Chapter 4

N32 Examples and Case Studies

This chapter provides examples and case studies of programs that have been converted from o32 to n32. Each step in the conversion is presented and examined in detail. Examples include:

- "An Example Application"
- "Building and Running the o32 Application"
- "Porting Issues"
- "Building and Running the N32 Application"
- "Building Multiple Versions of the Application"

An Example Application

An examination of the following application, *app1*, illustrates the steps necessary to port from o32 to n32. As you can see, *app1* is trivial in functionality, but it is constructed to point out several of the issues involved in converting code from o32 to n32.

App1 contains the following files:

- *main.c*, which contains the function *main()*.
- *foo.c,* which contains *foo()* a varargs function.
- *gp.s*, which contains the assembly language leaf routine, *get_gp()*. This function returns the value of the global pointer register (*$gp*).
- *regs.s*, which contains the assembly language function **regs()**. This function is linked separately into its own DSO. The function **regs()** returns the value of *$gp*, the return address register (*$ra*), and the stack pointer (*$sp*). This function also makes calls to the *libc* routines **malloc()** and **free()** as well as calculating the sum of two double precision values passed to it as arguments and returns the sum through a pointer that is also passed to it as an argument.

Figure 4-1 shows a call tree is for the *app1* program. It illustrates that **main()** calls *get_gp()*, *foo()* and *printf()*. The function **foo()** calls **regs()** and **printf()**, while **regs()** calls **malloc()** and **free()**. The figure also shows that **app1** is linked against two shared objects, *libc.so* and *regs.so*.

