TEXAS INSTRUMENTS

Application Report
SPRA574

A Practical Application of the TMS320C54x Host Port Interface (HPI)

Arun Chhabra and Ramesh A. Iyer *Digital Signal Processing Solutions*

Abstract

The host port interface (HPI) is an 8/16-bit parallel port used to interface a host processor or device to the Texas Instruments (TI™) TMS320C54x digital signal processor (DSP). The HPI enables a glueless interface with host processors containing single or dual data strobes in addition to separate or multiplexed address and data buses. This application report presents a hardware interface and the accompanying software protocol that are involved in communicating between the host and target units through the 8-bit HPI resident on the target processor.

Contents

Theory of Operation 2
HPI Functional Description 2
Setup Description 3
    Signal Connections 3
    Mode of Operation 5
    Sequence of Operation 5
Software Description 6
    Host Routine 6
    Kernel 6
    COFF-to-Hex Extraction Utility 7
Appendix A The Host Routine 8
Appendix B The Kernel 24

Theory of Operation

Information exchange between the host and target processors is carried out through the on-chip HPI memory, which resides on the target ’54x and can be accessed by both processors. The 8-bit HPI memory on the ‘542, ‘545, ‘548, and ‘549 devices is a 2K × 16-bit word block of dual-access on-chip RAM addressed at 1000h in data memory. The HPI interfaces to the host as a peripheral with the host device as master of the interface. The host communicates with the HPI memory by using the three HPI registers – the HPI address (HPIA), the HPI data (HPID) and the HPI control (HPIC). The ’54x has direct access to only the HPIC register.

The HPI has two modes of operation, shared-access mode (SAM) and host-only mode (HOM). In SAM, the normal mode of operation, both the ’54x and the host can access the HPI memory. In case of a conflict between host and ’54x cycles, the host has access priority while the ’54x waits one cycle. In HOM, the host has exclusive access to the HPI memory while the ’54x remains in reset or IDLE2 [1].

HPI Functional Description

An 8-bit data bus [HD7:0] exchanges information with the host. Because of the 16-bit word structure of the ’54x, all data transfers with a host must consist of two consecutive bytes. The dedicated HBIL pin indicates whether the first or second byte is being transferred.

The HCNTL0 and HCNTL1 control inputs indicate which one of the three internal HPI registers is being accessed, as well as the type of the access. The host address bus bits usually drive these two control inputs along with the HBIL input on the ’54x. The two control inputs specify access to the HPI registers in four different ways. The host can specify an access to the HPIC, the HPIA, and the HPID. The HPIA register serves as a pointer to HPI memory. The HPID is used to transfer data to the address pointed to by the HPIA. The HPID can also be referenced with the option of automatic address increment. In this mode, a data read causes a post-increment of the HPIA, and a data write causes a pre-increment of the HPIA.

The HDS1\ and HDS2\ strobes control the transfer of data during host-access cycles. Hosts with separate read and write strobes connect those to either HDS1\ or HDS2\. The HDS1\ and HDS2\ lines are internally exclusive-NORed. As a result, hosts with a single data strobe connect it to either HDS1\ or HDS2\, and care is taken to connect the unused strobe line high.

The HAS\ strobe is connected to a logic-1 level in hosts with separate address and data buses. Hosts with a multiplexed data and address bus are connected to the HAS\ pin through the address latch enable (ALE) pin or equivalent.

The HR/W strobe is driven high by a host to perform a read and low to perform a write.

The HCS\ line serves as the enabling input for the HPI and must be low during an access.

Host communication with the HPI is dependent on whether the HPI is ready to perform a transfer as indicated by the HRDY output line. When HRDY is high, the HPI is ready for a transfer to occur. When HRDY is low, it reflects that the HPI is busy completing the previous transaction. Since HCS\ enables the HPI and it is inactive (low) when an access occurs, it may be inferred that HRDY – which is always active high except for the duration of an access – is always active when HCS\ is active.

The HPIENA signal on the target is used as a select line that enables the HPI.

Host interrupts from the HPI are initiated by setting the HINT bit in the HPIC. The value in the HINT bit is reflected correspondingly on the HINT\ line that is output from the HPI port. At reset, the HINT bit is 0, causing the HINT\ line to be inactive (high). When the ‘C54x is brought out of reset, the on-chip boot-loader drives the HINT\ line low causing the HINT bit to have a value of 1.

Setup Description

The ’54x DSP starter kit (DSKplus) serves as the host unit and a DSP Research Tiger 542 board with a TMS320C542 serves as the target unit. The host processor, also a ’542, is rated at 40 MIPS and the target DSP is rated at 50 MIPS. Since this application report deals with the ’54x as both host and target, it is important to clarify here that the HPI in use resides on the target side of the interface. Although the host also has an independent HPI port, none of its pins are used to drive the target processor. This is important to remember, since the interface can be between a target ’54x and any generic host processor, in which case the host may not have an HPI port.

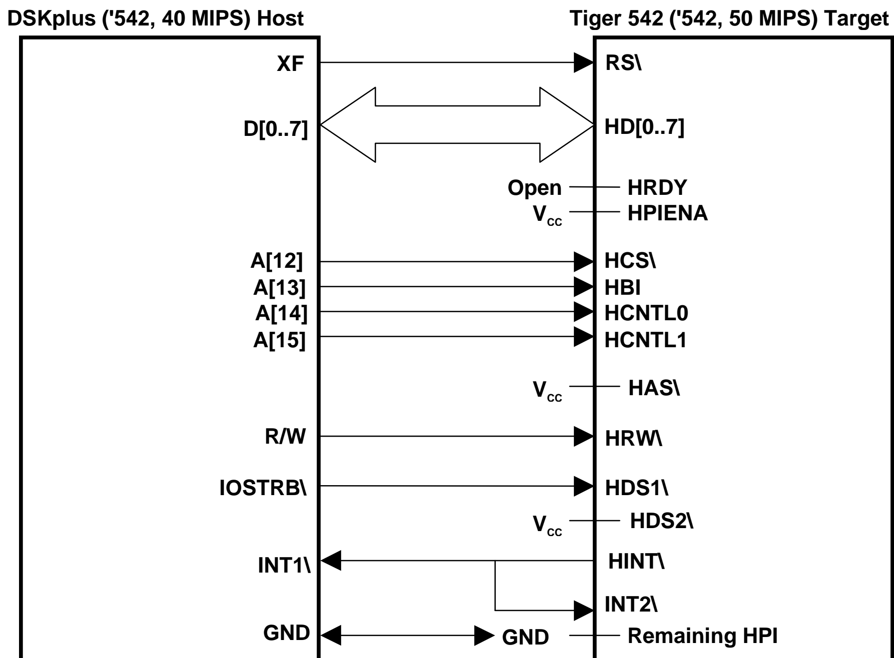
The host processor board is driven by a 5-V power supply and is connected to the PC through a JTAG cable and an XDS510 emulator card. The target DSP board is a plug-in ISA card.

