Freescale Semiconductor
Application Note

Document Number: AN3565
Rev. 0, 02/2008

USB and Using the CMX USB Stack with 9S08JM Devices

by: William Jiang
Asia & Pacific Operations Microcontroller Division

1 Introduction

Universal serial bus (USB) is a low cost, fast, bidirectional, isochronous, dynamically attachable serial interface consistent with the requirements of the PC platform. It is widely used in the PC connection.

Freescale's 9S08JM series devices include JM60, JM32, JM16, JM8, and their derivatives. These derivatives integrate a USB device controller and help customers interconnect their devices to PCs and/or any other devices with USB On-The-Go (OTG) capability. Freescale provides CMX USB stack software for these devices, which can be freely downloaded at http://www.freescale.com.This all around USB solution makes 9S08JM devices easy to use, and enables rapid time to market for products.

This application note gives an overview of USB and 9S08JM USB device controller. It shows you how to integrate the CMX USB stack into a CodeWarrior project with an HID joystick application.

Contents

freescale™ semiconductor

2 USB Overview

This section provides an overview of USB. For more detailed information on the USB protocol, please refer to USB specification 2.0 at http://www.usb.org. All USB class related documents can be found on this website.

USB is a cable bus that supports data exchange between the host computer and a wide range of simultaneously accessible peripherals. USB allows dynamic attachment and detachment of peripherals. The attached peripherals share USB bandwidth through a scheduled host on token-based protocol. In a USB system, the host is the master with all the peripherals as slaves.

A USB device logically consists of a USB bus interface, a USB logical device, and a function layer. A USB logical device appears to the USB host as a collection of endpoints. Endpoints are grouped into endpoint sets that implement an interface.

Interfaces are views to the function, i.e., the function is a collection of interfaces. The USB host system software manages the device using the default control pipe (endpoint 0). The host client software manages an interface by using pipe bundles (associated with an endpoint set). The host client software requests that data be moved across the USB between a buffer on the host and an endpoint on the USB device. The host controller (or USB device, depending on the transfer direction) packetizes the data prior to the transfer. The host controller also coordinates when to move the packet of data over the USB.

The USB connection topology supports seven tiers. There are five non root hubs, which is the maximum in a communication path between the host and any device.

USB supports three types of bit rates in USB specification:

- version 2.0: high speed (480 MHz)
- full speed (12 MHz)
- low speed (1.5 MHz)

The communication between the host and a peripheral is initiated by the host via USB transfers. 9S08JM devices support full speed bit rate.

USB defines four transfer types:

- Control transfer (bidirectional) — Control transfer is burst, non-periodic, host software-initiated request/response communication, and typically used for command/status operations.
- Interrupt transfer (unidirectional) — Interrupt transfer is low-frequency, bounded-latency communication.
- Isochronous transfer (unidirectional) — Isochronous transfer is periodic, continuous communication between the host and device, typically used for time-relevant information.
- Bulk transfer (unidirectional) — Bulk transfer is non-periodic, large packet burst communication, typically used for data that can use any available bandwidth and can also be delayed until bandwidth is available.

All peripherals must support control transfer. USB low-speed peripherals also support interrupt transfer. It does not support bulk transfer or isochronous transfer. USB full-speed and high-speed peripherals support all four types of transfers.

A USB transfer consists of a few USB transactions. A USB transaction is the smallest unit of a USB transfer and is made up of up to three phases in order: token phase, data phase, and handshake phase. In certain circumstances, the data phase is absent. For example, if the device is unable to complete a USB command the data phase is not present.

The host send requests to a peripheral on what to do by using control transfers. It polls the state of a peripheral by using interrupt transfers. It sends or receives bulk data by using bulk transfers. It exchanges the time related data (e.g. audio, and video) with a peripheral by using isochronous transfer. The USB device can respond to the host requests. It has no capability to initiate any bus transaction.

A USB device has six types of states:

- Attached — A device is attached to the USB. It is not powered by the USB
- Powered — A device is attached to the USB and powered by the USB. This device has not been reset
- Default — A device is attached to the USB and powered and has been reset. It has not been assigned a unique address, the device responds at the default address and is in default state
- Address — If a device is in default state and a unique device address has been assigned, and the device is not configured, then the device is transitioned to address state.
- Configured — If a device is already in address state and configured, and is not suspended, then the device enters configured state in which the host can use the functions provided by the device.
- Suspended — If a device is in powered, default, address, configured, and has no bus activity for 3 ms, it enters suspend state. The host may not use the function of a suspended device.

When a USB device is attached to the USB, the host uses a process of bus enumeration to identify and manage the device state changes. When a USB device is attached to a port, the following actions take place:

- The device is plugged into the host (in attached state). The host provides power to the device with a current limit of 100 mA.
- The host determines low-speed/full-speed capability by pullup resistors connected to the D+ or D– lines. At this point, the device is in the powered state.
- The host sends a reset to the device by setting D+ and D– low for at least 10 ms. If the host removes the reset, the device goes into the default state.
- In the default state, the device is ready to respond to control transfers at endpoint 0. The host communicates with the device by using the default address of 0. The device can draw up to 100 mA from the host.
- The host sends a GET_DESCRIPTOR request to endpoint 0, address 0, to get the device descriptor.
- The eighteen byte device descriptor contains the maximum packet size supported by endpoint 0 and other important information for proper communication.
- The host assigns a unique address to the device by sending a SET_ADDRESS request. The device is in the address state.
- The host sends a GET_DESCRIPTOR request to the new address to read the full device descriptor.
- The host then requests any additional descriptors specified in the device descriptor. Each descriptor begins with its length and type.

- The host assigns a device driver based on the data in the descriptors. Windows® uses the device's vendor ID and product ID to search for an appropriate INF file to determine which drivers to load. If no match file is found, Windows uses a default driver according to class.
- If the device supports multiple configurations, the host sends a SET_CONFIGURATION
- Request to the device to select the desired configuration.

There are various classes defined by the USB: HID, CDC, mass storage, audio, and video. Each of these classes has its class-specific data that are exchanged between the host and device and the related class requests for the host to request the device to execute.

HID class consists primarily of devices that are used by humans to control the computer operation system, such as mouse, keyboard, joystick, slider, knob, throttle, bar code reader, etc.

HID class specification defines the process for HID class driver to extract data from USB devices by introducing the HID class descriptors (report descriptors, physical descriptors, and physical descriptor sets), HID descriptors and HID class specific requests as well.

HID report descriptors are self-describing data structures containing different items with associated tags, types, sizes, and data, that enable the host to recognize and manage all data report that are coming through the USB.

HID physical descriptor is a data structure that provides information about the specific part or parts of the human body that activates a control or controls. A report descriptor may be associated with a physical descriptor using report descriptor's designator index items (indicate that part of the body affects the item). A HID descriptor specifies the number of class descriptors (always at least one report descriptor.) A report descriptor specifies three types of reports: input, output, and feature reports.

A report is the same as a transfer. It returns the structure(s) in which each data field is sequentially represented as described by the report descriptor, and/or physical descriptor. Only input reports are sent via the interrupt in pipe. Feature and output reports must be initiated by the host via the control pipe or an optional interrupt out pipe.

Please refer to Section 5.1.2, "HID Report," for an example of HID report descriptor and joy_report_descriptor.

3 9S08JM USB Device Controller

9S08JM families have similiar USB device controller. It provides a single chip solution for full speed (12 Mbps) USB device applications, and integrates the required transceiver with serial interface engine (SIE), 3.3 V regualtor, endpoint RAM, and other control logics.

The USB device controller includes the following features:

- USB 2.0 compliant
  - — 12 Mbps full speed (FS) data rate
  - — USB data control logic:
    - – Packet identification and decoding/generation
    - – CRC generation and checking
    - – NRZI (non-return-to-zero inverted) encoding/decoding
    - – Bit stuffing
    - – Synchronization detection
    - – End-of-packet detection
- Seven USB endpoints
  - — Bidirectional endpoint 0
  - — Six unidirectional data endpoints configurable as interrupt, bulk, or isochronous
  - — Endpoints 5 and 6 support double buffering
- USB RAM
  - — Total of 256 bytes of buffer RAM shared between system and USB module
  - — RAM may be allocated as buffers for USB controller or extra system RAM resource
- USB reset options
  - — USB module reset generated by MCU
  - — Bus reset generated by the host, which triggers the CPU interrupt
- Suspend and resume operations with remote wakeup support
- Transceiver features
  - — Converts USB differential voltages to digital logic signal levels
- On chip USB pullup resistor
- On chip 3.3 — V regulator

