TEXAS INSTRUMENTS

Using the TMS320C80 Multitasking Executive

Application Report

1998

Digital Signal Processing Solutions

Texas Instruments

Printed in U.S.A., March 1998

SPRA175

Using the TMS320C80 Multitasking Executive

Henry Yiu

SPRA175
March 1998

Printed on Recycled Paper

IMPORTANT NOTICE

Texas Instruments (TI) reserves the right to make changes to its products or to discontinue any semiconductor product or service without notice, and advises its customers to obtain the latest version of relevant information to verify, before placing orders, that the information being relied on is current.

TI warrants performance of its semiconductor products and related software 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, INTENDED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT APPLICATIONS, DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS.

Inclusion of TI products in such applications is understood to be fully at the risk of the customer. Use of TI products in such applications requires the written approval of an appropriate TI officer. Questions concerning potential risk applications should be directed to TI through a local SC sales office.

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

TI assumes no liability for applications assistance, customer product design, software performance, or infringement of patents or services described herein. Nor does TI 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.

Contents

List of Figures

Contents

List of Examples

Using the TMS320C80 Multitasking Executive

ABSTRACT

The *TMS320C80 Multitasking Executive User's Guide* (literature number SPRU112) provides users with a complete list of the multitasking executive functions available. These functions enable users to access a variety of functions that take advantage of the power of this processor family. Multitasking, message, signal, event control, and other functions provide master/parallel processor command-buffer interface functions, which allow multiple applications to run simultaneously and transparently to each other.

1 Introduction

The *TMS320C80 Multitasking Executive User's Guide* (literature number SPRU112) provides users with a complete list of the multitasking executive functions available. Additional information is available from the *TMS320C80 Multimedia Video Processor (MVP) Data Sheet* (literature number SPRS023). 'C80 functions can be grouped into five major categories:

- Task functions
- Message functions
- Signal functions
- Event functions
- Master processor - parallel processor (MP-PP) command-buffer interface functions

Instead of explaining what each function does, this application report shows how to use these functions in a typical 'C8x program. In order to avoid problems, note the sequences and steps that the kernel function performs.

The multitasking executive is not confined within the 'C8x processor itself, but allows interprocessor communications using the same software conventions. This application report focuses on its use with the 'C8x processor only.

Four major kinds of resources are managed by the 'C8x multitasking executive kernel routines:

- Tasks
- Semaphores
- Local ports
- Global ports

The programmer is required to remember the resource ID. Once the resource ID is known, a programmer can access the kernel routines to find the information relevant to that particular resource.

The multitasking executive runs only on the ’C8x master processor. The ’C8x parallel processors are designed to run a single task one time only (single-threaded). Total code size of the multitasking executive is slightly more than 10K bytes, so it is designed to be efficient with code and use very little overhead.

The multitasking executive can best be described by an example. The objective of the example is to build a simple system containing many peripherals, such as a keyboard, display, telecom interfaces, motor drive outputs, and sensor inputs. To perform the functions, which connect all the external peripherals, without multitasking would require a very large CASE statement. The CASE statement would make program management difficult and waste processor resources.

Multitasking is able to handle the multiple situations described. Each processor can manage a major task and numerous internal tasks. Connecting these tasks together requires passing messages and signals. For example, when a keystroke is detected by the keyboard task, its sole function is to determine which key is pressed. Then the keyboard task sends a message to the decoder task to decode the keystroke data. The message contains information about which key is pressed. The decoder task sends a message to the process task to process the keystroke. The system designer decides whether the response time required to handle keystroke data, or any other task data, is fast or slow by assigning a priority to the tasks. Task priorities can be changed easily using function calls to the kernel. On the other hand, without multitasking, a programmer would need to rearrange the complete structure of the program to assign a higher or lower priority to the keyboard handler.

Tasks should be viewed as independent programs running concurrently with other tasks. In actuality, the tasks are not running concurrently, although they actually share finite amounts of processor time. Since there is only one master processor in the ’C8x, most task functions are constructed with an infinite loop inside. The multitasking executive kernel switches the processor from one infinite loop to the other loop(s).

