Note 227: 64-Bit Compiler Considerations

Wes Young and Athol Kemball

March 09, 2000

Contents

1 Introduction

1.1 Motivation

To take advantage of large memory machines available at the NCSA we need to move beyond 32-bit pointers because they limit the amount of memory to 1 Gb. Additionally, with a 64bit Linux and the IA64 Intel processor series on the horizon, it behooves us to be fully "64-bit" clean sooner rather than later. AIPS++ already has working 64-bit builds on the Dec Alpha and SGI Origin series using architecture-specific #ifdef macros (e.g. __alpha or _sgi*) to handle the critical 64-bit differences. All 64-bit differences have not been addressed throughout the library, however, and we need to define coding rules and conventions for this purpose. They may be implemented incrementally however.

1.2 Background

Here's a table lifted from the "SGI MIPSpro 64-bit Porting and Transition Guide". It shows how long each of the types are in bits. Note that longs and pointers have different sizes depending on compiling using 64- or 32-bit modes.

C type	32-bit	64-bit
char	8	8
short int	16	16
int	32	32
long int	32	64
long long int	64	64
pointer	32	64
float	32	32
double	64	64
long double	64	128

2 Porting Considerations

2.1 Common Problems

We have identified several 64-bit porting problems in the AIPS++ code tree. This list below includes these problems and other potential problems reported by 64-bit porting guides from some of the major vendors.

1. Assuming the equivalence of Long and Int
2. Assuming pointers and Ints are equivalent
3. Use of "C style" formating functions
4. Explicit constant typing

5. Bit shifting and bit-wise operations
6. Library return types, parameters and implementation differences
7. Memory alignment assumptions (eg. structs)
8. C++/FORTRAN interface
9. Persistent storage of Int and Long types
10. Long Double and Long Long types

Compilation on the SGI using CC -64 (64 bit compilation) finds a handful of problems of type 2 and 3 ($N \sim 10$). The most prevalent problem ($N \sim 250$) is assuming the equivalence of int and long usually found with returns from standard library functions.

2.2 Examples

This section contains examples of the problem categories listed above and recommendations for coding conventions to address each type.

2.2.1 Library Return Types

```
cc-1506 CC: REMARK File = /home/argus/aips++/daily/code/include/aips/Tables/BucketFile.h,
Line = 168
  There is an implicit conversion from "long" to "unsigned int"; rounding, sign
          extension, or loss of accuracy may result.

      { return ::lseek (fd_p, 0, SEEK_END); }

   lseek returns an off_t not a uInt
               ^
blockio (opt)
cc-1506 CC: REMARK File = /home/argus/aips++/daily/code/aips/implement/FITS/blockio.cc, Line = 117
  There is an implicit conversion from "long" to "int"; rounding, sign
          extension, or loss of accuracy may result.

            nreadlast = ::read(fd, buffer + iosize, (unsigned)ntoread);
                        ^
   read returns an ssize_t not an Int
```

```
cc-1506 CC: REMARK File = /home/argus/aips++/daily/code/aips/implement/FITS/blockio.cc, Line = 152
  There is an implicit conversion from "long" to "int"; rounding, sign
          extension, or loss of accuracy may result.

                iosize = ::write(fd,buffer,current);

   write returns an ssize_t not an Int
                         ^
cc-1506 CC: REMARK File = /home/argus/aips++/daily/code/aips/implement/FITS/fits.cc, Line = 2068
  There is an implicit conversion from "unsigned long" to "int"; rounding, sign
          extension, or loss of accuracy may result.

                vallen = strlen((char *)val);
                         ^

   strlen returns a size_t not an int
```

The following two lists document the list of library and system calls that return off_t, size_t (unsigned), and ssize_t (signed).

Library routines that return off_t, size_t or ssize_t:

- strftime
- strcspn
- strlen
- strspn
- strxfrm
- strlen
- strspn
- strcspn
- wcstombs
- telldir

System calls that return off_t, size_t or ssize_t:

- getpagesize
- read
- write
- lseek

Comments These examples point out a need to check the return types of various standard function calls and not assume they are Int/uInt. These function calls should be embedded in specialized classes where possible. However, a new AIPS++ data type Size is proposed, defined as the largest Int encompassing (off_t, size_t and ssize_t), for use in interfaces which require the specification of file offsets, sizes in bytes or related types. An unsigned counterpart, uSize, will also be added.

