GNUPRO® TOOLKIT

User's Guide for IBM AIX™

July 2000
Version Beta 4.0

redhat.

GNUPro Warranty

How to Contact Red Hat

Red Hat Corporate Headquarters
2600 Meridian Parkway
Durham, NC 27713 USA
Telephone (toll free): +1 888 REDHAT 1
Telephone (main line): +1 919 547 0012
Telephone (FAX line): +1 919 547 0024
Website: `http://www.redhat.com/`

Part #: 300-400-1010096-B4.0

Contents

Introduction

The GNUPro® Toolkit from Red Hat is a complete solution for C and C++ development for AIX on PowerPC®. The tools include the compiler, interactive debugger and utilities libraries.This User's Guide consists of an introduction to the features of the GNUPro Toolkit, as well as a tutorial and reference for IBM AIX-specific features of the main GNUPro tools.

See `http://www.redhat.com/apps/support` for GNUPro Toolkit documentation.

Toolkit Features

The following describes IBM AIX-specific features of the GNUPro Toolkit.

Supported Target:
PowerPC/rs6000

Supported Host:

CPU	***Operating System***
PowerPC	AIX 4.3.3

Object File Format

The IBM AIX tools support the XCOFF object file format.

GNUPro Toolkit Components

The AIX PowerPC package includes the following supported tools:

Tool Description	***Tool Name***
GCC compiler	`gcc`
C++ compiler	`g++`
GAS assembler	`as`

Binary Utilities	`ar`
	`nm`
	`objcopy`
	`objdump`
	`ranlib`
	`readelf`
	`size`
	`strings`
	`strip`

GDB debugger	`gdb`

Case Sensitivity

The following strings are case sensitive:

- command line options
- assembler labels
- linker script commands
- section names
- file names
- file names within makefiles

The following strings are not case sensitive:

- GDB commands
- assembler instructions and register names

Filenames are case sensitive when passed to GCC.

Document Conventions

This documentation uses the following general conventions:

Italic Font
: Indicates a new term that will be defined in the text and items called out for special emphasis.

Bold Font
: Represents menus, window names, and tool buttons.

Bold Italic Font
: Denotes book titles, both hardcopy and electronic.

`Plain Typewriter Font`
: Denotes code fragments, command lines, contents of files, and command names; also indicates directory, file, and project names where they appear in body text.

`Italic Typewriter Font`
: Represents a variable for which an actual value should be substituted.

1

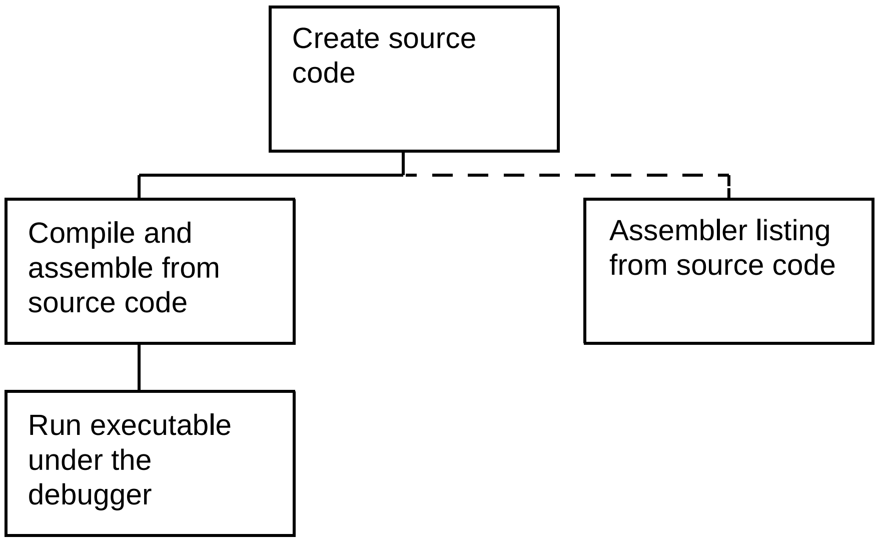
Tutorials

This section gives examples of how to use the main utilities. For more detail, refer to the individual utility manuals.

NOTE It is important to remember that the GNUPro Toolkit is case sensitive on all operating systems. Therefore, enter all commands and options exactly as indicated in this document.

Overview

The following chart outlines the sequence of steps in the tutorial. The assembler listing from source code is optional.


Create source code
Compile and assemble from source code
Assembler listing from source code
Run executable under the debugger

Tutorial

The following examples were created using GDB (GNUPro Debugger) in command-line mode. They may also be reproduced using the command prompt in the **Console Window** of Insight (the GUI interface to the GNUPro Debugger).

Create Source Code

Create the following sample source code and save it as `hello.c`. Use this program to verify correct installation.

```
#include <stdio.h>

int a, c;

void foo(int b)
{
  c = a + b;
  printf("%d + %d = %d\n", a, b, c);
}

int main()
```

```
{
  int b;

  a = 3;
  b = 4;
  printf("Hello, world!\n");
  foo(b);
  return 0;
}
```

Compile and Assemble from Source Code

To compile, assemble and link this example to run on the simulator, type:

```
gcc -g -o hello hello.c
```

The `-g` option generates debugging information and the `-o` option specifies the name of the executable to be produced. Other useful options include `-O` for standard optimization, and `-O2` for extensive optimization. When no optimization option is specified GCC will not optimize. Refer to “GNU CC Command Options” in *Using GNU CC* in ***GNUPro Compiler Tools*** for a complete list of available options.

