Application Report
SPRA626 - March 2000

Multichannel Assembly Code Processing on C5000™ DSP Family Devices

Servanne Dauphin
Wireless Communications

ABSTRACT

This document describes the multichannel processing methodology used in multichannel algorithm implementations available for wireless applications where the optimization level is very high. This efficient and reliable processing methodology can also be used in any domain where it is necessary to run several instances of the same algorithm. With little overhead, this processing methodology adds value to fully optimized assembly algorithms, and takes advantage of some C5000™ DSP architecture features that allow you to write multichannel assembly code with almost no impact on code size or performance in comparison with single-channel code versions.

This document describes how to use the methodology to write multichannel assembly code, provides examples of multichannel processing implementation on the TMS320C54x™ DSP, and describes how to avoid some common problems related to writing multichannel code.

Contents

List of Figures

List of Tables

List of Examples

1 Multichannel Code Processing

With increased digital signal processor (DSP) performance, new applications have emerged that require processing the same algorithm for several instances on the same device. This multichannel processing ability is particularly necessary for base station applications, or modem pools, where each instance of the algorithm does the processing for one communication channel. Using this multichannel processing saves program memory because it allows you to run several instances of the same algorithm using the same code. With its low power consumption and powerful architecture, the Texas Instruments (TI) C5000™ DSP family is well-suited for these telecommunication applications.

The purpose of multichannel code processing is to assign to each instance of the assembly code (that is, each channel) a unique memory space containing all the data related to this instance. Multichannel code processing can easily be implemented on TI C5000™ DSPs by using a specific addressing mode called direct addressing. When using this addressing mode, the generated address depends on a register value, either the Data Page pointer (DP) or the Stack Pointer (SP). DP-related addressing is the most convenient method for assembly code; thus, it is the only mode discussed in this document.

It is possible to assign a unique data page number to each instance of the assembly code to link it with the corresponding data page. Unfortunately, the size of each data page is limited to 128 words, which is too small for most applications. But, it is possible to overcome this limitation by using pointers to address large data arrays, which can be located anywhere in memory.

1.1 DP-Referenced Direct Addressing Description

When direct addressing is used, only the 7 lower bits of the 16-bit data memory address (DMA) are encoded in the opcode. When the instruction is executed, the 7 bits are concatenated with the 9-bit DP (Data Page Pointer), to produce a full 16-bit address, as shown in Figure 1. This addressing mode divides the memory space in 512 data pages, of 128 words each.

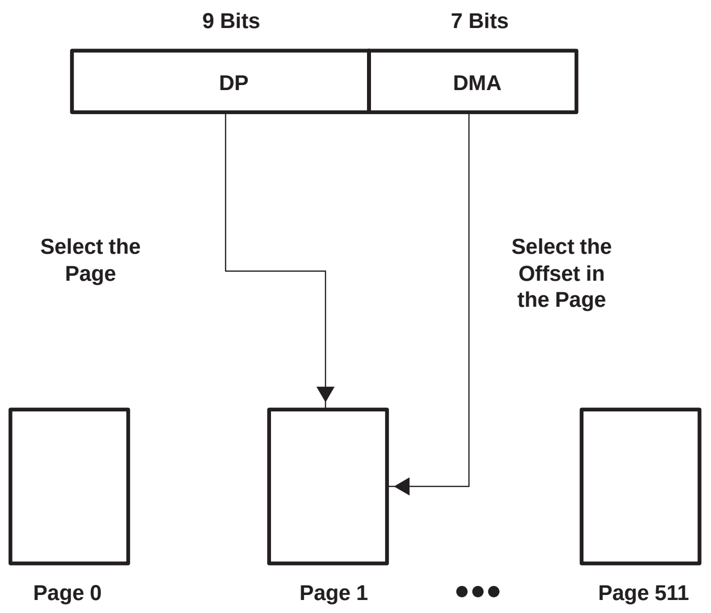

Figure 1. DP-Related Direct Addressing Mechanism

The generated address can be considered as two parts: the data page number and the offset value of the variable inside the data page. When the DP value changes, the data page number changes, but the offset value stays the same. Thus, by changing only the DP value, it is possible to address a different memory space.

1.2 Partitioning of Data

Because the DMA field of the generated address is 7 bits long, the range of memory that can be addressed in one data page is 128 words long. For most applications, the data does not fit into this 128-word page. Thus, it is necessary to limit the use of direct addressing for some data, and to use a different method for other data. This can be done by dividing data into two different types:

- Variables

  Variables are usually addressed individually. They are 1 word long (2 words long for 32-bit data) and are located in a data page. They are usually addressed using DP-related direct addressing.