2 Task Kernel Functions

Example 1 shows a typical use of kernel functions for multitasking. The **main()** program is the default task for the master processor and is assigned the lowest priority. The default task, however, cannot be suspended or exited by the user. **TaskInitTasking** can be called once and only once. Once called, the **main()** program is converted to the default task. Most of the **TaskCreate** arguments are described in more detail in the *TMS320C80 Multitasking Executive User's Guide* (literature number SPRU112). The StackSize argument, however, must be large enough to hold the entire context of the task. Should any task be preempted, the kernel saves the current executing state of the task to the stack. Therefore, the stack must be large enough to hold the entire task context. The rest of the kernel functions are self-explanatory.

Example 1. Typical Usage of Task Kernel Functions

```
main ()
{
   TaskInitTasking (); /*Calls this routine once and only once. */
   TaskID = TaskCreate ( -1, TaskFunc, TaskArg, Priority, StackSize );
   /* -1 creates a valid task ID. StackSize must be set large enough for the
      entire task descriptor. */

   TaskResume (TaskID); /* Task is initially suspended until signalled to
      resume. */

   for (;;);
   /* Default task cannot be suspended or blocked, so do something until
      finished. */
}

TaskFunc (arg)   /* This allows receipt of a single argument from main. */
{
   /* The following functions can be placed anywhere and called as many times as
      needed. */
   arg = TaskGetTaskArg (); /* Another way to get a task argument. */
   Pri = TaskGetPriority (); /* Gets its own task priority. */
   TaskID = TaskGetTask (); /* Gets its own task ID. */
   TaskSetPriority (Pri); /* Sets its own priority. */
   TaskYield (-1); /* Yields itself to other tasks of the same priority. */
```

Example 1. Typical Usage of Task Kernel Functions (Continued)

```
    /* The following functions affect the state of the tasks being processed. */
    TaskSuspend (-1); /* Suspends itself. Needs another task to resume task. */
    TaskExit (); /* Exits and frees itself. This is the opposite of TaskCreate. */
}
```

3 Message Kernel Functions

Example 2 shows the use of kernel functions for messages. A port can be viewed as a place to hold messages. A port is not part of a task; therefore, it is good practice to create a port in **main ( )** routine instead of within a task. This ensures that the port IDs are valid before the tasks are resumed and any calls to **TaskReceiveMsg** are made. A special port, called the reclamation port, can be considered as a place to hold empty messages. You can create as many reclamation ports as you like. Empty messages are allocated in memory by a call to **TaskAllocMsg**. Unlike a regular port ID, the reclamation port ID is saved in the message header. A call to **TaskReclaimMsg** without knowing a reclamation port ID can put the message back to its reclamation port.

Each message can exist in one place only. A message can be held in a port or in a task, but not both. For instance, when a task calls **TaskReceiveMsg** to receive a message from a port, the task holds the message. The port no longer contains the message. When a task has finished using the message, it calls either **TaskSendMsg** or **TaskReclaimMsg** to put the message back to the port, and the task no longer holds the message. It is not good practice to send the same message to more than one port and it is not possible for more than one task to receive the same message. Therefore, in order to send the same message content to several tasks, you must allocate as many messages as necessary and duplicate the message content before calling **TaskSendMsg**.

TaskReceiveMsg is a blocking function. This means that a calling task must wait for a message to arrive at its port before the calling task can continue execution. There are a few cases when you might want to use the non-blocking version, the **TaskAcceptMsg** kernel function. One of the cases is shown in the section, “Event Kernel Functions”.

Example 2. Typical Usage of Message Kernel Functions

```
main ()
{
   TaskInitTasking ();

   ReclaimPortID = TaskOpenPort (-1);
   SendPortID = TaskOpenPort (-1);
   /* Generate (-1 means generate) a valid port ID. Two ports are created. One
      is called the reclaimation port, the other is for sending messages. */

   for (i = 0; i < NMSG; i++) {
      MsgPtr = TaskAllocMsg (MsgSize, ReclaimPortID);
      TaskReclaimMsg (MsgPtr);
   }
   /* NMSG is the number of messages that you want to allocate space in the
      reclamation port. MsgSize is the byte size of the message body.
      TaskReclaimMsg puts the newly allocated message back to the reclamation
      port. */

   TaskResume (TaskCreate (-1, TaskFuncA, NULL, Priority, StackSize);
   TaskResume (TaskCreate (-1, TaskFuncB, NULL, Priority, StackSize);
   /* Resuming the tasks after the port IDs are created will make sure that
      the port IDs are valid before the tasks actually use them. */

   for (;;);
}

TaskFuncA ()
{
   for (;;) {
   /* A typical application loops forever and continues to send messages. */

      MsgPtr = TaskReceiveMsg (ReclaimPortID);
      /* This gets an empty but allocated message from the reclaimation port.
         MsgPtr points to the message body, not the message header. */

      /* Write code here to fill in the message info pointed to by MsgPtr .... */

      TaskSendMsg (MsgPtr, SendPortID);
      /* Send the filled message to the Send port. */
```