Run under the Debugger

To start GDB, type:

```
gdb -nw hello
```

The `-nw` option was used to select the command line interface to GDB, which is useful for making transcripts such as the one above. The `-nw` option is also useful when you wish to report a bug in GDB, because a sequence of commands is simpler to reproduce. If you are running an X11 based server and your `DISPLAY` environment variable is set, GDB will start the Insight interface by default.

After the initial copyright and configuration information GDB returns its own prompt: **`(gdb)`.**

Exiting GDB

To exit GDB, type `quit` at the **`(gdb)`** prompt. The default prompt returns.

Assembler Listing from Source Code

The following command produces an assembler listing:

```
gcc -c -g -O -Wa,-l hello.c
```

The `-c` option tells GCC to compile or assemble the source files, but not to link. The `-O` option produces optimized code. The `-Wa` option tells the compiler to pass the comma-separated list of options, which follows it, to the assembler. The assembler option `-l` requests an assembler listing. Here is a partial excerpt of the output.

```
0     221 |                              .foo:
0     222 |                              .stabx "foo:F-11",.foo,142,0
0     223 |                              .function .foo,.foo,16,>
0     224 |                                        .bf     6
0     225 |                              .stabx "b:R-1",5,132,0
0     226 |                                        .line   1
0     227 |                                        .extern __mulh
0     228 |                                        .extern __mull
0     229 |                                        .extern __divss
0     230 |                                        .extern __divus
0     231 |                                        .extern __quoss
0     232 |                                        .extern __quous
0     233 |  COM  .text  00000000  7c0802a6        mflr 0
0     234 |  COM  .text  00000004  90010008        stw 0,8(1)
0     235 |  COM  .text  00000008  9421ffc8        stwu 1,-56(1)
0     236 |  COM  .text  0000000c  7c651b78        mr 5,3
0     237 |                                        .line   2
0     238 |  COM  .text  00000010  81620000        lwz 11,LC..0(2)
0     239 |  COM  .text  00000014  81220004        lwz 9,LC..1(2)
0     240 |  COM  .text  00000018  80890000        lwz 4,0(9)
0     241 |  COM  .text  0000001c  7cc52214        add 6,5,4
0     242 |  COM  .text  00000020  90cb0000        stw 6,0(11)
0     243 |                                        .line   3
0     244 |  COM  .text  00000024  80620008        lwz 3,LC..3(2)
0     245 |  COM  .text  00000028  4bffffd9        bl .printf
0     246 |  COM  .text  0000002c  60000000        nop
0     247 |                                        .line   4
0     248 |                                        .ef     9
```

2

Reference

This section describes the ABI and PowerPC-specific attributes of the main GNUPro tools.

- Compiler
- ABI Summary
- Assembler
- Debugger

Compiler

This section describes PowerPC-specific features of the GNUPro Compiler.

Command Line Options

For a list of available generic compiler options, see “GNU CC Command Options” in *Using GNU CC* in ***GNUPro Compiler Tools***. The following options are supported for IBM RS/6000 and PowerPC.

These `-m` options are defined for the IBM RS/6000 and PowerPC:

`-mpower`
`-mno-power`
`-mpower2`
`-mno-power2`
`-mpowerpc`
`-mno-powerpc`
`-mpowerpc-gpopt`
`-mno-powerpc-gpopt`
`-mpowerpc-gfxopt`
`-mno-powerpc-gfxopt`

GCC supports two related instruction set architectures for the RS/6000 and PowerPC. The POWER instruction set are those instructions supported by the rios chip set used in the original RS/6000 systems and the PowerPC instruction set is the architecture of the Motorola MPC5xx, MPC6xx, MPC8xx microprocessors, and the IBM 4xx microprocessors.

Neither architecture is a subset of the other. However there is a large common subset of instructions supported by both. An MQ register is included in processors supporting the POWER architecture.

You use these options to specify which instructions are available on the processor you are using. The default value of these options is determined when configuring GCC.

NOTE The `-mcpu` option overrides the specification of the options listed above. We recommend you use the `-mcpu` option instead of these options.

The `-mpower` option allows GCC to generate instructions that are found only in the POWER architecture and to use the MQ register. Specifying `-mpower2` implies `-power` and also allows GCC to generate instructions that are present in the POWER2 architecture but not the original POWER architecture.

The `-mpowerpc` option allows GCC to generate instructions that are found only in the 32-bit subset of the PowerPC architecture.

Specifying `-mpowerpc-gpopt` implies `-mpowerpc` and also allows GCC to use the optional PowerPC architecture instructions in the General Purpose group, including floating-point square root.

Specifying `-mpowerpc-gfxopt` implies `-mpowerpc` and also allows GCC to use the optional PowerPC architecture instructions in the Graphics group, including floating-point select.

If you specify both `-mno-power` and `-mno-powerpc`, GCC will use only the instructions in the common subset of both architectures plus some special AIX common-mode calls, and will not use the MQ register. Specifying both `-mpower` and `-mpowerpc` permits GCC to use any instruction from either architecture and to allow use of the MQ register; specify this for the Motorola MPC601.

