Embedding it better...

µTasker Document

- **USB User's Guide**

Table of Contents

1. Introduction

USB (Universal Serial Bus) has firmly established itself in the world of PC connectivity. Originally a lot of people were very sceptical about whether it really represented the solution which its promoters reported it to be. How could it compete with Firewire? Did it really have any advantages in any function other than a new method of connecting dongles without having to pile them on the parallel ports? When will it be ready to really be used if Windows NT isn't going to support it?

But history has shown that it does have a number of benefits which has enabled it to become the new connectivity standard. The version 2.0 has also increased operating speed capabilities to almost the Firewire performance level, at lower costs, and so now there are only a few areas where the interface cannot be used.

The µTasker project includes USB device support and a demo showing useful CDC (Communication Device Class) operation is described in the document http://www.utasker.com/docs/uTasker/uTaskerV1.3_USB_Demo.PDF. This allows any board with a USB device interface, supported by the µTasker project, to be connected to a PC and controlled from a standard terminal emulator (using virtual COM).

The purpose of this document is to give the µTasker user of the USB support a guide to configuring and using the interface. It doesn't attempt to describe the USB specification or low-level operational detail themselves since this is described in many books and documents on the internet, as well as in the official USB specifications. Instead, its goal is to allow the user to quickly set up and use the code base for real projects with as little knowledge of the actual operating details as possible.

To use USB does however require PC SW support as well as the embedded SW part. There are also various details which are specified in the USB standard and so must be understood in order to be able to adhere to them. The necessary ones are introduced in this text since they also form a basis for correct USB workings.

Finally note that the µTasker USB implementation is essentially hardware independent. The code can thus be easily transferred from one processor type to another without having to adapt any application level interfaces – all details are controlled in the hardware driver. In case of differences as dictated by the hardware realisation in the devices themselves separate notes are maintained in Appendix A.

2. USB Classes

Before actually configuring a µTasker project for USB it is necessary to know which USB class will be used. There are several well known classes, probably the most popular being HID (Human Interface Device) and CDC (Communication Device Class). The first is typically used for keyboards, mouse, joy-sticks, etc. and the second used to allow a PC to communicate via a virtual COM port.

Since USB devices usually communicate with a PC host (they can of course also communicate with an embedded controller host), the USB PC driver used for this communication will be dictated by the choice of the USB class.

The µTasker USB interface must be configured accordingly so that the class can be correctly initialised and any specified class commands correctly handled. For the remaining embedded USB use, the class actually used doesn't have a great influence on the internal operation, and the methods of sending and receiving data from within the µTasker framework remains of very generic nature.

3. Starting a USB Project

You may have a product which presently does not have a USB interface but instead uses the serial or parallel port of a PC to communicate. Due to reasons of compatibility (new PCs often don't have these ports built in any more and the competition already supports USB...) you want to introduce a new variation which allows direct USB operation. Alternatively, your new product needs to be able to benefit from a (fairly) high speed connection to a PC and be easy to attach (plug-and-play) and remove (possibly powered by the PC too) so USB seems the perfect solution (*although Ethernet should not be overlooked*).

You have chosen your processor with built-in USB controller and know exactly what it needs to do and all that remains is to write the software to control it. First we will briefly look at the PC software which may already exist, although not communicating via USB, or may also be completely new.

The PC software needs to work with a USB driver. This driver can be one of several variants:

1. A dedicated driver written specifically for the project
2. A generic driver as supplied with Windows (or your alternative operating system)
3. A class driver as supplied with Windows (or your alternative operating system)
4. A generic/class driver supplied by a third party with improved application interface and support

Unless you are experienced with driver development some of these alternatives are rather daunting and could represent a major challenge, so we will concentrate here on the simplest path which is to use something which already exists, even if it may not be the most efficient solution in terms of lean performance.

As an example, consider the case of adapting an existing solution using the COM port to work with direct USB (rather than an RS-232-USB adapter, which would be another possibility). For this case there are class drivers delivered with Windows which appear to the application as a virtual COM port, thus allowing existing PC applications to continue operating without any changes what-so-ever. This will be used as base for this document since it avoids any further complication at the PC side and thus allowing us to concentrate on getting something up and running.

4. Configurations, Interfaces and Endpoints

USB supports plug-and-play and so you can simply connect your new device to a free USB interface on your PC or a hub and that's it.... but for this to actually be able to take place you will have to ensure that the device correctly informs the PC about itself and the driver which it needs to be able to operate. This is where the strict USB specification comes in for the first time and we need to configure the software to so that it gives the correct information.

When the device is attached to the USB bus the PC host will recognise it due to pull-up / pull-down resistors and start the enumeration sequence. During the enumeration sequence it asks the device for certain details, after which the correct driver is loaded (and installed on first attachment). Only after the successful enumeration and configuration can the device start communicating on the USB bus. In the case of bus powered devices full power operation is only allowed after the successful sequence too.

You must know what configuration your device is to use so let's show how this is defined.

Before and during enumeration the device communicates using endpoint 0. This control endpoint always exists and is the only active one during this stage. Control endpoints can both send and receive data and use a three stage handshake to achieve very reliable communication. The host will request the configuration descriptor from the device and it is this configuration descriptor which contains all important information allowing the device to operate as you want it to do after the enumeration sequence. The µTasker USB support handles the data exchange but you must set up the project and supply the configuration details as follows.

1. First of all, ensure that the project is configured for the device that you are using. For example `#define _M5225X` in `config.h` activates this device family when using the Coldfire™ family.
2. Also ensure that USB support is enabled in `config.h`

   ```
   #define USB_INTERFACE          // enable USB driver interface
   ```

3. Define the (maximum) number of endpoints which need to be supported – in `config.h` (eg.):

   ```
   #define NUMBER_USB     (3 + 1)  // 4 physical queues
            (control plus 3 endpoints) needed for USB interface
   ```

4. Decide whether your project wants to use strings (strings will allow details like manufacturer and product name to be displayed by the PC) – in `config.h`

   ```
   #define USB_STRING_OPTION         // support optional string
                                                descriptors
   ```

5. Decide whether you would like to use variable strings (*rather than just fixed strings*), useful for serial numbers which cannot be hard coded due to variation between individual devices – in `config.h`

   ```
   #define USB_USER_DEFINABLE_STRINGS   // enable user to define
                                  USB string contents at run-time
   ```

6. Make your own version of `usb_application.c` and modify the struct

   `stUSB_CONFIGURATION_DESCRIPTOR_COLLECTION` as explained in the following section.

```
// We define the contents of the configuration descriptor used for our specific device and then set its
// contents
//
typedef struct _PACK stUSB_CONFIGURATION_DESCRIPTOR_COLLECTION
{
    USB_CONFIGURATION_DESCRIPTOR               config_desc;         // compulsory configuration descriptor

    USB_INTERFACE_DESCRIPTOR                   interface_desc_1;    // first interface descriptor
        USB_CDC_FUNCTIONAL_DESCRIPTOR_HEADER   CDC_func_header;     // CDC function descriptors due to
                                                                                              class used
        USB_CDC_FUNCTIONAL_DESCRIPTOR_CALL_MAN CDC_call_management;
        USB_CDC_FUNCTIONAL_DESCRIPTOR_ABSTRACT_CONTROL  CDC_abstract_control;
        USB_CDC_FUNCTIONAL_DESCRIPTOR_UNION    CDC_union;
    USB_ENDPOINT_DESCRIPTOR                    endpoint_3;          // end point of first interface

    USB_INTERFACE_DESCRIPTOR                   interface_desc_2;    // second interface descriptor
    USB_ENDPOINT_DESCRIPTOR                    endpoint_1;          // end points of second interface
    USB_ENDPOINT_DESCRIPTOR                    endpoint_2;
} USB_CONFIGURATION_DESCRIPTOR_COLLECTION;
```

Code 1 - USB_CONFIGURATION_DESCRIPTOR_COLLECTION struct

This is a local struct which is defined to suit a particular USB class and configuration. In this case it is specifically for a CDC device with three endpoints. The struct begins always with the compulsory USB_CONFIGURATION_DESCRIPTOR which is the first that will be requested by the USB host when enumeration begins.

Unfortunately the construction of the configuration descriptor may involve some detailed understanding of certain interface classes as shown by the CDC (communication class descriptor) interface in the example. Here it is best to take a little time reading the class descriptions in the USB specification and hoping that it falls in to place.

