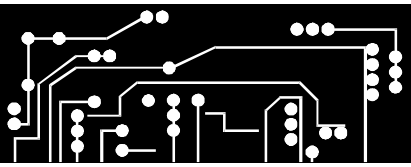
PowerPC Embedded Processors Application Note



Developing PowerPC Embedded Application Binary Interface (EABI) Compliant Programs

Microcontroller Applications
IBM Microelectronics
Research Triangle Park, NC
ppcsupp@us.ibm.com
Version: 1.0

September 21, 1998

Abstract – This application note describes the PowerPC Embedded Application Binary Interface (EABI) and is of interest to software developers writing assembly language code to interface with high-level language programs and software development tool vendors. The EABI is a set of conventions for embedded applications and development tools designed to insure compatibility for a described set of functionality. It provides software developers the opportunity to choose among various vendors of compiled libraries and software development tools. As long as they are EABI compliant, they will work correctly with each other.

1. Overview

An Application Binary Interface (ABI) specifies an interface for compiled application programs to system software. The Embedded Application Binary Interface (EABI) is derived from the PowerPC ABI Supplement to the UNIX System V ABI. The PowerPC ABI Supplement was designed for workstation operating systems such as IBM's AIX and Apple's Mac OS. The EABI differs from the PowerPC ABI Supplement with the goal of reducing memory usage and optimizing execution speed, as these are prime requirements of embedded system software. The EABI describes conventions for register usage, parameter passing, stack organization, small data areas, object file, and executable file formats. This application note covers the following EABI topics:

- Development tool file formats
- Data types and alignment
- Register usage conventions
- Stack frame creation and organization
- Function parameter passing
- Small data area usage and organization

2. File Formats

Object and executable files complying with the EABI are in the Extended Linking Format (ELF) and debug information is contained in Debug with Arbitrary Record Format (DWARF) records. The current revision of the DWARF standard is 1.1.0. There is a proposed revision of the standard currently known as DWARF 2.0.0 which primarily adds features to support C++ code debugging. Although not yet an official standard, many tool providers have implementations of, or close to, the current proposed DWARF 2.0.0 standard.

3. Data Types and Alignment

The PowerPC architecture defines scalar (integer) data type sizes as shown in Table 1.

Table 1 - PowerPC scalar data types

Data type	Size (bytes)
Byte	1
Halfword	2
Word	4
Doubleword	8
Quadword	16

All data types are aligned in memory, and in the stack frame, on addresses that are a multiple of their size. For example, a word, since its size is 4 bytes, is aligned on an address evenly divisible by 4. The address of a halfword is evenly divisible by 2. An exception to this rule is quad-words when they are not contained in a union or structure; they only require alignment to eight byte boundaries. Arrays are aligned on the boundary required by the size of the data type of the array elements.

A structure (or union) is aligned based on the alignment requirements of the structures largest member. Thus, if the structure contains a doubleword, the doubleword member must begin on an address evenly divisible by 8. Padding of prior and subsequent members is done as needed to maintain their individual alignment requirements. The size of a structure is always a multiple of the structure's alignment. If necessary a structure is padded after the last member to increase its size to be a multiple of its alignment. EABI compliant compilers and assemblers will automatically create correctly aligned data allocations but the padding required may cause problems for some applications. An example would be a networking- protocol data packet, which has specific alignment requirements. For these situations some compilers allow the alignment feature to be turned off, or overridden with different boundary values. For example, the IBM HighC/C++ compiler has a `#pack` pragma for this purpose. Since non-aligned data access requires multiple bus cycles for reads and writes, and perhaps even software assistance through an exception handler, performance will be decreased. Non-aligned data access should be avoided if at all possible.

Table 2 shows the ANSI C language data types and their sizes. For all types, NULL is defined as the value zero. Signed and unsigned integer types have the same size in all cases.

