This chapter contains two sections:

- “Alignment, Size, and Value Ranges” describes how the Fortran compiler implements size and value ranges for various data types as well as how data alignment occurs under normal conditions.
- “Access of Misaligned Data” describes two methods of accessing misaligned data.

### Alignment, Size, and Value Ranges

Table 2-1 contains information about various Fortran scalar data types. (For details on the maximum sizes of arrays, see “Maximum Memory Allocations” on page 14.)

<table>
<thead>
<tr>
<th>Type</th>
<th>Synonym</th>
<th>Size</th>
<th>Alignment</th>
<th>Value Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>BYTE</td>
<td>INTEGER*1</td>
<td>8 bits</td>
<td>Byte</td>
<td>–128…127</td>
</tr>
<tr>
<td>INTEGER*2</td>
<td></td>
<td>16 bits</td>
<td>Half word(a)</td>
<td>–32,768…32,767</td>
</tr>
<tr>
<td>INTEGER</td>
<td>INTEGER*4(b)</td>
<td>32 bits</td>
<td>Word(c)</td>
<td>(-2^{31}…2^{31}-1)</td>
</tr>
<tr>
<td>INTEGER*8</td>
<td></td>
<td>64 bits</td>
<td>Double word</td>
<td>(-2^{63}…2^{63}-1)</td>
</tr>
<tr>
<td>LOGICAL*1</td>
<td></td>
<td>8 bits</td>
<td>Byte</td>
<td>0…1</td>
</tr>
<tr>
<td>LOGICAL*2</td>
<td></td>
<td>16 bits</td>
<td>Half word(a)</td>
<td>0…1</td>
</tr>
<tr>
<td>LOGICAL</td>
<td>LOGICAL*4(d)</td>
<td>32 bits</td>
<td>Word(c)</td>
<td>0…1</td>
</tr>
<tr>
<td>LOGICAL*8</td>
<td></td>
<td>64 bits</td>
<td>Double word</td>
<td>0…1</td>
</tr>
<tr>
<td>REAL</td>
<td>REAL*4(e)</td>
<td>32 bits</td>
<td>Word(c)</td>
<td>See Table 2-2</td>
</tr>
<tr>
<td>DOUBLE PRECISION</td>
<td>REAL*8(f)</td>
<td>64 bits</td>
<td>Double word(g)</td>
<td>See Table 2-2</td>
</tr>
</tbody>
</table>
Chapter 2: Storage Mapping

The following notes provide details on some of the items in Table 2-1.

- Table 2-2 lists the approximate valid ranges for REAL*4 and REAL*8.

### Table 2-2 Valid Ranges for REAL*4 and REAL*8 Data Types

<table>
<thead>
<tr>
<th>Range</th>
<th>REAL*4</th>
<th>REAL*8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum</td>
<td>(3.40282356 \times 10^{38})</td>
<td>(1.7976931348623158 \times 10^{308})</td>
</tr>
<tr>
<td>Minimum normalized</td>
<td>(1.17549424 \times 10^{-38})</td>
<td>(2.2250738585072012 \times 10^{-308})</td>
</tr>
<tr>
<td>Minimum denormalized</td>
<td>(1.40129846 \times 10^{-46})</td>
<td>(1.1125369292536006 \times 10^{-308})</td>
</tr>
</tbody>
</table>

- **REAL*16** constants have the same form as **DOUBLE PRECISION** constants, except the exponent indicator is **Q** instead of **D**. Table 2-3 lists the approximate valid range...

---

**Table 2-1 (continued) Size, Alignment, and Value Ranges of Data Types**

<table>
<thead>
<tr>
<th>Type</th>
<th>Synonym</th>
<th>Size</th>
<th>Alignment</th>
<th>Value Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>REAL*16</td>
<td></td>
<td>128 bits</td>
<td>Double word</td>
<td>See Table 2-3</td>
</tr>
<tr>
<td>COMPLEX</td>
<td>COMPLEX*8(^h)</td>
<td>64 bits</td>
<td>Double word(^d)</td>
<td>See the fourth bullet item below</td>
</tr>
<tr>
<td>DOUBLE COMPLEX</td>
<td>COMPLEX*16(^i)</td>
<td>128 bits</td>
<td>Double word(^g)</td>
<td>See the fourth bullet item below</td>
</tr>
<tr>
<td>COMPLEX*32</td>
<td></td>
<td>256 bits</td>
<td>Double word</td>
<td>See the fourth bullet item below</td>
</tr>
<tr>
<td>CHARACTER</td>
<td></td>
<td>8 bits</td>
<td>Byte</td>
<td>-128.127</td>
</tr>
</tbody>
</table>

a. Byte boundary divisible by two.
b. When the -i2 option is used, type INTEGER is equivalent to INTEGER*2; when the -i8 option is used, INTEGER is equivalent to INTEGER*8.
c. Byte boundary divisible by four.
d. When the -i2 option is used, type LOGICAL is equivalent to LOGICAL*2; when the -i8 option is used, type LOGICAL is equivalent to LOGICAL*8.
e. When the -i8 option is used, type REAL is equivalent to REAL*8.
f. When the -d16 option is used, type DOUBLE PRECISION is equivalent to REAL*16.
g. Byte boundary divisible by eight.
h. When the -i8 option is used, type COMPLEX is equivalent to COMPLEX*16.
i. When the -d16 option is used, type DOUBLE COMPLEX is equivalent to COMPLEX*32.
Alignment, Size, and Value Ranges

for REAL*16. REAL*16 values have an 11-bit exponent and a 107-bit mantissa; they are represented internally as the sum or difference of two doubles. So, for REAL*16 "normal" means that both high and low parts are normals.

