Notes on using Analog Devices' DSP, audio, & video components from the Computer Products Division
Phone: (800) ANALOG-D or (781) 461-3881, FAX: (781) 461-3010, EMAIL: dsp.support@analog.com

C Programs on the ADSP-2106x

Last Modified: 6/25/96

This application note gives a brief introduction in programming Analog Devices SHARC DSP in C language. Attached to this note are two C examples files irq1.c and linkdma.c. A brief description of their purpose is given below. All programs have been written with the development software ADDS-210XX-SW-PC release 3.3 and have been tested on the ADDS-2106x-EZ-LAB. Note that there is a new C compiler on the FTP site (ftp.analog.com).
Attached files:

21062c.ach
21060c.ach
def06xc.h
linkdma.c
irq1.c

1. Read The Release Note

There are some restrictions regarding the software tools. Please read the release notes to be firm with the restrictions of your development software.

2. Writing An Architecture File

The architecture file is used to describe your target system on which the C program should run. There are two .ACH files attached to this application note. 21062C.ACH could be used to run C code on the EZ-LAB and the 21060c.ACH is for people who upgraded their EZ-LAB with an ADSP-21060. The /CINIT, /CSTACK and /CHEAP directives are only used for C programming and are described in the C TOOLS MANUAL on page 3-3 ff. The attached .ACH files are only examples to allow a quick start and if you wish to divide up your memory in another way refer to the USER'S MANUAL in chapter 5.

3. Writing And Compiling C Code

After you wrote your c code the compiler should be invoked in the following way:

g21k code.c -a 21062C.ach -o output -g

The -g switch is used to generate debuggable code for the simulator and emulator. If you want the compiler to produce an optimized code use one of the following switches.

-O, -O2, -O3

But remember that it is not possible to use the CBUG feature for code that has been optimized.
Furthermore the iterators (ITER) of the numerical C extension does not work with the -O3 switch.
Using any optimization switches often cause the following problem. If a variable is checked in a loop but is not updated in that loop the compiler removes the test of this variable out of the loop to minimize the number of instructions. This could cause problems when the value of this variable is updated within an interrupt service routine and should be evaluated in the main loop. To avoid this behavior it is necessary to declare this variable as volatile. e.g.

volatile int test_flag;

Sometimes customers wish to place C-Functions not in the default segment *seg_pmco*, but in a different one. This could be easily done with the G21K.EXE compiler. e.g. There is a main program (sum.c) and an external function (ext.c) both written in C language. The main program calls the external function in the alternative code segment *alt_pmco*.

```
/* PRG that calls an external c function in an alternate code segment */

extern int sum(int a,int b);

void main(void)
{
/* ---  VAR --- */
int i=5;
int j=3;
/* --- MAIN --- */
i= sum(i,j);
idle();
}
/* PRG END  */
```

The external functions could look like the following code:

```
int sum(int a, int b)
{
return a+b;
}
```

To compile this example use the command lines

g21k sum.c -c -a 21062c.ach
g21k ext.c -c -mpmcode=alt_pmco
g21k sum.o ext.o -a 21062c.ach -o sum -map -g

The *-mpmcode* switch chooses the alternative code segment. Variables placed in the dm, pm memory space could be placed in an alternative segment by invoking the compiler with a similar switch.

-mdmdata, -pmdata

4. Extensions to ANSI C

The Analog Devices C compiler is based on the GNU software and supports ANSI C. Nevertheless there are some extensions which are very useful. As it is possible to store data either in the program or data memory of the Analog Devices DSP an additional keyword indicates where variables are placed by the linker. Use the PM directive to place variables in program memory and the DM directive to place them in data memory. e.g.

float dm source[*16*]
float pm dest[*16*]

When no directive is given the linker will place the variable in the data memory by default. The default memory segment in which the dm variables are placed is labeled seg_dmda, and the default segment for variables which reside in program memory space is called seg_pmda. Often it is useful to place the variables in different segments for example external memory space. An easy way to do this is to use the delivered include file "macros.h". To use this comfortable feature add the directive

#include<macros.h>

to your source code and use the following syntax:

DEF_VAR (name, type, seg, mem)

name: variable name
type: the actual C type
seg: the segment in the architecture file.
mem: the memory that the variable exists in pm,dm

Another useful macro is called "DEF_PORT" which has nearly the same syntax as the above mentioned one. This macro could be used to define an IO port in C language. The syntax is:

DEF_PORT(name, type, seg, mem)

name: variable name
type: the actual C type
seg: the segment in the architecture file
mem: the memory that the variable exists in pm,dm

The necessary segment declaration in the architecture file could look like the following line.

.SEGMENT /PORT /BEGIN=0x00404000 /END=0x00404000 /DM /width=32 mafeadrs;

It is worthy to have a look at the macros.h file as there are a lot of useful declarations, that are not documented yet. Further extensions are described in the C TOOLS MANUAL chapter 5.

5. Accessing IOP Registers

Most of the SHARC's control registers are memory mapped to the address range from 0x0000 to 0x0100. To access these memory mapped IOP registers you could use the following syntax in your C code.

