Application Report
SPRA556B- February 2000

A Multichannel/Algorithm Implementation on the TMS320C6000 DSP

Xiangdong Fu
Zhaohong Zhang

C6000 Applications

ABSTRACT

This application report describes how to build digital signal processing (DSP) algorithm modules for multichannel applications running on the Texas Instruments (TI™) TMS320C6000 DSP. In addition, the basic requirements for multichannel/algorithm implementations, that is, reentrant and relocatable, are presented along with practical approaches for multichannel implementation. This document includes example programs to illustrate those approaches.

Contents

List of Figures

1 Introduction

Multichannel/algorithm applications often require the following:

- One DSP algorithm is applied to multiple channels.
- The type of algorithms and the channels being applied can change during run-time.

To meet these requirements, each individual algorithm must be *reentrant*; that is, the algorithm program may be entered repeatedly or before previous executions have been completed, and each execution of the program is independent of all other executions. This requires that the algorithm not be self-modifying or, equivalently, that all program and data be read-only.

Because typical DSP algorithms almost never modify their own code, this application report does not discuss this case. However, many existing or off-the-shelf algorithm codes, such as the fixed-point C source codes of the ITU standard voice coders (downloaded from the ITU's web site), cannot directly be used in a multichannel/algorithm system because of non-constant global and static variables. This application report presents the basic requirements for multichannel implementation for DSP algorithms as well as practical approaches for specific algorithm implementation.

The second typical requirement for the multichannel implementation of a DSP algorithm is that all its program and data be fully relocatable.

The algorithm reentrance and relocatability are also required by the eXpressDSP™ Algorithm Standard (XDAIS). In order to comply with XDAIS, however, the algorithm needs to follow more restrict rules and to support a generic algorithm API, the IALG interface, as well as an algorithm specific API. Details on the XDAIS standard and on how to make algorithms XDAIS compliant are beyond the scope of this application report. Refer to section 5, numbers [3][4][5][6][7] for more information.

2 Basic Requirements for Multichannel Implementation

2.1 Reentrant Program

In general, there are two kinds of variables in any given program:

- Variables with local lifetimes
- Variables with global lifetimes

Variables with local lifetimes, such as automatic and register variables, are allocated new storage each time execution control passes to the block in which they are defined. When execution returns, the variables no longer have meaningful values.

eXpressDSP is a trademark of Texas Instruments Incorporated.

On the other hand, variables with global lifetimes, such as static and global variables, are allocated when the program begins and de-allocated only when the whole program ends.

All variables with global lifetimes must be constant to make a program reentrant. This is illustrated in the following IIR filter example.

In the example below, *i* and *result* are automatic variables with local lifetimes. They are allocated new storage each time execution control passes to function *iir()* and de-allocated when *iir()* returns. The variables *q* and *state* are static variables with global lifetimes. The only difference between these two is that *q* is a constant table and values in *state* are variables.

```
#define ORDER 2
const static float q[ORDER] = { 0.9, -0.2 };
static float state[ORDER] = { 0.0,0.0 };
float iir(float in)
{
    int i;
    float result = in;
    /* iir filtering */
    for(i = 0; i < ORDER; i++){
        result += q[i] * state[i];
    }
    /* update filter state */
    for( i = 0; i < (ORDER-1); i++){
        state[i+1] = state[i];
    }
    state[0] = result;
    return result;
}
```

Figure 1. Non-Reentrant iir Filter Function

Now suppose this IIR filter is applied to two channels of signals with a main function that resembles the following:

```
void main()
{
    volatle int data_available = 1;
    float in1, in2, out1, out2;
    while(data_available)
    {
         /* Get input from channel 1 */
         in1 =  GetIn1();
         /* iir filtering */
         out1 = iir(in1);
         /*send output to channel 1 */
         sendOut1(out1);

         /* Get input from channel 2 */
         in2 =  GetIn2();
         /* iir filtering */
         out2 = iir(in2);
         /*send output to channel 2 */
         sendOut1(out2);
    }
}
```

Figure 2. Main Function Calling Non-Reentrant iir (Does Not Work)

The code in does not work properly because the iir filter function is not reentrant. The existence of non-constant static array state[] causes this problem because each time iir() executes, the state[] gets modified, which means the program is self-modifying and hence disqualifies itself to be reentrant.