<table>
<thead>
<tr>
<th>Range</th>
<th>Precise Exception Mode w/FS Bit Clear</th>
<th>Fast Mode or Precise Exception Mode w/FS Bit Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum</td>
<td>1.797693134862315807937289714053023* 10^{308}</td>
<td>1.797693134862315807937289714053023* 10^{308}</td>
</tr>
<tr>
<td>Minimum normalized</td>
<td>2.0041683600089730005034939020703004* 10^{-292}</td>
<td>2.0041683600089730005034939020703004* 10^{-292}</td>
</tr>
<tr>
<td>Minimum denormalized</td>
<td>4.940656458412465441765687928682214* 10^{-324}</td>
<td>2.225073858507201383090232717332404* 10^{-308}</td>
</tr>
</tbody>
</table>

- Table 2-1 states that REAL*8 (that is, DOUBLE PRECISION) variables always align on a double-word boundary. However, Fortran permits these variables to align on a word boundary if a COMMON statement or equivalencing requires it.

- Forcing INTEGER, LOGICAL, REAL, and COMPLEX variables to align on a halfword boundary is not allowed, except as permitted by the –align8, –align16, and –align32 command line options. See Chapter 1, “Compiling, Linking, and Running Programs.”

- A COMPLEX data item is an ordered pair of REAL*4 numbers; a DOUBLE COMPLEX data item is an ordered pair of REAL*8 numbers; a COMPLEX*32 data item is an ordered pair of REAL*16 numbers. In each case, the first number represents the real part and the second represents the imaginary part. Therefore, refer to Table 2-2 and Table 2-3 for valid ranges.

- LOGICAL data items denote only the logical values TRUE and FALSE (written as .TRUE. or .FALSE.). However, to provide VMS compatibility, LOGICAL variables can be assigned all integral values of the same size.

- You must explicitly declare an array in a DIMENSION declaration or in a data type declaration. To support DIMENSION, the compiler:
  - allows up to seven dimensions
  - assigns a default of 1 to the lower bound if a lower bound is not explicitly declared in the DIMENSION statement
  - creates an array the size of its element type times the number of elements
  - stores arrays in column-major mode
• The following rules apply to shared blocks of data set up by COMMON statements:
  – The compiler assigns data items in the same sequence as they appear in the
    common statements defining the block. Data items are padded according to the
    alignment compiler options or the compiler defaults. See “Access of Misaligned
    Data” on page 24 for more information.
  – You can allocate both character and noncharacter data in the same common
    block.
  – When a common block appears in multiple program units, the compiler
    allocates the same size for that block in each unit, even though the size required
    may differ (due to varying element names, types, and ordering sequences) from
    unit to unit. The size allocated corresponds to the maximum size required by
    the block among all the program units except when a common block is defined
    by using DATA statements, which initialize one or more of the common block
    variables. In this case the common block is allocated the same size as when it is
    defined.

Access of Misaligned Data

The Fortran compiler allows misalignment of data if specified by special options.

As discussed in the previous section, the architecture of the IRIS 4D™ series assumes a
particular alignment of data. ANSI standard Fortran 77 cannot violate the rules
governing this alignment. Many opportunities for misalignment can arise when using
common extensions to the dialect. This is particularly true for small integer types, which
• allow intermixing of character and non-character data in COMMON and
  EQUIVALENCE statements
• allow mismatching the types of formal and actual parameters across a subroutine
  interface
• provide many opportunities for misalignment to occur

Code using the extensions that compiled and executed correctly on other systems with
less stringent alignment requirements may fail during compilation or execution on the
IRIS 4D. This section describes a set of options to the Fortran compilation system that
allow the compilation and execution of programs whose data may be misaligned. Be
forewarned that the execution of programs that use these options is significantly slower
than the execution of a program with aligned data.
This section describes the two methods that can be used to create an executable object file that accesses misaligned data.

**Accessing Small Amounts of Misaligned Data**

Use the first method if the number of instances of misaligned data access is small or to provide information on the occurrence of such accesses so that misalignment problems can be corrected at the source level.

This method catches and corrects bus errors due to misaligned accesses. This ties the extent of program degradation to the frequency of these accesses. This method also includes capabilities for producing a report of these accesses to enable their correction.

To use this method, keep the Fortran front end from padding data to force alignment by compiling your program with one of two options to `f77`.

- Use the `-align8` option if your program expects no restrictions on alignment.
- Use the `-align16` option if your program expects to be run on a machine that requires half-word alignment.

You must also use the misalignment trap handler. This requires minor source code changes to initialize the handler and the addition of the handler binary to the link step (see the `fixade(3f)` reference page).

**Accessing Misaligned Data Without Modifying Source**

Use the second method for programs with widespread misalignment or whose source may not be modified.

In this method, a set of special instructions is substituted by the IRIS 4D assembler for data accesses whose alignment cannot be guaranteed. The generation of these more forgiving instructions may be opted for each source file independently.

You can invoke this method by specifying one of the alignment options (`-align8`, `-align16`) to `f77` when compiling any source file that references misaligned data (see the `f77(1)` reference page). If your program passes misaligned data to system libraries, you may also have to link it with the trap handler. See the `fixade(3f)` reference page for more information.