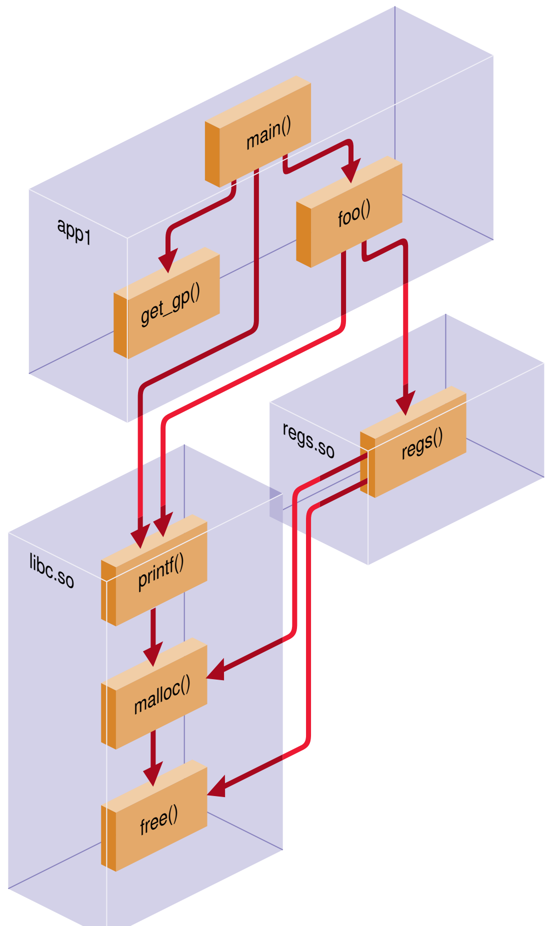

Figure 4-1 Call Tree for App1

The source code for the original versions of *main.c*, *foo.c*, *gp.s*. and *regs.s* are shown below.

```
/* main.c */
extern void foo();
```

```
main()
{
   unsigned gp,ra,sp, get_regs();
   double d1 = 1.0;
   double d2 = 2.0;
   double res;

   gp = get_gp();
   printf("gp is 0x%x\n", gp);
   foo(7, 3.14, &gp, &ra,
       &sp, d1, &d2, &res);

}

/* foo.c */

#include <stdarg.h>

void foo(int narg, ...)
{
  va_list ap;
  double d1;
  double daddr1, *daddr2, *resaddr;
  unsigned *gp, *ra, *sp;

  va_start(ap, narg);
  printf("Number of Arguments is: %d\n",narg);

  d1 = va_arg(ap, double);
  printf("%e\n",d1);

  gp = va_arg(ap, unsigned*);
  ra = va_arg(ap, unsigned*);
  sp = va_arg(ap, unsigned*);

  daddr1 = va_arg(ap, double);
  daddr2 = va_arg(ap, double*);
  resaddr = va_arg(ap, double*);

  printf("first  double precision argument is %e\n",daddr1);
  printf("second double precision argument is %e\n",*daddr2);

  regs(gp, ra, sp, daddr1, daddr2, resaddr);
```

```
  printf("Back from assembly routine\n");
  printf("gp is 0x%x\n",*gp);
  printf("ra is 0x%x\n",*ra);
  printf("sp is 0x%x\n",*sp);
  printf("result of double precision add is %e\n",*resaddr);

  va_end(ap);
}

/* gp.s */
#include <regdef.h>
#include <asm.h>

LEAF(get_gp)
        move  v0, gp
        j     ra
        .end  get_gp

/* regs.s */
#include <regdef.h>

        .text
        .globlregs          # make regs external
        .entregs 2
regs:
        .set noreorder
        .cploadt9           # setup gp
        .set reorder
        subu   sp, 32       # create stack frame
        sw     ra, 28(sp)   # save return address
        .cprestore     24   # for caller saved gp
                            # save gp 24(sp)
        sw     gp, 0(a0)    # return gp in first arg
        sw     ra, 0(a1)    # return ra in second arg
        sw     sp, 0(a2)    # return sp in third arg

        li     a0, 1000     # call libc routines
        jal    malloc       # for illustrative purposes
        move   a0, v0       # to make regs
        jal    free         # a nested function

        lw     t0, 56(sp)   # get fifth argument from stack
        lwc1   $f4, 4(t0)   # load it in fp register
        lwc1   $f5, 0(t0)   # fp values are stored in LE
                            # format
```

```
    lwc1   $f6, 52(sp)  # get fourth argument from stack
    lwc1   $f7, 48(sp)  # fp values are stored in LE
                        # format

    add.d  $f8, $f4, $f6  # do the calculation
    lw     t0, 60(sp)     # get the sixth argument
                          # from the stack
    swc1   $f8, 4(t0)     # save the result
    swc1   $f9, 0(t0)     # fp values are stored in LE

lw     ra, 28(sp)     # get return address
addu   sp, 32         # pop stack
j      ra             # return to caller
.end regs
```

Building and Running the o32 Application

The commands used to build *app1* are shown below. As mentioned previously, *regs.s* is compiled and linked separately into its own DSO, while *main.c*, *foo.c* and *gp.s* are compiled and linked together.

```
%cc -32 -O -shared -o regs.so regs.s

%cc -32 -O -o app1 main.c foo.c gp2.s regs.so
```

In order to run the application, the *LD_LIBRARY_PATH* environment variable must be set to the directory where regs.so resides.

```
%setenv LD_LIBRARY_PATH .
```

Running the application produces the following results. Note that the value of *$gp* is different when code is executing in the *regs.so* DSO.

```
%app1

   gp is 0x100090f0

   Number of Arguments is: 7

   3.140000e+00

   first  double precision argument is 1.000000e+00

   second double precision argument is 2.000000e+00

   Back from assembly routine

   gp is 0x5fff8ff0
```

```
ra is 0x400d10
sp is 0x7fff2e28
result of double precision add is 3.000000e+00
```

Porting Issues

If the files *foo.c* and *main.c* were recompiled for n32, the resulting executable would not work for a variety of reasons. Each reason is examined below and a solution is given. The resulting set of files will work when compiled either for o32 or for n32. This section covers:

- "Varargs Routines"
- "Assembly Language Issues"