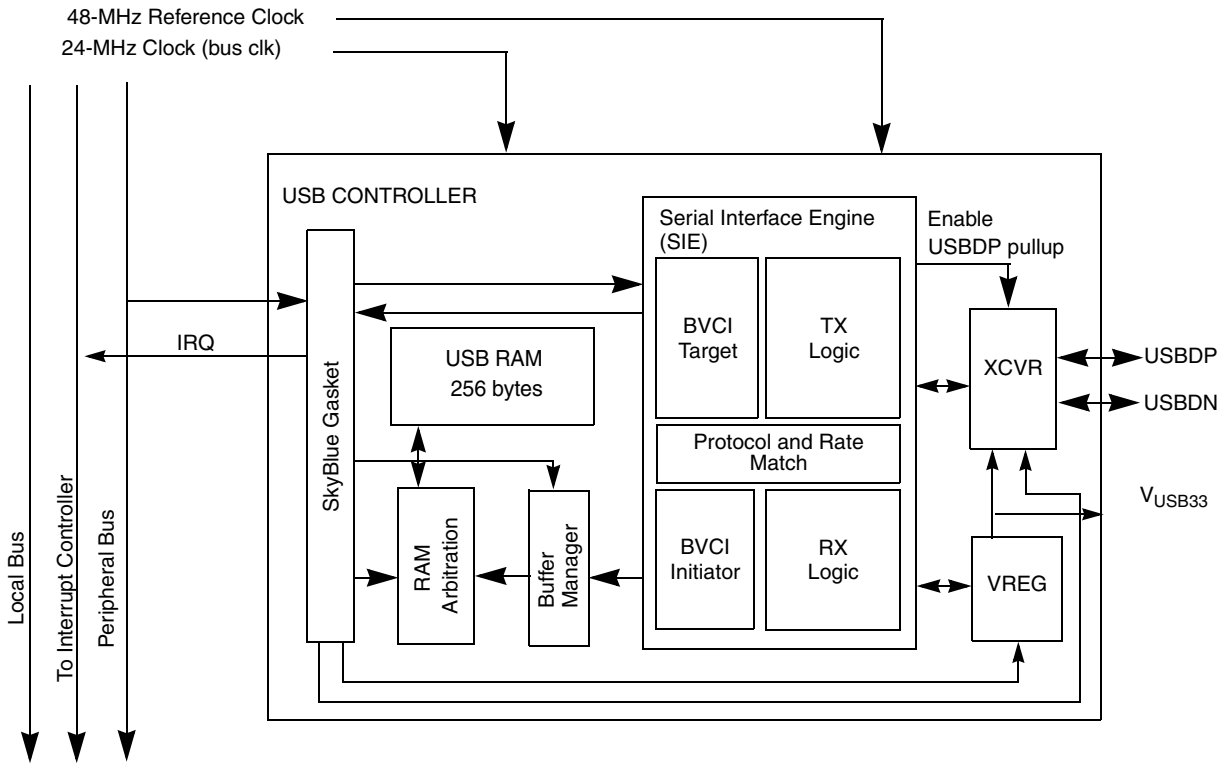

Figure 1. USB Device Controller Block Diagram

Figure 1 shows the USB device controller block diagram.

For more information about the USB device controller, please refer to *MC9S08JM60 Series Data Sheet.*

4 CMX USB Stack

CMX USB stack is developed by CMX System, Inc. It can be downloaded at: http://www.freescale.com/usb.

This section provides the following information:

- The architecture of the CMX USB stack and stack examples.
- Summary of all source files for the CMX USB Stack
- Introduction of USB Driver, HID Class Driver and the resource usage of the CMX USB stack.

4.1 CMX USB Stack Architecture

Figure 2 shows the CMX USB stack architecture. The stack architecture includes three layers:

- USB driver

  The USB driver sits on the USB controller. It manages the USB protocol and the standard USB device requests, and reports 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.

- Class drivers

  The HID class driver manages the HID protocols, while 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).

- USB applications

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

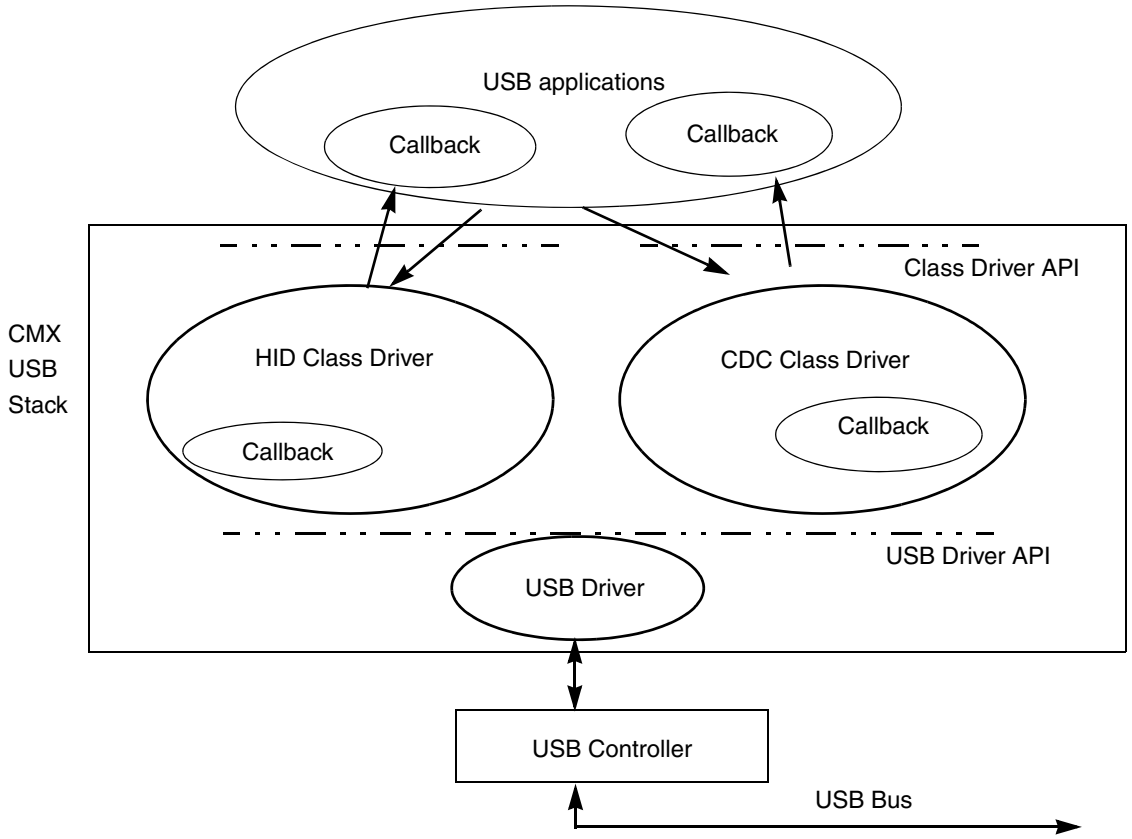

Figure 2. CMX USB Stack Architecture

4.2 CMX USB Stack Examples

There are HID class demonstration and CDC class demonstration included in the CMX USB stack package. The HID class demonstration includes an HID keyboard, HID mouse, and HID generic device. The CDC class demonstration includes a USB to UART bridge and a CDC terminal.

Table 1. CMX USB Stack Demonstrations with 9S08JM Devices

Demos	Description
HID keyboard device	Emulates a keyboard
HID mouse device	Emulates a mouse
HID generic device	Implements a user-defined HID device
HID PC software (VC++)	Used to communicate with the HID generic device
USB to UART bridge demo (CDC-serial)	Creates a virtual COM port on PC and exchanges data via USB
CDC terminal (CDC-terminal)	Emulates a terminal and can recognize and execute simple commands

For more information about these demonstrations, please refer to USB Device Demo_hc08sjm60.pdf in the Freescale USB Lite by CMX downloaded.

4.3 CMX USB Stack Files

CMX USB stack contains many files in different folders. Table 2 lists all source files along with their descriptions.

Table 2. CMX USB Stack Files for 9S08JM Devices

Files		Description
USB Driver files	usb.c	Low-level USB driver and usb_it_handler() interrupt service routine
	usb.h	Header file for USB driver
	usb_config.h	USB driver compile time configuration parameters
HID class driver files	hid.c	HID device layer including state machine
	hid.h	Header file for hid.c
	hid_usb_config.c	Data structures, callbacks for keyboard, mouse, and generic HID demonstrations
	hid_usb_config.h	Header file for hid_usb_config.c
HID class driver demo files	hid_generic.c	Demonstration application of a generic report structure and is used to communicate with the board via the HID PC application
	hid_generic.h	Header file for hid_generic.c
	hid_kbd.c	HID keyboard demonstration
	hid_kbd.h	Header file for hid_kbd.c
	hid_mouse.c	HID mouse demonstration
	hid_mouse.h	Header file for hid_mouse.c
	hid_main.c	Entry point for HID device demonstrations

Table 2. CMX USB Stack Files for 9S08JM Devices (continued)

CDC class driver files	usb_cdc.c	CDC class driver routines
	usb_cdc.h	Header file for usb_cdc.c
	cdc_usb_config.c	USB configuration structures for the serial to USB CDC demonstrations
	cdc_usb_config.h	Header file for cdc_usb_config.c
CDC class driver demo files	cdc_main.c	Main loop, reads/writes the UART, and transfers data to the USB stack

4.4 USB Driver

The USB driver manages USB device controller, sets up USB buffer descriptor table (BDT) in USB RAM space for different transfers, monitors USB packets, and encodes and decodes USB packets based on USB transactions and transfers. The driver also manages the standard USB device requests. To manage the standard USB device requests, callbacks must be defined by its user to return standard USB descriptors. HID demonstration code implements these callbacks in hid_usb_config.c.

The USB driver also requires macros to be defined by users. These macros must be defined in usb_config.h or any other user header file that is included in usb_config.h:

EP0_PACKET_SIZE — endpoint 0 packet size in bytes, normally is defined to 64 for full speed USB device. HID demonstration applications define it in hid_usb_config.h.

