SMSC™

STANDARD
MICROSYSTEMS
CORPORATION

APPLICATION NOTE 7.19
V1.1

USB97C100 Programmers Reference Guide

CHAPTER 1 - INTRODUCTION

The basic architectural concept of the USB97C100 device is that all high bandwidth data flow be handled entirely in hardware, with an integrated MCU (MicroController Unit, an 8051 derivative) acting only to manage the flow of data through the various hardware engines. At the center of the device is a multi-ported MMU (Memory Management Unit) that dynamically allocates and frees memory pages grouped into virtual packets both automatically in response to USB traffic, and also under software control by the MCU. On one port of the MMU is the SIE (Serial Interface Engine) that provides a fully hardware-driven interface to the USB, while another port on the MMU is connected to a partial ISA bus interface containing a DMAC (Direct Memory Access Controller, an enhanced 8237 type) that provides a fully hardware-driven interface to external peripheral devices. All three (3) hardware engines (SIE/MMU/DMAC) are capable of operating concurrently with each other and with the MCU, while the DMAC is also capable of interleaving transfers to/from multiple devices concurrently.

As part of its function, the firmware must establish flow control in order to prevent overrun from either the USB or the DMAC in the event that the target of the transfer is not as fast as the data source. Also, while the SIE takes care of all bit and packet level USB protocol issues in hardware, the higher levels of the USB device protocol (e.g., Default Pipe traffic) are the responsibility of the firmware. Note that since this traffic is not bandwidth intensive, the firmware implementation results in absolute flexibility combined with reduced cost, without adversely affecting performance.

This document, along with the companion code disk, describes detailed register level programming considerations, along with working examples, of the device architecture in general, and USB applications in specific.

INTENDED AUDIENCE

The intended audience of this document is primarily software engineers, and perhaps their managers, involved with writing firmware for the USB97C100 device. Hardware and systems engineers can also benefit from at least skimming this document in order to better understand the effect of their design decisions on the firmware component of the system.

The reader is assumed to be experienced in register level hardware programming, either in an embedded or Device Driver context. A basic knowledge of the C programming language is required, since all of the example code is written in C, as is an understanding of basic USB operation. Prior knowledge of the 8051 MCU and 8237 DMAC is desirable, but is not a prerequisite.

80 Arkay Drive
Hauppauge, NY 11788
(516) 435-6000
FAX (516) 273-3123

TABLE OF CONTENTS

REFERENCES

Universal Serial Bus Specification Revision 1.1 September 23, 1998 as well as various Device Class Specifications and White Papers are all available at http://www.usb.org. This site also has links to many other useful sites.

SMSC USB97C100 and FDC37C67x Data sheets, both available at http://www.smsc.com. The 67x is the SIO device on the EVB97C100.

8051 Data Sheet, available from numerous manufacturers, including Intel Corporation, the originator. This is the MCU inside the USB97C100.

8237 Data Sheet, Intel Corporation. This is the DMAC inside the USB97C100.

SMSC LAN91C94/5/6 Data Sheets, available at http://www.smsc.com. These devices have a very similar MMU to the USB97C100, with explanation.

The C Programming Language, Brian W. Kernighan and Dennis M. Ritchie, Prentice Hall; can be useful for those having little experience with the language.

ACRONYMS

BER -- Bit Error Rate
CFES -- CLEAR_FEATURE(ENDPOINT_STALL), a USB command
COM -- For the purposes of this document, an RS-232 serial COMunications port
DMA -- Direct Memory Access
DMAC -- Direct Memory Access Controller
EP -- EndPoint, as in a USB EP
EP0 -- Endpoint zero, the default pipe for a USB device
GPIO -- General Purpose Input/Out, as in a device pin
IRQ -- Interrupt Request
ISO -- For the purposes of this document, ISOchronous, as in a USB ISO transfer, not International Standards Organization
ISR -- Interrupt Service Routine
LPT -- For the purposes of this document, a PC legacy parallel Line PrinTer port
MCU -- Microcontroller Unit
MMP -- Memory Management Policy
MMU -- Memory Management Unit
MPU -- MicroProcessor Unit
POR -- Power-On Reset
RTL -- Run Time Library
RxEP -- Receive EndPoint, as in a USB RxEP
SFES -- SET_FEATURE(ENDPOINT_STALL), a USB command
SIE -- Serial Interface Engine, as in a USB SIE
SIO -- Super Input/Output
SOF -- Start Of Frame, as in a USB frame
TxEP -- Transmit EndPoint, as in a USB TxEP
USB -- Universal Serial Bus

CHAPTER 2 – BACKGROUND INFORMATION

This chapter is a potpourri, covering a range of topics including:

- A primer about writing C code for the MCU, including a discussion and example program illustrating the performance impact of coding style and address space usage, as well as a variety of Interrupt handler considerations.
- A description of the development and build environments, including how to set them up and use them to execute and rebuild the example programs.
  A description of the example programs, including coding conventions and common files

Why C?

The examples in this document use the C language, as opposed to either ASM or C++ for a variety of more clear because they are less cluttered with detail than they would be if they were written in ASM. The quality of C compilers for the MCU has improved to the point that there is there once was. When combined with increased time to market pressure, this makes C a reasonable choice for implementing an

language, again making it a reasonable choice for a document of this type. For embedded developers

smart) ASM generator (just turn on the ASM listing option), which actually is not a bad way to think of the compiler anyway when the target is a small embedded MCU.

seem to have had as much penetration as C at the present time, especially for low-end MCUs. Maybe some day C++ will be the language of choice for a document of this type, but not today.

use of non-standard, and hence non-portable, extensions to the ANSI specification of the language. This makes writing C code that is even portable to different compilers for the same MCU an exhaustive Also, the very purpose of this document is to illustrate coding techniques that are specific to a particular hardware architecture so, even if the source code could be recompiled for a

being present. Having said this, the example programs are written in such a way as to reduce the porting burden to the extent possible (e.g., non-ANSI typedef's are in header files, no "//" in-line comments are

C on the 8051

This section makes extensive reference to the detailed operation of the 8051 this architecture are referred to the data sheet for a complete description. Developers with substantial experience writing C for embedded MCUs will likely be familiar with this material, and so this section is

later sections will discuss the 8237 DMAC in detail, at which point the tables will be turned. For the developers with a PC background, the closest PC equivalent to programming this MCU is the old MS-

with fixed offsets to the beginning, etc. If you wanted to touch or change any memory or I/O port, you could have your way with it without having to ask any O/S for any appproval or assistance. You were maybe a 1 MIPs machine, and not a modern PC with hundred of MIPs and nearly as many MB of RAM, and a big fat OS between you and the hardware. If you are old enough to remember those

do that for a living, then read on...

When C is compiled for MPUs, arguments are usually passed on the stack, and automatic variables are allocated on the stack. For MPUs with a stack of substantial size, and with the capability to efficiently address the stack memory, this makes a good deal of sense. However, the MCU has neither of these capabilities. As a result, parameter passing is usually done through registers, or in fixed memory locations, or on a "simulated" stack if the arguments cannot fit in registers. Use of a simulated stack is very slow, so it is suggested that any functions for which speed is a consideration limit their arguments to the types and numbers that can be passed in registers. Even then every extra argument passed increases the overhead of the call, so "less is more" when it comes to performance. One way of satisfying this suggestion would be to pass a single pointer to a structure, but that is not a good solution for the MCU because it does not have the flexible addressing modes (e.g., based indexed displaced addressing) that CISC MPUs usually have. The MCU is also not very efficient at doing the address calculation necessary to access a complex data structure. Although it is generally frowned upon (and for good reasons), the use of global variables is the best solution from the perspective of performance.

In addition, short functions are good candidates for implementation as macros; as macros, there is no call and return overhead, which can be a substantial portion of the total execution time when functions are short. Also, the final code size might not even be larger because all of the code related to parameter passing and register saving/restoring is eliminated. Using macros instead of small functions is a good thing for the MCU. Another benefit of macro functions is that they avoid the issue of reentrancy, which is discussed next.

Automatic variables are usually not placed on the MCU stack both because the MCU would not be able to efficiently access them if they were, and because the stack tends to be small. Instead, automatic variables are usually allocated in one of the data address spaces of the MCU, which are described in more detail in a later section. In order to avoid wasting valuable data memory for automatic variables that are not presently being used, a good Linker/Locator will use data overlays within the procedures of a given thread. Since different threads (e.g., foreground and ISR) can execute concurrently, their data cannot be overlayed. This brings up the issue of recursive and reentrant code. While it is possible to compile code for the MCU so that it is reentrant (i.e., executed by more than one thread at a time), such code cannot have its data overlaid with the data from any other thread, so it consumes more memory. Also, its data must be relocatable because it might be necessary to have multiple instances at the same time. Due to a lack of efficient addressing mechanisms for this type of allocation in the MCU, such code will execute much more slowly than standard functions. The situation is essentially the same for recursive functions, which call themselves within the same thread, because of the need for multiple simultaneous data instances. In general, both are undesirable from a performance point of view. Avoiding recursion mostly involves algorithm design, so there is not much that can be presented here with respect to universal techniques. However, there are two easy ways to avoid reentrancy: one is to use macro's, as was described above, and the other is to cut-and-paste the function and to give each copy a slightly different name. This can work very well if the reentrancy is between two different threads, for example the foreground thread and a single ISR. Having duplicate functions may increase code size somewhat, but each copy is smaller and executes much faster than a reentrant version of the same function, so the increase in code size, if any, is often worth it.

In order for data overlays to work properly, the Linker/Locator must be able to unambiguously determine the execution context for every code section, which can be difficult if the code calls through function pointers. The easiest solution to this is not to use function pointers, but if they must be used, then the compiler should be set to produce an ASM listing, and this file and the Linker output files should be carefully inspected to be sure that the tools are correctly implementing the software design as intended.

Note that the Linker/Locator will often mistake uncalled functions for separate threads and will not overlay their data with any of the actual threads. This behavior can be somewhat annoying during initial coding, in which case it is common to write a number of functions before actually using any of them. One solution, albeit not a pretty one, is to create a dummy function that calls each of the uncalled functions with dummy arguments conditional on an argument to the function being TRUE; this dummy function is then called from e.g., main() with a binary flag that is FALSE, so the code never actually executes, but it gives the Linker enough information to know how to overlay the data; see Chapter 3 and Chapter 6 for an example of this. Some compilers also permit defining the thread each function is supposed to execute in, which achieves the same result. The only problem with this approach is that each function's source code needs to be changed in order to move it from one thread to another.

Considering the combination of the above issues, it is usually a good idea to develop a function hierarchy that is relatviely flat (i.e., not too deeply nested) and orderly, without a lot of cross-calling. This combination reduces the impact of call/return/parameter passing overhead and gives the linker good opportunity for achieving RAM savings through the data overlay mechanism.

The MCU does not contain a barrel shifter; as a result of this, shifts are implemented one bit at a time in a loop, so they are best avoided where possible. However, a good compiler will recognize shifts by a multiple of 8 as being a change in address, and this can be handled quite efficiently, especially if the argument is in DATA space. Examples of this would include macro's for HIBYTE(a), LOBYTE(a) and MAKEWORD(a,b) (see Type.H for the definitions of these macro's).

Another efficient sequence involves a conditional jump based on a single bit in a byte being either set or cleared -- the MCU has JB (Jump if bit set) and JNB (jump if bit clear) instructions, and a good compiler will use these whenever it gets a chance. For example,
if (myVar & 0x08) { }.

Most compilers offer the option of producing an ASM listing in addition to the C listing, and it is often useful to enable this feature and at least glance at what the compiler is doing, especially when working in a new environment. Sometimes small changes in the source code can make big changes in the resulting object code. The linker output file should also be inspected, especially to make sure that the linker is correctly understanding the memory map (e.g., 256 bytes of internal RAM, as opposed to some other size), that the stack is of sufficient size, and that any external data RAM or firmware ROM is properly located. It is also the author's personal preference to set all warning levels to maximum and to take any warnings seriously, but it is recognized that this is a matter of individual taste.

Speaking of ASM, if it happens that there is some function for which execution speed is absolutely critical, it is always possible to write some of the code in ASM and call these ASM functions from C. Since the quality of the code produced by compilers has improved much in recent years, this should not be necessary, and the performance gain will likely be small if it is done, but it is always an option.

MCU Address Spaces

The MCU contains 256 bytes of internal RAM. The low 128 bytes of this address space can be accessed using either direct or register indirect (e.g., @R0, @R1) addressing, while the high 128 bytes can only be accessed using indirect addressing. Since direct addressing is somewhat faster than indirect addressing, it is desirable to locate variables whose access speed is important in the low 128 bytes. A portion (16 bytes) of the low 128 bytes can also be addressed as individual bits and, unlike most MPUs, the MCU contains a special Boolean Processor and so is quite efficient at manipulating these. The MCU also has an external data memory address space of 64 KB. However, access to this address space is substantially slower than either direct or indirect addressing modes because it involves the use of the 16-bit DPTR register.

[For completeness, the upper 128 bytes of internal memory, when used with direct addressing, accesses yet another address space: the Special Function Register, SFR, and there is also an external address space mechanism that only uses an 8-bit address with paging. However, neither of these is salient to the discussion which follows.]

C compilers for the MCU are quite flexible in their ability to enable the programmer to define where variables are stored and how they are accessed based on how they are declared in the source code. Since ANSI C has no provision for this, compilers implement extensions to the ANSI language, which limits portability between different compilers, even for the same MCU. For the Keil compiler, variables located in the low 128 bytes of internal RAM are referred to as type "data" and direct addressing is used to access them. Variables located anywhere in the 256 byte internal RAM are referred to as type "idata" and indirect addressing is required to access them, even if they are ultimately located in the low 128 bytes, since their actual location is not known at compile time. Variables located in the external data address space are referred to as type "xdata", and the DPTR register is always used to access them. Bit addressable variables are referred to as type "bit" for obvious reasons.

If all of this seems a little bit bizarre, well it is. If all of this seems a little confusing, do not worry about it. The C language makes all of this easy to hide in header files (for the most part), which are included in the example programs.

The on-chip peripherals in the UCB97C100 (e.g., the SIE and MMU) are mapped into the xdata address space. In addition, there are 3 addressing windows, all of which are mapped into the xdata address space of the MCU, that permit access to the address spaces of the external buses. There is a window for the ISA I/O address space, the ISA Memory address space and the "Flash" Bus address space. [As an aside, note that the name "Flash" Bus is something of a misnomer, in that the bus can be used to interface to any combination of memory devices (ROM/EPROM/EEPROM/FLASH/RAM, etc.) and/or memory-mapped peripheral devices; a more descriptive name would be the MCU Bus, since it is a generic bus that is owned full time by the MCU.] Each memory window has its own bank select register associated with it, which is also mapped into xdata space, that permits moving the window anywhere in the entire address space of the corresponding bus. The following table summarizes the xdata address windows:

Address Space (Bus & Type)	Total Size	Window Size	Window Location (xData)	Bank Select Register (xData)
ISA I/O	64 KB	256 bytes	0x4000-0x40ff	IOBASE [0x7F71]
ISA Memory	1 MB	4 KB	0x5000-0x5fff	MEMBASE [0x7F72]
Flash Memory	1 MB	16 KB	0xc000-0xffff	MEM_BANK [0x7F29]

The MEM_BANK register also controls paging in the code address space of the MCU, and this must be considered if the entire firmware is larger than 16 KB. In particular, the bottom 16KB of code address space (0x0000-0x3FFF) always maps straight-through to the Flash Bus (i.e., 0x00000-0x03FFF). The next 16 KB of code space (0x4000-0x7FFF) is a movable 16 KB window whose 6 MSB's are controlled by the MEM_BANK register. The upper 32 KB of both the code and xdata spaces is a duplicate (i.e., an alias) of the lower 32 KB of code space. The following table summarizes Flash Bus mapping into the MCU address spaces:

ADDRESS (SIZE)	CODE	XDATA
0xC000-0xFFFF (16 KB)	Set by MEM_BANK	Set by MEM_BANK
0x8000-0xBFFF (16 KB)	0x0000-0x3FFF	0x0000-0x3FFF
0x4000-0x7FFF (16 KB)	Set by MEM_BANK	(On-chip peripherals & ISA)
0x0000-0x3FFF (16 KB)	0x0000-0x3FFF	(On-chip SFR's, etc)

Unlike the other on-chip peripherals, the on-chip DMAC is mapped into the ISA I/O address space, so its registers are accessed in the MCU xdata address space, just like the SIE and MMU, once the IOBASE register is set.

From the discussion above, it might seem undesirable that the on-chip peripherals are all mapped into the xdata address space, and it is when considering the performance penalty involved. However, the other address spaces are very limited in size, and it would be even worse to lose a substantial portion of one of those address spaces instead (especially data or idata). It is important for the device programmer to recognize this situation because it has an impact on firmware performance. Firmware should be written so as to access device registers as little as possible. For example, if the value contained in a register is needed multiple times, then it should be read once and saved in a variable, and then this variable can be read multiple times with far greater speed.