What this descriptor is specifying is that the configuration consists of 2 interfaces. The first is a CDC class interface (it can use a CDC class driver to control typical communication functions) with one end point (endpoint number 3). The second interface has no class properties and two endpoints (endpoints 1 and 2), which will be used for the transfer of data between the host and the device.

The total number of endpoints is 3 (*in addition to the standard control endpoint 0*) as set by the define:

```
#define NUMBER_OF_ENDPOINTS      3    // uses 3 endpoints (2 IN and 1 OUT) in addition to the
                                                                  default control endpoint 0
```

Additional interfaces and endpoints can be added by simply extending `USB_CONFIGURATION_DESCRIPTOR_COLLECTION` with more entries, where a new interface is always followed by optional class descriptors and its endpoint descriptors.

This can be displayed graphically (*as is often done in USB books*) as follows:

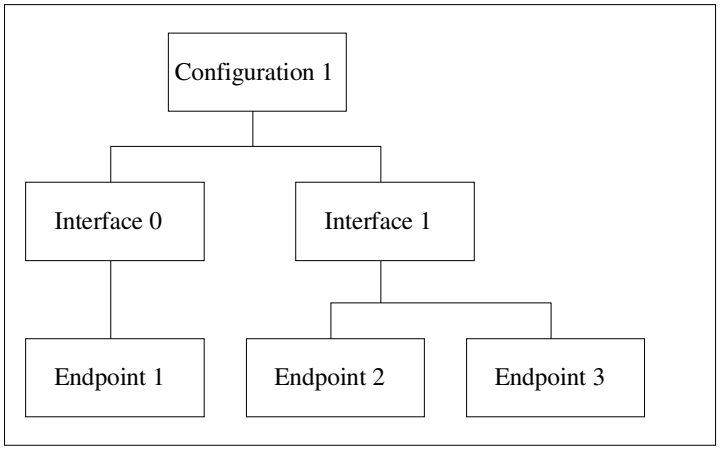

Figure 1. Configuration hierarchy corresponding to the example

It is also possible to have multiple configurations as well as alternative interfaces, but we won't discuss this here in order to keep things as simple as possible.

It should be possible to see that our `struct` is a representation of this configuration model.

The host will activate a configuration to terminate the enumeration sequence, which causes all of its standard interfaces to be activated (in this case there will be 3 endpoints available in addition to the default control endpoint 0).

On a side note (*not supported in our test configuration*), the host could decide to change configurations or switch to an alternative interface if these are available – an example of where a configuration change can be useful is if the USB bus bandwidth becomes restricted by multiple devices on the bus and a different configuration specifies operation with reduced bandwidth requirements. An example of the use of alternative interfaces is when a device has an active mode and an idle mode. In the active mode its endpoint may be configured to transfer a lot of data so the host will reserve greater bandwidth for it on the USB bus. When it goes to an idle mode it will not need to have this bandwidth reserved and so the host is allowed to switch interfaces and thus no longer need to account for the bandwidth. Once the device is activated again the default interface is selected again.

What is not yet visible are various details about the configuration, interfaces and the endpoints. The `struct` has only defined the hierarchy and now we must fill in its details. For this use, a const table conforming to our `struct` filled out with all that is required by the host to configure the later operation.

```
static const USB_CONFIGURATION_DESCRIPTOR_COLLECTION config_descriptor = {
    {                                                   // config descriptor
    DESCRIPTOR_TYPE_CONFIGURATION_LENGTH,               // length (0x09)
    DESCRIPTOR_TYPE_CONFIGURATION,                      // 0x02
    {LITTLE_SHORT_WORD_BYTES(sizeof(USB_CONFIGURATION_DESCRIPTOR_COLLECTION))},
                                                        // total length (little-endian)
    0x02,                                               // configuration number
    0x01,                                               // configuration value
    #ifdef USB_STRING_OPTION
    CONFIGURATION_STRING_INDEX,                         // string index to configuration
    #else
    0,                                                  // zero when strings are not supported
    #endif
    (SELF_POWERED | ATTRIBUTE_DEFAULT),                 // attributes for configuration,
    0                                                   // consumption in 2mA steps (eg. 100/2 for 100mA)
    },                                                  // end of compulsory config descriptor

    {                                                   // interface descriptor
    DESCRIPTOR_TYPE_INTERFACE_LENGTH,                   // length (0x09)
    DESCRIPTOR_TYPE_INTERFACE,                          // 0x04
    0,                                                  // interface number 0
    0,                                                  // alternative setting 0
    1,                                                  // number of endpoints in addition to EP0
    USB_CLASS_COMMUNICATION_CONTROL,                    // interface class (0x02)
    USB_ABSTRACT_LINE_CONTROL_MODEL,                    // interface sub-class (0x02)
    0,                                                  // interface protocol
    #ifdef USB_STRING_OPTION
    INTERFACE_STRING_INDEX,                             // string index for interface
    #else
    0,                                                  // zero when strings are not supported
    #endif
    },                                                  // end of interface descriptor

    {                                                   // function descriptors
    USB_CDC_FUNCTIONAL_DESCRIPTOR_HEADER_LENGTH,        // descriptor size in bytes (0x05)
    CS_INTERFACE,                                       // type field (0x24)
    HEADER_FUNCTION_DESCRIPTOR,                         // header descriptor (0x00)
    {LITTLE_SHORT_WORD_BYTES(USB_SPEC_VERSION_1_1)}     // specification version
    },
    {
    USB_CDC_FUNCTIONAL_DESCRIPTOR_CALL_MAN_LENGTH,      // descriptor size in bytes (0x05)
    CS_INTERFACE,                                       // type field (0x24)
    CALL_MAN_FUNCTIONAL_DESCRIPTOR,                     // call management function descriptor (0x01)
    1,                                                  // capabilities
    0                                                   // data interface
    },                                                  // end of function descriptors
    {
    USB_CDC_FUNCTIONAL_DESCRIPTOR_ABSTRACT_CONTROL_LENGTH,  // descriptor size in bytes (0x04)
    CS_INTERFACE,                                       // type field (0x24)
    ABSTRACT_CONTROL_FUNCTION_DESCRIPTOR,               // abstract control descriptor (0x02)
    2                                                   // capabilities
    },
    {
    USB_CDC_FUNCTIONAL_DESCRIPTOR_UNION_LENGTH,         // descriptor size in bytes (0x05)
    CS_INTERFACE,                                       // type field (0x24)
    UNION_FUNCTIONAL_DESCRIPTOR,                        // union function descriptor (0x06)
    0,                                                  // control interface
    1                                                   // subordinate interface
    },                                                  // end of function descriptors

    {                                                 // interrupt endpoint descriptor for first interface
    DESCRIPTOR_TYPE_ENDPOINT_LENGTH,                    // descriptor size in bytes (0x07)
    DESCRIPTOR_TYPE_ENDPOINT,                           // end point descriptor (0x05)
    (IN_ENDPOINT | 0x03),                               // direction and address of endpoint
    ENDPOINT_INTERRUPT,                                 // endpoint attributes
    {LITTLE_SHORT_WORD_BYTES(64)},                      // endpoint FIFO size (little-endian)
    10                                                  // polling interval in ms
    },                                                  // end of endpoint descriptor

    {                                                   // the second interface
    DESCRIPTOR_TYPE_INTERFACE_LENGTH,                   // descriptor size in bytes (0x09)
    DESCRIPTOR_TYPE_INTERFACE,                          // interface descriptor (0x04)
    1,                                                  // interface number
    0,                                                  // no alternative setting
    2,                                                  // 2 end points
    INTERFACE_CLASS_COMMUNICATION_DATA,                 //
    0,                                                  // sub-class
    0,                                                  // interface protocol
    #ifdef USB_STRING_OPTION
    INTERFACE_STRING_INDEX,                             // string index for interface
    #else
    0                                                   // zero when strings are not supported
    #endif
    },
    {                                             // bulk out endpoint descriptor for the second interface
```

```
    DESCRIPTOR_TYPE_ENDPOINT_LENGTH,                    // descriptor size in bytes (0x07)
    DESCRIPTOR_TYPE_ENDPOINT,                           // end point descriptor (0x05)
    (OUT_ENDPOINT | 0x01),                              // direction and address of end point
    ENDPOINT_BULK,                                      // endpoint attributes
    {LITTLE_SHORT_WORD_BYTES(64)},                      // endpoint FIFO size (little-endian - 64 bytes)
    0                                                   // polling interval in ms - ignored for bulk
    },
    {                                                 // bulk in endpoint descriptor for the second interface
    DESCRIPTOR_TYPE_ENDPOINT_LENGTH,                    // descriptor size in bytes (0x07)
    DESCRIPTOR_TYPE_ENDPOINT,                           // end point descriptor (0x05)
    (IN_ENDPOINT | 0x02),                               // direction and address of end point
    ENDPOINT_BULK,                                      // endpoint attributes
    {LITTLE_SHORT_WORD_BYTES(64)},                      // endpoint FIFO size (little-endian - 64 bytes)
    0                                                   // polling interval in ms - ignored for bulk
    }
};
```

Code 2 - USB_CONFIGURATION_DESCRIPTOR_COLLECTION content

Each descriptor contains information as defined by the USB specification. The endpoints are fairly obvious since they detail the endpoint number and characteristics like whether it is an IN (transfers data to host) or OUT (receives data from the host) in interrupt, bulk or isochronous mode.

