Using the GNX-Version 3 C Optimizing Compiler in the VMS Environment

National Semiconductor
Application Note 606
June 1989

1.0 INTRODUCTION

To optimize the performance of systems built around National's Embedded System Processor™ and Series 32000® microprocessors, National has developed a set of advanced optimizing compilers. Four compilers are available to support the C, Pascal, FORTRAN 77, and Modula 2 high-level languages. They are offered with Release 3.0 of the GENIX™ Native and Cross-Support (GNX™) Language Tools. By generating high-quality code specifically tailored to the Series 32000 architecture, these compilers allow Series 32000 microprocessors to achieve their full performance potential.

National's optimizing compilers use advanced optimization techniques to improve speed or save space. When code size is critical, the compilers can produce code that is more compact than code generated by other compilers. When speed is important, they can produce code that is 30%–200% faster.

Figure 1-1 shows the compilation process performed by National's optimizing compilers. When a program is compiled, the compiler performs syntactic and semantic verification of the source code and then translates it into a unique intermediate language called IR32.

Next, the IR32 code is passed to a dedicated optimizer. The optimizer performs four optimization steps to tailor the code to the processor architecture.

The first step is local optimization. During this step, the IR32 code is partitioned into basic blocks. Each basic block consists of a straight sequence of code. The only branches allowed in a basic block are at the entry or exit of the sequence. Some of the local optimizations performed include constant folding, value propagation, and the elimination of redundant assignments.

The second optimization step is flow optimization. During this step, a flow graph is constructed in which each basic block of code is represented by a node. Optimizations of the flow and elimination of dead code are performed during this step.

The third optimization step is global optimization. During this step, global code transformations are performed to speed program execution. Optimizations performed include loop-invariant code motion and the elimination of fully and partially redundant expressions.

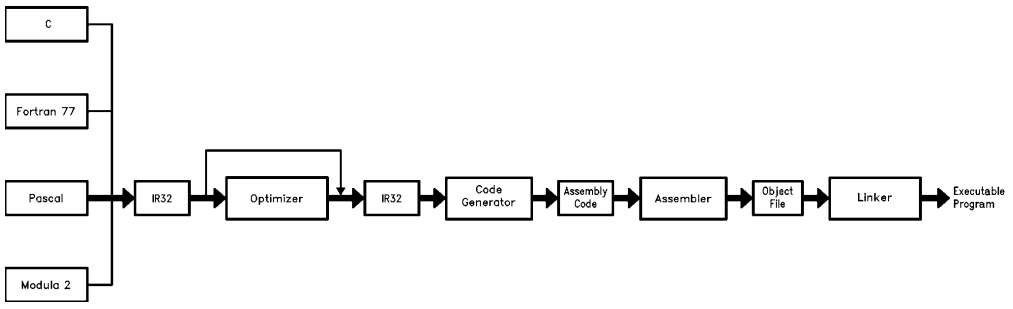
Register allocation is the fourth optimization step performed by the optimizer. During this step, variables are placed in registers instead of main memory. The use of volatile registers and the allocation of register parameters are also optimized.

After the IR32 code has been optimized by the optimizer, it is passed to the code generator. The code generator further optimizes the code by selecting optimal code sequences, performing peephole optimizations, aligning the code and data, and performing frame optimizations. It then translates the optimized IR32 code into assembly code.

Finally, an assembler generates object files from the assembly code, and a linker links the files together for execution.

This application note presents guidelines for using the GNX-Version 3 C Optimizing Compiler. However, much of the information presented here also applies to the optimizing compilers for Pascal, FORTRAN 77, and Modula 2. Topics presented here include:

- Optimization options for VMS systems.
- VMS command-line optimization options.
- Porting existing C programs to the GNX-Version 3 C Optimizing Compiler.
- Debugging optimized code.
- Additional techniques to improve code quality.
- Time requirements for compilation.
- Specifying a target machine.