`-mnew-mnemonics`
`-mold-mnemonics`

Select which mnemonics to use in the generated assembler code. Specifying `-mnew-mnemonics` requests output that uses the assembler mnemonics defined for the PowerPC architecture, while `-mold-mnemonics` requests the assembler mnemonics defined for the POWER architecture. Instructions defined in only one architecture have only one mnemonic; GCC uses that mnemonic irrespective of which of these options is specified.

GCC defaults to the mnemonics appropriate for the architecture in use. Specifying `-mcpu=`*`CPU_TYPE`* sometimes overrides the value of these option. Unless you are building a cross-compiler, you should normally not specify either `-mnew-mnemonics` or `-mold-mnemonics`, but should instead accept the default.

`-mcpu=`*`CPU_TYPE`*

Set architecture type, register usage, choice of mnemonics, and instruction scheduling parameters for machine type *`CPU_TYPE`*.

Supported values for *`CPU_TYPE`* are:

rs6000	rios1	rios2	rsc	601	602
603	603e	604	604e	620	740
750	power	power2	powerpc	403	505
801	821	823	860	common	

`-mcpu=power`, `-mcpu=power2`, and `-mcpu=powerpc` specify generic POWER, POWER2 and pure PowerPC (i.e., not MPC601) architecture machine types, with an appropriate, generic processor model assumed for scheduling purposes.

Setting `-mcpu` equal to `rios1`, `rios2`, `rsc`, `power`, or `power2` enables the `-mpower` option and disables the `-mpowerpc` option.

Setting `-mcpu=601` enables both the `-mpower` and `-mpowerpc` options.

Setting `-mcpu` equal to `602`, `603`, `603e`, `604`, or `620` enables the `-mpowerpc` option and disables the `-mpower` option.

Setting `-mcpu` equal to `403`, `505`, `821`, `860`, or `powerpc`, enables the `-mpowerpc` option and disables the `-mpower` option.

Setting `-mcpu=common` disables both the `-mpower` and `-mpowerpc` options.

AIX versions 4 or greater select `-mcpu=common` by default, so that code will operate on all members of the RS/6000 and PowerPC families. In that case, GCC will use only the instructions in the common subset of both architectures plus some special AIX common-mode calls, and will not use the MQ register. GCC assumes a generic processor model for scheduling purposes.

Setting `-mcpu` equal to `rios1`, `rios2`, `rsc`, `power`, or `power2` also disables the `-new-mnemonics` option.

Setting `-mcpu` equal to `601`, `602`, `603`, `603e`, `604`, `620`, `403`, or `powerpc` also enables the `-new-mnemonics` option.

Setting `-mcpu` equal to `403`, `821`, or `860` also enables the `-msoft-float` option.

`-mtune=`*`CPU_TYPE`*

Sets the instruction scheduling parameters for machine type *`CPU_TYPE`*, but does not set the architecture type, register usage and choice of mnemonics like the `-mcpu` option. The same values for *`CPU_TYPE`* are used for the `-mtune` option as for the `-mcpu` option. The `-mtune` option overrides the `-mcpu` option in terms of instruction scheduling parameters.

`-mfull-toc`
`-mno-fp-in-toc`
`-mno-sum-in-toc`
`-mminimal-toc`

Modifies generation of the TOC (Table Of Contents), which is created for every executable file. The `-mfull-toc` option is selected by default. In that case, GCC will allocate at least one TOC entry for each unique non-automatic variable reference in your program. GCC will also place floating-point constants in the TOC. However, only 16,384 entries are available in the TOC.

If you receive a linker error message that saying you have overflowed the available TOC space, you can reduce the amount of TOC space used with the `-mno-fp-in-toc` and `-mno-sum-in-toc` options. The `-mno-fp-in-toc` option prevents GCC from putting floating-point constants in the TOC and the `-mno-sum-in-toc` option forces GCC to generate code to calculate the sum of an address and a constant at run-time instead of putting that sum into the TOC. You may specify one or both of these options. Each causes GCC to produce very slightly slower and larger code at the expense of conserving TOC space.

If you still run out of space in the TOC even when you specify both of these options, specify `-mminimal-toc` instead. This option causes GCC to make only one TOC entry for every file. When you specify this option, GCC will produce code that is slower and larger but which uses extremely little TOC space. You may wish to use this option only on files that contain less frequently executed code.

`-mxl-call`
`-mno-xl-call`

On AIX, pass floating-point arguments to prototyped functions beyond the register save area (RSA) on the stack in addition to argument FPRs. The AIX calling convention was extended but not initially documented to handle an obscure K&R C case of calling a function that takes the address of its arguments with fewer arguments than declared. AIX XL compilers access floating-point arguments which do not fit in the RSA from the stack when a subroutine is compiled without optimization. Because always storing floating-point arguments on the stack is inefficient and rarely needed, this option is not enabled by default and only is necessary when calling subroutines compiled by AIX XL compilers without optimization.

`-mthreads`
: Supports AIX Threads. Links an application written to use pthreads with special libraries and startup code to enable the application to run.

