GNUPro® Toolkit User’s Guide for IBM® AIX™ Development

April 2001

redhat.

GNUPro Warranty

The GNUPro Toolkit is free software, covered by the GNU General Public License, and you are welcome to change it and/or distribute copies of it under certain conditions. This version of GNUPro Toolkit is supported for customers of Red Hat.

For non-customers, GNUPro Toolkit software has NO WARRANTY.

Because this software is licensed free of charge, there are no warranties for it, to the extent permitted by applicable law. Except when otherwise stated in writing, the copyright holders and/or other parties provide the software “as is” without warranty of any kind, either expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. The entire risk as to the quality and performance of the software is with you. Should the software prove defective, you assume the cost of all necessary servicing, repair or correction.

In no event, unless required by applicable law or agreed to in writing, will any copyright holder, or any other party who may modify and/or redistribute the program as permitted above, be liable to you for damages, including any general, special, incidental or consequential damages arising out of the use or inability to use the program (including but not limited to loss of data or data being rendered inaccurate or losses sustained by you or third parties or a failure of the program to operate with any other programs), even if such holder or other party has been advised of the possibility of such damages.

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/`

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.

For documentation, see `http://www.redhat.com/support/manuals/gnupro.html`, and see `http://sources.redhat.com/sourcenav` for Source-Navigator documentation. For the most current release notes, find the `README` at the top level directory of the distribution.

The supported processor version is the PowerPC. The supported host is the AIX 4.3.2 PowerPC operating system. The supported target is PowerPC/RS6000.

The IBM AIX tools support the XCOFF object file format.

The AIX PowerPC package includes the tools shown in Table 1.

Table 1: Tools and their naming conventions

Tool description	***Tool name***
GCC compiler	`gcc`
C++ compiler	`g++`
Assembler	`as`

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

Debugger	`gdb`

IMPORTANT! Binaries for the Windows hosted toolchain use an `.exe` suffix. However, the `.exe` suffix does not need to be specified when running the executable.

Case sensitivity for Windows is dependent on system configuration. By default, file names under Windows are not case sensitive. File names are case sensitive under UNIX. File names are case sensitive when passed to the GNU C compiler (GCC), regardless of the operating system.

The following strings are case sensitive: command line options, assembler labels, linker script commands, section names, and file names within makefiles. The following strings are not case sensitive: debugger commands, assembler instructions, and register names.

For the tools to function properly, you must set environment variables.

- For the Microsoft Windows operating system, use the following examples as input for setting enviornment variables for the tools. Replace*`installdir`* with your default installation directory; *`yymmdd`* indicates the release date printed on the CD. Replace `H-`*`host`* (where *`host`* signifies the toolchain's name) with `H-i686-cygwin` as a name.

  ```
  SET PROOT=C:\installdir\aix-yymmdd
  SET PATH=%PROOT%\H-host\BIN;%PATH%
  SET INFOPATH=%PROOT%\info
  REM Set TMPDIR to point to a ramdisk if you have one
  SET TMPDIR=%PROOT%
  ```

- For the Sun Solaris and Red Hat Linux operating systems, use the following examples as input for setting environment variables for the tools. Replace *`installdir`* with your default installation directory; *`yymmdd`* indicates the release date printed on the CD. Replace `H-`*`host`* (where *`host`* signifies the toolchain's name) with `H-sparc-sun-solaris2.6` for Sun Solaris or `H-i686-pc-linux-gnu` for Red Hat Linux 6.0.

- For Bourne-compatible shells (`/bin/sh`, bash, or Korn shell), use the following example's input:

```
PROOT=installdir/aix-yymmdd
PATH=$PROOT/H-host/bin:$PATH
INFOPATH=$PROOT/info
export PATH SID_EXEC_PREFIX INFOPATH
```

- For C shells, use the following example's input:

```
set PROOT=installdir/aix-yymmdd
set path=($PROOT/H-host/bin $path)
setenv INFOPATH $PROOT/info
```

This documentation uses some general conventions (see Table 2).

Table 2: Documentation conventions

Text appearance	***Meaning***
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, file contents, and command names; also indicates directory, file, and project names where they appear in text.
`Italic Typewriter Font`	Represents a variable to substitute.
`Bold Typewriter Font`	Indicates command lines, options, and text output generated by the program.