- Arrays

  Arrays are usually addressed sequentially. They are usually several words long and are located in data memory space. They are addressed using indirect addressing.

In addition to the variable and array data, it is necessary to create pointers that will be used to keep the start address of the arrays.

- Pointers

  Pointers are always addressed individually. They are 1 word long and are located in a data page. They are always addressed using direct addressing. One pointer is associated with each array. The pointer contains the immediate address of the corresponding array in memory. Each pointer is initialized at the start of each channel with an address that depends on the channel number.

1.3 Multichannel Processing Implementation

Using the features described in the partitioning of data, it is now possible to implement multichannel processing for an application using the following rules:

- Separate variables and arrays, using the description detailed in Section 1.2, *Partitioning of Data*.
- Allocate data for the reference channel by performing the following:
  - Set the DP to a specific value, DPref.
  - Allocate variables in a data page. This data page is called DPref and is used as the reference data page for this application. All other data pages are generated from this data page.
  - Allocate arrays in memory. These arrays are the reference arrays and are located at address (start_address_arrays)ref.
  - Initialize pointers for the reference channel: (pointer_arrayi)ref = (address_arrayi)ref.

  This allocation is performed at link time and associates each data label to a physical address in memory. At run time, the reference channel does not necessarily need to be activated. This channel is used to generate the data for all the other channels.
- Allocate data for a new channel by performing the following:
  - Set the DP to a new value DPn.
  - Reserve space in the new data page for variables. Initialize variables if necessary.
  - Reserve space in memory for the arrays, located at an offset OFFn from the reference channel arrays: (start_address_arrays)n = (start_address_arrays)ref + OFFn.
  - Initialize pointers for this channel: (pointer_arrayi)n = (address_arrayi)ref + OFFn.

  This part can be repeated several times, depending on the number of channels to run.

Figure 2 illustrates the data pages and the memory organization used to implement the multichannel code processing method. It shows the reference channel data page and arrays, and how they are used to initialize the pointers of any other channel.