This section describes the driver callbacks and the driver APIs that are used. For the detailed internal information of the USB driver, please consult AN3492, *USB and Using the CMX USB Stack.*

4.4.1 USB Driver Callbacks

This section describes all callbacks that are called by the USB driver and shall be defined. For examples of how to define these callbacks, please refer to hid_usb_config.c and hid.c.

4.4.1.1 get_device_descriptor

Prototype: void* get_device_descriptor(void);

This callback must return a pointer to the user-defined 18 byte USB device descriptor.

4.4.1.2 is_cfgd_index

Prototype: hcc_u8 is_cfgd_index(hcc_u16 cndx); where “cfgd” stands for configuration descriptor.

This callback must return a non-zero if the specified configuration is a valid user defined configuration, and return 0 otherwise. This configuration is indexed by cndx. So the configuration index is the same as the configuration number specified in the configuration descriptor. This convention also applies to the following sections for other USB descriptors until otherwise specified.

4.4.1.3 get_cfg_descriptor

Prototype: void *get_cfg_descriptor(hcc_u8 cndx); where “cfg” stands for configuration descriptor.

This callback must return a pointer to the user defined configuration descriptor.

4.4.1.4 is_str_index

Prototype: hcc_u8 is_str_index(hcc_u8 sndx); where “str” stands for string descriptor.

This callback must return a non-zero if the specified index is the valid index of a user defined string descriptor and return 0 otherwise.

4.4.1.5 get_str_descriptor

Prototype: void *get_str_descriptor(hcc_u8 sndx);

This callback must return a pointer to the user defined string descriptor specified by the string index.

4.4.1.6 is_ifc_ndx

Prototype: hcc_u8 is_ifc_ndx(hcc_u8 cndx, hcc_u8 indx, hcc_u8 iset); where “ifc” stands for interface, “ndx” stands for index.

This callback must return a non-zero if the specified interface exists, and return 0 otherwise. This interface is indexed by an index set that comprises of configuration index (cndx), interface index (indx) and the index of interface alternate setting (iset).

4.4.1.7 is_ep_ndx

Prototype: hcc_u8 is_ep_ndx(hcc_u8 cndx, hcc_u8 indx, hcc_u8 iset, hcc_u8 endx); where “ep” stands for Endpoint.

This callback must return a non-zero if the specified endpoint exists. This endpoint is index by a index set that comprises of configuration index (cndx), interface index (indx), interface alternate setting index (iset) and the endpoint index.

4.4.1.8 get_ep_descriptor

Prototype: void *get_ep_descriptor(hcc_u8 cndx, hcc_u8 indx, hcc_u8 iset, hcc_u8 endx);

This callback must return a pointer to the endpoint descriptor specified by the index set as described above.

4.4.1.9 usb_wakeup_event

Prototype: void usb_wakeup_event(void);

This callback is called if a USB resume event occurs.

4.4.1.10 usb_suspend_event

Prototype: void usb_suspend_event(void);

This callback is called if a USB suspend event occurs.

4.4.1.11 usb_reset_event

Prototype: void usb_reset_event(void);

This callback is called if a USB reset event occurs.

4.4.1.12 usb_bus_error_event

prototype: void usb_bus_error_event(void);

This callback is called if a USB module error event occurs.

4.4.1.13 usb_ep0_callback

prototype: callback_state_t usb_ep0_callback(void);

This callback is called if the current received USB device request is not a standard request for a device, or not any of the following standard request for a device:

SET_ADDRESS, GET_DESCRIPTOR for a device descriptor, a configuration descriptor, and string descriptor, GET_CONFIGURATION, SET_CONFIGURATION.

It is also called if the USB device request is not any of the following standard requests for an endpoint: CLEAR_FEATURE for ENDPOINT_HALT feature selector. HID driver implements this callback in hid.c as ep0_hid_callback.

4.4.2 USB Driver APIs

This section describes utility macros and all USB driver application programming interface routines (APIs) that may be used. Utility macros are defined in usb.h and usb driver APIs are defined in usb.c. It is declared in usb.h.

4.4.2.1 Utilitiy Macros

Utility macros that may be used are summarized in Table 3.

Table 3. USB Driver Utility Macros

Macros	Description
USB_FILL_DEV_DESC	Fill a 18-byte device descriptor with given values
USB_FILL_CFG_DESC	Fill a 9-byte configuration descriptor with given values
USB_FILL_IFC_DESC	Fill a 9-byte interface descriptor with given values
USB_FILL_EP_DESC	Fill a 7-byte endpoint descriptor with given values

4.4.2.2 usb_init

Prototype: hcc_u8 usb_init(void) ;

This function initializes the internal data structures of the USB driver, enables USB interrupts and the USB module. It returns 0 if successful; 1 if not.

4.4.2.3 usb_send

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 bytes to the host. The data to be transmitted is pointed to by data parameter.

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

All packet transmissions on the USB are started by the host. The device must know how many bytes are transferred during a transfer. The host always tells the device how many bytes it can receive. This is specified by req_length parameter. On the other hand, the device may have less data ready (tr_length).

If a transaction is complete and hence the endpoint buffer is empty, the USB driver notifies 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 report state and prepare more data to be sent. In general, it can remain NULL. Please refer to usb_ep0_callback as an example of using usb_send.

4.4.2.4 usb_receive

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

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

After the specified number of data is received from host by the USB driver, the callback function f is called. Please refer to usb_ep0_callback for an example of using usb_receive.

4.4.2.5 usb_stop

Prototype: void usb_stop(void);

This function disables the USB module and stops the USB driver.

4.4.2.6 usb_ep_is_busy

Prototype: hcc_u8 usb_ep_is_busy(hcc_u8 ep);

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

4.4.2.7 usb_ep_error

Prototype: hcc_u8 usb_ep_error(hcc_u8 ep);

This function returns the endpoint-specific error code that is defined by the USB driver. For all error code, please refer to USBEPERR_xx macro definitions in usb.h.

4.4.2.8 usb_get_done

Prototype: hcc_u32 usb_get_done(hcc_u8 ep);

This function returns the number of bytes that were transferred for the given endpoint ep.

4.4.2.9 usb_get_state

Prototype: hcc_u8 usb_get_state(void);

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

4.4.2.10 usb_abort_ep

Prototype: void usb_abort_ep(hcc_u8 ep);

This function stops the ongoing transfer on the selected endpoint (early stop) and sets the corresponding error flag for the endpoint. Early stop is possible only if there is pending data remaining in the endpoint buffer. If not, then the transfer stops anyway.

4.4.2.11 usb_get_rx_pptr

Prototype: hcc_u8 *usb_get_rx_pptr(hcc_u8 ep);

This function returns a pointer to the endpoint buffer containing the data of the last received packet.

4.5 HID Class Driver

This section describes the data structure, descriptor_info_t, which must be initialized by HID class driver callbacks and also describes all HID class driver APIs which may be used.

descriptor_info_t structure is defined as below:

```
typedef struct
{
      void *start_addr;
      hcc_u16 size;
}
descriptor_info_t;
```

start_addr — points to a USB descriptor defined by users;

size — indicates the size of a USB descriptor;

4.5.1 HID Class Driver APIs

HID Class driver requires macros defined in the hid_usb_config.h:

HID_IT_EP_NDX — interrupt IN endpoint index/number

Users must also define the macros required by the USB driver in hid_usb_config.h:

EP0_PACKET_SIZE — endpoint 0 packet size in bytes, normally is defined to 64 for a full-speed USB device. The HID class driver supports three types of reports: input report, output report, and feature report. They are defined as below:

```
typedef enum
{
      rpt_in,                                                    /* input report */
      rpt_out,                                                  /* output report */
      pt_feature                                               /* feature report */
} hid_report_type;
```

The HID class driver supports up to MAX_NO_OF_REPORTS HID reports with their buffer size of up to MAX_REPORT_LENGTH in bytes. They are defined as 2 and 8 in hid.c. Users can change them if required.

All HID class driver APIs are defined in hid.c and summarized hereafter. For an example of using these APIs, please refer to the HID process function hid_joy described in Section 5.2, “Modules and Functions.”

4.5.2 Utility Macros

These macros are defined in hid_usb_config.c and summarized in Table 4.

Table 4. HID Class Driver Utility Macros

Macros	Description
USB_FILL_HID_DESC	Fill an HID descriptor with given values
USB_FILL_OTG_DESC	Fill an OTG descriptor with given values

4.5.3 HID_init

Prototype: void HID_init(hcc_u16 default_idle_time, hcc_u8 ifc_number);

This function initializes the HID class driver. The default_idle_time is the idle time in ms, used to silence a particular report on the interrupt in pipe until a new event occurs or the specified amount of time passes. Hence, it limits the reporting frequency of an interrupt in the endpoint.

The specification recommends an idle time of 500 ms, for the keyboards, 0 ms for the mouse. The ifc_number specifies the interface number to start the HID protocol on.

4.5.4 hid_process

Prototype: void hid_process(void);

This function walks through the reports and sends pending rpt_in reports to the host. Reports are transmitted on endpoint HID_IT_EP_NDX, which defaults to 1 and can be changed in hid_usb_config.h.

4.5.5 hid_add_report

Prototype: hcc_u8 hid_add_report(hid_report_type type, hcc_u8 id, hcc_u8 size);

This function defines a new report/transfer for the HID driver and returns a report number to be used with write, read, and pending function calls.The type can be one of the rpt_in, rpt_out, or rpt_feature as defined in hid.h. “id” is the report ID created in a report descriptor. “size” is the number of bytes in the report (must match the report descriptor).

