Portability Issues and the GNX™ Version 3 C Optimizing Compiler

National Semiconductor
Application Note 601
Series 32000 Applications
March 1989

INTRODUCTION

This application note describes compiler implementation aspects which may differ between those of the GNX-Version 3 C Optimizing compiler and other compilers and which may affect code portability. Portability issues are recognized by the C standard as issues that may differ from one compiler implementation to another.

The GNX-Version 3 C Optimizing Compiler is one of a family of compatible optimizing compilers targeted to the Series 32000® architecture. The compiler fully implements the C Language as defined in *The C Programming Language* by B. Kernighan and D. Ritchie. The C Optimizing Compiler is also compatible with the UNIX® System V Compiler(pcc).

This Application Note contains three sections:

1.0 Implementation Aspects

2.0 Standard Calling Conventions

3.0 Undefined Behavior

1.0 IMPLEMENTATION ASPECTS

This section describes aspects of the implementation of the GNX-Version 3 C compiler of which one should be knowledgeable in order to write portable programs or to port programs written for compilation using other C compilers.

The topics addressed are:

1.1 Memory Representation of Data Types

1.2 External Linkage Considerations

1.3 Data Types and Conversions

1.4 Variable and Structure Memory Alignment

1.5 Functions that Return a Structure

1.6 Mixed-Language Programming

1.7 Order of Evaluation of Parameters

1.8 Order of Allocation of Memory

1.9 Register Variables

1.10 Floating-Point Arithmetic

1.1 MEMORY REPRESENTATION OF DATA TYPES

The representation of the various C types in this compiler are:

C Type	Series 32000 Data Type
int	32-Bit Double-Word
long	32-Bit Double-Word
short	16-Bit Word
char	8-Bit Byte
float	32-Bit Single-Precision Floating-Point
double	64-Bit Double-Precision Floating-Point

- The set of values stored in a **char** object is signed.
- The padding and alignment of members of structures as described in Section 1.4.
- A field of a structure can generally straddle storage unit boundaries.
- While signed bitfields are implemented, it is not recommended to use them since their implementation is slow. Bitfields are not allowed to straddle a double-word boundary.

1.2 EXTERNAL LINKAGE CONSIDERATIONS

- There is no limit to the number of characters in external names.
- Case distinctions are significant in an identifier with external linkage.

1.3 DATA TYPES AND CONVERSIONS

- A right shift of a signed integral type is arithmetic, i.e., the sign is maintained.
- When a negative floating-point number is converted to an integer, it is truncated to the nearest integer that is less than or equal to it in absolute value. The result is returned as a signed integer.
- When a double-precision entity is converted to a single-precision entity, it is converted to the nearest representation that will fit in a **float** with default rounding performed to the nearest value.
- The presence of a **float** operand in an operation not containing double-operands causes a conversion of the other operand to **float** and the use of single-precision arithmetic. If double-operands are present, conversion to double occurs.

1.4 VARIABLE AND STRUCTURE MEMORY ALIGNMENT

The alignment of entities in a program is a trade-off issue. Most Series 32000 CPUs are more efficient when dealing with entities aligned to a double-word boundary. This normally makes it necessary to have some amount of padding added to a program. This padding represents an overhead in storage space.

The GNX-Version 3 C compiler allows the user to tailor the alignment of structures/unions and their members and, independently, the alignment of other variables. Function parameters are always double-word aligned. This allows the calling of functions across modules without dealing with alignment issues.

1.4.1 Alignment of Variables

Extern, **static**, and **auto** variables are aligned in memory according to their size and the buswidth setting. Table I lists variable size, buswidth, and the alignment determined by these two parameters.

RRD-B30M75/Printed in U. S. A.

TABLE I. Variable Alignment

Bus Width	Variable Size (Bytes)		
	1	**2**	**≥4**
1	byte	byte	byte
2	byte	word	word
4	byte	word	double-word

Variables of size 1 are of the C type **char**, variables of size 2 are of the C type **short**, and variables of size 4 or greater are of the C types **int**, **long**, **float**, and **double** (size 8).

