



# **Universal Serial Bus (USB) Client Device Validation for the StrongARM™ SA-1100 Microprocessor**

**Application Note**

---

*November 1998*



Information in this document is provided in connection with Intel products. No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted by this document. Except as provided in Intel's Terms and Conditions of Sale for such products, Intel assumes no liability whatsoever, and Intel disclaims any express or implied warranty, relating to sale and/or use of Intel products including liability or warranties relating to fitness for a particular purpose, merchantability, or infringement of any patent, copyright or other intellectual property right. Intel products are not intended for use in medical, life saving, or life sustaining applications.

Intel may make changes to specifications and product descriptions at any time, without notice.

Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them.

The StrongARM may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an ordering number and are referenced in this document, or other Intel literature may be obtained by calling 1-800-548-4725 or by visiting Intel's website at <http://www.intel.com>.

Copyright © Intel Corporation, 1998

\*Third-party brands and names are the property of their respective owners.

ARM and StrongARM are trademarks of Advanced RISC Machines Limited.

# Contents

---

1.0	Introduction.....	5
1.1	Test Description .....	5
1.2	Document Scope.....	5
1.3	References Documents.....	5
1.4	System Configuration .....	6
1.4.1	Required Materials .....	6
1.4.2	2.2 Setup .....	7
1.5	Procedure.....	7
1.5.1	Loading Host Software .....	7
1.5.2	Testing Procedure .....	8
2.0	Description of the UDC Controller code .....	8
3.0	Description of the USB Test Suite.....	21
4.0	SA-1100 Microprocessor Assembly Code.....	29

## Figures

1	SA-1100 USB Controller Test Setup .....	7
2	Endpoint 0 Routine.....	10
3	Endpoint 0 Idle Routine .....	11
4	Set Address Routine .....	12
5	Get Descriptors Routine.....	13
6	Set Descriptors Routine .....	13
7	Endpoint 0 Input Routine.....	14
8	Endpoint 0 Output Routine .....	15
9	Endpoint 0 End Routine .....	16
10	Endpoint 1 Routine (OUT).....	18
11	Endpoint 2 Routine (IN).....	20
12	Sending a Reset Packet.....	21
13	Host Starts Setup Transaction .....	22
14	Assigning the UDC Controller a Specific Address.....	23
15	Ensuring the UDC Controller was able to set its Address .....	23
16	Requesting the GET_DESCRIPTOR Information .....	24
17	Sending Multiple Bulk Data Packets .....	25
18	Testing the Data Toggling Mechanism.....	26
19	Error Recovery from Missing Acknowledgment .....	27
20	Error Recovery from Corrupt Data .....	28

## Tables

1	Reference Documents.....	5
---	--------------------------	---





## 1.0 Introduction

This section provides a description of the tests used for validating the proper operation of Intel's StrongARM™ SA-1100 universal serial bus (USB) device controller and provides a list of related documentation.

## 1.1 Test Description

This document describes a series of tests used for validating the proper operation of the StrongARM SA-1100 USB device controller at the component and application level. In general, the USB devices consist of three components:

- A serial interface engine (SIE), which is implemented in silicon and is responsible for the transmission and reception of USB structured data.
- A hardware and firmware combination responsible for data transfer between the SIE and the device endpoints and their corresponding pipes.
- The third element corresponds to the actual functionality that the device brings to the system, for example, mouse functionality.

These tests confirm the functionality of first two components mentioned above. The first test verifies the operation of the SA-1100's USB registers, interrupt bits, data FIFO's, and reset. The second test verifies that the USB controller can be configured, transfer bulk data packets, and perform multiple transactions.

## 1.2 Document Scope

This document details the procedures that are performed to test the functionality of the USB controller on the SA-1100, called the UDC. Any required equipment, along with the setup of the equipment, is listed with the test procedure. This document also contains the assembly language and flow charts used to control the SA-1100 microprocessor.

## 1.3 References Documents

Other documents that may be helpful while reading this document are described in the following table:

Table 1. Reference Documents

Title	Web Address
<i>Universal Serial Bus Revision 1.1</i>	<a href="http://www.usb.org">http://www.usb.org</a>
<i>Universal Serial Bus System Architecture</i>	<a href="http://www.mindshare.com/html/list_of_books.html">http://www.mindshare.com/html/list_of_books.html</a>
StrongARM™ SA-1100 Microprocessor Technical Reference Manual	<a href="http://developer.intel.com">http://developer.intel.com</a>
<i>ARM Software Development Toolkit User Guide</i>	<a href="http://www.arm.com">http://www.arm.com</a>
<i>Traffic Generator</i>	<a href="http://www.catc.com">http://www.catc.com</a>
<i>Inspector</i>	<a href="http://www.catc.com">http://www.catc.com</a>

## 1.4 System Configuration

This section describes the required hardware and software, and an overview of the SA-1100 USB test setup.

### 1.4.1 Required Materials

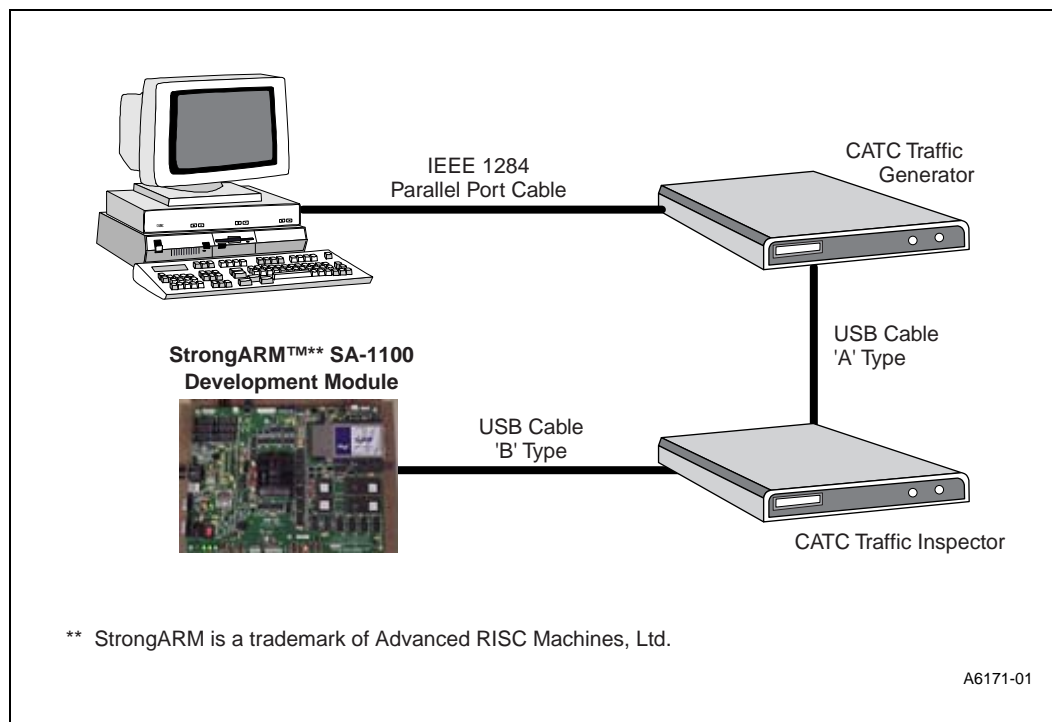
The following hardware and software are required for testing the StrongARM SA-1100 USB device controller:

- Personal Computer with IEEE1284 bidirectional parallel port card
  - CPU - Pentium® processor or Pentium® II processor
  - CPU speed - 100 Megahertz or greater
  - RAM - 8 Megabytes or greater
  - Hard drive space - 200 Kilobytes or greater
- Windows\* 3.1, Windows95,\* or Windows NT 4.0\* operating system
- SA-1100 Development Board (DE-1S110-OA) with Angel 1.05 ROMs
- *ARM Software Development Toolkit* (Version 2.11a)
- *CATC Traffic Generator* (Version 2.0)
- *CATC Inspector Advanced USB Bus & Protocol Analyzer* (Version 2.2)
- USB Cable (A-type)
- USB Cable (B-type)

## 1.4.2 2.2 Setup

The test environment must be configured as shown in Figure 1 with the bi-directional parallel port of the USB Traffic Generator connected to the IEEE 1284 parallel port card of the PC. The USB Traffic Generator is connected to the USB Inspector via the A-type USB Cable. The USB Inspector is then connected to the SA-1100 development board via the B-type USB Cable. The CATC Inspector does not cause any interference to the bus traffic because it is transparent to the network.

**Figure 1.** SA-1100 USB Controller Test Setup



## 1.5 Procedure

The following sections describe the procedures for loading the host software and testing the SA-1100 USB device controller.

### 1.5.1 Loading Host Software

Use the following procedure to load the host software:

1. Turn on the PC and install the CATC traffic generator software. This software sends USB packets across the universal serial bus.
2. Install the CATC Inspector software. This software provides a visual inspection of the USB traffic.
3. Install the ARM™ software development toolkit. This toolkit runs software on the SA-1100 that controls data transfers to and from the USB device controller on the SA-1100.

## 1.5.2 Testing Procedure

Use the following procedure for testing:

1. Start the ARM SDT Project Manager.
  - a. Open the project called “udc\_lab.apj”.
  - b. Compile the project by clicking on the Force Build icon.
  - c. Execute the project by clicking the Execute icon—this will download the software image to DRAM on the SA-1100 development board and begin running the UDC controller code.
2. Start the CATC Inspector.
  - a. Select the Recording Options... menu item from the SETUP menu, then select the trigger option to be the Event Trigger.
  - b. Click on the Setup... button when it becomes highlighted. Select the Frame Number option and enter 1 in the Frame # box—this will cause the CATC Inspector to trigger on the Start Of Frame packet #1 and capture the test packets that follow.
  - c. Click OK to get back to the main menu.
  - d. Click the RECORD button to begin recording USB traffic.
3. Start the CATC Traffic Generator.
  - a. Click on the File Open icon and open up the traffic data file called “test\_usb\_sa1100.gen”.
  - b. Click on the Generator button, and select the Download menu (download all possible packets in Memory Partition #0 and then exit).
  - c. Select the Playback menu item from under the Generator button.
  - d. Press the Start button—this will start the USB test suite of packets.
4. Compare the results captured by the CATC Inspector with the proper results shown below in Section 3.0.

## 2.0 Description of the UDC Controller code

This section describes the major portions of the assembly code that makes up the project called “udc\_lab.apj.”

