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

Language Extensions: Memory Storage Types, ASM & Inline Constructs

Last Modified: 05/11/96

This EE Note has been conceived as guidelines for users programming our 21xx-DSPs in C. It consists of several pieces of examples as opposed to a whole project. These examples have been conceived general-purposed to allow the users to run them without any piece of hardware, just using the Simulator. The aim of these guidelines is namely to illustrate some fundamentals with appropriate examples, making thus the theory better understandable.

Language extensions

The G21 C compiler supports a set of extensions to the ANSI standard for the C programming language. These extensions are specific to the ADSP processors and comprise:

- Support for separate program and data memory (keywords pm, dm)
- Support for inline functions
- Support for asm() inlining of assembly language

The following demo code will help illustrate the use of these extensions and provide further information on them.

LANG_EXT.C

```
/* lang_ext.c */
/* program demonstrating the use of the G21 extensions */
/* to the C language */


1     #include "pointdef.h"              /* contains predefined pointers */

      int aux;

      static int dm x ;                  /* placing variables in specific */
      static int dm y ;                  /* memory spaces */
5     static int pm z ;
      static int dm i[3]={10,20,30};

      DINT pointer;                      /* DINT is defined in pointdef.h */
                                         /* for a pointer to a dm integer */

      static inline int add(void)        /* define add() as inline */
```

```
      {
10    z=x+y;
      return z;
      }

      static inline void Set_Fl(short flag)     /* define Set_Fl() as inline */
                                                /* this function just sets */
                                                /* Flag 0, 1 or 2 */
      {
15    switch (flag)
            {
            case 0:     asm volatile("set fl0;");
                        break;
            case 1: asm volatile("set fl1;");
20                      break;
            case 2: asm volatile("set fl2;");
                        break;
            }
      }

25    static inline void reset_fl1(void)        /* to reset Flag 1 and count */
                                                /* the amount of resets */
      {
      static int pm rst_fl1_count;
      rst_fl1_count++;
      asm volatile("reset fl1;");
30    }

      void main (void)
      {                             /* several pointer operations */
                                    /* based on the use of the keywords */
                                    /* pm/dm and the file pointdef.h */

      pointer=i;                    /* pointer points to i[0]=10 */
      x = *pointer++;               /* x=10 ; pointer points to i[1]=20 */
35    aux=(*pointer)++;             /* aux=20; i[1]=21; */
      y=i[1];                       /* y=21 */
      add();

      Set_Fl(0);                    /* calling the inline functions */
      asm volatile("reset fl0;");

40    Set_Fl(1);
      reset_fl1();

      asm volatile("toggle fl1;");
      reset_fl1();
      }
```

Dual Memory Support (pm dm)

The two keywords (pm, dm) support the dual memory space (separate program and data memory spaces where program memory can store both data and opcodes and data memory stores data), modified Harvard architecture of the ADSP-2100 Family processors, as opposed to the Von Neumann architecture where program instructions and data are all stored in one common memory space.

The keywords (pm, dm) are used to specify the location of a static (or global) variable. Being thus able to explicitly access PM data locations (e.g. for fetching filter coefficients) and DM data locations, you can take advantage of the DSP's architecture where two data words and an instruction can be fetched in a single cycle.

Lines 3-6

These statements illustrate the placement of variables in PM or DM spaces.

Some rules applying to the use of the dual memory support keywords are:

- Without specification of pm/dm, G21 uses data memory as the default memory space. For example the variable *aux* **(line 2)** is placed into data memory.

- Program memory is the dedicated memory space for functions

- Automatic variables (= local variables within a function) are placed onto the stack, which resides in data memory. To be able to use the pm keyword inside a function, you have to give the variable the storage class static by using the qualifier of the same name. **(line 27)**

Pointer declarations using pm/dm

A further usage of the dual memory support keywords is qualifying pointer declarations within the dual memory space. For instance, the following statement

```
int dm * pm pd;
```

declares pd as a pointer residing in program memory and pointing to a data memory integer.