Everything is in *little-endian* format so there are some macros used (like `LITTLE_SHORT_WORD_BYTES()`) to ensure that this remains hardware independent. Standard defines are contained in `usb.h` whilst project specific ones are declared in `config.h`.

If you add extra interfaces and/or endpoints, there must also be corresponding entries to describe them.

The next step is to configure the device descriptor:

```
static const USB_DEVICE_DESCRIPTOR device_descriptor = {       // constant device descriptor
    STANDARD_DEVICE_DESCRIPTOR_LENGTH,                         // standard device descriptor length (0x12)
    DESCRIPTOR_TYPE_DEVICE,                                    // 0x01
    {LITTLE_SHORT_WORD_BYTES(USB_SPEC_VERSION_1_1)},           // USB1.1 or USB2
    DEVICE_CLASS_COMMUNICATION_AND_CONTROL,   // device class, sub-class and protocol (communication class)
    ENDPOINT_0_SIZE,                                           // size of endpoint reception buffer
    {LITTLE_SHORT_WORD_BYTES(USB_VENDOR_ID)},                  // our vendor ID
    {LITTLE_SHORT_WORD_BYTES(USB_PRODUCT_ID)},                 // our product ID
    {LITTLE_SHORT_WORD_BYTES(USB_PRODUCT_RELEASE_NUMBER)},     // product release number
#ifdef USB_STRING_OPTION                                       // if we support strings add the data here
    MANUFACTURER_STRING_INDEX, PRODUCT_STRING_INDEX, SERIAL_NUMBER_STRING_INDEX,
                                                               // fixed string table indexes
#else
    0,0,0,                                                     // used when no strings are supported
#endif
    NUMBER_OF_POSSIBLE_CONFIGURATIONS                          // number of configurations possible
};
```

Code 3 - USB_DEVICE_DESCRIPTOR content

This is also in `usb_application.c` and is rather easier to set up.

It is in fact the first descriptor requested by a USB host and contains details about which USB version is being respected. Basically there is no big difference between 1.1 and 2.0 when high speed mode is not supported by the hardware and USB1.1 may as well be used. In this example the class is defined already in the device descriptor, although often this is set with `DEVICE_CLASS_AT_INTERFACE`, where the details follow at the interface descriptor rather than here.

There are two defines contained which need to be known globally (also by the USB driver) so are defined in `config.h`:

```
#define NUMBER_OF_POSSIBLE_CONFIGURATIONS  1   // one USB configuration
#define ENDPOINT_0_SIZE                    8   // maximum packet size for endpoint 0.
                                               Low speed devices must use 8 whereas full
                                               speed devices can chose to use 8, 16, 32 or 64
```

Two entries in the device descriptor are critical to the correct operation of the USB interface with the PC. These are `USB_VENDOR_ID` and `USB_PRODUCT_ID`. The vendor ID is the one which you pay for... and the product ID specifies one of many USB based products that you may create. For first tests we will of course not need to purchase a USB vendor ID from www.usb.org but can use the one allocated for the µTasker project by the corresponding semi-conductor manufacturer. More details about how these work together with the particular windows USB driver, the `.inf` file that you supply for installation, certified drivers and the use of the USB logo will be given later.

4.1. PID/VID

As seen in the previous section, the device descriptor contains the Vendor ID (VID) and the Product ID (PID). This has the function of allowing the host to determine which driver is to be installed on first connection and later which installed driver is to be loaded on each subsequent connection.

A USB device must have a PID/VID which has been defined for its use. A VID can be purchased from http://www.usb.org and the options available are the following (quote from usb.og):

- *Option 1: Join the USB-IF*
  If your company chooses to become a member of the USB-IF, the annual fee for membership is US$4000. A few of the benefits of membership are: only members are eligible to participate in free USB-IF sponsored quarterly Compliance Workshops, participate in USB Device Working Groups, a waived logo administration fee when joining the new USB-IF logo program and have their company and product information included on the usb.org web site.
- *Option 2: Become a non-member USB-IF Logo Licensee*
  If your company executes the USB-IF Trademark License Agreement in conjunction with the assignment of your company's vendor ID number, the fee is US$2000 (your company must execute and return the USB-IF Trademark License Agreement along with a vendor ID number application to the address below.) Please keep in mind that becoming a USB-IF Logo Licensee alone does not entitle your company to USB-IF membership benefits, however this option allows your company to be assigned a USB Vendor ID Number and enjoy the benefit of a waived USB-If Logo License Fee.
- *Option 3: Purchase a Vendor ID Number without signing the USB-IF Logo License Agreement*
  The fee is a one time administrative fee of US$2000 for the Vendor ID Number alone. If your company chooses to execute the USB-IF Trademark License Agreement at a later time, the USB-IF Logo License Fee of US$2000 will apply.

For a first USB product, the fact that a VID must be purchased can often pose problems due to the costs involved and also the administrative tasks of understanding the details and getting such a purchase authorized by project supervisors, who possibly didn't realize what was involved.

However, in some circumstanced, it is possible to obtain permission to use the vendor ID of the silicon manufacturer and obtain a Product ID for little or no cost. In some cases this is restricted to a certain maximum quantity of devices but it allows a simplified entry route to a first USB based product.

Note that to be able to use the USB logo on a product it will still be necessary to purchase a Vendor ID and also submit the product for approval tests. Whether the USB logo is necessary for a product is a decision left to the manufacturer and depends on the target market and other factors.

The µTasker project has been registered with several semiconductor manufacturers and users of the µTasker project may use the VID/PID as allocated for the particular target officially for all test, research and development work. Once a project has reached production phase it is possibly to apply for a unique PID under the semiconductor manufacturer's VID. Details about the application procedure can be obtained on request by sending an email to one of the contact addresses on the contact address on the µTasker web site.

The allocated VID/PIDs are listed in Appendix A.

4.2. String Support

String support is the only optional part of the descriptors. If you don't want to use any strings, simply remove the define (from config.h)

```
#define USB_STRING_OPTION                                // support optional string descriptors
```

When strings are used, the string table is also defined in `usb_application.c`

```
#ifdef USB_STRING_OPTION                                 // if our project supports strings
    #define MANUFACTURER_STRING_INDEX       1            // index must match with order in the string list
    #define PRODUCT_STRING_INDEX            2
    #define SERIAL_NUMBER_STRING_INDEX      3
    #define CONFIGURATION_STRING_INDEX      4
    #define INTERFACE_STRING_INDEX          5

    #define UNICODE_LANGUAGE_INDEX          UNICODE_ENGLISH_LANGUAGE // English language used by strings
    #define LAST_STRING_INDEX               INTERFACE_STRING_INDEX   // last string entry - used for
                                                                        protection against invalid string
                                                                        index requests
#endif
```

Code 4 – String Indexes as defined for the project

Each string has a reference. 0 is always used as a language string and so the definable ones start from an index 1, and the value `LAST_STRING_INDEX` closes the list.