4.5.6 hid_write_report

Prototype: void hid_write_report(hcc_u8 r, hcc_u8 *data);

This function writes/updates report data that parameter data points to into a specified report returned by hid_add_report. The report is marked pending, so it is transmitted on the next call to hid_process().

It is called if the state of a report item/control is changed.

4.5.7 hid_read_report

Prototype: void hid_read_report(hcc_u8 r, hcc_u8 *data);

This function reads report data that second parameter data points to from a specified report returned by hid_add_report. It then clears report’s pending flag.

4.5.8 hid_report_pending

Prototype: hcc_u8 hid_report_pending(hcc_u8 r);

This function returns the pending status of the report r.

4.5.9 HID Class Driver Callbacks

The HID class driver includes USB driver callbacks and HID class driver extended callbacks.

The HID class driver is the user of the USB driver. It must implement all the USB driver callbacks. Different users may have different USB descriptors to be returned by their callbacks. It is more flexible to define all USB driver callbacks by upper-layer applications. In addition, each HID class driver has its own exteneded callbacks. These callbacks must also be implemented by upper-layer applications. All callbacks are defined in hid_usb_config.c.

This section describes HID class driver extended callbacks. Please refer to the aforementioned USB driver callbacks for USB driver callbacks.

4.5.9.1 Get_hid_descriptor

Prototype: descriptor_info_t *get_hid_descriptor(void);

This callback must return a pointer to a user-defined descriptor_info_t structure that points to a user-defined HID descriptor. It also must set the field size of the descriptor_info_t structure to 9.

4.5.9.2 Get_report _descriptor

Prototype: descriptor_info_t *get_report_descriptor(void);

This callback must return a pointer to a user-defined descriptor_info_t structure that points to a user-defined report descriptor. It also must set the field size of the descriptor_info_t structure to the size of the report descriptor.

4.5.9.3 Get_physical_descriptor

Prototype: descriptor_info_t *get_physical_descriptor(hcc_u8 id);

This callback must return a pointer to a user-defined descriptor_info_t structure that points to the user-defined physical descriptor specified by the physical index id. It also must set the field size of the descriptor_info_t structure to the size of the physical descriptor.

4.5.9.4 Got_usb_reset

Prototype: void got_usb_reset(void);

This callback is called if the USB reset occurs.

4.6 Resource Usage

CMX stack occupies a very small amount of memory in bytes as listed in Table 5. This data is calculated directly from the linker map file generated by CodeWarrior 6.1. The size of HID demo counts in the size of all of its components including the USB driver, HID driver, hid-keyboard, hid-mouse, and hid-generic device. The same rule applies to the size of CDC-terminal demo and CDC-serial demo. Flash memory includes code, initialized data, and constant variables like USB descriptors. RAM includes initialized data, uninitialized data, heap, and stack which resides in the system RAM area, and as well as the endpoint buffers in the USB RAM area. But for the drivers, the RAM size does not include the endpoint buffers because it depends on the upper lay applications.

Table 5. CMX USB Stack Memory Usage

Memory	USB Driver	HID Driver	CDC Driver	HID Demo	CDC-Terminal Demo	CDC-Serial Demo
Flash	3436	1613	575	7567	6068	5994
RAM	111	34	112	429	573	703

5 HID Joystick Demonstration

The HID joystick demonstration demonstrates the two axis of X and Y, one throttle, one hat switch, and four buttons.

The demonstration boards DEMOJM for 9S08JM devices contain a 3-axis accelerometer, the MMA7260Q. The MMA7260Q is low-cost capacitive micro-machined accelerometer that features signal conditioning, a 1 pole low-pass filter, temperature compensation, and g-select that allows for the selection among four sensitivities (1.5g/2g/4g/6g).

X output of the accelerometer is connected to the ADC channel 3. Y output is connected to the ADC channel 0. Z output is connected to the ADC channel 1. This demonstration application uses the accelerometer to simulate the X, y, and throttle movement:

- Tilt the board left or right to move X left or right
- Tilt the board forward or backward to move Y up or down
- Pull up or down the board to move throttle up or down

In addition, it uses potentiometer (W1) to simulate the movement of the hat switch control. W1 is connected to the ADC channel 9.

To build an HID joystick application, some data structures like USB descriptors must be defined.

This section describes the required data structures and functions. It describes how to create an HID joystick project from scratch with CodeWarrior and how to integrate the CMX USB stack with this application project. This section also covers the procedure to run the HID joystick demonstration.

5.1 Data Structures

5.1.1 USB Descriptors

The USB descriptors include a standard USB device descriptor, configuration descriptor, interface descriptor, HID class descriptor, endpoint descriptor, string descriptor, and HID report descriptor.

If a device works as a USB OTG B device, the OTG descriptor must be defined as an extention to the standard configuration descriptor. In this demonstration, the OTG descriptor is also defined for such purpose.

All USB descriptors are defined in hid_usb_config.c.

The USB device descriptor for the HID joystick is defined as below:

```
const hcc_u8 joy_device_descriptor[] = {
      USB_FILL_DEV_DESC(0x0101, 0, 0, 0, EP0_PACKET_SIZE,
      JOY_VENDOR_ID, JOY_PRODUCT_ID,JOY_DEVICE_REL_NUM,
      1, 2, 3, 1)
};
```

The USB configuration descriptor is defined as below :

```
#define CONFIGURATION_VALUE 1
const hcc_u8 joy_config_descriptor[] =
{
      USB_FILL_CFG_DESC(9+3+9+9+7, 1, CONFIGURATION_VALUE, 4, CFGD_ATTR_SELF_PWR, 0x10),
                                                    /* Configuration descriptor */
      USB_FILL_OTG_DESC(1, 1),                              /* OTG descriptor */
      USB_FILL_IFC_DESC(JOY_IFC_INDEX, 0, 1, 0x3, 0x0, 0x0, 5),
                                        /* Interface descriptor: HID,0, 0 (3/1/2) */
      USB_FILL_HID_DESC(9, 0x0100, 0x0, 1, 0x22, sizeof(joy_report_descriptor)),
                                                    /* HID descriptor */
      USB_FILL_EP_DESC(0x1, 1, 0x3, 8, 0x20),               /* Endpoint descriptors */
};
```

To enable the host Windows system to recognize a device as a joystick or a game pad, a self-explained HID report descriptor is to be sent to the host. It must declare its top-level collection as belonging to the generic desktop page (0x01), and deploy usage joystick (0x04) or game pad (0x05).

An HID report descriptor is defined to tell the host to interpret the report data format coming through the interrupt pipes as the following format:

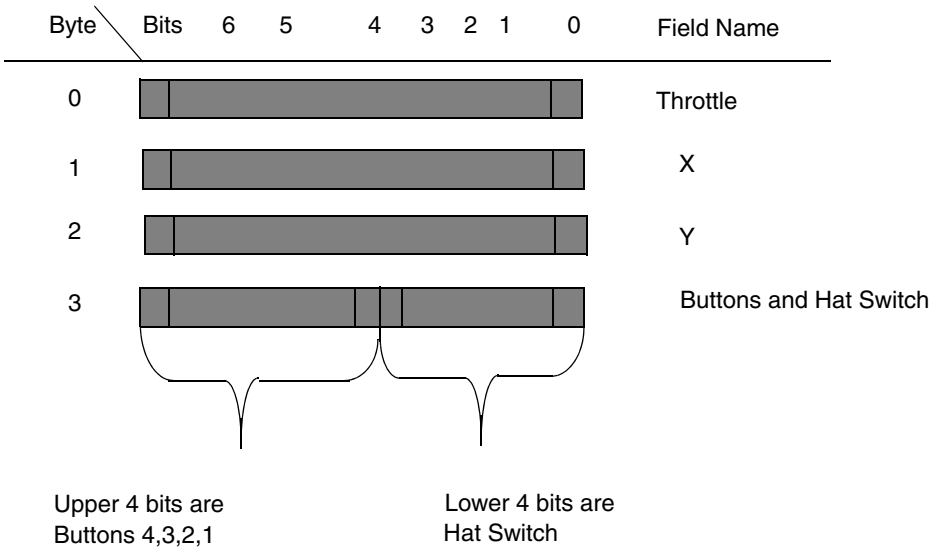

Figure 3. HID Joystick Report Structure

Shown below is the HID report descriptor that describes the aforementioned report structure, which can be created by USB HID descriptor tool DT.exe (http://www.usb.org/developers/hidpage/dt2_4.zip):

```
const hcc_u8 joy_report_descriptor[76] = {
      0x05, 0x01, // USAGE_PAGE (Generic Desktop)
      0x09, 0x04, // USAGE (Joystick)
      0xa1, 0x01, // COLLECTION (Application)
      0x05, 0x02, // USAGE_PAGE (Simulation Controls)
      0x09, 0xbb, // USAGE (Throttle)
      0x15, 0x81, // LOGICAL_MINIMUM (-127)
      0x25, 0x7f, // LOGICAL_MAXIMUM (127)
      0x35, 0x00, // PHYSICAL_MINIMUM (0)
      0x46, 0xff, 0x00, // PHYSICAL_MAXIMUM (255)
      0x75, 0x08, // REPORT_SIZE (8)
      0x95, 0x01, // REPORT_COUNT (1)
      0x81, 0x02, // INPUT (Data,Var,Abs)
      0x05, 0x01, // USAGE_PAGE (Generic Desktop)
      0x09, 0x01, // USAGE (Pointer)
      0xa1, 0x00, // COLLECTION (Physical)
      0x09, 0x30, // USAGE (X)
      0x09, 0x31, // USAGE (Y)
      0x95, 0x02, // REPORT_COUNT (2)
      0x81, 0x02, // INPUT (Data,Var,Abs)
      0xc0, // END_COLLECTION
      0x09, 0x39, // USAGE (Hat switch)
      0x15, 0x00, // LOGICAL_MINIMUM (0)
      0x25, 0x03, // LOGICAL_MAXIMUM (3)
      0x35, 0x00, // PHYSICAL_MINIMUM (0)
      0x46, 0x0e, 0x01, // PHYSICAL_MAXIMUM (270)
      0x65, 0x14, // UNIT (Eng Rot:Angular Pos)
      0x75, 0x04, // REPORT_SIZE (4)
```

```
      0x95, 0x01, // REPORT_COUNT (1)
      0x81, 0x02, // INPUT (Data,Var,Abs)
      0x05, 0x09, // USAGE_PAGE (Button)
      0x19, 0x01, // USAGE_MINIMUM (Button 1)
      0x29, 0x04, // USAGE_MAXIMUM (Button 4)
      0x25, 0x01, // LOGICAL_MAXIMUM (1)
      0x15, 0x00, // LOGICAL_MINIMUM (0)
      0x75, 0x01, // REPORT_SIZE (1)
      0x95, 0x04, // REPORT_COUNT (4)
      0x81, 0x02, // INPUT (Data,Var,Abs)
      0xc0 // END_COLLECTION
};
```

For all other USB descriptors, please refer to the source code of hid_usb_config.c.

5.1.2 HID Report

This structure is defined to store the real HID report data transferred via interrupt pipe to the host. Its data format is described by the HID report descriptor joy_report_descriptor and it has four bytes. It is shown as below where HID_JOY_REPORT_IN_SIZE is defined to four:

```
static signed char hid_joy_report_in[HID_JOY_REPORT_IN_SIZE] = {
0};
To facilitate the access to the HID joystic report structure, the following macros are defined
in hid_joy.c:
// Get the corresponding fields of the HID joystic report.
#define DIR_REP_THROTTLE(h) ((h)[0])
#define DIR_REP_BUTTONS(h) ((h)[3]) /* upper 4 bits */
#define DIR_REP_HAT(h) ((h)[3])
#define DIR_REP_X(h) ((h)[1])
#define DIR_REP_Y(h) ((h)[2])
#define GET_RPT_IN_HAT(h) (DIR_REP_HAT(h) & 0x0f)

// Change the corresponding fields of the HID joystic report.
#define CHANGE_RPT_IN_X(h,x) DIR_REP_X(h) = x
#define CHANGE_RPT_IN_Y(h,y) DIR_REP_Y(h) = y
#define CHANGE_RPT_IN_THROTTLE(h,thrtl) DIR_REP_THROTTLE(h) = thrtl
#define CHANGE_RPT_IN_HAT(h,hat) DIR_REP_HAT(h) = (DIR_REP_HAT(h) & 0xf0) | (hat & 0x0f)

// Set the corresponding fields of the HID joystic report.
#define SET_RPT_IN_BUTTONS(h, btn) DIR_REP_BUTTONS(h) |= (1<<(btn+3))

// Clear the corresponding fields of the HID joystic report.
#define CLEAR_RPT_IN_BUTTONS(h, btn) DIR_REP_BUTTONS(h) &= ~(1<<(btn+3))
```

Where the parameter h stands for the HID joystick report, i.e., hid_joy_report_in and btn stands for the number of a button, hat for hat switch control field, thrtl for throttle field, and x and y for X and Y field of the HID joystick report (refer to Figure 3).

5.1.3 Descriptor_infor_t

This structure is used by HID callbacks to return the pointer to a HID descriptor and is described in Section 4.5, “HID Class Driver .”

5.2 Modules and Functions

This section summarizes all the software modules and the related functions.

Table 6. HID Joystick Demonstration Software Modules

Software Modules	Description
Hid_joy.c	This module defines HID report process routines that may call HID driver class APIs: Hid_joy, joy_got_reset, joy_scan_matrix, joy_start_mS_timer, TPMCHnEvent
Adc.c	The adc driver that contains adc operation routines: ADC_Init, ADC_Cvt, ADC_Poll.
Hid_usb_config.c	This module defines all callbacks required by HID class driver and USB driver; also defines the required data structures like USB device descriptors; also initializes USB driver: get_hid_descriptor, get_report_descriptor, get_physical_descriptor, get_device_descriptor, is_cfgd_index, get_cfg_descriptor, is_str_index, get_str_descriptor, is_ifc_ndx, is_ep_ndx, get_ep_descriptor, usb_ep0_callback, usb_cfg_init, got_usb_reset
Hid_main.c	This module is the entry point of the application that call target module to initialize the target hardware and then call hid_joy module to process HID report process
Target.c	This module defines all target related functions: initialize the clock, parallel port as well as interrupt: hw_init, init_clock, init_board,irq_restore,_irq_disable.

5.2.1 Application Entry Point — main

The main() function is shown as below:

```
{
/* Initialize the hardware.*/
hw_init();

/* Initialize the USB driver. */
usb_cfg_init();

/* Process HID report requests.*/
hid_joy();
return 0;
}
```

5.2.2 Initialize the Chip and Board — hw_init

The function hw_init first disables the watchdog and then configures the CPU to 48 MHz. It also configures the parallel ports for the LEDs output and keyboard inputs on the board and finally enables the interrupt. Its code is listed below:

```
{
      Disable watchdog.
      Call init_clock() to configure the MCG so that the cpu works at 48MHz;
      Call init_board() to configure the parallel ports for LEDs output and buttons inputs;
      Enable interrupts;
}
```

5.2.3 Initialize the USB Driver — usb_cfg_init

The function usb_cfg_init simply calls usb_init to initialize the USB driver.

5.2.4 HID Process — hid_joy

The hid_joy function is the key function to process HID reports (refer to Figure 4):