The existance of all of these different address spaces raises an interesting question: what happens with pointers? The answer is: it depends. Each address space (data, idata, xdata and code) needs either a 1 or 2 byte pointer to span it. As a result, if the particular address space that a pointer references is known or implied by usage (e.g., perhaps it is explicitly typed when it is defined), then a straight pointer is all that is needed. However, if a pointer is to be "generic" in that the same pointer can be used to reference any address space, then an extra byte must be added to it in order to identify the address space to which the pointer is currently assigned -- hence, the 3-byte pointer. Some compilers provide support for 3-byte pointers, some only support 3-byte pointers, and some do not support 3-byte pointers at all, comprising yet another portability issue. It is the author's opinion that generic pointers are a mixed blessing: they are terrible from a performance point of view, but they do permit writing code with fewer non-ANSI directives in it, and they make it trivial to move data items from one address space to another because none of the pointers to the data need to be changed when the data is moved (by re-typedef'ing it).

Interrupt Service Routines (ISRs)

One very nice feature of the MCU is that it contains multiple Register banks. The current register bank can be changed quickly by setting 2 bits in the PSW. This can be used to substantially reduce the interrupt latency time by avoiding having to save the entire register bank on entry to the ISR (and restoring it on RETI). Compilers for the MCU support this feature by (surprise) using extensions to ANSI C. For the Keil compiler, the "using" function attribute causes the compiler to insert code in the function that will save the existing register bank and switch at entry, and will restore the previous register bank at return.

In the USB97C100 architecture, the most important ISR is for IRQ 0, which is the one that handles USB traffic. It is desirable to keep the latency for servicing this IRQ as short as possible, and the use of Register Bank switching is strongly encouraged. Note that each additional register bank used is located in the "data" address space, so some RAM is lost this way, but the alternative would be to push the registers on the Stack (usually in "idata" space), so the total RAM consumed is the same, and the bank switch is much faster.

One thing to be careful about is the handling of the **ISR_0** register, which contains the Interrupt Status bits -- **<u>this register self-clears on read!</u>** If a bit in this register representing a condition that should cause an IRQ is cleared by accident, then no IRQ will happen, and losing IRQs is not usually a good thing. One simple way to avoid this problem completely is to read the physical **ISR_0** register in exactly one place in the entire program, and that place is right at the entry to the ISR. A copy of the register can then be passed to the individual handler functions if desired. This method might cause the ISR to be executed spuriously in the case where one of the handlers takes care of a condition after the physical **ISR_0** register was read, but it guarantees that no IRQs will ever be lost. Of course, this is just one way, and certainly not the only way, of dealing with the **ISR_0** register, and any other method that yields a properly working solution is perfectly valid. The point being made here to be aware that mishandling **ISR_0** can cause serious problems. In particular, it is probably never a good idea to read **ISR_0** from a foreground thread.

As in any interrupt-driven system, care must be exercised if any hardware or software resources are shared between the foreground and background. The classic situation to avoid is the one in which, while the foreground code is in the middle of a read-modify-write, the ISR executes and changes the value, and then the foreground over-writes the value from the ISR. An example of such a case in this device is the **GPIOA_OUT** register in the situation when both the foreground and ISR threads are manipulating GPIO pins, but the same class of situation can result as a matter of sharing software resources (e.g., RAM variables) rather than hardware registers. One way to avoid this problem is for the foreground thread to disable IRQs while accessing the shared resource, but other mechanisms are possible as well. If IRQs are disabled by the foreground thread, then it should be for the shortest amount of time possible (ideally just a couple of microseconds) in order to avoid a significant negative impact on the IRQ latency time.

In other situations, a portion of the hardware is shared between the foreground and ISR threads, but there is no read-modify-write issue; a common example of this is a numeric coprocessor in an MPU system. In these cases, the ISR can simply save/restore the register set so that the foreground thread does not even know the ISR had used the hardware. An example of such a situation in the USB97C100 is the MMU, with the **PNR**, **PRL** and **PRH** registers.

Also as usual, any variables used by the ISR need to be globals in one way or another and, in order to avoid problems with reentrancy, do not call any of the ISR code from the foreground. Of course, the ISR itself can never be called from any foreground thread because it uses a RETI, rather than a RET.

Performance Perspective and Strategy

Although much of the previous discussion was concerned with the impact of specific coding techniques on performance, it is useful to realize that much of the code in any given application for this component is not related to performance in any way. For example, when a USB device is first attached to the Bus, it is enumerated, reset, configured, etc. Required execution times for these operations (per the USB Specification) is measured in units of milliseconds, so performance is not a concern here. There are also occasional Control Transfers to the Default Pipe (EP0), but these are also not performance sensitive since they do not occur with high frequency. When writing code of this type, it is permissible to use any and all of the techniques (e.g., complex data structures, etc.) that might have an adverse effect on performance if using those techniques is appropriate. For example, the USB descriptors are essentially a complex data structure, so writing them that way is a natural expression of the coding solution.

As far as the remaining code is concerned, it is helpful to quantify just what performance level is required or desired. For example, Isochronous applications are essentially hard real time -- it is an absolute requirement that the software be fast enough to handle the stream, but any further speed improvement serves no useful purpose. In the case of Bulk data applications, once the software is fast enough to saturate either the USB or the peripheral device, the same situation results. From the software perspective, performance in this component architecture is really an issue of how many packets are handled in each USB frame. Note that this is different from the situation in many other USB components in which the actual data bytes must flow through the MCU. **Based on this definition, a USB camera that delivers 1,000 byte Isochronous packets is a low-performance application with respect to software since only a single packet needs to be handled each USB frame.**

Once specific performance numbers are established, the data flow needs to be planned:

Generally, USB OUT packets will arrive in an Isr0() function, which is the ISR that handles interrupts from USB traffic flow (any RX, any TX, etc.). At arrival, the Rx packets need to be validated and saved for consumption by the foreground device handlers, either in software or hardware queues (the component has both). This is also the time to update the current packet count for each BULK EP and, if an EP has reached its limit, to make that EP "busy" so that further OUT packets to that EP will be NAKd. In the foreground, each device handler checks the state of its peripheral device. If the peripheral device has just completed transferring a packet, then the handler must free the packet in the MMU so that the packet memory can be used for receiving additional packets. If the EP is Bulk, then the handler must update the packet count and, if it is low enough, make the EP "not busy" so that future OUT packets sent to that EP will be received and ACKd. Finally, if the peripheral device is ready for another packet and one is available in its packet queue, the handler must start sending the next packet to the peripheral device. For high performance peripheral devices, a DMA hardware interface should be used (rather than PIO), so starting the next packet involves setting up a new DMA session.

The data flow for Tx is similar to Rx, but backwards. If the peripheral device is below its limit on the number of packets it is permitted, and the peripheral device is ready to fill a new packet, then the handler must allocate a packet and start the peripheral device filling it. If the peripheral device has finished filling a packet and is below its limit on how many it is allowed to queue for transmission on the USB, then the handler must queue the full packet. In both cases, the count of packets owned and packets queued must be adjusted. After the Host reads the packet on the USB, the packet will appear in the Tx Completion queue. It is a matter of choice by the programmer whether to handle this in Isr0() or in the foreground but, regardless of wherever it is done, the packet must be removed from the completion fifo, and the count of packets owned and packets queued must be updated.

[To be truthful, the word "must" in the paragraphs above is a little bit strong, since simplifications are often possible, but the description represents the most general case.]

From the above, it can be seen that the MMU acts in a managerial role supervising the traffic flow and ownership of shared finite resources such as packet memories and Tx queues. There are at least two obvious ways that code like this can be implemented: either centralized or distributed. In the centralized case, there is a set of functions that encapsulate data structure(s) that maintain the present state of the system, and decide when to busy/un-busy RX EPs and when to grant or refuse requests for packet allocations and tx packet queuing. In the distributed case, each individual EP handler function maintains its own state and decides when to allocate, free and queue packets, etc. As usual with software, still other approaches are possible, and any approach that yields a correct solution is equally valid.

Since all of the above code executes for every packet that is transferred on the USB, the execution speed of this code is critical if high performance, defined in terms of the number of packets per USB frame handled, is to be achieved. From the previous discussion, a number of techniques can be applied in order to obtain best performance:

locate all variables in data or idata address space
implement any short functions as macros
bank switch the registers in the ISR
keep the function hierachy shallow
consider global variables -- they are necessary for the ISR anyway
if any arguments are passed to functions, pass them in registers
avoid complex data structures -- use simple arrays or scalars
avoid 3-byte pointers if possible

Using all of these techniques in combination will make a big difference in the execution speed of the code, and can still result in very legible and maintainable code if thoughtfully applied. If at that point the performance is still less than desired, it becomes important to understand clearly where the MCU is spending its time. An easy way to determine this is by pulsing GPIO pins at the entry/exit of the major functions. Pulsing GPIO pins in this fashion does add a couple of microseconds to the execution time, but that is relatively small, and the information it provides is critical to understanding where the MCU is spending its time. The functions that are consuming the most time can then be inspected in the ASM listing of the compiler in order to understand why they are taking so long, and to see what, if anything, can be done to reduce their execution times. Sometimes recoding a function in ASM can help, but modern compilers generate reasonably efficient code, so the improvement is usually not much. Any substantial improvement usually comes from changes in algorithms, data structures, or the address space in which variables are located, which is the reason why it is so important to think all of this through carefully before writing the code in the first place.

Example Program

The example program **Chpt2.Hex** illustrates the effects of address spaces and coding style on performance. Unlike the other examples in this document, this program is *NOT* meant to be executed. Instead, the ASM listing should be inspected.

The program consists of a set of 4 functions that each push an entry onto the head of a software queue. The functions differ in the address space and organization of the queue. By inspecting the ASM listing, quantitative performance differences can be determined. The following analysis is for the typical case in which the queue is neither full, nor does the head pointer wrap around.

PushDqueHead() uses simple scalers and an array in data space. The function executes 15 instructions that consume 20 processor cycles.

PushIqueHead() is the same, except that the variables are located in idata space. The function executes 17 instructions in 22 processor cycles. By inspecting the ASM code, it can be seen that the extra instructions and cycles are a result of the head pointer being located in idata space, not the array. If the head pointer was in data space, then the instruction and cycle count would be the same as the previous example. From this it can be concluded that:

1. idata access is only a little slower than data access.
2. idata access for array elements is identical in speed to data access, so arrays should usually be located in idata space.

PushISqueHead() uses a data structure located in idata space. The function executes 24 instructions in 31 processor cycles. From this it can be concluded that there is a major performance impact involved in the use of data structures (in this case 50%), even fairly simple ones. Inspection of the ASM code reveals that the extra time is spent doing the address arithmetic to access the structure elements.

PushIXqueHead() uses the same data structure as above, but it is located in xdata space. The function executes 39 instructions in 64 processor cycles, making it about twice as slow as the same example in idata space, and 3 times slower than the first 2 examples. From this it can be concluded that there is a major performance penalty associated with xdata access. Inspection of all of the DPTR manipulation in the ASM listing shows why this is the case.

Development Equipment

It is not the purpose of this document to specifically recommend or endorse any particular product(s) of any particular manufacturer(s). For each of the items described below, a variety of manufacturers offers a range of products that appear to be suitable. However, in order to provide concrete examples with explicit instructions, it is necessary to do so in the context of a particular hardware and software environment. Following is a list of the environment used to develop the example programs in this document:

1. USB Host System --

A standard PC running Windows 98 Gold (4.10.1998) and/or NT5. The extra Host software consists of RW2.Exe and UsbSmsc.Sys, both supplied on the companion code disk. Since this system will often be used for the purpose of Driver and/or Application software testing, it is the author's preference to treat this as a test machine -- it is assumed that it could crash at a moment's notice, losing everything on its hard disk and disrupting any LAN that it might be connected to. As a result, no development work should be done on it, no important data should be stored on it, and it should not be connected to any network. In addition, nothing but Host test software should be executed on it in order to avoid corrupting any test results due to interactions with other hardware and/or software, except as an explicit part of the testing.

2. MCU Development System --

A standard PC running Windows 95 Gold (4.00.950), or any other OS capable of hosting all of the development tools used. This system hosts the MCU Compiler, USB Protocol Analyzer, and ROM Emulator, each of which is described below. Ideally, everything on this system should be full production quality, with no Beta or pre-Release anything. Connection to a network for backup and/or printing services is encouraged. Any Host software development (either drivers and/or applications) can be performed on this machine, provided that all tools used are of suitable production status. Of course, any such Host software should never be executed on this [development] system.

Because one of the particular Analyzers and the Emulator used each requires a parallel port, a board containing a second parallel port (set as ECP with Legacy LPT2 assignments) was added to the system. The Emulator can also make use of a serial port for various purposes, and it was connected to COM1, but this connection is not used in the context of this document.

Since a number of the programs make use of the COM2 port on the 67x SIO device on the EVB97C100 for console I/O, a null modem was used to connect the COM2 port on the EVB to the COM2 port on the MCU Development System. Hyperterminal was used to establish the communications link, with settings of 115.2 Kbaud, 8 data, 1 stop, no parity, no flow control, TTY emulation. Because no hardware flow control is used, the null modem can be of the simple 3-wire type, with the TxD and RxD crossed, and the grounds connected together. For this to function properly, the jumpers on the EVB must be set to connect the COM2 transceivers to the SIO, rather than the USB97C100, device. For Assy 6075 Rev. B, JP7 and JP8 must have jumpers between pins 2 and 3. For EVB's without a COM2 port (e.g., EVB6104), use the COM1 port instead.

3. MCU Development Tools --

The Keil 8051 C Compiler V5.0 (www.keil.com). For installation, just use the default settings.

4. USB Protocol Analyzer --

When executing the programs in this document, access to an Analyzer is not required, since the software has already been developed and tested. However, when developing new software, at least part time access to an Analyzer is a practical necessity, at least in the author's opinion. An external trigger input is desirable in order to permit triggering the Analyzer from software controlled pulses on either the Host or the Target system. A trigger output is desirable to permit triggering an oscilloscope or Logic Analyzer.

Representative manufacturers of Analyzers include Genoa Technology [www.gentech.com] and CATC [www.catc.com]. When using the CATC USB Inspector, the companion software searches the LPT ports for the Analyzer, so it is best to install the device on LPT1 so that the software does not touch the Emulator while searching for the Analyzer.

5. ROM Emulator --

The TechTools UniROM UR08-1M-90 ROM Emulator (www.tech-tools.com). This device was connected to LPT2 during the code development for this document.

Some target hardware has only a surface mount PLCC socket for connecting an EPROM/FLASH device or Emulator, rather than a more rugged 32 pin ZIF DIP socket. These (PLCC) sockets can be somewhat fragile so, if it is intended to use such hardware for serious software development, it is suggested that a more rugged connection be established. For early EVB97C100s (ASSY 6075 Rev. B), this can be achieved by replacing the JP9 (Ext. Bus) connector with a wire-wrap header, to which a small wire wrap board can be soldered with a suitable socket, connector (34 pin 2-row header for the Emulator used here), etc. with which to interface to the ROM Emulator. This new connector can then be wire wrapped (NOTE: the pinout for EPROM emulation is different from FLASH emulation!) to the J9 replacement connector, and a short (e.g., 4") ribbon cable assembly can then be used to connect to the Emulator.

If the ROM Emulator has a suitable Reset output with which to drive the Target, it is most convenient to connect it. Otherwise, it will be necessary to hold the Target reset (using the manual push button on the EVB97C100) during firmware downloads. For the Emulator used here, the Reset output if fully programmable, with a setting of LOW TRISTATE being correct for interfacing with the EVB97C100 reset circuitry. The EVB (ASSY 6075 Rev. B) connection is at JP25 pin 2, and the Emulator connection is at the Feature Connector pin 5; micro-hook cables suitable for making this connection are supplied with the Emulator.

A workable alternative to a ROM Emulator is a Monitor ROM, but that is not the approach used in this document for a variety of reasons: Monitor ROMs are extremely compiler-specific, require that an external RAM device be available, and often consume hardware resources beyond just their memory footprint (e.g., a Bank Select register when executing the application from RAM, a COM port for download, etc.). As a result of these characteristics, it is often not possible to use a Monitor ROM on final target hardware because the necessary resources might not be present. In addition, the target hardware might not contain even a single COM port for a debugging console, much less an additional one for code download. For situations like this, the ROM Emulator used here contains a COM port that the firmware can use as a debugging console even in situations where the target hardware has no COM port of its own. Also, since ROM Emulators usually interface to the development system using a parallel port, rather than a COM port, code downloads are much faster than with a ROM Monitor; this is especially true if the MCU's serial port is used instead of a 550A type.

6. Oscilloscope or Logic Analyzer --

An HP54645D Mixed Signal Oscillosope (http://www.hp.com).

When executing the programs in this document, access to a scope is not required, since the software has already been developed and tested. However, when developing new software, at least part time access to a scope is a practical necessity, at least in the author's opinion.

Key features to look for are digital storage, numerous channels (8 or more is desirable), and deep memory (at least 100 K points/channel is desirable). Assuming that the scope is only being used for software development, it is not necessary to have analog channels or to sample at high speeds (10 Msps is about enough). It is also not necessary to have the elaborate trigger capabilities that are standard in Logic Analyzers these days, since it is a simple matter to have the software pulse a GPIO pin and provide a direct trigger when the desired event occurs, etc.

Executing and/or re-Building the Examples

The following procedures are based on the use of the hardware and software just described. If different equipment is used, then other means will need to be used to achieve a port, which is beyond the scope of this document.