A buswidth setting of 1 means ''align to 1 byte''. Variables start on a byte boundary, in other words, there is no alignment and no padding. When allocating storage for variables, bytes are allocated sequentially with no padding between bytes.

A buswidth setting of 2 means ''align to an even byte.'' Variables that are larger than 1 byte start on a word boundary. This means that there may be padding of single bytes.

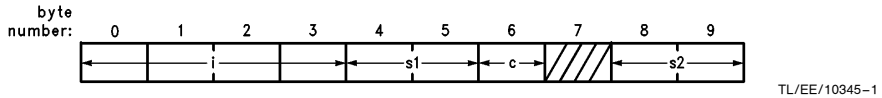
A buswidth setting of 4 means ''align to a double-word boundary'' (a byte whose address is divisible by four). Variables that are 2 bytes long start on a word boundary; variables that are 4 bytes or larger in size start on a double-word boundary. This means that there may be padding of up to three bytes.

Arrays are aligned as the alignment of their element type. Structures are aligned according to the alignment of the largest structure members. This is affected by the **-J (/ALIGN)** option. See ''Structure/Union Alignment'' and ''Allocation of Bit-Fields'' for more details.

Example: The arrangement of

int i; short s1; char c; short s2;

with a buswidth of 2 or 4 is



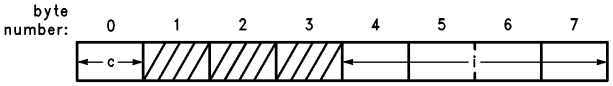
TL/EE/10345–1

Note that to align s2 to a word boundary, padding space of one byte is needed after **c**. This padding does not exist with a buswidth of 1.

Example: The arrangement of

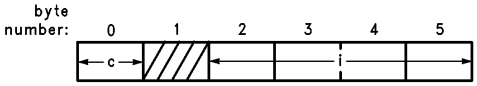
```
char c; int i;
```

with a buswidth of 4 is



TL/EE/10345–2

With a buswidth of 2, the arrangement is



TL/EE/10345–3

With a buswidth of 1, there is no padding.

It is important to note that the order in memory is the same as the declaration order only for **extern** and **static** variables. The optimizer may reorder **auto** variables in order to minimize padding space.

Fastest code is achieved by setting the default alignment to that of the data buswidth of the CPU (4 for all but the NS32008, the NS32CG16, and the NS32016). This can be accomplished by setting the BUS parameter in the target specification file, or by overwriting that file on the command line with the **-KB** (**/TARGET**) option.

1.4.2 Structure/Union Alignment

Structure members are aligned within the structure, relative to the beginning of the structure, in the same way that variables are aligned in memory. In order to maintain the alignment of the members relative to memory, the structure itself is aligned in memory according to the alignment of its largest members. This alignment may be controlled by putting **-J** (**/ALIGN**) on the command line.

In addition, the total size of a structure is such that it also ends on an alignment boundary of its largest member. This maintains the alignment of individual members in arrays of structures. This is illustrated in the **FILE struct** example at the end of this section.

For unions, there is no padding. The alignment of the union's largest members determine the alignment of the union itself.

1.4.3 Allocation of Bit-Fields

To understand the way bit-fields are handled, think of the situation where a field is fetched from memory. The number of bits fetched is determined by buswidth. For instance, if a bus is 2-bytes wide, then 2 bytes are fetched, even if only the first few bits are needed. For convenience, the number of bits fetched is called the ''fetching unit''.

Note that for the purpose of structure member alignment, the align switch value (1 byte, 2 bytes or 4 bytes) is taken as a ''virtual buswidth,'' even if it is different from the actual buswidth.