Example 2. Typical Usage of Message Kernel Functions (Continued)

```
   }/* Loop end */
}

TaskFuncB ()
{
   for (;;) {
   /* A typical application loops forever and continues to receive messages. */

      MsgPtr = TaskReceiveMsg (SendPortID);
      /* This task waits for a message to arrive at this port before
         continuing. */

      /* Write code here to process the message info pointed to by MsgPtr .... */

      TaskReclaimMsg (MsgPtr);
      /* Once processed, send msg back to the reclaimation port as an
         empty message. */

   }/* Loop end */
}
```

4 Signal Kernel Functions

Example 3 shows the usage of kernel functions for signals. Calling conventions are very similar to those of the message kernel functions. A semaphore is viewed as a place to store a signal count. Unlike message kernel functions, which require memory space allocation, there is no need to allocate memory-space for signals. Several kernel functions between messages and signals are placed side-by-side for comparison (see Table 1).

Example 3. Typical Usage of Signal Kernel Functions

```
main ()
{
   TaskInitTasking ();

   SendSemaID = TaskOpenSema (-1, 0);
   ReceiveSemaID = TaskOpenSema (-1, 0);
   /* Generate (using -1) valid semaphore IDs. Initial signal counts are 0. */

   TaskResume (TaskCreate (-1, TaskFuncA, NULL, Priority, StackSize);
   TaskResume (TaskCreate (-1, TaskFuncB, NULL, Priority, StackSize);
   /* Resume the tasks after the semaphore IDs are created will guarantee that
      the semaphore IDs are valid before the tasks actually use them.       */

   for(;;);
}

TaskFuncA ()
{
   for (;;) {
   /* A typical application loops forever to continues to send signals. */

      /* Write code here to process something before signaling TaskFuncB .... */

      TaskSignalSema (SendSemaID);

      TaskWaitSema (ReceiveSemaID);

   } /* Loop end */
}
```

Example 3. Typical Usage of Signal Kernel Functions (Continued)

```
TaskFuncB ()
{
   for (;;) {
   /* A typical application loops forever and continues to receive signals. */

      TaskWaitSema (SendSemaID);

      /* Write code here to process something before signaling TaskFuncA .... */

      TaskSignalSema (ReceiveSemaID);

   } /* Loop end */
}
```

Message / Port	Signal / Semaphore
TaskOpenPort	TaskOpenSema
TaskSendMsg	TaskSignalSema
TaskReceiveMsg	TaskWaitSema
TaskAcceptMsg	TaskCheckSema
TaskClosePort	TaskCloseSema

Table 1. Kernel Functions

5 Event Kernel Functions

Example 4 shows the usage of kernel functions for events. An event is an OR combination of messages and signals. Each task contains one 32-bit event register which can select up to 32 messages or signals to be bound. When a signal or message is bound to a bit in the event register, it cannot be bound to another bit in the same event register or be bound to the event register of another task. Once a message or signal is bound, the task can select which OR combinations of signal or message events to wait for by generating a mask for the event register. A call to **TaskWaitEvents** requires waiting for at least one of the bound and masked bits in the event register to turn from 0 to 1.

The non-blocking functions **TaskCheckSema** and **TaskAcceptMsg** are used here as an example. If no other tasks receive messages or signals from the bound ports or semaphores, the blocking functions **TaskWaitSema** and **TaskReceiveMsg** have the same effect here because the return value of **TaskWaitEvents** already indicates that a signal or message has arrived. However, the kernel does not prevent a bound port or semaphore from having its messages or signals received by other tasks. Therefore, if another task calls **TaskReceiveMsg** to receive the message right after this task has called **TaskWaitEvents**; this task will be waiting and waiting for the message which was stolen by the other task. The use of **TaskAcceptMsg** is used to detect this condition.