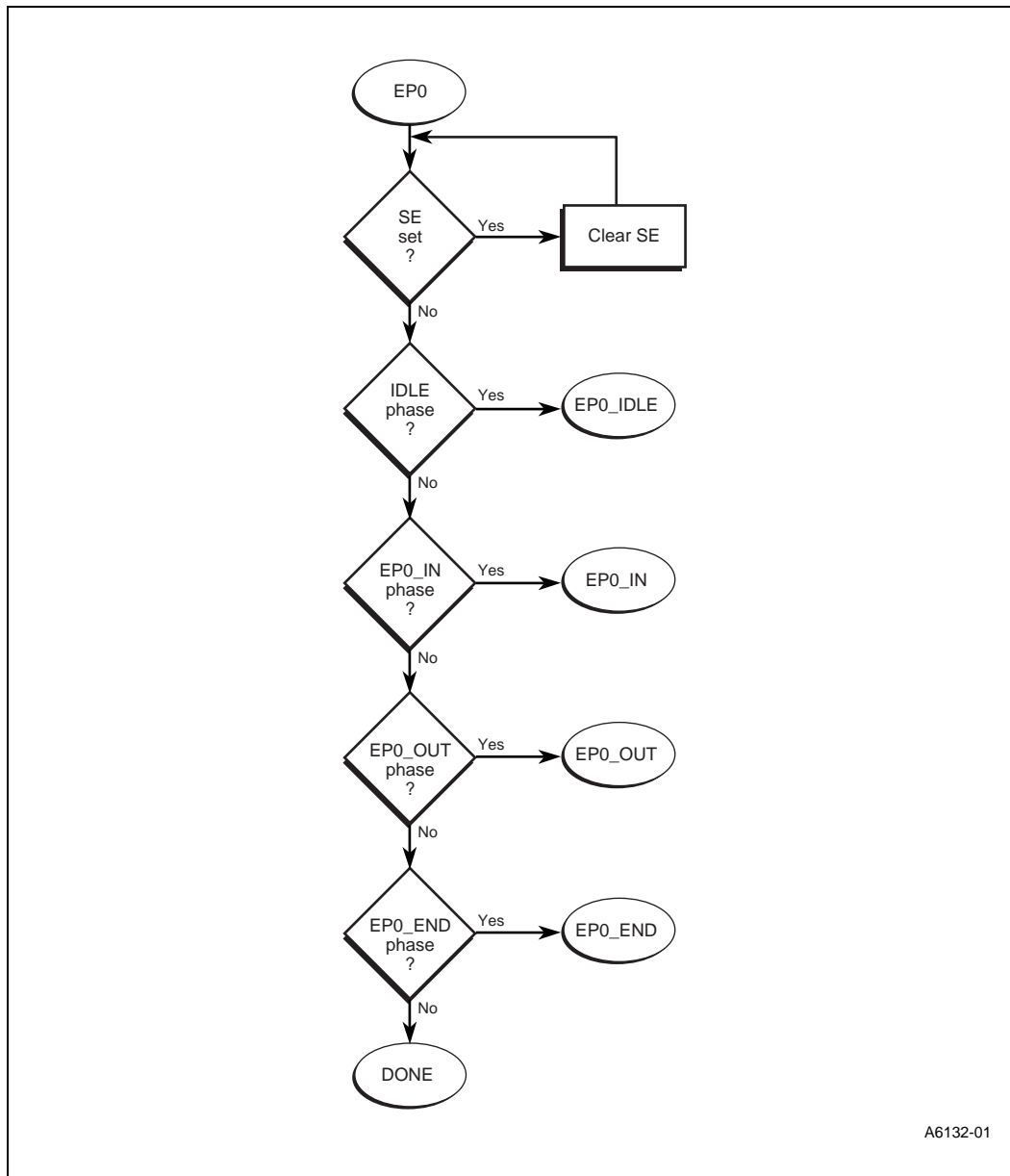
1. **Initialization of the UDC:** In this portion of the code, the program resets the UDC, which ensures that the USB Device Controller is initialized to the proper state. The UDC is disabled and then re-enabled, which confirms that the UDC can be paused by software control. The Max\_Packet registers is set, which holds the value of the maximum number of bytes of data per packet that can be transferred to and from the UDC core.
2. **Initialization of the DMAs:** Within the DMA Controller of the SA-1100, DMA0 is configured to receive data (data is moved from the USB into a receive FIFO within the UDC). The DMA0 moves data from the receive FIFO to memory, where the SA-1100 core processes the data. This movement of data is called an *OUT* transaction since, from the view of the USB host, data is ultimately sent from the USB host out to the USB client. Also within the DMA Controller of the SA-1100, DMA1 is configured to transmit data (DMA1 moves data from memory to the transmit FIFO within the UDC, where it will subsequently be moved to the USB). This movement of data is called an *IN* transaction since, from the view of the USB host,

data is ultimately sent into the USB host from the USB client.

After initialization, the program polls the UDC Service Request bit in the SA-1100 Interrupt Controller Pending Register located in the System Control Module. Once the UDC service request bit is seen, program flow jumps to one of the three endpoints routines described in steps a, b, and c:

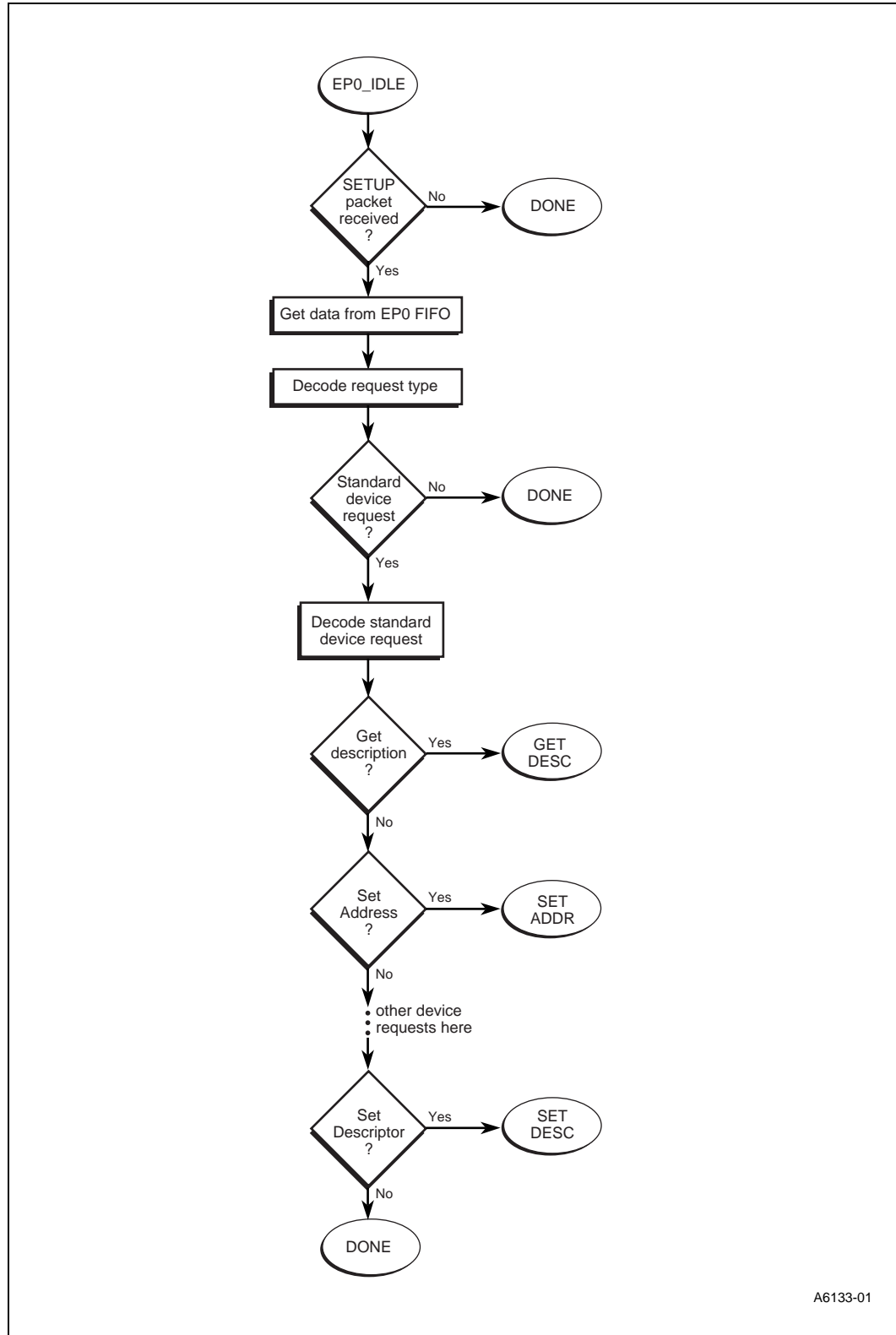
- a. **Endpoint 0 routine (Control/Status):** Once the program determines that an Endpoint 0 interrupt has occurred, a software state machine is used to decide what to do. If the state machine is in the idle state and a setup packet is received, the program parses the packet to determine which standard device made the request. If the task is to request information from the device, the program will enter the Endpoint 0 IN data phase of the state machine. If the task is to send information to the device, the program will update the state machine to enter the Endpoint 0 OUT data phase. During the IN data phase, the setup information is put into the Endpoint 0 bi-directional FIFO and sent to the host when requested. During the OUT data phase, any data received in the Endpoint 0 bi-directional FIFO will be gathered and parsed and handled appropriately. The last phase of the state machine is the End Transfer phase which configures the proper status and control bits and proper handshaking.