Our demo program *lang_ext.c* uses this pointer declaration feature, too, by including **(line 1)** the user header file pointdef.h, which contains some predefined pointer types. This file can be seen as a shell that can be customized to be used in any code dealing with pointers.

POINTDEF.H

```
typedef int pm *  PINT;                    /* PINT defines a type which is a pointer
                                           to an integer which resides in pm */

typedef int *  DINT;                       /* DINT defines a type which is a pointer
                                           to an integer which resides in dm */
                                           /* other syntax: typedef int dm *  DINT;
*/

typedef float pm *  PFLOAT;                /* PFLOAT defines a type which is a
pointer                                          to a float which resides in pm */

typedef float *  DFLOAT;                   /* DFLOAT defines a type which is a
pointer                                          to a float which resides in dm */
```

```
                                        /* also: typedef float dm *  DFLOAT; */
```

Line 7 of the C program

```
DINT pointer;
```

uses for example the predefined pointer type DINT of the file *pointdef.h* to declare the variable, *pointer* as a pointer residing in dm (per default) and pointing to an integer which resides in dm.

One could wonder: why not also defining the memory space (pm/dm) of the pointer within the typedef statement?

It is not possible to do so. The compiler would for instance flag an error message on this construct:

```
typedef int * pm DINTP;     /* trying to define a type which is a pm pointer
                               pointing to a dm integer */
```

The pointer storage type must be defined by the programmer, i.e. :

```
#include "pointdef.h"
DINTD pm dd1;                   /* = int * pm dd1 */
                                /* dd1 is a pm pointer pointing to a dm integer */
```

Lines 33-36 of the C program shows some pointer operations that illustrate the handling with variables and pointer explicitly declared to be in pm/dm.

To finish with this section on pointer declarations, it could be added that, since functions always reside in program memory, pointers to functions always point to program memory.

Inline Function Support Keyword (inline)

The G21 inline keyword directs G21 to integrate the code for the function declared as inline into the code of its callers. This makes execution faster by eliminating the function-call overhead.

Lines 8, 13 and 25 of our C example show examples of function definitions that use the inline keyword.

Note that the definitions of the functions precede the calls to the functions. This is to prevent the calls from not being integrated into the caller.

How inlining functions affect code and execution can be seen under the simulator in the C code window (CBUG) and the assembly code window (Program Memory Window).

An alternative to explicitly declaring functions as inline is to use the switch -finline-functions when compiling your C code. This switch forces function inlining by directing the compiler to integrate all simple functions into their callers. Which functions are simple is decided by the compiler.

Inline Assembly language Support Keyword (asm)

The asm() construct allows the coding of assembly language instructions within a C function and is thus useful for expressing assembly language statements that cannot be expressed easily or efficiently with C constructs. Some examples are:

- `asm(" ifc=0xff ")` /* accessing non-memory-mapped registers can only be done in assembly language */
  / Please refer to EZ NOTES Number 01 */*

- `asm volatile ("set fl0;");` /* **line 17** of lang_ext.c */
  /* `set fl0` is a processor specific instruction
  that can only be expressed in assembly language */

The extension **volatile** is used to prevent the asm() instruction from being deleted or moved significantly.

- A particular register variable can be assigned using asm(). The following statement assigns the register *AX0* to the C variable *x*:

```
register int x asm("AX0");
```

The following batch file has been used to compile the demo program:

LANG_EXT.BAT

```
g21 lang_ext.c -a lang_ext -g -v -Wall -save-temps -mlistm -fkeep-inline-functions -
o lang_ext -map
```

- The -map switch directs the linker to generate a memory map file of all symbols. This is useful for debugging purposes.

- The -fkeep-inline-functions (force keeping of inlined functions) switch directs the compiler to output a separate runtime callable version of the inlined functions. This switch is necessary, for instance if you want to set breakpoints within the functions, because it makes G21 output own assembler code for the functions (what the C compiler does not do otherwise).

After compiling the example, you can run it using the C Debugger of the simulator (CBUG). For information on how to use CBUG, please refer to the C Tools Manual and the Release