The string contents are then constructed as follow:

```
#ifdef USB_STRING_OPTION    // if our project supports strings
// The characters in the string must be entered as 16 bit unicode in little-endian order!!
// The first entry is the length of the content (including the length and descriptor type string entries)
static const unsigned char usb_language_string[] = {4,  DESCRIPTOR_TYPE_STRING,
                                                        LITTLE_SHORT_WORD_BYTES(UNICODE_LANGUAGE_INDEX)};
                                                        // this is compulsory first string

static const unsigned char manufacturer_str[] = {10, DESCRIPTOR_TYPE_STRING, 'M',0, 'a',0, 'n',0, 'u',0};

static const unsigned char product_str[] = {16, DESCRIPTOR_TYPE_STRING, 'M',0, 'y',0, ' ',0,
                                                                        'P',0, 'r',0, 'o',0, 'd',0};
    #if defined USB_RUN_TIME_DEFINABLE_STRINGS
static const unsigned char serial_number_str[] = {0};   // the application delivers this string
                                                           (generated at run time)
    #else
static const unsigned char serial_number_str[] = {10, DESCRIPTOR_TYPE_STRING, '0',0, '0',0, '0',0, '1',0};
    #endif

static const unsigned char config_str[]      = {10, DESCRIPTOR_TYPE_STRING, 'C',0, 'o',0, 'n',0, 'f',0};

static const unsigned char interface_str[]   = {8,  DESCRIPTOR_TYPE_STRING, 'I',0, 'n',0, 't',0};



static const unsigned char *ucStringTable[]  = {usb_language_string, manufacturer_str, product_str,
serial_number_str, config_str, interface_str};
#endif
```

Code 5 – String content using Unicode

Here the English language is defined. All strings must use *Unicode*, where the English language simply needs each character to be followed by a 0 (*others languages need a bit more effort*...).

All descriptors and strings are const types and so will be fixed in FLASH in the software and not occupy and RAM space. However there are some cases where a dynamic content string is of use. A typical example would be a serial number which will change with every device.

This is supported as show above when the following option (also in `config.h`) is enabled:

```
#define USB_RUN_TIME_DEFINABLE_STRINGS  // enable USB string content to be
                                           defined at run time (variable)
```

The application can in this case generate a string in RAM with the required content, which can then be requested by an application on the PC via a standard string request. Some of these strings are visible when a device is installed and so their use does improve the impression that the solution gives the user.

Admittedly, the set up of these tables could cause some problems initially and it is recommended to consult a good USB book which describes such details in a practical way. But once you have managed to set them up correctly the USB code in the uTasker project will do the rest for you so that you can later communicate in a simple and flexible manor without needing to know much more about the subsequent USB operating details.

As a reminder, all configuration tables are declared as `static const` types. This means that they are private to the USB application and also reside in FLASH memory on the target, thus not consuming and RAM.

5. Opening the USB interface for Application use

Up to now we have only defined how the USB interface should be configured by declaring a `const struct` in memory, but have not actually activated the interface for use. To do this the standard µTasker interface routines are used, whereas the endpoint (*as well as the default control endpoint 0*) receive their own communication handles.

The following shows the configuration for the example configuration, which uses 3 endpoints in addition to the default control endpoint 0:

```
// The USB interface is configured by opening the USB interface once for the default control endpoint 0,
// followed by an open of each endpoint to be used (each endpoint has its own handle).
// Each endpoint can use an optional callback or can define a task to be woken on OUT frames.
// Transmission can use direct memory method or else an output buffer (size defined by open),
// and receptions can use an optional input buffer (size defined by open).
//
static void fnConfigureUSB(void)
{
    USBTABLE tInterfaceParameters;                          // table for passing information to driver

    tInterfaceParameters.Endpoint = 0;               // set USB default control endpoint for configuration
    tInterfaceParameters.usConfig = USB_FULL_SPEED;         // full-speed, rather than low-speed
    tInterfaceParameters.usb_callback = control_callback;   // callback for control endpoint to enable
                                                                  class exchanges to be handled
    tInterfaceParameters.queue_sizes.TxQueueSize = 0;       // no tx buffering
    tInterfaceParameters.queue_sizes.RxQueueSize = 0;       // no rx buffering
    tInterfaceParameters.ucClockSource = EXTERNAL_USB_CLOCK; // use 48MHz crystal directly as USB clock
                                                                  (recommended for lowest jitter)
    tInterfaceParameters.ucEndPoints = NUMBER_OF_ENDPOINTS;  // number of endpoints, in addition to EP0
    tInterfaceParameters.owner_task = OWN_TASK;             // local task receives USB state change events
    USB_control = fnOpen( TYPE_USB, 0, &tInterfaceParameters);  // open the default control endpoint with
                                                                  defined configurations (reserves
                                                                  resources but only control is active)
    tInterfaceParameters.Endpoint = 2;                      // set USB endpoints to act as an input/output
                                                                                pair - transmitter (IN)
    tInterfaceParameters.Paired_RxEndpoint = 1;             // receiver (OUT)
    tInterfaceParameters.usEndpointSize = 8;                // endpoint queue size (2 buffers of this size
                                                                        will be created for reception)
    tInterfaceParameters.usb_callback = 0;   // no callback since we use rx buffer - the same task is owner
    tInterfaceParameters.queue_sizes.RxQueueSize = 256;     // optional input queue (used only when no
                                                                                    callback defined)
    tInterfaceParameters.queue_sizes.TxQueueSize = 1024;    // additional tx buffer
    #ifdef WAKE_BLOCKED_USB_TX
    tInterfaceParameters.low_water_level = (tInterfaceParameters.queue_sizes.TxQueueSize/2);
                                                            // TX_FREE event on half buffer empty
    #endif
    USBPortID_comms = fnOpen( TYPE_USB, 0, &tInterfaceParameters); // open the endpoints with defined
                                                                  configurations (initially inactive)

    tInterfaceParameters.Endpoint = 3;                      // set USB channel - endpoint
    tInterfaceParameters.Paired_RxEndpoint = 0;             // no pairing
    tInterfaceParameters.owner_task = 0;                    // don't wake task on reception
    tInterfaceParameters.usb_callback = 0;                  // no call back function
    tInterfaceParameters.usEndpointSize = 64;               // endpoint queue size (2 buffers of this size
                                                                        will be created for reception)
    tInterfaceParameters.queue_sizes.TxQueueSize = 0;       // no buffering
    USBPortID_interrupt_3 = fnOpen( TYPE_USB, 0, &tInterfaceParameters); // open the endpoint with defined
                                                                  configurations (initially inactive)
}
```

Code 6 – Opening the USB interface

Note that two endpoints can be grouped together to form a virtual bi-directional channel (1 and 2 in this example).

This example shows the endpoints used with different buffering and call back strategies (*which will be explained later*) but the important points are that the default endpoint 0 is first opened, which configures the basic hardware, followed by the additional endpoints, which are given their own endpoint sizes (these MUST correspond with the maximum endpoint

sizes as defined in the endpoint descriptors) and various other buffer and call back characteristics.

When enumeration is successful the USB task (defined as owner task of endpoint 0) will receive an event `E_USB_ACTIVATE_CONFIGURATION` with the configuration number so that the application is aware that the USB interface is now active. It can subsequently send data over IN endpoints using the standard write call (eg.):

```
    fnWrite(USBPortID_comms, ptrData, Length);              // write Length bytes of data from the
                                                               pointer ptrData to USB interface
```

Code 7 – Writing to a buffered USB interface

OUT endpoints can read buffered data using the standard read routines (eg.):

```
while (fnMsgs(USBPortID_comms) != 0) {                          // reception from endpoint 1
    Length = fnRead(USBPortID_comms, ucInputMessage, MEDIUM_MESSAGE); // read the content
    fnWrite(SerialPortID, ucInputMessage, Length);              // received data is sent to serial output
}
```

Code 8 – Reading data from a buffered USB interface (and subsequent transmission to a UART)

The previous example shows how easy it is to transfer received USB data from an OUT endpoint to the serial port. Both buffered USB and UART use the same interface routine with their corresponding interface handles.

To do the opposite (*send received serial data to the USB interface*) it would look like this:

```
while (fnMsgs(SerialPortID) != 0) {                             // reception from serial interface
    Length = fnRead(SerialPortID, ucInputMessage, MEDIUM_MESSAGE);  // read the content
    fnWrite(USBPortID_comms, ucInputMessage, Length);           // received data is sent to USB endpoint 2
}
```

Code 9 – Sending data to a buffered USB interface (after reception from a UART)

In fact the use of the USB interface was intentionally made to be very similar to the use of a normal serial interface, where the user simply has to specify the USB handle rather than the serial handle and the USB driver does the rest. Note also that the grouping of an IN and an OUT endpoint to a virtual channel allows this to be used bi-directionally (a read is from the OUT endpoint and a write is to the corresponding IN endpoint). A further advantage is that the µTasker debug interface `DebugHandle` can be set with the USB handle value to create a compatible bidirectional debug interface over a USB connection by simply setting `DebugHandle = USBPortID_comms`. To revert back to a UART based debug interface is just as simple using `DebugHandle = SerialPortID`.

