

**AFRL-IF-RS-TM-2003-2**  
**In-House Technical Memorandum**  
**February 2003**



## **COMMON PITFALLS IN F77 CODE CONVERSION**

Walter A. Koziarz

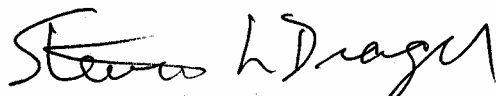
***APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.***

**AIR FORCE RESEARCH LABORATORY**  
**INFORMATION DIRECTORATE**  
**ROME RESEARCH SITE**  
**ROME, NEW YORK**

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

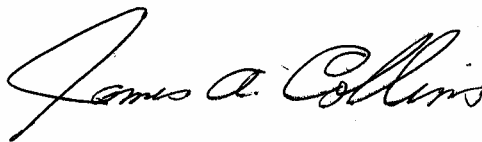
AFRL-IF-RS-TM-2003-2 has been reviewed and is approved for publication.

APPROVED:



STEVEN L. DRAGER, Technical Advisor  
Advanced Computing Architecture Branch  
Information Technology Division  
Information Directorate

FOR THE DIRECTOR:



JAMES A. COLLINS, Acting Chief  
Information Technology Division  
Information Directorate

<b>REPORT DOCUMENTATION PAGE</b>			<i>Form Approved</i> <i>OMB No. 074-0188</i>	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing this collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503				
<b>1. AGENCY USE ONLY (Leave blank)</b>		<b>2. REPORT DATE</b> FEBRUARY 2003	<b>3. REPORT TYPE AND DATES COVERED</b> In-House Technical Memo, June 2001 – June 2002	
<b>4. TITLE AND SUBTITLE</b>  COMMON PITFALLS IN F77 CODE CONVERSION			<b>5. FUNDING NUMBERS</b>  PE - 637550 PR - 558T TA - PR WU - OJ	
<b>6. AUTHOR(S)</b>  Walter A. Koziarz				
<b>7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)</b>  AFRL/IFTC 26 ELECTRONIC PKY ROME, NY 13441-4514			<b>8. PERFORMING ORGANIZATION REPORT NUMBER</b>  AFRL-IF-RS-TM-2003-2	
<b>9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)</b>  AFRL/IFTC 26 ELECTRONIC PKY ROME, NY 13441-4514			<b>10. SPONSORING / MONITORING AGENCY REPORT NUMBER</b>  AFRL-IF-RS-TM-2003-2	
<b>11. SUPPLEMENTARY NOTES</b>  AFRL Project Engineer: Walter A. Koziarz/IFTC/315-330-2536				
<b>12a. DISTRIBUTION / AVAILABILITY STATEMENT</b>  APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.				<b>12b. DISTRIBUTION CODE</b>
<b>13. ABSTRACT (Maximum 200 Words)</b>  Differences between recognized standardization, i.e. ANSI standards and common usage 'de-facto' standards, occasionally lead to incompatibilities and inconsistencies. These often impede efforts to compile and execute supposedly standard software in a different hardware and software environment than that in which it was originally created/targeted. One such instance is described herein as a service to persons who may one day be called upon to perform similar code porting. The code addressed in this example performs a statistical analysis of the background clutter content of imagery data. This analysis can be used to pre-select appropriate digital filtering algorithms to de-emphasize the predominant background and other clutter data. The original input data is RADAR imagery, but the statistical analyses are applicable to sources such as acoustical and video data as well. Code examples are included as appropriate to illustrate changes required.				
<b>14. SUBJECT TERMS</b> software code conversion, code porting				<b>15. NUMBER OF PAGES</b> 27
				<b>16. PRICE CODE</b>
<b>17. SECURITY CLASSIFICATION OF REPORT</b>  UNCLASSIFIED	<b>18. SECURITY CLASSIFICATION OF THIS PAGE</b>  UNCLASSIFIED	<b>19. SECURITY CLASSIFICATION OF ABSTRACT</b>  UNCLASSIFIED	<b>20. LIMITATION OF ABSTRACT</b>  UL	

## Table of Contents

<b>List of Example Code Fragments</b>	<b>ii</b>
<b>Acknowledgements</b>	<b>iv</b>
<b>Summary</b>	<b>1</b>
<b>Typographic Conventions</b>	<b>1</b>
<b>Introduction and Background</b>	<b>2</b>
<b>Methods, Procedures, and High-level Discussion</b>	<b>2</b>
<b>Absence of structured data support</b>	<b>3</b>
<b>Default ‘MXUNIT’ of 99</b>	<b>4</b>
<b>Absence of Data Translation and Variable-         Expressions in FORMAT Statements</b>	<b>4</b>
<b>Conclusions</b>	<b>5</b>
<b>Reference</b>	<b>6</b>
<b>Appendices</b>	<b>7</b>
<b>Structured Data Issues</b>	<b>7</b>
<b>Larger Input/Output Unit Specifier</b>	<b>12</b>
<b>Data Translation</b>	<b>13</b>
<b>Keeping Your Fortran Programs Portable</b>	<b>17</b>
<b>Is it “FORTRAN” or “Fortran”?</b>	<b>19</b>
<b>Symbols, Abbreviations, and Acronyms</b>	<b>20</b>

