TMS320 DSP
DESIGNER'S NOTEBOOK

Writing TMS320C8x PP Code Under the Multitasking Executive

APPLICATION BRIEF: SPRA269

Leor Brenman
Digital Signal Processing Products
Semiconductor Group

Texas Instruments
May 1996

TEXAS INSTRUMENTS

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 application 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.

TRADEMARKS

TI is a trademark of Texas Instruments Incorporated.

Other brands and names are the property of their respective owners.

CONTACT INFORMATION

US TMS320 HOTLINE	(281) 274-2320
US TMS320 FAX	(281) 274-2324
US TMS320 BBS	(281) 274-2323
US TMS320 email	dsph@ti.com

Contents

Figures

Examples

Writing TMS320C8x PP Code Under the Multitasking Executive

Abstract

The purpose of this document is to supplement the TMS320C8x Multitasking Executive (ME) User's Guide and to provide useful guidelines for writing Parallel Processor (PP) assembly language or C code that can run under the executive.

ME tasks running on the Master Processor (MP) may issue commands to the PPs. These commands are requests for the PPs to execute a routine and then, optionally, interrupt the MP to notify the MP that the routine is done. The routine that the PP executes is a PP assembly language subroutine or a C routine. In order to write PP subroutines or C routines that run under the ME, it is important to understand the PP environment set up by the ME. Tasks running under the ME communicate with the PPs via a PP command interpreter. The PP command interpreter is a PP assembly language routine that waits for commands to be sent from tasks running on the MP.

Design Problem

How do I write TMS320C8x Parallel Processor (PP) code to run under the Multitasking Executive (ME)?

Solution

Introduction

The purpose of this document is to supplement the TMS320C8x ME User's Guide and to provide useful guidelines for writing PP assembly language or C code that can run under the executive.

ME tasks running on the Master Processor (MP) may issue commands to the PPs. These commands are requests for the PPs to execute a routine and then, optionally, interrupt the MP to notify the MP that the routine is done. The routine that the PP executes is a PP assembly language subroutine or a C routine. In order to write PP subroutines or C routines that run under the ME, it is important to understand the PP environment set up by the ME. Tasks running under the ME communicate with the PPs via a PP command interpreter. The PP command interpreter is a PP assembly language routine that waits for commands to be sent from tasks running on the MP.

The main guidelines for launching a PP routine from an ME task are outlined below:

MP:
ME task initializes command buffers in PP parameter RAM.
ME task installs PP command interpreter entry point address as the PP task vector.
ME task un-halts PP causing it to execute PP command interpreter.
ME task installs ISR to respond to PP command interpreter interrupt when the command is complete (optional).
ME task initializes argument buffers in PP parameter or data RAM.
ME task loads command buffer with parameters and issues command to PP.

PP:
PP routine written as a standard called routine.
PP uses a9 or d1 as address of argument buffer.
PP program returns stack to value upon entry to routine.
PP program does not overwrite memory used by ME.
PP program restores interrupt environment prior to exiting.

The remainder of this document details these steps.

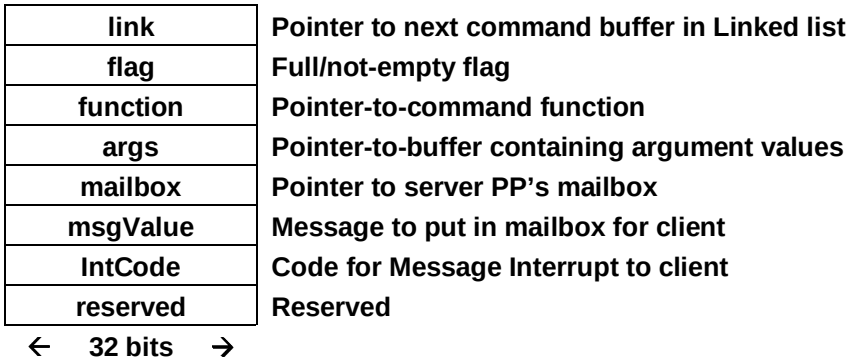
MP Responsibilities

ME tasks use functions in the PP command library, mp_ppcmd.lib, to set up command buffers in PP parameter RAM, load the PP command interpreter on the PP, un-halt the PP, and issue commands to a PP by loading the command buffers. The PP command interpreter is a PP routine that is supplied with the ME in the library ppcmd.lib. Its entry point is PpCmdInterp. Once the command buffers are created and the PP is running the PP command interpreter, the PP command interpreter waits for the first command buffer to be filled by an ME task. The command buffers are allocated in PP parameter RAM at an offset of 0x200 from the start of PP parameter RAM. Each command buffer is 32 bytes long. This requires the PP programmer not to use this area of parameter RAM.

When an ME task wants the PP to execute a routine it issues a command to the PP command interpreter via the command buffers. To issue a command, the MP task loads the command buffer with the following items: (1) the address of the entry point of the PP C or assembly language routine, (2) the address of the argument buffer for the PP routine, (3) an optional interrupt code for the PP command interpreter to load into the cmnd word to interrupt the MP when the PP completes the desired routine, and (4) an optional message value for the PP command interpreter to load into the mailbox before interrupting the MP. (When interrupted, the MP interrupt service routine (ISR) reads the mailbox to identify the source of the interrupt and then clears the mailbox). The mailbox is at an offset of 0xf0 in PP parameter RAM. This value is hard coded into both the PP command interpreter and the ME. PP user programs must not overwrite this location

The complete command buffer is described in Figure 1.

Figure 1. Command Buffer



The command buffer is a C data structure called PPCMDBUFF.