Table 2 - PowerPC ANSI C data types

ANSI C data type	PowerPC Data type	Size (bytes)
char	byte	1
short	halfword	2
int	word	4
long int	word	4
enum	word	4
pointer	word	4
float	word	4
double	doubleword	8
long double	quadword	16

4. Register Usage Conventions

The PowerPC architecture defines 32 general purpose registers (GPRs) and 32 floating-point registers (FPRs). The EABI classifies registers as volatile, nonvolatile, and dedicated. Nonvolatile registers must have their original values preserved, therefore, functions modifying nonvolatile registers must restore the

original values before returning to the calling function. Volatile registers do not have to be preserved across function calls.

Three nonvolatile GPR's are dedicated for a specific usage, R1, R2, and R13. R1 is dedicated as the stack frame pointer (SP). R2 is dedicated for use as a base pointer (anchor) for the read-only small data area. R13 is dedicated for use as an anchor for addressing the read-write small data area. Dedicated registers should never be used for any other purpose, not even temporarily, because they may be needed by an exception handler at any time. All the PowerPC registers and their usage are described in Table 3.

Table 3 - PowerPC EABI register usage

Register	Type	Used for:
R0	Volatile	Language Specific
R1	Dedicated	Stack Pointer (SP)
R2	Dedicated	Read-only small data area anchor
R3 - R4	Volatile	Parameter passing / return values
R5 - R10	Volatile	Parameter passing
R11 - R12	Volatile	
R13	Dedicated	Read-write small data area anchor
R14 - R31	Nonvolatile	
F0	Volatile	Language specific
F1	Volatile	Parameter passing / return values
F2 - F8	Volatile	Parameter passing
F9 - F13	Volatile	
F14 - F31	Nonvolatile	
Fields CR2 - CR4	Nonvolatile	
Other CR fields	Volatile	
Other registers	Volatile	

5. Stack Frame Conventions

The PowerPC architecture does not have a push/pop instruction for implementing a stack. The EABI conventions of stack frame creation and usage are defined to support parameter passing, nonvolatile register preservation, local variables, and code debugging. They do this by placing the various data into the stack frame in a consistent manner. Each function which either calls another function or modifies a nonvolatile register must create a stack frame from memory set aside for use as the run-time stack. If a function is a leaf function (meaning it calls no other functions), and does not modify any nonvolatile registers, it does not need to create a stack frame.

The SP always points to the lowest addressed word of the currently executing functions stack frame. Each new frame is created adjacent to the most recently allocated frame in the next available lower addressed memory. The stack frame is created by a function's prologue code and destroyed in its epilogue code. Stack frame creation is done by decrementing the SP just once, in the function prologue, by the total amount of space required by that function. To insure the SP update is an atomic operation that cannot be interrupted, a store-with-update (stwu) instruction is used. The prologue will also save any nonvolatile registers the function uses into the stack frame. Below is an example function prologue.

```
FuncX: mflr %r0              ; Get Link register
       stwu %r1,-88(%r1)     ; Save Back chain and move SP
       stw  %r0,+92(%r1)     ; Save Link register
       stmw %r28,+72(%r1)    ; Save 4 non-volatiles r28-r31
```

The stack frame is removed in the function's epilogue by adding the current stack frame size to the SP before the function returns to the calling function. The epilogue code of a function restores all registers saved by the prologue, de-allocates the current stack frame by incrementing the SP, then returns to the calling function. The following function epilogue example corresponds to the above prologue.

```
lwz  %r0,+92(%r1)     ; Get saved Link register
mtlr %r0              ; Restore Link register
lmw  %r28,+72(%r1)    ; Restore non-volatiles
addi %r1,%r1,88       ; Remove frame from stack
blr                   ; Return to calling function
```

Figure 1 illustrates the stack frame concept by using a 2-level deep, function calling example. At time 1, function A exists and calls function B. At 2, B's prologue code has executed and created B's frame. At 3, B has called C and C's prologue code has executed. At 4, function C has terminated and C's epilogue code has destroyed C's frame by incrementing the value in the SP.