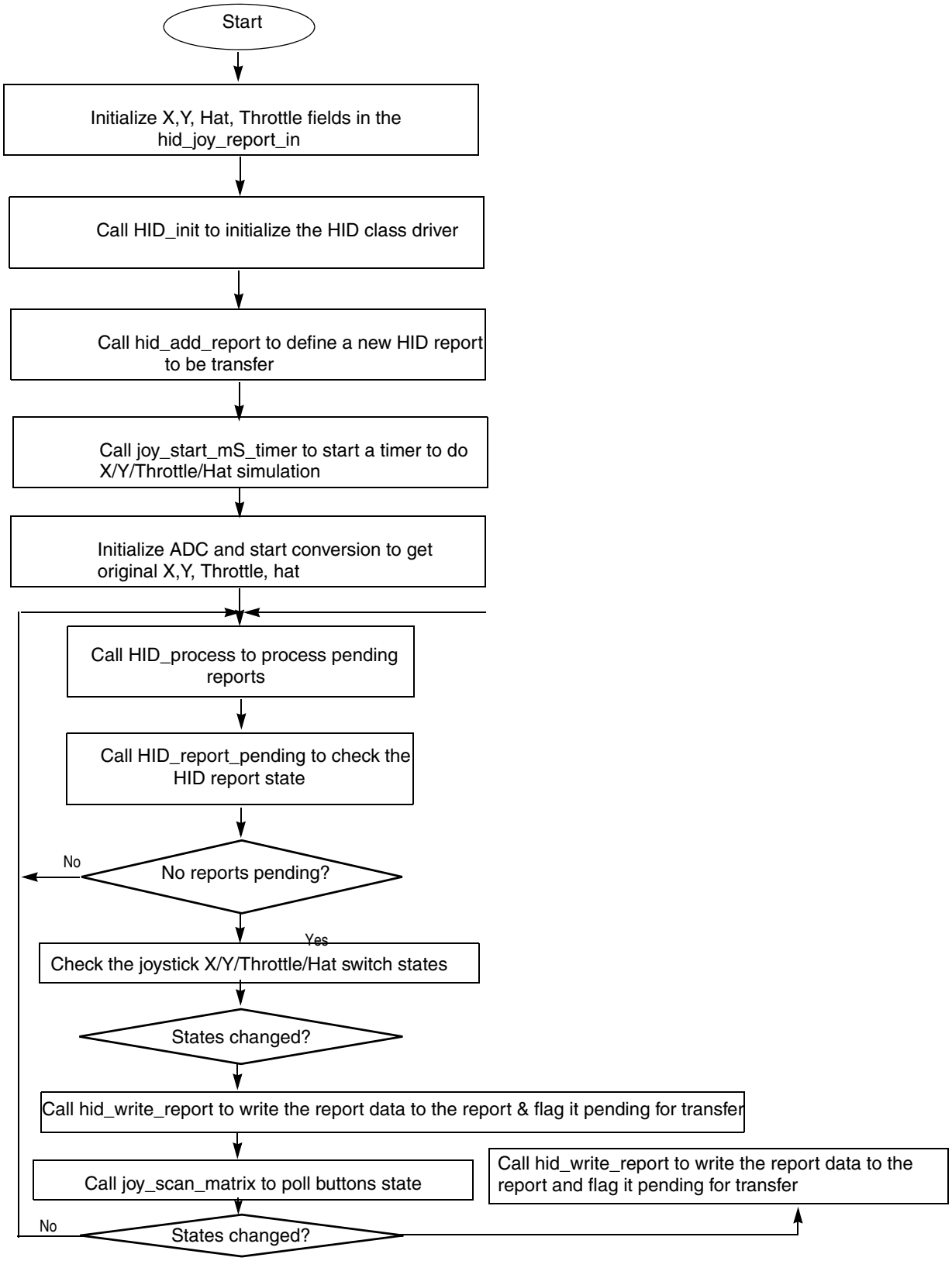

Figure 4. Hid_joy Control Flow

```
{
      const int delta=5;
      hcc_u8 in_report;
      hcc_u8 cur_state,last_state;

      /* Per requirement of Windows HID-compliant game controller,
      * Hat switch controls must report a Null value when not pressed.
      * When pressed, the logical minimum value represents north,
      * and increasing logical values represent directions equally spaced clockwise around the
      compass
      */
      CHANGE_RPT_IN_HAT(hid_joy_report_in,-1);
      DIR_REP_X(hid_joy_report_in) = 0;
      DIR_REP_Y(hid_joy_report_in) = 0;
      DIR_REP_THROTTLE(hid_joy_report_in) = -127;

      /* Initialize the HID driver */
      HID_init(500, JOY_IFC_INDEX);

      /* Define a new report for the joystick */
      in_report=hid_add_report(rpt_in, 0, HID_JOY_REPORT_IN_SIZE);

      /* Initialize ADC. */
      ADC_Init();
      /* Get initial X value with scaled result.
      * AdcResult of 0xFFF --> X of 127, and 0 --> X of -127 */
      ADC_Cvt(3);
      x = (AdcResult >>4);

      /* Get initial Y value with scaled result. */
      ADC_Cvt(0);
      y = (AdcResult >>4);

      /* Get initial throttle value with scaled result.
      * throttle is corresponding Z output of accerometer. */
      ADC_Cvt(1);
      throttle = (AdcResult >>4);
      /*
      * Get hat value. */
      ADC_Cvt(9);
      hat = (AdcResult>>10);
      /* Initialize the timer. */
      joy_start_mS_timer(100);

      while(!device_stp)
      {
          /* Handle the pending HID reports. */
          hid_process();

          /* Check if there is not any pending report. */
          if (!hid_report_pending(in_report))
          {
              if(xy_state_changed) {
                  xy_state_changed = FALSE;

                  /* Write report data. */
```

```
                hid_write_report(in_report, hid_joy_report_in);
            }
            if(hat_state_changed) {
            hat_state_changed = FALSE;
            /* Write report data. */
            hid_write_report(in_report, hid_joy_report_in);
        }
        /* Look for buttons state. */
        cur_state = joy_scan_matrix();

        /* If the status of some buttons has been changed, update input
        * report. */
        if (cur_state != last_state)
        {
            hid_write_report(in_report, hid_joy_report_in);
            last_state=cur_state;
        }
    }
    return(0);
}
```

5.2.5 Initialize the ADC — ADC_Init

The function ADC_Init configures the ADC module clock as 1.5 MHz, and conversion mode to single conversion.

```
{
    ADCCFG = 0x61; //busclk/2, Div by 8,ADCK = 1.5MHz
        /* 0b0000000 0
        * ||||||| |__ bit0,1: ADICLK : input clock select
        * |||||||_|
        * ||||||_____ bit2,3: MODE : Conversion Mode selection
        * |||||_|
        * ||||______ bit4: ADLSMP: long sample time configuration
        * |||_______ bit5,6 : ADIV: Clock Divide Select
        * ||______|
        * |_________ bit7: ADLPC: Low power configuration
        */

    ADCSC2 = 0x00; //
        /* 0b00000000
        * ||||||||__ bit0:
        * |||||||___ bit1:
        * ||||||____ bit2:
        * |||||_____ bit3:
        * ||||______ bit4: ACFGT: Compare function greater than enable
        * |||_______ bit5: ACFE : Compare enable
        * ||________ bit6: ADTRG: Conversion trigger select
        * |_________ bit7: ADACT: Convert active
        */

    #ifdef TENBIT_MODE
    ADCCFG |= 0x08;
    #endif
    #ifdef TWELVEBIT_MODE
    ADCCFG |= 0x04;
    #endif
```

```
      /*Change the channel, to check the releation between pins and channels */
      APCTL1 = 0xFF; //disable all ADC ports
      APCTL2 = 0x0F;
}
```

5.2.6 Start the ADC Conversion — ADC_Cvt

The function ADC_Cvt starts the ADC conversion on the specified channel and blocks till the conversion is completed if using poll mode instead of interrupt mode:

```
{
      #ifdef ADC_INT_EN
      ADCSC1 = (Channel & ADCSC1_ADCH_MASK) | ADCSC1_AIEN_MASK; //start the single conversion
      by software
      #else
      ADCSC1 = (Channel & ADCSC1_ADCH_MASK) ; //start the single conversion by software
      #endif
      #ifdef ADC_POLLING
      ADC_Poll();
      #endif
}
```

5.2.7 Initialize the TPM — joy_start_ms_timer

The function Joy_start_ms_timer initializes TPM to start a timer at 1 kHz if the delay parameter is 1 as below:

```
{
      /* Write TPM timer 1 counter module and channel 0 value register with (delay*5); */
      TPM1MOD = (delay*5);
      TPM1C0V = (delay*5);
      / * Clear TPM timer 1 counter register; */
      TPM1CNT = 0;
      /* Configure the TPM timer 1 channel 0 mode to output compare mode; */
      TPM1SC_CPWMS = 0;
      TPM1C0SC_MS0x = 1;
      TPM1C0SC_ELS0x = 1; /* toggle output */
      /* Enable the timer channel 0 interrupt; */
      TPM1C0SC_CH0IE = 1;
      /* Select Fixed clock as source clock and prescalor of 128; */
      TPM1SC_PS = 7; /* prescalor 128 */
      TPM1SC_CLKSA = 0;
      TPM1SC_CLKSB = 1; /* select Fixed clock as source clock */
}
```

5.2.8 Operate X,Y, and Throttle — TPMCHnEvent

The function TPMChnEvent is the TPM1 channel 0 interrupt service routine, which checks the X,Y, and Z outputs of the accelerometer and changes the report data accordingly:

```
{
      signed short delta;
      /* Clear interrupt flag */
      TPM1C0SC_CH0F = 0;
      /* Get X value with scaled result.
      * AdcResult of 0xFFF --> X of 127, and 0 --> X of -127 */
      ADC_Cvt(3);
      delta = x -(AdcResult >>4);
      if( ((delta) >= XY_THRESHOLD) ||
      ((delta) < -XY_THRESHOLD)
      )
      {
          delta += DIR_REP_X(hid_joy_report_in);
          if(delta < -127)
          {
              delta = -127;
          } else if(delta > 127)
          {
              delta = 127;
          }
          CHANGE_RPT_IN_X(hid_joy_report_in,delta);
          xy_state_changed = TRUE;
      }
      /* Get Y value with scaled result. */
      ADC_Cvt(0);
      delta = y -(AdcResult >>4);
      if( ((delta) >= XY_THRESHOLD) ||
      ((delta) < -XY_THRESHOLD)
      )
      {
          delta += DIR_REP_Y(hid_joy_report_in);
          if(delta < -127)
          {
              delta = -127;
          } else if(delta > 127)
          {
              delta = 127;
          }
          CHANGE_RPT_IN_Y(hid_joy_report_in,delta);
          xy_state_changed = TRUE;
       }

       /* Get throttle value with scaled result.
       * throttle is corresponding Z output of accerometer.
       */
       ADC_Cvt(1);
       delta = throttle -(AdcResult >>4);
       if( ((delta) >= THROTTLE_THRESHOLD) ||
       ((delta) < -THROTTLE_THRESHOLD)
       )
       {
          delta += DIR_REP_THROTTLE(hid_joy_report_in);
```

```
          if(delta < -127)
          {
              delta = -127;
          } else if(delta > 127)
          {
              delta = 127;
          }
           CHANGE_RPT_IN_THROTTLE(hid_joy_report_in,delta);
           xy_state_changed = TRUE;
       }
      /* Get hat value */
      ADC_Cvt(9);
      hat = (AdcResult>>10);
      if( hat != (GET_RPT_IN_HAT(hid_joy_report_in))) {
      CHANGE_RPT_IN_HAT(hid_joy_report_in,hat);
      hat_state_changed = TRUE;
}
```

5.2.9 Scan Buttons — joy_scan_matrix

The function Joy_scan_matrix scans the joystick buttons to see the states of buttons and changes the buttons field in hid_joy_report_in structure. If a button is pressed, the corresponding button field is set, and cleared otherwise. Its pseudo code is listed below:

```
{
      if (button 1 is pressed)
      {
          Set button 1 bit in hid_joy_report_in;
      } else
      {
          Clear button 1 bit in hid_joy_report_in;
      }
      if (button 2 is pressed)
      {
          Set button 2 bit in hid_joy_report_in;
      } else
      {
          Clear button 2 bit in hid_joy_report_in;
      }
      if (button 3 is pressed)
      {
          Set button 3 bit in hid_joy_report_in;
      } else
      {
          Clear button 3 bit in hid_joy_report_in;
      }
      if (button 4 is pressed)
      {
          Set button 4 bit in hid_joy_report_in;
      } else
      {
          Clear button 4 bit in hid_joy_report_in;
      }
      Return the buttons field of this report;
}
```

5.3 Create Hid Joystick Project with CodeWarrior

1. Start CodeWarrior. Click File menu and select “New Project…” item. The HC(S)08 New Project window pops up as shown in Figure 5. Select “MC9S08JM60” derivative under the HC08 JM Family item and “P&E Multilink/Cyclone Pro” connection as the default connection. Click Next.

