeft:1 ;                              /* This stores the state of S7 from
Figure 5. */
261           BITFIELD bfRight:1 ;                             /* This stores the state of S8 from
Figure 5. */
262           } P1 ;
263           uchar8 B ;                                       /* This stores all the states of
Gamepad buttons */
264           } TGamePort ;
265
266      #define GAMEPAD_bfRight          0x80                /* Mask to read state of right
directional button - S8*/
267      #define GAMEPAD_bfLeft           0x40                /* Mask to read state of left
directional button - S7 */
268      #define GAMEPAD_bfDown          0x20          /* Mask to read state of down directional
button - S6*/
269      #define GAMEPAD_bfUp             0x10                /* Mask to read state of up directional
button - S5*/
270
271      TGamepad_Report       Gamepad_Report ;       /* Gamepad INPUT report buffer    */
272      TGamePort             GamePortOldValue ;     /* Last value read from gamepad port  */
273      TGamePort             GamePortNewValue;       /* Most recent value read from gamepad port
*/
274      boolean                      GamePadChanged ;       /* If ON, last gamepad report has not
been sent */
275      #define ReadGamePort()   (pRegM6805JB2.PORTA.B)    /* Port A has gamepad pins       */
276
277      #define JOYSTICK_MIN   0x00          /* Directional Pad for a set left or down button */
278      #define JOYSTICK_MID   0x40          /* Directional Pad for no buttons set */
279      #define JOYSTICK_MAX   0x7F          /* Directional Pad for a set right or up button */
280
281      /*************************************************************************/
282      /* Name     : Read_Gamepad                                                */
283      /* Inputs   :  None.                                                      */
284      /* Outputs :  Boolean flag.                                                    */
285      /* Purpose : Reads gamepad port, checks to see if the gamepad button states have changed
*/
286      /*                and returns a boolean flag.
           */
287      /*************************************************************************/
288      boolean Read_Gamepad(void)
289           {
290           GamePortNewValue.B = ReadGamePort() ;          /* Read value on Port A for Gamepad */
291           if( GamePortNewValue.B!=GamePortOldValue.B )   /* Check value with old to see if
changed */
292           {
293           GamePortOldValue.B = GamePortNewValue.B ;             /* If changed, set old value to
new value */
294           return TRUE ;                                  /* TRUE informs that value changed */
295           }
296           return FALSE ;                                 /* FALSE informs of no change */
297           }
298
299      /*************************************************************************/
300      /* Name     :  Build_Gamepad_Report
*/
301      /* Purpose :  Reads the game port, and decodes which buttons have been pressed based on the
*/
302      /*                schematic of Figure 2.  If any joystick control is pressed, it is
mapped to one of       */
303      /*                three values [0x00, 0x40, 0x7f].   Buttons not being used for the
joystick are stored */
304      /*                as is into the Gamepad report structure.
*/
305      /*************************************************************************/
306      void Build_Gamepad_Report(void)
307           {
308           uchar8 i ;
309           uchar8 b ;
310
311           b = ReadGamePort();                                        /* Read Port A */
312           Gamepad_Report.buttons = (b & 0x0F);                       /* Mask off values in
S1,S2,S3,S4 */
313
314           switch( b & (GAMEPAD_bfRight|GAMEPAD_bfLeft) )         /* Mask off  S7 and S8 */
315           {
316           case GAMEPAD_bfRight : i = JOYSTICK_MAX ;   break ;    /* If RIGHT set, set to MAX
value */
317           case GAMEPAD_bfLeft   : i = JOYSTICK_MIN ;   break ;   /* If LEFT is set, set to MIN
value */
318           default                                : i = JOYSTICK_MID ;   break ;   /* Else set to
MID */
```

```
319           }
320           Gamepad_Report.X = i ;                                    /* Set X to MAX, MIN, or MID */
321
322           switch( b & (GAMEPAD_bfUp|GAMEPAD_bfDown) )               /* Mask off S5 and S6 */
323           {
324           case GAMEPAD_bfUp   : i = JOYSTICK_MIN ;   break ;            /* If UP is set, set to
MIN value */
325           case GAMEPAD_bfDown : i = JOYSTICK_MAX ;   break ;     /* If  DOWNset, set to MAX
value*/
326           default             : i = JOYSTICK_MID ;   break ;                  /* Else set to
MID */
327           }
328           Gamepad_Report.Y = i ;                                    /* Set Y to MAX, MIN or MID */
329           }
330
331      /*************************************************************************/
332      /*  Name     :  Poll_Gamepad_Device_Interface
*/
333      /*  Purpose :  This function is called by application main routine in its polling loop.  It
reads     */
334      /*                   the gamepad port, and if it has changed builds an INPUT report to send
to the     */
335      /*                   host on next IN token. If the USB port is currently busy, this routine
sets a flag    */
336      /*                   to send the report out on the next opportunity.
*/
337      /*************************************************************************/
338      void Poll_Gamepad_Device_Interface(void)
339           {
340           if( GamePadChanged || Read_Gamepad() )
341           {
342           Build_Gamepad_Report();
343           if( Musb_HID_Write_INPUT_Report(0,(uchar8 *)&Gamepad_Report) )
344           GamePadChanged = 0 ;
345           else GamePadChanged = 1 ;
346           }
347           }
348
349      /*************************************************************************/
350      /* Name     :  Init_Gamepad_Device_Interface
*/
351      * Purpose :  This function is called after JB2 reset. It sets up the data direction
register for    */
352      /*                  inputs on Port A.  It does not adjust the pull down register on Port A
which is   */
353      /*                  set to 0 on reset.
*/
354      /*************************************************************************/
355      void Init_Gamepad_Device_Interface(void)
356           {
357           pRegM6805JB2.DDRA.B = 0 ;                /* Make port A pins as INPUT */
358           MUSB_DEVICE_STATE.bfSelf_Powered = 1 ;  /* Set the USB device to be self powered */
359           MUSB_DEVICE_STATE.nInterfaces = 1 ;      /* Set the number of interfaces to 1 */
360           MUSB_ENDPOINT_STATE[1].bfEndpoint_Type = 3 ;    /* Set endpoint 1 to INTERRUPT */
361           }
362
363      /*************************************************************************/
364      /*  Name :  Suspend_Device_Interface                                      */
365      /*  Purpose :  This function can be used to switch off or power down external hardware when
the  */
366      /*             is not communicating with the MC68HC705JB2.                */
367      /*************************************************************************/
368      void Suspend_Device_Interface(void)
369           {
370             Stop();
371           }
```

How to reach us:
USA/EUROPE/Locations Not Listed: Motorola Literature Distribution,
P.O. Box 5405. Denver, Colorado 80217.(800) 441-2447, (303) 675-2140
MfaxTM: RMFAX0@email.sps.mot.com-TOUCHTONE (602) 244-6609, U.S.and Canada only (800) 774-1848
INTERNET: http://motorola.com/sps
JAPAN: Nippon Motorola Ltd., SPD, Strategic Planning Office,
4-32-1, Nishi-Gotanda, Shinagawa-ku, Tokyo 141, Japan, 03-5487-8488
ASIA PACIFIC: Motorola Semiconductors H.K. Ltd.,
8B Tai Ping Industrial Park, 51 Ting Kok Road, Tai Po, N.T., Hong Kong. 852-266292298

AN1732/D

1ATX45039-0 Printed in USA 12/97 CSI 33749-19 3,000 LITAMCU