Figure 2. Endpoint 0 Routine



A6132-01

Figure 3. Endpoint 0 Idle Routine



A6133-01

Figure 4. Set Address Routine

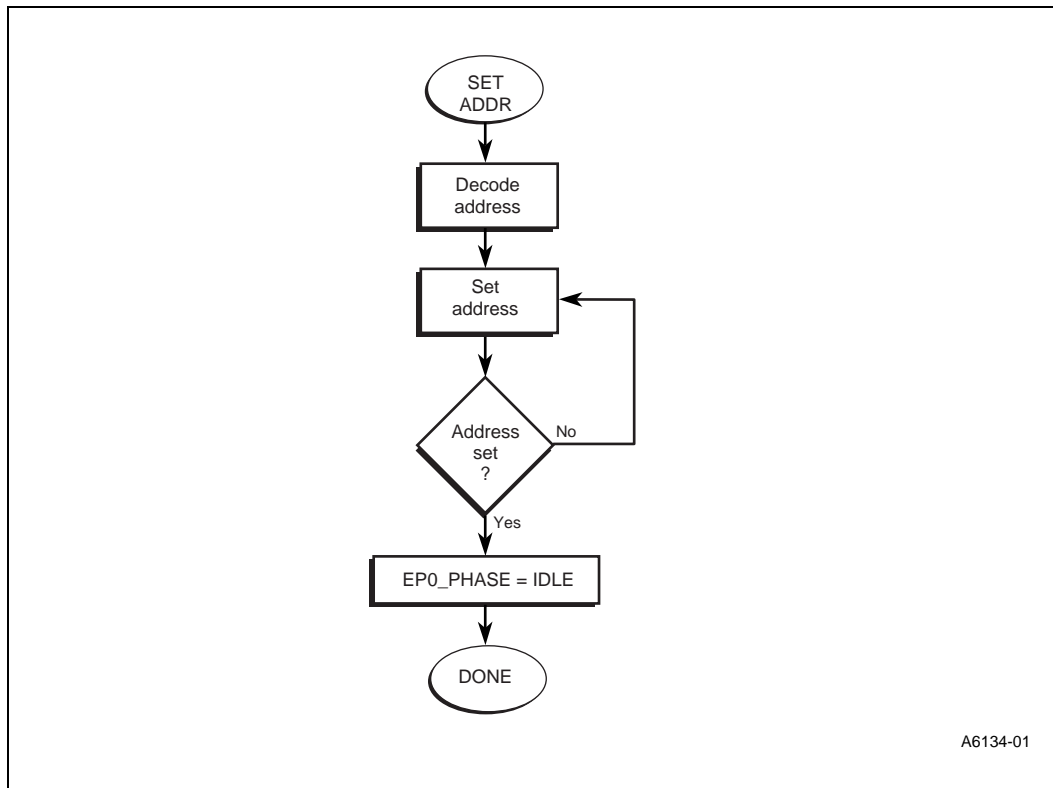


Figure 5. Get Descriptors Routine

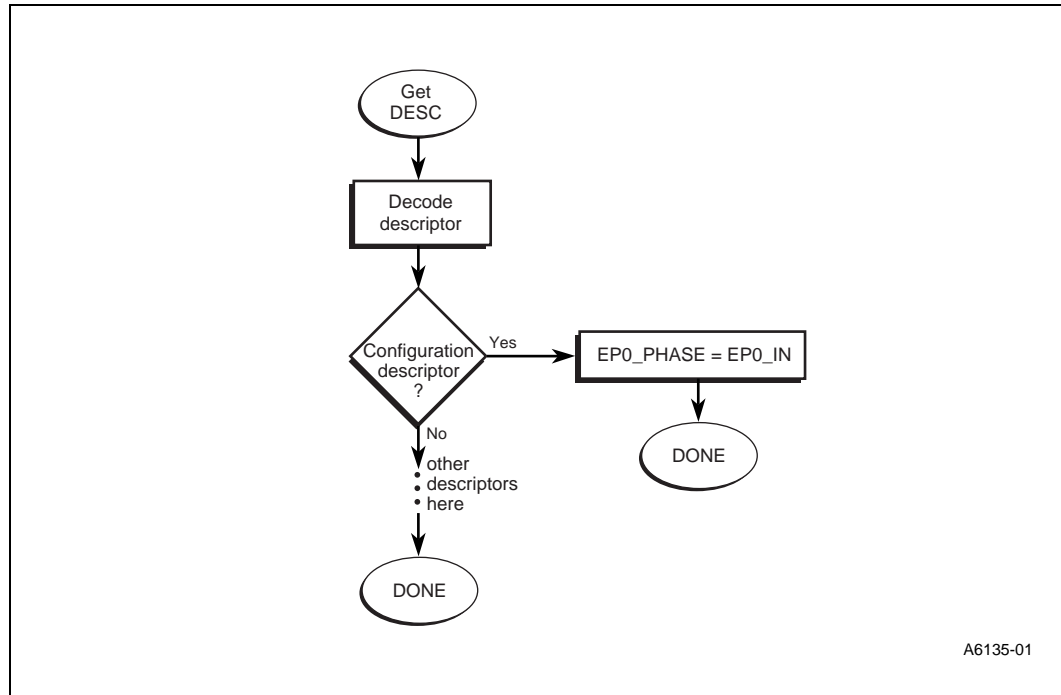


Figure 6. Set Descriptors Routine

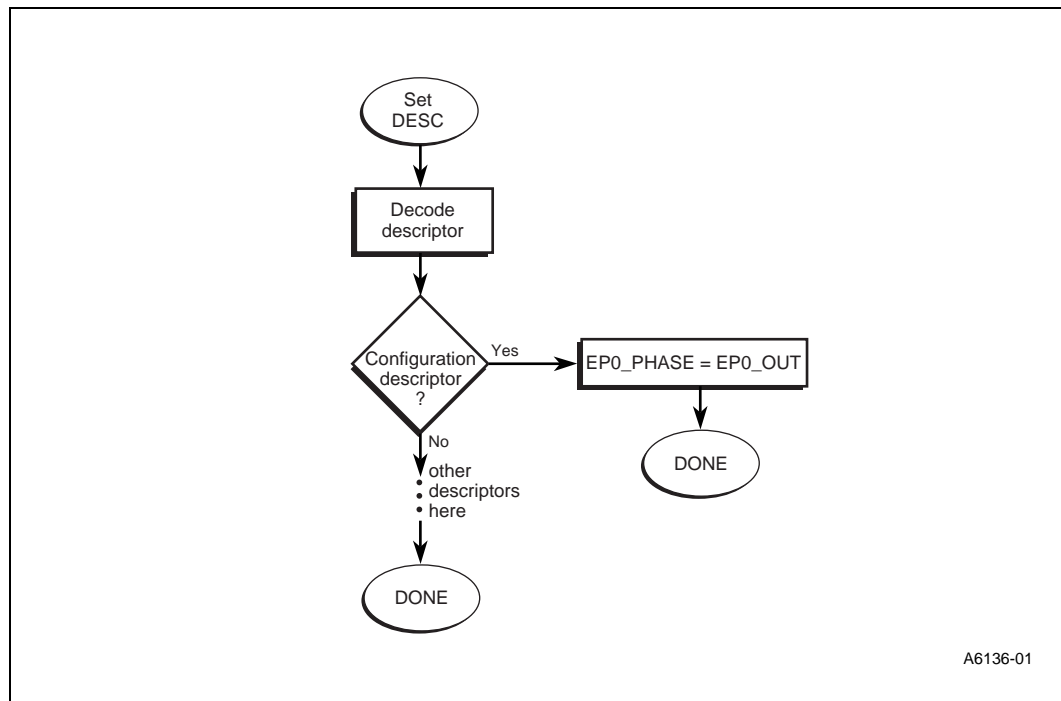


Figure 7. Endpoint 0 Input Routine

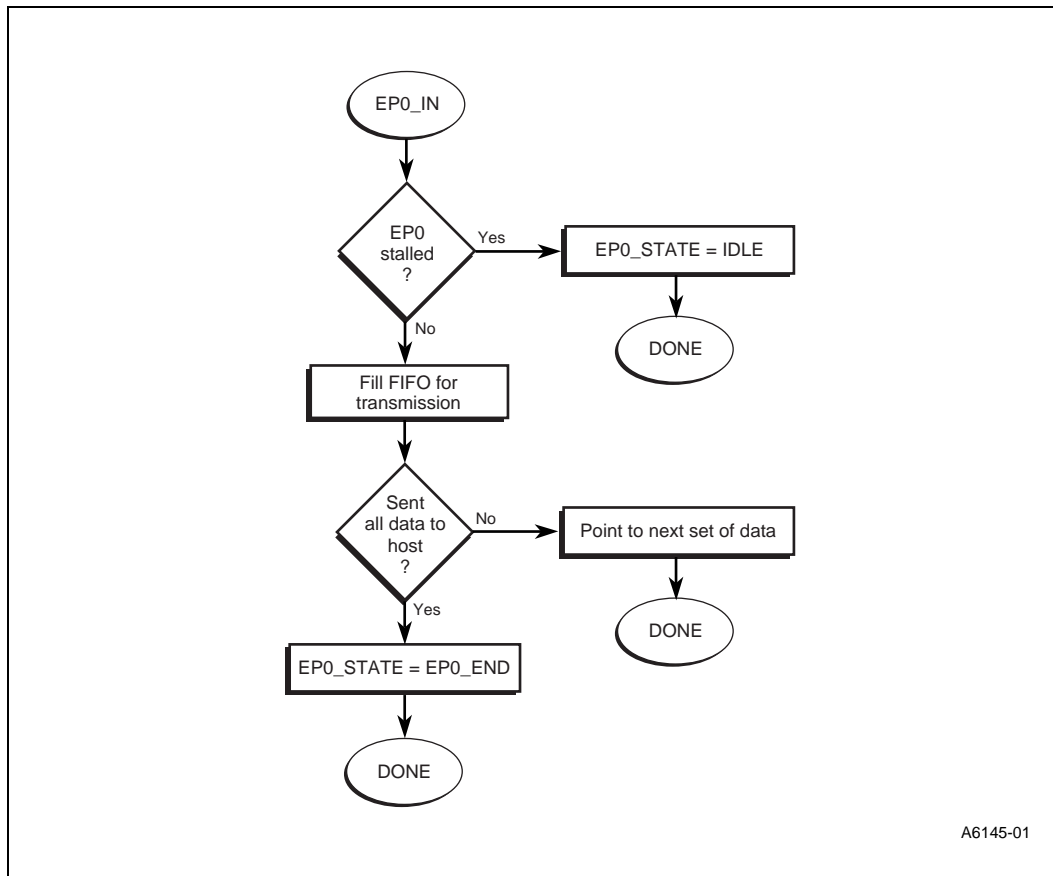


Figure 8. Endpoint 0 Output Routine

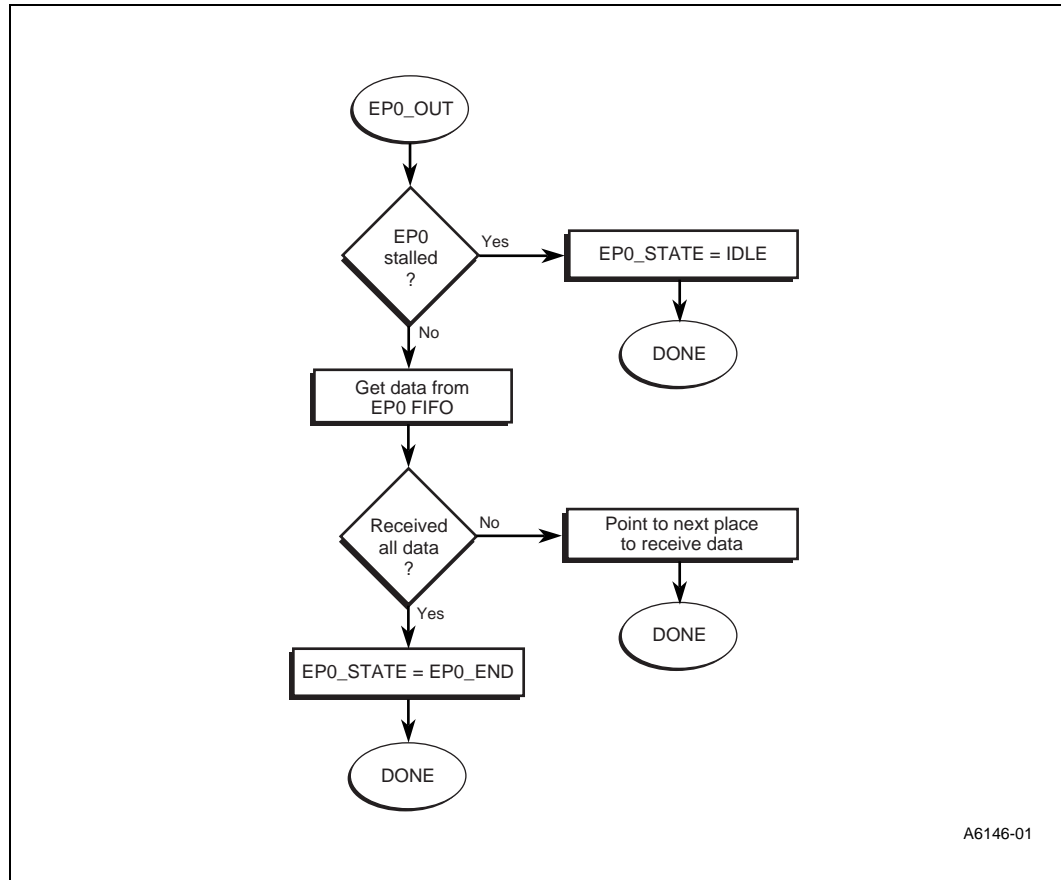
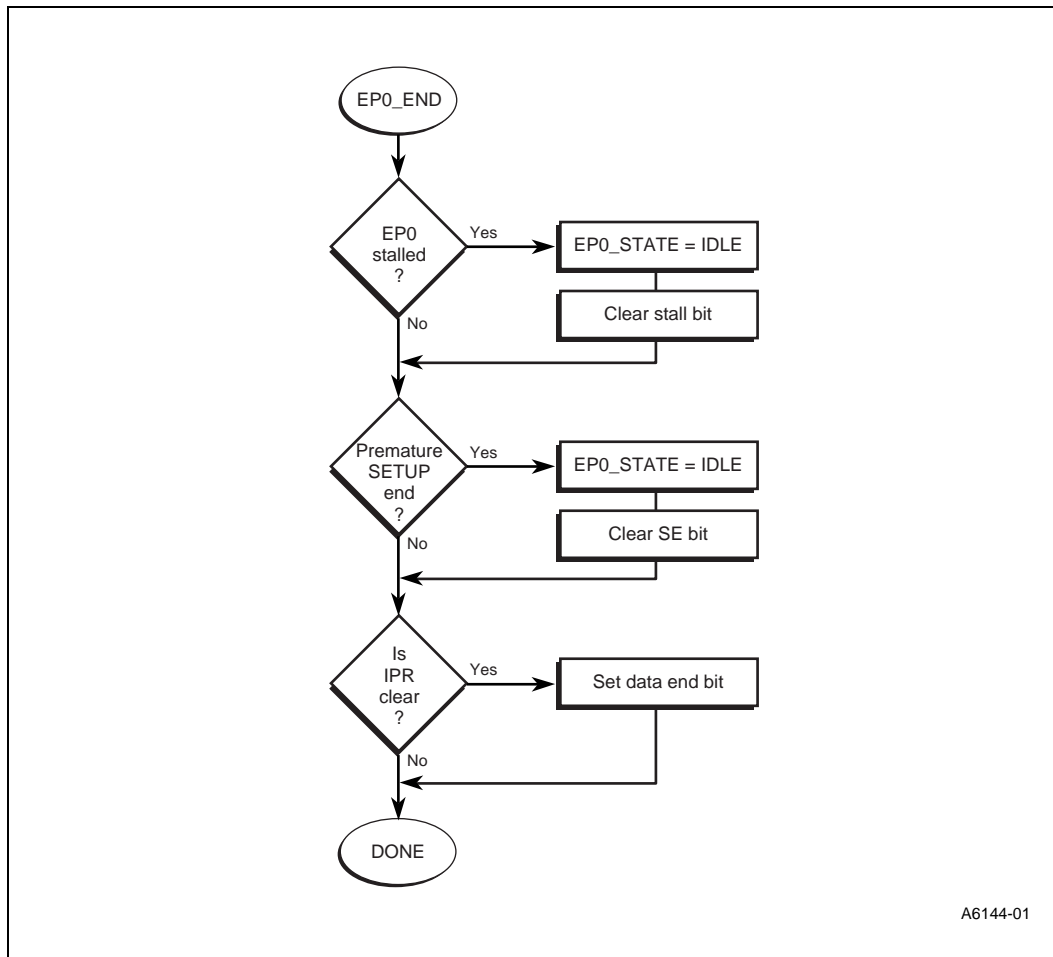