## List of Example Code Fragments

<b>Code Fragment 1 Extended Fortran 77</b> <b>Original extended Fortran 77</b> <b>record code construction</b>	<b>8</b>
<b>Code Fragment 2 ANSI Fortran 77</b> <b>Use of two arrays to duplicate functionality of</b> <b>the record in the code fragment Excerpt 1</b>	<b>9</b>
<b>Code Fragment 3</b> <b>Alternate form ANSI Fortran 77</b> <b>duplicates functionality of</b> <b>the statements in code fragment 2</b>	<b>9</b>
<b>Code Fragment 4</b> <b>Example showing usage of STRUCTURE to</b> <b>create data structure “foo”</b>	<b>10</b>
<b>Code Fragment 5</b> <b>Illustrating use of RECORD with the</b> <b>previously defined data structure “foo”</b>	<b>10</b>
<b>Code Fragment 6</b> <b>One possible alternate form implementing the</b> <b>equivalent variable definitions</b>	<b>11</b>
<b>Code Fragment 7</b> <b>Extended Fortran 77 illustrating use of ‘DECODE’</b> <b>within the legacy code</b>	<b>13</b>
<b>Code fragment 8A</b> <b>Example DECODE with fixed formatting</b>	<b>14</b>
<b>Code fragment 8B</b> <b>Example READ with fixed formatting</b> <b>identical functionality to fragment above</b>	<b>14</b>
<b>Code Fragment 9 ANSI Fortran 77</b> <b>Illustration of error checking utilized in ‘DECODE’</b> <b>work-around function (ensures <math>1 \leq \text{‘index’} \leq 20</math>)</b>	<b>15</b>

## **List of Example Code Fragments Continued**

<b>Code Fragment 10 ANSI Fortran 77 'computed goto'</b>	<b>15</b>
<b>Code Fragment 11 ANSI Fortran 77 Illustrates repetitive code structure utilized by the 'DECODE' replacement</b>	<b>16</b>

**Acknowledgements**

The author wishes to acknowledge the work of Mr. Keith Dittl, a summer student employee in The Advanced Computing Architectures Branch of the Air Force Research Laboratory. Mr. Dittl typed and debugged much of the ANSI Fortran 77 code required to replace the non-ANSI equivalents within the existing source code files. The author also wishes to acknowledge Mr. Zenon Pryk for his valuable assistance in the use and application of standard development tools in the LINUX environment.

## Summary

Differences between recognized standardization, i.e. ANSI standards and common usage ‘de-facto’ standards, occasionally lead to incompatibilities and inconsistencies. These often impede efforts to compile and execute supposedly standard software in a different hardware and software environment.

One such instance is described in detail herein as a service to persons who may one day be called upon to perform similar code porting. Three difficulties representative of those that may be encountered are analyzed in-depth to illustrate changes required. These were the absence in GNU Fortran 77 (used interchangeably with “g77” herein) of structured data support via **STRUCTURE** and **RECORD**, a default limit of 100 (0-99) input/output units, and retention of **DECODE** with variable format length. The code alterations ranged from simple editing of unit specifiers to implementation of specialized functionality provided in extended Fortran but absent in ANSI Fortran 77 or GNU Fortran 77.

The foundation of this code port effort is an in-house developed clutter characterization algorithm. Following its development, several additional individuals utilized and altered the original program, wrapping the algorithm with varying main program routines structured for individual requirements. Ultimately the algorithm was modified into a parallel processor hosted code. This program performs statistical analyses of the background clutter content of imagery data. The results can then be used to preselect appropriate digital filtering algorithms to de-emphasize the predominant background and other clutter data. The input data processed by the original program is RADAR imagery, but the statistical analyses can also work with additional sources such as acoustical and video data. The parallel processor code-variant was selected under the High Performance Computing to be re-hosted on a PC-based (Beowulf) cluster computer. This cluster has Linux as its operating system and the GNU software tools as its development environment. Ostensibly written in Fortran 77 and utilizing Message Passing Interface (MPI) for process communication, it appeared readily portable to any parallel processing system with Fortran 77 and MPI availability. This proved not to be the case and will be examined below as the significant work-arounds applied to this particular piece of code are described. The three examples presented below were selected for their impact on time consumed in identifying the source of an error reported by the compiler or logic flaw and developing an appropriate resolution. The criteria for selection of the illustrative examples was not complex issues, but instead the subtle and time-consuming. Some familiarity is assumed on the part of the reader with Fortran in general and GNU Fortran 77 in particular.

## Typographic conventions

Title of external documents: ***Bold Times New Roman Italic***

Quoted material from an external document: *Standard Times New Roman Italic*

Reference to identifiers, etc. within body text: *Standard Times New Roman Italic*

Fortran 77 keyword in body of text: **BOLD TIMES NEW ROMAN CAPITALS**

Example code fragments: `Courier New 10 point`

Fortran 77 keyword in code fragments: **Bold Courier New 10 point**



## **Introduction and Background**

Difficulties of varying severity are often encountered while porting source code from one host environment to another even when the code in question is authored in a language as well understood and as well established within the scientific community as Fortran 77. The case in point addresses a legacy piece of software, authored in non-ANSI-standard Fortran 77, that had previously undergone numerous modifications with varying degrees of in-line comments and little else for documentation. Combined with industry-standard, but not ANSI-standard, extensions to the Fortran 77 language used by interim programmers porting to a computing environment utilizing a strict ANSI Fortran 77 was time-consuming and difficult. This technical memo intends to provide insight from information gained through experience to facilitate future code porting efforts.