Signal Connections

Figure 1 shows the relevant connections between the two boards.

In this setup, since the target is faster than the host processor, the target ’54x will likely be through processing the previous transfer before receiving any new request from the host. In such a situation, the HRDY signal (pin 19) on the Tiger 542 HPI port, can be left open (unconnected).

Figure 1. Hardware Schematic



Mode of Operation

The target board is used in the microcontroller (MC) mode; this enables the on-chip ROM and bootloader.

The jumper configurations on the target board have been set to the values indicated in Table 1. A jumper satisfying the In configuration indicates that the jumper is enabled; similarly, a jumper satisfying the Out configuration indicates that the jumper is disabled.

Table 1. Jumper Configurations

Jumper	In	Out
JP 12		
ALN	√	
ROM		√
TBC		√
JP 14		
CPURST	√	

The bootloader program on the target DSP enables the HINT\ line by driving it low. An HPI boot from HPI memory only occurs if the HINT\ line is connected externally to the INT2\ line. The state of the INT2\ line is checked at least 30 clock cycles after the boot-loader program has been initiated. On the target board, the INT2\ signal is the output at pin 40 of a PAL device, labeled U30. Adhering to the jumper configurations of Table 1, the target board drives the INT2\ line at pin 40 of U30 to a midlevel logic swing. Connecting HINT\ to INT2\ causes HINT\ to reflect the midlevel logic swing, instead of a full TTL swing that is expected once the target DSP is released from reset. This midlevel logic swing prevents an HPI boot from occurring. Although the board manufacturer has been contacted on this issue, no definite explanation has been obtained. As a result, to circumvent the problem, pin 40 (INT2\) of U30 is raised and connected directly to the HINT signal.

Sequence of Operation

After loading the program on the host processor, single step or run through the code until the software reset command is encountered. At this point, reset the target board using the reset button that is provided on the board. While continuing to run through the code in this state, the kernel code is transferred from the host to the target. The target's reset button is released after witnessing a change in the ST0 register (indicating a change in the XF bit), soon after single stepping past the software reset-release command.

When the target DSP is released from reset, the bootloader program on it begins executing. Upon recognizing an HPI boot, the bootloader transfers program execution to resume from DARAM location 1000h.

Software Description

The software component of the interface consists of three primary parts – host routine, kernel, and COFF extraction routine, each is described below.

Host Routine

The main program that initiates communication between the two processors is referred to as the host routine (see *Appendix A*). The host routine begins by asserting the software component of the target reset. Subsequently, it sets the BOB bit values in the HPIC register on the target HPI. Following this, a call to the host2dsp_dnld_kernel routine occurs. This routine contains the kernel code that executes on the target processor and consists of a series of calls to the send_word function. The send_word function performs the actual transfer of the kernel chunk and is responsible for decomposing the transfer process into a series of I/O writes.

Host_hpiram_xfer is another function that resides in the host program; it is responsible for transferring the application code chunks to the target ’54x. This routine is also composed of a series of calls to the send_word function, which conducts the actual transfer of application code chunks from the host to the target DSP. When the kernel routine executing on the target ‘54x indicates to the host that it is ready to receive application chunks, then the host program invokes the host_hpiram_xfer routine.

Kernel

The host2dsp_dnld_kernel routine (see *Appendix B*) transfers kernel code to the target DSP starting at location 1010h. Locations 1000h to 100Fh contain a relocation code that is transferred along with the kernel. The bootloader begins by executing the relocation code, which relocates raw data from locations 1010h onward in HPI memory to location 0080h and beyond in internal program memory. Once this has been done, the last line of relocation code branches control to resume execution at location 0080h of internal program memory.

The kernel code is an independent program that has been designed to run on the target ’54x. As a result of the HINT\ line being enabled by the bootloading operation, the kernel program begins by monitoring the HPIC, until the host processor clears the HINT. Then, the HPI memory is cleared and the kernel asserts another HINT to the host. Subsequently, the application code is downloaded from the host processor. For the duration of the download, the target DSP monitors the status of the HINT bit, until it is cleared by the host. Once the HINT bit is cleared by the host, the target ’54x will recognize that one complete chunk has been downloaded. The HPI memory is cleared with 0s to prepare for the arrival of the next chunk. At this stage, the target ’54x is ready to perform the transfer of the application code chunk from HPI memory to the target ’54x internal memory. The target ’54x proceeds to decipher parameters for the transfer of the application code chunks to internal memory. These parameters are specified in the header information of the application code chunk and contain the number of chunks to be transferred, the destination address, and the length of each chunk. Each chunk is then transferred to some internal memory location. If the last chunk has been transferred, the kernel passes control to the starting address of the first chunk. However, if additional chunks remain to be transferred, then the target asserts another HINT and the process continues. Each application chunk transfer is monitored by a modulo-2 Cyclic

Redundancy Check (CRC). If the CRC reflects an error in a particular chunk transfer, then that chunk is retransmitted to the target HPI memory. This process continues until the CRC is passed.

COFF-to-Hex Extraction Utility

The original kernel and application code are written in assembly. However, to be able to transfer them from the host to the ’54x target, it is necessary to know the hex code corresponding to each line of assembly code. With these hex values known, it becomes a simple issue of performing a series of I/O writes to transfer the code from the host processor to the ’54x target. The conversion of the assembly code to hex values can be easily done by manually translating each command into it’s corresponding hex data; but, this can prove to be a very laborious procedure when the size of a typical application code is considered.

With this in mind, a COFF-to-hex extraction utility is used which performs the intended operation through a single executable program. This utility extracts hex code equivalents of a 16-bit COFF file.