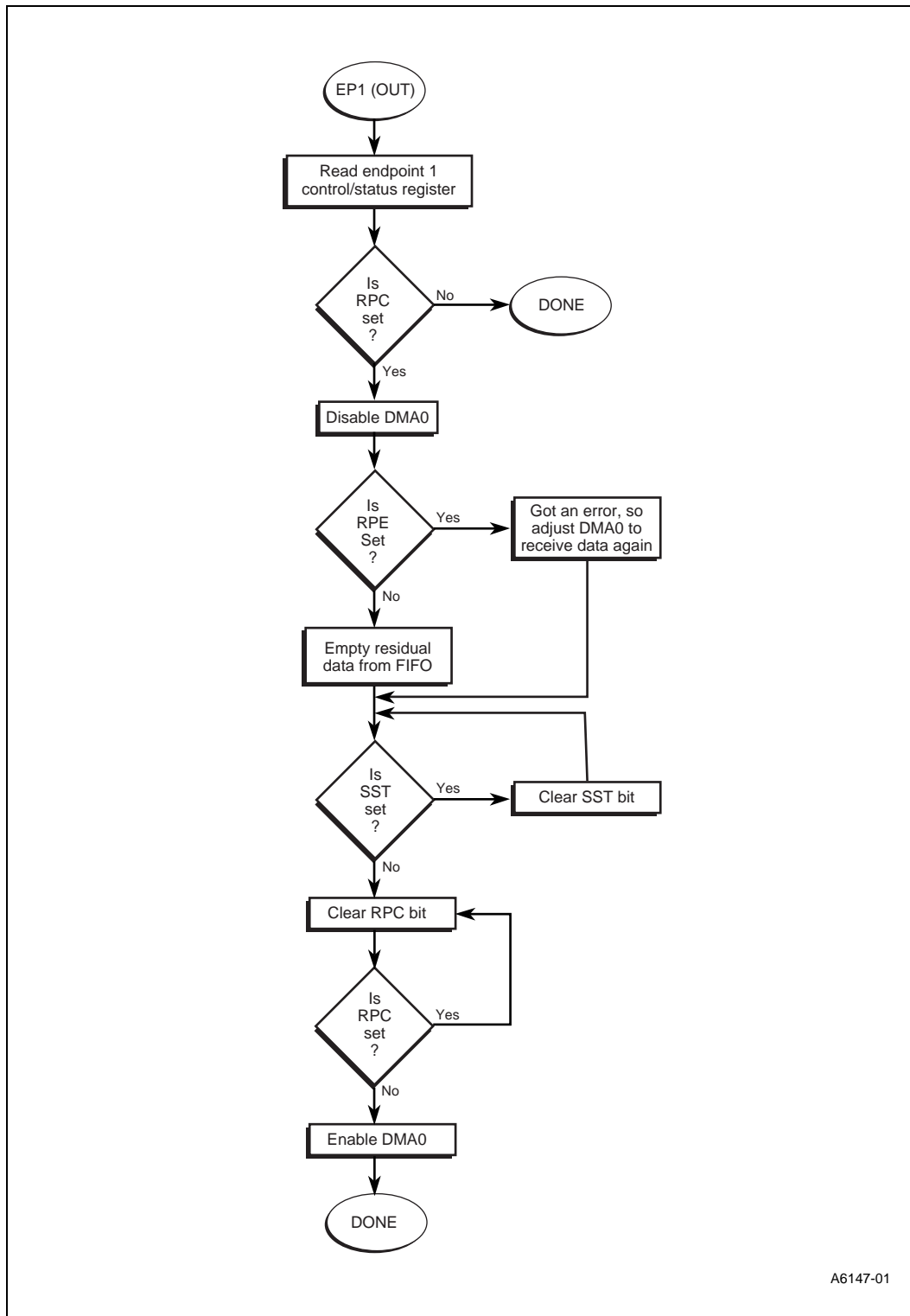
Figure 9. Endpoint 0 End Routine



A6144-01

- b. **Endpoint 1 routine (OUT):** Once the program determines that an Endpoint 1 interrupt has occurred, the RPC bit is checked to see if a data packet has been received and if the error/status bits are valid. If the data packet has errors, the program ignores the received data and prepares to receive the data again. If the data packet does not have any errors, then any residual data that the DMA0 did not service is gathered from the Receive FIFO and put into memory. DMA0 is adjusted to point to a new storage location to receive the next packet whether it is the old packet of data again or a new packet of data.

Figure 10. Endpoint 1 Routine (OUT)

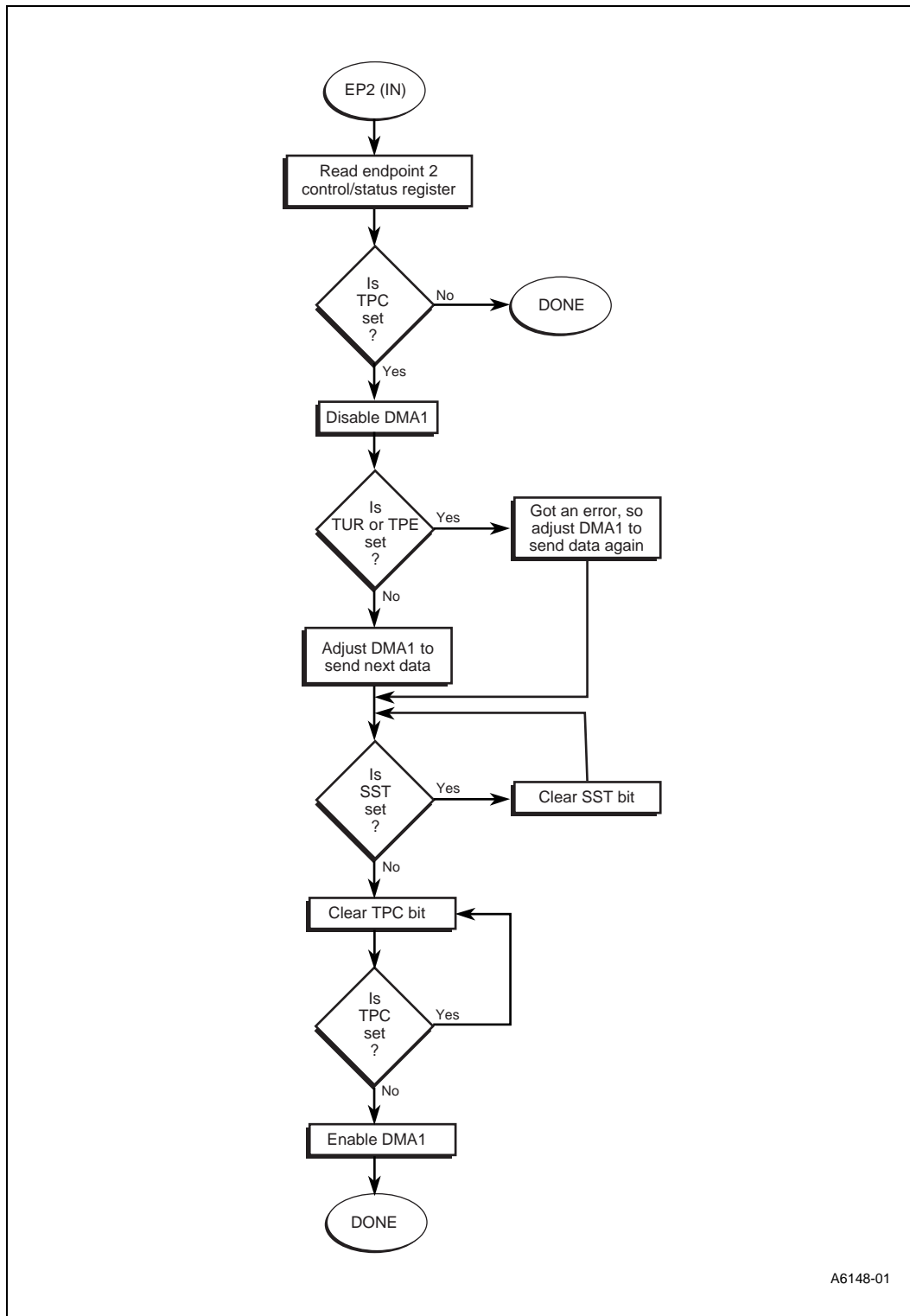


A6147-01



- c. **Endpoint 2 Routine (IN):** Once the program determines that an Endpoint 2 interrupt has occurred, the TPC bit is checked to see if a data packet has been transmitted and the error/status bits are valid. If the data packet has errors, the program must adjust DMA1 to resend the packet again. If the data packet does not have any errors, then DMA1 is adjusted to point to the next data packet to be sent.

Figure 11. Endpoint 2 Routine (IN)



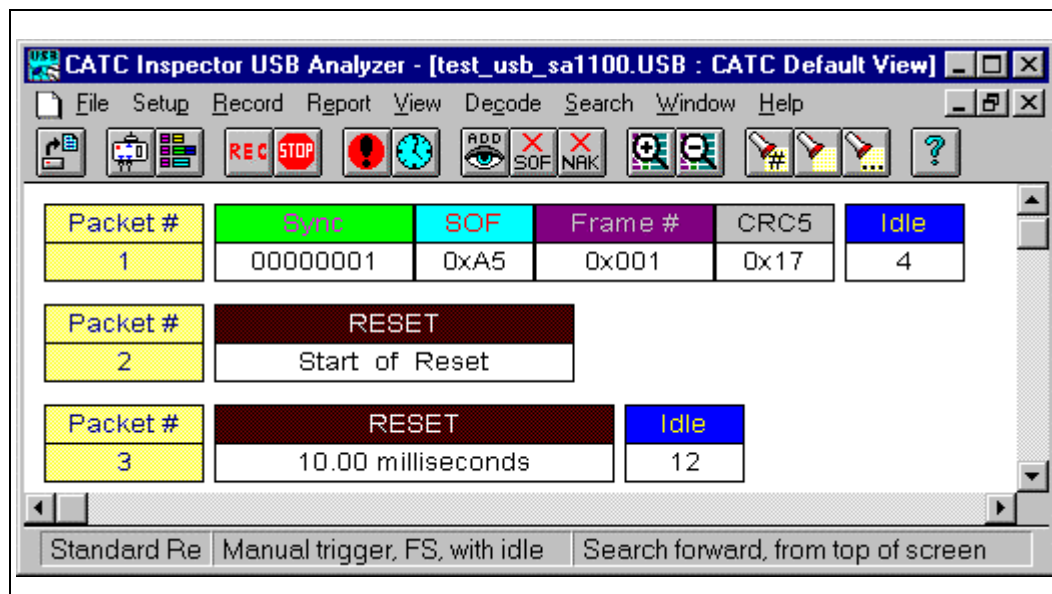
A6148-01

### 3.0 Description of the USB Test Suite

This section describes the USB test packets in the file called test\_usb\_sa1100.gen .

1. In this test, the host sends a reset packet for 10 milliseconds. The reset packet pulls both the UDC+ and UDC- pins low for more than 2.5 microseconds. The reset is shown in packets #1 through packet #3.

Figure 12. Sending a Reset Packet



2. In this test, the host starts a setup transaction, beginning with packet #6, to the device. The address is 0, which is the default address of an uninitialized USB device, and the endpoint is 0, which is the setup/control endpoint. Packet #7 is decoded as the GET\_DESCRIPTOR device request, where the host requests information about the USB device. The device responds by describing specific functional information about itself, such as a mouse, keyboard, or storage device. This ensures that the UDC Controller’s can receive setup data in the Endpoint 0 FIFO and tests the control/status bits of the Endpoint 0 Control/Status Register. The UDC controller sends Packet #10 back to the host with the first 8 bytes providing descriptor information. Packet #13 is the 9th byte of the descriptor information sent to the host. These two packets test the UDC Controller’s ability to load device information into the Endpoint 0 FIFO multiple times. Finally, packets #15 through packet #17 are the handshake transaction for the device request transfer.