The USB task receives, in addition to the E_USB_ACTIVATE_CONFIGURATION event, the following *interrupt* events on USB bus state changes:

EVENT_USB_RESET	– a reset occurred when in the configured state
EVENT_USB_SUSPEND	– the device has been suspended. If bus powered the HW should reduce power!
EVENT_USB_RESUME	– the USB device has resumed operation and the HW of a bus powered device can use full power again

6. Test Driving the Demo USB Communications Project

For full details about the USB demo application and running this on the target hardware and also in the µTasker simulator see the document
http://www.utasker.com/docs/uTasker/uTaskerV1.3_USB_Demo.PDF

This document explains how the USB interface is used as a command line interface, a USB <-> RS232 converter and also for uploading new firmware.

7. USB Demo Project Code Examples

This section gives more details about how the demo project USB application code interacts with the USB driver to achieve efficient code and also efficient USB transfers.

7.1. Receiving USB Messages and Events

The task which will receive USB events is entered when the USB interface endpoint 0 is opened. This is passed in the parameter list as

```
tInterfaceParameters.owner_task = OWN_TASK;
```

meaning that the local task is the destination task to which all USB events are sent.

The USB task (a fictional task which has no body in the project but registers itself as source when sending messages) informs the USB interface owner task when a configuration is activated. The demo code displays the fact that enumeration has completed and also the number of the activated configuration as shows in the following code excerpt:

```
        case TASK_USB:                                                  // USB interrupt handler is
requesting us to perform work offline
            fnRead( PortIDInternal, &ucInputMessage[MSG_CONTENT_COMMAND],
ucInputMessage[MSG_CONTENT_LENGTH] ); // get the content
            switch (ucInputMessage[MSG_CONTENT_COMMAND]) {
            case E_USB_ACTIVATE_CONFIGURATION:
                fnDebugMsg("Enumerated (");                             // the interface has been
activated and enumeration completed
                fnDebugDec(ucInputMessage[MSG_CONTENT_COMMAND+1], 1, 0); // the configuration
                fnDebugMsg(")\n\r");
                break;
            }
            break;
```

Code 10 – Handling E_USB_ACTIVATION message in the USB owner task

Also USB events (without data content) are sent to the USB owner task. On reception of these events the USB application knows the state of the USB bus and sends the present states to the debug output. When the USB interface is connected it can be used for debug output and so when the EVENT_USB_RESET or EVENT_USB_SUSPEND is received it also sets the debug output back to the UART (assumed to be the default debug interface).

The EVENT_USB_SUSPEND can also be used by USB powered circuits to configure low power modes in order to respect the maximum consumption in the suspended state. The EVENT_USB_RESUME can alternatively be used to return to full power mode when teh suspend state is removed.

```
case INTERRUPT_EVENT:                                  // interrupt event without data
    switch (ucInputMessage[MSG_INTERRUPT_EVENT]) {     // specific interrupt event type
    case EVENT_USB_RESET:                              // active USB connection has been reset
        if (usUSB_state != ES_NO_CONNECTION) {         // if the USB connection was being used for
                                                          debug menu, restore previous interface
            DebugHandle = SerialPortID;
            fnGotoNextState(ES_NO_CONNECTION);
            usUSB_state = ES_NO_CONNECTION;
        }
        fnDebugMsg("USB Reset\n\r");                   // display that the USB bus has been reset
        break;

    case EVENT_USB_SUSPEND:          // a suspend condition has been detected. A bus powered device
                                        should reduce consumption to <= 500uA or <= 2.5mA (high
                                        power device)
        if (usUSB_state != ES_NO_CONNECTION) {         // if the USB connection was being used for
                                                          debug menu, restore previous interface
            DebugHandle = SerialPortID;
            fnGotoNextState(ES_NO_CONNECTION);
            usUSB_state = ES_NO_CONNECTION;
        }
        fnSetUSBConfigState(USB_DEVICE_SUSPEND, 0);    // set all endpoint states to suspended
        fnDebugMsg("USB Suspended\n\r");
        break;

    case EVENT_USB_RESUME:                             // a resume sequence has been detected so
                                                          full power consumption can be resumed
        fnSetUSBConfigState(USB_DEVICE_RESUME, 0);     // remove suspended state from all endpoints
        fnDebugMsg("USB Resume\n\r");
        break;
    }
    break;
```

Code 11 – Handling standard USB events in the USB owner task

7.2. Receiving Data from a Buffered USB Interface

Buffered USB interfaces can be accessed in the same manner as serial UART interfaces. See Code 8 for an example of receiving from a USB interface and sending the received data to a UART output.

Each OUT endpoint, or endpoint communication pair, can be assigned to an owner task (as endpoint 0 is). In the demo USB application the same task is used to receive data from endpoint 1 as is used to receive endpoint 0 messages or events. When a USB OUT frame has been received by the USB driver it will wake the owner task so that it can read any waiting USB messages. The messages are put to the USB buffer (the length of this buffer is defined when the USB endpoint interface is configured) and received as a stream of data, rather than individual OUT tokens. If the USB application doesn't read this data, and the input buffer becomes full, the driver will no longer service the OUT tokens, causing the host to stop further transmission. Some host drivers will have a timeout as to how long they allow this blocked state to last but others will tolerate an infinite blocked state; when the USB

application reads data from the input queue, using the `fnRead()` call the USB driver will be able to accept further OUT tokens and the transfer of data from the USB host can continue.

The following example shows the USB reception being blocked by the USB application if the UART, which it wants to transfer the data to, doesn't have enough space in its buffer to accept it. This will happen often when data from a fast USB connection is transferred to a slower UART connection. It may also occur when the flow control on the UART stops transmission for a certain amount of time.

```
if (fnWrite(SerialPortID, 0, MEDIUM_MESSAGE) != 0) { // check that there is space for a block of data
    Length = fnRead(USBPortID_comms, ucInputMessage, MEDIUM_MESSAGE); // read the content
    fnWrite(SerialPortID, ucInputMessage, Length);                    // send input to serial port
}
else {
    fnDriver(SerialPortID, MODIFY_WAKEUP, (MODIFY_TX | OWN_TASK));    // we want to be woken when
                                                                          the queue is free again
    break;                                        // leave present USB data in the input buffer until we
                                                                        have enough serial output buffer space
                                                  // the TX_FREE event is not explicitly handled since
                                                     it is used to wake a next check of the buffer progress
}
```

Code 12 – Example of flow control from the USB receive

The `TX_FREE` event, send on request by µTasker drivers when their output buffer has adequate space to accept further data, is not explicitly handled in the USB task because its function of waking the USB task is adequate to allow the process to continue. The result, due to intermediate buffering, is a smooth data flow at the maximum possible communication path rate through the system.

7.3. Sending Data to a Buffered USB Interface

Transmission of data to a buffered USB interface uses the same technique as serial UART transmission. Examples of transmission are shown in Code 7 and Code 9.

The size of the USB output buffer is defined when the USB IN endpoint, or endpoint communication pair, is configured. This output buffer stores data to be sent over the USB bus when the host performs IN token requests. The buffer length has no relation to the endpoint token buffer and so can be used to hold much larger amounts of data than the token buffer ready to be sent by the USB driver. The USB driver takes over the responsibility of efficient transmission with no further intervention by the USB application.

As is the case with the serial interface, the amount of available space in the USB output buffer can be requested by calling fnWrite() with a zero data pointer. The USB transmitter driver can also send the `TX_FREE` event if required, however this is not usually necessary due to the high bandwidth that the USB host provides for USB transmissions from the device.