This memo derives directly from the project described above. The discussions and in-depth analyses are general so as not to tie too closely to a single application, but rather to provide a collection of generally applicable information. It is well to describe some common aspects of resolving standard versus non-standard software issues as background before delving deeply into the examples. The first step is to attempt to compile the original source file(s). Successful compilation is unlikely, but not impossible. Be aware, however, that successful compilation does not indicate the resulting executable will be logically correct. An example of this situation is included herein. The error messages generated by the compiler provide clues to portions of code requiring editing, but do not necessarily precisely locate erroneous code. This is a subtle and very significant point. Some knowledge of the original and intended language semantics is necessary to successfully identify the true sources of compile-time errors. Be aware that compiler errors are often cascading, one actual syntax error resulting in the generation of numerous subsequent error messages. Always begin with the first error reported, identify the cause, edit as needed, recompile, and continue this pattern until an error free compile is achieved. Reference to the target language processor documentation is essential in determining the best method by which to troubleshoot these errors. Approaches will vary according to actual products and code in use, but the Free Software Foundation's documentation for GNU Fortran 77 includes examples for common work arounds necessary when converting extended Fortran 77 to GNU Fortran 77. This document is available on the World Wide Web and the URL is provided in the References section of this memo. Reiterating, address compiler errors one at a time beginning with the earliest error and attempt compilation following each edit session. The preceding material summarizes one viable generalized approach to resolution of incompatibilities encountered.

## **Methods, Procedures, and High Level Discussion**

The GNU g77 Fortran compiler did not compile the source code received for this porting effort without errors due to incorporation of non-ANSI Fortran 77 constructs as well as the absence in g77 of one feature of ANSI standard Fortran 77. Proceeding methodically through the sequence of first locating the identified line that is the target of the earliest-occurring error message reported by the compiler, identifying the purpose of the statement or function, researching the best method of providing identical processing with available language elements, testing the

replacement code, inserting the tested lines into the source file, and finally recompiling to ensure that error message is no longer reported. These steps can be generalized as follows:

- 1) Attempt compilation
- 2) Identify the portion of source code to which the error is related
- 3) Determine the purpose of the original statement(s)
- 4) Determine appropriate substitution of code to provide the required result
- 5) Test the substitute code and ensure that range testing or error mitigation is addressed
- 6) Incorporate the tested substitute code into the original source and repeat Step 1 as necessary

This sequence is similar to a reasonable approach to debugging any program; but it differs in that language elements absent in one implementation versus another are the source of these errors rather than typography. It is well to use the practice of commenting-out original source file lines rather than deleting them until the new code is deemed logically correct. Examination of the code constructs in the files comprising the as-received legacy source code and the alterations and work-arounds utilized will highlight this discussion. Excerpts from the *Using and Porting GNU Fortran (1)*, *Sun Microsystems Workshop Documentation: Fortran 77 Language Reference (2)*, and *Keeping Your Fortran Programs Portable (3)* documents will be interspersed as necessary to illustrate discussions. World Wide Web links to these documents are provided in the References. The next three sections will examine the incompatibilities that were most time consuming to rectify. These three sections are written for persons familiar and experienced with Fortran. Detailed and tutorial treatments of these issues, with coding examples, are included as appendices.

### **Absence of Structured Data Support**

This difficulty results from the absence in g77 of the **STRUCTURE**, and **RECORD** keywords found in later revisions of Fortran and many extended implementations of Fortran 77. The benefit afforded by these language elements is code readability, ease of definition, and clarity during subsequent use of the data structures. The GNU documentation (3) states the following: *“This set of extensions is quite a bit lower on the list of large, important things to add to g77, partly because it requires a great deal of work either upgrading or replacing libg2c.”* While this may be true, their absence does hamper code portability in those cases where a previous author had access to these language elements and chose to utilize them. A successful work-around involved placing the data originally in the record into a **DATA** statement, and using an implied-read **DO** loop construct to perform the assignment of values. Two such constructs were used as the original record consisted of two arrays.

The difference the programmer must be aware of when utilizing this work-around is the change to the subsequent usage of the resultant data structures. The extended Fortran example results in creation of a record structure consisting of one or more arrays. Elements of the array(s) are identified via a “record-name.array-name” convention with an appropriate index into the array. The ANSI Fortran approach is to declare the required number of arrays to mimic the original record and simply utilize the arrays in the normal manner. This does place an added burden on the programmer to track usage of these arrays without benefit of the former record name. Please refer to Appendix 1 for in-depth treatment of this material.

### Default 'MXUNIT' of 99

This is a subtle and counter-intuitive effect of performing a default installation of GNU g77 development tools under LINUX. It is reasonable to presume virtually all g77 development environments were prepared using the default settings during software installation and will exhibit this problem. The observed effect is the result of the default limit on input/output (I/O) unit specifier; a maximum value for I/O unit specifier of 99 exists in a standard, unedited, installation of GNU g77 development tools under LINUX. For those unfamiliar with Fortran, a unit-specifier is a numeric label assigned to one of an input device, an output device, or a file. Subsequent input or output actions rely on the numeric label to uniquely identify the file or device. One may edit a *libg2c* source-file macro named *MXUNIT*, in the file "*f/runtime/libI77/fio.h*" in the g77 source file directory structure to a value larger than the default of 100 (recall a maximum of 100 yields 0-99 as legal unit identifiers; 1000 should suffice in all reasonable instances) prior to installation of the compiler. This fact is not clearly evident in the portions of the installation instructions for the g77 development tools. The work-around chosen ensures all input/output unit specifiers are less than 100. Obviously, in the event a large number of I/O units were required, a reinstallation of the g77 compiler would be necessary. This issue was, in this example, solved once the origin of the cryptic error message: '*illegal unit number*' issued at run-time was determined. The solution selected was identification of I/O unit specifiers and replacing three-digit specifiers with two-digit specifiers and inclusion of comments at each edited location reflecting original specifier and replacement specifier. Please refer to Appendix 2 for additional detail.