Example 4. Typical Usage of Event Kernel Functions

```
/* Assume that the other tasks, ports, semaphores, and main are already
   set up. */

TaskFunc ()
{
    TaskBindSema (SemaAID, Flag0);
    TaskBindSema (SemaBID, Flag1);
    TaskBindPort (PortAID, Flag2);
    /* etc. ... */
    /* Flag0, Flag1, and Flag2 must each be a unique number within each task. They
    can be from 0 to 31 to represent 32 different events. Each semaphore or port
    can be bound to at most one event flag in one task. One event flag can be
    bound to at most one semaphore or port. */
```

Example 4. Typical Usage of Event Kernel Functions (Continued)

```
Mask = (1 << Flag0) | (1 << Flag1) | (1 << Flag2);
/* etc. ... */
/* Select the bound event flags using the OR operator to generate a mask. Then
wait on that mask. */

for (;;) {
/* A typical application can loop forever to continue to wait for events. */

   Events = TaskWaitEvents (Mask); /* Wait until at least one event
   happens. The return value of TaskWaitEvents is a snapshot of the
   event register. If other tasks receive the message right after
   TaskWaitEvents here, then the Events value is no longer valid. */

   Events = Events & Mask; /* Only care about the selected events in this case. */

   switch (Events)
   {
      case (1 << Flag0):
         if (! TaskCheckSema (SemaAID) ) break;
         /* The purpose of this is to clear the semaphore and to make sure
            that only this task receives this semaphore. */

         /* Write code in here to respond to SemaAID.... */

      case (1 << Flag1):
         if (! TaskCheckSema (SemaBID) ) break;

         /* Write code in here to respond to SemaBID.... */

      case (1 << Flag2):
         (MsgPtr = TaskAcceptMsg (PortAID);
         if (MsgPtr == NULL) break;
         /* This removes the message from the port and makes sure that only
            this task receives this message. */

         /* Write code in here to respond to the message.... */

      default:
```

Example 4. Typical Usage of Event Kernel Functions (Continued)

```
            /* It is possible to have more than one event happen at the same
               time. If so, write code here to respond to this situation.... */

      } /* switch end */
   } /* Loop end */
   TaskUnBindSema (SemaAID);
   TaskUnBindSema (SemaBID);
   TaskUnBindPort (PortAID);
   /* Free the event flags. */
}
```

6 Master Processor-Parallel Processor Command Interface

Example 5 shows a typical master processor (MP) C-program interface with a parallel processor (PP) command buffer. In earlier documentation, the PPs were known as advanced digital signal processors (ADSPs), which is now replaced by the term “parallel processor”. The ’C8x PPs receive their commands from the MP through the command-buffer interface. The parameter RAM is used to pass the command buffer. If only a few arguments are to be passed to the PP, it is good practice, but not mandatory, to also put the argument buffer in the parameter RAM. The PP parameter RAM is also used by the PP stack. The programmer must make sure the PP stack does not overwrite the argument buffer or the command buffer.

PpCmdBufInit initializes the command buffer and unhalts the PP. One of the arguments to **PpCmdBufInit** is the number of command buffers. If the PP only receives a command from one MP task, then one command buffer is usually enough. However, it is possible to have more than one MP task issue commands to the same PP. Therefore, the number of command buffers for that PP must be larger than the number of MP tasks that issue commands to that PP. **PpCmdInterp** is the PP command-interpreter program included in the ppcmd.lib library to perform the command-buffer interface. However, programmers are allowed to write other command-interpreter functions if necessary. This is why **PpCmdInterp** is passed to **PpCmdBufInit** as an argument. Once the PP is unhalted, the PP command interpreter is waiting for the MP to issue commands to the PP.