TL/EE/10401–1

FIGURE 1-1. The Compilation Process

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

2.0 OPTIMIZATION OPTIONS

Table 2-1 lists all of the optimization options for the GNX-Version 3 C Optimizing Compiler. Different combinations of optimization flags can be used to tailor the optimizations for specific applications. For example, some applications must be optimized for speed, while others require smaller code size.

3.0 VMS COMMAND-LINE OPTIONS

The fastest possible code is generated by specifying /OPTIMIZE on the command line. This is equivalent to entering:

/OPTIMIZE = (FIXED__FRAME, CODE__MOTION,
REGISTER__ALLOCATION, FLOAT__FOLD,
SPEED__OVER__SPACE, NOVOLATILE,
STANDARD__LIBRARIES, NOUSER__REGISTERS)

In special cases, such as when compiling operating-system code, it may be necessary to change some of the optimization flags from their default settings. Table 3-1 suggests situations in which turning off an optimization option may be desirable.

Note that specifying the compiler debug option (/DEBUG) on the command line automatically turns off the optimizer fixed-frame option (/FIXED__FRAME) unless otherwise specified by the user.

Also note that using the compiler option /TARGET = (BUSWIDTH=1) favors space over speed by saving alignment holes normally produced when the buswidth is the default (n = 4).

Even when the optimizer pass is omitted, some optimizations are performed by the code generator. Therefore, specifying /NOOPTIMIZE (the default for this qualifier) is equivalent to entering:

/OPTIMIZE=(NOOPT, NOFIXED__FRAME,
NOCODE__MOTION, NOREGISTER__ALLOCATION,
NOFLOAT__FOLD, SPEED__OVER__SPACE,
NOVOLATILE, NOSTANDARD__LIBRARIES,
USER__REGISTERS)

4.0 PORTING EXISTING C PROGRAMS

Almost every program that runs when compiled by other C compilers, will compile and run under the GNX-Version 3 C compiler without any changes in the source code. Occasionally, however, a program may operate differently than

TABLE 2-1. Optimization Options

VMS	Description
NOOPT	does not invoke the optimizer phase.
NOFLOAT__FOLD	does not compute floating-point constant expressions at compile time.
FLOAT__FOLD	performs floating-point constant folding.
FIXED__FRAME	uses fixed frame references, avoids use of the FP register or the ENTER/EXIT instruction.
NOFIXED__FRAME	compiles for debugging: uses slower FP and TOS addressing modes.
NOVOLATILE	applies all optimizations to all variables (including global variables).
VOLATILE	compiles system code: assumes that all global and static memory variables and pointer dereferences are volatile.
STANDARD__LIBRARIES	assumes use of standard run-time library.
NO STANDARD__LIBRARIES	assumes that all routines have corrupting side effects.
CODE__MOTION	performs global code motion optimizations.
NOCODE__MOTION	does not perform global code motion optimizations.
NOUSER__REGISTERS	ignores user register declarations.
USER__REGISTERS	allocates user-declared register variables in registers as done by pc.
REGISTER__ALLOCATION	performs the register allocation pass of the optimizer.
NOREGISTER__ALLOCATION	does not perform the register allocation pass of the optimizer.
SPEED__OVER__SPACE	optimizes for speed only.
NOSPEED__OVER__SPACE	does not waste space in favor of speed.