`-mpe`
: Supports the IBM RS/6000 SP Parallel Environment (PE). Link an application written to use message passing with special startup code to enable the application to run. The system must have PE installed in the standard location (`/usr/lpp/ppe.poe/`), or the `specs` file must be overridden with the `-specs` option to specify the appropriate directory location. The Parallel Environment does not support threads, so the `-mpe` option and the `-mthreads` option are incompatible.

`-msoft-float`
`-mhard-float`
: Generate code that does not use (uses) the floating-point register set. Software floating-point emulation is provided if you use the `-msoft-float` option, and pass the option to GCC when linking.

`-mmultiple`
`-mno-multiple`
: Generates code that uses (or does not use) the load multiple word instructions and the store multiple word instructions. These instructions are generated by default on POWER systems, and not generated on PowerPC systems.

  Do not use `-mmultiple` on little-endian PowerPC systems, since those instructions do not work when the processor is in little-endian mode. The exceptions are PPC740 and PPC750, which permit the instructions usage in little-endian mode.

`-mstring`
`-mno-string`
: Generate code that uses (does not use) the load string instructions and the store string word instructions to save multiple registers and do small block moves. These instructions are generated by default on POWER systems, and not generated on PowerPC systems. Do not use `-mstring` on little endian PowerPC systems, since those instructions do not work when the processor is in little-endian mode. The exceptions are PPC740 and PPC750 which permit the instructions usage in little endian mode.

`-mupdate`
`-mno-update`
: Generate code that uses (does not use) the load or store instructions that update the

base register to the address of the calculated memory location. These instructions are generated by default. If you use `-mno-update`, there is a small window between the time that the stack pointer is updated and the address of the previous frame is stored, which means code that walks the stack frame across interrupts or signals may get corrupted data.

`-mfused-madd`
`-mno-fused-madd`

Generates code that uses (or does not use) the floating-point multiply and accumulate instructions. These instructions are generated by default if hardware floating-point is used.

`-maix64`
`-maix32`

Enable either the 32-bit or 64-bit PowerPC ABI and calling conventions (64-bit pointers, 64-bit long type, and the infrastructure needed to support them). The default is `-maix32`.

Preprocessor Symbols

The compiler supports the following preprocessor symbols:

`_IBMR2`
`_POWER`
`_AIX`

Are always defined.

`_AIX32`

`_AIX64`

Indicates either 32-bit or 64-bit mode. Defined when `-maix32`, or `-maix64` are specified respectively.

`_LONG_LONG`

Is always defined. Indicates support for the `long long` data type.

`_ARCH_PWR`

Defined when compiling for the POWER architecture.

`_ARCH_PWR2`

Defined when compiling for the POWER2 architecture.

`_ARCH_PPC`

Defined when compiling for the PowerPC architecture.

`_ARCH_COM`

Defined when compiling for the common subset of the POWER and PowerPC architectures.

32-Bit ABI Summary

This section describes the 32-bit AIX ABI, which the tools adhere to by default.

Data Type Sizes and Alignments

The following table shows the size and alignment for all data types:

Type	***Size (bytes)***	***Alignment (bytes)***
`char`	1 byte	1 byte
`short`	2 bytes	2 bytes
`int`	4 bytes	4 bytes
`unsigned`	4 bytes	4 bytes
`long`	4 bytes	4 bytes
`long long`	8 bytes	8 bytes
`float`	4 bytes	4 bytes
`double`	8 bytes	4 bytes
pointer	4 bytes	4 bytes

- Alignment within aggregates (structures and unions) is as above, with padding added if needed
- Aggregates have alignment equal to that of their most aligned member
- Aggregates have sizes which are a multiple of their alignment

Register Usage

The following table shows how the registers are used:

Register	***Usage***
`r0`	Volatile register used in function prologs
`r1`	Stack frame pointer
`r2`	TOC pointer
`r3, r4`	Volatile parameter and return value register
`r5` through `r10`	Volatile registers used for function parameters
`r11` through `r13`	Volatile registers used during function calls
`r14` through `r31`	Nonvolatile registers used for local variables
`f0`	Volatile scratch register
`f1` through `f4`	Volatile floating point parameter and return value registers
`f5` through `f13`	Volatile floating point parameter registers
`f14` through `f31`	Nonvolatile registers
`LR`	Link register (volatile)
`CTR`	Loop counter register (volatile)

Register	***Usage***
`r0`	Volatile register used in function prologs
`XER`	Fixed point exception register (volatile)
`FPSCR`	Floating point status and control register (volatile)
`CR0-CR1`	Volatile condition code register fields
`CR2-CR4`	Nonvolatile condition code register fields
`CR5-CR7`	Volatile condition code register fields

Registers `r1`, `r14` through `r31`, and `f14` through `f31` are nonvolatile, which means that they preserve their values across function calls. Functions which use those registers must save the value before changing it, restoring it before the function returns. Register `r2` is technically nonvolatile, but it is handled specially during function calls.

Registers `r0`, `r3` through `r12`, `f0` through `f13`, and the special purpose registers `LR`, `CTR`, `XER`, and `FPSCR` are volatile, which means that they are not preserved across function calls. Furthermore, registers `r0`, `r2`, `r11`, and `r12` may be modified by cross-module calls, so a function can not assume that the values of one of these registers is that placed there by the calling function.