The Code Disk contains project files, listings, and final HEX files (in Intel Hex Format) for all of the example programs for both the EVB6075 and EVB6104. As a result, building the programs is not necessary in order to download and execute them.

In order to download the HEX files to the ROM Emulator, a set of **CHPTxx.BAT** files is provided. Each of these BAT files in turn executes the **Download.Bat** file, which produces a **Download.Cfg** file, which is finally passed to the **UrLoad.Exe** program to do the actual download to the ROM Emulator. The reason for all of this indirection is to ease the task of porting to different Host/Target/Emulator environments.

The **DownLoad.Bat** file should be modified in order to change:

1. The Host drive/path/name of the download program (e.g., c:\unirom\UrLoad.Exe)
2. The Host port to which the Emulator is attached (e.g., LPT2)
3. The Target memory type, size and base address being emulated (e.g., 128K Flash @ 0)
4. The Target reset circuit type (e.g., LOW TRISTATE)

Of course, if a different ROM Emulator, or a Monitor ROM, is used then other means will need to be used to achieve a port, which is beyond the scope of this document.

Rebuilding the example programs can be done from the Keil compiler by opening the desired project (e.g., **ChptXX.PRJ**, from the Project--OpenProject dialog), and building it (from the Project -- Make: Build Project dialog). Of course, if a different compiler, etc. is used, then other means will need to be used to achieve a port, which is beyond the scope of this document.

If it is desired to make new projects using the Keil compiler, then it is important to adjust several settings:

1. Set the compiler to produce an assembly listing (if desired) (using Options -- C51 Compiler... -- Listing -- Include Assembly Code checkbox).

   Define the symbols: **,DBG** (using the Options -- C51 Compiler... -- Misc. -- Symbols for

3. Set the internal RAM size to 256 bytes (using Options -- BL51 Code Banking Linker ... -- Size/Location).

4.

It is also possible to compile the example programs in Microsoft Visual C/C++, although the resulting binary cannot be executed. Version 4.0 was used in the development of the code examples. Even

portability of the code, and in situations where the developer is more comfortable with that environment. The companion code disk contains a single MSVC project file that contains subprojects for each of the example programs in this document. Since MSVC embeds absolute paths in the project

1 and build from there. Under this directory

; this is where the

binaries for each subproject are placed.

If it is desired to make a new project in MSVC, do the following:

1. Create a new project using File -- New -- Project Workspace -- Console Application, and give it a name (e.g., ChptXX).
2. Use Insert -- Files into Project... to add the desired source (*.C) files.
3. Define **_MSVC_,DBG** using Build -- Settings -- C/C++ -- Category General -- Preprocessor Definitions. Optionally, set the Warning Level to 4 (maximum) on the same sheet.
4. Optionally, on the C/C++ Customize sheet, select the Disable Language Extensions checkbox in order to obtain the best assurance that the source code is generic ANSI C.

Note that because of the way the Keil compiler handles SFRs and SBITs, it was necessary to add some special code to **USB97C100.H** for non-Keil compilers like MSVC. This code only defines the subset of the SFRs and SBITs used by the example programs, so it might be necessary to add more if any new code makes use of additional SFRs and/or SBITs.

The Example Programs

Chpt2.Hex:
Described earlier, provides an example of different ways of implementing a software queue. Unlike the other examples, this code is not meant to be executed, but the ASM listing should be inspected to see and understand the impact of address spaces and coding style on performance.

Chpt3.Hex:
This program illustrates basic programming techniques for the USB97C100 device and EVB. Topics covered are initialization, RAM access and testing (to qualify the hardware setup), DBGPRINT(), console I/O, use of GPIOs, software generated time delays, etc.

Chpt5.Hex:
This program illustrates a variety of DMA programming techniques using the LPT port as an example.

Chpt6.Hex:
This program illustrates an actual (albeit simple) USB application that is fully Chapter 9 compliant. While the primary emphasis is on coding techniques, the code is intentionally structured in a way that completely separates the core USB software from the actual application, with the intent that the application code can be easily replaced while leaving the core USB code intact.

Coding Style:

Significant effort was expended to make the code examples as fully ANSI C as possible. All non-ANSI typedef's, etc. are declared in Types.H (with the exception of SFRs and SBITs, as was already discussed). There are a few places where some pragma's were needed in the code, but these are only used where needed. Obviously, no "//" in-line comments are used.

All variable names begin with a lower-case letter, while all function names begin with an upper case letter.

All function names begin with an upper case letter and contain no embedded underscrores (e.g., MyFunction(a,b)). Macro functions that are called in the same way as an actual function use the same naming convention. However, macro's that use a calling convention that is different from what an equivalent function would use (e.g., if a variable would be passed by reference to an actual function, but the variable name is passed to the macro) are named with all upper case letters and underscores separating words. In order to avoid confusion with register definitions, parentheses () are ALWAYS used with macro's, even when no arguments are passed (e.g., MY_MACRO(a,b), YOUR_MACRO()).

All variable names begin with a lower case letter, and type BYTE is assumed, unless one of the following prefixes is encountered:

bFOO -- bit
cFOO -- code const

wzFOO -- unicode-Z string
pFOO -- pointer

discerned by context. No attempt was made to prefix variables more fully (e.g., with their memory space, etc.) for fear that having too long a prefix on every variable would do more harm than good.

;

they can be differentiated from macro's because no parentheses are used (e.g., BUS_REQ). Bit fields

which it corresponds (e.g., BUS_REQ_HREQ_ is the HREQ bit field in the BUS_REQ register).

Manifest constants (i.e., #define's) use all upper case letter and underscores as well, but are prefaced with a lower case c (e.g., cUSB_EP0_MAX_SIZE) so they can be differentiated from register names.

A number of files are common to many of the example programs, and will likely be useful in other applications:

contains definitions for all non-ANSI data types.
USB97C100.H
USB97C100.C contains some useful misc. device-specific functions (e.g., HwReset).
configures the SIO device and provides low-level console I/O functions.
Debug.C/H
Usb.H provides definitions for USB related data types, etc.
provides core code for handling connection and tranfers on the USB.

CHAPTER 3 - FIRMWARE BASICS

All of the examples in this document make use of the **Types.H** file, which encapsulates all of the non-ANSI data type declarations, as well as defining a number of useful macro's, etc. All of the executable examples (i.e., everything except **Chpt2.Hex**) also make use of the **Usb97100.H** file, which contains defines for every register and bit field in the USB97C100 device; all references to data types, registers and bits make use of the nomenclature in these two files. Experienced C programmers will recognize the use of the "volatile" keyword, which must be used with all memory-mapped I/O devices in order to prevent the compiler's optimizer from removing [what it thinks are] redundant memory accesses.

Several additional files are used in every example in this document, and every function contained in them is described in this section:

Debug.C/H -- provides debug console I/O
SIO.C/H -- provides code to initialize the SIO device and operate the COM port
Usb97100.C -- provides initialization code, etc.

Initialization

At the end of POR (Power-On Reset), each core in the USB97C100 device is initialized, every register (except for the DMAC) is filled with defined values, and the MCU begins executing code at address zero using the Ring Oscillator.

Each of the example programs calls the **Usb97C100HwInit()** function in **Usb97100.C** early in the execution of **main()** in order to complete the reset process. Since many of the values used by this function are application dependent, the code makes use of several global variables in code space, whose values are set by the application, in order to determine which values to place in the various device registers.

Even though this function configures the SIE and resets the MMU, it is important that it does *NOT* enable any USB Endpoints, even EP0, since this must wait until a USB Reset is received. This, and many other USB related issues, is discussed in detail in a later chapter.

Since the DMAC registers do not have defined values after POR, the function **DmacReset()** in **USb97100.C**, which is called by **Usb97C100HwInit()**, places valid entries in every DMAC register.

CLOCK_SEL Register

One of the first things to do is to select the desired clocks for the MCU and the DMAC and to disable the Ring Oscillator. Note that it is important to select the new MCU clock source **BEFORE** disabling the Ring Oscillator; doing otherwise can shut off the clock to the MCU which would halt execution.

Usually it is desired to use the highest speed clocks for the MCU and the DMAC in order to obtain best performance, but there are situations in which this is not the case. One example is any application in which power consumption is more important than performance. Since power consumption increases approximately linearly with clock frequency for CMOS devices like the USB97C100, lower clock frequencies translate directly into lower power consumption. Another consideration is the speed capabilities of the various hardware devices connected to the USB97C100, some of which might not be capable of operation with the fastest clocks. For the MCU, it is possible to slow the clock frequency before accessing a slow external peripheral device and then to restore the higher speed clock after the access. Since the DMAC operates asynchronously with respect to the software, the same technique cannot be used, and the DMAC should usually be set at a clock frequency, based on the capabilities of the slowest attached device, that remains constant.

UTIL_CONFIG Register

The four lowest GPIO pins are multi-function, and this register provides control of the function of each of the individual pins. Although this register can be manipulated dynamically at run-time, it is most common to "set it and forget it" at initialization time.

In applications that use the MCU's internal Serial Port, this register is used to enable the Tx and Rx signals onto the GPIO0 and GPIO1 pins. This Serial Port is not nearly as capable as the 550A UARTs found in contemporary SIO devices, but it can be useful in some applications.

This register also permits input trigger selection for the MCU timers either from the USB SOF or from a GPIO pin. The ability to hardware trigger a timer from the SOF has a variety of uses, including (i) detection of a USB suspend, (ii) detection and reconstruction of missing SOFs, and (iii) intra-frame time measurement for isochronous rate feedback. The ability to trigger a timer from a GPIO pin, combined with the fact that the timers can create interrupts, can be used to provide additional GPIRQs. However, the standard GPIRQs are more than sufficient for most intended applications of the device.

Since the desired setting of this register is application dependent, the sample code again makes use of a global BYTE in code space whose value is defined in the application code.

GPIOA_DIR Register

All GPIO pins are bi-directional, and this register controls the direction of each individual pin. Note that a pin's direction must be explicitly set in this register, regardless of the pin's function having been set in the **UTIL_CONFIG** register. For example, setting the GPIO1/TXD pin function to TXD in the **UTIL_CONFIG** register does *NOT* automatically make the pin direction an output pin; the corresponding bit in **GPIOA_DIR** register must be set in order to do this. Note that at POR, all pins are defined as inputs, so it is necessary to use a resistor (or other mechanism) in order to pull each signal high or low if a defined logic level is needed prior to the time when the software can set the appropriate registers.

GPIOA_OUT Register

This register defines the logic level applied to GPIO pins whose direction is set as output. Note that if this register is manipulated in an ISR(s) in addition to the foreground, then the foreground code should disable interrupts before access to this register and then restore the IRQ enable when finished; see **Chapt3.C** for an example of this.

MEM_BANK Register

This register controls a movable 16 KB window that appears identically in both the CODE and XDATA address spaces (i.e., Von Neuman model) of the MCU at address 0xC000. The memory accesses are to the FLASH Bus, and this register controls the high 6 bits of the 20 bit physical address. Among other things, this register can be used for bank-selecting code pages, which is described later in this section.

For applications that need to store more data than will fit in the internal RAM, this address space can be used to access external RAM. ROM Monitors often require this (Von Neuman) style of memory as well. On the EVB97C100, there is a 128 KB RAM located at physical address 0x40000 (i.e., 256K above address zero); to map the start of this RAM, fill the **MEM_BANK** register with 0x10 (the high 6 bits of the address, right-aligned in this register), which is what **Chpt3.Hex** does.

There are a number of considerations if this address space is used for RAM that contains variables declared as BYTE_X, etc. First of all, the compiler's start up code will execute before the application has an opportunity to set the **MEM_BANK** register contents. This means that either any variables allocated there must not be initialized by the startup code, or that the start up code must be modified to set this register before doing the initialization. Of course, the Linker also needs to be informed of the location of the 0xC000 window in order to generate the proper code.

Do not be misled by the name "FLASH Bus" since nothing about this bus restricts its usage to only Flash Memory devices. With its 8-bit Data, 20-bit Address and ISA-style nRD and nWR signals, it can be used for any combination of RAM, ROM, EPROM, EEPROM, FLASH, and peripheral I/O devices. The two primary differences between this bus and the ISA bus are (1) the ISA bus has DMA, while this bus does not, and (2) the ISA bus must be shared with the DMAC while this bus is owned full time by the MCU. As a result of this last fact, it is preferable to interace all non-DMA peripherals to this bus.

In addition to the movable window described above, there is also a single fixed 16 KB window mapped to address zero both on the Flash Bus and in the MCU code address space. For applications that can fit in 16 KB of code space, this window is sufficient to contain the program code and the movable window can be used for external data access as was previously described. For applications that require more code space, the movable window is also mapped into 0x4000-0x7FFF in the MCU code space, and can be used for program storage. If the MEM_BANK register is set to 0x01 (the POR default value), then this second window will map straight through to the Flash Bus, which provides a contiguous 32 KB code space starting at address zero on the Flash Bus; this is what Chpt6.Hex does (V1.1 and later). Code overlays can be supported by treating the low 16 KB window as the "root" segment, placing the overlaid code in the upper 16 KB movable window, and using the MEM_BANK register to select the desired overlay. For applications that require both large code space (with or without overlays) as well as external RAM or I/O access, a simple decoder (e.g., a PAL, CPLD, FPGA, etc.) can be placed on the Flash Bus that maps the top (e.g., 1 KB) portion of each 16 KB page (except probably for the bottom page) to the RAM or I/O device(s), with the lower portion (e.g., 15 KB) of each page selecting the desired ROM page.

IOBASE Register

This register controls the mapping of a 256 byte movable window into the ISA I/O space that appears at XBYTE[0x4000]; the high 8 bits of the 16-bit ISA I/O address are contained in the **IOBASE** register. Since it is common to have peripheral devices spread throughout the ISA address space, it is common for this register to be re-written extensively at run-time.

It is common for foreground code to perform this type of manipulation, so if this register is manipulated in an ISR as well, it is important that it be saved and restored in the ISR in order to avoid disrupting the foreground code.

MEMBASE Register

This register performs a similar function to the **IOBASE** register, except that it controls a 4 KB movable window into the ISA memory address space that appears at XBYTE[0x5000]; the high 8 bits of the 20-bit ISA memory address are contained in the **MEMBASE** register. All of the comments about the **IOBASE** register above apply to this register as well.

Shared Bus Architecture Basics:
Note that the ISA bus is shared between the MCU and the DMAC, so the MCU must acquire ownership of the ISA bus before it can access it. A variety of issues related to this is discussed in detail in a later chapter. For the moment, it is sufficient to mention that the functions **IsaAcquire()** and **IsaRelease()** in **Usb97100.H** provide the mechanism by which the MCU can acquire ISA bus ownership and release it back to the DMAC respectively. Since ownership of the ISA bus by the MCU suspends all DMA traffic, it is usually desirable to disable interrupts for the duration of the ownership in order to prevent DMA suspension for an extended period of time while the ISR executes; the functions **InterruptDisable()** and **InterruptEnable()**, also in **Usb97100.H**, provide the means for doing this. Note that, like any 8051 derrivative, the **EA** bit provides a global interrupt enable (when TRUE), and it is usually desirable to save/restore this bit when manipulating it.

()

This function, which is contained in , initializes the FDC37C672 SIO device on the EVB. Since the examples in this document only make use of a single COM port and the LPT port, these are the only For other applications, the device contains an additional COM port (with a full set of handshake lines on the EVB), an FDC (Floppy Disk Controller), P/S Keyboard/Mouse IrCC (Communications Controller) that supports IrDA InfraRed), as well as a variety of Consumer IR For detailed information about the SIO device, the interested reader is referred to the device Data Sheet.

BIOS. As a result of this, all hardware resource assignments are fully configurable and each device can be disabled as well. The **()** function simply configures the COM2 and LPT devices to use the **SIO.H**. then defines various registers and bit fields based on these values. It is instructive to look carefully at the manner in which SIO.H performs the register xdata address is calculated at compile time, and not at run time, which results in optimum performance source code.

Note that this function saves and restores the state of the bit and disables interrupts before acquiring the ISA bus in order to access the SIO. On return, the and registers are restored to their previous settings, along with the bit. Although this register save/restore is not necessary for a

basic code sequence will be seen often in the examples.

Since the examples use the COM2 port for debug console I/O, **()** calls the function to initialize the COM port before it returns. The COM port is set for 115.2 no parity. Note that the COM2 port on the EVB has no handshake lines, so there can be no hardware flow control.

Some EVB's produced after the EVB's, a new file (EVB.H) was added to the projects. Among other things, this file conditionally defines the symbol EVB_COM2 for port, and does not define the symbol for EVB's that do not have a COM2 port. () is conditionally compiled, based on this symbol, to use the COM1 port for However, all of the functions that use the COMx port (e.g., Com2GetByte(), and Com2IsRxRdy()) still retain the COM2 name, even if it is actually COM1 on a particular EVB.

()

One lines. This function, which is contained in **Debug.C** in this document, the value displayed at reset is "Cx", where "x" is the chapter number of the the early stages of debugging if the ; by placing calls to this function at various points in the code, it becomes possible to determine what section of code executed last before the relatively short, on the order of about 10 microseconds, so even relatively frequent calls will not have a significant impact on execution timing.

