Application Note

ZDS II for eZ80Acclaim!: Calling C Functions from Assembly and Vice Versa

AN033301-0711

Abstract

In certain situations, developers find that they must combine source code files written in Assembly code into C routines; the reverse is often also true. This application note aims to provide a straightforward method for combining Assembly routines and C language routines into one project in ZDS II, and discusses how ZDS II allocates arguments into the stack for parameter passing and where return values are stored. A discussion about naming conventions for functions used by ZDS II is also covered.

Discussion

The eZ80Acclaim! C-Compiler is a conforming freestanding 1989 ANSI C implementation with some exceptions.[1] In accordance with the definition of a freestanding implementation, the compiler accepts programs that confine the use of the features of the ANSI standard library to the contents of the standard headers `<float.h>`, `<limits.h>`, `<stdarg.h>` and `<stddef.h>`. The eZ80Acclaim! compiler release supports more of the standard library than is required of a freestanding implementation, as listed in the *Run-Time L*ibrary section of the Zilog Developer Studio II – eZ80Acclaim! User Manual (UM0144).

The eZ80Acclaim! C-Compiler supports language extensions for the easy programming of the eZ80Acclaim! processor architecture. The language extensions are described in the *Language Extensions* section of the Zilog Developer Studio II – eZ80Acclaim! User Manual (UM0144).

The following sections describe the features of the eZ80Acclaim! C-Compiler:

The eZ80Acclaim! C-Compiler is optimized for embedded applications in which execution speed and code size are crucial.

Calling Conventions

The C-Compiler imposes a strict set of rules on function calls. Except for special run-time support functions, any function that calls or is called by a C function must follow these

1. These exceptions are described in the *ANSI Standard Compliance* chapter of the Zilog Developer Studio II – eZ80Acclaim! User Manual (UM0144).

rules. Failure to adhere to these rules can disrupt the C environment and cause a C program to fail.

The following sections describe the calling conventions:

- Function Call Mechanism
- Special Cases

Function Call Mechanism

A (caller) function performs the following tasks when it calls another (called) function:

1. Save all registers (other than the return value register, to be defined below) that might be needed in the caller function after the return from the call, that is, a "caller save" mechanism is used. The registers are saved by pushing them on the stack before the call.
2. Push all function parameters on the stack in reverse order (the rightmost declared argument is pushed first, and the leftmost is pushed last). This places the leftmost argument on top of the stack when the function is called. For a varargs function, all parameters are pushed on the stack in reverse order.
3. Call the function. The call instruction pushes the return address on the top of the stack.
4. On return from the called function, caller pops the arguments off the stack or increments the stack pointer.
5. The caller then restores the saved registers by popping them from the stack.
6. The following example illustrates what must be done in an assembly procedure that calls a C function, including "caller save" of the BC register:

```
LD BC,123456h ; BC is used across function call
PUSH BC       ; Must be pushed before arguments are pushed
PUSH HL       ; Push argument
CALL _foo     ; Might modify
BC POP BC     ; Remove argument
POP BC        ; Must be popped after arguments are deallocated
ADD HL,BC     ; Wrong value of BC could be used if BC not
              ; saved by caller
```

In the eZ80® MPU, a multiple of 3 bytes is always used when pushing arguments onto the stack. Table 1 shows how differing types of arguments are passed.

Table 1. Arguments of Differing Type

Type	Size	Memory (Low to High)
char	3 bytes	xx ?? ??
short	3 bytes	xx xx ??
int	3 bytes	xx xx xx

Table 1. Arguments of Differing Type (Continued)

Type	Size	Memory (Low to High)
long	6 bytes	xx xx xx xx ?? ??
float	6 bytes	xx xx xx xx ?? ??
double	6 bytes	xx xx xx xx ?? ??
pointer	3 bytes	xx xx xx

The called function performs the following tasks:

1. Push the frame pointer (i.e., the IX Register) onto the stack and allocate the local frame:
   - Set the frame pointer to the current value of the stack pointer
   - Decrement the stack pointer by the size of locals and temporaries, if required
2. Execute the code for the function.
3. If the function returns a scalar value, place it in the appropriate register, as defined in Table 2. For functions returning an aggregate value, see the Special Cases section on page 4.
4. Deallocate the local frame (i.e., set the stack pointer to the current value of the frame pointer) and restore the Frame Pointer Register (IX) from the stack.
5. Return.

Table 2 specifies how scalar values – those other than structs or unions – are returned.