A complication exists when allocating bit-fields. The complication arises from the fact that different base types for bit-fields (**char short**, and **int**) are supported. The maximum length of a bit-field is the size of its base type; therefore, there may be times when a bit-field is larger than the buswidth. When the size of the base type is larger than the buswidth, the size of the fetching unit is considered to be the base-type size.

The precise rules for determining the start of the fetching unit are quite complicated. In general, it is determined by the current position in the allocation of structure members and by the base-type of the first bit-field in a group of consecutive bit-fields.

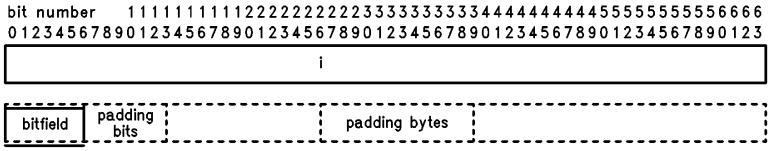
An attempt is made to pack consecutive bit-fields as much as possible, as long as the bit-fields remain in the same fetching unit. As soon as a field ''spills over'' into the next fetching unit, the alignment is set to the next memory unit (byte, word, or double-word, according to the align switch value and the base type of the field). A hole of padding bits remains, and the beginning of the spill-over field determines the start of a new fetching unit for following bit-fields. Using this method, bit-fields are packed as much as possible while still maintaining the alignment.

If, because of the bit-fields, the structure as a whole does not terminate on a byte boundary, padding bits are added to it to fill up to the end of the last byte it occupies. Additional padding bytes may be needed to fill to the alignment boundary of the largest structure member. This is seen in *Figure 1*. The bit-field does not quite reach the byte boundary; therefore, padding bits are added until the byte boundary is reached. Additional padding bytes are added to fill to the alignment boundary of the double-word structure member. See *Figure 1.*

Example:

```
struct A {
int i;
unsigned bitfield: 4;
} a;
```

The arrangement of **a**'s fields in memory will be:



TL/EE/10345–4

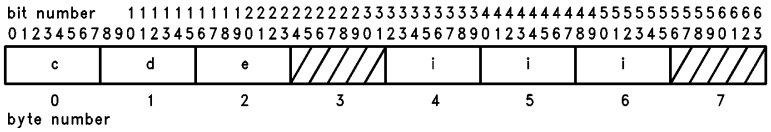
FIGURE 1. Bitfield Padding

Figure 2 is an example of the alignment on bit-fields given the different align switch settings. To summarize, the **−J (/ALIGN)** switch affects:

- the alignment and padding used for structure members and the alignment of variables of the structure type.
- the total storage alocated to a structure by determining if, and how many, padding bytes will be added after its last field.

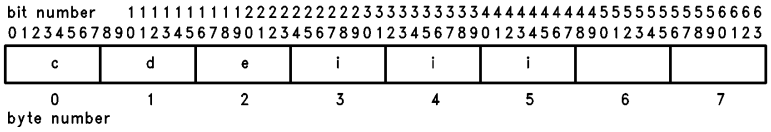
Example:
```
struct X {
char c,d,e;
int i: 24;
}
```

ALIGN = 4



TL/EE/10345–5

ALIGN = 2/1



TL/EE/10345–6

FIGURE 2. Alignment on Bitfields

CAUTION

The user must make sure that all parts of the program, including library routines, use the same alignment for the same structures; otherwise, problems result. The following example illustrates this point.

Suppose the example program includes <**stdio.h**>. The file <**stdio.h**> contains the following definitions.

```
extern FILE_iob [_NFILE];
typedef struct {
        int       cnt;
        unsigned char *_ptr;
        unsigned char *_base;
        char      _flag;
        char      _file;
} FILE;
```

Note that FILE has two char members at its end. If align = 4, any variable declared to be of type FILE will have two padding bytes added at its end in order to make it occupy an integral number of double-words. When align = 1 or align = 2, no padding is performed.