EA, and **IOBASE** **SioInit()** did but, in , then it would not be possible to call it from within an arbitrary code block, since that code block might be

FatalError()

Also in , this function is very similar to **CheckPoint()** anything since it hangs in an infinite loop with IRQ's disabled after it displays the value. Note that **<u>there</u>**

should be no use for this function in an actual embedded application, but it can be useful during initial code debugging.

DBGPRINT()

This is a macro function defined in **Debug.H** that is conditionally compiled based on a "**DBG**" command-line switch to the compiler. When this switch is defined, the function simply passes its argument to **printf()** in the RTL (Run-Time Library); when this switch is not defined, **DBGPRINT()** compiles to nothing. This permits keeping extensive debugging information in the source code, while being able to make a compact production release by simply undefining **DBG**. It also permits disabling **DBGPRINT** in various sections of the source code by using preprocessor directives (#ifndef DBG / #define DBG, #ifdef DBG/ #undef DBG).

There is also a **DBGTRACE** macro function that behaves in a similar way, except that it is also gated by **TRACE_ON**. Changing the value of this symbol in the source code permits disabling TRACE messages while leaving PRINT messages enabled. One use for this capability is to use TRACE messages to show the code execution sequence for "normal" conditions, and to use PRINT messages to display any error conditions. Once the code seems to be functional, the TRACE messages can be disabled, resulting in smaller code size and MUCH better performance (since printing through a COM port is really S--L--O--W), but any errors are still displayed. If the cause of an error is not obvious, then the TRACE messages can be enabled again simply by uncommenting a single line in **Debug.H**. The author found all of this very useful when developing the example programs.

printf()

This RTL function eventually calls the **putchar()** function, which appears in **Debug.C**. In the example code, this function just redirects to **Com2SendByte()** from **Sio.C**. For other target hardware platforms this code could instead make use of the MCU Serial Port, a COM port contained in a ROM Emulator, or any other output device that happens to be available.

Unfortunately, this function is usually not reentrant, even in RTL's intended for embedded applications, so none of this can be used in an ISR. However, the execution time of a printf(), especially through a COM port, is so long that you would not want to do it in an ISR anyway. If it is necessary to display information from an ISR, it is suggested that a circular buffer be used in which the ISR places very short messages, perhaps a single DWORD or so, and the foreground polling loop can then send them to the terminal. By keeping ISR's as short and as simple as possible, it should not usually be necessary to do this, so none of the sample code illustrates the technique; as the text books say, "It is left as an exercise for the interested reader.".

Com2SendByte()

This function, from **Sio.C**, does the usual dance with **EA**, **BUS_REQ** and **IOBASE**. The hardware resource assignments, register access, etc. are all from **Sio.H**.

The one interesting twist is that, while it is waiting for the UART to be ready for Tx, it restores the registers and the **EA** bit in order to permit DMA and IRQs to proceed. This is critical if **DBGPRINT**'s are to be used successfully in an actual USB application, since not doing so would suspend IRQs and DMA for up to 100 uS (@ 115.2 Kbps, and much longer at slower speeds) every time this function was called, which would not be a good thing.

kbhit()

This function, from **Debug.C**, simply redirects to **Com2IsRxRdy()** in **Sio.C**, similar to the arrangement with **putchar()** above. Once again, redirection could be to any other available input device for other target hardware platforms. **Com2IsRxRdy()**, as usual, gets its hardware assignments from **Sio.H**, does the usual **EA**, etc. dance as before, and simply returns the appropriate bit from the UART's **LSR**. Note that this function is fairly fast since it never has to wait for anything, unlike **Com2SendByte()**.

_getkey()

Debug.C, simply redirects to a function in **Sio.C** **Com2GetByte()**. The

HINT: Note that this function will wait for a keystroke, so it can be useful to check first before calling this function in order to avoid waiting for an indefinite period of time.

From **Usb97100.C** this function implements an accurate software time delay based on an MCU clock of 24 MHZ. Note the function does not change the IRQ enable, so it is up to the caller to do so if that is what is really desired
very special circumstances (e.g., doing precise timing interfaces to a hardware device) it has its uses.

Code Example: Chpt3.Hex

performs a RAM test on both the ISA and FLASH busses, and pulses a GPIO pin in an infinite loop.

In order to execute this program, the COM2 jumpers on the EVB must be set to select the SIO, as

between pins 2 and 3. See Chapter 2 for a general description of the equipment setup.

The use of "goto" statements in C code is something of a religious issue. Many C purists feel that it was

setjmp/longjmp; go figure... It is the author's style to use goto's in lieu of C++ Structured Exception Handling (e.g., try throw/catch or try/ finally), especially in situations in which there is

entire sample code for this document in which goto's are used; readers who are offended by this style

The **RamTest()**
traveling-zero's/traveling-one's algorithm. Note that it does the dance with **EA** **BUS_REQ** even
; the same can be said for disabling/enabling IRQs around the GPIO pulsing in the final loop. However, that exact

accessing the same **GPIOA_OUT** It was the author's judgement that the reader would be better served by illustrating these techniques right from the beginning, rather than showing coding techniques at

a real application.

Just as some EVB's do not have a COM2 port (as was described previously in the
some EVB's do not have a RAM on the Flash Bus. The symbol EVB_FLASH_RAM is defined (in Evb.h) for EVB's that do have a RAM, and the code is conditionally compiled, based on this symbol, in order to

The reader is **<u>strongly encouraged</u>**
In particular, failures on either of the RAM tests can be caused by an improper interface to the ROM Emulator, as was described in Chapter 2.

CHAPTER 4 - THE MMU

The MMU (Memory Management Unit) is responsible for managing the 4 KB Data Buffer RAM that is used for all USB communications. The memory is organized as 32 pages of 128 bytes each. Since each USB packet has an 8 byte packet header associated with it, this means that a maximum size (64 bytes) USB BULK packet can fit in a single MMU page. Larger packets are handled by the MMU concatenating multiple (up to 10) pages together. Even when packets are physically comprised of multiple pages, the MMU creates the illusion of a single large virtually contiguous packet for all 3 ports: the SIE (interface to USB), the MCU (for PIO), and the DMAC.

The MMU in this device is extremely similar to the MMU in the SMSC LAN91C94/5/6 devices, and the interested reader is referred to those data sheets for a more complete discussion of the architure; the focus of this chapter is on the register-level programming instead.

Allocating and Freeing Packet Memories with the MCU

The MCU can instruct the MMU to allocate a packet of a desired number of pages, or to free a specific packet, by using the **MMUCR** (MMU Command) register. The MCU can tell when an allocation has completed by checking the **ARR** (Allocation Result) register. Assuming that the MMU has enough free memory pages to satisfy the request, the allocation is very fast (a couple of microseconds or so), so the firmware should just wait for the allocation to complete (as opposed to leaving the procedure and doing other work).

The following block of code will allocate a packet:

```
MMUCR = (MMUCR_ALLOCATE_ | (numPages-1));
while (ARR & ARR_FAILED_); /* wait for MMU to finish */
pkt = ARR & PN_MASK_;        /* save result */
```

The following block of code will free a packet:

```
PNR = pkt;
MMUCR = MMUCR_RELEASE_;
```

Note that the packet number, saved in the variable "pkt" in the above code examples, is analogous to a handle -- it tells the MMU which packet is being referred to, but has no particular relationship to any physical memory address, etc. The details of what to do with these packet numbers is discussed in the following sections.

The **MakePkt()** function in **Chpt5.c** shows an example of allocating a packet, and also of filling it with data, which is discussed in the USB Transmission section below. The **FreePkt()** function in **Chpt5.C** shows how to free a packet using these methods.

USB Reception

As each Rx packet arrives from the USB and the MMU allocates a packet memory for it, it pushes the packet number onto a queue that the MCU can access by reading the **RXFIFO** register. The MCU can then inspect the packet header (the first 8 bytes of each packet) in order to decide what to do with it, based on things like the Data Toggle PID, Endpoint Address, etc. The MCU then has a choice of dropping the packet and freeing the memory, or of just removing the packet from the **RXFIFO** queue but leaving the memory allocated.

Normally, handling USB Rx packets is done in Isr0(), which is the ISR that handles IRQ0 interrupts; an example of unmasking this interrupt is shown in **main()** in **Chpt6.C**. To do this, the corresponding bit in the **IMR_0** register must be cleared, and IRQ0 must be enabled in the MCU's **EX0** SBIT:

```
EX0 = 1
IMR_0 = ~INT0_RX_PKT_;
InterruptEnable();
```

Having done this, the ISR for IRQ0 will be executed each time a USB Rx packet arrives. In the ISR, the MCU can tell if the IRQ is from a USB Rx by inspecting the bit in the **RXFIFO** **RXFIFO**, the MCU can access the packet at **RXFIFO** by doing the following:

```
                  /* packet header begins at offset zero */
PRH = PRH_RCV_ | PRH_READ_ | PRH_AUTO_INCR_;


/* TODO:  read any other bytes of interest */
```

Setting the field in the **PRH** **RXFIFO**, rather than the one in the , for subsequent transfers through the **MMU_DATA** The other fields in the **PRH** packet, and to increment addresses after each access. If it is desired to read from a packet that is not at the head of the , this can be done by placing the packet number in the **PNR** **PRH** for . For example:

```
PNR =
bOldEA = EA;                    ion in Reference Guide
InterruptDisable();
                  /* byte count is at offset 6 */
PRH = PRH_PNR_ | PRH_READ_ | PRH_AUTO_INCR_;
      bOldEA;
cntLo = MMU_DATA;
```

Note that, in the code samples above, the PRL register is ALWAYS written before the PRH register; this requirement and is not optional. As a result of this fact, that interrupts be disabled in any foreground thread that uses these registers if they are also shared by an ISR. Failing to

foreground has written the PRH register, which violates the device requirement for accessing these registers. in the example (see Chpt5.C and UsbCore.C)

A careful reading of the USB97C100 Data Sheet reveals that the MMU can take up to 1.218 uS after the **PNR** setting **PRH** **MMU_DATA**, but this is not the case. The reason for this is the way the

DPTR register, and that the **PRH** **MMU_DATA** are at different addresses in the XDATA address **MMU_DATA** register after **PRH** register is as follows:

```
        DPTR, #6000H
MOV
```

Each of these is a 2-cycle instruction, with the actual XDATA access happening in the 2nd cycle of the ,@DPTR instruction. As a result, an absolute minimum of 3 instruction cycles is guaranteed, which is more than the time that the MMU needs.

MMU_DATA register requires a MOV ,@DPTR instruction, which takes 2 instruction cycles, this access time is satisfied, even when the MCU operates at its maximum clock frequency of 24 MHz, so no additional

Getting back to the received packet, Byte0 in the packet header is the most interesting when handling a USB Rx packet. First of all, there is the **PKT_HDR_BAD_CRC_** bit that will be set if the packet arrived with a bad CRC; such packets should always be dropped:

```
        if (pktHdr0 & PKT_HDR_BAD_CRC_) {
                MMUCR = MMUCR_REMOVE_RELEASE_;
                goto _next_pkt;
        }
```

Note that the **REMOVE_RELEASE_** command applies to the packet at the head of the **RXFIFO**, and not to the packet number in the **PNR**; the command both removes the packet from the **RXFIFO** and releases the memory that had been backing the packet number.

Since in practice the USB has a very low BER, it should not be often that a packet will arrive with a bad CRC, but properly written software will check for it nonetheless. The fact that the occasional bad CRC packet is received is useful for ISO applications in which it is necessary to know in which specific USB frame the packet arrived, even though the data is unusable. As an aside, the packet header also contains the USB frame number in which the packet arrived, which is also useful in ISO applications.

The pktHdr0 byte also contains the EP address in the low nibble:

```
        ep = pktHdr0 & EP_MASK_;
```

The target EP is significant in order to determine how to handle the reception. For example, the Data Toggle PID should be ignored for ISO, but it must be checked for BULK packets, and Control Write packets on EP0, in order to detect duplicate packets which should be dropped. The pktHdr0 byte contains 2 bits for this purpose: **PKT_HDR_LAST_TOG_** contains the Data Toggle value of the previous packet received on the same RxEP, while the **PKT_HDR_SAME_TOG_** bit indicates if the current packet's Data Toggle is the same as the previous packet. One might assume that simply checking the **PKT_HDR_SAME_TOG_** bit would be sufficient for BULK packets, but it is not because a Control Transfer (e.g., CFES, etc.) might have executed between the receptions. The solution is to maintain a bit variable in software for each RxEP that keeps track of the expected Data Toggle. As each packet arrives, the Data Toggle for that packet is calculated and compared with the expected value; packets with the incorrect Data Toggle value are discarded.

```
switch (ep) {
case 1:
        if (pktHdr0 & PKT_HDR_LAST_TOG_)/* expect opposite toggle from last time */
                bThisTog = 0;
        else
                bThisTog = 1;
        if (pktHdr0 & PKT_HDR_SAME_TOG_)          /* unless it turns out to be the same */
                bThisTog = ~bThisTog;

        if (bThisTog != bEp1RxToggle) {        /* mismatch, so drop pkt as a duplicate */
                MMUCR = MMUCR_REMOVE_RELEASE_;
                goto _next_pkt;
        }

        bEp1RxToggle = ~Ep1RxToggle;        /* invert toggle for next time */

        /* TODO:  whatever you do with packets for this EP */
        MMUCR = MMUCR_REMOVE_;            /* remove from RXFIFO, but keep memory */

        break;

case xxx:
```

The situation for duplicate packets is anlogous to that for bad CRC -- it should not be expected to happen often, but it can happen, and properly written software will handle it as described.

As can be seen from the above code samples, it is necessary to use at least the and **PRL** and sometimes also the **PNR** these registers are also used by the foreground whenever it has to access a packet memory. Assuming that the receptions are handled in an ISR, it is necessary for the firmware to save these registers on entry and to It is also necessary for the foreground thread to disable interrupts while accessing the PRL/PRH register pair, as was previously described.

```
{
if (!(RXFIFO & RXFIFO_EMPTY_)) {

        oldPNR = PNR;
        oldPRL = PRL;


        do {
                /* TODO:  Handle all receive packets */


        /* Restore MMU Registers */

        PRL = oldPRL;
        PRH = oldPRH;
```

A careful reading of the USB97C100 Data Sheet reveals that the **PNR PRH**, and registers should not be modified for at least 2.5 **MMU_DATA** when writing to a packet,

MMU registers takes much more than 2.5
any extra delay.

One last detail of USB Rx in an ISR concerns the register, which clears on read. One way to avoid missing **ISR_0** register in ; a copy of the register value can be passed to any IRQ handlers if needed. For example:

```
{
        BYTE isr0;
                      ();
        while (       ISR_0)    ~IMR_0)
                       (isr0
        InterruptEnable();
```

All of these techniques can be seen in combination in the **Isr0()** **UsbCore.C**, and in the function in **Chpt6.C**

An alternative strategy for ISO applicatons is to use the IRQ instead of the **INT0_RX_PKT_** ; doing so will cause the ISR to execute exactly once for each USB frame, which is a desirable characteristic in some ISO applications. In addition, if the code is carefully structured, it might be

with minimal branching and branch balancing, then the execution of every part of the ISR will exhibit low jitter from one USB frame to the next, which is a desirable characteristic in most ISO applicatons.

It is the author's preference to do it in the ISR so that the corresponding packet memories can be freed as quickly as possible, and it also keeps all of this code in a single place, instead of distributing it among transfers like CFES. However, either approach is valid, provided that the checks are performed someplace in the code.

In applications that use 64-byte BULK packets, a fast Host can deliver a packet roughly every 50 uS. In most cases, it is desirable to have an **RXFIFO** loop that is at least as fast as the packet arrival time in order to avoid an overrun situation. This suggests that the minimum necessary processing should occur in the **RXFIFO** loop, and the code should be written for the best possible execution speed. This in turn affects the style with which the code should be written, as was previously discussed in Chapter 2. It is this concern for speed that makes it reasonable to even consider deferring the CRC and Data Toggle validation.

Regardless of how the receive handling code is designed, it is essential that it be implemented in such a way that the **RXFIFO** never overflows and the MMU never runs out of free pages. The reason for this is that it is a requirement that every USB device must always be capable of receiving a SETUP packet on EP0; if either of the above conditions occurs, the device will not be capable of receiving anything, including a Setup packet, rendering it out of compliance with the USB Specification. The details of both Rx and Tx Memory Management Policy (MMP) code are discussed in later sections of this chapter.

USB Transmission

In order for the MCU to request that the MMU queue a packet for transmission on the USB, the first thing the MCU needs is a packet. This can either be a packet that the MCU allocated, or it can be one that was allocated by the MMU as a result of a USB reception. In fact, taking a packet number that arrived in the **RXFIFO** and retransmitting it is the most basic form of Loopback test with this device.

Once a packet has been obtained, the next thing to do is to write the desired byte count to the header portion of the packet; this is how the SIE knows how many bytes to send on the USB. Finally, the data portion of the packet can be filled, starting at a byte offset of 8, since the first 8 bytes are considered to be the header by the SIE.