The COFF-to-hex extraction utility used in this application is meant to extract hex value equivalents of a COFF2 type executable file. During the extraction process, the COFF2 utility also resolves all address references based on the memory map specified in the linker command file. The COFF2 utility is based on an original COFF extraction utility that extracts hex code equivalents of a COFF1 type executable file.

COFF1 and COFF2 file types have different header formats, while the sections themselves – which contain raw data – are identical [2]. The TI ’54x version 1.16 tools generate a COFF1 type executable file. All TI ’54x code generation tools, version 1.2 tools and higher generate a COFF2 type executable file. The only distinguishing factor between these two COFF file formats is the section header length; COFF1 file types have a 39-byte long section header, whereas COFF2 file types have a 47-byte long section header [2].

References

1. *TMS320C54x DSP Reference Set, Volume 1: CPU and Peripherals*, (SPRU131), 1997
2. *TMS320C54x Assembly Language Tools User’s Guide*, (SPRU102), 1997

Appendix A The Host Routine

```
/*****************************************************************
*   FILENAME: HOST2.C
*
*   AUTHOR: ARUN CHHABRA
*
*   DATE: 1/30/98  3:20pm
*
*            The DSP Research Board will be used as the TARGET
*
*            The DSKPlus board will be used as the HOST
*
*
******************************************************************/


#define A_SEND     0x8000
#define C_SEND     0x0000
#define D_SEND     0x4000
#define N_SEND     0xC000
#define HBIL_HI       0x2000
#define BYTE_HI(w) ((w>>8) & 0x00FF)
#define BYTE_LO(w) (w & 0x00FF)

unsigned int hpimode;
unsigned char val1, val2;
unsigned int word;
unsigned const count = 4;  /* should be done globally */
unsigned int index = 1;   /* done globally */
              /* index will always start out as 1 */
```

```
/*******************************************************************
*             send_word()
*
* Send a word (16 bits) to the DSP. A pre-increment occurs for each
* write operation.
* Arguments: i) word
*             ii) hpimode:   A_SEND: write to HPIA
*                            C_SEND: write to HPIC
*                            D_SEND: write to HPID with auto increment
*                            N_SEND: write to HPID (NO auto increment)
*
*******************************************************************/

void send_word(unsigned int word, unsigned int hpimode)
{
  asm("OUT  .set 07h");
  val1 = BYTE_LO(word);
  asm(" STL A, OUT ");

  asm(" LD #0h, B ");
                   /* Clear the B acc at the beginning of each
                      incoming "word"...used later to total two bytes
                      to form the "word" */

  if (hpimode==0x8000) {

   asm(" PORTW OUT, 0x8000 ");



   hpimode = (hpimode | HBIL_HI);
                   /* This line doesn't actually need to be here, since */
                   /* the Port address is hard-coded in the next */
                   /* statement...similarly done in the elseif below */


   val2 = BYTE_HI(word);
   asm(" STL A, OUT ");

   asm(" PORTW OUT, 0xa000 ");
  }

  else if (hpimode==0x4000) {

    asm(" PORTW OUT, 0x4000 ");

    asm(" ADD OUT, B ");

    hpimode = (hpimode | HBIL_HI);

    val2 = BYTE_HI(word);
    asm(" STL A, OUT ");
    asm(" PORTW OUT, 0x6000 ");

    asm(" ADD OUT, 8, B ");
                   /* Adds the upper and lower bytes to give the "word"
                      value */
    asm(" XOR *AR2, B ");
```

```
                    /* XOR the result of the last XOR operation
                       stored in AR2 with the current word ;
                       if dealing with the first incoming word of
                       a group of words, then AR2 will equal zero */
    asm(" NOP ");
    asm(" STL B, *AR2 ");
                    /* Save the result of the latest operation into AR2. */

  }

  else if (hpimode==0x0000) {

    asm(" PORTW OUT, 0x0000 ");

    hpimode = (hpimode | HBIL_HI);

    val2 = BYTE_HI(word);
    asm(" STL A, OUT ");
    asm(" PORTW OUT, 0x2000 ");
  }

  return;
}
```

```
/*******************************************************************
*                  relocate()
*
*  Download the relocation code along with the embedded kernel of the
*  first 2K set of code. The relocation code will be responsible for
*  transferring the embedded kernel from HPI RAM to internal DSP
*  memory.
*  The embedded kernel will be the first 2K (or less) chunk of code
*  that is sent from the HOST to the DSP. It will contain calls to
*  the HOST to transfer subsequent 2K chunks of data to some
*  specified space in external SRAM. The embedded kernel may also
*  contain the total number of such 2K blocks to be transferred.
*
*******************************************************************/

void relocate (int addr, int length)
{


  send_word(0x0fff, A_SEND);
      /* Set the HPIA to begin location 1000h in the HPI buffer */



  send_word(0x7710,D_SEND);     /*1000: STM #1010h, AR0) */
  send_word(0x1010,D_SEND);
  send_word(0xf070,D_SEND);     /*1002: RPT        */
  send_word(length-1,D_SEND);   /*1003: #length  */
  send_word(0x7190,D_SEND);     /*1004: MVDK *AR0+, addr */
  send_word(addr,D_SEND);
  send_word(0xf073,D_SEND);     /*1006: B #addr (entrypoint) */
  send_word(addr, D_SEND);

  return;
                   /* The MVDK instruction will take care of
                      incrementing addr, in each iteration */
}
```

```
/*******************************************************************
*                  host2dsp_dnld_kernel()
*
* Download the kernel relocation code. The embedded kernel will
* begin at location 1010h. The kernel relocation code starts at
* location 1000h.
* NOTE: Kernel execution does not occur in this module.
