Topics

21 Symbolic Debugging Directives

The MSP430 fixed–point assembler supports several directives that a high level programming language can use for symbolic debugging:

- The **.sym** directive defines a global variable, a local variable, or a function. Several parameters allow you to associate various debugging information with the symbol or function.
- The **.stag**, **.etag**, and **.utag** directives define structures, enumerations, and unions, respectively. The **.member** directive specifies a member of a structure, enumeration, or union. The **.eos** directive ends a structure, enumeration, or union definition.
- The **.func** and **.endfunc** directives specify the beginning and ending lines of a function.
- The **.block** and **.endblock** directives specify the bounds of blocks.
- The **.file** directive defines a symbol in the symbol table that identifies the current source file name.
- The **.line** directive identifies the line number of a source statement.

These symbolic debugging directives are not usually listed in the assembly language file that the compiler creates. If you want them to be listed, invoke the compiler shell with the -g option, as shown below:

```
cl430 -g input file
```

This appendix contains an alphabetical directory of the symbolic debugging directives. Each directive contains an example of C source and the resulting assembly language code.

Syntax **.block** [*beginning line number*]

.endblock [*ending line number*]

Description The .block and .endblock directives specify the beginning and end of a block. The line numbers are optional; they specify the location in the source file where the block is defined. Line numbers are relative to the beginning of the current function.

Note that block definitions can be nested. The assembler will detect improper block nesting.

Example Here is an example of C source that defines a block and of the resulting assembly language code.

C source:

```
   .
   .
   .
{                    /*  Beginning of a block  */
     int  a,b;
     a = b;
}                    /*  End of a block        */
   .
   .
   .
```

Resulting assembly language code:

```
.block   4
.sym     _a,1,4,1,8
.sym     _b,2,4,1,8
.line 5
MOV   2(SP), 1(SP)
.endblock    6
```

Syntax **.file** *filename*

Description The .file directive allows a debugger to map locations in memory back to lines in a source file. The *filename* is the name of the file that contains the original C source program. The first 14 characters of the filename are significant; any pathname information is stripped away.

You can use the .file directive in assembly code to provide a name in the file and improve program readability.

Example Here's an example of the .file directive. The file named *text.c* contained the C source that produced this directive.

```
.file    "text.c"
```

Syntax **.func** [*beginning line number*]

.endfunc [*ending line number*] [*, register save mask1*] [*, register save mask2*] [*, frame size*]

Description The .func and .endfunc directives specify the beginning and end of a function. The line numbers are optional; they specify the location in the source file where the function is defined. The register save masks indicate which registers were saved by this function. If bit 0 of mask2 is 1, R0 was saved by the function; if bit 1 of mask2 is 1, R1 was saved; if bit 0 of mask1 is 1, R16 was saved; etc. The frame size parameter indicates how many bytes were reserved for the local frame of this function.

Note that function definitions cannot be nested.

Example Here is an example of C source that defines a function and of the resulting assembly language code.

C source:

```
power(x, n)            /*  Beginning of a function
   */
int x,n;
{
      register int i, p;
      p = 1;
      for (i = 1; i <= n; ++i) p*= x;
      return p;         /*  End of function          */
}
```

Resulting assembly language code:

```
       .sym   _power,_power,36,2,0
       .global   _power
       .text
       .func  1
*****************************************************
;* FUNCTION DEF : _power
;*****************************************************
;
_power:;
       INCW   #4,STK
       POP    A
       MOV    A,-2(STK)
       POP    A
       MOV    A,-3(STK)
       MOV    FP,-1,A
       MOV    A,-1(STK)
       MOV    FP,A
       MOV    A,@STK
       MOVW   STK,FP
       INCW   #2,STK
       MOV    R23,A
       MOV    A,-1(STK)
       MOV    R24,A
       MOV    A,@STK
       .sym   _x,-4,4,9,8
       .sym   _n,-5,4,9,8
       .sym   _i,23,4,4,8
       .sym   _p,24,4,4,8
       .line  3
       .line  5
       MOV    #01h,R24
       .line  6
       MOV    #01h,R23
       JMP    L2
L1:
       MOV    -4(FP),A
       MPY    R24,A
       MOV    B,R24
       INC    R23
L2:
       MOV    -5(FP),A
       CMP    R23,A
       JGE    L1
       .line  7
       MOV    R24,R8
EPI0_1:
       .line  8
       MOV    @STK,A
       MOV    A,R24
       MOV    -1(STK),A
       MOV    A,R23
       MOVW   FP,STK
       MOV    @STK,A
       MOV    A,FP
       MOV    -1(STK),A
       MOV    A,FP-1
       MOV    -2(STK),A
       MOV    A,B
       MOV    -3(STK),A
       INCW   #-4,STK
       BR     @R1
       .endfunc  8,00180H,00000H,0
       .end
```

Syntax **.line** *line number [, address]*

Description The .line directive creates a line number entry in the object file. Line number entries are used in symbolic debugging to associate addresses in the object code with the lines in the source code that generated them.

The .line directive has two operands:

- *Line number* indicates the line of the source that generated a portion of code. Line numbers are relative to the beginning of the current function. This is a required parameter.
- *Address* is an expression that is the address associated with the line number. This is an optional parameter; if you don't specify an address, the assembler will use the current SPC value.

Example The .line directive is followed by the assembly language source statements that are generated by the indicated line of C source. For example, assume that the lines of C source below are lines 5 and 6 in the original C source; lines 5 and 6 produce the assembly language source statements that are shown below.