If a module using <stdio.h> is compiled with align = 4 and later linked with a module compiled with align = 1 or align = 2 that tries to use iob[n] when n > 0, the result will be wrong. This is because the two modules disagree on the size of the elements in the array. This situation actually does arise if a user module, compiled with align = 1 or align = 2, is linked with the default library libc, which is compiled with align = 4.

The solution to this problem is to make sure all modules are compiled using either the same alignment setting, including all include files and libraries, or a revised header file that has been made insensitive to the setting of the alignment switch. This is performed by including the necessary padding to enforce equal sizes and offsets. If the latter solution is chosen, FILE is revised to look like:

```
typedef struct {
        int      cnt;
        unsigned char *_ptr;
        unsigned char *_base;
        char     _flag;
        char     _file;
        /*padding*/  int:16;
} FILE;
```

No padding is added by the compiler, and the size of the structure is the same for all switch settings.

1.5 FUNCTIONS THAT RETURN A STRUCTURE

In the GNX-Version 3 C compiler, structure returning functions have a hidden argument which is the address of an area the size of the returned structure. This area is allocated by the caller and its address is passed as a first argument to the structure returning function. Structure returning functions are, therefore, re-entrant and interruptible.

Note: At the optimizer's discretion, small structures (less than 5 bytes) may be passed and/or returned in a register.

1.6 MIXED-LANGUAGE PROGRAMMING

Mixed-language programs are frequently used for two reasons. First, one language may be more convenient than another for certain tasks. Second, code sections already written in another language (e.g., an already existing library function) can be reused simply by calling them.

A programmer who wishes to mix several programming languages needs to be aware of subtle differences between the compilation of the various languages. An Application Note is available that describes the issues one needs to be aware of when writing mixed-language programs and compiling and linking such programs successfully.

1.7 ORDER OF EVALUATION OF PARAMETERS

The evaluation order of expressions and actual parameters in theGNX-Version 3 C compiler may differ from those of other compilers. Therefore, programs that rely on a specific order of evaluation may not run correctly when compiled. In particular, the following orders of evaluation are unspecified:

- The order in which expressions are evaluated.
- The order in which function arguments are evaluated.
- The order in which side effects take place. For instance, **a[i++] = i** may be evaluated as

```
a[i] = i;
i++
```

or as

```
t = i;
i++
a[t] = i;
```

1.8 ORDER OF ALLOCATION OF MEMORY

The order of allocation of local variables in memory is compiler-dependent. After the optimizer of the GNX-Version 3 C compiler performs register allocation, it reorders the local variables left in memory. This reordering reduces memory space requirements and minimizes displacement length. User programs that rely on any order of allocation of local variables may not run correctly.

1.9 REGISTER VARIABLES

By default, register variables, as well as other local variables, are equal candidates for register allocation. When given complete freedom, the programmer generally performs a better job of register allocation than when forced to follow the allocation. For programs which make assumptions about variables which reside in specific registers, an optimization flag (**-Ou** or **-O -Fu** on UNIX and **USER__REGISTERS** on VMS™) is available to enforce the **pcc** allocation scheme for register variables of scalar types and of type double.

1.10 FLOATING-POINT ARITHMETIC

The floating-point arithmetic conversion rules of the GNX-Version 3 C compiler differ from most other C compilers.

In an operation not containing double-operands, if one of two operands is of type **float**, the other operand is converted to type **float** and single-precision arithmetic is used. The result of the operation is of type **float**. This behavior differs from previous compilers which perform such operations in double precision.

In old C compilers, the result of float-returning functions was actually returned in double-format and placed in the F0–F1 register pair. When compiled by the GNX-Version 3 C compiler, such functions return the return value result in float format and place the result in the F0 register. Note that assembly programs that interface with float-returning functions may now incorrectly expect a double precision result.

Float parameters, however, are passed as double because the C language semantics do not require type identity between actual and formal parameters. Code is generated in the called function to convert these actual double values back to float if necessary.