Table 2. Arguments of Differing Type

Type	Register	Register Contents: Most to Least Significant
char	A	xx
short	HL	?? xx xx
int	HL	xx xx xx
long	E:HL	xx: xx xx xx
float	E:HL	xx: xx xx xx
double	E:HL	xx: xx xx xx
pointer	HL	xx xx xx

The function call mechanism described in this section is a dynamic call mechanism. In a dynamic call mechanism, each function allocates memory on stack for its locals and temporaries during the run time of the program. When the function has returned, the memory

that it was using is freed from the stack. Figure 1 shows a diagram of the eZ80Acclaim! C-Compiler's dynamic call frame layout.

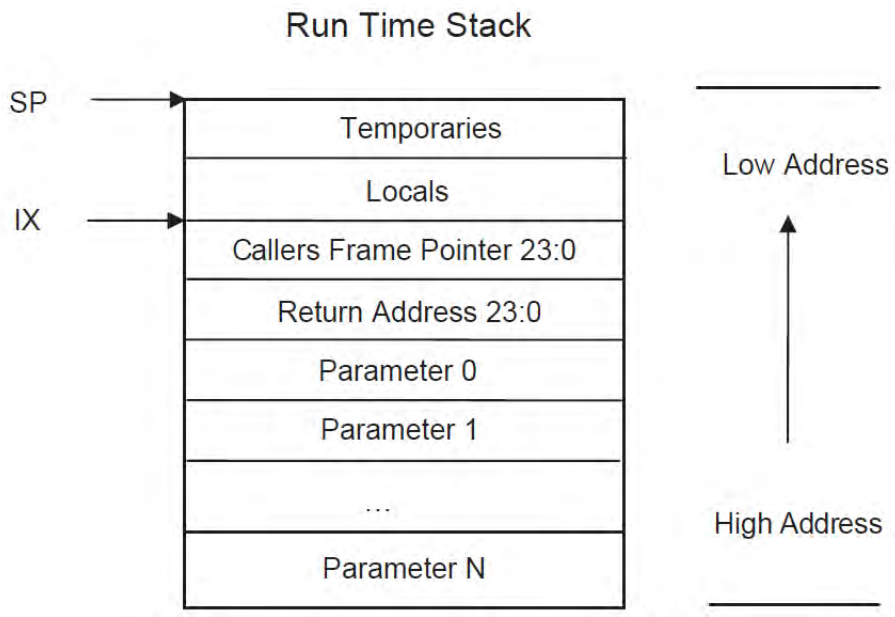

Figure 1. the Dynamic Call Frame Layout of the eZ80Acclaim! C-Compiler

Special Cases

Some function calls do not follow the mechanism described in the the Function Call Mechanism section on page 2. The *Returning Structure* and the *Not Allocating Local Frame* are examples of special case call mechanisms that do not follow the norm, as follows.

Returning Structure. If the function returns a structure, the caller allocates the space for the structure and then passes the address of the return space to the called function as an additional and/or first argument. To return a structure, the called function then copies the structure to the memory block pointed to by this argument.

Not Allocating A Local Frame. The compiler does not allocate a local stack frame for a function in the following cases:

- The function does not have any local stack variables, stack arguments or compiler-generated temporaries on the stack
- The function does not return a structure
- The function is compiled without the debug option

Calling Assembly Functions from C

The eZ80Acclaim! C-Compiler allows mixed C and assembly programming. A function written in assembly can be called from C if the assembly function follows the C calling conventions as described in the Calling Conventions section on page 1.

The following sections describe how to call assembly functions from C.

Function Naming Convention

Assembly function names must be preceded with an "_" (underscore) in order to be callable from C. The compiler prefixes C function names with an underscore in the generated assembly. For example, a call to `myfunc()` in C is translated to a call to `_myfunc` in generated assembly by the compiler.

Variable Naming Convention

When the compiler generates an assembly file from C code, all names of global variables are prefixed with an underscore.

Names of local static variables are prefixed with an underscore followed by a function number to avoid assembly errors when the same local static variable occurs more than once in the same file.

Argument Locations

The assembly function assigns the location of the arguments following the C calling conventions as described in the Calling Conventions section on page 1.

For example, if you are using the following C prototype:

```
  void myfunc(short arga, long argb, short *argc, char argd, int
arge)
```

The arguments are placed on the stack and their offsets from the Stack Pointer (SP) at the entry point of an assembly function are:

```
  arga: -3(SP)
  argb: -6(SP)
  argc: -12(SP)
  argd: -15(SP)
  arge: -18(SP)
```

Return Values

The assembly function returns the value in the location specified by the C calling convention, as described in the Calling Conventions section on page 1.

For example, if you are using the following C prototype:

```
  long myfunc(short arga, long argb, short *argc)
```

The assembly function returns the long value in registers `E:HL`.