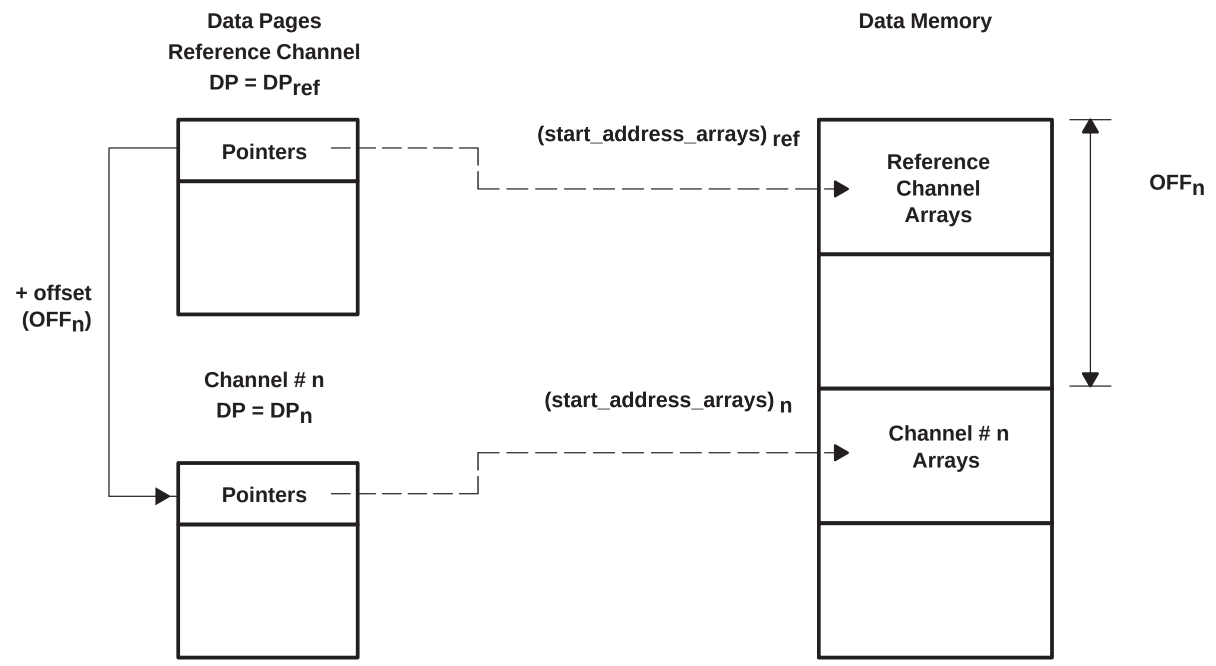

Figure 2. Multichannel Memory Organization and Initialization

1.4 Impact of Multichannel Processing on Code Performance

Implementation of multichannel code processing in an application has a very low impact on the performance of the total code because it uses DP-related addressing, a built-in feature of the DSP. Implementing the multichannel processing generates the following changes:

- A small increase in code size, mainly due to the initialization of pointers
- A very small increase in the number of cycles, due to loading pointers with an offset value
- A small increase in data memory size (limited to 128 words per channel) for the pointers

A DSP algorithm performance is usually measured by the number of cycles and by the code size. An analysis has been performed of the GSM enhanced full rate (EFR) speech codec implementation on the TMS320C54x™ DSP generation device. The code size, data memory size, and number of cycles were measured for the single-channel case and for the multichannel case. The results are shown in Table 1.

Table 1. Performance Comparison—Single-Channel and Multichannel Modes

	Single-Channel	Multichannel	Increase %
Number of cycles measured on 10 frames	2850019	2852103	0.07 %
Code size (in words)	9348	9858	5.5 %
Data memory (in words)	4191	4262	1.7 %

These results show that the overhead due to the implementation of the multichannel code processing is small. More important, the increase in the number of cycles is near zero.

2 Implementation of a Multichannel Application

The different pieces of assembly code that must be used to write a multichannel application following the methodology described in Section 1 describe some macros that are useful for a multichannel implementation and provide some tips to avoid common non multichannel problems.

A compilation flag, *multichannel*, can be defined to enable the switch between a single-channel and a multichannel implementation of the same application. For example, a GSM speech coding algorithm can be compiled in the single-channel mode if it is used in a mobile station or in the multichannel mode if it is used in a base station.

The following convention can also be used to define arrays and pointers:

- a_table is an array called *table*.
- p_table is the pointer in the data page that contains the a_table array address for the current channel.

2.1 Calling a Multichannel Application

Calling a multichannel application is easy. The only difference between a regular application and a multichannel application is that you must specify the channel number before calling an instance of the multichannel application. The channel number corresponds to the DP value for the channel.

Before calling the application itself, you must run a routine to initialize the pointers with the correct values. This initialization routine must be provided with the application and requires two input parameters—the DP number (that is, the channel number) and the offset value between the arrays of the reference channel and the arrays of the current channel (OFFn).

Example 1 illustrates the initialization process for a TI C54x™ DSP.

Example 1. Initialization Process

```
; first channel initialization

LD   #pg0, DP              ; DP = page number of reference channel
LD   #0, B                 ; BL = offset for reference channel = 0
CALL init_application      ; call the initialization routine

; second channel initialization

LD   #pg1, DP              ; DP = page number of current channel
LD   #off1, B              ; BL = offset for current channel
CALL init_application      ; call the initialization routine

; run application for first channel

LD   #pg0, DP              ; DP = page number of reference channel
CALL application

; run application for second channel

LD   #pg1, DP              ; DP = page number of current channel
CALL application
```

C54x is a trademark of Texas Instruments.

2.2 Writing the Initialization Routine

In addition to the initialization of the static variables related to the algorithm, this routine must include the initialization of the pointers used to access the arrays in memory that correspond to the current channel.

A macro can be defined to facilitate this initialization. Depending on the mode (single-channel or multichannel), the macro initializes the pointer value using the array address directly or by using the reference array value and the offset value that corresponds to the current channel.

Example 2 shows a macro defined for initialization on a TI C54xx™ DSP.

Example 2. Macro Defined for Initialization

```
INITPTR .macro  array
 .if     multi_channel
        LD      #a_:array:, A         ; AL = address of a_array for
                                      ; the reference channel
        ADD     B, A                  ; AL = AL + OFFn (stored in BL
                                      ; when calling init)
        STL     A, @p_:array:         ; p_array = AL
        .else
        ST      #a_:array:, @p_:array:    ; p_array = a_array address
        .endif
        .endm
Example of usage : INITPTR  table
```

2.3 Using Pointers in the Assembly Code

When an array address is needed in the application code, it is collected using the corresponding pointer located in the data page that corresponds to the current channel. This can be done with macros, which load the array address for the current channel in a register or an accumulator, and eventually add a fixed offset value to access a specific element of the array.