7.4. Handling Control Endpoint Reception

When the control endpoint 0 of the USB interface is configured it should include the definition of a handling routine for control receptions. This is a callback function as defined by the following configuration parameter:

```
tInterfaceParameters.usb_callback = control_callback;
```

The callback is executed on each received SETUP token which cannot be handled by the USB driver (all standard SETUP tokens, such as requests for strings or standard descriptors, are automatically handled by the driver). The routine is called from within the USB interrupt, which means that it should be kept as short and fast as possible. The following example shows the handling of the CDC control endpoint in the demo project but its framework is generally valid for all such use.

```
// Endpoint 0 callback for any non-supported control transfers. This can be called with either setup frame
// content (iType != 0) or with data belonging to following OUT frames. TERMINATE_ZERO_DATA must be
// returned to setup tokens with NO further data, when there is no response sent.
// BUFFER_CONSUMED_EXPECT_MORE is returned when extra data is to be received.
// STALL_ENDPOINT should be returned if the request in the setup frame is not expected.
// Return BUFFER_CONSUMED in all other cases.
// If further data is to be received, this may arrive in multiple frames and the callback needs to
// manage this to be able to know when the data is complete
//
static int control_callback(unsigned char *ptrData, unsigned short length, int iType)
{
    static unsigned short usExpectedData = 0;
    static unsigned char  ucCollectingMode = 0;
    int                   iRtn = BUFFER_CONSUMED;
    switch (iType) {
    case SETUP_DATA_RECEPTION:
        {
            USB_SETUP_HEADER *ptrSetup = (USB_SETUP_HEADER *)ptrData; // interpret the received data as a
                                                                      setup header
            if ((ptrSetup->bmRequestType & ~STANDARD_DEVICE_TO_HOST) != REQUEST_INTERFACE_CLASS) { // 0x21
                return STALL_ENDPOINT;                            // stall on any unsupported request types
            }
            usExpectedData = ptrSetup->wLength[0];                // the amount of additional data which is
                                                 expected to arrive from the host belonging to this request
            usExpectedData |= (ptrSetup->wLength[1] << 8);
            if (ptrSetup->bmRequestType & STANDARD_DEVICE_TO_HOST) { // request for information
                switch (ptrSetup->bRequest) {
                case GET_LINE_CODING:
                    fnInterruptMessage(OWN_TASK, EVENT_RETURN_PRESENT_UART_SETTING);
                    break;
                case SET_CONTROL_LINE_STATE:                      // OUT - 0x22 (controls RTS and DTR)
                    if (ptrSetup->wValue[0] & CDC_RTS) {
                        fnDriver( SerialPortID, (MODIFY_CONTROL | SET_RTS), 0 );
                    }
                    else {
                        fnDriver( SerialPortID, (MODIFY_CONTROL | CLEAR_RTS), 0 );
                    }
                    if (ptrSetup->wValue[0] & CDC_DTR) {
                        fnDriver( SerialPortID, (MODIFY_CONTROL | SET_DTR), 0 );
                    }
                    else {
                        fnDriver( SerialPortID, (MODIFY_CONTROL | CLEAR_DTR), 0 );
                    }
                    break;
                default:
                    return STALL_ENDPOINT;                // stall on any unsupported requests
                }
            }
            else {                                        // command
                iRtn = TERMINATE_ZERO_DATA;               // acknowledge receipt of the request if
                                                                  we have no data to return (default)
                switch (ptrSetup->bRequest) {
                case SET_LINE_CODING:                     // 0x20 - the host is informing us of parameters
                    ucCollectingMode = ptrSetup->bRequest; // the next OUT frame will contain the settings
```

```
                    iRtn = BUFFER_CONSUMED_EXPECT_MORE;    // the present buffer has been consumed but
                                                              extra data is subsequently expected
                    break;

                default:
                    return STALL_ENDPOINT;                 // stall on any unsupported requests
                }
            }

            if (length <= sizeof(USB_SETUP_HEADER)) {
                return iRtn;                               // no extra data in this frame
            }
            length -= sizeof(USB_SETUP_HEADER);            // header handled
            ptrData += sizeof(USB_SETUP_HEADER);
        }
        break;
    case STATUS_STAGE_RECEPTION:  // this is the status stage of a control transfer - it confirms that
                                     the exchange has completed and can be ignored if not of interest to us
        return BUFFER_CONSUMED;
    default:                                               // OUT_DATA_RECEPTION
        break;
    }

    if (usExpectedData != 0) {
        switch (ucCollectingMode) {
        case SET_LINE_CODING:
            fnNewUART_settings(ptrData, length, usExpectedData);  // set the new UART mode (the complete
                               data will always be received here so we can always terminate now, otherwise
                                        BUFFER_CONSUMED_EXPECT_MORE would be returned until complete)
            iRtn = TERMINATE_ZERO_DATA;
            break;
        default:
            break;
        }
    }
    if (length >= usExpectedData) {// handle any (additional) data here
        usExpectedData = 0;        // all of the expected data belonging to this transfer has been received
    }
    else {
        usExpectedData -= length;  // remaining length to be received before transaction has completed
    }
                                   // handle and control commands here
    return iRtn;
}
```

Code 13 – Control endpoint callback routine

The callback routine is called with one of the following `iType` parameters:

- `SETUP_DATA_RECEPTION` – SETUP token
- `STATUS_STAGE_RECEPTION` – acknowledgement that a status stage was successful
- `OUT_DATA_RECEPTION` – OUT token

A pointer to the frame's start and length are passed using the parameters `ptrData` and `length` respectively.

The example shows the only SETUP tokens of `REQUEST_INTERFACE_CLASS (0x21)` are accepted and all others would cause a stall to be executed.

Requests for `GET_LINE_CODING` (*to inform the host about the serial interface settings*) and commands `SET_LINE_CODING` (*to set serial interface settings*) and `SET_CONTROL_LINE_STATE` (*to set serial interface control lines*) are supported.

Depending on the request and command in question, and also the USB endpoint's token buffer length, there will either be all corresponding data received in a single command or else the data will be collected over multiple calls. The amount of data is contained in the request or command and stored in the static variable `usExpectedData`. Note that the USB host will only ever send one complete request or command at a time, so there is no complication requiring handling multiple transactions.

If the complete data content is present in the SETUP frame it can be handled immediately, otherwise data is collected over multiple frames. This is shown by the SET_LINE_CODING command which may send the data content in following OUT tokens. The static variable ucCollectingMode is set to the request type so that it can be completed when the full data content has arrived. The routine fnNewUART_settings() is used to convert the CDC command content to the local serial interface setting.

In the case of the SET_CONTROL_LINE_STATE command the data content will always fit into one SETUP token (since the command length is 8, the size of the smallest possible endpoint token buffer) and so is immediately executed.

In the case of the GET_LINE_CODING request, the command is not processed in the callback itself but rather 'offline'. An event of type EVENT_RETURN_PRESENT_UART_SETTING is sent to the local task so that it can then prepare the data and send it outside of the interrupt routine. This avoids the interrupt routine from having to handle possibly time consuming tasks, where the response time to the GET_LINE_CODING answer is also not critical. How the data is then transmitted to the control endpoint is discussed in the following section.

Note that the STATUS_STAGE_RECEPTION is not explicitly handled, but it does signify the successful completion of the last data exchange and so can be used if required when this confirmation is critical to the application.

The OUT_DATA_RECEPTION type is not handled specifically since it only occurs when data follows the SETUP_DATA_RECEPTION type when the data needs to be received in multiple tokens, as discussed earlier.

7.5. Sending to an Un-buffered USB interface

The response to the GET_LINE_CODING request from the previous section is an example of data transmission to an un-buffered interface.

```
fnWrite( USB_control,
        (unsigned char *)&uart_setting,
         sizeof(uart_setting));       // return directly (non-buffered)
```

Although the format is very similar to the buffered case, there is one important difference.

- In the buffered case the data to be sent to the USB host will be copied to the intermediate output buffer and then sent by the USB driver in as many IN tokens as necessary to complete the transmission. *The contents of the data passed to the* fnWrite *() function can be destroyed once the* fnWrite() *call returns.*
- In the un-buffered case the fnWrite() call informs the USB driver where the data to be transmitted is situated. There is no intermediate buffer and the USB driver transmits the data directly from its source in as many IN tokens as necessary to complete the transmission. During this time the data needs to remain stable – it can be fixed data from a const location (such as FLASH) or it can be on a statically declared buffer to achieve these requirements, as is the case in the demo example.

7.6. Using Strings and Controlling Dynamic Strings

When string support is activated, the user must deliver the following function:

```
extern unsigned char *fnGetUSB_string_entry
                            ( unsigned short usStringRef,
                              unsigned short *usLength  );
```

This is responsible for returning strings based on a string index, where it returns a pointer to the string and also its length. A return value of 0 represents an invalid string index and causes a stall on the USB bus. Note that strings need to be in Unicode format!

The demo project simply returns the requested string from a list of local strings, but also shows how NULL strings are interpreted as user definable ones. Specifically the serial number string is handled as a dynamic one when the option `USB_RUN_TIME_DEFINABLE_STRINGS` is set. In this case, it builds a string based on the device's unique serial number and stores it for transmission later. Since the string requests are standard requests on the control endpoint 0, which doesn't use a buffered USB interface, the string content must remain stable during transmission as discussed in the previous section.

The demo example shows a typical method of supporting the string index, which should be suitable for most uses; it requires simply that string indexes start at 0 (*the standard language string*) and increment for each additional string defined. The method of matching string indexes to Unicode strings can however be realised in any manner as desired.