The byte count is usually written to the packet by the MCU using PIO (Programmed I/O) through the **MMU_DATA** register. The data portion of the packet is usually filled by the DMAC (which is discussed in a later chapter) for high performance devices, but can also be filled by the MCU for slower non-DMA devices, and is almost always filled by the MCU when handling EP0 traffic. When writing to a packet using the MCU, the sequence is as follows:

```
PNR = pkt;
bOldEA = EA;
InterruptDisable();
PRL = 6;                    /* offset of count low byte in packet header */
PRH = PRH_PNR_ | PRH_WRITE_ | PRH_AUTO_INCR_;
EA = bOldEA;

MMU_DATA = LOBYTE(wSize);
MMU_DATA = HIBYTE(wSize);

while(bytesToSend--)
        MMU_DATA = *pBuf++;

/*
 *      TODO:  be sure to wait at least 2.52 uS before changing PRH or PNR,
 *                  which is not hard to do on this MCU
 */
```

Setting the **PRH_PNR_** field (actually, it clears a bit) in the **PRH** register causes the MMU to use the packet number in the **PNR** register, rather than the head of the **RXFIFO**, for the subsequent access through the **MMU_DATA** register. For an example using this technique, refer again to the **MakePkt()** function in **Chpt5.c.**

Each of the 16 USB TxEPs has a 5-deep TxFIFO associated with it. The empty/full status of each TxFIFO is available to the MCU in the **TXSTAT_A** - **TXSTAT_D** registers. When the MCU wants to transmit a packet on a given TxEP, it must first check to make sure the corresponding TxFIFO is not already full, and then the MCU must push the packet onto the TxFIFO. Once the packet is filled with data, its packet size is set (the order is not important, just as long as they both get done), and it is known that there is room in the desired TxFIFO, then it is time to queue the packet for transmission. To do this, the packet number must be written to the **PNR**, the desired USB Endpoint Address must be written to the **TX_SEL** register, and the **MMUCR** must be written with the command: **MMUCR** = **MMUCR_ENQUEUE_**. The following code sequence illustrates these techniques by queuing a packet for transmission on txEP2:

```
if (!(TXSTAT_A & TXSTAT_A_EP2TX_FULL_)) {
        PNR = pkt;
        TX_SEL = 2;
        MMUCR = MMUCR_ENQUEUE_;
}
```

When IN tokens arrive from the Host, the SIE will transmit the packets in the order they were pushed on the corresponding TxFIFO. As each packet transmission is completed, the packet number is pushed onto the Tx Completion Queue, which the MCU can access in the **TX_MGT2** register. Note that the MCU is not required to inspect this queue, it is simply available if the firmware author wishes to use it; there is no problem with letting the queue overflow. One use for this queue is the situation in which the firmware is implementing Memory Management Policy code, and as a part of that code, it is keeping track of how many packets are presently owned by each TxEP. By inspecting the Tx Completion Queue, the code can correctly decrement the count for the corresponding TxEP as each packet is actually sent. For example:

```
        while (! ((temp=TX_MGMT2) & CTX_EMPTY_) ) {
                pkt = temp & PN_MASK_;
                /* TODO:  whatever you want to do with the packets */
        }
```

When a packet is transmitted, the default case for the MMU is to free the corresponding packet memory automatically. However, this feature can be disabled using the **TX_MGT_MEM_DALL_** bit in the **TX_MGT** register. One possible reason for not having the packets freed automatically is to permit the packet memories to be recycled after transmission without having to go through a new allocation. However, the MMU can allocate packets quickly, so there is usually no obvious benefit to this approach.

Flushing a TxFIFO:

There are situations in which it is necessary to flush the packets that have already been queued for transmission on a particular EP; an example of this is when handling a CFES command. There is a **RESET_TX_** command in the **MMUCR**, but this will ONLY reset the specified TxFIFO -- it will NOT free the associated packet memories. In order to release the packets, the MCU must remove the packets from the TxFIFO and free them one at a time; the **POP_TX** register provides the mechanism for doing this. The following code sequence illustrates the correct way to flush a TxFIFO:

```
        MMUTX_SEL = 2;
        while (! (TXSTAT_A & TXSTAT_A_EP2TX_EMPTY)) {
                PNR = POP_TX & PN_MASK_;
                MMUCR = MMUCR_RELEASE_;
        }
        MMUCR = MMUCR_RESET_TX_;
```

Although the methods used to control the state of each EP are described in a later chapter, it is important to note here that the EP corresponding to a TxFIFO that is being flushed must *NOT* be enabled during the flushing operation due to the possibility of the Host issuing an IN token during the flush. Note that when handling a CFES command, the EP is already STALL'd, so this requirement is automatically satisfied.

Memory Management Policy (MMP)

PAGS_FREE Register:

The MCU can determine the current number of free memory pages by inspecting the **PAGS_FREE** register, but this is of limited use in a USB application while pages are constantly being allocating and freed. There is a **PAGS_FREE_NAK_ALLRX_** bit in the **PAGS_FREE** register that can be used to reduce the traffic, but this should not be used often, if at all, due to performance considerations. Potentially this combination could be used to implement a stochastic MMP algorithm.

It is the author's preference to use a deterministic MMP approach in which the peak memory usage of each EP is planned in advance, and the run-time code consists of making sure that no EP ever goes over its limit. An example of such an approach appears in a later chapter.

USB Tx MMP:

From the discussion above, it can be seen that there are at least 3 ways to implement Memory Management Policy (MMP) firmware for USB transmission.

Perhaps the simplest method is to allocate one packet at a time to a given TxEP; this is easily done using the corresponding EMPTY bit in the TXSTAT register. Such an approach is quite suitable for low bandwidth EPs, such as EP0 or IRQ EPs (e.g., HID devices). An example of this method for EP0 appears in a later chapter.

Almost as simple is the case where up to 5 packets are allocated for a given TxEP; this is easily done using the corresponding FULL bit in the TXSTAT register. This is often suitable for BULK EPs, but the fixed choice of 5 is not always desired. An example of this method for BULK EPs appears in a later chapter.

The ultimate in flexibility is achieved by making use of the **TX_MGMT2** register. By monitoring the actual transmission of each individual packet, the firmware has the ability to implement an arbitrary MMP algorithm. As a minimum, such algorithms will usually need to keep track of how many packets are presently owned by each TxEP by incrementing a counter for each TxEP whenever a packet is allocated, and decrementing the counter after the packet is transmitted. In order to avoid losing packets as a result of the CTX FIFO overflowing, it is typically necessary to limit the sum of all queued Tx packets to 16 maximum. However, all of this flexibility does come at a price, both in terms of RAM space and MCU bandwidth utilization; it is suggested that these factors be carefully evaluated before making a decision to use this approach.

USB Rx MMP:

For Rx MMP, there is no analogy to the simple Tx case. It was previously shown that packets addressed at all enabled RxEPs arrive in a single RXFIFO, and the SIE does not provide any automatic flow control, the way it does when a TxFIFO is empty.

For some types of EPs, the USB traffic makes MMP easy. For example, an IRQ OUT EP will be sent a single packet once every "N" frames, as defined in its endpoint descriptor. An ISO OUT EP is similar, except that there will always be a single packet in every USB frame (neglecting bus errors). For both of these cases, no flow control is necessary or desirable, and the MMP firmware simply consists of making sure that the packets are consumed at the rate at which they are sent. About the only complication relates to ISO EPs, in which case is it is desirable to provide at least 3 USB frames of buffering in order to handle asynchronous isochronous devices, or at least 2 USB frames of buffering in order to handle synchronous isochronous devices.

For BULK EPs and Control Write transfers on EP0, the situation is not as simple. It must be assumed that a fast Host will attempt to send many packets in a single USB frame. Assuming that the MaxSize is set at 64 bytes, up to 19 packets/frame is theoretically possible. In order to prevent the Host from overrunning a slow device, the device must flow control the Host. Although a complete description of the **EPCTRL** registers appears in a later chapter, suffice it to say for now that firmware can use this set of registers to mark any given RxEP (or combination of RxEPs) "busy", which will cause OUT tokens addressed at that EP to be NAK'd to the Host, and the packets will not be received; this is the basis of flow control. Firmware is at complete liberty regarding how it uses this capability, but the simplest algorithm consists of deciding at software design time the maximum number of packets each EP will be permitted to consume at any given time. At run time, a count is maintained for each EP that is incremented as packets arrive in the RxISR and decremented in the foreground as the packets are freed. In the ISR, when the packet count exceeds a given threshold, the firmware marks the EP busy; in the foreground, when the packet drops below a given threshold, the EP is made unbusy. As was discussed in Chapter 2, if the packet count variable or the **EPCTRL** register is accessed by both the foreground and the ISR, then the foreground code must mask IRQ's during the access; fortunately, this only takes a couple of microseconds. The busy and unbusy threshold values can be the same, or a hysteresis can be inserted by making them different; it is up to the firmware author to decide. An example of this method appears in a later chapter.

It should be noted that this method is not completely deterministic, due to firmware timing; however, once the firmware timing is included in the analysis, then it is quite deterministic, which is absolutely necessary for USB BULK transfers. First of all, the firmware will not see a packet in the RXFIFO until it has completed being received; assuming that the packet at the head of the RXFIFO puts a given EP at its limit, by the time the firmware makes the EP busy, the next packet may have started arriving (depending on the current traffic pattern), so the total number of packets will be one more than the threshold. Assuming that the foreground software disables IRQs for brief periods of time, and considering that it takes a finite amount of time to get into the RXFIFO loop in the ISR, at least one more packet could arrive. The result is that it is not uncommon to receive 2 packets more than the threshold value. As a rule of thumb, it is a good idea to plan on a peak usage that is 3 packets more than the defined threshold; unless the foreground software behaves very badly with respect to IRQ disabling, this should be sufficient. However, since software timing varies with each individual application, there is no substitute for proper design, analysis and measurement in order to determine the precise values to use in any given application.

As an example, suppose that the IRQ latency into the RxFIFO loop is 20 uS (including MMU register save), the loop takes 50 uS until the EP is made busy, and the foreground disables IRQ's for a maximum time of 60 uS; the resulting time from USB IRQ to EP busy is 130 uS maximum. If the Host is sending packets to the same EP every 50-55 uS, then the EP will be made busy while the third frame above threshold is in the process of arriving. The precise values will vary with the application, but this is a typical example. In order to measure precise values for a particular application, the code can be instrumented with GPIO pulses in the RxISR and wherever the foreground disables IRQ's; some visual inspection of the foreground code can help in locating the candidate areas that might have the longest IRQ disable time. In order to measure the USB Rx IRQ latency, it is also necessary to monitor the physical USB signals.

GP_FIFO's:

In the Data Sheet, these are lumped in with the ISA Bus Control Registers. In this document, they appear here in with the MMU Registers. In the actual device, they do not have anything to do with either of these, but they are so similar to the FIFO's that are part of the MMU, that this seems like the best place to put them.

Each of these FIFO's is byte-wide and 8 deep, with its own status register that indicates the empty/full status, just like the TxFIFO's. At POR, they are cleared empty. Software should never read from them when empty, or write to them when full, because the result is not defined. Use of these FIFO's is as follows:

```
/* push a packet onto GP_FIFO1 */
if (!(GPFIFO1_STS & GPFIFO_FULL_))
        GP_FIFO1 = pkt;

/* pop a packet from GP_FIFO1 */
if (!(GPFIFO1_STS & GPFIFO_EMPTY_))
        pkt = GP_FIFO1;
```

There is an example of using these FIFO's in **Ep0RxIsr()** in **UsbCore.C**, which is discussed in a later chapter.

CHAPTER 5 - DMA

Shared Bus Architecture Details:

The USB97C100 contains an 8237 DMAC, which is the same device used in the PC. However, the interface to the MCU in this device is different than the MPU interface in the PC because, unlike an MPU, the MCU does not have any HRQ (Hold ReQuest) or HLDA (HoLD Acknowledge) signals with which to accomplish the traditional interface. Although this section summarizes the DMAC operation, the interested reader is referred to the 8237 Data Sheet for a complete description of the device.

In order to understand the operation of this interface, it is useful to first understand how the DMAC normally interfaces to an MPU. In this situation, when the DMAC wants to perform a transfer on the bus, it issues an HRQ to the MPU. The MPU will complete whatever instruction it is currently executing, it will then release the bus, and it will indicate this release to the DMAC by activating its HLDA signal. Once the DMAC sees the HLDA, it proceeds to drive the bus, and signals this by activating its AEN signal. Devices attached to the bus can then differentiate between MPU and DMAC transfers by inspection of the AEN signal. When the DMAC has finished with its transfer, it will release the bus and the AEN and HRQ signals. When the MPU sees the HRQ signal release, it will release its HLDA and will then drive the bus again. Because of this hardware handshaking between the MPU and the DMAC, the DMA transfers can occur in between the MPU instruction executions; this technique is sometimes called "cycle stealing" because the DMAC is effectively stealing bus cycles from the MPU.

Since the MCU has no HRQ or HLDA signals, the handshaking described above needs to be accomplished in software on this device, which is why it is important for the programmer to fully understand the operation of the traditional hardware approach. The key to the software approach is the **BUS_REQ** register. Setting the **BUS_REQ_HLDA_** bit in this register enables a hardware gate that issues an HLDA signal to the DMAC whenever the DMAC issues an HRQ; in this state, the DMAC will obtain ownership of the bus "immediately" whenever it asks for it. If the MCU clears the **BUS_REQ_HLDA_** bit, then the DMAC will complete its current transfer, after which it will release the bus and deactivate AEN. Since the MCU can read the state of AEN in the **BUS_REQ_AEN_**bit, it can use this bit to tell when the DMAC has actually released the bus and the MCU then owns it.

There are **IsaAcquire()** and **IsaRelease()** macro functions defined in **Usb97100.H** that illustrate this process; fortunately the macro's themselves are much shorter than the explanation of how they work:

```
#define IsaRelease()    BUS_REQ = BUS_REQ_HLDA_

#define IsaAcquire()    BUS_REQ = 0x00;  while (BUS_REQ & BUS_REQ_AEN_)
```

[Aside: note that the trailing semicolons are intentionally omitted from the #defines so that these macros are used exactly like functions in the source code that calls them. Also note that, since they are macro's, there is no issue of reentrancy should it be desired to use them in multiple threads e.g., foreground and an ISR.]

Note that the **IsaRelease()** macro always executes "immediately" because all it has to do is set the HLDA bit. However, the **IsaAcquire()** macro is another story. In the same way that an MPU cannot release a bus in the middle of an instruction, a DMAC cannot release a bus in the middle of a transfer. As a result, the **IsaAcquire()** macro can take an appreciable amount of time to execute, depending on what the DMAC is doing at the time; this is described in detail in the following sections.

It is important to realize that all DMA transfers are suspended for the duration of the time that the MCU owns the ISA bus. This is unlike the cycle-stealing approach in the PC in which, while the MPU is setting up one DMA channel for a future session, the other channels can continue to transfer concurrently. As a result of this, it is important for the MCU to acquire the bus as infrequently as possible, and to retain ownership for the shortest amount of time possible. As part of this, it is usually a good idea to disable IRQ's before acquiring ownership and to not reenable them until the bus has been released; failure to do so can cause the MCU to retain ownership for an extended period of time if substantial time is spent executing ISR code during the ownership interval. For example:

```
oldEA = EA;
        InterruptDisable();
        IsaAcquire();

        /* set up the next DMA session as quickly as possible */

        IsaRelease();
        EA = oldEA;
```

This type of code sequence was seen extensively in Chapter 3, and this is the reason for it.

DMA Channels

The DMAC contains 4 independent DMA channels. Each DMA channel can be individually programmed for Mode and transfer Type, as described in the following sections, and can be individually enabled. The exception to this is for Memory-To-Memory DMA, which always uses both DMA channels 0 and 1, as described below.

DMA Transfer Modes

The DMAC supports three (3) different transfer modes: Single, Demand, and Block. As part of discussing these, it is important to differentiate between a DMA cycle, a DMA transfer and a DMA session: a "session" is the movement of the entire block of data programmed on a given DMA channel, and is composed of transfer(s); a transfer is an indivisible unit of data movement within which the DMAC cannot release the HRQ signal (analogous to an MPU not being able to release the bus within a single instruction) and is composed of DMA cycle(s); a DMA cycle is an individual bus cycle, which is also indivisible.

In **Single Transfer Mode**, a single data byte is transferred for each DRQ issued by the attached device. The DMAC is able to release HRQ after each individual byte transfer in response to the MCU deactivating HLDA. As a result, **IsaAcquire()** executes rapidly if all of the attached devices use this mode. However, this mode offers low performance because of the hardware overhead involved, so most high bandwidth peripherals do not use it.

In **Demand Transfer Mode**, bytes are transferred for the duration of the time that the attached device holds DRQ active. Most devices that use Demand Transfer Mode (e.g., LPT in ECP mode, audio codec, etc.) contain a counter that limits the maximum size of each burst in order to prevent the device from holding the bus so long that other devices are starved for data. Note that the DMAC cannot release the bus in the middle of a transfer, only in between transfers, so the execution time of **IsaAquire()** becomes bounded by the largest burst size of the attached devices; this is usually on the order of 10 uS or so, which is not too bad. Examples of this type of transfer are included in the example code for this chapter.