1

Tutorial

This tutorial gives examples of how to use the tools. For more information about the tools, see `http://www.redhat.com/support/manuals/gnupro.html`.

IMPORTANT! Remember that GNUPro Toolkit is case sensitive Enter all commands and options exactly as indicated in this document.

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. See “GNU CC Command Options” in *Using GNU CC* in ***GNUPro Compiler Tools*** for a complete list of available options.

Run Executable 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 starts the Insight interface by default.

After the initial copyright and configuration information GDB returns its own prompt, **`(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

The following documentation describes the ABI and PowerPC-specific features of the GNUPro tools.

Compiler Features

The following documentation describes PowerPC-specific features of the GNUPro compiler. The following options are supported for IBM RS/6000 and PowerPC. For generic compiler options, see “GNU CC Command Options” in *Using GNU CC* in ***GNUPro Compiler Tools***.

`-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 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 MPC5*xx*, MPC6*xx*, MPC8*xx* microprocessors, and the IBM 4*xx* 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.

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.

IMPORTANT! The `-mcpu` option overrides the specification of the options listed above. It is recommended that you use the `-mcpu` option instead of these options.

`-mpower` allows GCC to generate instructions that are found only in the POWER architecture and to use the MQ register. Specifying `-mpower2` implies `-power`, allowing GCC to generate instructions that are present in the POWER2 architecture but absent in the original POWER architecture.

`-mpowerpc` 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`

Specifying `-mnew-mnemonics` requests output that uses the assembler mnemonics defined for the PowerPC architecture.

`-mold-mnemonics`

Specifying `-mold-mnemonics` requests the assembler mnemonics defined for the POWER architecture.

IMPORTANT! Instructions defined in only one architecture have only one mnemonic; GCC uses that mnemonic regardless 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`*

Sets architecture type, register usage, choice of mnemonics, and instruction scheduling parameters for machine type, *`cpu_type`*. Supported values for *`cpu_type`* are: `common`, `power`, `power2`, `powerpc`, `rs6000`, `rios1`, `rios2`, `rsc`, `403`, `505`, `601`, `602`, `603`, `603e`, `604`, `604e`, `620`, `740`, `750`, `801`, `821`, `823`, and `860`.

`-mcpu=power`, `-mcpu=power2`, and `-mcpu=powerpc` specify generic POWER, POWER2, and pure PowerPC (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 `-mpower` and `-mpowerpc`.

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

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

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

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

AIX version 4 or greater selects `-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 `-new-mnemonics`.

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

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

`-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 `-mcpu`. The same values for *`cpu_type`* are used for `-mtune` as for `-mcpu`. `-mtune` overrides `-mcpu` 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. `-mfull-toc` 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 saying you have overflowed the available TOC space, you can reduce the amount of TOC space used with `-mno-fp-in-toc` and `-mno-sum-in-toc`. `-mno-fp-in-toc` prevents GCC from putting floating-point constants in the TOC and `-mno-sum-in-toc` 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 causing 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, which causes GCC to make only one TOC entry for every file. When you specify `-mminimal-toc`, GCC will produce code that is slower and larger, using extremely little TOC space. Use `-mminimal-toc` only on files that contain less frequently executed code.

`-mxl-call`

Enables AIX XL compiler handling.

`-mno-xl-call`

Disables AIX XL compiler handling. This is the default setting.

IMPORTANT! 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,

`-mno-xl-call` 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`
: Generates code that does not use the floating-point register set.

`-mhard-float`
: Generates code that uses the floating-point register set. This is the default setting.

`-mmultiple`
: Generates code that uses 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.

`-mn-multiple`
: Generates code that 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.

`-mstring`
: Generates code that uses 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 in little endian mode.

`-mno-string`
: Generate code that 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.

`-mupdate`
: Generate code that uses the load or store instructions that update the base register to the address of the calculated memory location. These instructions are generated by default.

`-mno-update`
: Generate code that 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; this means code that walks the stack frame across interrupts or signals may get corrupted data.

`-mfused-madd`
: Generates code that uses the floating-point multiply and accumulate instructions. These instructions are generated by default if hardware floating-point is used.

`-mno-fused-madd`
: Generates code that does not use the floating-point multiply and accumulate instructions. These instructions are generated by default if hardware floating-point is used.