To use the **PpMsgIntSetSema** function, the **main()** program of the MP must first call the **InterruptInit** function once, and only once, to install the default interrupt service routine. **PpMsgIntSetSema** binds the message interrupt from the PP to the semaphore. This only does the binding; that is, if there is a PP message interrupt, the semaphore is signalled, but **PpMsgIntSetSema** does not ensure that there is an interrupt from the PP when the PP finishes with the task. An index value is passed to this function as an argument, which allows up to four different commands, such as those from four different MP tasks to be issued to the same PP by way of the same command buffer interface. For example, MP taskA might want PP0 to run PPfuncA. MP taskB might want PP0 to run PPfuncB. MP taskA can use the **PpCmdBufSetFunc** function to select PPfuncA and MP taskB can select PPfuncB. But when the PP0 is done with either PPfuncA or PPfuncB, the command interpreter might send an interrupt command to the MP to signal that PP0 is done. The MP knows which task has triggered a message interrupt by the index value assigned to the interrupt. When the index value sent by the PP0 matches the index value in **PpMsgIntSetSema**, the selected semaphore is signaled.

Commands to the PP can be issued by **PpCmdBufNotifyIssue** or **PpCmdBufIssue**. The difference between the two is that when notification is used, the PP command interpreter sends a message interrupt back to the MP when the PP is done. In most cases, using notification is preferable because it makes the program easier to follow. No MP processor time is wasted by waiting for notification when using **TaskWaitSema**, which is a blocking function that allows other MP tasks to continue. However, with notification, the MP must run the interrupt service routine each time there is a message interrupt from the PP. This could reduce the efficiency of the MP-PP interface if the MP passes commands to the PP at a high rate but the PP task only takes a short time to complete. In this case, the no-notification method is preferable. The MP finds out if the PP task is done by using the **PpCmdBufBusy** function. If the command buffer is not busy, it does not request the PP command interpreter to send a message interrupt to the MP when the PP is done with the task. However, if the command buffer is busy, the **PpCmdBufBusy** function makes the PP send an interrupt when done. In that case, there is no advantage to using the no-notification method of MP-PP command interfacing.

To avoid having the PP send interrupts to the MP using the no-notification method, the MP programmer might make the MP perform some useful functions between the **PpCmdBufIssue** and the **while(PpCmdBufBusy)** statement. These useful functions for the MP should take longer to complete than the PP task issued. In some less probable cases, when the PP task takes less time, the **while(PpCmdBufBusy)** and **TaskWaitSema** statements make sure that the PP task is completed by waiting for the PP message interrupt before continuing.

The sample program listing shown in Example 5 assumes that this is the only MP task that interfaces with the PP command buffer. However, as mentioned previously, it is possible to have up to four different MP tasks issue commands to the same PP by way of the same command-buffer interface.

Example 5. Typical MP-PP Command Interface - MP C Program

```
CbPtr = PpCmdBufInit ( PPnum, &PpCmdInterp, NCMD );/* PPnum is the PP number.
PPnum can be 0 to 3 for the 'C80. NCMD is the number of command buffers, typically
1 to 4. CbPtr is a pointer to the command buffer. PpCmdInterp is a PP library
function defined in ppcmd.lib. */

SemaID = TaskOpenSema ( -1, 0 );/* Create (-1 means create) a valid semaphore with
initial signal count of 0. */

PpMsgIntSetSema ( PPnum, Index, SemaID ); /* Bind message interrupt from PP to the
semaphore. Index can be 1 to 4. Signals semaphore only if Index matches what PP
writes to the message reply value. */

for (;;) { /* A typical application loops forever to continue to send commands
             to the PP. */

          AgPtr = PPCMDBUF_ADDR  +  PPnum << 12  +  NCMD * sizeof (PPCMDBUF);
          /* PPCMDBUF_ADDR is the start address of the command buffer defined to
             be 0x01000200 in mp_ppcmd.h. PPCMDBUF is the command buffer structure
             defined in mp_ppcmd.h. PPnum << 12 sets the correct address to the
             corresponding PP PRAM. The purpose of this statement is to set
             the argument buffer right after the command buffer. */

          /* Write code in here to fill in the PP arguments pointed to by AgPtr */

          PpCmdBufSetArgs (CbPtr, *AgPtr); /* This puts the argument pointer to
          the command buffer. */

          /* Write code in here to decide what PP function (usually the same) to
             execute .... */

          PpCmdBufSetFunc (CbPtr, Ppfunc); /* This puts the PPfunc address to the
          command buffer. */

#if NOTIFY
          PpCmdBufNotifyIssue (CbPtr); /* This makes the PP send a msg interrupt
          when done. */
          TaskWaitSema (SemaID);/* Issue the command and wait for it to finish. No
          MP processor time is wasted here because TaskWaitSema blocks this task
          and allows other MP tasks to continue. */
#else
```