Varargs Routines

Attempting to recompile *main.c foo.c* **–n32** results in two sets of warnings shown below:

```
%cc -n32 -O -o app1 main.c foo.c gp2.s
    foo.c
    !!! Warning (user routine 'foo'):
    !!! Prototype required when passing floating point parameter to
    varargs routine: printf
    !!! Use '#include <stdio.h>' (see ANSI X3.159-1989, Section 3.3.2.2)
    ld32: WARNING 110: floating-point parameters exist in the call for
    "foo", a VARARG function, in object "main.o" without a prototype --
    would result in invalid result.  Definition can be found in object
    "foo.o"
    ld32: WARNING 110: floating-point parameters exist in the call for
    "printf", a VARARG function, in object "foo.o" without a prototype
    -- would result in invalid result.  Definition can be found in
    object "/usr/lib32/mips4/libc.so"
```

The first warning points out that **printf()** is a varargs routine that is being called with floating point arguments. Under these circumstances, a prototype must exist for **printf()**. This is accomplished by adding the following line to the top of *foo.c*:

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

The second warning points out that *foo()* is also a varargs routine with floating point arguments and must also be prototyped. This is fixed by changing the declaration of *foo()* in *main.c* to:

```
foo(int, ...)
```

For completeness, <*stdio.h*> is also included in *main.c* to provide a prototype for *printf()* should it ever use floating point arguments.

As a result of these small changes, the C files are fixed and ready to be compiled **–n32**. The new versions are shown below.

```
/* main.c */
#include <stdio.h>
extern void foo(int, ...);

main()
{
   unsigned gp,ra,sp, get_regs();
   double d1 = 1.0;
   double d2 = 2.0;
   double res;

   gp = get_gp();
   printf("gp is 0x%x\n", gp);

   foo(7, 3.14, &gp, &ra,
       &sp, d1, &d2, &res);

}


/* foo.c */
#include <stdio.h>
#include <stdarg.h>

void foo(int narg, ...)
{

  va_list ap;
  double d1;
  double daddr1, *daddr2, *resaddr;
  unsigned *gp, *ra, *sp;

  va_start(ap, narg);
```

```
  printf(“Number of Arguments is: %d\n”,narg);

  d1 = va_arg(ap, double);
  printf(“%e\n”,d1);

  gp = va_arg(ap, unsigned*);
  ra = va_arg(ap, unsigned*);
  sp = va_arg(ap, unsigned*);

  daddr1 = va_arg(ap, double);
  daddr2 = va_arg(ap, double*);
  resaddr = va_arg(ap, double*);

  printf(“first  double precision argument is %e\n”,daddr1);
  printf(“second double precision argument is %e\n”,*daddr2);

  regs(gp, ra, sp, daddr1, daddr2, resaddr);
  printf(“Back from assembly routine\n”);
  printf(“gp is 0x%x\n”,*gp);
  printf(“ra is 0x%x\n”,*ra);
  printf(“sp is 0x%x\n”,*sp);
  printf(“result of double precision add is %e\n”,*resaddr);

  va_end(ap);
}
```

Assembly Language Issues

Since **get_gp()** is a leaf routine that is linked in the same DSO where it is called, no changes are required to port it to n32. However, you have to recompile it.

Since **get_gp()** is a leaf routine that is linked in the same DSO where it is called, no changes are required to port it to n32. However, you have to recompile it.

On the other hand, **regs()** requires a lot of work. The issues that need to be addressed are detailed below.

gp register

As explained throughout this book, the o32 ABI follows the convention that *$gp* (global pointer register) is caller saved. This means that the global pointer is saved before each function call and restored after each function call returns. This is accomplished by using the *.cpload* and *.cprestore* assembler pseudo instructions respectively. Both lines are present in the original version of *regs.s*.

The n32 ABI, on the other hand, follows the convention that *$gp* is callee saved. This means that *$gp* is saved at the beginning of each routine and restored right before that routine itself returns. This is accomplished through the use of *.cpsetup*, an assembler pseudo instruction.

The recommended way to deal with these various pseudo instructions is to use the macros provided in <*sys/asm.h*>. The macros below will provide correct use of these pseudo instructions whether compiled for o32 or for n32.

- SETUP_GP expands to the *.cpload t9* pseudo instruction for o32. For n32 it is null.
- SAVE_GP(GPOFF) expands to the *.cprestore* pseudo instruction for o32. For n32 it is null.
- SETUP_GP64(GPOFF, regs) expands to the *.cpsetup* pseudo instruction for n32. For o32 it is null.