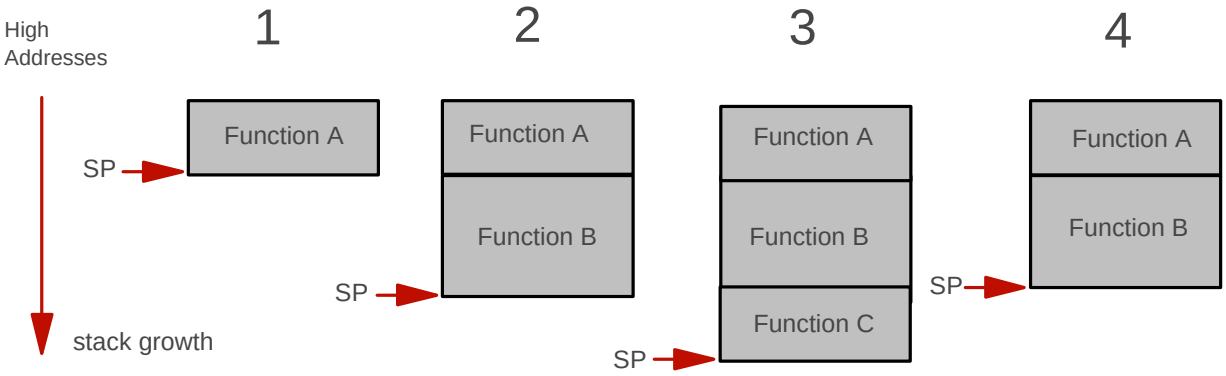


Figure 1 - Stack Frame creation and destruction

The stack frame is always doubleword aligned (on 8 byte boundaries) by using padding, if necessary. Figure 2 shows the stack frame organization including all optional areas.

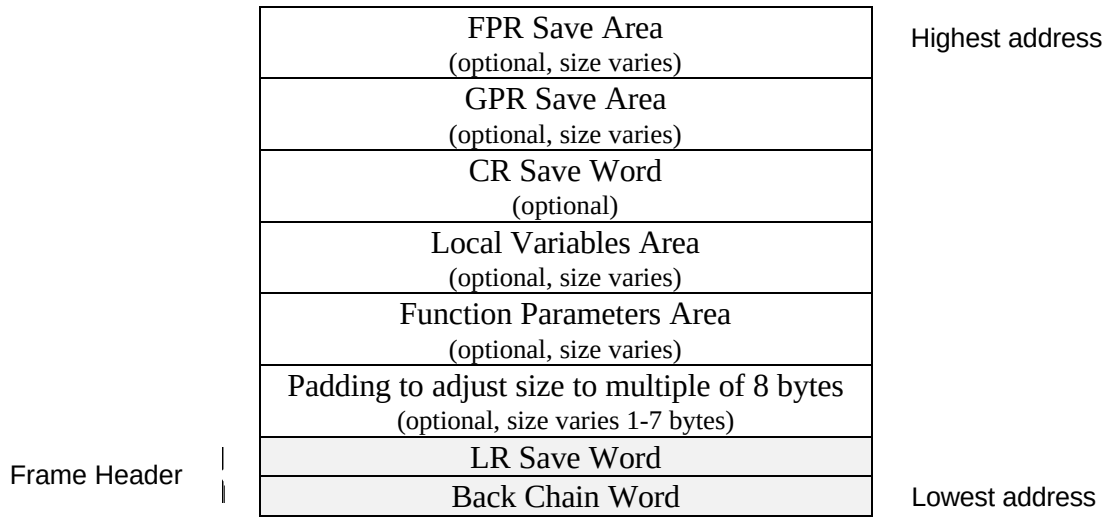


Figure 2 - EABI Stack Frame

All stack frames have a header consisting of two fields - the Back Chain Word and the Link Register (LR) Save Word. The Back Chain Word contains the address of the previous frames Back Chain Word field, thereby forming a linked-list of stack frames. The Back Chain Word is always located at the lowest

address of the stack frame. The LR Save Word is used by functions to store the current value of the Link Register prior to modifying it. The value in the LR upon entry into a subroutine represents the return address to the calling function. It is located in the word immediately above the Back Chain Word field.

The Function Parameter Area is optional and varies in size. It contains additional function arguments when there are to many to fit into the designated registers R3-R10. It is located just above the LR Save Word in the callers stack frame. The Local Variables Area is for functions local variables when there are more than can be contained in the available volatile registers. If a function modifies any of the nonvolatile Condition Register (CR) fields it must save the entire CR in the CR Save Area.