The condition code register fields `CR0`, `CR1`, `CR5`, `CR6`, and `CR7` are volatile. The condition code register fields `CR2`, `CR3`, and `CR4` are nonvolatile; so a function which modifies them must save and restore them.

Parameter Passing

The linkage convention specifies the methods for parameter passing and whether return values are placed in floating-point registers, general-purpose registers, or both. The general-purpose registers available for argument passing are `r3-r10`. The floating-point registers available for argument passing are `fp3-fp13`.

Prototyping affects how parameters are passed and whether parameter widening occurs. In nonprototyped functions, floating-point arguments are widened to type double, and integral types are widened to type int. In prototyped functions, no widening conversions occur except in arguments passed to an ellipsis function. Floating-point double arguments are only passed in floating-point registers. If an ellipsis is present in the prototype, floating-point double arguments are passed in both floating-point registers and general-purpose registers.

When there are more argument words than available parameter registers, the remaining words are passed in storage on the stack. The values in storage are the same as if they were in registers. Space for more than 8 words of arguments (floating-point and nonfloating-point) must be reserved on the stack even if all the arguments were passed in registers.

The size of the parameter area is large enough to contain all the arguments passed on any call statement from a procedure associated with the stack frame. Although not all the arguments for a particular call actually appear in storage, they can be regarded as forming a list in this area, each one occupying one or more words.

In C, all function arguments are passed by value, and the called function receives a copy of the value passed to it.

Call-by-value Parameters

In prototype functions with a variable number of arguments (indicated by an ellipsis as in a `function(...)`) the compiler widens all floating-point arguments to double precision.

Integral arguments (except for long int) are widened to int.

The following information refers to call by value. In the following list, arguments are classified as floating-point values or nonfloating-point values:

- Each nonfloating scalar argument requires one word and appears in that word exactly as it would appear in a general-purpose register.
- Each floating-point value occupies one word. Float doubles occupy two successive words in the list.
- Structure values appear in successive words as they would anywhere in storage, satisfying all appropriate alignment requirements. Structures are aligned by rounding up to the nearest full word, with any padding at the end. A structure smaller than a word is left-justified within its word or register. Larger structures can occupy multiple registers and can be passed partly in storage and partly in registers.
- Other aggregate values are passed by the caller making a copy of the structure and passing a pointer to that copy.
- A function pointer is passed as a pointer to the routine’s function descriptor. The first word contains the entry-point address.

TOC (Table of Contents)

The TOC is used to access global data by holding pointers to the global data.

The TOC section is accessed via the dedicated TOC pointer register `r2`. Accesses are normally made using the register indirect with immediate index mode supported by the PowerPC processor, which limits a single TOC section to 65,536 bytes, enough for 8,192 GOT entries.

The value of the TOC pointer register is called the TOC base. The TOC base is typically the first address in the TOC plus 0x8000, thus permitting a full 64-kilobyte TOC.

Pointers to Functions

A function pointer is a data type whose values range over function addresses. Function pointers are supported in contexts such as the target of a call statement or an actual argument of such a statement.

A function pointer is a full word quantity that is the address of a function descriptor. The function descriptor is a three-word object. The first word contains the address of the entry point of the procedure, the second has the address of the TOC of the module in which the procedure is bound, and the third is the environment pointer. There is only one function descriptor per entry point. It is bound into the same module as the function it identifies, if the function is external. The descriptor has an external name, which is the same as the function name, but without a leading dot (`.`). This descriptor name is used in all import and export operations.

Function Return Values

Functions pass their return values according to type:

Type	***Register***
`int`	`r3`
`short`	`r3`
`long`	`r3`
`long long`	`r3+r4`
`float`	`fp1`
`double`	`fp1`
`structure/union`	See Note

NOTE Structures and unions that will fit into general-purpose registers are returned in `r3`, or in `r3` and `r4` if necessary.