Register Size

Under o32, registers are 32 bits wide. Under n32, they are 64 bits wide. As a result, assembly language routines must be careful in the way they operate on registers. The following macros defined in <*sys/asm.h*> are useful because they expand to 32-bit instructions under o32 and to 64-bit instructions under n32.

- REG_S expands to *sw* for o32 and to *sd* for n32.
- REG_L expands to *lw* for o32 and to *ld* for n32.
- PTR_SUBU expands to *subu* for o32 and to *dsubu* for n32.
- PTR_ADDU expands to *addu* for o32 and to *daddu* for n32.

Argument Passing

The **get_regs()** function in *regs.s* is called with six arguments. Under o32, the first three are passed in registers *a0* through *a2*. The fourth argument (a double precision parameter) is passed at offset 16 relative to the stack. The fifth and sixth arguments are passed at offsets 24 and 28 relative to the stack, respectively. Under n32, however, all of the arguments are passed in registers. The first three arguments are passed in registers *a0* through *a2* as they were under o32. The next parameter is passed in register *$f15*. The last two parameters are passed in registers *a4* and *a5* respectively. Table 4-1 summarizes where each of the arguments are passed under the two conventions.

Table 4-1 Argument Passing

Argument	o32	n32
argument1	a0	a0
argument2	a1	a1
argument3	a2	a2
argument4	$sp+16	$f15
argument5	$sp+24	a4
argument6	$sp+28	a5

Note: Under o32, there are no *a4* and *a5* registers, but under n32 they must be saved on the stack because they are used after calls to an external function.

The code fragment that illustrates accessing the arguments under n32 is shown below:

```
        mov.d   $f4,$f15            # 5th argument in 5th fp
                                    # arg. register
        l.d     $f6,0(a4)           # fourth argument in
                                    # fourth arg. register
        s.d     $f8,0(a5)           # save in 6th arg. reg
```

Extra Floating point Registers

As explained in Chapter 3, “N32 Compatibility, Porting, and Assembly Language Programming Issues,” floating point registers are 64 bits wide under n32. They are no longer accessed as pairs of single precision registers for double precision calculations. As a result, the section of code that uses the pairs of *lwc1* or *swc1* instructions must be changed. The simplest way to accomplish this is to use the *l.d* assembly language instruction. This instruction expands to two *lwc1* instructions under **-mips1**; under **-mips2** and above, it expands to the *ldc1* instruction.