Preserving Registers

The eZ80Acclaim! C-Compiler implements a caller save scheme. The assembly function is not expected to save and restore the registers it uses (unless it makes calls to C functions; in that case, it must save the registers it is using by pushing them on the stack before the call).

```
#include <ez80.h>

extern int addfunction(char var1, char var2);
extern float angle;
int x,y,sum=0;

void main(void)
{
  x=2;
  y=2;
  sum=addfunction(x,y);
  angle +=1;
  asm(“nop”);
}
```

Assembly Code _addfunction

The following routine adds two parameter values and returns the sum as an integer.

```
  .include <ez80f91.inc>
  .assume ADL=1

  XDEF _addfunction
  XDEF _angle

  segment DATA
_angle: df -.523599

  segment CODE
start:

_addfunction:
  push ix           ; push ix onto the stack and allocate local frame
  ld ix, 0
  add ix, sp        ; set ix to sp

  ld de, (ix+6)     ; get first variable and load onto register de
  ld hl, (ix+9)     ; get second variable and load onto register hl
  add hl, de        ; add the two values and store results on
                    ; register hl

  ld sp, ix         ; set sp to ix
  pop ix            ; restore ix from stack
  ret               ; return
```

```
  end
```

Calling C Functions from Assembly

The C functions that are provided with the compiler library can also be used to add functionality to an assembly program. You can also create your own C functions and call them from an assembly program.

Note: The C-Compiler precedes the function names with an underscore in the generated assembly. See the Function Naming Convention section on page 5.

Assembly File

The following example shows an assembly source file referencing the *sin* function, which is defined in the C math library.

```
  XREF _sin
  segment DATA
_angle:
  df 0.523599        ; angle in radians
_res:                ; result
  ds 4

  segment CODE
_myfunc:
  ...
  push DE            ; save the live data, if any
  ld BC, (_angle)
  push BC            ; push the argument
  ld A, (_angle+3)
  ld C,A
  push BC
  call _sin          ; call the C function
  pop BC             ; restore the stack by popping out the arguments
  pop BC
  ld (_res),HL       ; result is in the E:HL registers
  ld A,E
  ld (_res+3),A
  pop DE             ; restore the live data
  ...
```

Referenced C Function Prototype

The following math function is called by the assembly example above and returns a value as double.

```
double sin (double x);
```

Note: The eZ80 Acclaim! C Compiler treats doubles as if they are floats. For additional details, see the *Double Treated as Float* section of the Zilog Developer Studio II – eZ80Acclaim! User Manual (UM0144).

Configuration

The following tools were used to test the application described in this document.

- ZDS II – eZ80Acclaim! version 5.1.1
- eZ80F91 Development Kit (eZ80F910x00ZCOG)

Software

The following code example calls for Assembly routing that adds two parameters of character type and prints the sum via the UART in HyperTerminal.

Example 1A. A C routine calling an Assembly function with two parameters of the same type of character.

```
#include <ez80.h>
#include <stdio.h>

extern int addfunction(char var1, char var2);
extern float angle;


int x,y,sum=0;

void main(void)
{
x=2;
  y=2;
  sum = addfunction(x,y);
  printf("%s%d\n","Sum: ", sum);
  angle += 1;
  asm("nop");
}
```

The assembly code called by the C routine above adds two parameters of character type, with each parameter featuring a stack offset point of entry, as described in the the Argument Locations section on page 5 of this document. Each type varies in Stack offset point of entry, in this example the parameter is in character type, it has three bytes of stack offset point of entry that means if the argument which is a character type located at stack offset point 6 then the next argument will be located at stack offset point 9. See below code.

Example 1B. An Assembly routine with two parameters of the same type of character.

```
.include "ez80f91.inc"
.assume             ADL=1

XDEF _addfunction
XDEF _angle

segment DATA

_angle:             df 0.523599

segment CODE
start:

_addfunction:
push ix             ; push ix onto stack and allocate local frame
ld ix, 0
add ix, sp          ; set ix to sp
ld de, (ix+6)       ; get first variable and load onto register de
ld hl, (ix+9)       ; get second variable and load onto register hl
add hl, de          ; add the two values and store results on
                    ; register hl
ld sp, ix           ; set sp to ix
pop ix              ; restore ix from stack
ret                 ; return

end
```

Stack point-of-entry of arguments depend on the types of the arguments, and allow developers to include up to *n* arguments as long as these arguments are still within the capacity of MCU memory, as shown in the following two examples.