C source:

```
for (p = 1; i = 1; i <= n; ++i) p*=x
return p;
```

Resulting assembly language code:

```
   .line 5
   MOV #01h, R24
   MOV #01h, R23
   JMP L2
L1:
   MOV -4(FP), A
   MPY R24, A
   MOV B, R24
   INC R23
L2:
   MOV -5(FP), A
   CMP R23, A
   JGE L1
   .line 6
   MOV R24, R8
```

Syntax **.member** *name, value [, type, storage class, size, tag, dims]*

Description The .member directive defines a member of a structure, union, or enumeration. It is valid only when it appears in a structure, union, or enumeration definition.

- *Name* is the name of the member that is put in the symbol table. The first 32 characters of the name are significant.
- *Value* is the value associated with the member. Any legal expression (absolute or relocatable) is acceptable.
- *Type* is the type of the member. Appendix A contains more information about types.
- *Storage class* is the storage class of the member. Appendix A contains more information about storage classes.
- *Size* is the number of bits of memory required to contain this member.
- *Tag* is the name of the type (if any) or structure of which this member is a type. This name **must** have been previously declared by a .stag, .etag, or .utag directive.
- *Dims* may be one to four expressions separated by commas. This allows up to four dimensions to be specified for the member.

The order of parameters is significant. *Name* and *value* are required parameters. All other parameters may be omitted or empty (adjacent commas indicate an empty entry). This allows you to skip a parameter and specify a parameter that occurs later in the list. Operands that are omitted or empty assume a null value.

Example Here is an example of a C structure definition and the corresponding assembly language statements:

C source:

```
struct doc {
     char title;
     char group;
     int  job_number;
} doc_info;
```

Resulting assembly language code:

```
     .stag   doc,24
     .member  _title,0,2,8,8
     .member  _group,8,2,8,8
     .member _job_number,16,4,8,8
     .eos
```

Syntax

.**stag** *name [, size]*
member definitions
.eos

.etag *name [, size]*
member definitions
.eos

.utag *name [, size]*
member definitions
.eos

Description

The .stag directive begins a structure definition. The .etag directive begins an enumeration definition. The .utag directive begins a union definition. The .eos directive ends a structure, enumeration, or union definition.

Name is the name of the structure, enumeration, or union. The first 32 characters of the name are significant. This is a required parameter.

Size is the number of bits the structure, enumeration, or union occupies in memory. This is an optional parameter; if omitted, the size is unspecified.

The .stag, .etag, or.utag directive should be followed by a number of .member directives, which define members in the structure. The .member directive is the only directive that can appear inside a structure, enumeration, or union definition.

The assembler does not allow nested structures, enumerations, or unions. A C compiler “unwinds" nested structures by defining them separately and then referencing them from the structure they are referenced in.

Example 1

Here is an example of a structure definition.

C source:

```
struct doc
{
    char  title;
    char  group;
    int   job_number;
} doc_info;
```

Resulting assembly language code:

```
        .stag   _doc,24
        .member _title,0,2,8,8
        .member _group,8,2,8,8
        .member _job_number,16,4,8,8
        .eos
```

Example 2 Here is an example of a union definition.

C source:

```
union u_tag {
    int   val1;
    float val2;
    char  valc;
} valu;
```

Resulting assembly language code:

```
        .utag   _u_tag,24
        .member _val1,0,4,11,8
        .member _val2,0,6,11,24
        .member _valc,0,2,11,8
        .eos
```

Example 3 Here is an example of an enumeration definition.

C Source:

```
{
    enum o_ty { reg_1, reg_2, result } optypes;
}
```

Resulting assembly language code:

```
        .etag   _o_ty,8
        .member _reg_1,0,4,16,8
        .member _reg_2,1,4,16,8
        .member _result,2,4,16,8
        .eos
```

Syntax **.sym** *name, value [, type, storage class, size, tag, dims]*

Description The .sym directive specifies symbolic debug information about a global variable, local variable, or a function.

- *Name* is the name of the variable that is put in the object symbol table. The first 32 characters of the name are significant.
- *Value* is the value associated with the variable. Any legal expression (absolute or relocatable) is acceptable.
- *Type?* is the type of the variable. Appendix A contains more information about types.
- *Storage class?* is the storage class of the variable. Appendix A contains more information about storage classes.
- *Size?* is the number of bits of memory required to contain this variable.
- *Tag* is the name of the type (if any) or structure of which this variable is a type. This name **must** have been previously declared by a .stag, .etag, or .utag directive.
- *Dims* may be up to four expressions separated by commas. This allows up to four dimensions to be specified for the variable.

The order of parameters is significant. *Name* and *value* are required parameters. All other parameters may be omitted or empty (adjacent commas indicate an empty entry). This allows you to skip a parameter and specify a parameter that occurs later in the list. Operands that are omitted or empty assume a null value.

Example These lines of C source produce the .sym directives shown below:

C source:

```
struct s { int member1, member2; } str;
int ext;
int array[5][10];
long *ptr;
int strcmp();

main(arg1,arg2)
   int arg1;
   char *arg2;
{
   register r1;
}
```

Resulting assembly language code:

```
.sym     _str,_str,8,2,16,_s
.sym     _ext,_ext,4,2,8
.sym     _array,_array,244,2,400,,5,10
.sym     _ptr,_ptr,21,2,16
.sym     _main,_main,36,2,0
.sym     _arg1,-4,4,9,8
.sym     _arg2,-6,18,9,16
.sym     _r1,23,4,4,8
```