`-maix64`
: Enables the 64-bit PowerPC ABI and calling conventions (64-bit pointers, 64-bit long type, and the infrastructure needed to support them).

`-maix32`
: Enables the 32-bit PowerPC ABI and calling conventions. This is the default setting.

Preprocessor Symbols

The compiler supports the following preprocessor symbols:

`_IBMR2`
`_POWER`
`_AIX`

: Are always defined.

`_AIX32`

: Indicates 32-bit mode. Defined when `-maix32` is specified.

`_AIX64`

: Indicates 64-bit mode. Defined when `-maix64` is specified.

`_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 Application Binary Interface (ABI), which the tools adhere to by default.

Table 3 shows the size and alignment for all data types.

- Alignment within aggregates (structures and unions) is as shown, 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.

Table 3: Data type sizes and alignments for 32-bit ABI

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

Table 4 shows register usage.

Table 4: Register usage for 32-bit ABI

Register	***Usage***
`r0`	Volatile register used in function prologs
`r1`	Stack frame pointer
`r2`	TOC pointer
`r3` and `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)
`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` through `r10`. The floating-point registers available for argument passing are `fp3` through `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 eight 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 `*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 align 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 GOTO 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; see Table 5 for 32-bit ABI.

Table 5: Functions and value returned for 32-bit ABI

Type	***Register***
`int`	`r3`
`short`	`r3`
`long`	`r3`
`long long`	`r3` and `r4`
`float`	`fp1`
`double`	`fp1`
`structure` and `union`	*

* 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.

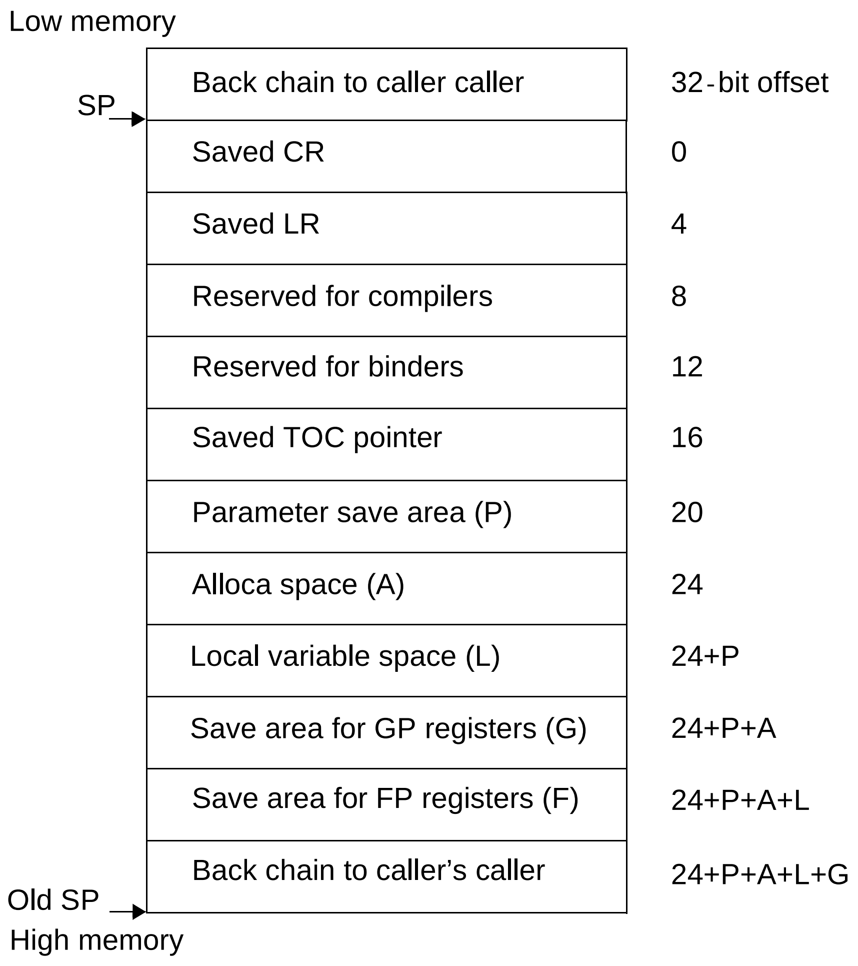
Stack Frames

This section describes 32-bit PowerPC stack frames:

- 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 (FP) need not be allocated.
- The stack pointer (SP) shall always be aligned to 16-byte boundaries.