Example 2A. A C routine calling an Assembly function with three parameters of the same type of character.

```
#include <ez80.h>
#include <stdio.h>

extern int addfunction(char var1, char var2, char var3);
extern float angle;
```

```
int x,y,z,sum=0;

void main(void)
{
  x=2;
  y=2;
  z=5;
  sum = addfunction(x,y);
  printf("%s%d\n","Sum: ", sum);
  angle += 1;
  asm("nop");
}
```

Example 2B. An Assembly routine with three parameters of the same type of character.

```
.include "ez80f91.inc"
.assume ADL=1

XDEF _addfunction
XDEF _angle

segment DATA

_angle: df 0.523599

segment CODE
start:

_addfunction:
push ix            ; push ix onto stack and allocate local frame
ld ix, 0
add ix, sp         ; set ix to sp
ld de, (ix+6)      ; get first variable and load onto register de
ld hl, (ix+9)      ; get second variable and load onto register hl
add hl, de         ; add the two values and store results on ld de,
(ix+12)            ; get third variable and load onto register de
add hl, de         ; add the two values and store results on
                   ; register hl
ld sp, ix          ; set sp to ix
pop ix             ; restore ix from stack
ret                ; return

end
```

Example 3A. A C routine calling an Assembly function with three different types of parameters.

```
#include <ez80.h>
#include <stdio.h>
```

```
extern int addfunction(char var1, long var2, int var3);
extern float angle;


int x,y,z,sum=0;

void main(void)
{
x=2;
y=2;
z=5;
sum = addfunction(x,y);
printf("%s%d\n","Sum: ", sum);
angle += 1;
asm("nop");
}
```

Example 3B. An Assembly routine with three different types of parameters.

```
.include "ez80f91.inc"
.assume ADL=1

XDEF _addfunction
XDEF _angle

segment DATA

_angle: df 0.523599

segment CODE
start:

_addfunction:
push ix            ; push ix onto stack and allocate local frame
ld ix, 0
add ix, sp         ; set ix to sp
ld de, (ix+6)      ; get first variable and load onto register de
ld hl, (ix+12)     ; get second variable and load onto register hl
add hl, de         ; add the two values and store results on ld de,
(ix+15)            ; get third variable and load onto register de
add hl, de         ; add the two values and store results on
                   ; register hl
ld sp, ix          ; set sp to ix
pop ix             ; restore ix from stack
ret                ; return

end
```

References

The document referenced below is from an online course offering by the University of Illinois at Urbana-Champaign.

Mixing Assembly and C

The following documents describe the functional specifications and toolsets for the eZ80Acclaim! MCU. Each is available for download from the Zilog website.

Zilog Developer Studio II – eZ80Acclaim! User Manual (UM0144)

eZ80 CPU User Manual (UM0077)

eZ80F91 MCU Product Specification (PS0192)

Customer Support

To share comments, get your technical questions answered, or report issues you may be experiencing with our products, please visit Zilog's Technical Support page at http://support.zilog.com.

To learn more about this product, find additional documentation, or to discover other facets about Zilog product offerings, please visit the Zilog Knowledge Base at http://zilog.com/kb or consider participating in the Zilog Forum at http://zilog.com/forum.

This publication is subject to replacement by a later edition. To determine whether a later edition exists, please visit the Zilog website at http://www.zilog.com.

Warning: DO NOT USE THIS PRODUCT IN LIFE SUPPORT SYSTEMS.

LIFE SUPPORT POLICY

ZILOG'S PRODUCTS ARE NOT AUTHORIZED FOR USE AS CRITICAL COMPONENTS IN LIFE SUPPORT DEVICES OR SYSTEMS WITHOUT THE EXPRESS PRIOR WRITTEN APPROVAL OF THE PRESIDENT AND GENERAL COUNSEL OF ZILOG CORPORATION.

As used herein

Life support devices or systems are devices which (a) are intended for surgical implant into the body, or (b) support or sustain life and whose failure to perform when properly used in accordance with instructions for use provided in the labeling can be reasonably expected to result in a significant injury to the user. A critical component is any component in a life support device or system whose failure to perform can be reasonably expected to cause the failure of the life support device or system or to affect its safety or effectiveness.

Document Disclaimer

 Information in this publication concerning the devices, applications, or technology described is intended to suggest possible uses and may be superseded. ZILOG, INC. DOES NOT ASSUME LIABILITY FOR OR PROVIDE A REPRESENTATION OF ACCURACY OF THE INFORMATION, DEVICES, OR TECHNOLOGY DESCRIBED IN THIS DOCUMENT. ZILOG ALSO DOES NOT ASSUME LIABILITY FOR INTELLECTUAL PROPERTY INFRINGEMENT RELATED IN ANY MANNER TO USE OF INFORMATION, DEVICES, OR TECHNOLOGY DESCRIBED HEREIN OR OTHERWISE. The information contained within this document has been verified according to the general principles of electrical and mechanical engineering.