The General Purpose Register (GPR) Save Area is optional and varies in size. When saving any GPR, all the GPRs from the lowest to be saved up through R31, inclusive, are saved. For example, if a function is modifying R17, it must create a stack frame large enough to contain R17 through R31 in its GPR Save Area. The same conventions apply for the Floating point Register Save Area for saving the FPR nonvolatile registers. Code for implementations of the PowerPC that do not have floating-point hardware would not create the FP Save Area as there are no FPRs to save.

6. Parameter Passing

For PowerPC processors it is more efficient to pass arguments in registers than using memory. Up to eight scalar values are passed by using R3 through R10 and return values are passed back in R3 and R4. Up to eight arguments of the floating point data type can be passed using F1 to F8 and F1 is used to return a floating-point value. If there are more arguments than can be passed using the registers, space for the additional arguments is allocated for them in the stack frame's Function Parameters Area. Likewise, returned values that will not fit into R3 and R4 (or F1) are also passed by using the Function Parameters Area. The following C code fragment illustrates the concept.

```
#include "stdio.h"
void func1(int);
int var1;

main(){
   var1 = 4;
   func1(var1);
}

void func1(int arg1){
   printf("func1 - arg1 value: %d\n",arg1);
}
```

To implement the C language statements the following assembly language instructions illustrate loading and passing the value in `var1` to `func1`. After var1 is set to 4, R3 is loaded with the value in `var1` in order to pass it as an argument. The `lwz` instruction is used to load R3. Notice that after the instructions to set `var1 = 4`, R12 contains the high order 16-bits of the address of `var1` and is therefore used by the `lwz` instruction. R3 is used since it is the first available parameter passing register for integer values.

```
var1 = 4;
   li     %r11,4
   addis  %r12,%r0,var1@ha
   stw    %r11,var1@l(%r12)

func1(var1);
   lwz    %r3,var1@l(%r12)
   bl     func1
```

7. Small Data Areas

The EABI has a construct known as the Small Data Area (SDA) designed to take advantage of the PowerPC base plus displacement addressing mode. The displacement is a signed 16-bit value, therefore a total of 64k (plus or minus a 32K offset) bytes may be addressed without changing the value in a base register. The 16-bit displacement fits, along with the instruction op-code, into a single instruction word. This fact means it is a more memory efficient method of accessing a variable than referencing it by using a full 32-bit address. That's because only one instruction word is required instead of the two needed to access it as a 32-bit address. SDAs are useful for global and static variables and constants.

There are two SDAs, one for read-write variables and a second for read-only variables. The small data areas are referenced by a base register loaded once, when the C runtime environment is initialized. R2 is the base for the read-only (const type) small data area, and R13 is the base for the read-write (non-const type) small data area.

Variables for the R13 based read-write SDA are contained in one of two ELF segments, either .sdata, or .sbss. For initialized read-write variables, .sdata is used and .sbss is used for non-initialized read-write variables. Typically the .sbss variables are given a default initial value of 0 at run-time. Since this SDA is read-write it must be located in RAM.

Here is an example instruction to fetch the value of a read-write small data area variable. It is located at an offset of 32 bytes greater than the anchor value:

```
lwz  r29,32(r13)
```

Variables for the R2 based read-only SDA are contained in one of the segments .sdata2 and .sbss2. For initialized read-only variables, .sdata2 is used. For non-initialized variables, .sbss2 is used. Typically, non-initialized variables are given a default initial value of 0 at run-time. Since the SDA is read-only, it may be located in ROM as long as initialization of the .sbss2 segment contained variables is not required.

The PowerPC architecture treats R0 as a value of zero (not the content of R0) when used as the base register for the base + displacement addressing mode for some instructions. These instructions include the load, store, and various cache management instructions. Therefore, R0 acts as an anchor for a third, implicit, small data area which includes the lowest and highest 32k bytes of the processor memory address space.