*                   Only kernel DOWNLOAD occurs here.
*
*       The hex values in the embedded kernel below are derived by
*       applying the COFF-to-hex conversion utility [SPRA573] on the
*       kernel code shown in Appendix B.
********************************************************************/
void host2dsp_dnld_kernel()
{

  /*****************************************************
  * EMBEDDED KERNEL BEGINS HERE           *
  *                                          *
  *****************************************************/

relocate(0x0080, 0x0092);
send_word(0x100F, A_SEND);

/* section length = 0x0092 */


/* Writing a 0xa into HPIC i.e. asserting HINT in SAM mode */

send_word(0x6807, D_SEND);
send_word(0x2300, D_SEND);
send_word(0xF495, D_SEND);
send_word(0xF495, D_SEND);
send_word(0x7700, D_SEND);
send_word(0x0000, D_SEND);
send_word(0xF495, D_SEND);
send_word(0xF495, D_SEND);
send_word(0xF495, D_SEND);
send_word(0x7728, D_SEND);
send_word(0x7E7F, D_SEND);
send_word(0x7718, D_SEND);
send_word(0x0112, D_SEND);
send_word(0x482C, D_SEND);
send_word(0xF495, D_SEND);
send_word(0xF495, D_SEND);
send_word(0xF495, D_SEND);
send_word(0xF030, D_SEND);
send_word(0x0008, D_SEND);
send_word(0xF495, D_SEND);
send_word(0xF495, D_SEND);
send_word(0xF844, D_SEND);
send_word(0x008D, D_SEND);
send_word(0xF495, D_SEND);
send_word(0xF495, D_SEND);
send_word(0xF495, D_SEND);
send_word(0x7714, D_SEND);
send_word(0x1000, D_SEND);
send_word(0xF070, D_SEND);
```

```
send_word(0x0800, D_SEND);
send_word(0x7694, D_SEND);
send_word(0x0000, D_SEND);
send_word(0x7310, D_SEND);
send_word(0x1010, D_SEND);
send_word(0xF495, D_SEND);
send_word(0xF495, D_SEND);
send_word(0xF495, D_SEND);
send_word(0x772C, D_SEND);
send_word(0x000A, D_SEND);
send_word(0xF495, D_SEND);
send_word(0xF495, D_SEND);
send_word(0x482C, D_SEND);
send_word(0xF495, D_SEND);
send_word(0xF495, D_SEND);
send_word(0xF495, D_SEND);
send_word(0xF030, D_SEND);
send_word(0x0008, D_SEND);
send_word(0xF495, D_SEND);
send_word(0xF495, D_SEND);
send_word(0xF844, D_SEND);
send_word(0x00A9, D_SEND);
send_word(0xF495, D_SEND);
send_word(0xF495, D_SEND);
send_word(0xF495, D_SEND);
send_word(0xF495, D_SEND);
send_word(0xE800, D_SEND);
send_word(0x7714, D_SEND);
send_word(0x1010, D_SEND);
send_word(0x771A, D_SEND);
send_word(0x07EF, D_SEND);
send_word(0xF072, D_SEND);
send_word(0x00BF, D_SEND);
send_word(0x1C94, D_SEND);
send_word(0xF495, D_SEND);
send_word(0xF495, D_SEND);
send_word(0x8013, D_SEND);
send_word(0x7210, D_SEND);
send_word(0x1005, D_SEND);
send_word(0xF495, D_SEND);
send_word(0xF495, D_SEND);
send_word(0xF495, D_SEND);
send_word(0xF4AB, D_SEND);
send_word(0xF495, D_SEND);
send_word(0xF495, D_SEND);
send_word(0xF495, D_SEND);
send_word(0xF830, D_SEND);
send_word(0x00DC, D_SEND);
send_word(0xF495, D_SEND);
send_word(0xF495, D_SEND);
send_word(0xF495, D_SEND);
send_word(0x7710, D_SEND);
send_word(0x100A, D_SEND);
send_word(0xF495, D_SEND);
send_word(0xF495, D_SEND);
send_word(0x7680, D_SEND);
send_word(0x0001, D_SEND);
send_word(0xF495, D_SEND);
```

```
send_word(0xF495, D_SEND);
send_word(0x7710, D_SEND);
send_word(0x0001, D_SEND);
send_word(0xF073, D_SEND);
send_word(0x00E2, D_SEND);
send_word(0x7710, D_SEND);
send_word(0x100A, D_SEND);
send_word(0xF495, D_SEND);
send_word(0xF495, D_SEND);
send_word(0x7680, D_SEND);
send_word(0x0000, D_SEND);
send_word(0xF495, D_SEND);
send_word(0xF495, D_SEND);
send_word(0x7710, D_SEND);
send_word(0x0000, D_SEND);
send_word(0xF495, D_SEND);
send_word(0xF495, D_SEND);
send_word(0xF495, D_SEND);
send_word(0xF495, D_SEND);
send_word(0xF495, D_SEND);
send_word(0xF495, D_SEND);
send_word(0xF495, D_SEND);
send_word(0xF495, D_SEND);
send_word(0xF495, D_SEND);
send_word(0x7715, D_SEND);
send_word(0x1004, D_SEND);
send_word(0xF495, D_SEND);
send_word(0xF495, D_SEND);
send_word(0x7711, D_SEND);
send_word(0x1002, D_SEND);
send_word(0x7713, D_SEND);
send_word(0x1000, D_SEND);
send_word(0x7212, D_SEND);
send_word(0x1001, D_SEND);
send_word(0x7208, D_SEND);
send_word(0x1003, D_SEND);
send_word(0x1183, D_SEND);
send_word(0x6D89, D_SEND);
send_word(0x4781, D_SEND);
send_word(0x7F92, D_SEND);
send_word(0xF495, D_SEND);
send_word(0x6085, D_SEND);
send_word(0x0000, D_SEND);
send_word(0xF495, D_SEND);
send_word(0xF820, D_SEND);
send_word(0x0107, D_SEND);
send_word(0xF073, D_SEND);
send_word(0x0110, D_SEND);
send_word(0x7714, D_SEND);
send_word(0x1000, D_SEND);
send_word(0xF070, D_SEND);
send_word(0x0800, D_SEND);
send_word(0x7694, D_SEND);
send_word(0x0000, D_SEND);
send_word(0xF495, D_SEND);
send_word(0xF073, D_SEND);
send_word(0x00A0, D_SEND);
send_word(0xF073, D_SEND);
```

```
send_word(0x2000, D_SEND);
return;
}
/**********************************************************************
*      host_hpiram_xfer(ushort dst, uchar *src, int count)
*
*      This function moves a blocks of application code from
*      host to C54x HPIram.
*
*      See  document SPRA573 for detailed information on preparing
*      application code chunks for transfer. i.e. the steps
*      taken to convert the assembly code into the
*      corresponding hex values, as seen throughout the four
*      chunk transfers in this routine.
*
*********************************************************************/

void host_hpiram_xfer()
{
unsigned int new_count;

while (index<=count){

new_count = count - index;
            /* Earlier, this statement was declared outside
              the while loop and thus the contents of
              location 1004h never reflected the correct
              value of new_count after the first chunk
              transfer...now it is OK */

if (index==1){


send_word(0x0fff,A_SEND);

send_word(0x0000,D_SEND);
       /* COMMAND value = 0; provides for data download option */
       /* 1000h: #0h; ST #1000, 0 */

send_word(0x1010,D_SEND);
       /* destination address to where the application code is