### Absence of Data Translation and Variable Expressions in FORMAT Statements

This is the last of the purely-Fortran issues encountered. It is a two-part difficulty. The first part is that the g77 compiler does not include support for the **DECODE** statement. In this example **DECODE** simplified data extraction from an external initialization file used with the subject source code. Secondly, g77 does not provide support for variable expressions in **FORMAT** statements; this is included in the ANSI standard F77 language. Variable expressions in **FORMAT** statements allow field widths to be assigned as necessary at runtime. For example, **FORMAT(I<J>)** will provide I/O formatting for one integer with a number of digits specified by the value of "J", where "J" may be any integer-valued expression. The *Fortran 77 Language Reference* ([http://www.ictp.trieste.it/~manuals/programming/sun/fortran/f77rm/4\\_statements.doc.html](http://www.ictp.trieste.it/~manuals/programming/sun/fortran/f77rm/4_statements.doc.html) - 3560) states more formally: "*In general, any integer constant in a format can be replaced by an arbitrary expression enclosed in angle brackets.*" The *Using and Porting GNU Fortran* document offers the following insight: "*g77 doesn't support 'FORMAT(I<J>)' and the like. Supporting this requires a significant redesign or replacement of libg2c*" ([http://www.delorie.com/gnu/docs/g77/g77\\_618.html](http://www.delorie.com/gnu/docs/g77/g77_618.html)) and "*g77 doesn't support ENCODE or DECODE*" ([http://www.delorie.com/gnu/docs/g77/g77\\_625.html](http://www.delorie.com/gnu/docs/g77/g77_625.html)). Since the code to be ported incorporated **DECODE** statements, which further required a variable format expression, resolving this issue proved cumbersome. The solution described below is a reasonable, effective, and straightforward work-around.

The functionality of **DECODE** was required; but was complicated by the need to selectively apply **DECODE** to **DOUBLE PRECISION** or **INTEGER** or **CHARACTER** data which further required variable format lengths. A suitable parsing subroutine was available in the legacy code and returned the format length (as an integer) as well as the data (as **CHARACTER** data type with a number of characters equal to the format-length). Inspection of the legacy code provided data type information for each **DECODE** call. This solution made use of direct file input/output and a straightforward code architecture consisting of several **FORMAT** statements and an associated several **READ** statements. This was repeated for each of the required data types. Since these instructions were executed only once per run in the set-up portion of the code, little effort was applied to optimization. Please refer to Appendix 3 for greater detail and code examples.

## **Conclusions**

Legacy code conversion, while seeming simple to the uninitiated, can be very time-consuming. This is especially true when the starting point is code written for a specific platform, including a specific and extended (non-standard) language processor, and the resulting code must be platform-independent. Several examples illustrating the reduction of non-ANSI standard code to ANSI standard were examined. These portions of code were selected from a larger conversion task as these were time consuming to recognize and resolve. They do not represent exceptionally complex issues, but rather realistic situations. The motivation remains clear since a great body of scientific software written in any of a number of non-ANSI Fortran 77 dialects does exist. Therefore, it is desirable to edit portions of this legacy to benefit from the efforts of previous algorithm research and development on faster computing machinery.

## References

- (1) Online document -- ***Using and Porting GNU Fortran***

Reference material specific to GNU Fortran 77

([http://www.delorie.com/gnu/docs/g77/g77\\_toc.html](http://www.delorie.com/gnu/docs/g77/g77_toc.html))

- (2) Online document -- ***Sun Microsystems Workshop Documentation:  
Fortran 77 Language Reference***

Reference material specific to one of many defacto-standard Fortran compilers, representative of the extended language elements addressed here-in

(<http://www.ictp.trieste.it/~manuals/programming/sun/Fortran/f77rm/index.html>)

- (3) Web page document – ***Keeping Your Fortran Programs Portable***

Reference material with tips to aid portability of code

(<http://fusion.gat.com/docview/portable.html>)

- (4) Online document -- ***The frequently asked questions (FAQ) at the Fortran Company's web site***

(<http://www.fortran.com/fortran/FAQ/>)

## Appendix 1

### Structured Data Issues

The **STRUCTURE** and **RECORD** keywords available in many extended Fortran 77 implementations provide support for defining data structures in a convenient manner. The related data are accessed in a straightforward method and program readability as well as maintainability are improved. Two pieces of code immediately below are extracted from the original source code and the revised g77-compatible code. They are functionally similar, differing in method to access an arbitrary value within each data representation. The differences will be highlighted with additional code examples and reference material.