Example 5. Typical MP-PP Command Interface - MP C Program (Continued)

```
            PpCmdBufIssue (CbPtr); /* No msg interrupt is sent when PP done. */

            /* Do something useful here so that in most cases PP is finished before
               the call to PpCmdBufBusy.... */

            while (PpCmdBufBusy (CbPtr))
            TaskWaitSema (SemaID);
            /* An exit from this while statement indicates that PP is done with the
               task. If command buffer is busy, the PpCmdBufBusy function causes
               PP to send a message interrupt to MP when PP is done. */

#endif /* NOTIFY */
            CbPtr = PpCmdBufNext (CbPtr); /* Advance to next command buffer in the
            circular queue link list. */

} /* Loop end */

TaskCloseSema (SemaID);
/*Close out and finish. */
```

Example 6 shows two tasks, ServerTaskA and ServerTaskB wanting to use the same command buffer of the same PP. A semaphore named SemaIDShare is created to grant exclusive access of the command buffer to one server task at a time until the command is issued. Since there are two server tasks, setting the number of command buffers to equal or greater than two ensures that a free command buffer is always available to the tasks. More than two command buffers may be required if one of the server tasks sends more than one command to the PP before waiting for the PP to complete. However, there is no advantage in doing so. The function GetFreeCmdBuf gets the next empty command buffer. Since there are two server tasks, separate argument buffers must be used. This is why ServerTaskB puts the argument buffer sizeof(ARGBUF) offset from that of ServerTaskA.

Example 6. Two Tasks Sending Commands to the Same Command Buffer

```
#include <mp_ppcmd.h>
static PPCMDBUF *CbPtr; /* Make this visible to InitCmdBuf and GetFreeBuf only. */

void InitCmdBuf ()  {
   PpCmdBufInit ( PPnum, & PpCmdInterp, NCMD ); /* NCMD >= 2 */
   SemaIDShare = TaskOpenSema ( -1 , 1 );
   SemaIDA = TaskOpenSema ( -1 , 0 );
   SemaIDB = TaskOpenSema ( -1 , 0 );
   PpMsgIntSetSema ( PPnum, IndexA, SemaIDA );
   PpMsgIntSetSema ( PPnum, IndexB, SemaIDB );
}

PPCMDBUF *GetFreeCmdBuf ()  { /* Since NCMD >= 2, there must be a free cmnd buf.*/
   while (CbPtr -> flag) CbPtr = PpCmdBufNext ( CbPtr );
   return CbPtr;
}

ServerTaskA ()  {
   PPCMDBUF *CbPtrA;
   for ( ;; ) {
      TaskWaitSema ( SemaIDShare ); /* ServerTaskB cannot use cmnd buf. */
      CbPtrA = GetFreeCmdBuf ()
      AgPtrA = PPCMDBUF_ADDR  +  PPnum << 12  +  NCMD * sizeof (PPCMDBUF)
      + 0 * sizeof (ARGBUF); /* Must use different argument buffers than B. */
      PpCmdBufSetArgs ( CbPtrA, *AgPtrA );
      PpCmdBufSetFunc ( CbPtrA, PPfuncA ); /* Returns IndexA in PPfuncA. */
      PpCmdBufNotifyIssue ( CbPtrA );
      TaskSignalSema (SemaIDShare ); /* ServerTaskB can now use cmnd buf. */
      TaskWaitSema ( SemaIDA );
   }
}
```

Example 6. Two Tasks Sending Commands to the Same Command Buffer (Continued)

```
ServerTaskB ()  {
   PPCMDBUF *CbPtrB;
   for ( ;; ) {
      TaskWaitSema ( SemaIDShare ); /* ServerTaskA cannot use cmnd buf. */
      CbPtrB = GetFreeCmdBuf ()
      AgPtrB = PPCMDBUF_ADDR  +  PPnum << 12  +  NCMD * sizeof (PPCMDBUF)
      + 1 * sizeof (ARGBUF); /* Must use different argument buffers than A. */
      PpCmdBufSetArgs ( CbPtrB, *AgPtrB );
      PpCmdBufSetFunc ( CbPtrB, PPfuncB ); /* Returns IndexB in PPfuncB. */
      PpCmdBufNotifyIssue (CbPtrB );
      TaskSignalSema (SemaIDShare ); /* ServerTaskA can now use cmnd buf. */
      TaskWaitSema ( SemaIDB );
   }
}
```

Example 8 shows a typical PP assembly program for executing commands from the command buffer. The source code of the PP command interpreter is included in the 'C8x software toolkit in the file ppcmd.src. Basically, the command interpreter **PpCmdInterp** does what is shown in Example 8.