        placed for interim storage in HPIram...always 0x1010 */
       /* 1001h: #1010h (DEST) ;LD DUPADDR, #1010h*/

send_word(0x000a, D_SEND);

send_word(0x2000,D_SEND);
       /* destination address where application code chunks will
         be placed in target program memory...for sample code,
         this value is 2000h...this is fed through the lower
         two bytes of the accumulator */
       /* 1003h: #2000h (SRC) ;LD PUDDADDR, #2000h*/

       /* 2000h is the assigned target program memory address */
       /* 001dh is equivalent to      */
       /* 1010h is the src address in HPIRAM */
```

```
send_word(new_count,D_SEND);
        /* This sets the number of chunks */

send_word(0x100f,A_SEND);
                /* This sets up the address to where *
                * data chunks from the host     *
                * (say, 1010h) are transferred to  *
                * in HPIRAM.            */

             /* section length = 0x000d  */


/* TO IMPLEMENT THE CHECKSUM...TOTAL THE VALUES AND PLACE
THE ACCUMULATING VALUE IN SOME AR on the host side. */

/* BEFORE BEGINNING THE ADDITION OF A NEW SET OF DATA VALUES, REMEMBER
TO CLEAR THE AR IN USE FOR TOTALLING!!!  below:  */


asm(" STM #27f0h, AR2 ");
asm(" ST #0h, *AR2 ");

/************ FIRST CHUNK of  application code*************/

send_word(0xF495, D_SEND);
send_word(0xF495, D_SEND);
send_word(0xF495, D_SEND);
send_word(0x7712, D_SEND);
send_word(0x1400, D_SEND);
send_word(0x7711, D_SEND);
send_word(0x0060, D_SEND);
send_word(0xF495, D_SEND);
send_word(0xF495, D_SEND);
send_word(0x7791, D_SEND);


send_word(0x1004, A_SEND);
/* The checksum value will be placed in location
           0x1005 in HPIram of the target */
asm(" LD *AR2, B ");
asm(" NOP ");
asm(" AND #00ffh, B, A ");
asm(" STL A, OUT ");
asm(" PORTW OUT, 0x4000 ");
/* sends over the lower byte of checksum */

asm(" AND #0ff00h, B, A ");
asm(" SFTA A, -8 ");
asm(" NOP ");
asm(" STL A, OUT ");
asm(" PORTW OUT, 0x6000 "); /* sends over upper byte of checksum */

asm(" LD #0909h, A ");
asm(" LD #255, B ");
asm(" NOP ");
asm(" AND B, A ");
asm(" NOP ");
asm(" STL A, OUT ");
```

```
asm(" PORTW OUT, 0x0000 ");
asm(" PORTW OUT, 0x2000 ");


} /* end of index==1 case */


else if (index==2){

asm(" NOP ");

send_word(0x0fff,A_SEND);

send_word(0x0000,D_SEND);
       /* COMMAND value = 0; provides for data download option */
       /* 1000h: #0h; ST #1000, 0 */

send_word(0x1010,D_SEND);
       /* destination address to where the application code is
        placed for interim storage in HPIram...always 0x1010 */
       /* 1001h: #1010h (DEST) ;LD DUPADDR, #1010h*/

send_word(0x000a, D_SEND);  /* for the segmented simple.out.c case */

send_word(0x200a, D_SEND);

       /* 1003h: #2000h (SRC) ;LD PUDDADDR, #2000h*/

       /* 2000h is the assigned target program memory address */
       /* 001dh is equivalent to      */
       /* 1010h is the src address in HPIRAM */


send_word(new_count,D_SEND);
        /* This sets the number of chunks */

send_word(0x100f,A_SEND);
                /* This sets up the address to where *
                * data chunks from the host     *
                * (say, 1010h) are transferred to  *
                * in HPIRAM.            */

             /* section length = 0x0029  */

/* BEFORE BEGINNING THE ADDITION OF A NEW SET OF DATA VALUES, REMEMBER
TO CLEAR THE AR IN USE FOR TOTALLING!!!  below:  */

asm(" STM #27f0h, AR2 ");
asm(" ST #0h, *AR2 ");

/********* SECOND CHUNK of  application code *********/

send_word(0xCCCC, D_SEND);
send_word(0x7791, D_SEND);
send_word(0xAAAA, D_SEND);
send_word(0x7791, D_SEND);
send_word(0x5555, D_SEND);
send_word(0x7781, D_SEND);
```

```
send_word(0x0000, D_SEND);
send_word(0xF495, D_SEND);
send_word(0xF495, D_SEND);
send_word(0xF495, D_SEND);


send_word(0x1004, A_SEND);
/* The checksum value will be placed in location
           0x1005 in HPIram of the target */
asm(" LD *AR2, B ");
asm(" NOP ");
asm(" AND #00ffh, B, A ");
asm(" STL A, OUT ");
asm(" PORTW OUT, 0x4000 ");
 /* sends over the lower byte of checksum */

asm(" AND #0ff00h, B, A ");
asm(" SFTA A, -8 ");

asm(" STL A, OUT ");
asm(" PORTW OUT, 0x6000 "); /* sends over upper byte of checksum */


asm(" LD #0909h, A ");
asm(" LD #255, B ");

asm(" AND B, A ");

asm(" STL A, OUT ");

asm(" PORTW OUT, 0x0000 ");
asm(" PORTW OUT, 0x2000 ");
} /* end of index==2 case */

else if (index==3){



send_word(0x0fff,A_SEND);

send_word(0x0000,D_SEND);
      /* COMMAND value = 0; provides for data download option */
        /* 1000h: #0h; ST #1000, 0 */

send_word(0x1010,D_SEND);
       /* destination address to where the application code is
        placed for interim storage in HPIram...always 0x1010 */
       /* 1001h: #1010h (DEST) ;LD DUPADDR, #1010h*/

send_word(0x000a, D_SEND);  /* for the segmented simple.out.c case */

send_word(0x2014, D_SEND);


       /* 2000h is the assigned target program memory address */
       /* 001dh is equivalent to      */
       /* 1010h is the src address in HPIRAM */
```

```
send_word(new_count,D_SEND);
        /* This sets the number of chunks */

send_word(0x100f,A_SEND);

/* BEFORE BEGINNING THE ADDITION OF A NEW SET OF DATA VALUES, REMEMBER
TO CLEAR THE AR IN USE FOR TOTALLING!!!  below:  */

asm(" STM #27f0h, AR2 ");
asm(" ST #0h, *AR2 ");

/******** THIRD CHUNK of  application code ********/
send_word(0x771A, D_SEND);
send_word(0x0003, D_SEND);
send_word(0x7710, D_SEND);
send_word(0xE000, D_SEND);
send_word(0xF072, D_SEND);
send_word(0x2020, D_SEND);
send_word(0x3089, D_SEND);
send_word(0x8c82, D_SEND);
send_word(0x1082, D_SEND);
send_word(0x0882, D_SEND);


send_word(0x1004, A_SEND);
   /* The checksum value will be placed in location
             0x1005 in HPIram of the target */
asm(" LD *AR2, B ");

asm(" AND #00ffh, B, A ");