Example 3 shows the LDPTR macro loading the array address in an auxiliary register (AR0, .. AR7).

Example 3. LDPTR Macro Loading the Array Address in an Auxiliary Register

```
LDPTR   .macro  ARRAY, AR, OFFSET
  .nolist
        .if     $symlen(OFFSET) = 0
        .eval   0, OFFSET
        .endif
        .list
        .if     multi_channel
        MVDK    @p_:ARRAY:, :AR:         ; ARx = address of array for
                                            ; the current channel
```

C54xx is a trademark of Texas Instruments.

```
      if     OFFSET != 0
       MAR     *+:AR:(OFFSET)           ; ARx = ARx + OFFSET
      .endif
      .else
      STM     #a_:ARRAY:+:OFFSET:, :AR:  ; ARx = address of
                                           ; reference array + OFFSET
      .endif
      .endm
```

In multichannel mode, `LDPTR table, AR2, 10` is coded as:

```
MVDK    @p_table, AR2
MAR     *+AR2(10)
```

In single-channel mode, `LDPTR table, AR2, 10` is coded as:

```
STM     #a_table+10, AR2
```

The input parameters to the LDPTR macro are:

- The array name (for example, *table*)
- The auxiliary register name (for example, AR2)
- An optional offset value (for example, 10)

Example 4 shows the LDPTA macro loading the array address in an accumulator (A, B).

Example 4. LDPTA Macro Loading the Array Address in an Accumulator

```
LDPTA   .macro   array, Acc
        .if      multi_codec
        LD       @p_:array:, :Acc:    ; Acc = @p_array
        .else
        LD       #a_:array:, :Acc:    ; Acc = #a_array
        .endif
        .endm
```

The input parameters to the LDPTA macro are:

- The array name (for example, *table*)
- The accumulator name (for example, A)

These macros are very useful for writing multichannel code because they allow you the flexibility to compile the code in single-channel mode and improve the readability of the code.

2.4 Using Indirect Addressing on Data Page Variables

Variables located in the data page of each channel should be addressed via DP-related addressing most of the time. In some cycle-consuming loops, it may be more efficient to temporarily use indirect addressing on these variables, allowing Xmem/Ymem dual addressing. This means that the full 16-bit address of the variable (which depends on the DP value) must be stored in one of the auxiliary registers.

An efficient way to do indirect addressing is to define a pointer that keeps the start address of the temporary variables in the data page. This pointer is called p_tempv, and it is initialized in the initialization routine using a specific piece of code like that shown in Example 5.

Example 5. Defining a Pointer

```
.if     multi_channel
LDM     ST0, A
AND     #1FFh, A        ; A = DP
STL     A, @tempv       ; tempv = DP
LD      #tempv, A       ; A = address of tempv for the reference
                        ; channel
AND     #7Fh, A         ; A = offset of tempv in the Data Page
ADD     @tempv, 7, A    ; A = DP<<7 + offset
STL     A, @p_tempv     ; p_tempv = address of tempv for the current
                        ; channel
.else
ST      #tempv, @p_tempv  ; p_tempv = address of tempv for the
                          ; reference channel
.endif
The temporary variable, tempv, can be used to store the data that needs to be
addressed via indirect dual addressing :
MVDK    @p_tempv, AR2    ; AR2 = address of tempv for the current
                         ; channel
ST      #0x8000, *AR2    ; tempv = 0x8000
MAC     *AR2, *AR3, A    ; Use AR2 to do dual addressing with tempv
```

Declaring the p_tempv pointer on a temporary variable allows you to save cycles in some critical loops by using indirect addressing on a data page variable. Otherwise, it would be necessary for you to declare the temporary variable as an array of 1 word, with the associated pointer, thus making inefficient use of the data page memory. Furthermore, defining a pointer on a temporary variable located in the data page allows you to mix DMA addressing and indirect dual addressing on the same variable, without transferring the value.