Example 7. PP Command Interpreter PpCmdInterp Operation

```
   Sets stack pointers
   Initializes some registers
   Initializes interrupt vectors
   Clears mailbox
   does forever {
      Waits for command buffer flag to be full
      Sets argument pointer to a9 and pushes command buffer pointer to stack
      Calls the command buffer PP function
      Returns from PP function
      Clears command buffer flag to empty
      if (command buffer intCode requires an interrupt to the MP)   {
         Waits for mailbox to be cleared to make sure last mailbox has been read
         by MP.
         Puts command buffer MSGVALUE to mailbox to allow MP to read it.
```

Example 7. PP Command Interpreter PpCmdInterp Operation (Continued)

```
            Message interrupt MP
        }
        command buffer = next command buffer in circular link list
    }
```

The mailbox is located at **pba+0xFC** which is considered a reserved space in the PP parameter RAM. There are two ways the MP can determine if the PP has completed its tasks, by reading the flag using the **PpCmdBufBusy** function or by waiting for the interrupt.

Example 8. Typical MP-PP Command Interface - PP Assembly Program

```
.include “ppcmd.i” ; PP command interface definitions.

; The source code of the PP command interpreter PpCmdInterp is included with
; the ’C8x software tool kit. The PpCmdInterp function manages the command
; interface and calls the ppfunc as a subroutine.

.global _ppfunc; ; Makes ppfunc visible to other files.

; All MP C symbols have an underscore added to the the beginning of the
; symbol after the program is compiled to MP assembly language.

_ppfunc: /*; Start of the PP function.

   d1 =w *( preg + [0] )  ; Get first argument.
   d2 =w *( preg + [1] )  ; Get second argument.
; etc. ...

; The argument is passed via argument pointer preg, which is defined to be
; address register a9 in the include file ppcmd.i. The pointer to the command
; buffer is pushed to the stack before calling ppfunc.

; Write PP code to do the requested processing here ....
; ..................................................

br = iprs     ; return to command interpreter. The following are delay slot
              ; instructions that are executed before the actual return.
              ; The command interpreter uses the intCode value (PPCMDBUF.INTCODE)
              ; of the command buffer to determine if a message interrupt should
              ; be sent to the MP.
```

Example 8. Typical MP-PP Command Interface - PP Assembly Program (Continued)

```
a10 = w *sp   ; get command buffer pointer from the stack.
*a10.PPCMDBUF.MSGVALUE = Index ; Put Index value to MSGVALUE to signal the
              ; corresponding semaphore.

; MSGVALUE is defined to be the sixth word of PPCMDBUF, both are defined in
; ppcmd.i. Index can be 1 to 4. It must match the MP Index value in the
; argument of PpMsgIntSetSema to cause MP to receive the semaphore.
```

7 Summary

This application report discusses the capabilities of using the multitasking executive by showing some sample programs and calling conventions. The example programs shown in this report cannot be compiled because many of the C programming syntax and data-handling rules are omitted to make these examples more readable. The author can make the original program code available on the internet for those who desire to use this capability. The author can be contacted at *hyiu@ti.com* for questions or comments.

IMPORTANT NOTICE

Texas Instruments (TI) reserves the right to make changes to its products or to discontinue any semiconductor product or service without notice, and advises its customers to obtain the latest version of relevant information to verify, before placing orders, that the information being relied on is current and complete.

TI warrants performance of its semiconductor products and related software 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, INTENDED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT APPLICATIONS, DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS.

Inclusion of TI products in such applications is understood to be fully at the risk of the customer. Use of TI products in such applications requires the written approval of an appropriate TI officer. Questions concerning potential risk applications should be directed to TI through a local SC sales office.

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

TI assumes no liability for applications assistance, customer product design, software performance, or infringement of patents or services described herein. Nor does TI 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.