Floating-point constants are of type **double**, unless they are typecast to **float** or are suffixed by the letter **f** or **F**. By preference, constants of type **float** should be used in float expressions to avoid the unnecessary casting of other operands to double precision. For example,

```
fmax+ = 17.5f;
```

is more efficient than

```
fmax+ = 17.5;
```

The following examples are of double constants and float constants.

Example:	Double Constants	Float Constants
	14.5 e6	14.5e6f
	14.5	(float) 14.5

2.0 SERIES 32000 STANDARD CALLING CONVENTIONS

The main goal of standard calling conventions is to enable the routines of one program to communicate with different modules, even when written in multiple-programming languages. The standard calling conventions support various special language features (such as the ability to pass a variable number of arguments, which is allowed in C), by using the different calling mechanisms of the Series 32000 architecture. These conventions are employed only to call externally visible routines. Calls to internal routines may employ even faster calling sequences by passing arguments in registers, for instance.

The standard Series 32000 calling conventions are used by the C compiler for calls to external routines of all languages. It is, therefore, unnecessary to use the *fortran* keyword in C programs, (if present, the keyword is ignored). However, local or internal routines (functions which in C are preceded by the static keyword) are called by more efficient calling sequences.

Basically, the calling sequence pushes arguments on top of the stack, executes a call instruction, and then pops the stack while using the fewest possible instructions to execute at the maximum speed. The following sections discuss the various aspects of the Series 32000 standard calling conventions.

2.1 CALLING CONVENTION ELEMENTS

Elements of the standard calling sequence are as follows:

2.1.1 The Argument Stack

Arguments are pushed on the stack from right to left; therefore, the leftmost argument is pushed last. Consequently, the leftmost arguments are always at the same offset from the frame pointer, regardless of how many arguments are actually passed. This allows functions with a variable number of arguments to be used.

Note: This does not imply that the actual parameters are always evaluated from right to left. Programs cannot rely on the order of parameter evaluation.

The run-time stack must be aligned to a full double-word boundary. Argument lists always use a whole number of double-words; pointer and integer values use a double-word (by extension, if necessary), floating-point values use eight bytes and are represented as **long** values; structures (records) use a multiple of double-words.

Note: Stack alignment is maintained by all GNX-Version 3 compilers through aligned allocation and de-allocation of local variables. Interrupt routines and other assembly-written interface routines are advised to maintain this double-word alignment.

The caller routine must pop the arguments off the stack upon return from the called routine.

Note: The compiler uses a more efficient organization of the stack frame if the **FIXED__FRAME** (**–OF**) optimization is enabled. In that case, programs should not rely on the organization of the stack frame.

2.1.2 Saving Registers

General registers R0, R1, and R2 and floating registers F0, F1, F2, and F3 are temporary or scratch registers whose values may be changed by a called routine. Also included in this list of scratch registers is the long register L1 of the NS32381 FPU. It is not necessary to save these registers on procedure entry or restore them before exit. If the other registers (R3 through R7, F4 through F7, and L3 through L7 of the NS32381) are used, their values should be saved (onto the stack or in temps) by the called routine immediately upon procedure entry and restored just before executing the return instruction. This should be performed because the caller routine may rely on the values in these registers not changing.

Note: Interrupt and trap service routines are required to save/restore all registers that they use.

2.1.3 Return Value

An integer or a pointer value that returns from a function, returns in (part of) register R0.

A long floating-point value that returns from a function, returns in register pair F0-F1. A float-returning function returns the value in register F0.

If a function returns a structure, the calling function passes an additional argument at the beginning of the argument list. This argument points to where the called function returns the structure. The called function copies the structure into the specified location during execution of the return statement. Note that functions that return structures must be correctly declared as such, even if the return value is ignored.