In **Block Transfer Mode** the entire DMA session is executed in response to a single DRQ from the attached device (i.e., the entire session is treated as a single transfer), and the DMAC cannot release the bus for the duration of the transfer. In this mode, the only limit on the amount of time that the bus will be owned is the size of the session programmed by the MCU. For the USB97C100 device, the primary application for Block Transfers is when performing Memory-To-Memory DMA (discussed below) to move a packet between the MMU and external ISA RAM; since this is most common with BULK packets, whose size is limited to 64 bytes maximum, the transfer time is 64 uS at 8 MHZ. It is desirable that **IsaAcquire()** not be called with IRQ's disabled while such a transfer in progress, especially if the session size is large, and techniques for avoiding this are discussed in a later section of this chapter.

DMA Transfer Types

Although the DMAC offers others, the three (3) transfer types of greatest relevance are Memory Read, Memory Write and Memory-To-Memory.

In Memory Read and Memory Write, the transfer is between the memory and the attached device. For these transfers, the DRQ is generated by the attached device in hardware, and usually either Single Transfer or Demand Transfer mode is employed. As a result of this, **IsaAcquire()** executes fairly rapidly when all enabled channels are used in this way. Examples of this type of transfer are included in the example code for this chapter. If the target memory is the ISA RAM, then only channels 2 or 3 can be used, while all 4 channels are capable of device DMA with the MMU.

For Memory-To-Memory transfers, channels 0 and 1 are both used. In addition, a bit in the **DMA_CMD** register must be set in order to establish this mode of operation. Unlike device DMA, Memory-To-Memory transfers must always use Block Transfer Mode, which means that **IsaAcquire()** can take a long time to execute. In addition, the channels should remain masked to hardware DRQ's, and a software DRQ is used instead. Examples of this type of transfer are included in the example code for this chapter.

It should be noted that Memory-To-Memory transfers have an adverse effect on performance since each individual data movement involves two back-to-back DMA cycles: the first cycle reads from the source memory into a temporary register inside the DMAC, and the second cycle writes to the destination memory from the temporary register. In addition, there is usually a transfer either to or from a Device as the ultimate source or sink of the data, so there is a total of 3 DMA bus cycles involved in each byte movement. As a result of this, Memory-To-Memory DMA as a method of performing scatter/gather should not be done for very high bandwidth devices. However, it can be extremely useful for certain types of devices, a prime example of which is an FDC, in which it is desired to be able to read or write an entire track of the media in order to obtain the best performance from the physical (i.e., mechanical) device; since the sustained throughput is fairly low, the relative inefficiency of the Memory-To-Memory DMA is not an issue, and the overall performance is increased due to the improved utilization of the mechanical device.

BUS_REQ_INH_TCx:

Normally, the DMAC activates the TC (Terminal Count) signal during the final transfer of a DMA session; this informs the attached device that this is the final transfer. In the USB97C100 device, firmware can control the gating of the TC signal for each individual DMA channel in order to prevent a device from seeing the TC signal. This provides a mechanism for doing Scatter-Gather DMA in software. Since disabling the TC to a device will normally prevent that device from issuing an IRQ, the DMAC has the ability to issue an IRQ as a result of a channel TC and/or DRQ, which is globally enabled and disabled using the **INT0_ISADMA_** bit in the **IMR_0** register; individual DMA channel DRQ and/or TC IRQ's are enabled and disabled using the bits in the **BUS_MASK** register. Of course, TC can be polled, as previously described, instead of interrupt driven, if desired.

As an example, suppose that a number of small individual packets (e.g., 64 byte BULK packets) arrives on the USB, and that it is desired to create the illusion of a single large contiguous DMA buffer and session when transferring the packet contents to a device. Due to the way the MMU allocates packet numbers, the organization of the DMAC address space for the packets, and the 8-byte packet headers prepended by the SIE, it is never possible for the payload data of the packets to be physically contiguous. However, by setting up multiple DMA sessions, one for each packet, and masking the TC to the device for all but the final session, from the perspective of the device, the multiple small DMA sessions appear to be one large DMA session.

DMA_STS and BUS_STAT_CHxTC:

The DMAC contains a **DMA_STS** register that, among other things, contains a bit indicating the TC status of each channel. These bits can be useful in order to determine whether a given channel has completed the programmed DMA session or not. It is important to note that these bits **CLEAR ON READ**, so if this technique is used for multiple channels, then it will be necessary to shadow these bits in software. A suggested technique is to make a function e.g., **BYTE DmaGetSts()** which reads from the physical **DMA_STS** register and OR's the contents into a shadow byte, and returns the final shadow byte. A companion function e.g., **DmaClearTC()** can do the same, but will also clear the corresponding bit in the shadow copy; this same functionality also needs to occur each time a DMA session is setup on a channel. Of course, every experienced firmware author has developed a favorite set of techniques for dealing with such hardware, and any solution is valid that provides a properly working result.

However, since the DMAC is mapped in the ISA address space, accesssing the **DMA_STS** register requires that the MCU must first acquire ISA bus ownership, which defeats one of its best possible uses -- to determine if a DMA session is done BEFORE acquiring the ISA bus, not afterwards. However, the **DMA_STS** register is shadowed in the **BUS_STAT** register, which is mapped directly in the XDATA space, so the MCU can read this shadow register without acquiring the ISA bus. In addition, since this is a shadow register, and not the physical **DMA_STS** register, reads of this register are non-destructive i.e., no bits get cleared by reading the shadow copy. This permits firmware to rapidly determine if the DMA session on any combination of channels has completed or not without having to acquire the ISA bus in order to do it. It is important to note that the physical **DMA_STS** register must be read during session setup in order for the shadow register to be useful and that, since the shadow register is backing the physical **DMA_STS** register with the clear-on-read characteristic, the **BUS_STAT** register must be shadowed in software as well if the TC from multiple DMA channels is to be checked.

DMAC Address Space:

The DMAC is capable of addressing a 64 KB address space. As is shown in the USB97C100 Data Sheet, the low 32 KB of this address space maps straight-through to the bottom 32 KB of the ISA Memory address space. This permits the DMAC to access an external RAM on the ISA bus, assuming that one is placed in this region. This can be useful if it is desired to transfer more data to or from a peripheral than will fit in the MMU buffers (e.g., reading/writing an entire track on an FDC, handling multiple max. size IrDA frames, etc.).

The DMAC can directly address each of the 32 packet memories in the high 32 KB of its address space. Each 1 KB block of this region corresponds directly to each of the 32 packets. Note that this tacitly places a limit of 1016 bytes on the payload data, since each packet has an 8 byte header.

Sample Code: Chpt5.Hex

This program sends a short test page to an HP-PCL printer (e.g., HP Deskjet, HP Laserjet, or compatible) attached to the EVB using 2 different DMA techniques. If a physical printer is not available, it is possible to create a "NULL Printer" by jumping pins 11 (BUSY) and 24 (GND) on a male DB25 connector, or making the equivalent connection on the EVB. The program makes use of DMA channel 3 for the LPT DMA, so the DMA3 jumpers for the SIO device must be installed on the EVB; for Model 6075 Rev. B, these are at JP13 and JP15. The program displays its progress on the COM2 port, with the same arrangement as **Chpt3.Hex**.

There is a set of 3 BYTE_C strings that contain a Prolog to be sent to the printer before the actual message text, an actual message, and an Epilog to be sent after the message to tell the printer to render the page. These are contained in **cProlog[]**, **cMsg[]** and **cEpilog[]** respectively. The **MakePkt()** function allocates 3 packet memories and copies these strings into them, using techniques discussed in Chapter 4. Although it is not necessary for this example, the packets are treated as having an 8-byte header in the same way as they would if they were to be transmitted or received on the USB. The companion function **FreePkt()** is used to free the packet memories at the end of the program.

Although the details of ECP operation are beyond the scope of this document, suffice it to say that there is a mode called "Parallel Port Fifo Mode" (mode 010) in which data is sent using DMA with standard Centronics handshakes; flow control uses the BUSY signal only. **SioInit()** in **Sio.C** configures the LPT device for ECP operation. Setting the ECR for 0x40 sets mode 010 with DMA disabled, and setting the ECR to 0x48 sets the same mode with DMA enabled. The interested reader is referred to the SIO Data Sheet for a more complete discussion.

The purpose of this code example is to focus on the issues related to DMA in general, rather than ECP in particular. In order to reduce ECP related clutter, things like manipulations of the ECR occur in functions that are separated from the DMA related functions, which would hardly be considered best practice for an actual ECP application. There are numerous comments in the code suggesting changes to be made in order to improve performance for an actual ECP application.

DMA Direct From MMU to LPT:

In the first DMA method, the function **LptSendPkt()** is used to send each packet directly from the MMU to the LPT. The function begins by calling **LptStopDma()** to disable DMA in the ECP device while a new DMA session is being set up. This function sets the ECR to 0x40, with the usual IRQ disable/restore, **IsaAcquire()**, etc. **DmaPktWithDev()** (discussed below) is then called to setup and start the DMA session. **LptStartDma()** is then called to enable DMA in the ECP device by setting the ECR to 0x48, and the function then waits for the TC in the **BUS_STAT** register before returning, which indicates that the DMA session has completed. Note that the ECR must be re-written to 0x48 each time it sees a TC, which happens on every DMA session in this example. If the TC were masked in the **BUS_REQ** register, then the device would never see any TC's, so the code to write the ECR each session could be removed, which would reduce the execution time of the software. In this example, the TC mask for **BUS_STAT** is the manifest constant **cLPT_TC_MASK**, which is derrived from the **LPT_DMA** constant in **SIO.H**. The use of constants in this way is good for execution speed, since everything is calculated at compile time, rather than at run time. The MCU is also quite efficient at testing for a single bit set using a JB instruction, as was discussed in Chapter 2.

DmaPktWithDev() is a general purpose function that will perform a DMA transfer between any packet memory on any DMA channel using any Mode and Transfer Type in either direction (i.e., memory read or write). As such, it should prove to be a useful point of departure for any particular DMA application.

The function begins by reading the packet size from the packet header; it then validates the size by making sure that there is at least 1 payload data byte; it does not check for a maximum of 1016 bytes. Note that in many applications the packet size is limited to 64 payload data bytes (e.g., USB BULK packets), and in many cases the packet size is already known to be valid before being passed to a DMA setup function, so this code could be made more efficient by doing the size check on a BYTE basis, or skipping it altogether. The size is next adjusted for the DMAC by subtracting 9, which accounts for the 8 byte header and the fact that the DMAC requires that the session size be the number of bytes **MINUS 1**.

The key to the flexibility about the arbitrary channel, mode, transfer type etc. is the **dmaChMode** argument that is passed -- this is the value that will ultimately be placed in the **DMA_MODE** register, which contains all of this information. In order to be able to determine which address and count registers to use, the **dmaCh** portion of the byte is masked off, and a (volatile) pointer (**pDmaReg**) is initialized to point to the address register of the specified channel. Looking at **Usb97100.H**, it can be seen that the **DMA_ADDRx** and **DMA_CNTx** registers are arranged in consecutive order starting at 0x4000, which is why **pDmaReg** is initialized the way that it is. In the example code, the **dmaChMode** value is obtained from **cLPT_DMA_CH_MODE**, which is based on manifest constants in **Usb97100.H** and **SIO.H**.

The IRQ state is then saved and disabled, and the ISA bus acquired as usual. In order to be able to access any of the DMAC registers, the **IOBASE** register is set to **DMA_IOBASE**. Since DMA channels 0 and 1 can be used for both Memory-To-Memory and device DMA in the same application, a check is made to see if the current channel is either of these and, if it is, then the **DMA_CMD** register is written to clear Memory-To-Memory mode. Note that this could be skipped in an application if this is not relevant.

In general, it is a good idea to mask any channel while programming it, and then to unmask it when finished. This is accomplished by writing the **DMA_MASK** register, which is done next. Next, the **DmaClearByteFF()** macro function is called in order to clear the byte pointer flip-flop. This flip-flop controls the reading/writing of WORD registers in the DMAC -- the first access after clearing the flip-flop is the low byte, followed by the high byte. For this reason, if DMA is ever done in an ISR as well as in the foreground, the foreground thread **MUST** disable IRQ's, at least while programming any WORD registers in order to avoid having the ISR upset the state of this flip-flop. The byte flip-flop toggles after each byte write so, if consecutive word registers are programmed, it is not necessary to re-clear the flip-flop before each word.

The starting address of the session is then written to the appropriate **DMA_ADDRx** through **pDmaReg**. Of course, this pointer could have been dereferenced as an array, but that results in slower code execution, so the pointer notation was used. Note that the address value is 8 in the low byte to skip past the packet header. The high byte of the address has the MSB set to 1 in order to select the upper 32 KB of the DMAC address space, which is where the packet memories are mapped, and the packet number is shifted right by 2 bits in order to take into account the 1 KB packet locations. The **pDmaReg** pointer is post-incremented after writing the high address byte so that it points to the appropriate **DMA_CNTx** register, and the 2 bytes of count are written.

Next, the **DMA_MODE** register is written to set the transfer mode, direction, etc. Next, the **DMA_STS** register is read in order to clear any old TC's that might be left over from a previous session, since this code will detect the end of the session by polling for the TC in the **BUS_STAT** shadow register. Finally, the DMA channel is unmasked to enable it, the ISA bus is released to the DMAC, and the IRQ enable state is restored.

In USB DMA applications that use BULK packets, a DMA session must be set up for each packet, so the efficiency of the code that does this is usually of great concern. As a result, it is suggested that **every reasonable effort be made to streamline this code as much as possible**. For example, do not waste time checking the packet size if it is already known to be valid; do not use a WORD if a BYTE will do. Do not waste time setting the **DMA_MODE** each session if it is the same every time; do it at initialization instead. Do not pass the **dmaChMode** at all, but have a separate dedicated function for each DMA device; doing so also elminates the need to use a pointer for register access, which also improves performance. Consider masking TC's in order to avoid having to set e.g., the **ECR** each time a session is set up. If it is necessary to set a register like **ECR**, do it right in the dedicated DMA function just before releasing the ISA bus, rather than in a separate function where the entire **EA/BUS_REQ** dance has to be done again.

DMA From ISA RAM to LPT:

Although this is out of order with respect to the example code, the **DmaIsaWithDev()** function is so similar to the previous function that it naturally follows here in the discussion. The primary difference is that the address and count of a buffer in ISA RAM are passed, instead of a packet number with the size contained in its header. On entry, the address and size are validated, with the criterion being that the entire session must be contained in the low 32 KB. In an actual application, this should be streamlined or removed, as was previously discussed. The only other difference is that the count is only adjusted by 1 for the DMAC, instead of 9, since there is no packet header to skip. As was previously mentioned, only DMA channels 2 and 3 are capable of using an ISA memory target for device DMA.

DMA From MMU to ISA RAM:

In order to have data in ISA RAM suitable to DMA to the LPT, the function **DmaPktToIsa()** is called for each of the 3 packets to create a single DMA buffer, which is then sent to the LPT using **DmaIsaWithDev()**.

As was previously mentioned, Memory-To-Memory Transfers must always make use of DMA channels 0 and 1, and must always use Block Mode. The **DMA_CMD_MEM2MEM_** bit in the **DMA_CMD** register must be set. Channel 0 address must be set for the source memory block, and Channel 1 address must be set for the destination. Channel 1 Count must be set for the session size, and it is Channel 1 that will TC when the session is completed. Both channels must be set for Block mode. Both channels should be masked before programming the session, and remain masked when complete. The session is actually started by issuing a "Software DRQ", as opposed to the hardware DRQ used with peripheral devices, on Channel 0 using the **DMA_REQ** register. As usual, the IRQ enable state is saved/restored, the ISA bus acquired/released, and the **IOBASE** set to **DMA_IOBASE** to make the DMAC addressable.

CHAPTER 6 - GETTING ON THE BUS

This chapter describes everything that is needed to implement a fully Chapter 9 compliant USB Device using the USB97C100. In addition to the USB Specification, this text makes frequent reference to the example code, which is contained in the following files:

Chpt6.C -- application code
Chpt6.H -- application data structures, especially USB descriptors
UsbCore.C -- core USB code
UsbCore.H -- interface between Core code and application code
Usb.H -- USB related data structures (descriptors, Setup packets, etc.)

The example code implements a complete USB device which passes UsbCheck V2.6 for Chapter 9 compliance. The device is limited to a single USB Configuration, but supports an arbitrary number of Interfaces, as defined by **cUSB_NUM_IFs** in **Chpt6.H**. Each Interface supports an arbitrary number of alternate settings, as defined by **cUsbNumAltIFs[]** in **Chpt6.c**. Each alternate setting contains an arbitrary combination of endpoints, as defined by **usbTotalCfgDesc** in **Chpt6.h**. The basic architecture is that all application dependent code is contained in **Chpt6.C** and **Chpt6.H**, while all USB related code is contained in the other files, which act as a Kernel or Core to which new applications can be linked. In the case of most EP0 Transfers, application dependent information is obtained by having the Core code perform a function call into the Application code in order to retrieve the information; a description of each individual Transfer is included in a following section.