Presented below, as code excerpt 1, is the definition of the data structure named “*keywords*” and the declaration of the record named “*kywrđ*” which is of type “*keywords*”. The result of executing these statements was creation of a data structure comprised of two unique arrays which are accessible as “*kywrđ.klen(index)*” and “*kywrđ.name(index)*”. The value contained in the **INTEGER** variable “*ientries*” was initialized elsewhere in the code and equaled 28, as there were 28 entries in both the arrays. The resulting arrays were utilized during initialization to receive data from an external initialization file. These data subsequently were passed to a series of **DECODE** calls. More detail relating to **DECODE** can be found in Appendix 3. The ampersand (&) in the code segments below is a continuation character. Fortran historically placed a limit of 72 characters on a source code line. This limit arose from the 80 column cards onto which code was once punched. The first five columns were, and remain, reserved for labels. Column 6 is reserved for a continuation character; this is used in instances where the source code lines are too long and/or where the author wishes to improve code readability. Columns 7 through 72 were reserved for Fortran statements. Columns 73-80 held a sequence number intended to facilitate automated card re-ordering.

```

structure /keywords/
  integer klen(ientries) /7,7,9,5,8,3,6,5,5,9,2,2,6,6,5,
&                                7,6,6,4,5,4,6,5,4,4,7,12,7/
  character*12 name(ientries) /
&      'infile1      ',
&      'infile2      ',
&      'data_type    ',
&      'nsame        ',
&      'ave_tech     ',
&      'nid          ',
&      'njoint       ',
&      'nptsb        ',
&      'ntype        ',
&      'n_ref_est    ',
&      'al           ',
&      'bl           ',
&      'gamma1       ',
&      'gamma2       ',
&      'itype        ',
&      'npoints      ',
&      'nstart       ',
&      'answer       ',
&      'ifit         ',
&      'idfam        ',
&      'nans         ',
&      'nidans       ',
&      'idist        ',
&      'par3         ',
&      'par4         ',
&      'errfile      ',
&      'conf_ellipse',
&      'samples      '/
end structure
record /keywords/ kywrd

```

**Code Fragment 1 Extended Fortran 77**  
**Original extended Fortran 77**  
**record code construction**

The statements comprising code excerpt 2 provide the same utility, i.e. initialization of two arrays named (note subtle difference in naming from previous discussion) “*klen(index)*” and “*name(index)*”; the prefix (“*kywrd.*”) is absent. As above, the value assigned to “*ientries*” is 28. The **DATA** statement initializes the 28 elements of the *klen* array with the 28 integer values following the implied **DO** loop, (*klen(i)*, *i* = 1, *ientries*). The implied **DO** is an alternative form for the sequence of statements shown in code excerpt 2a. The resulting functionality is identical between the two examples.

```

C      comment - the arrays klen and name are declared elsewhere
C      as - INTEGER klen(ientries)
C      CHARACTER *12 name(ientries)
      DATA (klen(i), i=1, ientries)
& / 7,7,9,5,8,3,6,5,5,9,2,2,6,6,5,7,6,6,4,5,4,6,5,4,4,7,12,7 /
      DATA (name(i), i=1, ientries) /
&      'infile1      ', 'infile2      ', 'data_type      ',
&      'nsame        ', 'ave_tech    ', 'nid            ',
&      'njoint       ', 'nptsb      ', 'ntype          ',
&      'n_ref_est     ', 'al         ', 'bl            ',
&      'gamma1        ', 'gamma2      ', 'itype          ',
&      'npoints       ', 'nstart      ', 'answer         ',
&      'ifit          ', 'idfam       ', 'nans           ',
&      'nidans        ', 'idist       ', 'par3           ',
&      'par4          ', 'errfile     ', 'conf_ellipse',
&      'samples      ' /

```

### Code Fragment 2 ANSI Fortran 77 Use of two arrays to duplicate functionality of the record in the code fragment Excerpt 1

```

C      comment - alternative form without IMPLIED DO
C      comment - the arrays klen and name are declared elsewhere
C      as: INTEGER klen(ientries)
C      CHARACTER *12 name(ientries)
      DATA klen
& / 7,7,9,5,8,3,6,5,5,9,2,2,6,6,5,7,6,6,4,5,4,6,5,4,4,7,12,7 /
      DATA names
&      / 'infile1      ', 'infile2      ', 'data_type      ',
&      'nsame        ', 'ave_tech    ', 'nid            ',
&      'njoint       ', 'nptsb      ', 'ntype          ',
&      'n_ref_est     ', 'al         ', 'bl            ',
&      'gamma1        ', 'gamma2      ', 'itype          ',
&      'npoints       ', 'nstart      ', 'answer         ',
&      'ifit          ', 'idfam       ', 'nans           ',
&      'nidans        ', 'idist       ', 'par3           ',
&      'par4          ', 'errfile     ', 'conf_ellipse',
&      'samples      ' /

```

### Code Fragment 3 Alternate form ANSI Fortran 77 duplicates functionality of the statements in code fragment 2

The preceding lines of Fortran code represent the translations actually required and performed. They illustrate the real code substituted and verified. Following is a more generalized and formal discussion of the work around for similar situations. The ***Fortran 77 Language Reference (2)*** ([http://www.ictp.trieste.it/~manuals/programming/sun/fortran/f77rm/4\\_statements.doc.html](http://www.ictp.trieste.it/~manuals/programming/sun/fortran/f77rm/4_statements.doc.html) - 4672) provides the very concise description “A **STRUCTURE** statement defines a form for a record by specifying the name, type, size, and order of the fields that constitute the record. Optionally, it can specify



*the initial values.*” A structure, once defined, is a template for a record. The record is a generalization of a variable or an array; it differs from an array in that the fields of the record (as defined by a structure) can be of different data types while the elements of an array must all be the same data type. The example below illustrates this material.

```
C      Comment - use of STRUCTURE statement
      STRUCTURE /foo/
      INTEGER ifool
      INTEGER ifoo2  /9999/
      REAL    rfool
      REAL    rfoo2  /99.99/
      CHARACTER*16 cfool  /'SOMETHING'/
      END STRUCTURE
```