See Figure 1 for AIX stack frames.

Figure 1: AIX stack frames for 32-bit ABI



64-Bit ABI Summary

This section describes the 64-bit AIX ABI.

Table 6 shows the size and alignment for all data types for 64-bit ABI.

- Alignment within aggregates (structures and unions) is as shown, 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.

Table 6: Data type sizes and alignments for 64-bit ABI

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

Table 7 shows how the registers are used for 64-bit ABI.

Table 7: Register usage for 64-bit ABI

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)
`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` through `r10`. The floating-point registers available for argument passing are `fp3` through `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 eight 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; 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 align 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 GOTO 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; see Table 8.

Table 8: Function return values by type for 64-bit ABI

Type	***Register***
`int`	`r3`
`short`	`r3`
`long`	`r3`
`long long`	`r3`
`float`	`fp1`
`double`	`fp1`
`structure` and `union`	*

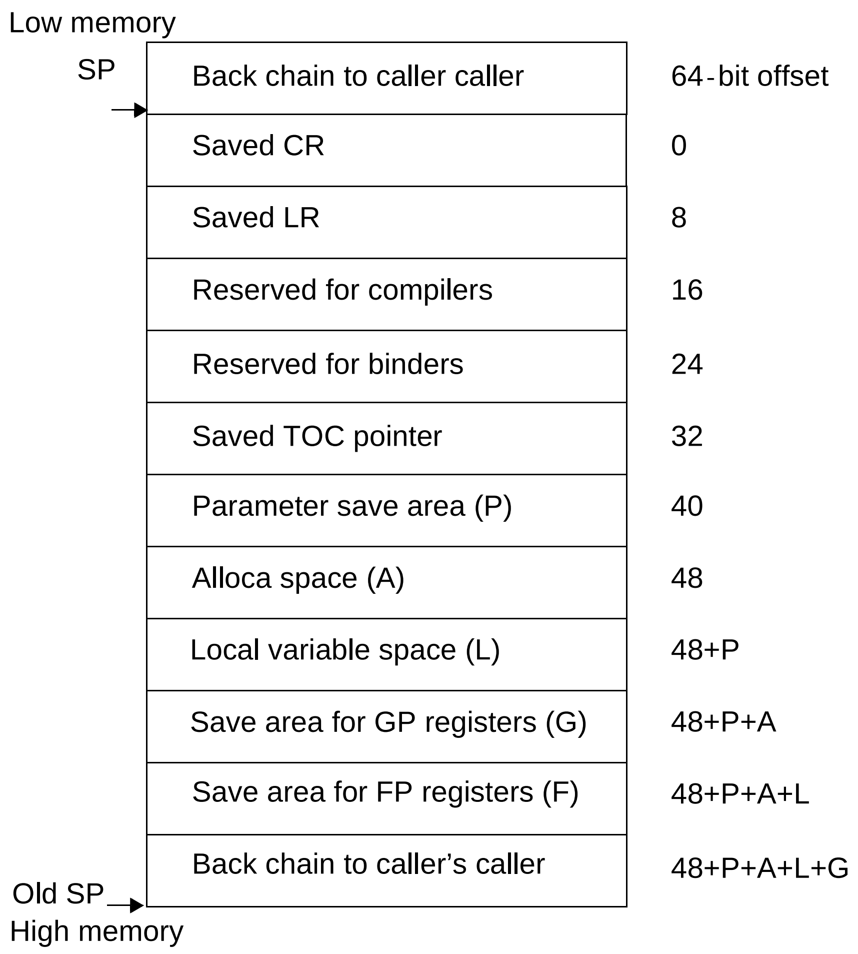
* The caller handles structures and unions by passing a pointer to space allocated to receive the return value. The pointer is passed as a hidden first argument.

Stack Frames

This section describes 64-bit PowerPC stack frames.

- 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 (FP) need not be allocated.
- The stack pointer (SP) shall always be aligned to 32-byte boundaries.