With the IBM HighC/C++ compiler, placement of variables into SDAs is enabled using the pragma Push_small_data. By default, global variables are not placed into an SDA. The pragma can be invoked by using a compiler option. The option "-Hpragma=Push_small_data(4;0)" instructs that read-write variables that are 4 bytes or less in size should be stored in the read-write SDA. It also instructs that no read-only variables are to be placed into the read-only SDA by specifying the value 0, for the second argument. The following C program fragment will help illustrate the machine instructions used to access a global variable.

```
int var1;

main()
{
  var1 = 4;
  func1(var1);
}
```

Below are the assembly language instructions generated by the compiler for writing the value 4 to the global variable `var1,` when it is not in an SDA. Three instructions are required to store a value into `var1`.

```
li      %r11,4
addis   %r12,%r0,var1@ha
stw     %r11,var1@l(%r12)
```

1. `li` gets the value to be set into register R11.
2. `addis` is used to load R12 with the high-order halfword of the address of var1.
3. `stw` uses the base + displacement addressing mode. The displacement, from the base address in R12, is the low halfword of `var1's` address.

When in the read-write SDA, the resulting two instructions for writing a value to `var1` are:

```
li      %r12,4
stw     %r12,var1@sdaxr(%r13)
```

Note that to use any SDAs, you need the C runtime environment creation code to initialize the small data area anchor registers. For the IBM evaluation kit user, you can do this by adding the following code to the ./samples/bootllib.s routine right before the jump to the _kernel_entry routine. The macros *_SDA*_BASE_ and *_SDA2*_BASE_ are defined automatically by the linker if the associated SDAs are used. For *_SDA*_BASE_, the value is the address to which all data in the .sdata and .sbss sections can be addressed using a 16-bit signed offset. If an SDA is not used the associated macro's value will be zero.

```
!*************************************************************************
! INITIALIZATION OF BASE REGISTERS FOR SMALL DATA AREAS:
!*************************************************************************
   lis     %r2,_SDA2_BASE_@ha          ! r2 is the read-only SDA anchor
   addi    %r2,%r2,_SDA2_BASE_@l
   lis     %r13,_SDA_BASE_@ha          ! r13 is the read-write SDA anchor
   addi    %r13,%r13,_SDA_BASE_@l
```

For comparison, Table 4 shows the resulting changes in code size and execution speed for a Dhrystone benchmark. The compiler used was the IBM HighC/C++ compiler v3.61 and execution took place on the IBM PowerPC 401GF evaluation board. Row 1 shows the results of using options for pure speed (-O6) and allowing inlining of functions called up to 150 times (-Hic=150) that have fewer than 150 nodes (-Hit=150). Row 2 adds the use of SDAs for any read-write variable whose data type has a size of 8 bytes or fewer, and any read-only variable whose data type size is of 4 bytes or smaller.

Compile options	dhry_1.o code size	dhry_2 code size	benchmark result
-O6 -Hic=150 -Hit=150	2300 bytes	572 bytes	88KDhry/sec
-O6 -Hic=150 -Hit=150 -Hpragma=Push_small_data(8;4)	1988 bytes	552 bytes	77kDhry/sec

Table 4 - Small data area effects on Dhrystone application

8. Conclusion

The EABI provides for vendor independent tool interoperability via the ELF/DWARF file format standards. This allow developers to mix and match various EABI compliant components to create a software development tool chain for their needs. In addition, the EABI standards on register usage and parameter passing also allow independently developed code to be reused without modification.

9. References

The following references are presented in a suggested reading order progressing from a comprehensive overview through the standards documents from most PowerPC specific to most general.

- *Programming PowerPC Embedded Applications*, Embedded Systems Programming magazine, December 1995. Back issue information available from embedded@halldata.com
- *PowerPC Embedded Application Binary Interface, Version 1.0*, IBM and Motorola, January 10, 1995. Available from ESOFTA at http://www.esofta.com/pdfs/ppceabi.pdf
- *System V Application Binary Interface, PowerPC Processor Supplement*, Sun Microsystems and IBM, September 1995. Available from ESOFTA at http://www.esofta.com/pdfs/SVR4abippc.pdf
- *System V Application Binary Interface*, *Third Edition*, UNIX Systems Laboratories, 1994
- *DWARF Debugging Information Format, Revision 1.1.0,* UNIX International October 6, 1992. Available from ESOFTA at http://www.esofta.com/pdfs/dwarf.v1.1.0.pdf