#### Code Fragment 4 Example showing usage of **STRUCTURE** to create data structure “foo”

The above defines a structure with 5 fields (*ifool*, *ifoo2*, *rfool*, *rfoo2*, and *cfool*). Three of the fields are initialized: *ifoo2* is initialized to contain an integer value of 9999, *rfoo2* is initialized to contain a real value of 99.99, and *cfool* is initialized to contain a character value of ‘SOMETHING’ (the quotes are not part of the value). The fields can be any valid Fortran data type: scalars, vectors, or arrays. Once the structure is defined as above, records may be defined as below.

```
C      Comment - use of RECORD statement
C      with previously-defined structure (foo)
      RECORD /foo/ bar1, bar2, lots(10)
```

#### Code Fragment 5 Illustrating use of **RECORD** with the previously defined data structure “foo”

Both of the variables, *bar1* and *bar2* are records which have the foo structure and *lots* is an array of 10 such records. Each such record has its *ifoo2* initially set to 9999, its *rfoo2* initially set to 99.99, and its *cfool* initially set to SOMETHING. Porting the data structures resulting from execution of the **RECORD** statement above to a strict ANSI F77 environment requires, obviously, one new variable of appropriate data type to correspond to each of the variables declared in the structure “foo” for each record created. The resulting increased number of independent variable identifiers can be cumbersome and care must be used in selecting names to ease future maintenance of the program. The code fragment below illustrates the necessary translation of the above. Notice this author’s combination of the existing variable name (*ifool*, for example) and the record name (*bar1*, for example) into *bar1ifool*, etcetera. Individuals

may select any naming pattern, but should, as a service to those who may subsequently edit the code use comments liberally to explain such variable names.

```

C      Comment - replacement code for the structure and record example
      INTEGER barlifool, barlifoo2 = 9999
      REAL barlrfool, barlrfoo2 = 99.99
      CHARACTER#16 barlcfool = 'SOMETHING'
C      Comment - replacement for record "bar1"
C
      INTEGER bar2ifool, bar2ifoo2 = 9999
      REAL bar2rfool, bar2rfoo2 = 99.99
      CHARACTER#16 bar2cfool = 'SOMETHING'
C      Comment - replacement for record "bar2"
C
      INTEGER lotsfix, lotsifool(10), lotsifoo2(10)
      REAL lotsrfool(10), lotsrfoo2(10)
      CHARACTER#16 lotscfool(10)
C      Comment - must assign values as present in the original structure
      DO 1 lotsfix = 1, 10
         lotsifoo2(lotsfix) = 9999
         lotsrfoo2(lotsfix) = 99.99
         lotscfool(lotsfix) = 'SOMETHING'
1      CONTINUE
C      Comment - replacement for record "lots"

```

**Code Fragment 6**  
**One possible alternate**  
**form implementing the**  
**equivalent variable definitions**

As mentioned above, the variable name selection can ease the burden of readability and maintainability. The above information and examples should prove to be sufficient to allow translation of these code types to ANSI Fortran 77. Further information can be found in reference (2).

## Appendix 2

### Larger Input/Output Unit Specifier

The following paragraph is excerpted from *Using and Porting GNU Fortran(1)*

([http://www.delorie.com/gnu/docs/g77/g77\\_529.html](http://www.delorie.com/gnu/docs/g77/g77_529.html)) and explains very well the edit required in the event code to be ported requires an I/O specifier greater than 99. Editing I/O unit specifiers within the source code is preferable to performing the edit described below. Note that various system configurations may place additional restriction on the maximum number of files that may be open simultaneously.

*“As distributed, whether as part of f2c or g77, libf2c accepts file unit numbers only in the range 0 through 99. For example, a statement such as ‘WRITE (UNIT=100)’ causes a run-time crash in libf2c, because the unit number, 100, is out of range. If you know that Fortran programs at your installation require the use of unit numbers higher than 99, you can change the value of the ‘MXUNIT’ macro, which represents the maximum unit number, to an appropriately higher value. To do this, edit the file ‘f/runtime/libI77/fio.h’ in your g77 source tree, changing the following line: **#define MXUNIT 100**. Change the line so that the value of ‘MXUNIT’ is defined to be at least one greater than the maximum unit number used by the Fortran programs on your system. (For example, a program that does ‘WRITE (UNIT=255)’ would require ‘MXUNIT’ set to at least 256 to avoid crashing.) Then build or rebuild g77 as appropriate. Note: Changing this macro has no effect on other limits your system might place on the number of files open at the same time. That is, the macro might allow a program to do ‘WRITE (UNIT=100)’, but the library and operating system underlying libf2c might disallow it if many other files have already been opened (via OPEN or implicitly via READ, WRITE, and so on). Information on how to increase these other limits should be found in your system's documentation.”*

No code examples are required or presented for this appendix.

## Appendix 3

### Data Translation

Data translation in this appendix refers to the inclusion in the legacy source code which provided motivation to write this tech memo the **DECODE** instruction. The frequently asked questions (FAQ) at the Fortran Company's web site (<http://www.fortran.com/fortran/FAQ/tech.html> - 3.3.2)

*“3.3.2) What are ENCODE and DECODE statements, and how are they translated to standard Fortran? How can I convert numbers to character strings (and vice-versa)?*

*ENCODE and DECODE are vendor extensions to Fortran (invented in the sixties, long before X3.9-1978 added internal I/O to the language) which are most often used to convert data between numeric and character representations. They may be viewed as formatted writes to (ENCODE) or reads from (DECODE) memory. The standard-conforming alternatives are internal write and internal read statements respectively.”*