Since the ME tasks can start a PP routine on any PP, the location and size of the particular arguments for a particular PP may be determined at runtime based on processor loading. This requires the PP and ME programmer to establish a protocol for avoiding conflicts in memory usage since the location and size of the arguments for a particular PP are not fixed at link time. For example, one could restrict the PP routine from accessing parameter RAM entirely so that PP parameter RAM can be used for command buffers and argument buffers. As an example, the address of the argument buffer can be calculated and loaded into the command buffer in the following fashion:

```
ASTRUCT *ppArg;

ppArg = (ASTRUCT *)
                (0x1000200+0x20*numCmdBufs+(ppNum<12));

PpCmdBufSetArgs(cmdBuf, ppArg);
```

If another argument buffer is to be set up, say for the second command, then the ME programmer must know, or determine, how large the argument buffer is. This is easily accomplished using the C sizeof( ) function. Then, for example, the second argument buffer can be placed in memory immediately following the first.

Alternatively, if a fairly static system is being designed where each PP always executes the same routine and the argument buffer is always the same, then the argument buffers can be statically allocated in a particular PP's memory. This is done by having the ME task create a data structure that is linked into the particular PP's memory. See Linking C Data Objects Separate From the .bss Section (Literature Number SPRA258), for details. The ME task still needs to load the argument buffer address into the command buffer.

Once all elements have been written to the command buffer, the ME task issues the command by setting the flag entry to 1 which causes the PP command interpreter to call the routine or subroutine specified in the command buffer. The following code segment is an example of how to start a routine on PP0 from an ME task:

Example 1. Code Listing

```
#include         ask.h    /* incl header file for ME */
#include         .h       /* incl header file for PP cmd interp */
...
void ppFunc(ARG *);       /* prototype PP routine */
...
PPCMDBUF *cmdBuf;         /* create ptr to command buffer(s) */
...
/*
* create 2 command buffers in PP0's parameter RAM, install PP
* command interpreter and un-halt PP0
*/
cmdBuf = PpCmdBufInit(0, &PpCmdInterp, 2);

/*
* load address of routine and argument buffer into command buffer
*/
PpCmdBufSetFunc(cmdBuf, &ppFunc);
PpCmdBufSetArgs(cmdBuf, ppArg)

/*
* issue command buffer (set flag=1)*/
*/
PpCmdBufIssue(cmdBuf);
...
```

Once the ME task calls the PpCmdBufIssue( ) routine, the PP command interpreter reads the command buffer's flag and starts the PP routine. If the ME wants to be interrupted upon completion of the PP routine, the ME task must request to be interrupted by the PP command interpreter. This is accomplished through the command buffer and requires that an ISR be installed in the MP to respond to the particular PP's message (MSG) interrupt. (See the ME function: PpCmdBufNotifyIssue( )).

PP Programming Guidelines

When the PP command interpreter reads a ready command buffer (flag=1), it reads the address of the argument buffer from the command buffer and places it in a9 and d1. PP assembly language subroutines can use a9 to read the arguments from the argument buffer while C routines expect d1 to contain the first passed parameter which in this case is a pointer to the argument buffer. The PP command interpreter pushes all of the registers it was using onto the stack and makes a call to the routine that was specified in the command buffer.

Entry Point

PP routines running under the ME can either be ASM routines or C routines. In order to create PP routines to run under the ME, the entry point (function label) must be made visible to the ME task so that it can load this address into the command buffer as shown in the previous ME code example. The .system assembly language directive and the C shared keyword make the entry point visible to MP programs for PP-defined ASM and C routines, respectively. The following two examples illustrate the use of .system assembler directive and shared C keyword for the previous example:

Ex2 (ASM)

```
                .system _ppfunc
_ppFunc:        d5=h *a9++
                ...
```

Ex2b (C)

```
shared void ppFunc(ARG *argPtr)
{
   int Acount = argPtr-imageSize;
                  ...
```

All arguments are passed to the routine in the argument buffer and any returned value should also be placed in the argument buffer.

Accessing Arguments

The argument buffer, set up by the ME task, is typically a C data structure. PP C code can share that data structure via a header file for accessing the arguments from the argument buffer. However, PP assembly language routines have no way of sharing C data structures with C routines running under the ME on the MP. The PP assembler has facilities for creating assembly language data structures. The PP programmer can make the assembly language data structure match the C data structure and this will help automate accessing arguments. For example, consider the following data structure used by the ME to set up an argument buffer:

```
typedef struct {
   int arraySize;
   int *arrayPtr;
   int result;
} ASTRUCT;
```

The corresponding PP assembly language data structure could be created as follows:

```
            .struct
sASIZE      .word
sAPTR       .word
sRESULT     .word
            .endstruct
```

To access the element arraySize (sASIZE in assembly language) use the following PP assembly language syntax:

```
d0 = *a9.sASIZE
```

Register Usage

Since the PP command interpreter pushes all the PP data and address registers it uses onto the stack, the PP programmer is free to use all of the data and address registers for implementing their subroutine. However, it is up to the user to maintain the interrupt environment set up by the PP command interpreter. Since the address of the argument buffer is in a9 and d1 assembly language routines can use a9 to access the arguments and can disregard d1 while C programs, by definition, expect to find the first (and in this case, the only) passed argument in d1. Therefore, writing both C and/or assembly language routines are easily supported.

Exit

Since the PP routine was called from the PP command interpreter, the PP routine can exit simply as a called routine by branching to the iprs register. This is handled automatically by the C compiler for C routines. Also, assembly language routines must return the stack pointer to its position on entry to the routine. Again, this is handled automatically by the C compiler for C routines.

Summary

The file me_exmpl.tar contains a simple example of running a PP assembly language routine as well as a C routine from the ME. Figure 2 summarizes the concepts discussed in this document.

Figure 2. Graphical Description of the Concepts