8. Conclusion

This document has given detailed information about how the USB interface is configured, opened and used. Is has shown that, although some USB experience is required to configure the project to be able to achieve successful enumeration with the chosen host driver, the use of the embedded USB interface is very generic in nature. The actual USB class used usually only effects the operation in a minor way.

Data communication via USB can be compared to the operation of a standard bi-directional serial port due to the fact that the driver interface supports high speed buffered operation and also allows endpoint pairs to be grouped together to function equivalently to a bi-directional channel. This allows reusing serial interface code for USB communication and vise-versa. Command and control interfaces can be made fully compatible, connected to either interface type.

Modifications:
- V0.0 6.3.2009 First draft. Chapter 7 still open.
- V0.1 10.3.2009 Second draft. Add Appendix A with Coldfire details. Chapter 7 still partly open.
- V0.2 10.3.2009 First release. Chapter 7 completed.
- V0.3 10.3.2009 PID/VID details added.
- V0.4 26.6.2010 SAM7X hardware dependencies extended. Document name changed to from `uTaskerV1.3_USB_User_Guide` to `uTasker_USB_User_Guide`

Appendix A – Hardware Dependencies

a) Freescale M522XX

The USB controller in the M522XX is realized in *little-endian* format. This means that, although the Coldfire™ processor itself operates in *big-endian* mode, all addresses programmed and read from the USB controller need to be correspondingly converted.

The µTasker driver software does this transparently and so the user doesn't need to be aware of the details. The only time when this may be of interest is when the USB driver operation is analyzed or during debugging within the USB driver itself.

The operation is based on a Buffer Descriptor Table (BDT) which caters for up to 16 endpoints, including the default control endpoint 0. The BDT resides in SRAM and must be aligned to a 512 byte boundary. The buffers used by the endpoints are also in SRAM, their location being saved within the BDT (in little-endian format). This memory configuration is illustrated in figure A-1, whereby the actual number of entries in the BDT depends on the number of endpoints used, defined by `NUMBER_OF_ENDPOINTS` in `usb_application.c`. Since `NUMBER_OF_ENDPOINTS` represents the number of endpoints required by the active USB connection, one is also added for the control endpoint 0.

There are two methods which can be used for the management of the BDT in SRAM:

- ***Default method, requiring no programmer intervention:***

  The µTasker USB driver allocates the required BDT (actual size depending on the number of endpoints) and the endpoint buffers (size of each depends on the endpoint buffer sizes, whereby endpoint 0 uses buffers of `ENDPOINT_0_SIZE` – defined in `config.h`. The additional endpoint sizes are defined individually when the interface is configured, according to their respective endpoint buffer sizes). The endpoint buffers are long word aligned in SRAM, taken from the heap memory and the BDT is 512 byte aligned in SRAM, also taken from the heap memory.

  The advantage of this method is its simplicity since the details about the SRAM use are fully hidden to the user.

  The disadvantage of this method is that the heap use efficiency depends on the heap memory state as the BDT is requested. Since it needs to be aligned to the next available 512 byte boundary, it is possible that this leaves a hole in the heap memory before the BDT start location (*in order to ensure that the boundary condition is fulfilled*) of up to 511 bytes. The heap usage in a project can thus change from +0 to +511 bytes depending on the address of the top of the heap as the USB interface is configured.

- ***Linker method, requiring programmer intervention***

  The second method which is supported is to allocate the BDT at a fixed location defined in the linker script file. This allows its start location to be defined directly after the interrupt vector table (which is 1k in size and thus its end is automatically aligned to the required 512 byte boundary). The actual size of the space reserved by the linker script does however require the size of the BDT (which is project specific since its size is given by `(ENDPOINT_BD_SIZE * (NUMBER_OF_ENDPOINTS + 1)))`, where `ENDPOINT_BD_SIZE` is 32 bytes) to be know and manually entered.

The programmer can thus use this method to optimise the memory use by the BDT to match exactly the BDT requirements. It is however important to correctly configure the linker script to respect the real size of the BDT as required by the specific project.

The potential SRAM saving by using the linker method lies between 0 and 511 bytes. It can thus be used when this saving is critical in a project. If it is not used the heap size should be set to adequately respect the fact that the used heap size has a tolerance of +511 bytes. If this tolerance can be added without memory resource difficulties, the use of the first method is fully acceptable.

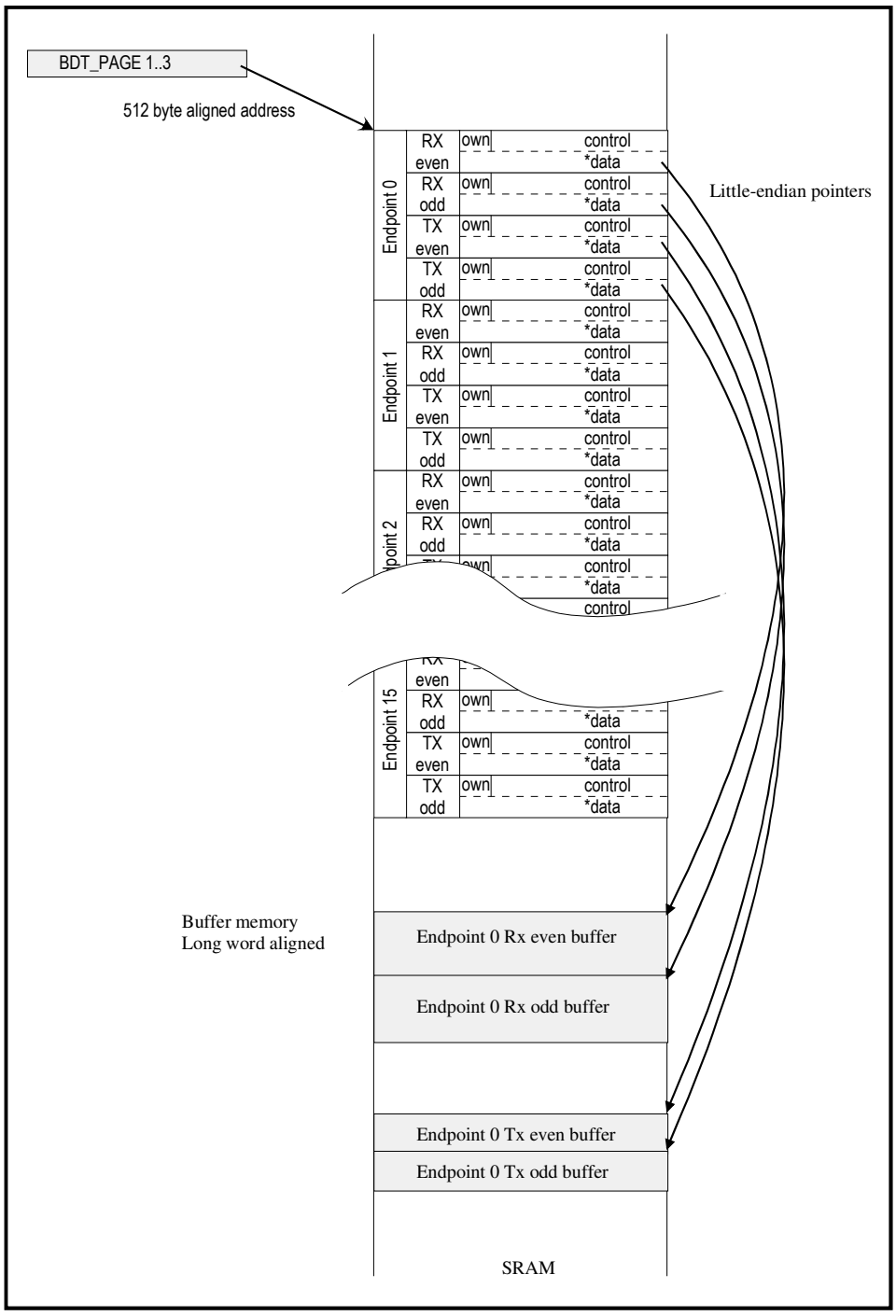

Figure A-1. M522XX Buffer Descriptor Table and endpoint buffers in SRAM

The following illustrates how to modify the linker script file for the Codewarrior project. The GCC linker script is extremely similar and the technique can be applied there too in a similar fashion.