Internal read statements provided the fix for this extension. There was another part to this problem. The legacy code utilized variable expressions within the **DECODE** statement. This is significant because GNU F77, as previously shown, does not support variable expressions within **FORMAT** statements. Code fragment 7 shows the use of **DECODE**. The parameters of the instruction *len2*, *field2* are an integer and a character representation of an integer value, respectively. The *'(i<len2>)'* is the variable expression within this statement and the complicating factor. This particular variable expression provides formatting for an integer with length, of course, determined by *len2*; character and floating point were also required with similar format length determination. This requirement for three data types was satisfied in the work around by writing three functions, one for each data type. Code fragments 8A and 8B show the substitution that can be used when variable expression formatting is not required.

```
decode (len2, '(i<len2>)' , field2) ntype_z
```

#### Code Fragment 7 Extended Fortran 77 illustrating use of 'DECODE' within the legacy code

Comparing Code Fragment 8A with Code Fragment 8B one sees the similarity of form. In both cases, *'chr1'* is the desired data represented as text contained within a character variable. The value held by *int1* in Code Fragment 8A is equal to 4, as 4 is the format length set by the *(i4)* in the parameter list of the **decode** call. Note in Code Fragment 8B the absence of *int1*. The result of executing either of these code fragments is assignment of the equivalent numeric value held as character data in *chr1* to the 4-digit integer variable *int2*. These examples are sufficient to allow replacement of fixed format length **decode** instructions. The program utilizes variable format lengths and requires the additional work to address this omission from GNU F77.

```
decode (int1, '(i4)' , chr1) int2
```

**Code fragment 8A**  
**Example DECODE**  
**with fixed formatting**

```
4 read (chr1, 4) int2  
   format (i4)
```

**Code fragment 8B**  
**Example READ**  
**with fixed formatting**  
**identical functionality**  
**to fragment above**

The above examples and discussion conclude the steps necessary to translate **decode** into **read** statements when the format length is not variable. Where the format length is variable, this author chose a significantly different approach. Examination of the original source file revealed presence of a subroutine to return the format length. Inspection further revealed that under no condition possible in the program would that length ever exceed twenty, nor would it ever be less than 1. In general, the range of variable format length can be determined apriori just as it was in the case-study example. Some method to check the range of the length variable should be provided and the arithmetic **IF** was selected. This conditional test takes the form **IF(expression) label1, label2, label3** where *expression* is any valid arithmetic expression, either integer or floating point, *label1*, *label2*, and *label3* are target statement labels and control is transferred to *label1* if *expression* evaluates to a result less than 0, to *label2* if *expression* evaluates to exactly 0, and to *label3* if *expression* evaluates to a result greater than 0. Code Fragment 9 is taken from the case study program and is a portion of the **decode** work around for integer value variables. A Fortran **FUNCTION** subprogram was, as is evident from the declaration of function *idecode* in the first line of the code fragment, chosen to implement the required processing. Inspection of this code reveals the hard-coded limit of 20 as the maximum format length, hence the declaration of *field* as **character** data type of length 20. Additionally, inspection of the conditional tests reveal *index* is being verified as  $1 \leq index \leq 20$ . This can be generalized somewhat, but for the requirement of the rest of the function, use of the limits determined through inspection of the program may as well be used. Code Fragment 10 is the first line in the normal processing flow following verification that the index is in a valid range.

```

    function idecode(field, index)
    character field*20
C   COMMENT - conditional test: 'is the expression in parentheses
C                               less than 0, equal to 0, or greater
C                               than 0'
C       control will be transferred to the appropriate labeled
C       statement: (expression)<0 - left-label (line number)
C                   (expression)=0 - center-label
C                   (expression)>0 - right-label
    if (index) 997, 997, 998
998  continue
    if (index-20) 999, 999, 997
999  continue
    [normal processing sequence]
    .
    return
997  continue
    [appropriate error handler]
    .
    return
end

```

**Code Fragment 9 ANSI Fortran 77**  
**Illustration of error checking**  
**utilized in 'DECODE' work-**  
**around function (ensures**  
 **$1 \leq \text{'index'} \leq 20$ )**

Persons familiar with C, Pascal, and other comparatively modern programming languages are familiar with the **CASE** and **SWITCH** conditional transfer structures. Fortran 77 has a similar facility in the computed **goto** statement. The variable *index* is presumed here to be the same variable which was tested in code fragment 9 and contains an integer value between 1 and 20. The result of executing this computed **goto** is transfer of control to statement label 1 if *index* = 1, to statement label 2 if *index* = 2, to statement label 3 if *index* = 3, and so on through *index* = 20. It is obvious the computed **goto** as shown in code fragment 10 may be extended as needed to accommodate the required range of possibilities. Note, however, there is no requirement for the statement labels to progress from 1 to n as was chosen for convenience here. The statement label to which processing will transfer is determined by position, i.e. an *index* value of 4 will transfer to the *fourth statement label in the list*.

```
goto (1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20) index
```

**Code Fragment 10 ANSI Fortran 77**  
**'computed goto'**

With an understanding of the computed **goto** described above, examination of the code fragment 11 below reveals the processing associated with each target of the **goto**. This repetitive construct was necessitated by g77 lacking variable format length. The code fragment below will,

assuming the omitted statements are supplied, provide the integer decode functionality required by the program. It is a simple matter to change format descriptor to *f* (for floating point) or to *a* (for character) while retaining the rest of the code. This work around required insertion of 100 lines of code to replace 3 lines of code.

```

1      read (field, 100) idecode
100    format(i1)
      return
2      read (field, 200) idecode
200    format(i2)
      return
3      read (field, 300) idecode
300    format(i3)
      return
4      read (field, 200) idecode
400    format(i4)
      return

[several lines omitted for brevity]

19     read (field, 1900) idecode
1900   format(i19)
      return
20     read (field, 2000) idecode
2000   format(i20)
      return