asm(" STL A, OUT ");

asm(" PORTW OUT, 0x4000 ");
/* sends over the lower byte of checksum */
asm(" AND #0ff00h, B, A ");
asm(" SFTA A, -8 ");
;
asm(" STL A, OUT ");
asm(" PORTW OUT, 0x6000 "); /* sends over upper byte of checksum */

asm(" LD #0909h, A ");
asm(" LD #255, B ");

asm(" AND B, A ");

asm(" STL A, OUT ");
asm(" PORTW OUT, 0x0000 ");


asm(" PORTW OUT, 0x2000 ");

} /* end of the index==3 case */

else if (index==4){



send_word(0x0fff,A_SEND);
```

```
send_word(0x0000,D_SEND);
       /* COMMAND value = 0; provides for data download option */
       /* 1000h: #0h; ST #1000, 0 */

send_word(0x1010,D_SEND);
        /* destination address to where the application code is
        placed for interim storage in HPIram...always 0x1010 */
        /* 1001h: #1010h (DEST) ;LD DUPADDR, #1010h*/

send_word(0x000b, D_SEND);  /* for the segmented simple.out.c case */

send_word(0x201e, D_SEND);


       /* 2000h is the assigned target program memory address */
       /* 001dh is equivalent to      */
       /* 1010h is the src address in HPIRAM */

send_word(new_count,D_SEND);
        /* This sets the number of chunks */

send_word(0x100f,A_SEND);


/* BEFORE BEGINNING THE ADDITION OF A NEW SET OF DATA VALUES, REMEMBER
TO CLEAR THE AR IN USE FOR TOTALLING!!!  below:  */

asm(" STM #27f0h, AR2 ");
asm(" ST #0h, *AR2 ");

/********* FOURTH CHUNK of  application code *********/
send_word(0xF495, D_SEND);
send_word(0xF495, D_SEND);
send_word(0xF495, D_SEND);
send_word(0x7711, D_SEND);
send_word(0x0063, D_SEND);
send_word(0x6D92, D_SEND);
send_word(0xF4AA, D_SEND);
send_word(0xF820, D_SEND);
send_word(0x2014, D_SEND);
send_word(0xF495, D_SEND);
send_word(0xF495, D_SEND);


send_word(0x1004, A_SEND);
/* The checksum value will be placed in location
          0x1005 in HPIram of the target */
asm(" LD *AR2, B ");
asm(" NOP ");
asm(" AND #00ffh, B, A ");
asm(" STL A, OUT ");
asm(" PORTW OUT, 0x4000 ");
/* sends over the lower byte of checksum */
asm(" AND #0ff00h, B, A ");
asm(" SFTA A, -8 ");
asm(" STL A, OUT ");
asm(" PORTW OUT, 0x6000 "); /* sends over upper byte of checksum */
```

```
asm(" LD #0909h, A ");
asm(" LD #255, B ");
asm(" AND B, A ");
asm(" STL A, OUT ");
asm(" PORTW OUT, 0x0000 ");
asm(" PORTW OUT, 0x2000 ");

} /* end of the index==4 case */


/**** The lines below cause the host to loop until it senses another
    HINT assertion by the target, only then will it re-enter the
    while loop and send the next chunk...
******/

asm("circle: PORTR 0x0000, IN ");
asm(" LD IN, A ");
asm(" PORTR 0x2000, IN ");
asm(" ADD IN, 8, A ");
asm(" AND #0808h, A ");
asm(" BC circle, AEQ ");

/* Specify the address (it will be post-incremented) where you want to
start reading from */

send_word(0x100a, A_SEND);

asm(" PORTR 0x4000, IN "); /* read the lower byte of what is in the
                specified address location */
asm(" LD IN, A ");
asm(" PORTR 0x6000, IN "); /* read upper byte of what is in specified
                address location */
asm(" ADD IN, 8, A ");   /* at this point the whole word is read and
                is in the accumulator A */
asm(" STM #27f1h, AR1 ");  /* initialize AR1 */
asm(" STL A, *AR1 ");  /* store value of flag into address pointed to
              by AR1 */
asm(" CMPM *AR1, #0h ");
asm(" BC over, NTC ");
          /* if flag value indicates that the checksum has not
           matched, then re-send most recently sent block */

index++;        /* if flag value indicates that the checksum has
             matched, then increment counter and prepare to
             send new block */

if (index > count){
 ;
    }
  else {
    asm(" B over ");
    }

asm("over: NOP ");

} /* end of the while loop */

} /* end of host_hpiram_xfer */
```

```
main()
{
    asm("OUT  .set 07h");
    asm("IN   .set 09h");
    asm(" LD #1, DP ");
/* This data page pointer setting doesn't
                seem to have any affect...since the CPL
                bit is configured to show SP mode, then
                values of OUT and IN are in SP mode as
                well */

    asm(" STM #1020h, PMST ");
           /* This is in keeping with the new value of the
           vector table location at 0x100a...i.e. IPTR=32 */

          /* Set the IPVT to indicate the vector table
           location for the host device (for the DSK, I am
           using location 0x0100a = IPVT set to 32 */


/*************************************/
/************************************
* The BOB bit is set here, by PORTW commands to the target.
* We may recall that when the host is writing any value to the HPIC
* it is necessary for it to write identical byte values in both the
* upper and lower bytes.
* For that reason, we may be sure that the particular value being fed
* in will not be affected by the fact that there is no BOB bit setting
* prior to itself, i.e. since the two byte entries are identical, then
* inevitably the BOB bit will be set.
*************************************/
/*************************************/


    asm(" RSBX XF ");
            /* Software reset asserted for target board */

    asm(" LD #0101h, A ");
    asm(" LD #255, B ");
    asm(" AND B, A ");
    asm(" STL A, OUT ");
asm(" PORTW OUT, 0x0000 ");

    asm(" PORTW OUT, 0x2000 ");


    /* This sets the BOB bit = 1 for
    data transfer from host to hpiram */

    host2dsp_dnld_kernel();

    asm(" RSBX INTM ");

    asm(" STM #0001h, IMR ");  /* setting up INT0 */


    asm(" SSBX XF ");
```

```
    asm("round: LD #0909h, A ");
    asm(" LD #255, B ");
                   /* Clearing the HINT asserted by the bootloader*/

    asm(" AND B, A ");
asm(" STL A, OUT ");

    asm(" PORTW OUT, 0x0000 ");

    asm(" PORTW OUT, 0x2000 ");


    asm("loop: B $ ");
             /* Wait for external interrupts to occur */


}                  /* end of main */
```