For example:

- The piece of code shown in Example 5 could also be written as:

```
ST      #0x8000, @tempv    ; tempv = 0x8000
MVDK    @p_tempv, AR2      ; AR2 = address of tempv for the current
                           ; channel
MAC     *AR2, *AR3, A      ; Use AR2 to do dual addressing with tempv
```

- Without the p_tempv pointer, the same piece of code would be:

```
ST      #0x8000, @tempv    ; tempv = 0x8000
LDPTR   temp_array, AR2    ; AR2 = address of temp_array of 1 word
LD      @tempv, A          ; Transfer the tempv value to temp_array
STL     A, *AR2            ; temp_array[0] = tempv
MAC     *AR2, *AR3, A      ; Use AR2 to do dual addressing with
                           ; temp_array
```

2.5 Common Problems When Writing Multichannel Code

This section lists common problems that you may encounter when writing multichannel assembly code. Although not exhaustive, this list gives an overview of the most common situations.

- When writing the code using the proposed syntax, several obvious non-multicodec associations must be avoided:

  - #a_array: address of a_array for the reference channel only. This can be used only in the initialization routine (INITPTR).
  - #var: address of a variable for the reference channel only. This can be used only in the initialization routine (for example, init of p_tempv).
  - @ARx: This will address the ARx register only if DP = 0.
- Never use Dmad addressing (16-bit immediate addressing) with variables or pointers of the data page. Be careful when using MVDK, MVKD, MVDM, MVMD instructions.
- If you must modify the DP value, save it first, and then restore it afterward. Do not use DP-related addressing in between these tasks.
- Do not use the scratch pad (DP = 0) for channel-dependent variables.
- The maximum number of variables per data page is 128 words.
  - To avoid overflow, declare a memory section of this size at the link and store all the variable and pointer sections in it.
  - The linker provides an error message if the number of variables is too large.
  - If the number of variables is too large, try to overlap the variables that allow it in order to save space. The pointers can be overlapped in the same way as the corresponding arrays.

3 Converting a Single-Channel Application to a Multichannel Application

When you convert a single-channel application to a multichannel one, the assembly code is not the only change. The memory mapping must also be changed to differentiate the variables from the arrays. This is usually not an easy task because it was not planned from the beginning when the original code was written. This section describes the different steps to follow for the conversion, and gives some examples of changes to implement.

3.1 Analyzing the Data in Memory

To analyze the data in memory, do the following:

- For each data, check which addressing (immediate, direct, indirect addressing, ...) is used in the program.
- Determine if the data should be moved to a Data Page structure (variables) or stay in the RAM space (arrays). This determination depends on the length of the data and of the addressing used with it.
- Determine the constraints and changes on the memory map (alignments, groups, unions, for example).

3.2 Implementing the Changes

After preparing the changes, you must follow the methodology described here to ensure that nothing is overlooked.

3.2.1 Values in Data Page Variables

To change values in data page variables, do the following:

- Change the section name.
- Check that the data is always addressed with DP-related addressing. Make changes if necessary.
- Leave a hole in the original memory to keep the alignment constraints. Optimization can be performed later with care (for example, using UNION statements in the linker command file for variables that are not needed at the same time).

3.2.2 Values in Memory Arrays

To change values in data memory arrays, do the following:

- Change the array name (add a_).
- Add the pointer declaration.
- Replace the instructions, as described in Table 2.
- Add the pointer initialization into the initialization routine.

Table 2 shows the conversion between single-channel and multichannel processing.

Table 2. Conversion Between Single-Channel and Multichannel Versions

Replace:	**With:**
STM #aname, ARx	LDPTR aname, ARx
STM #aname+k, ARx	LDPTR aname, ARx, k
MVDM #data, ARx	MVDK @data, ARx
MVMD ARx, #data	MVDK ARx, @data
RPT #n MVDK *ARx+, #data	PSHM ARy ; if needed LDPTR data, ARy RPT #n MVDD *ARx+, *ARy+ POPM ARy ; if needed
ST #k, *(data)	ST #k, @data
LD *(data), Acc	LD @data, Acc
ADD #array, Acc	.if multi_channel ADD @p_array, Acc .else ADD #a_array, Acc .endif
;In page 0 : ST #K, T	STM #K, T

4 Conclusion

The methodology described in this document is being used in multichannel algorithm implementations in a variety of products on the market. Multichannel assembly code processing is efficient and reliable in wireless applications where the optimization level is very high due to power consumption and cost requirements. This methodology can also be used in any domain that must run several instances of the same algorithm. With a small overhead, multichannel assembly code processing adds value to fully-optimized assembly algorithms and makes optimum use of the C5000™ DSP family of devices.