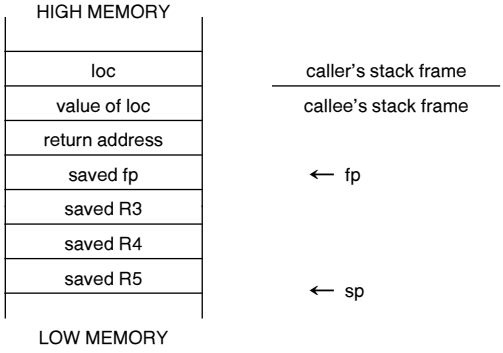
Example:

```
int iglob;
m( )
{
 int loc;
 a = if unc(loc);
}
if unc(pl)
int pl;
}
 int i, j, k;
 j = 0;
 for (i = l; i ≤ pl; i++)
  j = j + f(i);
 return(j);
}
```

The compiler may generate the following code:

```
_m:
          enter   [ ],4          #Allocate local variable
          movd    -4(fp),tos     #Push argument
          bsr     _if unc
          adjspb  $(-4)          #Pop argument off stack
          movd    r0,_iglob      #Save return value
          exit    [ ]
          ret     $(0)
_ifunc:
          enter   [r3,r4,r5],0   #Save safe registers
          movd    8(fp),r5       #Load argument to temp register
          movqd   $(0),r4        #Initialize j
          cmpqd   $(l),r5
          bgt     .LLl
          movqd   $(l),r3        #Initialize i
.LL2:
          movd    r3,tos         #Push argument
          bsr     _f
          adjspb  $(-4)          #Pop argument off stack
          addd    r0,r4          #Add return value to j
          addqd   $(l),r3        #Increment i
          cmpd    r3,r5
          ble     .LL2
.LLl:
          movd    r4,r0          #Return value
          exit    [r3,r4,r5]     #Restore safe registers
          ret     $(0)
```

After the enter instruction is executed by ifunc( ), the stack will look like this:



3.0 UNDEFINED BEHAVIOR

In the following cases, the behavior of the GNX-Version 3 C compiler is undefined:

- The value of a floating-point or integer constant is not representable.
- An arithmetic conversion produces a result that cannot be represented in the space provided.
- A volatile object is referred to by means of a pointer to a type without the volatile attribute.
- An arithmetic operation is invalid, such as division by 0, or produces a result that cannot be represented in the space provided, such as overflow or underflow.
- A member of a union object is accessed using a member of a different type.
- An object is assigned to an overlapping object.
- The value of a register variable has been changed between a **setjmp** call and a **longjmp** call.

Lit. # 100601

National Semiconductor Corporation
2900 Semiconductor Drive
P.O. Box 58090
Santa Clara, CA 95052-8090
Tel: 1(800) 272-9959
TWX: (910) 339-9240

National Semiconductor GmbH
Livry-Gargan-Str. 10
D-82256 Fürstenfeldbruck
Germany
Tel: (81-41) 35-0
Telex: 527649
Fax: (81-41) 35-1

National Semiconductor Japan Ltd.
Sumitomo Chemical
Engineering Center
Bldg. 7F
1-7-1, Nakase, Mihama-Ku
Chiba-City,
Ciba Prefecture 261
Tel: (043) 299-2300
Fax: (043) 299-2500

National Semiconductor Hong Kong Ltd.
13th Floor, Straight Block,
Ocean Centre, 5 Canton Rd.
Tsimshatsui, Kowloon
Hong Kong
Tel: (852) 2737-1600
Fax: (852) 2736-9960

National Semiconductores Do Brazil Ltda.
Rue Deputado Lacorda Franco
120-3A
Sao Paulo-SP
Brazil 05418-000
Tel: (55-11) 212-5066
Telex: 391-1131931 NSBR BR
Fax: (55-11) 212-1181

National Semiconductor (Australia) Pty, Ltd.
Building 16
Business Park Drive
Monash Business Park
Nottinghill, Melbourne
Victoria 3168 Australia
Tel: (3) 558-9999
Fax: (3) 558-9998