Appendix B   The Kernel

```
***********************************************************************
;
;     Original Author:   GERALD CAPWELL
;     File Name:      DSPKERN.ASM [Default Communication Kernel for
;                     TMS320C542 DSKplus (written using the algebraic
;                     instruction set).
;     Date:           July 19, 1996
;
;
;     Author of Modified version: ARUN CHHABRA
;     File Name:      kern_ram.asm
;                     (the extracted hex file corresponding to this
;                     code is embedded in the file host2.c in the
;                     routine host2dsp_dnld_kernel().
;                     The hex file is extracted by applying the
;                     COFF conversion utility [SPRA573] to this Kernel
;                     code.)
;     Date:           January 30, 1998
; *********************************************************************

                    .def  start
                    .def  main

     .width  80
     .length 55
     .title "TMS320C542 Debugger Kernel loaded via HPI"
     .mmregs

     .bss  COMMAND,1
     .bss  PUDDADDR,1
     .bss  LENGTH,1
     .bss  DUPDADDR,1
     .bss  HPIbuf,1
;======================================================================

; BEGIN OF MAIN PROGRAM

;======================================================================
     .text

; ********************** Buffer Structure ***************************

;  +-------+
;  | 1000h | COMMAND <-- All cases:
;  +-------+              Command   =0 dataul, progdl
;                                   =1 progul, datadl
;                                   =2 run
;  +-------+
;  | 1001h | PUDDADDR<-- Data Download & Program Upload: Source
;  +-------+              address
;                         Data Upload & Program Download: Destination
;                         address
;                         Run: Address to run
```

```
;  +-------+
;  | 1002h | LENGTH  <-- Upload and Download: Length value
;  +-------+             Run: not used
;

;  +-------+
;  | 1003h | DUPDADDR<-- Data Download & Program Upload: Destination
;  +-------+             address
;                        Data Upload & Program Download: Source
;                        address
;                        Run: not used
;  +-------+
;  | 1004h | HPIbuf <-- All cases: Start of HPI comm buffer.
;  +-------+


;***********************************************************************

start:


    ANDM #2300h, ST1  ;Enable global interrupts
STM #0000h, IMR   ;Disable INT2 so that further HINT's will
                      ;not cause bootloader to kick in again.

    STM #7e7fh, SWWSR    ;SWWSR for I/O writes = 1.

    STM stkptr, sp
                   ;sp = #stkptr; set stack location after kernal.

main

clear                 ;this portion is intended to wait for the
                      ;kernel code to be transferred internally
                      ;by the host from HPIram to internal memory
                      ;before proceeding to clear the HPIram.
     LDM HPIC, A

     AND #8h, A

     BC clear, ANEQ



     STM #1000h, AR4        ;This is a loop that will clear out the buffer
                            ;locations

     RPT #2048

     ;all locations from 0x1000 to 2048 locations

     ST #0h, *AR4+     ;ahead, with zeros

execmd
     MVMD AR0, 1010h   ;Move the checksum comparison result that
               ;was stored temporarily in AR0, back into
               ;location 1010h
```

```
;;;;;; For interrupting host in SAM mode ;;;;;
     STM #000ah, HPIC    ;interrupt host and retain SMOD=1



;;;;;;;; For interrupting host in HOM mode if required ;;;;;;;;;;;;
;     STM #8h, HPIC     ;hpic = #8h;
;                 ;interrupt the host
;     NOP          ;The two instructions following a SAM to
;                ;HOM change
;     NOP          ;must be NOP's (see section 9.5.2 in 54x
;                ;user manual).
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


wait
     LDM HPIC, A         ;a = hpic

     AND #8h, A          ;a &= #8h

     BC wait, ANEQ

;if (ANEQ) B wait; Loop till next command ready

;;;;;; needed if using HOM mode for interrupting (not currently used)
;     STM #2h, HPIC      ;hpic = #2h; Shared access mode.
;                 ;(next 3 instructions cannot access HPI
;                ;see section 9.5.2 in C54x user manual )
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


; at this point, calculate the checksum of the values received from
;locations 1010h through 1050h (or end of HPIram...)

    LD #0h, A            ;initialize accumulator
    STM #1010h, AR4      ;starting address

;    STM #1008h, AR3     ;initialize AR3

    STM #07efh, BRC      ;set loop count
    RPTB turn-1
    XOR *AR4+, A
    NOP
turn:
    STL A, AR3
;    STL A, *AR3
                      ;checksum calculated by target of the incoming host
                      ;data is stored in address pointed to by AR3

; having XORed all the relevant values...now check it against the
; checksum sent by the host. So, compare checksum in A with checksum
; sent over by host into location 0x1005.

    MVDM 1005h, AR0   ; checksum sent by host is stored in address
                      ;pointed to by AR0
```

```
    CMPR 0, AR3     ;check to see if received checksum is equal to
                    ;locally calculated checksum
                    ;if equal then TC = 1, else TC = 0

    BC checkpass, TC

    STM #100ah, AR0  ; initialize AR0

    ST #1h, *AR0
                    ;if checksum DOES NOT match, then write a value of
                    ;1 to location 100ah.

    STM #1h, AR0    ;write result of checksum comparisons in AR0
                    ;because location 1010h will be cleared before
                    ;this result is read by host from location 1010h
                    ; the aim is to store the result in AR0 directly,
                    ;since that MMR is never used any place else, so
                    ;hopefully no chance of overwriting it.
    B checkfail

checkpass
    STM #100ah, AR0    ; initialize AR0

    ST #0h, *AR0       ;if checksum DOES match, then write a value of 0
                       ;to location 100ah.
checkfail

    STM #0h, AR0       ;write result of checksum comparisons in AR0
                       ;because location 1010h will be cleared before
                       ;this result is read by host from location 1010h
                       ; the aim is to store the result in AR0 directly,
                       ;since that MMR is never used any place else, so
                       ;hopefully no chance of overwriting it.



     STM #1004h, AR5
                       ;ar5 is set to total number of chunks left to be
                       ;xferred
                       ; ************


     STM #1002h, AR1   ;ar1 = #1002h
                       ; ar1 points to length value


; The three values of COMMAND, DUPDADDR and PUDDADDR are now
; called upon by referring to their actual data memory address values

     STM #1000h, AR3      ;ar3 = data(COMMAND); ar3 = command value
     MVDM 1001h, AR2      ;ar2 = data(DUPDADDR); ar2 = DUPDADDR
     MVDM 1003h, AL       ;al = data(PUDDADDR); acclow has address
                          ; ************

     LD *AR3, B    ;Accumulator B contains the contents of AR3
```

```
     MAR *AR1-      ;indicate one less than the actual length
                    ;for the purposes of the repeat loop below

    RPT *AR1        ;repeat(*ar1)

    WRITA *AR2+     ;prog(A) = *ar2+
                    ;The MVDP instruction can't be used due to
                    ;the necessity of having a hardcode dest. addr.
                    ;The WRITA instruction achieves the same result
                    ;and when in a RPT loop, it auto-increments the
                    ;dest addr specified in the Acc through the PAR
                    ;write contents of locations 1010h onward into
                    ;locations 2000h onwards, respectively


    CMPM *AR5, 0h   ;check if all chunks have been transferred

    BC again, NTC

    B out

again
    STM #1000h, AR4    ;This is a loop that will clear out
    RPT #2048          ;all locations from 0x1000 to 2048 locations
    ST #0h, *AR4+      ;ahead, with zeros

    B execmd

out

    B 2000h            ;Branch to the starting address of the first chunk

stkptr  .end
```