Physically, when IIR filters are applied to different signal channels, each channel should maintain its own filter state memory. But in the program above, the state memory belongs to the iir() function itself and is shared by all channels. To make the program reentrant, the iir() function is modified as follows and the main function is modified accordingly.

```
#include <stdlib.h>
#define ORDER 2
const static float q[ORDER] = { 0.9, -0.2 };

float* new_iir(){
  return (float *)calloc(sizeof(float), ORDER);
}
void delete_iir(float state[]){
    free(state);
}
float iir(float state[], float in){
    int i;
    float result = in;
    /* iir filtering */
    for(i = 0; i < ORDER; i++){
        result += q[i] * state[i];
    }
    /* update filter state */
    for( i = 0; i < (ORDER-1); i++){
        state[i+1] = state[i];
    }
    state[0] = result;
    return result;
}
```

Figure 3. Modified Reentrant iir Filter Function

Notice that two new functions are added to complement the iir() function: the new_iir() and the delete_iir() functions are used to allocate and de-allocate the memory of filter states for each channel. The static variable array state[] no longer exists.

Now no non-constant variable with global lifetime exists in the program; hence, the program is reentrant. The main() function, which has also been modified accordingly, will work fine with the IIR filter applied to two independent signal channels concurrently.

```
void main()
{
    float in1, in2, out1, out2;
    float *state1, *state2;
    volatile int data_available = 1;
    /* create iir filter for channel 1*/
    state1 = new_iir();
    state2 = new_iir();

    while(data_available)
    {
         /* Get input from channel 1 */
         in1 =  GetIn1();
         /* iir filtering */
         out1 = iir(state1, in1);
         /*send output to channel 1 */
         sendOut1(out1);

         /* Get input from channel 2 */
         in2 =  GetIn2();
         /* iir filtering */
         out2 = iir(state2, in2);
         /*send output to channel 2 */
         sendOut1(out2);
    }
    delete_iir(state1);
    delete_iir(state2);
}
```

Figure 4. Modified Main Function Calling Reentrant iir() (Works Well)

In general, To make a DSP algorithm reentrant, non-constant static and global variables, which have global lifetime, must be removed.

2.2 Relocatable Data

Because of the enormous processing power of the C6000 DSP, a single DSP core in real time can handle multiple channels of complicated DSP algorithms, such as ITU speech coders. However, the on-chip data memory is relatively limited and not all data can be placed on-chip all of the time in a multichannel, multi-algorithm application. This requires all data (context as well as table) in each algorithm to be relocatable, such that they can be moved on- and off-chip during context switching.

To show how this is achieved, we revisit the iir example. Changes have been made to make data relocatable. Now context switching is performed each time the function is called. Also, filter coefficients are different for channel 1 and channel 2 now to illustrate the idea of context switching more clearly. Figure 5 and Figure 6 show the program.

```
#include <stdlib.h>
float* new_iir(int size){
   float temp* = (float *)calloc(size,1);
}
void delete_iir(float* state){
    free(state);
}
float iir(float *state, float *q, float in){
    int i;
    float result = in;
    /* iir filtering */
    for(i = 0; i < ORDER; i++){
        result += q[i] * state[i];
    }
    /* update filter state */
    for( i = 0; i < (ORDER-1); i++){
        state[i+1] = state[i];
    }
    state[0] = result;
    return result;
}
```

Figure 5. Modified IIR Filter Function With Reentrant Program and Relocatable Data

```
#define ORDER 2
/* iir filter coefficients table is located off-chip originally */
const static float ext_q1[ORDER] = { 0.9, -0.2 };

const static float ext_q2[ORDER] = { -0.9, -0.2 };

void main()
{
    float in1, in2, out1, out2;
    /* pointers to external memory */
    float *ext_state1, *ext_state2;
    /* pointers to internal data memory */
    float int_state[ORDER];
    /* allocate on-chip memory for coeff-table */
    float int_q[ORDER];
    volatile int data_available = 1;
    /* create iir filter for channel 1*/
    ext_state1 = new_iir(ORDER*sizeof(float));
    ext_state2 = new_iir(ORDER*sizeof(float));

    while(data_available)
    {
         /* Get input from channel 1 */
         in1 =  GetIn1();
        /* context switching */
         copy(ext_state1, int_state,ORDER*sizeof(float));
         copy(ext_q1, int_q,ORDER*sizeof(float));
         /* iir filtering */
         out1 = iir(int_state, int_q, in1);
         /*send output to channel 1 */
         sendOut1(out1);
         /* context switching */
         copy(int_state, ext_state1,ORDER*sizeof(float));
         copy(ext_state2, int_state,ORDER*sizeof(float));
         copy(ext_q2, int_q,ORDER*sizeof(float));
         /* Get input from channel 2 */
         in2 =  GetIn2();
         /* iir filtering */
         out2 = iir(int_state, int_q, in2);
         /*send output to channel 2 */
         sendOut1(out2);
         /* context switching */
         copy(int_state, ext_state2,ORDER*sizeof(float));
    }
    delete_iir(state1);
    delete_iir(state2);
}
```

Figure 6. Modified Main Function Performing Run-Time Context Switching

3 Practical Approaches

Now that we understand the basic requirements of multichannel implementation for DSP algorithms, we consider more complex program examples to see how this can be done in reality.