A number of other application dependent values are defined in Chpt6.X:

- The choice of MCU and DMA clocks; GPIO function, direction and value; MEM_BANK setting, etc. (using our old friend Usb97C100HwInit())
- All USB String Descriptors
- USB VID, PID, Class, Subclass, Protocol, etc.
- EP0 max. packet size

In order to illustrate the technique for doing so, the example code makes use of an external USB Transceiver. In order to execute this code on the EVB hardware, the jumpers for the external transceiver must be set. **On Assy 6075 Rev. B, this corresponds to intalling JP26 TUSB(0:7), JP27 1-2 and 3-4, and open JP28 1-2 and 3-4. Also GPIO jumpers JP22 must be inserted for GPIO7-HIGH and GPIO6-LOW.** Just as some EVB's do not have a COM2 port or a Flash RAM device, as was discussed in Chapter 3, some EVB's do not have an external USB Transceiver. The symbol EVB_USB_EXT_PHY is defined (in Evb.h) for EVB's that do have an external USB Transceiver, and the code is conditionally compiled, based on this symbol, in order to select the external USB Transceiver only for those EVB's that have one.

Device States:

For a more complete description of USB Device States, see section 9.1.1 in the USB Specification.

When a device is first attached to the USB after POR, it must be dormant to all bus activity, including Address0/EP0. In **UsbCore.H**, this is the **DEVICE_DEFAULT** State.

After it receives its first UsbReset, it must respond at Address0/EP0 only. UsbCore.H does not assign a new value to the software State at this point because it is not needed for anything. In this State, the Host can query the device for its USB Descriptors, etc. and will eventually send it a SET_ADDRESS command, at which point the device must move from Address0 to whichever USB Address the Host assigns; this is the **DEVICE_ADDRESS** State.

Eventually, the Host will send a SET_CONFIGURATION command; a Configuration of zero is the Unconfigured State, which means that only EP0 remains active. A non-zero Configuration is a Running State; this is the **DEVICE_CONFIGURED** State. For the most part, device software is really only concerned with whether the device is in CFG1 or not, since software behavior is mostly the same for all of the other States.

From any State it is possible for the Host to suspend the device if it is bus powered; this is the **DEVICE_SUSPENDED** State. While in Suspend, it is possible for the Host to Resume the device and, for devices that support the feature, it is possible for the device to Remote Wakeup the Host. Although the example code does not implement these features, the following sections describe the techniques involved.

USB Suspend/Resume:

The Host can instruct a USB device to Suspend by stopping all traffic (i.e., establishing a J state) to the device for a period of at least 3 mS. The Host can then Resume the device through a variety of mechanisms, each of which involves a transition out of the J state. The SIE provides the ability for the software to detect when SOF's arrive, either by triggering an MCU timer from the SOF (using the **UTIL_CONFIG** register bits), or by issuing an IRQ when the SOF arrives (using the **INT1_SOF_** bit in **IMR_1**), or by polling the **FRAMEL** register for a change in USB frame number and either triggering a timer or reading a free running timer, etc. From any of these techniques, the software can determine when the device is supposed to enter a Suspend State.

Entering the Suspend State consists of first suspending the SIE, which places it into a low power state by stopping its clock; this is set in the **SIE_CONFIG** register. The next step is to suspend the DMAC by stopping it's clock, switching the MCU to its ring oscillator and stopping the MCU clock, and enabling the ring oscillator to be stopped with the **PCON** register LSB; all of these are accomplished in the **CLOCK_SEL** register. Finally, the MCU ring oscillator is stopped by setting the LSB in the **PCON** register high. The following code sequence illustrates the technique of Suspending:

```
SIE_CONFIG |= SIE_CONFIG_SIE_SUSPEND_;
CLOCK_SEL = CLKSEL_SLEEP_ | CLKSEL_ROSC_EN_;
PCON |= 0X01;
```

When the Resume signaling arrives, the MCU will begin executing its **Isr2()** function. In that function, the **WK1_RESUME_** bit in the **WU_SRC_1** register must be cleared by reading the **WU_SRC_1** register (otherwise the ISR will keep getting re-entered, since the IRQ condition has not been cleared). It is also the author's preference to restore the **CLOCK_SEL** register in the ISR, but that could be done in the foreground instead if preferred. After returning from the ISR, execution will continue in the foreground thread at the instruction immediately after the **PCON** register LSB was set to 1. Following is some example **Isr2()** code:

```
isr = ISR_1;      /* clear bits in read */
if (isr & INT1_PWR_MNG_) {
        src = WU_SRC_1;        /* clear bits in read */

        if (src & WK1_RESUME_) {      /* just like in Usb97c100HwInit() */
                CLOCK_SEL = (BYTE)(CLKSEL_ROSC_EN_ | cClocks);
                CLOCK_SEL |= CLKSEL_MCUCLK_SRC_;
                CLOCK_SEL &= ~CLKSEL_ROSC_EN_;
        }
}
```

In order to enable the wake from Resume event, it is necessary to unmask the corresponding bit in the **WU_MSK_1** register; it is convenient to do this early in **_main()** when the other interrupt related masks are configured. **Note that if this is not done, then the device will get stuck in the Suspend State!**

```
EX1 = 1;
IMR_1 = (BYTE)~(INT1_PWR_MNG_);
WU_MSK_1 = (BYTE)(~(WK1_RESUME_ | WK1_USB_RESET_));
```

USB Remote Wakeup

Implementing this feature involves doing the Suspend/Resume activity above but, in addition to enabling wakeup from USB Resume or USB Reset, a wakeup is also enabled from one or more GIRQ signals using the bits in the **WU_SRC_2** register. When the external device asserts the corresponding GIRQ signal, the MCU wakes up with the same type of PWR_MNG IRQ as for a USB Resume; of course, the software can determine the source of the wakeup event by reading the WU_SRC_X registers. The Resume code sequence is similar, except that the software must also cause the SIE to issue Resume signaling to the Host by setting the **SIE_CONFIG_USB_RESUME_** bit in the **SIE_CONFIG** register.

It is common that the same peripheral device and GIRQ used for Remote Wakeup is also used for normal operation. When this happens, the **WU_SRC_2** register should be unmasked to enable the wakeup just before Suspending, and it should be masked again as part of the Resume. The opposite procedure should be done with the corresponding bit in the **IMR_0** register if the peripheral is interrupt driven while the device is not Suspended.

EPCTRL Registers

The USB97C100 contains a separate EPCTRL Register for each USB Endpoint. This set of registers permits software to define the state of each RxEP and TxEP as being either Disabled, Enabled, Busy or Stalled. In addition, each EP can be defined as being Isochronous or not. The behavior of an EP in each of these modes is dependent upon the state of the EP and the type of USB traffic addressed to it. The following discussion of behavior is for the case in which the MMU and RXFIFO are both not full. Note that allowing either of them to fill should not be permitted in any USB application, since every USB device must be capable of receiving a Setup Packet at any time (discussed in a later section), which is not possible if either the MMU or RXFIFO is full.

Non-ISO OUT EP's

RX_ENABLE_: Both SETUP and OUT packets are received, regardless of CRC or Data Toggle. For bad CRC, there is no handshake, otherwise the handshake is ACK. It is the responsibility of firmware to discard packets with bad CRC or Data Toggle. In addition, the **EPCTRL_RX_TOGGLE_** bit is read-only, so it is the responsibility of software to maintain a data toggle bit to be used for rejecting duplicate packets on Bulk, IRQ and Control EP's. The requirements for initializing the toggle bit vary with the EP type and are discussed in the following sections.

RX_STALL_: For OUT packets, same as **RX_ENABLE_** except that STALL handshake is sent instead of ACK. It is the responsibility of firmware to discard the packet. For SETUP packets, the packet is received and NO handshake is sent; it is the responsibility of firmware to discard the packet and clear the **RX_STALL_** condition so that the retransmitted Setup packet will subsequently be received and ACK'd.

RX_BUSY_: For OUT packets, the packet is not received and the handshake is NAK. For SETUP packets, the same situation as **RX_STALL_** above.

RX_DISABLE_: The EP is completely disabled; no packets are received and no handshakes are sent.

Non-ISO IN EP's:

TX_ENABLE_: when the TxFIFO for the EP is empty, IN tokens are responded to with a NAK handshake; if the TxFIFO is not empty, the packet is sent. The transmission is only considered complete when the ACK handshake from the Host is received, so a Host no handshake (from a bad CRC at the Host end) or a dropped ACK handshake will result in automatic retransmission in response to subsequent IN tokens from the Host. The **EPCTRL_TX_TOGGLE_** bits are writable, and define the DATAx PID to be used on the next transmission from the corresponding EP. The DATA PID toggles automatically with subsequent transmissions in response to ACK handshakes from the Host. It is the repsonsibility of software to initialize the **EPCTRL_TX_TOGGLE_** bit appropriately for the EP type (Bulk, IRQ or Control) and state, which is described in a following section. Note that, since RX flow control bits are in the same register as the writable Tx data toggle bit, **<u>Tx EP's cannot be used at the same EP address as an RxEP</u>** that will require concurrent flow control (i.e., BULK; IRQ, ISO, and Control EPs are OK) because doing so would overwrite the TxToggle in the course of doing the read-modify-write for Rx flow control. However, this restriction still provides a minimum of 15 (typically more) unidirectional pipes in addition to the Default Pipe, which should be plenty considering that there are only 32 pages of Packet Memory to share between all of the pipes anyway.

TX_STALL_: IN tokens get a STALL handshake; no packet is sent even if the TxFIFO is not empty.

TX_BUSY_: IN tokens get a NAK handshake; no packet is sent even if the TxFIFO is not empty.

TX_DISABLE: the EP is completely disabled; no packets are transmitted and no handshakes are sent.

ISO EP's

Software should only set the xx_ENABLE_ and xx_DISABLE_ values. Disabled EP's do not send or receive any packets. ISO EP's never send any handshakes.

Rx packets are received regardless of CRC errors, DATA PID or EP STALL condition. It is the responsibility of software to discard packets for CRC or STALL, but to mark their time. According to the USB Specification, DATA PID should be ignored on ISO Rx packets.

RESETS

POR

At POR time, the Core code begins execution in **_main()**, which calls **Usb97HwInit()** which was described in a previous chapter. After this, **ApHwInit()** in **Chpt6.C** is called to allow the application to initialize its hardware state; this consists of calling **SioInit()**, which was described in a previous chapter. At this point, **DBGPRINT**'s are operational. The core code then calls **ApSwInit()** in **Chpt6.C**, which causes the application to initialize its software state; this consists of calling **Ep1RxQueInit()** and **Ep3RxQueInit()** to initialize the receive queues, in addition to marking the ISA queues empty, the DMAC as unowned, and no ISR errors are pending. It should be noticed at this point that most of the data and functions in **Chpt6.C** are marked as static, since they are private to the application and should not be referenced from outside the file; by marking these items as static, the compiler's scoping rules will guarantee that no external references occur.

At this point, the core code sets up the IRQ masks, and enables the SIE and IRQ's. Finally, **ApUsbAttach()** in **Chpt6.C** is called to cause the application to attach the device to the USB; this is provided for application hardware that supports electronic attachment to the bus. For the EVB hardware, attaching to the bus is accomplished by setting the GPIO7 pin high. Since the **GPIOA_OUT** register is often shared with ISR's, the usual dance with EA is done.

Core execution continues with an infinite loop that handles USB Reset, EP0 activity, etc. as described below.

USB Reset

When the SIE detects a USB Reset, it signifies this by activating the **SIESTAT_USB_RESET_** bit in the **SIE_STAT** register. This event can also issue an IRQ using the **WK1_USB_RESET_** bit in the **WU_SRC_1** register, which in turn is enabled by the **INT1_PWR_MNG_** bit in **IMR_1**; if both of these conditions are simultaneously enabled, then a USB Reset will cause an IRQ_2. The sample code enables this condition and calls back to the Application with **ApUsbIsrReset()**, but the sample code does not have any work to do at interrupt time, so it just sets a **bUsbResetPending** flag and returns. The foreground code then uses this flag to call **UsbReset()** in **UsbCore.C**, which actually performs the reset activity.

UsbReset() sets USB Address0 and disables all EPs, before calling **ApUsbReset()** in **Chpt6.C**. **ApUsbReset()** then calls **ApSwInit()** to initialize its software state, **DmacReset()** to initialize the DMAC, and then it resets the MMU. When the application code returns to **UsbReset()**, this function waits for the USB Reset to finish before resetting the SIE and enabling EP0. The reason for this wait is that resetting the SIE also resets the counter that it uses to detect USB Reset events; if this is done while the USB Reset is still active, the SIE will detect another USB Reset event, which will then cause a new IRQ_2, etc. and the device will then get **caught in a loop** until the USB Reset is finally over. In order to avoid this condition, the code waits until the USB Reset is completed before resetting the SIE; this results in a single execution of the USB Reset software functions for each actual reset on the bus. Note that the USB Reset is required to last for a minimum of 10 mS, and the device must complete its reset processing within 10 mS after that; in practice, these timing requirements are easily satisfied without taking any special precautions, but the device programmer should certainly be aware of them anyway.

EP0 Control Transfers

There are three (3) basic types of Control Transfers that can occur, as described in the USB Specification, section 8.5.2; also, Chapter 9 describes the Standard Requests.

1) Control Read:
The Host sends a Setup packet, followed by one or more IN tokens. The wLength field in the Setup packet contains the maximum number of bytes to return; the device must limit the transfer to the specified size in the event that the requested item is larger. During the Data Stage, the Host sends IN tokens to read the packets from the device; the first data packet has a DATA1 PID, and subsequent data packets toggle the DATA PID. The device issues NAK handshakes during the Data Stage if it is busy. After the last data packet, the Host sends a zero-byte OUT packet during the Status Stage. The device handshakes this packet with either an ACK to indicate success, or a STALL for any error. If the device wishes to Stall the transfer, the first (and only) data packet should be zero length so that the Host will skip immediately to the Status Stage.

Standard Requests:
GET_STATUS, GET_DESCRIPTOR, GET_CONFIGURATION, GET_INTERFACE, SYNCH_FRAME (optional)

2) Control Write
The Host sends a Setup packet, followed by one or more OUT tokens. The wLength field in the Setup packet contains the total number of bytes that the Host will be sending. During the Data Stage, the Host sends OUT packets to the device, with the first packet having a DATA1 PID, and subsequent packets toggle the DATA PID. The device issues NAK handshakes during the Data Stage if it is busy. If the device wishes to Stall the transfer, the Stall handshake is sent during the Data Stage in response to the first (and possibly only) OUT packet. To accept the Transfer, the device receives and ACKs the packets during the Data Stage and sends a zero-byte handshake packet during the Status Stage with a DATA1 PID.

Standard Requests:
SET_DESCRIPTOR (optional)

3) No-Data Control
The Host sends a Setup packet. Since any required information is contained in the Setup packet, there is no Data Stage; this Transfer type is similar to a Control Write with zero data bytes. During the Status Stage, the Host sends IN token(s). The device sends a zero-byte packet with a DATA1 PID to indicate success, or a Stall handshake otherwise (i.e., to Stall the transfer).

Standard Requests:
CLEAR_FEATURE, SET_FEATURE, SET_ADDRESS, SET_CONFIGURATION, SET_INTERFACE

Note that when handling a SET_ADDRESS Transfer, it is necessary to not actually assign the new address until AFTER the Status Stage has completed; this is because the Host will try to read the handshake packet at the original address.

As described in sections 8.5.2.2 and 5.5.5 of the USB Specification, there are situations in which, from the device's perspective, the sequencing in a Control Transfer appears to be out of order; this can happen especially as a result of lost handshakes going back to the Host.

EP0 Stalls

Under a variety of circumstances, the device must Stall a Transfer to EP0. The method for doing this varies with the Transfer Type, as was described above. The previous description of the EP_CTRL register defines the bit fields that must be set in order to Stall each transfer type, but they are summarized here for convenience:

Control Read: Stall the Rx and then queue a zero-length Tx packet with a DATA1 PID; reading this packet causes the Host to advance to the Status Stage. Leave the Rx Stalled until the next Setup packet arrives, and then clear the Rx Stall.

Control Write: Stall the Rx and leave it Stalled until the next Setup packet arrives, then clear the Rx Stall.

No-Data Control: Stall both the Rx and Tx and leave them Stalled until the next Setup packet arrives, then clear both Stalls.

While it may not seem necessary to Stall the Rx in all cases, or to leave it Stalled until the next Setup packet, this turns out to be true when every case of lost handshakes, etc. is considered.

EP0 FSM

Since the transfers described above can involve multiple packets spread out over an appreciable amount of time, one suitable way to implement the software is as a Finite State Machine (FSM) that is called in a foreground polling loop, which is how the example code in this chapter does it. States are defined as follows:

IDLE: The device is waiting for a Setup packet to start a transfer.
SETUP: The device has received a Setup packet, but has not yet processed it.
RD_DATA: The device is in the Data Stage of a Control Read Transfer
WR_DATA: The device is in the Data State of a Control Write Transfer
WR_STATUS: The device is in the Status Stage of either a Control Write or a No-Data Control transfer.

The State transitions for each Transfer type are as follows:
CTL-READ: IDLE -> SETUP -> RD_DATA -> IDLE
CTL-WRITE: IDLE -> SETUP -> WR_DATA -> WR_STATUS -> IDLE
NoData-CTL: IDLE -> SETUP -> WR_STATUS -> IDLE

The code is implemented in **UsbCore.C**, using a foreground polling function **Ep0FSM()** and an RxISR called **Ep0RxIsr()**. Their interaction is described in the following sections. Together, these 2 functions comprise the majority of **UsbCore.C**, and are the key to reuse of this code in other applications.