Figure 13. Host Starts Setup Transaction

Packet #	Sync	Setup	ADDR	ENDP	CRC5	Idle
6	00000001	0xB4	0x00	0x0	0x08	4
Packet #	Sync	DATA0	DATA		CRC16	Idle
7	00000001	0xC3	80 06 00 02 00 00 09 00		0x7520	5
Packet #	Sync	ACK	Idle			
8	00000001	0x4B	2178			
Packet #	Sync	IN	ADDR	ENDP	CRC5	Idle
9	00000001	0x96	0x00	0x0	0x08	5
Packet #	Sync	DATA1	DATA		CRC16	Idle
10	00000001	0xD2	09 02 2E 00 01 01 00 80		0xD031	8
Packet #	Sync	ACK	Idle			
11	00000001	0x4B	301			
Packet #	Sync	IN	ADDR	ENDP	CRC5	Idle
12	00000001	0x96	0x00	0x0	0x08	5
Packet #	Sync	DATA0	DATA	CRC16	Idle	
13	00000001	0xC3	32	0x8356	9	
Packet #	Sync	ACK	Idle			
14	00000001	0x4B	16			
Packet #	Sync	OUT	ADDR	ENDP	CRC5	Idle
15	00000001	0x87	0x00	0x0	0x08	4
Packet #	Sync	DATA1	DATA	CRC16	Idle	
16	00000001	0xD2		0x0000	5	
Packet #	Sync	ACK	Idle			
17	00000001	0x4B	18			

Standard Requests      Manual trigger, FS, with idle      Search forward, from top of screen

- The next phase of the test assigns the UDC controller a specific address. Packets #19 through packet #24 are assigned the address of 0x55 with a handshake to acknowledge the transaction.

Figure 14. Assigning the UDC Controller a Specific Address

Packet #	Sync	TYPE	ADDR	ENDP	CRC5	Idle	
19	00000001	SETUP	0xB4	0x00	0x0	0x08	4
20	00000001	DATA0	0xC3	00 05 55 00 00 00 00 00		0x678F	5
21	00000001	ACK	0x4B				2078
22	00000001	IN	0x96	0x00	0x0	0x08	6
23	00000001	DATA1	0xD2		0x0000		9
24	00000001	ACK	0x4B				4

- The next series of tests ensures that the UDC controller is able to set its address and ignore any USB traffic that is not specifically addressed to it. Packets #26 through packet #30 tests that the UDC ignores a setup, IN, and OUT transaction all to address 0 and endpoint 0. Packets #31 through packet #33 test that the UDC ignores an IN and OUT transaction to address 0 and endpoint 2 and endpoint 1, respectively.

Figure 15. Ensuring the UDC Controller was able to set its Address

Packet #	Sync	TYPE	ADDR	ENDP	CRC5	Idle	
26	00000001	SETUP	0xB4	0x00	0x0	0x08	4
27	00000001	DATA0	0xC3	FF FF FF FF FF FF FF FF		0x7F0E	185
28	00000001	IN	0x96	0x00	0x0	0x08	31
29	00000001	OUT	0x87	0x00	0x0	0x08	4
30	00000001	DATA0	0xC3	FF FF FF FF FF FF FF FF		0x7F0E	31
31	00000001	IN	0x96	0x00	0x2	0x1C	31
32	00000001	OUT	0x87	0x00	0x1	0x05	4
33	00000001	DATA0	0xC3	FF FF FF FF FF FF FF FF		0x7F0E	31

- This test verifies the GET\_DESCRIPTOR information with the address to which the UDC Controller was assigned, which is 0x55. This test is contained within packets #35 through packet #46.

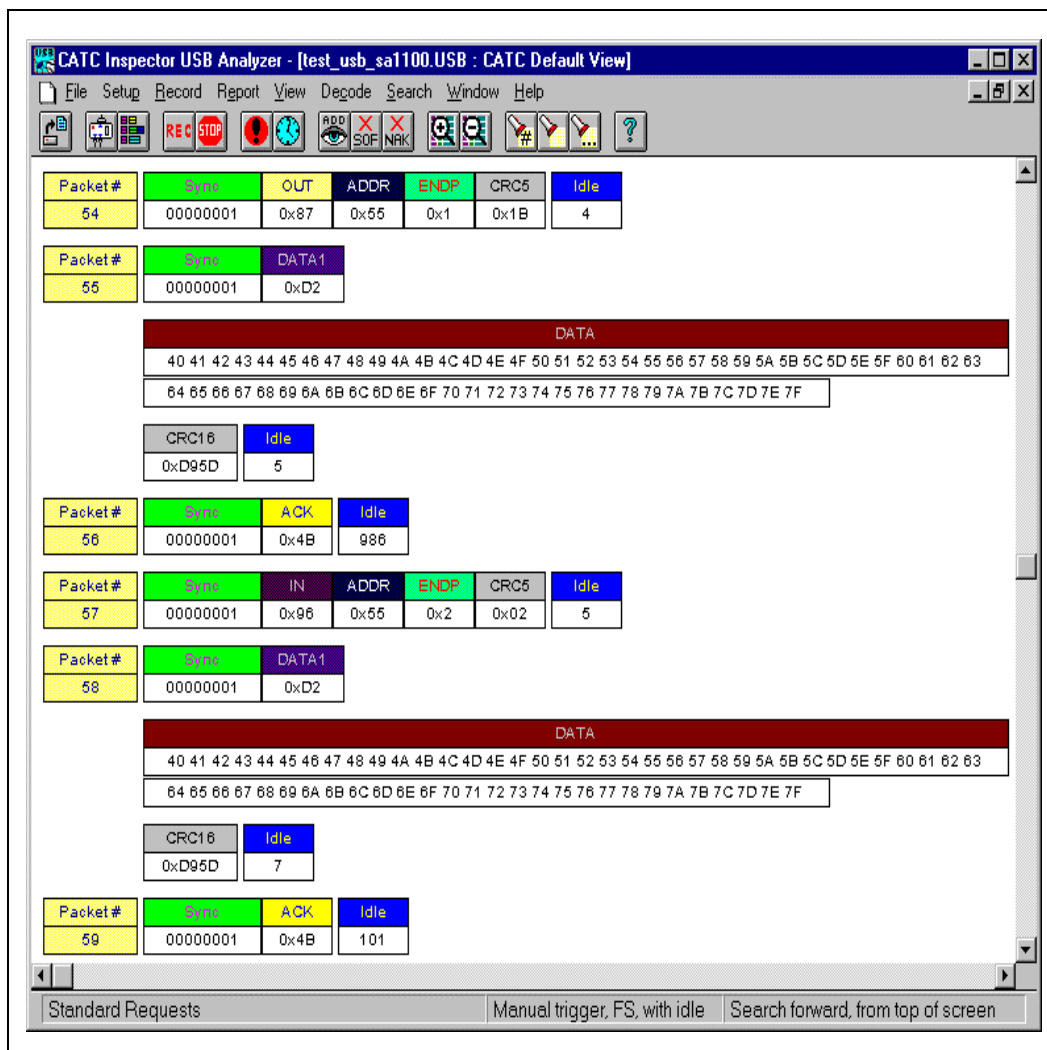
Figure 16. Requesting the GET\_DESCRIPTOR Information

Packet #	Sync	SETUP	ADDR	ENDP	CRC5	Idle
35	__000001	0xB4	0x55	0x0	0x16	4
Packet #	Sync	DATA0	DATA	CRC16	Idle	
36	00000001	0xC3	80 06 00 02 00 00 09 00	0x7520	5	
Packet #	Sync	ACK	Idle			
37	00000001	0x4B	2978			
Packet #	Sync	IN	ADDR	ENDP	CRC5	Idle
38	00000001	0x96	0x55	0x0	0x16	5
Packet #	Sync	DATA1	DATA	CRC16	Idle	
39	00000001	0xD2	09 02 2E 00 01 01 00 80	0xD031	8	
Packet #	Sync	ACK	Idle			
40	00000001	0x4B	301			
Packet #	Sync	IN	ADDR	ENDP	CRC5	Idle
41	00000001	0x96	0x55	0x0	0x16	7
Packet #	Sync	DATA0	DATA	CRC16	Idle	
42	__000001	0xC3	32	0x8356	10	
Packet #	Sync	ACK	Idle			
43	__000001	0x4B	4			
Packet #	Sync	OUT	ADDR	ENDP	CRC5	Idle
44	00000001	0x87	0x55	0x0	0x16	4
Packet #	Sync	DATA1	DATA	CRC16	Idle	
45	00000001	0xD2	0x0000	0x0000	5	
Packet #	Sync	ACK	Idle			
46	00000001	0x4B	18			

- This test sends multiple bulk data packets to the UDC Controller and verifies that they were received correctly by transmitting the data back to the host. In packets #48 through packet #50, the host sends 64 bytes of data to the UDC Controller. This tests the receive FIFO operation as well as the status/control bits of the Endpoint 1 status/control register. Packets #51 through packet #53 request data from the UDC Controller by the host. This tests the transmit FIFO operation and the status/control bits of the Endpoint 2 status/control register.