First some RAM is defined in the memory section which will be used by the BDT. The length is 128 (0x80) which suits the demo project example; this uses 3 endpoints as well as the default control endpoint 0 and subsequently requires 32 x 4 = 128 bytes of RAM, aligned to the 512 byte boundary. The system variables (in the sram section) thus avoid this region and start 128 bytes later in memory.

```
MEMORY
{
    flash1  (RX)    : ORIGIN = 0x00000000, LENGTH = 0x000400
    flashconfig   (RX)    : ORIGIN = 0x00000400, LENGTH = 0x00000018
    flash2  (RX)    : ORIGIN = 0x00000420, LENGTH = 0x003FBE0
    vectorram(RWX)  : ORIGIN = 0x20000000, LENGTH = 0x00000400
    usbrram(RWX)    : ORIGIN = 0x20000400, LENGTH = 0x00000080
    sram    (RWX)   : ORIGIN = 0x20000480, LENGTH = 0x00007B80
    ipsbar  (RWX)   : ORIGIN = 0x40000000, LENGTH = 0x0
}
```

Then the location is defined by

```
      ___VECTOR_RAM     = ___SRAM;
      ___IPSBAR         = ADDR(.ipsbar);
      ___USB_BDT_RAM  = ___SRAM + 0x400;
```

Finally the second method is enabled in the driver code by adding the following define to `app_hw_m5223x.h`:

```
#define USB_BDT_FIXED // used a fixed address in SRAM for the USB BDT. The
                                linker file needs to be set up accordingly
```

Rebuilding the project using this configuration will cause the USB buffer descriptor table to occupy 128 bytes from 0x20000400 and avoid allocating them on the heap.

Note that, as seen from the BDT diagram in figure A1, each endpoint is double buffered. This allows high throughput to be achieved since a second buffer can be prepared whilst the first buffer is still in the process of being transmitted.

A final point worth mentioning about the USB controller in the Coldfire™ is that it doesn't automatically have rights to access FLASH since it performs such accesses per DMA. Since the µTasker USB driver works with most configuration tables directly from FLASH, the driver automatically enables DMA accesses to FLASH when the USB driver is used. Furthermore it automatically routes message data which reside in FLASH to the "back-door" address range of FLASH. The result is that there are no limitations as to where data for transmission over the USB interface reside.

Thanks are expressed to Freescale for supplying the following PID/VID for the µTasker project development use with Freescale devices:

VENDOR ID 0x15a2 - Freescale's Vendor ID
PRODUCT ID 0x0044 - uTasker development project for Coldfire

b) Atmel AT91SAM7X

The SAM7X (generally SAM7) devices include an integrated USB device controller. This is FIFO based, meaning that the transmitted and received data are written to and read from dedicated FIFOs. These do not reside in SRAM memory but instead each endpoint has its own FIFO register address. Data read from a FIFO generally needs to be temporarily stored in a linear buffer for treatment of its content. Reading from the FIFO address a second time doesn't retrieve the same data content but instead retrieves the next waiting buffer content. For this reason, the USB driver code has some specific differences when FIFO buffers are used instead of linear data buffers.

The SAM7 has 6 dedicated endpoints which can be configured according to the following table:

Endpoint Number	Dual-Bank	Max. Endpoint Size	Endpoint Type
0	No	8	Control/Bulk/Interrupt
1	Yes	64	Bulk/Isochronous/Interrupt
2	Yes	64	Bulk/Isochronous/Interrupt
3	No	64	Control/Bulk/Interrupt
4	Yes	256	Bulk/Isochronous/Interrupt
5	Yes	256	Bulk/Isochronous/Interrupt

Table A1: SAM7 Fixed Endpoints

This restricts the number of endpoints that can be used, although this is not necessarily a problem in typical applications. Endpoint 0 supports only 8 bytes and the endpoints 4 and 5 are most suitable for isochronous use due to the fact that they have up to 256 byte FIFOs, which is however still less that the maximum FIFO depth of 1'023 bytes that could be used for a full-speed isochronous device.

Dual-banked FIFOs allow higher throughput to be achieved because the CPU can be handling or preparing a second FIFO buffer while the first is being treated by the USB controller. The double-buffer mechanism is mandatory on isochronous endpoints in order to achieve the continuous data rate.

Figure A2 shows double buffered reception (OUT) transfers when a data queue is involved. Specifically the case is shown where the input SW queue become full and the endpoint needs to be blocked while the application clears data from this input queue, thus making room for further reception.

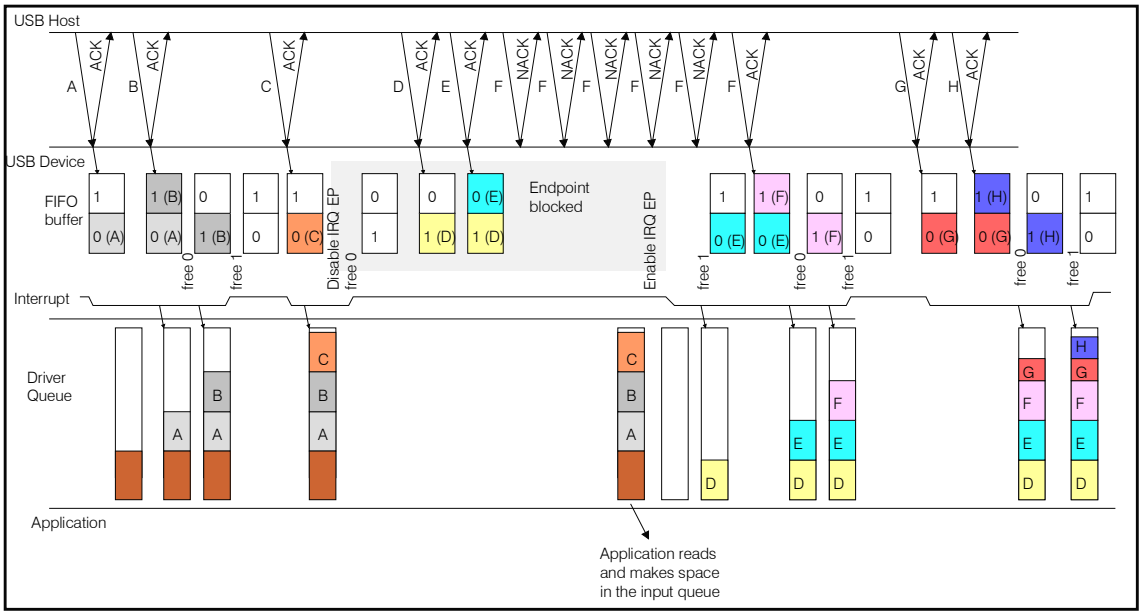

Figure A2: OUT endpoint dual-buffered flow control

The USB device processing is based on interrupts and the exact sequence is timing dependent because the interrupt routine may read one, two or even more buffers at a time depending on how many buffers are pending and whether a further buffer becomes pending during the interrupt processing itself.

The figure shows a typical case where the double buffered input operation is shown. The driver input buffer becomes full (not enough room to be able to accept a further complete buffer), causing the endpoint's interrupt to be masked (endpoint blocked). Although the endpoint is blocked, its buffers can still accept up to two frames, filling the double buffered FIFO, after which further attempted OUT transactions are automatically NAKed by the USB device. The USB host will continue repeating the OUT frame until it is finally ACKed. It is ACKed again after the application has removed data from the input queue so that there is enough room to accept at least a further full data transaction. After the endpoint is no longer blocked, the dual-buffer processing can continue until the flow control needs to be initiated again.

Full-speed bulk connection types will send up to 19 frames in a 1ms frame period, which means that the CPU must handle endpoint interrupts about once every 50us when continuous data is being received. Up to 64 bytes of data need to be read on a byte basis out of the FIFO buffer and possibly copied into a driver buffer (as is the case in figure A2) to allow the application to handle the data as larger blocks. The maximum throughput rate of about 9.7MBit/s (respecting overhead in the data stream) thus represents a high load for the CPU, which needs to be considered when other tasks are also performed in parallel.

The IN endpoint operation is effectively the inverse of that shown in figure A2.

Thanks are expressed to Atmel for supplying the following PID/VID for the µTasker project development use with Atmel devices:

VENDOR ID 0x03eb - Atmel's Vendor ID
PRODUCT ID 0x21fd - uTasker development project for SAM7X

c) Luminary Micro – LM3Sxxxx

The Luminary-Micro USB controller supports 3 configurable endpoint pairs and has 2 dedicated control endpoints when operating in device mode.

The interface is realizes as a 4k FIFO buffer, where the endpoint 0 uses the first 64 bytes as shared memory for both IN and OUT transactions.

Thanks are expressed to Luminary Micro for supplying the following PID/VID for the µTasker project development use with Luminary devices:

VENDOR ID 0x1cbe - Luminary Micro's Vendor ID
PRODUCT ID 0x0101 - uTasker development project for LM3Sxxxx