2.2.2 Equivalence of pointers and ints

```
/opt/MIPSpro/bin/cc -I.. -I../../include -I../.. -I../../../include -g -64 -DSGI64 -KPIC -c ../../sv.c
cc-3187 cc: WARNING File = ../../sv.c, Line = 575
  A pointer is converted to a smaller integer.

        iv      = (IV)pv;
                  ^

cc-3187 cc: WARNING File = ../../sv.c, Line = 935
  A pointer is converted to a smaller integer.

          return (IV)SvRV(sv);
                 ^
```

Comments Inter-conversion of ints and pointer types, or pointer arithmetic using ints is not recommended. However, there are rare cases where it may be necessary, such as the transfer of addresses from C++ to Glish. It is proposed that the new type Size be used in this case to deal with (and flag) these very limited cases. In addition, in C code, (char*)NULL must be used in preference to (char*)0, where required. Some 64-bit compilers will not accept the latter.

2.2.3 String conversion problems

```
cc-1178 CC: WARNING File = ../../ValKern.cc, Line = 68
  Argument is incompatible with the corresponding format string conversion.

                fprintf( stdout, " + %d {ValueKernel}\n", sizeof(ValueKernel) );
                                                          ^
```

Comments %d is for Int, sizeof returns a Long in 64bit and needs %ld. The recommended solution to this problem is to use iostreams.

2.2.4 Explicit Constant Typing

Comments There are many instances of explicit constants used in AIPS++, in general we get this correctly using only Int constants for Ints, i.e. we don't do

```
Long a = 0xffff;  //Assumes a long is 4 bytes.
```

Special constants that are sensitive to word length should be defined in OS/Bitmask.h or Constants.h. Some of these can be obtained directly from math limits, and should be used where possible.

2.2.5 Bit shifting and bit-wise operations

Comments We generally do the right thing. Random.cc has some 32/64-bit dependencies of this type. It's important to keep the word length in mind and take care when using the shift operators << and >> or other bitwise operators, and to use standard C++ bit addressing schemes. Where word lengths of a fixed size are required, such as real-time interfaces or data fillers, {u}Int{8,16,32,64,128} types could be introduced.

2.2.6 FORTRAN Interface

Again directly from "SGI MIPSpro 64-bit Porting and Transition Guide".

```
64-Bit Fortran Porting Guidelines

This section describes which sections of your Fortran source code you
need to modify to port to a 64-bit system.

Standard ANSI Fortran 77 code should have no problems, but the
```

following areas need attention:

. Code that uses REAL*16 could get different runtime results due to additional accuracy in the QUAD libraries.

. fcom and fef77 are incompatible with regard to real constant folding.

. Integer variables which were used to hold addresses need to be changed to INTEGER*8.

. C interface issues (Fortran passes by reference).

. %LOC returns 64-bit addresses.

. %VAL passes 64-bit values.

Source code modifications are best done in phases. In the first phase, try to locate all of the variables affected by the size issues mentioned above. In the second phase, locate variables that depend on the variables changed in the first phase. Depending on your code, you may need to iterate on this phase as you track down long sets of dependencies.

Examples of Fortran Portability Issues

The following examples illustrate the variable size issues outlined above:

Example 1: Changing Integer Variables

Integer variables used to hold addresses must be changed to INTEGER*8.

32-bit code:

```
integer iptr, asize
iptr = malloc(asize)
```

64-bit code:

```
integer*8 iptr, asize
iptr = malloc(asize)

Example 2: Enlarging Tables

Tables which hold integers used as pointers must be enlarged by a
factor of two.

32-bit code:

integer tableptr, asize, numptrs
numptrs = 100
asize = 100 * 4
tableptr = malloc(asize)

64-bit code:

integer numptrs
integer*8 tableptr, asize
numptrs = 100
asize = 100 * 8
tableptr = malloc(asize)
```

2.2.7 Memory alignment assumptions

Comments These include assumptions on the layout in memory of data in structs or unions. No known problems at present.

2.2.8 Library function parameters and implementation

Comments This includes differences in function parameters, names or implementations between 32- and 64-bit version of a library. Examples include the signature of memcpy(), dirent() and dirent64() on the SGI and the absence of the macro definition SCM_RIGHTS (which is not defined using the SGI 64-bit compiler).

2.2.9 Persistent storage of Long and Int types

Comments The system currently assumes that Longs are stored as 8 bytes. The IO classes should check on read whether there is an overflow problem when reading into a local long type.

2.2.10 Long Double and Long Long types

Comments It is recommended that the use of these types be forbidden at this time.

2.2.11 Architecture independence

Comments An ifdef macro AIPS_64BIT is proposed to isolate 64-bit dependencies in an architecture-independent manner. Dealing with these differences purely in run-time may be considered later.

3 Immediate impact

1. Addition of types Size and uSize to aips.h.
2. Retrofit use of AIPS_64BIT for current SGI and Dec Alpha 64-bit ifdef macros.
3. Change coding guidelines to include recommendations in Section 2.2.
4. Array indices $> (2^{32}-1)$ and related memory access issues are deferred for later consideration.