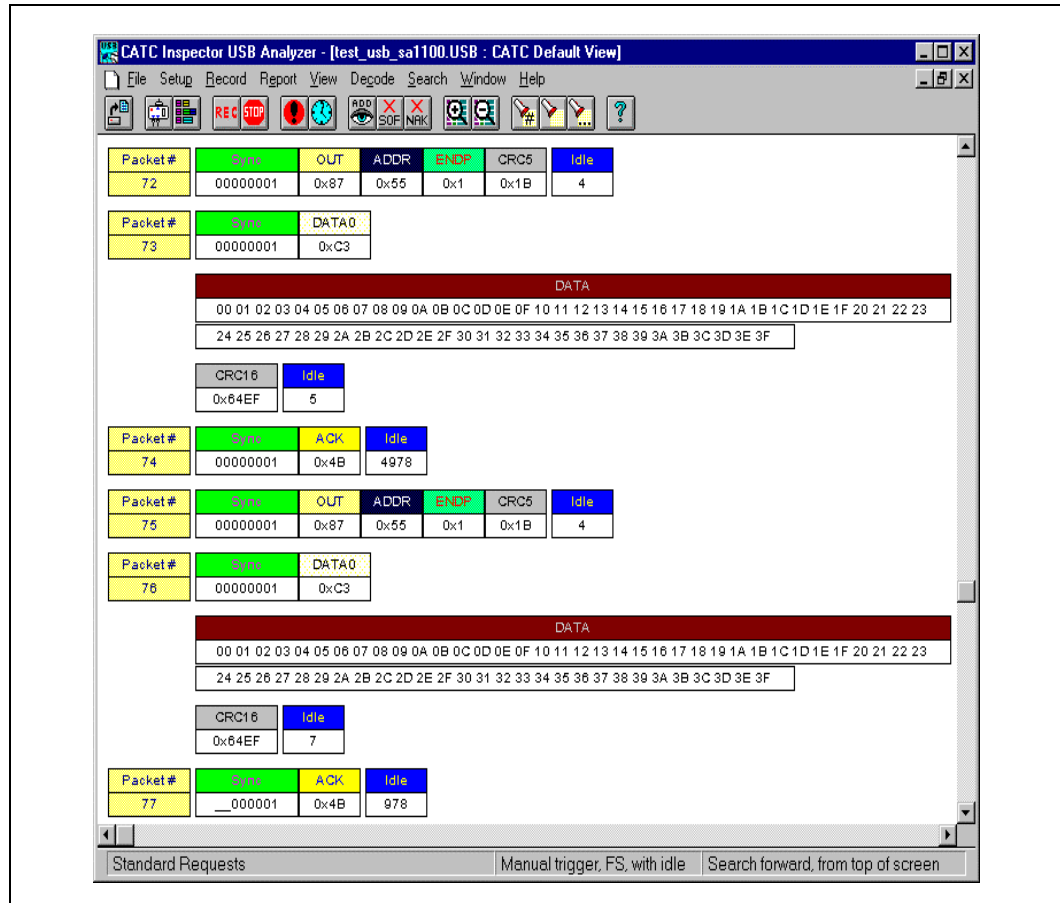


Figure 18. Testing the Data Toggling Mechanism



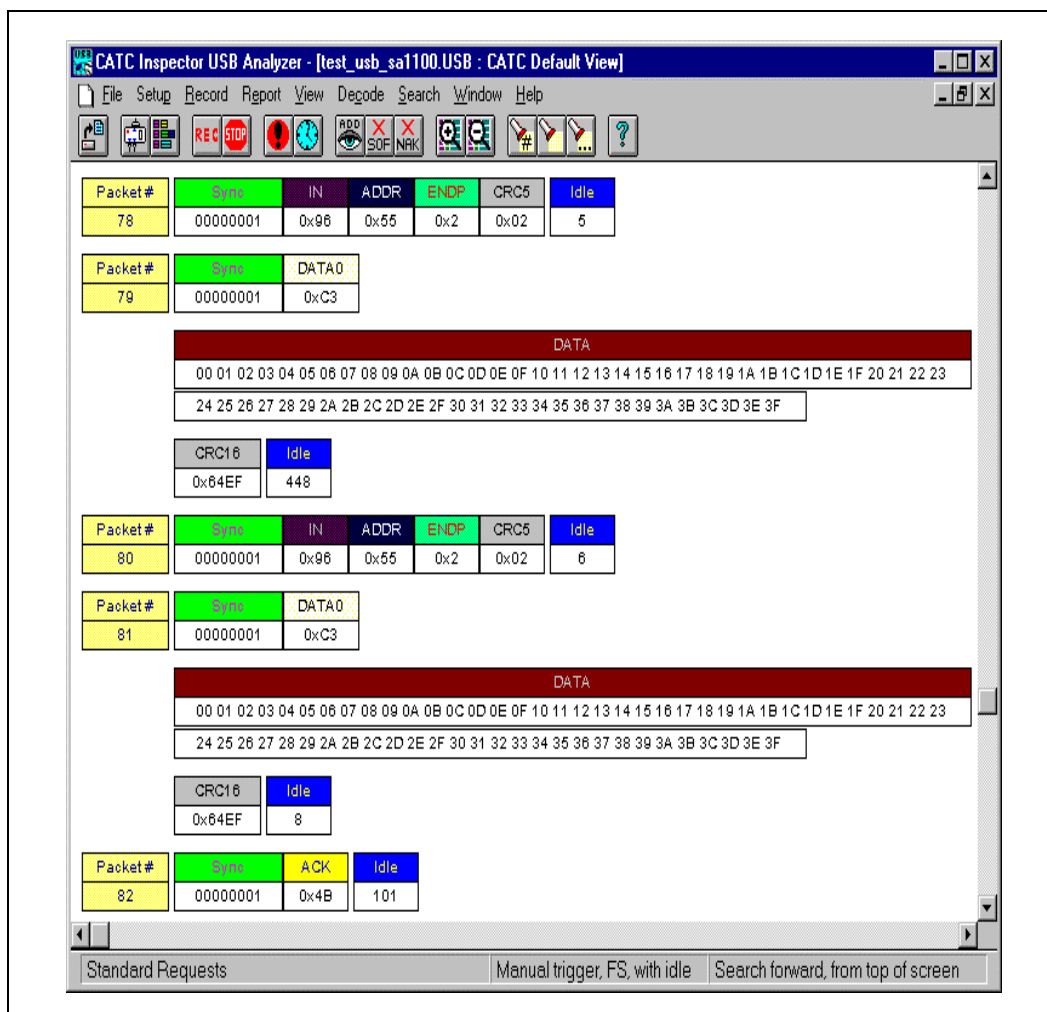
- This transaction tests the USB controller’s ability to handle error recovery from a simulated missing acknowledgment from the device to the host. In packets #72 through packet #74, the host sends data to the device, however, even though the device sends back an acknowledgment in packet #74, the host ignores it. This simulates a missing or corrupt handshake, and the host sends the data again with a packet identifier of DATA0 again. Once the UDC senses the DATA0 packet identifier again, it disregards the data packet, since the device has already received the data without error, and issues another ACK handshake in packet #77.

Figure 19. Error Recovery from Missing Acknowledgment



9. Packets #78 through #82 test the error recovery mechanism when the host does not acknowledge sent data. This test simulates corrupt data received by the host. The host requests the data again in packet #80 and the device recognizes this by the packet identifier being DATA0 again, instead of normally toggling to DATA1.

Figure 20. Error Recovery from Corrupt Data



## 4.0 SA-1100 Microprocessor Assembly Code

```

;-----
; test code for the USB Controller on the SA-1100
; 10/21/98
;-----

        AREA |udc_lab|, CODE, READWRITE

        ENTRY

;-----
; define variables
;-----

        MAX_IN_PKT_MINUS1      EQU      0x3F
        MAX_OUT_PKT_MINUS1     EQU      0x3F

        EPO_IDLE                EQU      0
        EPO_IN_DATA_PHASE      EQU      1
        EPO_OUT_DATA_PHASE     EQU      2
        EPO_END_XFER           EQU      3

        DDAR                    EQU      0x0
        DCSR                    EQU      0x4
        CLEAR                   EQU      0x8
        READO                   EQU      0xc
        DBSA                    EQU      0x10
        DBTA                    EQU      0x14
        DBSB                    EQU      0x18
        DBTB                    EQU      0x1c

        DMA_OUT_COUNT_MAX      EQU      0x400
        DMA_IN_COUNT_MAX       EQU      0x400

;-----
; Initialization
;-----

        bl      udc_rst          ; init the UDC by reset

; DMA0 init
        MOV     r0, #0xb0000000    ; DMA0 base address
        MOV     r1, #0x0000002b
        STR     r1, [r0, #CLEAR]   ; Disable DMA channel 0
        MOV     r1, #0x80000000    ; base of UDC
        ADD     r1, r1, #0x00000a00 ; address of TX/RX FIFO's
        ADD     r1, r1, #0x00000015 ; device=1,read(dev to mem)
    
```



```

8 byte burst
    STR    r1, [r0, #DDAR]        ; set up DMA0 for UDC read

; DMA1 init
    MOV    r0, #0xb0000000        ; DMA1 base address
    ADD    r0, r0, #0x00000020    ; offset for DMA1
    MOV    r1, #0x0000002b        ; Disable DMA channel 1
    STR    r1, [r0, #CLEAR]
    MOV    r1, #0x80000000
    ADD    r1, r1, #0x00000a00
    ADD    r1, r1, #0x00000004    ; device=0,write(mem to dev)
8 byte burst
    STR    r1, [r0, #DDAR]        ; set up DMA1 for UDC write

; Set up pointers to TX test data and RX memory

    MOV    r2, #DMA_OUT_COUNT_MAX
    MOV    r0, #0xb0000000        ; DMA0 set to receive data
(OUT)
    ADD    r0, r0, #0x00000000    ; offset for DMA0
    LDR    r1, =MBASE             ; address of RAM buffer
    ADD    r1, r1, #0xc0000000    ; Create Physical address
    STR    r1, [r0, #DBSA]        ; Start address = MBASE
    STR    r2, [r0, #DBTA]        ; set OUT Xfer count to Max

    MOV    r2, #DMA_IN_COUNT_MAX
    MOV    r0, #0xb0000000        ; DMA1 set to transmit data
(IN)
    ADD    r0, r0, #0x00000020    ; offset for DMA1
    LDR    r1, =MBASE             ; address of RAM buffer
    ADD    r1, r1, #0xc0000000    ; Create Physical address
    STR    r1, [r0, #DBSA]        ; Start address = MBASE
    STR    r2, [r0, #DBTA]        ; set IN Xfer count to Max

    MOV    r1, #0x11              ; Turn on DMA machines
    MOV    r0, #0xb0000000
    STR    r1, [r0, #0x04]        ; Turn on receive DMA - DMA0 (OUT)

; Once you have started (primed) the XMIT FIFO, there is no way to flush
; those primed bytes out except by UDC reset or by actual transmission on the USB!
; so don't turn on DMA1 until a data packet has been received.
;    STR    r1, [r0, #0x24]        ; Turn on transmit DMA - DMA1 (IN)

    bl     initLED
    mov    r0, #0xA               ; write an 'A' to the LED
    bl     writeLED

```



```
-----  
; main loop  
-----  
  
UDCL0    MOV     r0, #0x90000000  
         ADD     r0, r0, #0x00050000  
         LDR     r1, [r0, #0x20]  
         TST     r1, #0x00002000      ; Look for UDC interrupt pending  
         BNE     udcsvc  
         ldr     r1, =OUTCOUNT      ; get OUT count address  
         ldr     r2, [r1]             ; get OUT packet count  
         TEQ     r2, #8               ; Got 8 OUT packets yet?  
         MOVEQ   r2, #0              ; If so, reset counter  
         STREQ   r2, [r1]  
         BLEQ   pktcheck             ; If so, go check packets  
         B       UDCL0               ; otherwise, loop  
  
         SWI     0x11                 ; Finished  
  
-----  
; UDC Interrupt Service Routine  
-----  
  
udcsvc   STMEA   r13!, {r0-r7}  
         MOV     r8, #0x80000000      ; Load base of UDC  
         mov     r0, #0xB0000000      ; Load base of DMA  
         LDR     r9, [r8, #0x30]      ; Get the 2nd level source  
  
         ORR     r0, r0, r0           ; delay  
  
         STR     r9, [r8, #0x30]      ; Clear request bits  
  
         TST     r9, #0x20            ; Look for usb reset  
         BEQ     ep0  
  
         bl     udc_rst              ; Branch to UDC reset routine  
                                         ; (even though after host reset, UDC is on)  
  
usbrst1  ldr     r11, [r8, #0x30]      ; Get the 2nd level source  
         tst     r11, #0x20           ; see if Reset Int. Request is still active?  
         beq     done                 ; branch if no  
         orr     r11, r11, #0x20      ; Set the RSTIR bit for write-to-clear  
         str     r11, [r8, #0x30]    ;  
         ORR     r0, r0, r0           ; delay  
         ORR     r0, r0, r0  
         ORR     r0, r0, r0  
         ORR     r0, r0, r0  
         b       usbrst1             ; make sure RSTIR is clear
```

```

;-----
; Endpoint 0 routine
;-----

ep0      TST      r9, #0x01          ; Look for endpoint 0 interrupt
        BEQ      ep1

ep0_sr   LDR      r11, [r8, #0x10]   ; Read ep0 CSR

ep0_sr0  TST      r11, #0x20          ; Look for SE
        BEQ      ep0_sr1
        MOV      r12, #0x80
        STR      r12, [r8, #0x10]   ; Clear SE
        ORR      r12, r12, r12      ; Delay
        ORR      r12, r12, r12      ; Delay
        ORR      r12, r12, r12      ; Delay
        ORR      r12, r12, r12      ; Delay
        ORR      r12, r12, r12      ; Delay
        ORR      r12, r12, r12      ; Delay
        LDR      r11, [r8, #0x10]   ; Get ep0 CSR
        B        ep0_sr0           ; make sure SE cleared

ep0_sr1  LDR      r10, =EP0_STATE
        LDR      r12, [r10]         ; use state to decide what to do
        CMP      r12, #EP0_IDLE
        BEQ      do_idle
        CMP      r12, #EP0_IN_DATA_PHASE
        BEQ      do_idp
        CMP      r12, #EP0_OUT_DATA_PHASE
        BEQ      do_odp
        CMP      r12, #EP0_END_XFER
        BEQ      do_exfr
        B        done

do_idle  TST      r11, #0x01          ; Look for OPR
        BEQ      done
        LDR      r1, [r8, #0x20]     ; Get write count for ep0
        AND      r1, r1, #0xff       ; Filter out upper 1's
        MOV      r2, #0x0           ; Init counter
        LDR      r3, =DEVICE_REQ     ; Pointer to device request array

idle0    TEQ      r2, r1             ; Start loop
        LDRNE   r4, [r8, #0x1c]     ; load data from UDCD0 FIFO
        STRNEB  r4, [r3, r2]        ; Store byte, r3=base, r2=offset
        ADDNE   r2, r2, #0x01        ; Increment loop counter
        BNE     idle0              ; Branch until all bytes are read

        LDRB    r4, [r3, #0x06]     ; Get length of requested xfer
        LDR     r10, =SETUP_CNT
        STR     r4, [r10]           ; Save into SETUP_CNT variable