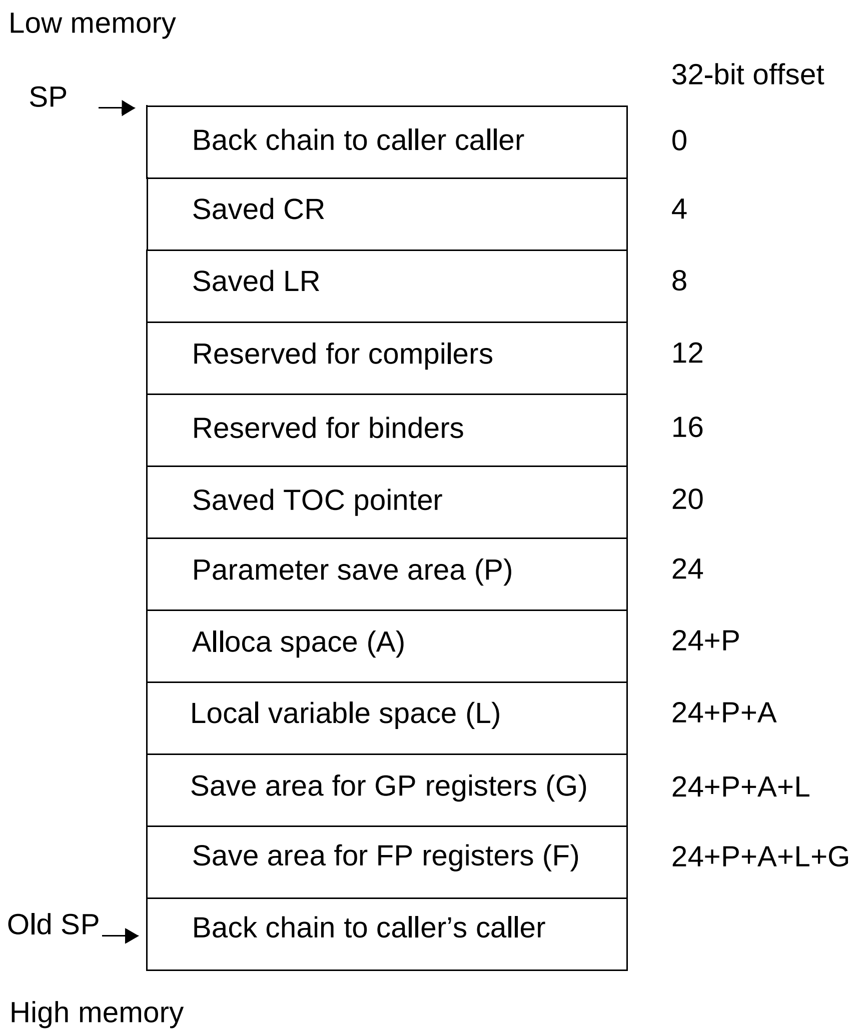
The caller handles larger structures and unions by passing a pointer to space allocated to receive the return value. The pointer is passed as a “hidden” first argument.

Stack Frame

This section describes the 32-bit PowerPC stack frame:

- The stack grows downwards from high addresses to low addresses.
- A leaf function does not need to allocate a stack frame if one is not needed.
- A frame pointer need not be allocated.
- The stack pointer shall always be aligned to 16-byte boundaries.

AIX stack frames look like this:



64-Bit ABI Summary

This section describes the 64-bit AIX ABI.

Data Type Sizes and Alignments

The following table shows the size and alignment for all data types:

Type	***Size (bytes)***	***Alignment (bytes)***
`char`	1 byte	1 byte
`short`	2 bytes	2 bytes
`int`	4 bytes	4 bytes
`unsigned`	4 bytes	4 bytes
`long`	8 bytes	8 bytes
`long long`	8 bytes	8 bytes
`float`	4 bytes	4 bytes
`double`	8 bytes	4 bytes
pointer	8 bytes	8 bytes

- Alignment within aggregates (structures and unions) is as above, with padding added if needed
- Aggregates have alignment equal to that of their most aligned member
- Aggregates have sizes which are a multiple of their alignment

Register Usage

The following table shows how the registers are used:

Register	***Usage***
`r0`	Volatile register used in function prologs
`r1`	Stack frame pointer
`r2`	TOC pointer
`r3`	Volatile parameter and return value register
`r4` through `r10`	Volatile registers used for function parameters
`r11` through `r12`	Volatile registers used during function calls
`r13`	Reserved for thread private data
`r14` through `r31`	Nonvolatile registers used for local variables
`f0`	Volatile scratch register
`f1` through `f4`	Volatile floating point parameter and return value registers
`f5` through `f13`	Volatile floating point parameter registers
`f14` through `f31`	Nonvolatile registers
`LR`	Link register (volatile)
`CTR`	Loop counter register (volatile)
`XER`	Fixed point exception register (volatile)
`FPSCR`	Floating point status and control register (volatile)

Register	***Usage***
`r0`	Volatile register used in function prologs
`CR0-CR1`	Volatile condition code register fields
`CR2-CR4`	Nonvolatile condition code register fields
`CR5-CR7`	Volatile condition code register fields

Registers `r1`, `r14` through `r31`, and `f14` through `f31` are nonvolatile, which means that they preserve their values across function calls. Functions which use those registers must save the value before changing it, restoring it before the function returns. Register `r2` is technically nonvolatile, but it is handled specially during function calls.

Registers `r0`, `r3` through `r12`, `f0` through `f13`, and the special purpose registers `LR`, `CTR`, `XER`, and `FPSCR` are volatile, which means that they are not preserved across function calls. Furthermore, registers `r0`, `r2`, `r11`, and `r12` may be modified by cross-module calls, so a function can not assume that the values of one of these registers is that placed there by the calling function.

The condition code register fields `CR0`, `CR1`, `CR5`, `CR6`, and `CR7` are volatile. The condition code register fields `CR2`, `CR3`, and `CR4` are nonvolatile; so a function which modifies them must save and restore them.

Parameter Passing

The linkage convention specifies the methods for parameter passing and whether return values are placed in floating-point registers, general-purpose registers, or both. The general-purpose registers available for argument passing are `r3-r10`. The floating-point registers available for argument passing are `fp3-fp13`.

Prototyping affects how parameters are passed and whether parameter widening occurs. In nonprototyped functions, floating-point arguments are widened to type double, and integral types are widened to type int. In prototyped functions, no widening conversions occur except in arguments passed to an ellipsis function. Floating-point double arguments are only passed in floating-point registers. If an ellipsis is present in the prototype, floating-point double arguments are passed in both floating-point registers and general-purpose registers.