Figure 2: AIX stack frames for 64-bit ABI



Assembler Features

This section describes PowerPC-specific features of the GNUPro assembler. For a list of available generic assembler options, see “Command Line Options” in *Using as* in ***GNUPro Utilities***. 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*** (IBM, MPRPPCFPE-01; also Motorola, MPCFPE/AD)

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.

See Table 9 for symbols to use as aliases for individual registers.

Table 9: Aliases for registers

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

The GNU tools recognize the PowerPC’s special registers; see Table 10.

Table 10: Special registers

Symbol	***Register***
`lr`	Link register
`ctr`	Count register
`cr0` through `cr7`	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.

The initial character in all assembler directives is the dot (`.`). The directives are: `..mri` (this first directive starts with two dots), `.ABORT`, `.abort`, `.align`, `.appfile`, `.appline`, `.appline`, `.ascii`, `.asciz`, `.balign`, `.balignl`, `.balignw`, `.bb`, `.bc`, `.bf`, `.bi`, `.bs`, `.bss`, `.byte`, `.comm`, `.comm`, `.common`, `.common.s`, `.csect`, `.data`, `.data`, `.dc`, `.dc.b`, `.dc.d`, `.dc.l`, `.dc.s`, `.dc.w`, `.dc.x`, `.dcb`, `.dcb.b`, `.dcb.d`, `.dcb.l`, `.dcb.s`, `.dcb.w`, `.dcb.x`, `.debug`, `.def`, `.dim`, `.double`, `.ds`, `.ds.b`, `.ds.d`, `.ds.l`, `.ds.p`, `.ds.s`, `.ds.w`, `.ds.x`, `.eb`, `.ec`, `.ef`, `.ei`, `.eject`, `.else`, `.elsec`, `.elseif`,

`.end`, `.endc`, `.endef`, `.endfunc`, `.endif`, `.equ`, `.equiv`, `.err`, `.es`, `.exitm`, `.extern`, `.extern`, `.fail`, `.file`, `.fill`, `.float`, `.format`, `.func`, `.function`, `.global`, `.globl`, `.hword`, `.ident`, `.if`, `.ifc`, `.ifdef`, `.ifeq`, `.ifeqs`, `.ifge`, `.ifgt`, `.ifle`, `.iflt`, `.ifnc`, `.ifndef`, `.ifne`, `.ifnes`, `.ifnotdef`, `.include`, `.int`, `.irep`, `.irepc`, `.irp`, `.irpc`, `.lcomm`, `.lcomm`, `.lflags`, `.lglobl`, `.line`, `.linkonce`, `.list`, `.llen`, `.ln`, `.loc`, `.long`, `.long`, `.lsym`, `.macro`, `.mexit`, `.mri`, `.name`, `.noformat`, `.nolist`, `.nopage`, `.octa`, `.offset`, `.optim`, `.org`, `.p2align`, `.p2alignl`, `.p2alignw`, `.page`, `.plen`, `.print`, `.psize`, `.purgem`, `.quad`, `.rename`, `.rep`, `.rept`, `.rva`, `.sbttl`, `.scl`, `.sect`, `.sect.s`, `.section`, `.section.s`, `.set`, `.short`, , `.single`, `.size`, `.skip`, `.sleb128`, `.space`, `.spc`, `.stabd`, `.stabn`, `.stabs`, `.stabx`, `.string`, `.struct`, `.tag`, `.tc`, `.text`, `.text`, `.this_GCC_requires_the_GNU_assembler`, `.this_gcc_requires_the_gnu_assembler`, `.title`, `.toc`, `.ttl`, `.type`, `.uleb128`, `.val`, `.vbyte`, `.version`, `.weak`, `.word`, `.xcom`, `.xdef`, `.xref`, `.xstabs`, and `.zero`. See “Assembler Directives” in *Using* `as` in ***GNUPro Utilities*** for a description of what these directives do (`http://www.redhat.com/support/manuals/gnupro.html`).

Linker Features

The GNU linker, `ld`, resolves code addresses and debug symbols, links the startup code and additional libraries to the binary code, and produces an executable binary image. `ld` attempts to emulated the native AIX linker, although there are differences.

The default output format is `xcoff32`, which can also be explicitly set by using the `-b32` command line option or by setting the environmental variable, `LDEMULATION=aixppc`. An optional output format is `xcoff64`, explicitly set by using the `-b64` command line option or by setting the environmental variable, `LDEMULATION=aixppc64`. The compiler utility, `collect2`, handles the mapping of `-maix32` and `-maix64` to the correct output format.