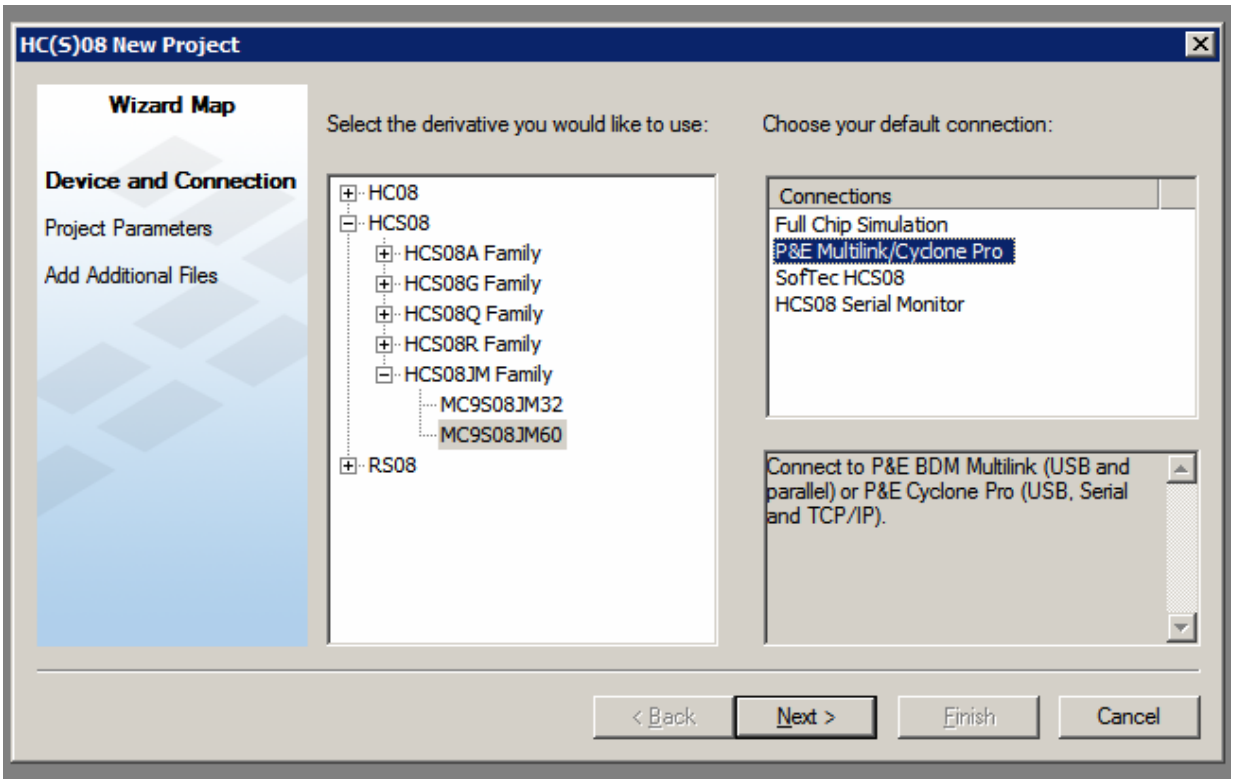

Figure 5. New Project Dialog

2. Choose the project name and location by clicking the “Set” button. Browse to the desired directory and type the project name. In Figure 5, it is Hid joystick. Now click the “Finish” button as shown in Figure 6. The HID joystick project has been created.

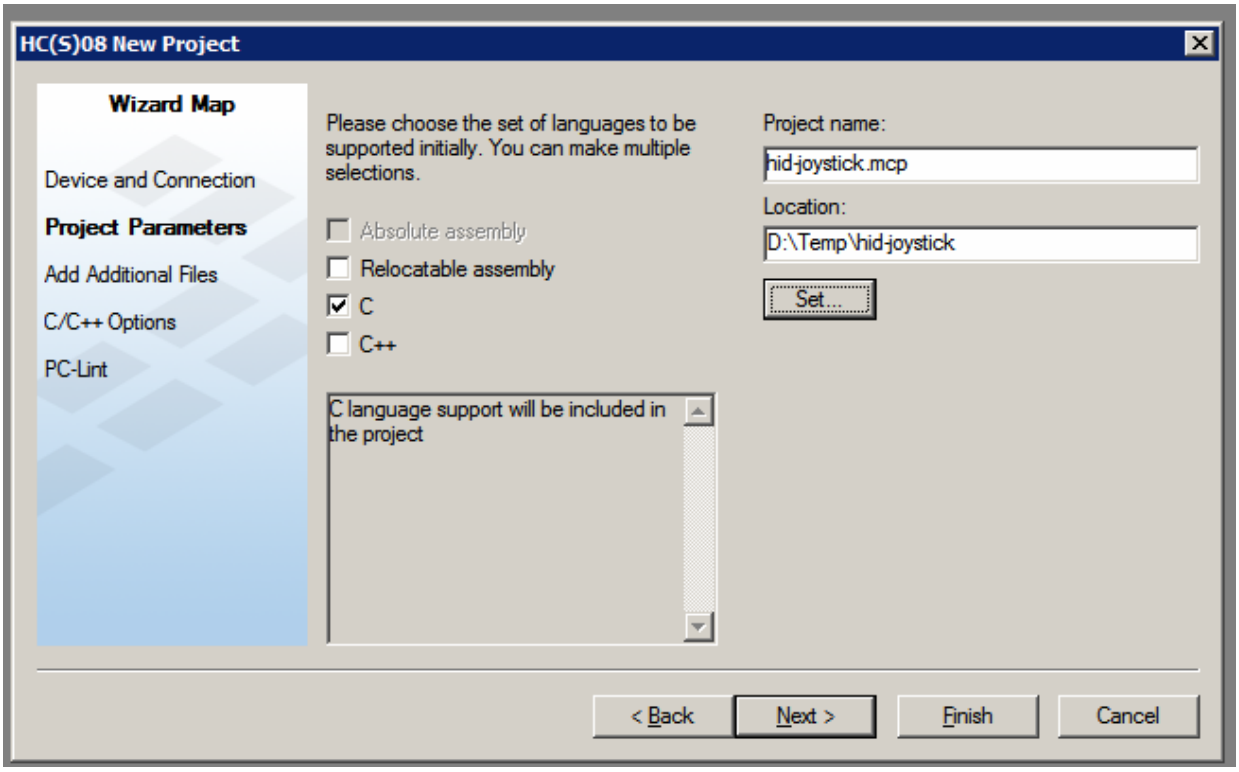

Figure 6. New Project

3. The next step is to copy CMX USB stack code and reorganize the file structure to suit your requirement. Close the project. Create a folder “CW project” in your project root directory. Move all files and folders created by CodeWarrior into this folder. Copy the USB DRV folder to the project directory from the CMX USB stack project directory. Create the HID DRV drv folder in the project directory and copy the following files from the CMX USB stack HID demo directory
   - Hid.c
   - hid.h
   - hid_usb_config.c
   - hid_usb_config.h

   Also create common, bsp and app folders as shown in Figure 7. Copy target.c and target.h from CMX USB usb common\hc9s08jmxx directory to bsp folder and hcc_types.h to common folder.

   Drag and drop bsp,app, usb drv, hid drv, common folder to the File tab of hid joystick.mcp project window and remove the Sources file group in the Files tab of the project window.

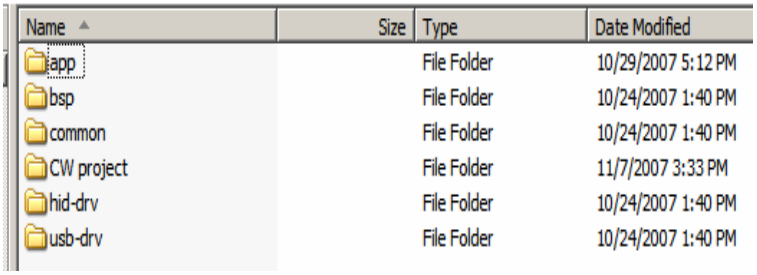

Figure 7. Project File Folder

4. Now build the project. It displays compiler errors: “Error: C5200: usb drv/usb.h file not found”. Click on this error message in the Errors & Warnings window to go to the related code line. Change the relative path of the usb.h header file as below:

```
#include "../usb-drv/usb.h"
```

   The compiler searches the user included header files from the current path and its relative path. The USB DRV folder is not its subfolder. It is located in the parent directory, it does not find usb.h without the “../” prefix that indicates the parent directory of the current path.

   Rebuild the project and some errors appear:

   “Error : C5200: hid_mouse.h file not found”, “Error : C5200: hid_kbd.h file not found”, “Error : C5200: hid_generic.h file not found”.