TI Contact Numbers

<u>INTERNET</u>

TI Semiconductor Home Page
www.ti.com/sc

TI Distributors
www.ti.com/sc/docs/distmenu.htm

<u>PRODUCT INFORMATION CENTERS</u>

Americas
Phone +1(972) 644-5580
Fax +1(972) 480-7800
Email sc-infomaster@ti.com

Europe, Middle East, and Africa
Phone
  Deutsch +49-(0) 8161 80 3311
  English +44-(0) 1604 66 3399
  Español +34-(0) 90 23 54 0 28
  Francais +33-(0) 1-30 70 11 64
  Italiano +33-(0) 1-30 70 11 67
Fax +44-(0) 1604 66 33 34
Email epic@ti.com

Japan
Phone
  International +81-3-3344-5311
  Domestic 0120-81-0026
Fax
  International +81-3-3344-5317
  Domestic 0120-81-0036
Email pic-japan@ti.com

Asia
Phone
  International +886-2-23786800
  Domestic
    Australia 1-800-881-011
      TI Number -800-800-1450
    China 10810
      TI Number -800-800-1450
    Hong Kong 800-96-1111
      TI Number -800-800-1450
    India 000-117
      TI Number -800-800-1450
    Indonesia 001-801-10
      TI Number -800-800-1450
    Korea 080-551-2804
    Malaysia 1-800-800-011
      TI Number -800-800-1450
    New Zealand 000-911
      TI Number -800-800-1450
    Philippines 105-11
      TI Number -800-800-1450
    Singapore 800-0111-111
      TI Number -800-800-1450
    Taiwan 080-006800
    Thailand 0019-991-1111
      TI Number -800-800-1450
    Fax 886-2-2378-6808
    Email tiasia@ti.com

TI is a trademark of Texas Instruments Incorporated.

IMPORTANT NOTICE

Texas Instruments and its subsidiaries (TI) reserve the right to make changes to their products or to discontinue any product or service without notice, and advise customers to obtain the latest version of relevant information to verify, before placing orders, that information being relied on is current and complete. All products are sold subject to the terms and conditions of sale supplied at the time of order acknowledgement, including those pertaining to warranty, patent infringement, and limitation of liability.

TI warrants performance of its semiconductor products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

CERTAIN APPLICATIONS USING SEMICONDUCTOR PRODUCTS MAY INVOLVE POTENTIAL RISKS OF DEATH, PERSONAL INJURY, OR SEVERE PROPERTY OR ENVIRONMENTAL DAMAGE (“CRITICAL APPLICATIONS"). TI SEMICONDUCTOR PRODUCTS ARE NOT DESIGNED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS. INCLUSION OF TI PRODUCTS IN SUCH APPLICATIONS IS UNDERSTOOD TO BE FULLY AT THE CUSTOMER'S RISK.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards must be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance or customer product design. TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used. TI's publication of information regarding any third party's products or services does not constitute TI's approval, warranty, or endorsement thereof.

Copyright © 1999 Texas Instruments Incorporated