To support AIX 4.2 and C++ constructors and destructors, a special linker option to the compiler, `-binitfini`, has special handling in the linker backend. The native linker uses the arguments to generate a table of `init` and `fini` functions for the executable. The function table is accessed by the runtime linker/loader by checking if the first symbol in the loader symbol table is `__rtinit`. The native linker generates this table and the loader symbol. `ld` looks for the `__rtinit` symbol and makes it the first loader symbol. So it is your responsibility to define the `__rtinit` symbol. The format for `__rtinit` is given by the AIX system file, `/usr/include/rtinit.h`.

Example 1 shows a 32 bit assembly file that defines `__rtinit`. `collect2` handles emitting the `__rtinit` symbol when the `-binitfin` option is given to GCC.

Example 1: Assembly file defining the `__rtinit` symbol

```
      .file     "my_rtinit.s"

      .csect .data[RW],3
      .globl __rtinit
      .extern init_function
      .extern fini_function

__rtinit:
      .long 0
      .long f1i - __rtinit
      .long f1f - __rtinit
      .long f2i - f1i
      .align 3
f1i:  .long init_function
      .long s1i - __rtinit
      .long 0
f2i:  .long 0
      .long 0
      .long 0
f1f:  .long fini_function
      .long s1f - __rtinit
      .long 0
f2f:  .long 0
      .long 0
      .long 0
      .align 3
s1i:  .string "init_function"
      .align 3
s1f:  .string "fini_function"
```

The following AIX linker options are not supported: `-f`, `-S`, `-v`, `-Z`, `-bbindcmds`, `-bbinder`, `-bbindopts`, `-bcalls`, `-bcaps`, `-bcror15`, `-bdebugopt`, `-bdbg`, `-bdelcsect`, `-bex?`, `-bfilelist`, `-bfl`, `-bgcbypass`, `-bglink`, `-binsert`, `-bi`, `-bloadmap`, `-bl`, `-bmap`, `-bnl`, `-bnobind`, `-bnocomprld`, `-bnocrld`, `-bnoerrmsg`, `-bnoglink`, `-bnoloadmap`, `-bnl`, `-bnoobjreorder`, `-bnoquiet`, `-bnoreorder`, `-bnotypchk`, `-bnox`, `-bquiet`, `-bR`, `-brename`, `-breorder`, `-btypchk`, `-bx`, `-bX`, and `-bxref`.

For generic GNU linker options, see “Command Language” in *Using* `ld` in ***GNUPro Utilities*** (see `http://www.redhat.com/support/manuals/gnupro.html`).

The GNU linker uses a script to determine how to process each section in an object file, and how to lay out the executable. The linker script is a declarative program consisting of a number of directives. For instance, the `ENTRY()` directive specifies the symbol in the executable that will be the executable’s entry point. Example 2 shows a typical linker script.

Example 2: Linker script

```
ENTRY(__start)
SECTIONS
{
  .pad 0 : { *(.pad) }
  . = 0x10000000;
  .text   : {
    PROVIDE (_text = .);
    *(.text)
    *(.pr)
    *(.ro)
    *(.db)
    *(.gl)
    *(.xo)
    *(.ti)
    *(.tb)
    PROVIDE (_etext = .);
  }
  . = 0x20000000;
  .data . : {
    PROVIDE (_data = .);
    *(.data)
    *(.rw)
    *(.sv)
    *(.sv64)
    *(.sv3264)
    *(.ua)
    . = ALIGN(4);
    CONSTRUCTORS
    *(.ds)
    *(.tc0)
    *(.tc)
    *(.td)
    PROVIDE (_edata = .);
  }
  .bss : {
    *(.tocbss)
    *(.bss)
    *(.bs)
    *(.uc)
    *(COMMON)
    PROVIDE (_end = .);
    PROVIDE (end = .);
  }
  .loader : {
    *(.loader)
  }
  .debug : {
    *(.debug)
  }
}
```

Debugger Features

There are no PowerPC-specific debugger command line options. For a complete description of the GNUPro debugger, see *Debugging with GDB* in ***GNUPro Debugging Tools***.

Index