*#define SYSCON *(int*)0x000*

Using this pointer assignment it is now possible to change the content of the System Control register in the C file with the following instruction:

SYSCON = 0x1234;

Attached to this note is a header file def06xc.h, that already defines some IOP registers. This file has to be included at the beginning of your program to grant this comfortable way of accessing IOP control registers. The syntax is:

#include<def06xc.h>

6. Embedded Assembly Language

6.1 Accessing Core Processor System Registers

The core processor system registers (MODE1, MODE2, ASTAT, STKY, IRPTL, IMASK, IMASKP, USTAT1 &

USTAT2) are not memory mapped and due to this they could not be accessed via the above mentioned syntax. These registers could only be modified by using the inline assembler feature.

```
asm("   1st instruction;
        2nd instruction;
        :
        last instruction;");
```

To be able to manipulate the bits of these core processor system registers in an easy way include the def21060.h header file at the beginning of your inline assembler code. e.g.

```
asm("   #include<def21060.h>;
        bit set imask LP2I;
        bit set mode1 IRPTEN;");
```

Any other assembler instruction could be placed within the "asm();" directive.

6.2 Inline Assembler Instructions

The above mentioned syntax could be used to place any valid assembler instruction within in c source code. This is sometimes very useful if you wish to have a minimum of overhead for example for time critical program parts.

6.3 Calling Assembler Subroutines

It is also possible to call assembler routines out of the c main program. A detailed description could be found in the C TOOLS MANUAL chapter 4.
If you want to call external assembler subroutines or functions from the main C code the easiest way to link these different programming languages is to invoke the g21k compiler in the following way:

g21k file1.c file2.asm -a 21062c.ach -o out.exe

In this way you do not have to care about how to invoke the linker ld21K.

```
/* File main.c calls external asm subroutine mul2 to multiply two
numbers */

extern float mul2(float x, float y);
main()
{
        float a = 0.25;
        float b = 0.75;
        float c;
        c = mul2(a,b);  /* call assembler subroutine */

        idle();
        exit(0);
```

```
/* File extmul.asm */
#include<asm_sprt.h>
.segment/pm     seg_pmco;
.global         _mul2;
_mul2:
leaf_entry;
f0=f4*f8;  /* fetch first and second parameter */
leaf_exit;
.endseg;
```

7. Example 1 (Interrupts in C)

The irq1.c is just a small demo program that illustrates how to program interrupts in C language. The demo could be run on the ADDS-2106x-EZ-LAB and by pressing the interrupt button on the board the LED connected to flag0 of the ADSP-2106x will toggle. The code is well documented and for further information refer to the C TOOLS MANUAL.

```
/* program that demonstrates the function of the irq1 button on the
   ADDS-2106x-EZ-LAB
   The flag0 is toggled when you  push the irq button
   Date:                  */

#include"def06xc.h"  /* defines IOP registers to access them in c
language */
#include<signal.h>    /* include file for the interrupt stuff */

volatile int irq1_occured;

void irq1_handler(int signal) /* irq1 routine that toggles flag on the lab
*/
{
irq1_occured=1;                 /* set irq1 flag  */
asm("bit tgl astat 0x080000;");   /* toggle flag0 using inline assembler */
}
void main(void)
{
/* --- INLINE ASSEMBLER --- */

asm("#include<def21060.h>;

/* configure flag direction */

    bit set mode2 FLG0O;        /* FLAG0 output */
    bit clr mode2 FLG1O;        /* FLAG1 input  */
    bit set mode2 FLG2O;        /* FLAG2 output */
    bit clr mode2 FLG3O;        /* FLAG3 input  */
    nop;");
/* --- END INLINE --- */

irq1_occured = 0;
interrupt(SIG_IRQ1, irq1_handler); /* irq1 handler routine */

while(1)
{
  idle();
  if(irq1_occured)
```

```
  irq1_occured = 0;
}
}
```

8. Example 2 (Linkport DMA in C)

The following code is referred to the assembler code in the SHARC user's manual page 9-29 and uses the link port loop back feature to transmit data via DMA channel 5 from a source buffer to a destination buffer. The loop back mode is enabled by assigning two different link buffers to the same link port. In this case both link buffer 2 and 3 are assigned to link port 3. The double clock rate of the Analog Devices DSP is used to transmit this data. For further details about link ports see in the SHARC user's manual.

```
/* PRG that generates some sample values stores them in buffer
source[]
and transmit them via LINKDMA to buffer dest[]
Date:                                                    */
#include<stddef.h>
#include<ctype.h>
#include<stdlib.h>
#include"def06xc.h"              /* header file of register definitions */

#define N 16

void main(void)
{
/* ---  VAR --- */

static dm float source[N];       /* source buffer for transfer */
static dm float dest[N];        /* destination buffer */

int i;

/* --- MAIN --- */

for(i=0;i<N;i++)
{
  source[i] = (float) i;                    /* create some values for the
transfer */
}

II5 = (int) &source;         /* set DMA pointer to start of source buffer
*/
II4 = (int) &dest;           /* set DMA pointer to start of dest. buffer */

IM5 = 1;                     /* set DMA modify source to 1 */
IM4 = 1;                  /* set DMA modify dest. to 1 */
C5 = N;                      /* set DMA count to lenght of buffer */
C4 = N;                      /* set dma count to length of buffer */

LCOM = 0x0c0;                 /* double clock rate for transmission */

LAR = 0x03f03f;               /* LAR register: set LBUF2->port0, */
                  /*              LBUF3->port0  */
LCTL = 0x0b300;               /* 32 Bit data, LBUF2=rx, LBUF3=tx */
                  /* enable DMA */
/* --- INLINE ASSEMBLER --- */

asm("#include<def21060.h>;
    bit set imask LP2I;    /* enable link buffer 2 interrupt */
    bit set mode1 IRPTEN;");      /* global interrupt enable */
/* --- INLINE END --- */
idle();                       /* set ADSP-2106x in idle state */
}
/* PRG END  */
```