3.1 Functions With Multiple Static/Global Variables

As the following example shows, all of the static/global variables must be grouped together in a well-defined structure.

```
static int x;
static float y;
static short z[100];
void foo(){
  ...
  ...
}
```

Figure 7. Original Non-Reentrant Function foo()

```
typedef {
   int x;
   float y;
   short z[100];
} foo_mblk, *foo_handle;
void foo_init(foo_handle *hmem){
    *hmem =  (foo_mblk *)malloc(sizeof(foo_mblk));
}
void foo_free(foo_handle *hmem){
   free(*hmem);
}
void foo(foo_handle* hmem){
  ...
  ...
}
```

Figure 8. Modified Reentrant Function foo()

3.2 Original Module and Its Submodules All Have Static/Global Variables

```
static short z[100];
short  b;
void foo_son1(...){
  ...
}
static int w[20];
static short q[120];
static short p[200];
int a[20];
void foo_son2(...){
  ...
}
static short c;
static int m[20];
extern short b;
extern int a[20];
void foo_parent(){
  ...
  foo_son1(...);
  ...
  ...
  foo_son2(...);
  ...
}
```

Figure 9. Original Module and Its Two Submodules All Having Global/Static Variables

Module foo_parent() has two submodules, foo_son1() and foo_son2(). All three modules have static/global variables.

Two approaches can usually be used to perform multichannel implementation:

1. Define all static and global variables in one big structure and pass the pointer of the structure to all submodules. This approach is simple and straightforward and works well if both foo_son1() and foo_son2() are called only by foo_parent().

```
void foo_son1(
foo_parent_handle *hmem...){
  ...
}
void foo_son2(
foo_parent_handle hmem...){
  ...
}
typedef struct{
  short c;
  int m[20];
  short z[100];
  short b;
  int w[20];
  short q[120];
  short p[200];
  int a[20];
} foo_parent_mblk, *foo_parent_handle;
void foo_parent(foo_parent_handle pmem){
  ...
  foo_son1(pmem, ...);
  ...
  ...
  foo_son2(pmem, ...);
  ...
}
void foo_parent_init(foo_parent_handle *hmem){
    *hmem =  (foo_parent_mblk *)malloc(sizeof(foo_parent_mblk));
}
void foo_parent_free(foo_parent_handle *hmem){
   free(*hmem);
}
```

Figure 10. Modified Module and Its Two Submodules With All Global/Static Variables Defined into One Top Structure

2. Static variables in each module are grouped into different structures. Global variables are defined in the structure of the parent module and passed into the submodule as needed.

```
typedef struct{
  short z[100];
}foo_son1_mblk,
*foo_son1_handle;
void foo_son1_init(
  foo_son1_handle *hmem){
   hmem = (foo_son1_handle) malloc(sizeof(foo_son1_mblk));
}
void foo_son1_free(
foo_son1_handle *hmem){
   free(*hmem);
}
void foo_son1(
foo_son1_handle *hmem, int a, ...){
  ...
}
typedef struct{
  int w[20];
  short q[120];
  short p[200];
}foo_son2_mblk, *foo_son2_handle;
void foo_son2_init(
  foo_son2_handle *hmem){
   hmem = (foo_son2_handle) malloc(sizeof(foo_son2_mblk));
}
void foo_son2_free(
foo_son2_handle *hmem){
   free(*hmem);
}
void foo_son2(
foo_son2_handle *hmem, int a, ...){
  ...
}
typedef struct{
  short c;
  int m[20];
  short b;
  int a[20];
  foo_son1_mhandle pfoo_son1_mem;
  foo_son2_mhandle pfoo_son2_mem;
} foo_parent_mblk, *foo_parent_handle;
void foo_parent(foo_parent_handle pmem){
  ...
  foo_son1(pmem->pfoo_son1_mem, pmem->b, ...);
  ...
  ...
  foo_son2(pmem->pfoo_son2_mem,pmem->a, ...);
  ...
  ...
}
void foo_parent_init(foo_parent_handle *hmem){
    *hmem =  (foo_parent_mblk *)malloc(sizeof(foo_parent_mblk));
     foo_son1_init(&((*hmem)->pfoo_son1_mem));
     foo_son2_init(&((*hmem)->pfoo_son2_mem));
}
void foo_parent_free(foo_parent_handle *hmem){
   fool_son1_free(&((*hmem)->pfoo_son1_mem));
   fool_son2_free(&((*hmem)->pfoo_son2_mem));
   free(*hmem);
}
```

Figure 11. Modified Module and Its Two Submodules that Each Defined Their Own Original Global/Static Variables into Self-Maintained Structures

3.3 Constant Tables

Although constant tables do not have to be modified for algorithm processing, they must be relocatable so that they can be moved on-chip before processing starts to increase processing speed. When a system contains many algorithms, there is not enough on-chip data memory to host every table of those algorithms. Normally, the tables are defined in external data memory and moved to internal data memory before processing starts. It takes time for an algorithm with many constant tables to determine which table is used by which subroutine. Use approaches similar to how context data are grouped. Constant tables can be made relocatable in a very short period of time.