In all cases, **Ep0RxIsr()** takes care of cleaning up after the previous transfer in the event that a Setup packet appears to arrive out of order from the perspective of the Device; this same function also takes care of cleaning up after any Stalls. As a result, whenever a Setup packet arrives, the software is placed into an IDLE State so that it can process the packet. Note that under some circumstances a handshake was not sent to the Host for the Setup packet; when this happens, the packet is dropped and the Host will retransmit it.

For all Transfer types, the transition from IDLE -> SETUP is performed in **Ep0RxIsr()**; this involves copying the packet into the **usbSetupPkt** structure, removing and releasing the packet, and setting the State.

Whenever **Ep0FSM()** needs to send any packets to the Host, it makes use of a companion function **Ep0SendPkt()**, which is also contained in UsbCore.C. This function takes a generic pointer to a data block to send which, although it makes the code execution relatively slow, permits sending a data block located in any memory space of the MCU. The function also takes a **bytesToSend** argument indicating the packet size; zero is a valid size for this function because zero-length packets are used as hanshakes in some Transfer types. The function allocates a packet from the MMU and saves the packet number in **ep0TxPkt**, since this value is sometimes needed in order to clean up after a transmission. The function does not perform any manipulation of the Data Toggle, which is left to the caller, since this varies depending upon the circumstance.

CTL-READ Transfers

The transition from SETUP -> RD_DATA is performed in **Ep0FSM()**. This consists of validating the Setup packet contents, and determining whether to Stall the transfer or not. In the sample code, a desire to Stall the transfer is indicated by a **wEp0BytesToSend** size of zero; the Stall is implemented as described in a previous section and the State is set to IDLE.

Validating the Setup packet for Standard Requests is performed in a large **switch** statement that has cases for each of the Standard Transfers; each case in turn validates the remaining bytes in the Setup packet according to the USB Specification. In most cases, application specific information is obtained by calling **ApUsbXXX()** functions defined in **UsbCore.H** and contained in **Chpt6.C**. A discussion of each Transfer type is contained in a following section. For Vendor or Class Requests, the Setup packet is passed to **ApUsbEp0Read()**, which is defined in **UsbCore.H** and contained in **Chpt6.C**, in order to validate the Transfer.

Assuming success, a data pointer and byte count is available for the data to be returned to the Host. The byte count is limited to the maximum size specified in the Setup Packet, and the State is set to RD_DATA. The **TX_TOGGLE_** is set to DATA1 PID in preparation for the first packet to be returned.

In the RD_DATA State, packets are sent to the Host using **cEP0_MAX_PKT_SIZE** (defined in **Chpt6.H**) packets until all of the data is sent. If the data is an exact fit in the packets, then a zero-byte packet is sent last, which the Host may or may not read. In order to avoid having to service IRQ's each time a USB packet is sent or a TxFIFO goes empty (the 2 choices for the device), EP0 transmits are cleaned up at the beginning of **Ep0FSM()** each time it is called. Eventually, the Host will send a handshake packet, which will be received in **Ep0RxIsr()**; this function will clean up any left over Tx packet and set IDLE State.

No-Data Control Transfers

Everything is very similar to the previous section, except that the State transition is to WR_STATUS if the Transfer is not Stalled, a **bResult** value of FALSE is used to request a Stall instead of **wEp0BytesToSend** being zero, and Vendor or Class requests are passed to **ApUsbEp0NoData()** instead of **ApUsbEp0Read()**. As before, most Transfers involve calls to **ApUsbXXX()** for application dependent data and validation. Any Stall is implemented as described in a previous section. Success is indicated in WR_STATUS by queueing a zero-byte Tx packet with a DATA1 PID; this transmission is cleaned up at the beginning of **Ep0FSM()** the next time it executes after the Host reads the packet. In the event that the Host handshake to the packet is lost, or if the next Setup packet arrives before the next **Ep0FSM()** execution, then the Tx is cleaned up when the next Setup packet arrives in **Ep0RxIsr()**.

Control-Write Transfers

Although none are used by the sample code, the framework is in place in case an application should need them for Vendor or Class transfers. In **Ep0FSM()** in the SETUP State, the Setup packet is passed to **ApUsbEp0Write()** in order to determine whether to Stall the Transfer or not. Any Stall is implemented as described previously, the State is set to IDLE, and the Stall is cleared in the **Ep0RxIsr()** when the next Setup packet arrives.

Assuming that the Application chooses to accept the Transfer, the State is advanced to WR_DATA, and subsequent OUT packets are queued in **GP_FIFO1** by **Ep0RxIsr()** and passed to **ApUsbEp0WriteNextPkt()** by **Ep0FSM()**. The end of the Transfer is detected in **Ep0FSM()** by receiving a less than **cEP0_MAX_PKT_SIZE** packet, at which time the State is advanced to WR_STATUS. In this State, a zero-byte handshake packet is queued, and the State is set to IDLE, as was previously described. Note that in a real application, the software should also keep track of the total bytes received from the Host during the transfer, and should Stall the transfer if it ever exceeds wLength from the Setup packet.

Ep0RxIsr() takes care of Data Toggles by initializing **bEp0RxTog** when the Setup packet arrives, and then rejecting packets and updating the toggle as each subsequent data packet arrives. It also makes the Rx BUSY as each packet arrives, and checks for overflows of **GP_FIFO1**. In the foreground, **Ep0FSM()** removes the busy whenever **GP_FIFO1** is emptied of packets in order to permit more packets to arrive.

EP0 Application Callbacks

As mentioned above, most Transfers involve application specific data, so callback functions are defined in **UsbCore.H** and the code is contained in **Chpt6.C**.

TRANSFER	TYPE	APPLICATION FUNCTION
GET_DESCRIPTOR	CTRL-READ	ApUsbGetDesc()
GET_CONFIGURATION	CTRL-READ	usbCfg (shared var.)
GET_INTERFACE	CTRL-READ	ApUsbGetIF()
GET_STATUS(DEVICE)	CTRL-READ	usbDevStatus (shared var.)
GET_STATUS(INTERFACE)	CTRL-READ	None (reserved transfer)
GET_STATUS(ENDPOINT)	CTRL-READ	ApUsbIsEpValid()
SET_ADDRESS	NO-DATA-CTRL	None (Ap doesn't care)
SET_CONFIGURATION	NO-DATA-CTRL	ApUsbSetCfg()
SET_INTERFACE	NO-DATA-CTRL	ApUsbSetIF()
CLEAR_FEATURE(EP_STALL)	NO-DATA-CTRL	ApUsbIsEpValid/ApUsbCFES()
CLEAR_FEATURE(WAKEUP)	NO-DATA-CTRL	ApUsbRemoteWakeDisable()
SET_FEATURE(EP_STALL)	NO-DATA-CTRL	ApUsbIsEpValid/ApUsbSFES()
SET_FEATURE(WAKEUP)	NO-DATA-CTRL	ApUsbRemoteWakeEnable()

For all Standard Transfers, the Core code handles validating the Setup packet contents, as well as confirming that the device is in a valid State for the Transfer (e.g., many Transfers are only valid when the device is in a CONFIGURED State). Following this, the Application callbacks are used to validate any application dependent values (e.g., whether a specified Alternate Setting is valid for a given application).

For each of the GET_XXX Transfers, the information is entirely application specific, so the Core software just calls the appropriate Application function, which then either provides the required information, or returns a zero or FALSE to indicate that the Transfer is not supported and should be Stalled.

For any Transfer that involves an EP as a Target, the Core code calls **ApUsbIsEpValid()**, since it is application specific which EP's are valid for the current combination of Alternate Interface Settings.

For SET_ADDRESS, the Core code handles setting the new USB Address after the Host has read the handshake packet. This can be seen when Tx packets are cleaned up near the entry to **Ep0FSM()**.

For SET_CONFIGURATION(0), the Core code handles disabling all EP's; the Application code is responsible for setting up any EP's for a non-zero configuration. The Application code is also responsible for configuring EP's for SET_INTERFACE, since any such settings are application dependent.

For CFES, the Core code handles flushing any TxFIFO's that the application does not flush, and it handles the EP_CTRL register for all EP's (including TxToggles), but the application is responsible for handling Rx Data Toggle initialization and flushing any Rx Queues, since these are application dependent. For SFES, the Core code handles the EP_CTRL register. For any Transfer with an EP target, the Core code validates the EP using **ApUsbIsEpValid()**, as was previously described.

For both WAKEUP Requests, all processing is handled by the application, which is also responsible for indicating the Feature state in the **usbDevStatus** shared variable.

As was previously described in the section on Control-Read Transfers, the Core code handles limiting the size (i.e., byte count) of the Data Stage, splitting it into multiple USB packets, handshakes, Stalls, etc.

Because of the amount of support provided by the Core functions, most of the Application Functions do absolutely nothing, with the possible exception of returning a pointer and/or a byte count, and the remainder do only a small amount of processing. It is by customizing these functions that new applications can be easily ported to this architecture.

APPLICATION POLLING

There are 2 polled callbacks to the application. As the name would imply, **ApCfg1Poll()** is called whenever the device State is CONFIGURED, and the Configuration is 1 (i.e., the Configured State). For this application, a total of 4 Endpoints is used, and the polling handler just calls each individual EP handler in turn. The function implemented by each EP handler is described in a later section.

ApPoll() is called any time the device is not SUSPENDED; its primary use is for debugging. The example code calls a slow polling function **ApSlowPoll()** either every USB frame while the device is receiving SOF's, or based on a timeout of 6 mS in the Configured State (CFG1), or 2 mS otherwise. **ApSlowPoll()** in turn displays and resets any ISR0 errors, and checks for any keystrokes from the debugging terminal, to which it responds by sending a DBGPRINT message. This code could be expanded to aid in debugging any other application by returning various data and/or register values, or by initiating programmer defined sequences in response to specific keystrokes. This function is instrumented by pulsing GPIO5 high for the duration of its execution. Since the **GPIOA_OUT** register is shared with the ISR, IRQ's are disabled during the access to this register.

DATA TRANSFER

Data Transfer Overview

The application code treats the low 32 KB ISA RAM as a pair of 16 KB circular buffers. It treats the 4 USB EP's as 2 pairs -- 1 Rx and 1 Tx in each pair. Within each pair, packets arriving on the RxEP are written to the ISA RAM, while packets already in the ISA RAM are queued for Tx back to the Host. In this way, each pair of EP's loops back data, with up to 127 USB packets (1 packet less than 8 KB) in each transfer. EP1 and EP3 are the 2 Rx EP's, and are identical except for using separate variables, while EP2 and EP4 are the Tx partners.

ep1RxQueue[] and **ep3RxQueue[]** are the Rx packet queues; packets are pushed on the head of each queue using **ep1RxHead** and **ep3RxHead**, and are popped from the tail of each queue using **ep1RxTail** and **ep3RxTail**, as pointers. Empty queue entries are marked with **INVALID_PN**. Each of these queues is initialized by **ApSwInit()**, which is called by the Core code during initialization and is also called by **ApUsbReset()** for a USB Reset, using **Ep1RxQueInit()** and **Ep3RxQueInit()**. The corresponding queue is flushed by **ApUsbCFES()** using **Ep1RxQueFlush()** and **Ep3RxQueFlush()**.

Since both RxEP's are for BULK packets, data toggles are maintained in **bEp1RxToggle** and **bEp3RxToggle**, as was described in Chapter 4. The data toggles are reset in **ApUsbSetCfg()** and **ApUsbCFES()**.

The memory management is based on what was described in Chapter 4. Each RxEP is allowed to use a peak of 9 MMU pages, and the Threshold is set for 6 (both numerical values are in #define's), which is enforced by making the corresponding RxEP busy in the RxISR, and unbusy in the foreground. Each TxEP is allowed to use a peak of 5 MMU pages, and this is enforced by making sure that the corresponding TxFIFO is not full before attempting to allocte a packet memory. This policy leaves a minimum of 4 free MMU pages for EP0 under peak conditions.

Each pair of Rx and TxEP's share an ISA circular buffer; packets are pushed on the head of each queue using **ep1IsaHead** and **ep3IsaHead**, and packets are popped from the tail of each queue using **ep1IsaTail** and **ep3IsaTail** as pointers. When each queue is empty, its head and tail pointers are equal; a queue is considered full when its head pointer is 1 behind its tail pointer. Each 16 KB buffer is treated as having 128 pages of 128 bytes each, so it can hold up to 127 USB packets. Memory-To-Memory DMA is used to perform the actual transfers to/from each ISA page, using **DmaEntirePktToIsa()** and **DmaIsaToEntirePkt()** respectively. The session size is always set to a full 72 bytes, which will handle a maximum size BULK packet, including its 8-byte header. This simplifies the task of handling variable size packets and, recognizing that most packets are maximum size anyway, results in good throughput as well. The details of the DMA functions are as was described in Chapter 5. The ISA queues are marked empty by **ApSwInit()**,which is called by the Core code during initialization and is also called by **ApUsbReset()** for a USB Reset.

Since all 4 EP's perform Memory-To-Memory DMA, they must all share DMA channels 0 and 1. In order to support this sharing, the **dmaOwner** variable is used; it is set to the corresponding EP [1,4] that currently owns the DMAC, or else it is cleared to zero if no EP is currently using the DMAC, signifying that the DMAC is available for use.

Data Transfer Details

When packets arrive from the USB, the Core code initially recieves the IRQ in its **Isr0()** function, and passes execution to **ApIsr0()**. Note that Register Bank switching is used for speed, as was described in a previous chapter. **ApIsr0()** saves and restores the MMU state on entry and exit of the RxFIFO loop, since it needs to use some of the MMU registers in order to handle the packets. If a packet is addressed to EP0, **ApIsr0()** calls back to **Ep0RxIsr()** in the Core code in order to handle the packet. For application packets, **ApIsr0()** validates the packet and discards it for Bad CRC, Stall, or Data Toggle (since these are Bulk packets). Assuming the packet is accepted, it is pushed on the corresponding software queue; if a queue overflows, which should never happen because of the MMP code, it is considered a Fatal Error, and the corresponding EP is Stalled, and the error code is logged in **apIsr0Error**, which will be displayed by the foreground the next time **ApSlowPoll()** is called. As each packet is queued, the packet count for the EP is incremented, and the EP is made BUSY if its count exceeds the defined threshold. For instrumentation, **Isr0()** pulses GPIO2 (see the comment in the GPIO section later in this chapter) high for the duration of its execution, **ApIsr0()** pulses GPIO1 high each time around the RxFIFO loop, and it also pulses GPIO0 high whenever it discards an EP1 or EP3 packet for bad Data Toggle.

Eventually, the corresponding RxEP handler (1 or 3) will execute in the foreground and will find that its Rx queue is not empty, its ISA queue is not full, and that the DMAC is available. At this point, it will claim ownership of the DMAC by setting the **dmaOwner** to its EP number, and will start a DMA session using **DmaEntirePktToIsa()**. Eventually, it will see that it is the **dmaOwner** and that the session has completed, at which point it will update its queue pointers and mark the **dmaOwner=0** to signify that another EP may use it. Once the DMA is complete, the packet is freed in the MMU, the packet count is decremented, and the EP is made unbusy if the total number of packets has dropped below threshold.

Note that the packet count and EPCTRL register are shared with the ISR, so IRQ's are disabled/enabled around these accesses. Also, EP1 raises GPIO3 high when starting a DMA session, and brings it back low when the DMA completes. Similarly, since **GPIOA_OUT** is shared with the ISR, IRQ's are disabled around accesses to this register.

Eventually, the corresponding TxEP handler (2 or 4) will execute in the foreground and will find that its Tx queue is not full, its ISA queue is not empty, and that the DMAC is available. At this point, it will claim ownership of the DMAC by setting the **dmaOwner** to its EP number, and will start a DMA session using **DmaIsaToEntirePkt()**. Eventually, it will see that it is the **dmaOwner** and that the session has completed, at which point it will update its queue pointers and mark the **dmaOwner=0** to signify that another EP may use it. Before starting the DMA session, it will allocate a packet from the MMU and save the packet number in a local static variable, which it will queue for Tx when the DMA session is complete. For the EP2 handler, GPIO4 is raised high when the DMA session is started, and brought back low when it completes; for EP4, no GPIO's are pulsed. Since **GPIOA_OUT** is shared with the ISR, IRQ's are disabled around accesses to this register.

GPIO Summary

The sample code is instrumented as follows:
GPIO7: set high in Ap to connect to USB at full speed (12 Mbps)
GPIO5: pulses high for the duration of ApSlowPoll().
GPIO4: pulses high for the duration of EP2 DMA from ISA to MMU
GPIO3: pulses high for the duration of EP1 DMA from MMU to ISA
GPIO2: pulses high for the duration of Isr0() [see comment below about GPIO2]
GPIO1: pulses high each time around the USB RxFIFO loop
GPIO0: pulses high for a USB Rx pkt on EP1 or 3 bad toggle

All GPIO's are manipulated in the Application code, with the exception of GPIO2. To be faithful to the architecture description in which all GPIO's are owned by the application, this should be removed from the Core code and either placed in the Application or discarded.