   Go to the related source code by clicking on these error messages. Remove the following lines:

```
#include "hid_mouse.h"
#include "hid_kbd.h"
#include "hid_generic.h"
And add the following line:
#include "hid_joy.h"
```

5. Create a hid_joy.h file in app folder, which contains the following lines:

```
#ifndef _USB_HID_H_
#define _USB_HID_H_
#ifndef FALSE
#define FALSE 0
#endif
#ifndef TRUE
#define TRUE  1
#endif
extern int hid_joy(void);

#endif
```

   Now modify the main function as aforementioned.

   Then copy hid_mouse.c from the CMX USB stack hid demo folder to app folder and change the name to hid_joy.c. Add it to app file group in the File tab of the hid joystick.mcp project window and edit this file. Also change hid_usb_config.c and hid_usb_config.h, and target.h accordingly.

   Figure 8 shows the final HID joystick project windows.

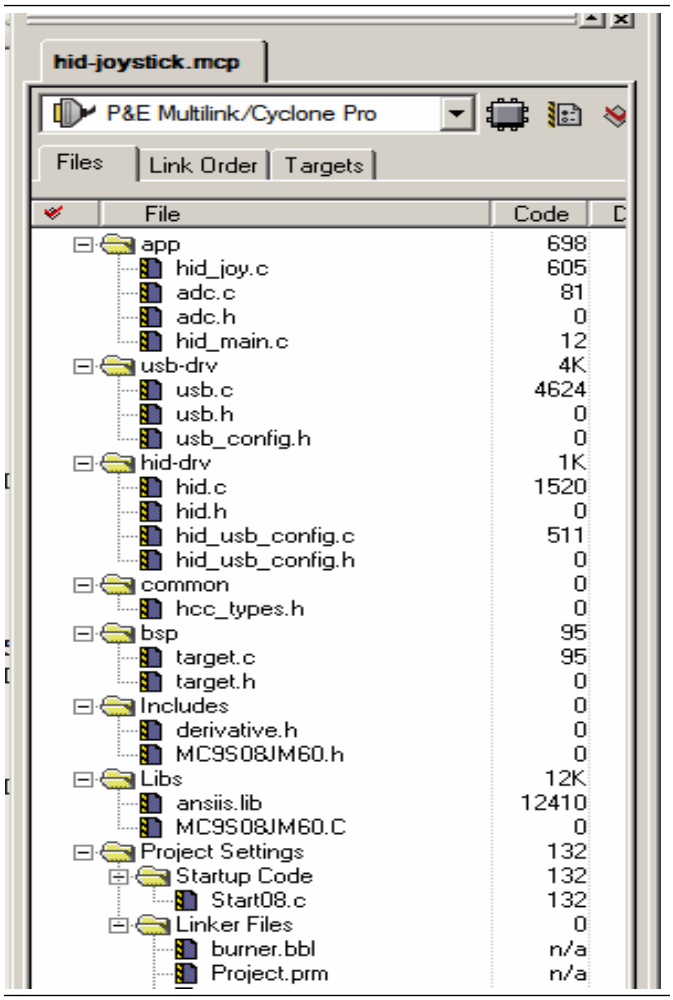

Figure 8. Hid joystick project window

6. Rebuild the project

5.4 Run the Demonstration

Demonstration program can be run on the DEMOJM REV C board.

Check the jumper settings as in Table 7.

Table 7. DEMOJM REV C Jumper Settings

Jumper Block	Label	Jumper
J3 (VDD_Select)	REG_VDD	Unconnected
	USB_VDD	Connected
	MiniUSB_VDD	Unconnected
	MCU_PORT_VDD	Unconnected
J4	TXD1	Connected
	RXD1	Connected
J5	CAN_PORT	Unconnected
J6	CAN_S	Pin 2 and 3 connected
J7	CTE	Connected
J8	CAN_EN	TXD, connected
		RXD, connected
J10	FAULT	Unconnected
J11	VHOST_EN	Pins 1 & 2 connected
J12	Pin 1 and 2 (DEVICE)	Connected
J13	DN_DOWN	Unconnected
J14	DP_DOWN	Unconnected
J15	USB_ID	Unconnected
J16	pullup	Unconnected
J17	LED_ENABLE	connected
J18	G-SEL1	Pins 2 & 3, connnected
J19	G-SEL2	Pins 2 & 3, connected
J20	SLEEP	Pins 2 & 3, connected
J21	Z/PTB1	Connected
	Y/PTB0	Connected
	X/PTB3	Connected
	X/PTD0	Unconnected
J27	KEY_ENABLE	Connected
J28	P&E INPUT_EN	Connected

Table 7. DEMOJM REV C Jumper Settings (continued)

J29	RESET_EN	Connected
J30	BUZ_EN	Connected
J31	IIC	Connected
J32	POT_EN	Connected

External power supply must not be plugged into P1.

Connect one USB cable to the board and the host PC. Connect another USB cable with one mini-AB connector to the mini AB plug on the board and the other end to the host PC.

Turn on the power by placing switch K6 to the on position. The host recognizes the new USB device and the request to install the driver. Follow the default settings on the driver installation dialog to the end of the installation.

Now load the code to the board by clicking on the debug icon in CodeWarrior and run it.

Click on windows start menu, select “Settings” and then “Control Panel” followed by clicking “Game Controller”. The Game Controllers window appears, where there is empty in the Installed game controllers list. The “USB HID Joystick demo for HC9S08JM devices” in the Installed game controller list Figure 9 appears.

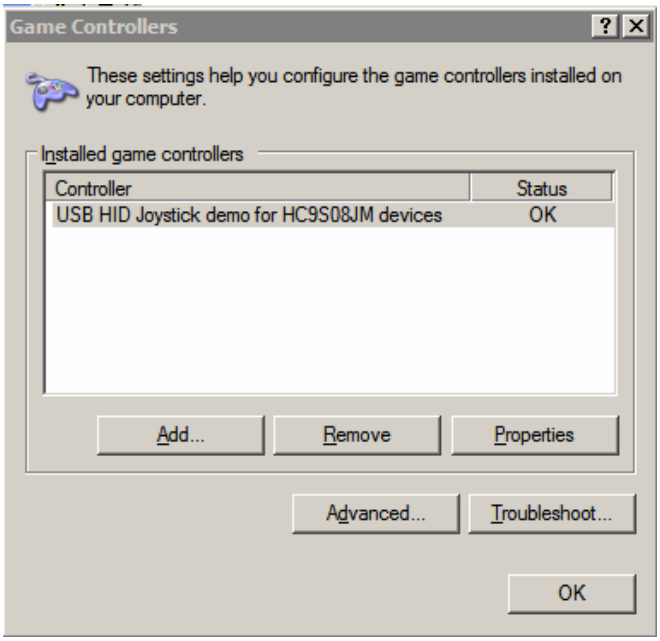

Figure 9. Game Controllers

Select this controller, and click on Properties button. The joystick properties dialog window appears as shown in Figure 10.

Press one of four buttons on the board (PTG0–PTG3), the related button icon on this properties window flashes.

Tilt the board to see axes and throttle movement. Tune W1 to see hat movement.

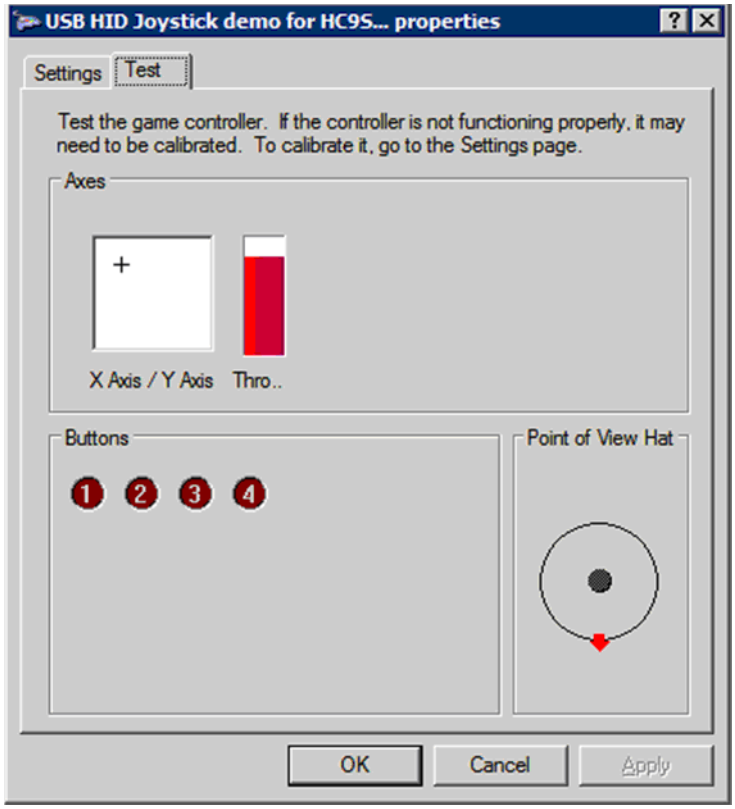

Figure 10. Joystick Properties

How to Reach Us:

Home Page:
www.freescale.com

Web Support:
http://www.freescale.com/support

USA/Europe or Locations Not Listed:
Freescale Semiconductor, Inc.
Technical Information Center, EL516
2100 East Elliot Road
Tempe, Arizona 85284
+1-800-521-6274 or +1-480-768-2130
www.freescale.com/support

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
www.freescale.com/support

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064
Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

Document Number: AN3565
Rev. 0
02/2008

freescale™
semiconductor