The approach consists of three steps.

1. Organize all of the constants into one data structure. Do not define any substructure, as was done in the previous section.
2. Define a data pointer with the structure type. Because this pointer is a global variable, you must choose a unique name, such as xyz_tablePtr.
3. Add data definitions to the files where those constants were originally defined. An example of this approach is illustrated from Figure 12 through Figure 14.

```
constant int x[4] = {1,2,3,4};
constant int y  = 10;
constant int z = 20;
void foo(…)
{
  ...
  ...
}
```

Figure 12. Function foo() Module With Non-Relocatable Constants

```
typedef struct
{
   int x;
   int y;
   int z;
}XYZ_TABLE;
XYZ_TABLE XYZ_Table =
{
     5,
     10,
     20
};
XYZ_TABLE *XYZ_TablePtr;
```

Figure 13. Data Structure and Pointer Definition for Relocatable Constants

```
#define  x   XYZ_TablePtr->x
#define  y   XYZ_TablePtr->y
#define  z   XYZ_TablePtr->z
foo()
{
  ...
  ...
}
```

Figure 14. Modified foo() Module With Relocatable Constants

3.4 Data Relocation

If there are no static/global/constant pointers, the previous approaches have already made those context and table memory blocks relocatable. But if there are static/global/constant pointers, those types of pointers exist in the code and must be changed to be offsets of a base pointer. Figure 15 shows how to change pointers to offsets.

```
static short z[100];
static short *pz = &z[10];
void foo(){
  ...
  ...
}
```

Figure 15. Non-Reentrant Function foo() With One Global Pointer (pz)

```
typedef {
  short z[100];
  short *pz;
} foo_mblk, *foo_handle;
void foo_init(foo_handle *hmem){
    *hmem =  (foo_mblk *)malloc(sizeof(foo_mblk));
    /* adjust the pointers, only save the offset */
    (*hmem)->pz = (short *)(((char *)&((*hmem)->z[10]) )-(int)(*hmem));
}
void foo_free(foo_handle *hmem){
   free(*hmem);
}
void foo(foo_handle hmem){
/* adjust the pointers before processing  */
    hmem->pz = (short *)(((char *)(hmem->pz ))+(int)(hmem));
  ...
  ...
/* adjust the pointers before processing  */
    hmem->pz = (short *)(((char *)(hmem->pz ))-(int)(hmem));
}
```

Figure 16. Modified Reentrant Function foo() With Relocatable Context

3.5 Memory—To Allocate or Not to Allocate

In the previous examples, all memory blocks are allocated inside the XXX_init() function by calling standard C run-time library functions malloc() and are freed in XXX_free() function by calling free(). Thus, the XXX module is the owner of its own context memory block.

However, in real applications the framework owns all system resources, including memory space. Individual algorithm modules cannot allocate or free their own memory or other resources. In this case, memory is allocated outside the XXX_init() and a pointer or pointers are passed into XXX_init() to perform data initialization. Similarly, XXX_free()does not free any memory block now but performs only data cleanup or even nothing at all.

3.6 Interruptability

To fully utilize the C6000's software pipeline for maximum performance, interrupts must be disabled before the execution of critical sections, such as tight loops. The maximum interrupt block period depends on system requirements.

4 Summary

In general, two requirements must be fulfilled to perform multichannel implementation for a DSP algorithm: reentrant and relocatable. This application report discussed these two requirements in detail and provided practical approaches for complicated situations as well as example programs.

5 References

Refer to the following application reports to learn more about the multichannel/algorithm system on TMS320C6000.

1. Xiangdong Fu and Zhaohong Zhang, *TMS320C6000 Multichannel Vocoder Technology Demonstration Kit Host Side Design*, SPRA558.
2. Xiangdong Fu, TMS320C6000 *Multichannel Vocoder Technology Demonstration Kit Target Side Design*, SPRA560.
3. *eXpressDSP Algorithm Standard (Rules & Guidelines)*, SPRU352.
4. *eXpressDSP Algorithm Standard (API Reference)*, SPRU360.
5. Stig Torud, *Making DSP Algorithms Compliant to the eXpressDSP Algorithm Standard*, SPRA579.
6. Carl Bergman, *Using the eXpressDSP Algorithm Standard in a Static DSP System*, SPRA577.
7. Carl Bergman, *Using the eXpressDSP Algorithm Standard in a Dynamic DSP System*, SPRA580.

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.