```



```
TEQ      r4, #0x0           ; Clear OPR only if SETUP_CNT != 0
MOVNE   r12, #0x40
STRNE   r12, [r8, #0x10]   ; Clear OPR

LDRB    r4, [r3, #0x0]     ; Get request type
AND     r4, r4, #0x60      ; Only care about bits 6:5
MOV     r4, r4, LSR #5
TEQ     r4, #0x00         ; Decide how to process request
BEQ     idle1             ; Only check for standard devices for now
;
;   CMP     r4, #0x??      ; Expand here . . .
;   BEQ     label
;   B      done

idle1    LDRB    r4, [r3, #0x01] ; Get brequest
TEQ     r4, #0x06         ; Check for GET DESCRIPTOR
BEQ     getdesc
TEQ     r4, #0x05         ; Check for SET ADDRESS
BEQ     setaddr
;
;   CMP     r4, #0x??      ; Expand here . . .
;   BEQ     label
;   B      done

getdesc  LDRB    r4, [r3, #0x03] ; Get wValue high
TEQ     r4, #0x02         ; Only check for Config Desc
BEQ     idle3
;
;   TEQ     r4, #0x??      ; Expand here . . .
;   BEQ     label
;   MOV     r12, #0x10     ; Set data end out by default
;   B      done

idle3    MOV     r12, #0x00   ; This might get overwritten below
LDR     r10, =MORE_SETUP_CNT
STR     r12, [r10]
LDR     r10, =SETUP_CNT
LDR     r5, [r10]           ; Get setup cnt & check it
LDR     r10, =CONFIG_DESC_SIZE
LDR     r6, [r10]
CMP     r5, r6
BLE     idle4
MOV     r12, #0x01         ; Note: too much data requested
LDR     r10, =MORE_SETUP_CNT
STR     r12, [r10]
LDR     r10, =SETUP_CNT
STR     r6, [r10]         ; Overwrite SETUP_CNT variable

idle4    MOV     r12, #EP0_IN_DATA_PHASE
LDR     r10, =EP0_STATE
STR     r12, [r10]         ; Change states
LDR     r12, =CONFIG_DESC
LDR     r10, =ROM_ADDR
```

```

        STR    r12, [r10]           ; Save start address of data to send
        B     do_idp               ; This is where the fifo gets loaded

setaddr  LDRB   r4, [r3, #0x02]     ; Get wValue low, which is 7 bit address
        AND   r4, r4, #0x7F        ; Filter out upper bits
        str   r4, [r8, #0x04]     ; Store 7 bit addr. in UDC address register
        LDR   r12, =UDC_ADDR      ; storage location for UDC address
        str   r4, [r12]           ; save UDC address
        MOV   r12, #0x50
        STR   r12, [r8, #0x10]    ; Clear OPR bit and Set DE bit
                                           ; (since no data phase)

chkaddr  orr    r12, r12, r12      ; delay
        orr   r12, r12, r12      ; delay
        orr   r12, r12, r12      ; delay
        ldr   r12, [r8, #0x04]    ; Get 7 bit address
        cmp   r4, r12            ; compare to saved address
        bne   chkaddr            ; make sure addr. is set in UDC core

        MOV   r12, #EP0_IDLE
        LDR   r10, =EP0_STATE
        STR   r12, [r10]         ; Change states
        B     done               ; No data phase, so done

do_idp   TST   r11, #0x04         ; Look to see if the EP0 is Stalled
        BEQ   idp0
        MOV   r12, #EP0_IDLE
        LDR   r10, =EP0_STATE
        STR   r12, [r10]         ; Return to idle if stalled
        MOV   r12, #0x04
        STR   r12, [r8, #0x10]    ; Clear stall bit

idp0     TST   r11, #0x20         ; Look for premature setup end
        BEQ   idp1
        MOV   r12, #EP0_IDLE
        LDR   r10, =EP0_STATE
        STR   r12, [r10]         ; Return to idle if SE set
        MOV   r12, #0x80
        STR   r12, [r8, #0x10]    ; Clear SE bit

idp1     TST   r11, #0x02         ; Make sure IPR is CLEAR!
        BNE   done               ; Do nothing if IPR is set
        MOV   r1, #0x08           ; This might get overwritten below
        LDR   r10, =SETUP_CNT
        LDR   r5, [r10]
        CMP   r5, r1              ; See if descriptor is bigger than maxp
        MOVLT r1, r5             ; adjust the loop variable
        MOV   r2, #0x0           ; Init index
        LDR   r10, =ROM_ADDR
        LDR   r3, [r10]          ; Get pointer to start of data

```



```
idp2    TEQ        r2, r1                ; See if we're done
        LDRNEB    r12, [r3, r2]        ; Get next byte
        STRNE    r12, [r8, #0x1c]     ; Place in FIFO
        ADDNE    r2, r2, #0x01        ; Increment loop
        BNE      idp2

        ADD      r3, r3, r1            ; Adjust ROM_ADDR for next time
        LDR      r10, =ROM_ADDR
        STR      r3, [r10]

        SUBS    r5, r5, r1            ; Adjust SETUP_CNT
        LDR      r10, =SETUP_CNT
        STR      r5, [r10]

        BNE     idp3                ; Skip if SETUP_CNT != 0
        LDR      r10, =MORE_SETUP_CNT
        LDR      r12, [r10]
        TEQ      r12, #0x0
        MOVNE    r12, #EP0_END_XFER   ; Change state to EP0_END_XFER
        LDRNE    r10, =EP0_STATE
        STRNE    r12, [r10]
        MOVEQ    r12, #0x10           ; Set Data End if MORE_SETUP_CNT==0
        STREQ    r12, [r8, #0x10]
        MOVEQ    r12, #EP0_IDLE       ; Change state to EP0_IDLE
        LDREQ    r10, =EP0_STATE
        STREQ    r12, [r10]

idp3    MOV      r12, #0x02            ; Set IPR
        ORR      r12, r12, r12        ; Delay
        ORR      r12, r12, r12        ; Delay
        ORR      r12, r12, r12        ; Delay
        ORR      r12, r12, r12        ; Delay
        STR      r12, [r8, #0x10]

        B       done

do_odp  ; Nothing for now
        B       done

do_exfr TST      r11, #0x04            ; Look to see if the EP0 is Stalled
        BEQ     exfr0
        MOV     r12, #EP0_IDLE
        LDR     r10, =EP0_STATE
        STR     r12, [r10]           ; Return to idle if stalled
        MOV     r12, #0x04
        STR     r12, [r8, #0x10]     ; Clear stall bit
```



```

exfr0  TST      r11, #0x20          ; Look for premature setup end
      BEQ      exfr1
      MOV      r12, #EP0_IDLE
      LDR      r10, =EP0_STATE
      STR      r12, [r10]          ; Return to idle if SE set
      MOV      r12, #0x80
      STR      r12, [r8, #0x10]    ; Clear SE bit

exfr1  TST      r11, #0x02          ; Make sure IPR is CLEAR!
      MOVEQ    r12, #0x10          ; Set Data end
      STREQ    r12, [r8, #0x10]
      B        done

;-----
; Endpoint 1 routine - OUT data xfer from host to UDC
;-----

ep1    TST      r9, #0x02          ; Look for endpoint 1 interrupt
      BEQ      ep2

      LDR      r11, [r8, #0x14]    ; Read ep1 CSR

      TST      r11, #0x02          ; Look for RPC
      BEQ      ep2

ep1_sr0  MOV      r0, #0xb0000000    ; DMA base
      MOV      r4, #0x7F           ; Disable DMA0
      STR      r4, [r0, #0x8]      ; 0x00=offset for DMA0 + 0x8=for clear
      LDR      r1, [r0, #0x10]     ; DBSA for DMA0-Points to next empty
mem loc  SUB      r1, r1, #0xc0000000 ; Convert real addr to virtual address

ep1_sr1  TST      r11, #0x04          ; Look for RPE
      BEQ      ep1_sr2
      ORR      r12, r12, r12        ; Got a valid RPE,
      ORR      r12, r12, r12        ; Must do some Packet Error Handling or
      ORR      r12, r12, r12        ; Receive FIFO Overrun Handling here..
      ORR      r12, r12, r12        ; RPE bit will be cleared when RPC cleared
      ORR      r12, r12, r12        ; assume DATA toggle and handshake error..
      ldr      r2, [r0, #0x14]      ; get DMA transfer count
      rsb     r2, r2, #DMA_OUT_COUNT_MAX ; MAX count - DMA count = # of bytes DMA'ed
      sub     r1, r1, r2            ; adjust DMA addr ptr back by the # of bytes
      b       ep1_sr3              ; skip emptying FIFO

ep1_sr2  TST      r11, #0x20          ; Look for RNE
      LDRNE    r12, [r8, #0x28]     ; Get byte from FIFO - empties residual data
      STRNEB   r12, [r1]           ; Make byte visible
      ADDNE    r1, r1, #1          ; Increment counter
      LDRNE    r11, [r8, #0x14]    ; Get ep1 CSR

```



```

        BNE      ep1_sr2          ; Loop if Receive FIFO is not empty

; got a good data packet, so count it
        ldr     r12, =OUTCOUNT   ; get address of OUT count variable
        ldr     r4, [r12]         ; get count
        add    r4, r4, #1         ; increment count
        str    r4, [r12]         ; save count

; Adjust DMA0 (OUT) pointer and xfer count
ep1_sr3  add    r1, r1, #0xc0000000 ; Convert virtual addr to real addr
        str    r1, [r0, #0x10]    ; set new adjusted DMA address
        MOV    r2, #DMA_OUT_COUNT_MAX
        str    r2, [r0, #0x14]    ; set new adjusted DMA transfer count

ep1_sr4  TST    r11, #0x08        ; Look for SST
        BEQ    ep1_sr5
        MOV    r12, #0x08
        STR    r12, [r8, #0x14]   ; Clear SST
        ORR    r12, r12, r12     ; SST is due to host sending more data
than
        ORR    r12, r12, r12     ; maximum packet size (UDCOMP)
        ORR    r12, r12, r12     ; Delay
        ORR    r12, r12, r12     ; Delay
        ORR    r12, r12, r12     ; Delay
        LDR    r11, [r8, #0x14]  ; Get ep1 CSR
        B     ep1_sr4           ; make sure SST cleared

ep1_sr5  MOV    r12, #0x02
        STR    r12, [r8, #0x14]   ; Clear RPC
        ORR    r12, r12, r12     ; Delay
        ORR    r12, r12, r12     ; Delay
        ORR    r12, r12, r12     ; Delay
        LDR    r11, [r8, #0x14]  ; Get ep1 CSR
        TST    r11, #0x02        ; Look for RPC
        BNE    ep1_sr5          ; make sure RPC cleared

        mov    r1, #0x11         ; Enable DMA0
        str    r1, [r0, #0x4]    ; 0x00=offset for DMA0 + 0x4=for set

;start DMA1 (IN) after we have received an OUT data packet
        mov    r1, #0x11         ; Enable DMA1
        str    r1, [r0, #0x24]   ; 0x20=offset for DMA1 + 0x4=for set

;-----
; Endpoint 2 routine - IN data xfer from UDC to host
;-----
```

```

ep2      TST      r9, #0x04          ; Look for endpoint 2 interrupt
        BEQ
        LDR      r11, [r8, #0x18]   ; Read ep2 CSR

ep2_sr0  TST      r11, #0x02        ; Look for TPC
        BEQ
        MOV      r3, #0x0          ; assume packet was sent successfully
        ; (i.e. don't need to resend)

ep2_sr2  TST      r11, #0x0C        ; Look for TUR or TPE
        BEQ      ep2_sr4
        ORR      r12, r12, r12     ; Got a valid TUR or TPE
        ORR      r12, r12, r12     ; Must set DMA1 ptr back to top of
        ; data to be resent
        mov      r3, #MAX_IN_PKT_MINUS1 ; get # of bytes that was transmitted minus 1
        add      r3, r3, #0x1      ; get number of bytes that was transmitted

; Adjust DMA1 (IN) pointer and xfer count
ep2_sr4  MOV      r0, #0xb0000000    ; DMA base address
        MOV      r1, #0x0000007F    ; Disable DMA1
        STR      r1, [r0, #0x28]    ; 0x20=offset for DMA1 + 0x8=for clear
        ORR      r12, r12, r12     ; Delay
        ORR      r12, r12, r12     ; Delay
        mov      r2, #DMA_IN_COUNT_MAX ; get max DMA1 xfer count
        ldr      r1, [r0, #0x34]    ; get current DMA1 xfer count
        sub      r1, r2, r1         ; calc no. of bytes moved from mem. to FIFO
        cmp      r1, #0x00         ; was there any bytes transferred?
        beq      ep2_sr5          ; skip if not
        str      r2, [r0, #0x34]    ; restore DMA1 xfer count = DMA_IN_COUNT_MAX
        mov      r2, #MAX_IN_PKT_MINUS1 ; get # of bytes that was transmitted minus 1
        add      r2, r2, #0x1      ; get number of bytes that was transmitted
        sub      r1, r1, r2         ; calc number of bytes the FIFO was primed
        ldr      r2, [r0, #0x30]    ; get current DMA1 address
        sub      r2, r2, r1         ; subtract # of bytes the FIFO was primed
        sub      r2, r2, r3         ; subtract whole packet (if an error)
        str      r2, [r0, #0x30]    ; restore DMA1 address

ep2_sr5  TST      r11, #0x10        ; Look for SST
        BEQ      ep1_sr6
        MOV      r12, #0x10
        STR      r12, [r8, #0x18]   ; Clear SST
        ORR      r12, r12, r12     ; SST is due to host sending more data than
        ORR      r12, r12, r12     ; maximum packet size (UDCOMP)
        ORR      r12, r12, r12     ; Delay
        ORR      r12, r12, r12     ; Delay
        ORR      r12, r12, r12     ; Delay
        LDR      r11, [r8, #0x18]   ; Get ep1 CSR