```

**Code Fragment 11 ANSI Fortran 77**  
**Illustrates repetitive code structure**  
**utilized by the ‘DECODE’ replacement**

The floating point code will require identical structure, but every instance of *idecode* must be replaced by *fdecode* (or other appropriate identifier), every *I* in the format field must be replaced with *f* followed by an appropriate floating point format designator. Floating point numeric formats are defined by the form ‘TotalDigitsPresent.DigitsToRightOfPoint’. For example, 100.25 is of format *f5.2*; 5 total digits with 2 to the right of the decimal point. Therefore, inspection of the data to be utilized will be necessary before implementing this fix. The creation of the text decoding function is simpler. One need declare the function as character type and change *idecode* as necessary to match new name and change the *I* in the **format** statements to *a* indicating alphabetic data.

## Appendix 4

### Keeping Your Fortran Programs Portable

The following excellent information is reproduced from the frequently asked questions at the Fortran Company's website (4).

*"Despite widespread efforts at standardizing programming languages and operating systems, software portability persists as one of the most time- and labor-consuming problems encountered when dealing with computers. The great majority of programming in the Fusion environment is done in Fortran, so at least conversion between languages is rarely a[sic] issue. On the other hand, there are so many dialects of Fortran, referencing so many libraries in so many operating systems on so many hardware platforms, that portability is still a major issue for us in the Fusion community.*

*The recent and ongoing conversion of the CRAY supercomputers at NERSC from the CTSS to the UNICOS operating system, though relatively painless from the users' point of view, brought up the question of software portability once again. What follows are some general programming guidelines that, if adhered to, will make life easier for future conversions - and there will always be future conversions.*

*There is no one "standard" Fortran, but Fortran-77 and, to a lesser degree, the new Fortran-90, come close. Writing strictly in Fortran-77, which is supported by nearly every compiler vendor, is a good first step in making your programs portable. Constructs like ENCODE/DECODE, NAMELIST, and packed Hollerith strings, though widely implemented, are not in fact part of the standard and should be avoided if possible. ENCODE/DECODE is particularly easy to avoid, as the equivalent functionality is available in the (standard) internal READ/WRITE. NAMELIST has no standard equivalent, but its implementation is so widespread that using it rarely causes problems.*

*On the other hand, packed Hollerith strings create problems not only because Hollerith is not supported in Fortran-77 (though most compilers have at least this extension), but also because the number of characters that can be packed into a word depends upon the machine architecture, often resulting in the need for significant recoding. (For example, VAXes hold four characters, while CRAYs hold eight.) Another complication arises when Hollerith text is stored in both integer and real variables - strange things can happen on some machines when one is assigned to the other. More generally, any code which is wordsize-dependent, involves bit manipulation (masking and shifting, for example), or depends upon the particular implementation of floating point arithmetic or storage, should be avoided or at least isolated.*

*Because most porting difficulties arise in the I/O, that is where coding should be kept most conservative. Using the full form of the formatted READ and WRITE statements:*

*READ (NIN, 10) iolist*

*WRITE (NOUT, 20) iolist*

*where NIN and NOUT are variables or parameters, is far preferable to using list-directed I/O, PRINT statements, and other shortcuts. And having the I/O unit numbers in a common block helps even more.*

*Finally, libraries are a major source of non-portability[sic], with graphics libraries one of the chief culprits. We in Fusion are standardizing on the NCAR Graphics package - new applications should use it whenever possible. Math libraries like NAG and IMSL are*



*indispensible[sic] but will cause serious problems if your codes depend heavily upon them and they are not available at the target site. And runtime library routines are usually operating system-specific, although most systems have a subset of these in common. If you embed runtime library calls in your programs you are almost guaranteeing that these sections will have to be reworked when the code is ported, unless you are willing to write your own replacements for the referenced routines.*

*Of course, not all code need be portable. If you are creating an application that is very closely tied to a particular architecture, such as the Screen Management Facility[sic] in VMS on the VAX, then your code will be so hopelessly non-portable that there is no point in writing it conservatively.*

*Most modern Fortran-77 compilers have extensions to the standard, but of course you are under no obligation to use them. These compilers usually will have a switch that can be turned on to test for Fortran-77 compliance. Warning messages are then issued when a non-complying statement is encountered. Most of us routinely use language extensions without even knowing it - running your code through the checker can be a rude awakening! Fortunately, most Fortran-77 manuals highlight the language extensions so you know what you are getting into.”*

## Appendix 5

### Is it “FORTRAN” or “Fortran”?

There was an effort to “standardize” on spelling of programming languages just after F77 became a standard. The rule: if you say the letters, it is all caps (APL, C); if you pronounce it as a word, it is not (Cobol, Fortran, Ada). See, for example the definitive article describing Fortran 77 in the Oct 1978 issue of the Communications of the ACM. Of course, there are those who still think it is not truly Fortran if not written with all caps.

## **Symbols, Abbreviations, and Acronyms**

ANSI --	American National Standards Institute
Fortran --	FORmula TRANslator
Fortran 77 --	A specific and standard version of Fortran
FSF --	Free Software Foundation
g77 --	GNU Fortran 77 compiler
gcc --	GNU C compiler
GNU --	A project of the FSF, it is not truly an acronym but, rather, is a palindrome for <u>G</u> NU is <u>N</u> ot <u>U</u> nix
MPI --	Message Passing Interface
$\leq$ --	is less-than or equal-to