Putting it together

The new version of *regs.s* is shown below. It is coded so that it will compile and execute for either o32 or n32 environments.

```
/* regs.s */
#include <sys/regdef.h>
#include <sys/asm.h>

.text

LOCALSZ=5            # save ra, a4, a5, gp, $f15

FRAMESZ= (((NARGSAVE+LOCALSZ)*SZREG)+ALSZ)&ALMASK

RAOFF=FRAMESZ-(1*SZREG)      # stack offset where ra is saved
A4OFF=FRAMESZ-(2*SZREG)      # stack offset where a4 is saved
A5OFF=FRAMESZ-(3*SZREG)      # stack offset where a5 is saved
GPOFF=FRAMESZ-(4*SZREG)      # stack offset where gp is saved
FPOFF=FRAMESZ-(5*SZREG)      # stack offset where $f15 is
                             # saved
                             # a4, a5, and $f15 don't have to
                             # be saved, but no harm done in
                             # doing so
NESTED(regs, FRAMESZ, ra)
                             # define regs to be a nested
                             # function
    SETUP_GP                 # used for caller saved gp
    PTR_SUBU sp,FRAMESZ      # setup stack frame
    SETUP_GP64(GPOFF, regs)  # used for callee saved gp
    SAVE_GP(GPOFF)           # used for caller saved gp

    REG_S   ra, RAOFF(sp)    # save ra on stack

#if (_MIPS_SIM != _MIPS_SIM_ABI32)
                             # not needed for o32
    REG_S   a4, A4OFF(sp)    # save a4 on stack (argument 4)
    REG_S   a5, A5OFF(sp)    # save a5 on stack (argument 5)
    s.d     $f15,FPOFF(sp)   # save $f15 on stack (argument 6)
#endif /* _MIPS_SIM != _MIPS_SIM_ABI32 */

    sw      gp, 0(a0)        # return gp in first arg
    sw      ra, 0(a1)        # return ra in second arg
    sw      sp, 0(a2)        # return sp in third arg

    li      a0, 1000         # call malloc
    jal     malloc           # for illustration purposes only

    move    a0, v0           # call free
    jal     free             # go into libc.so twice
```

```
                                # this is why a4, a5, $f15
                                # had to be saved

#if (_MIPS_SIM != _MIPS_SIM_ABI32)
                                # not needed for o32
        l.d     $f15,FPOFF(sp)  # restore $f15 (argument #6)
        REG_L   a4, A4OFF(sp)   # restore a4   (argument #4)
        REG_L   a5, A5OFF(sp)   # restore a5   (argument #5)
#endif /* _MIPS_SIM != _MIPS_SIM_ABI32 */

#if (_MIPS_SIM == _MIPS_SIM_ABI32)
                                # for o32 arguments will
                                # need to be pulled from the
                                # stack
        lw      t0,FRAMESZ+24(sp) # fifth argument is 24
                                  # relative to original sp
        l.d     $f4,0(t0)         # use l.d for correct code
                                  # on both mips1 & mips2
        l.d     $f6,FRAMESZ+16(sp)  # fourth argument is 16
                                    # relative to original sp
        add.d   $f8, $f4, $f6       # do the calculation
        lw      t0,FRAMESZ+28(sp)   # sixth argument is 28
                                    # relative to original sp
        s.d     $f8,0(t0)           # save the result there
#else
                                    # n32 args are in regs
        mov.d   $f4,$f15            # 5th argument in 5th fp
                                    # arg. register
        l.d     $f6,0(a4)           # fourth argument in
                                    # fourth arg. register
        add.d   $f8, $f4, $f6       # do the calculation
        s.d     $f8,0(a5)           # save in 6th arg. reg
#endif /* _MIPS_SIM != _MIPS_SIM_ABI32 */

        REG_L   ra, RAOFF(sp)     # restore return address
        RESTORE_GP64              # restore gp for n32
                                  # (callee saved)
        PTR_ADDU sp,FRAMESZ       # pop stack
        j        ra               # return to caller
.endregs
```

Building and Running the N32 Application

The commands for building an n32 version of *app1* are shown below. The only difference is in the use of the **–n32** argument on the compiler command line. If *app1* was a large application using many libraries, the command line or makefile would possibly need to be modified to refer to the correct library paths. In the case of *app1* the correct *libc.so* is automatically used as a result of the **–n32** argument.

```
%cc -n32 -O -shared -o regs.so regs.s

%cc -n32 -O -o app1 main.c foo.c gp.s regs.so
```

In order to run the application, the *LD_LIBRARY_PATH* environment variable must again be set to the directory where *regs.so* resides.

```
%setenv LD_LIBRARY_PATH .
```

Running the application produces the following results. Note that the values of some of the returned registers are different from those returned by the o32 version of *app1*.

```
%app1
   gp is 0x100090e8
   Number of Arguments is: 7
   3.140000e+00
   first  double precision argument is 1.000000e+00
   second double precision argument is 2.000000e+00
   Back from assembly routine
   gp is 0x5fff8ff0
   ra is 0x10000d68
   sp is 0x7fff2e30
   result of double precision add is 3.000000e+00
```

Building Multiple Versions of the Application

Following the procedure above generates new n32 versions of *app1* and *regs.so*; however, they overwrite the old o32 versions. To build multiple versions of *app1*, use one of the following methods:

- Use different names for the n32 and o32 versions of the application and DSO. This method is simple, but for large applications, you must rename each DSO.
- Create separate directories for the o32 and n32 applications and DSOs, respectively. Modify the commands above or makefiles to create *app1* and *reg.so* in the appropriate directory. This method offers more organization than the approach above, but you must set the *LD_LIBRARY_PATH* accordingly.
- Create separate directories as specified above, but add the **–rpath** argument to the command line that builds *app1*.