```



```

        B        ep1_sr5                ; make sure SST cleared

ep2_sr6  MOV     r12, #0x02
        STR     r12, [r8, #0x18]       ; Clear TPC
        ORR     r12, r12, r12         ; Delay
        ORR     r12, r12, r12         ; Delay
        ORR     r12, r12, r12         ; Delay
        LDR     r12, [r8, #0x18]       ; Get ep2 CSR
        TST     r12, #0x02            ; Look for TPC
        BNE     ep2_sr6                ; Branch if TPC not cleared

; Don't enable DMA1 (IN) until after you have received another OUT data packet
; Unless you had an error.

ep2_sr7  TST     r11, #0x1C            ; Look for SST or TUR or TPE
        BEQ     done
        mov     r1, #0x11             ; Enable DMA1
        str     r1, [r0, #0x24]

;-----
; End of service routine for UDC
;-----

done     STR     r9, [r8, #0x30]       ; Clear request bits
        LDMEA   r13!, {r0-r7}         ; get non banked registers from stack
        B       UDCL0

;-----
; Subroutines
;-----

;remember: this will invalidate any data in either XMIT/RCV FIFO's
udc_rst  MOV     r8, #0x80000000       ; UDC base address
        ldr     r12, [r8]             ; Read UDC Control Reg
        tst     r12, #0x02            ; Look for UDC Active
        bne     udc_rst               ; Loop until not active
udc_off  mov     r12, #0x01            ; Disable UDC
        STR     r12, [r8]             ; write to register
        ORR     r12, r12, r12         ; Delay
        ORR     r12, r12, r12         ; Delay
        ORR     r12, r12, r12         ; Delay
        ldr     r12, [r8]             ; look at the UDCCR
        cmp     r12, #0x41            ; disabled?
        BNE     udc_off               ; Loop if not disabled
udc_on   MOV     r12, #0x00            ; Enable UDC & all interrupts
        STR     r12, [r8]             ; write to register
        ORR     r12, r12, r12         ; Delay
        ORR     r12, r12, r12         ; Delay
        ORR     r12, r12, r12         ; Delay
```

```

        ldr    r12, [r8]                ; look at the UDCCR
        cmp    r12, #0x00              ; enabled?
        BNE   udc_on                   ; Loop if not enabled
maxpout MOV    r12, #MAX_OUT_PKT_MINUS1
        STR    r12, [r8, #0x08]       ; set OUT MaxP
        ORR    r12, r12, r12          ; Delay
        ORR    r12, r12, r12          ; Delay
        ORR    r12, r12, r12          ; Delay
        ldr    r12, [r8, #0x08]       ; look at OUT MaxP
        cmp    r12, #MAX_OUT_PKT_MINUS1; Correct?
        BNE   maxpout                 ; Loop if not correct
maxin  MOV    r12, #MAX_IN_PKT_MINUS1
        STR    r12, [r8, #0x0c]       ; set IN MaxP
        ORR    r12, r12, r12          ; Delay
        ORR    r12, r12, r12          ; Delay
        ORR    r12, r12, r12          ; Delay
        ldr    r12, [r8, #0x0c]       ; look at IN MaxP
        cmp    r12, #MAX_IN_PKT_MINUS1; Correct?
        BNE   maxin                   ; Loop if not correct

        mov    pc, lr                 ; Return from subroutine

;-----
pktcheck
        mov    r0, #0x1               ; assume good comparison
        ldr    r1, =MBASE              ; get address of MBASE
        ldr    r2, =OUTPKT1           ; get address of first packet
        mov    r5, #0x0               ; init index
        mov    r6, #508               ; check 127 long words (so, 508/4=127)
pktchkloop
        ldr    r3, [r1, r5]           ; get data sent via USB
        ldr    r4, [r2, r5]           ; get golden data
        cmp    r3, r4                 ; compare the two
        movne r0, #0x0               ; if not equal, show BADPKT sign
        bne   writeLED               ; go show it
        teq    r5, r6                 ; See if we're done
        addne r5, r5, #0x4            ; if no, increment index
        bne   pktchkloop             ; if no, loop

; r0 = data to display on the LED in register.
writeLED
        mov    r5, #0x80000000        ; base address of MCP
        add    r5, r5, #0x60000       ;
        mov    r6, #0x00010000       ; set the write bit
        orr    r6, r6, r0             ;
        str    r6, [r5, #0x10]       ; display to LED

        mov    pc, lr                 ; Return from subroutine

;-----

```



```
initLED
    mov    r3, #0x80000000    ; base address of MCP
    add    r3, r3, #0x60000    ;
    mov    r6, #0x00052000    ; turns on MCP
    add    r6, r6, #0x800    ;
    add    r6, r6, #0x00F    ;
    str    r6, [r3]    ; put 0x0005280F into reg 0x80060000

initLED2
    ldr    r6, [r3, #0x18]    ; get status
    tst    r6, #0x1000    ; test CWC
    beq    initLED2    ; wait for a one

    mov    r6, #0x38000    ; turn on two codec leds (red and
green)
    add    r6, r6, #0x7F    ;
    str    r6, [r3, #0x10]    ; put 0x0003807F into reg 0x80060010

    mov    pc, lr    ; Return from subroutine

;-----
; Data spaces
;-----

OUTPKT1
    DCB 0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08,0x09,0x0A,0x0B,0x0C,0x0D,0x0E,0x0F
    DCB 0x10,0x11,0x12,0x13,0x14,0x15,0x16,0x17,0x18,0x19,0x1A,0x1B,0x1C,0x1D,0x1E,0x1F
    DCB 0x20,0x21,0x22,0x23,0x24,0x25,0x26,0x27,0x28,0x29,0x2A,0x2B,0x2C,0x2D,0x2E,0x2F
    DCB 0x30,0x31,0x32,0x33,0x34,0x35,0x36,0x37,0x38,0x39,0x3A,0x3B,0x3C,0x3D,0x3E,0x3F

OUTPKT2
    DCB 0x40,0x41,0x42,0x43,0x44,0x45,0x46,0x47,0x48,0x49,0x4A,0x4B,0x4C,0x4D,0x4E,0x4F
    DCB 0x50,0x51,0x52,0x53,0x54,0x55,0x56,0x57,0x58,0x59,0x5A,0x5B,0x5C,0x5D,0x5E,0x5F
    DCB 0x60,0x61,0x62,0x63,0x64,0x65,0x66,0x67,0x68,0x69,0x6A,0x6B,0x6C,0x6D,0x6E,0x6F
    DCB 0x70,0x71,0x72,0x73,0x74,0x75,0x76,0x77,0x78,0x79,0x7A,0x7B,0x7C,0x7D,0x7E,0x7F

OUTPKT3
    DCB 0x80,0x81,0x82,0x83,0x84,0x85,0x86,0x87,0x88,0x89,0x8A,0x8B,0x8C,0x8D,0x8E,0x8F
    DCB 0x90,0x91,0x92,0x93,0x94,0x95,0x96,0x97,0x98,0x99,0x9A,0x9B,0x9C,0x9D,0x9E,0x9F
    DCB 0xA0,0xA1,0xA2,0xA3,0xA4,0xA5,0xA6,0xA7,0xA8,0xA9,0xAA,0xAB,0xAC,0xAD,0xAE,0xAF
    DCB 0xB0,0xB1,0xB2,0xB3,0xB4,0xB5,0xB6,0xB7,0xB8,0xB9,0xBA,0xBB,0xBC,0xBD,0xBE,0xBF

OUTPKT4
    DCB 0xC0,0xC1,0xC2,0xC3,0xC4,0xC5,0xC6,0xC7,0xC8,0xC9,0xCA,0xCB,0xCC,0xCD,0xCE,0xCF
    DCB 0xD0,0xD1,0xD2,0xD3,0xD4,0xD5,0xD6,0xD7,0xD8,0xD9,0xDA,0xDB,0xDC,0xDD,0xDE,0xDF
    DCB 0xE0,0xE1,0xE2,0xE3,0xE4,0xE5,0xE6,0xE7,0xE8,0xE9,0xEA,0xEB,0xEC,0xED,0xEE,0xEF
    DCB 0xF0,0xF1,0xF2,0xF3,0xF4,0xF5,0xF6,0xF7,0xF8,0xF9,0xFA,0xFB,0xFC,0xFD,0xFE,0xFF

OUTPKT5
    DCB 0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08,0x09,0x0A,0x0B,0x0C,0x0D,0x0E,0x0F
    DCB 0x10,0x11,0x12,0x13,0x14,0x15,0x16,0x17,0x18,0x19,0x1A,0x1B,0x1C,0x1D,0x1E,0x1F
    DCB 0x20,0x21,0x22,0x23,0x24,0x25,0x26,0x27,0x28,0x29,0x2A,0x2B,0x2C,0x2D,0x2E,0x2F
    DCB 0x30,0x31,0x32,0x33,0x34,0x35,0x36,0x37,0x38,0x39,0x3A,0x3B,0x3C,0x3D,0x3E,0x3F
```





```
OUTCOUNT DCD 0
UDC_ADDR DCD 0
EP0_STATE DCD EP0_IDLE
SETUP_CNT DCD SETUP_CNT
MORE_SETUP_CNTDCD 0
CONFIG_DESC_SIZEDCD9
DEVICE_REQDCB 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
CONFIG_DESCDCB 0x09, 0x02, 0x2e, 0x00, 0x01, 0x01, 0x00, 0x80, 0x32
DCB 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00
CONFIG_DESC_OLDDCB 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
ROM_ADDR DCD 0
BB DCD 0

END
```



# Support, Products, and Documentation

If you need technical support, a *Product Catalog*, or help deciding which documentation best meets your needs, visit the Intel World Wide Web Internet site:

<http://www.intel.com>

Copies of documents that have an ordering number and are referenced in this document, or other Intel literature may be obtained by calling **1-800-332-2717** or by visiting Intel's website for developers at:

<http://developer.intel.com>

You can also contact the Intel Massachusetts Information Line or the Intel Massachusetts Customer Technology Center. Please use the following information lines for support:

For documentation and general information:	
Intel Massachusetts Information Line	
United States:	1-800-332-2717
Outside United States:	1-303-675-2148
Electronic mail address:	techdoc@intel.com

For technical support:	
Intel Massachusetts Customer Technology Center	
Phone (U.S. and international):	1-978-568-7474
Fax:	1-978-568-6698
Electronic mail address:	techsup@intel.com