When there are more argument words than available parameter registers, the remaining words are passed in storage on the stack. The values in storage are the same as if they were in registers. Space for more than 8 words of arguments (floating-point and nonfloating-point) must be reserved on the stack even if all the arguments were passed in registers.

The size of the parameter area is large enough to contain all the arguments passed on any call statement from a procedure associated with the stack frame. Although not all the arguments for a particular call actually appear in storage, they can be regarded as forming a list in this area, each one occupying one or more words.

In C, all function arguments are passed by value, and the called function receives a copy of the value passed to it.

Call-by-value Parameters

In prototype functions with a variable number of arguments (indicated by an ellipsis as in a *`function`*`(...)`) the compiler widens all floating-point arguments to double precision.

Integral arguments (except for long int) are widened to int.

The following information refers to call by value. In the following list, arguments are classified as floating-point values or nonfloating-point values:

- Each nonfloating scalar argument requires one word and appears in that word exactly as it would appear in a general-purpose register.
- Each floating-point value occupies one word.
- Structure values appear in successive words as they would anywhere in storage, satisfying all appropriate alignment requirements. Structures are aligned by rounding up to the nearest full word, with any padding at the end. A structure smaller than a word is left-justified within its word or register. Larger structures can occupy multiple registers and can be passed partly in storage and partly in registers.
- Other aggregate values are passed by the caller making a copy of the structure and passing a pointer to that copy.
- A function pointer is passed as a pointer to the routine's function descriptor. The first word contains the entry-point address.

TOC (Table of Contents)

The TOC is used to access global data by holding pointers to the global data.

The TOC section is accessed via the dedicated TOC pointer register `r2`. Accesses are normally made using the register indirect with immediate index mode supported by the PowerPC processor, which limits a single TOC section to 65,536 bytes, enough for 4,096 GOT entries.

The value of the TOC pointer register is called the TOC base. The TOC base is typically the first address in the TOC plus 0x8000, thus permitting a full 64-kilobyte TOC.

Pointers to Functions

A function pointer is a data type whose values range over function addresses. Function pointers are supported in contexts such as the target of a call statement or an actual argument of such a statement.

A function pointer is a full word quantity that is the address of a function descriptor. The function descriptor is a three-word object. The first word contains the address of the entry point of the procedure, the second has the address of the TOC (table of contents) of the module in which the procedure is bound, and the third is the environment pointer. There is only one function descriptor per entry point. It is bound into the same module as the function it identifies, if the function is external. The descriptor has an external name, which is the same as the function name, but without a leading dot (`.`). This descriptor name is used in all import and export operations.

Function Return Values

Functions pass their return values according to type:

Type	***Register***
`int`	`r3`
`short`	`r3`
`long`	`r3`
`long long`	`r3`
`float`	`fp1`
`double`	`fp1`
`structure/union`	See Note

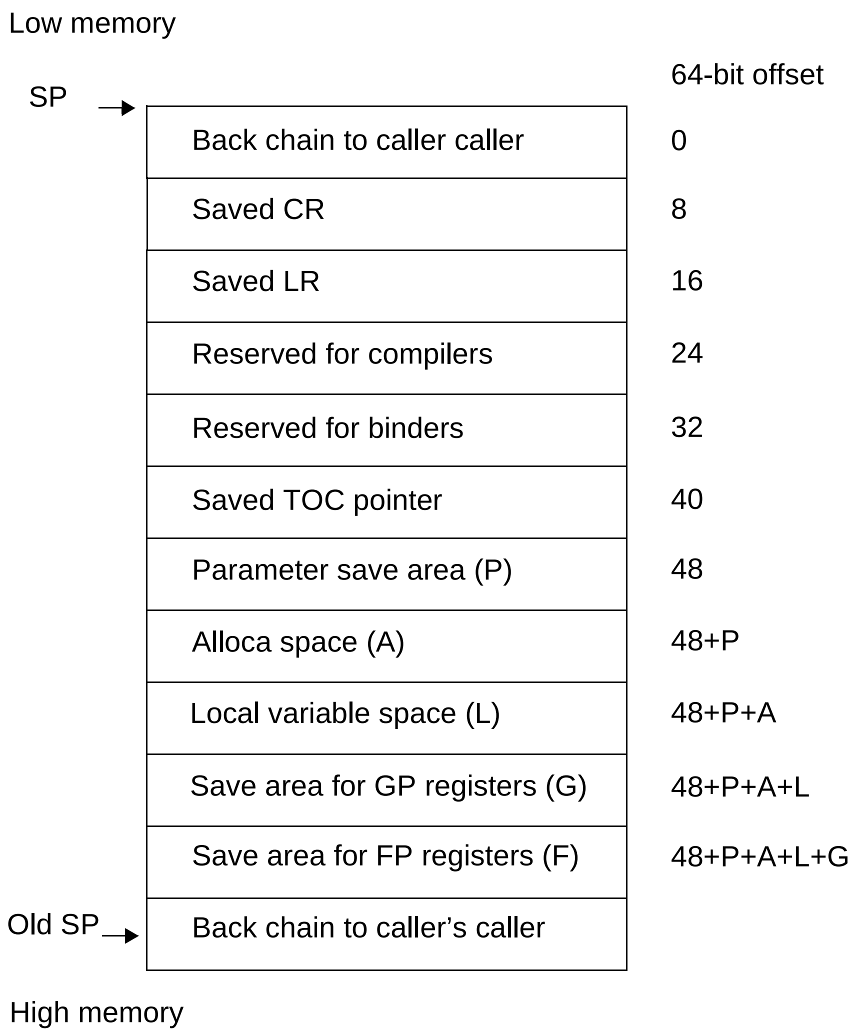
NOTE The caller handles structures and unions by passing a pointer to space allocated to receive the return value. The pointer is passed as a hidden first argument.

Stack Frame

This section describes the 64-bit PowerPC stack frame:

- The stack grows downwards from high addresses to low addresses.
- A leaf function does not need to allocate a stack frame if one is not needed.
- A frame pointer need not be allocated.
- The stack pointer shall always be aligned to 32-byte boundaries.

AIX stack frames look like:



Assembler

This section describes PowerPC-specific features of the GNUPro Assembler.

PowerPC-specific Command Line Options

For a list of available generic assembler options, refer to “Command Line Options” in *Using as* in ***GNUPro Utilities***.

Syntax

For more information about the PowerPC instruction set and PowerPC assembly conventions, see ***The PowerPC™ Architecture: A SPECIFICATION FOR A NEW FAMILIY OF RISC PROCESSORS*** (Morgan Kaufmann Publishers, Inc.), or ***PowerPC ™ Microprocessor Family: The Programming Environments*** (Published by both IBM (MPRPPCFPE-01) and Motorola (MPCFPE/AD))

Register Names

Integer registers depend upon whether you have a 32-bit, or a 64-bit chip. For 32-bit chips, there are 32 32-bit general (integer) registers, named `r0` through `r31`. For 64-bit chips, there are 32 64-bit general (integer) registers, named `r0` through `r31`. There are 32 64-bit floating-point registers, named `f0` through `f31`.

The compiler will generate assembly code, which uses the numbers zero through 31 to represent general-purpose registers.

The following symbols can be used as aliases for individual registers

Symbol	***Register***
`sp`	`r1`
`toc`	`r2`

Furthermore, the GNU tools recognize the PowerPC’s special registers:

Symbol	***Register***
`lr`	the link register
`ctr`	the count register
`cr0 ... cr7`	the condition registers

Other PowerPC special registers (`xer`, `fpscr`, etc.) are supported by the GNU tools, but do not have names since they are used implicitly by specific instructions (`qv: mcrx`). These registers may also be referenced in assembly language by number.

Assembler Directives

The initial character in all assembler directives is the dot (`.`). The first directive in the following chart starts with two dots (`..mri`).

`..mri`	`.ds.d`	`.ifnc`	`.rept`
`.ABORT`	`.ds.l`	`.ifndef`	`.rva`
`.abort`	`.ds.p`	`.ifne`	`.sbttl`

.align	.ds.s	.ifnes	.scl
.appfile	.ds.w	.ifnotdef	.sect
.appline	.ds.x	.include	.sect.s
.appline	.eb	.int	.section
.ascii	.ec	.irep	.section
.asciz	.ef	.irepc	.section.s
.balign	.ei	.irp	.set
.balignl	.eject	.irpc	.short
.balignw	.else	.lcomm	.short
.bb	.elsec	.lcomm	.single
.bc	.elseif	.lflags	.size
.bf	.end	.lglobl	.skip
.bi	.endc	.line	.sleb128
.bs	.endef	.linkonce	.space
.bss	.endfunc	.list	.spc
.byte	.endif	.llen	.stabd
.comm	.equ	.ln	.stabn
.comm	.equiv	.loc	.stabs
.common	.err	.long	.stabx
.common.s	.es	.long	.string
.csect	.exitm	.lsym	.struct
.data	.extern	.macro	.tag
.data	.extern	.mexit	.tc
.dc	.fail	.mri	.text
.dc.b	.file	.name	.text
.dc.d	.fill	.noformat	.this_GCC_requires_the_GNU_assembler
.dc.l	.float	.nolist	.this_gcc_requires_the_gnu_assembler
.dc.s	.format	.nopage	.title
.dc.w	.func	.octa	.toc
.dc.x	.function	.offset	.ttl
.dcb	.global	.optim	.type
.dcb.b	.globl	.org	.uleb128
.dcb.d	.hword	.p2align	.val
.dcb.l	.ident	.p2alignl	.vbyte
.dcb.s	.if	.p2alignw	.version
.dcb.w	.ifc	.page	.weak
.dcb.x	.ifdef	.plen	.word
.debug	.ifeq	.print	.word
.def	.ifeqs	.psize	.xcom
.dim	.ifge	.purgem	.xdef
.double	.ifgt	.quad	.xref
.ds	.ifle	.rename	.xstabs
.ds.b	.iflt	.rep	.zero

Debugger

For a complete description of the GNUPro Debugger, refer to ***GNUPro Debugging***

Tools.

PowerPC-specific Command Line Options

For the available generic debugger options, refer to *Debugging with GDB* in ***GNUPro Debugging Tools***. There are no PowerPC-specific debugger command line options.