TABLE 3-1. Reasons to Turn Off Optimization Options

VMS	Description
NOFIXED__FRAME	to debug the program or to compile non-portable programs that assume knowledge of the run-time stack.
VOLATILE	to compile system programs, such as device drivers, which contain variables that change or are referenced spontaneously.
NO__STANDARD__LIBRARIES	to compile programs which reimplement standard functions, in a way which does not agree with the optimizers assumptions (*i.e.*, have side effects).
NOFLOAT__FOLD	to compile programs whose correct execution depends on the order in which floating-point expressions are evaluated.
NOCODE__MOTION	to compile programs which contain huge functions, which are a drain on the system's resources and are time consuming to optimize.
USER__REGISTERS	to compile programs which rely on the register allocation scheme of pcc.
NOREGISTER__ALLOCATION	to run programs that cease to work when performing register allocation.
NOSPEED__OVER__SPACE	to compile programs which must fit as tightly as possible in memory.
NOOPT	when the optimizer phase is not required and another flag needs to be turned off as well.

before. Other programs may work when compiled without the optimizer, but will not work when the code is optimized. Possible causes for these problems are described in the following sections.

4.1 Undetected Program Errors

The single most common reason for a nonfunctioning program is an undetected program error. These errors become apparent when a different compiler is used or when the code is optimized. Many of these errors result from compiler-specific code in non-portable programs. The following lists some of the most common problems:

- Uninitialized local variables.

The memory and register allocation algorithms of the GNX-Version 3 C Optimizing Compiler are very different from those of other compilers. As a result, a local variable may end up in a completely different place than expected. Because of this, there is no guarantee that local variables will contain zero when the program is started. Therefore, all local variables should be initialized from within the program.

- Relying on memory allocation.

If two variables are declared in a certain order there is no guarantee that they will actually be allocated in that order. Therefore, a program, which uses address calculations to proceed from one declared variable to another declared variable may not work.

- Failing to declare a function.

A char returning function will return a value in the low-order byte of R0, without affecting the other bytes. A failure to declare that function where it is used may result in an error. For instance, assuming that get__code ( ) is defined to return a char, then:

```
main( ) {
        int  i;
        if ((i = get_code( )) = 17)
            do_something( );
}
```

may never execute do__something, even if get__code returns 17. This is because the whole register is compared to 17, not just the low-order byte.

A similar problem exists for functions which return **short** or **float**, or those which return a structure.

4.2 Compiling System Code

System code is distinguished from general ''high-level'' code by the fact that it is machine-dependent, often contains real-time aspects and interspersed **asm** statements, and is often driven by asynchronous events, such as interrupts. Examples of system code are interrupt routines, device handlers, and kernel code.

To the optimizer, ordinary-looking global variables can actually be semaphores or memory-mapped I/O that can be affected by external events not under the optimizer's control. Even so, it is still possible to optimize such code by taking some precaution and by activating some special optimization flags. Some of these issues are discussed in the following sections.

- Volatile variables

Volatile variables are variables that may be used or changed by asynchronous events, such as I/O or interrupts. The /VOLATILE flag treats all global variables, static variables, and pointer dereferences as volatile. This means that they are not subject to any optimizations. As a result, the number and nature of memory references to them will not change. (Note: Individual identifiers can be declared as volatile by using the volatile type modifier.) The following examples demonstrate the consequences of volatile variables and pointer dereferences.

Examples: 1. x = 17; x = 18;

If x is volatile, both of the two assignments to x are executed even though the first one seems redundant.

2. x = 9;

   y = x + 1;

If x is volatile, this program segment is not optimized to y = 10.

3. *p = b + c;

If *p is volatile, then this results in

```
movd  b, REG
addd  c, REG
movd  REG, 0(p)
```

and not

```
movd  b, 0(p)
addd  c, 0(p)
```

The difference stems from the fact that the second sequence, though faster, makes two references to 0(p) when the programmer may have wanted only one.

4.3 Timing Assumptions

Optimizing a program changes the timing of various constructs. In particular, delay-loops may now run faster than before.

4.4 Low-Level Interface

- Relying on register order

A program that relies on the fact that a given register variable resides in a specific register must be compiled with the /USER_REGISTERS flag turned on. (See section 6.7.)

- Relying on frame structure

A program that relies on a specific frame structure must be compiled with the /FIXED_FRAME flag turned off. This includes, in particular, programs that use the standard alloca( ) function that allocates space on the user's frame.

Referring to variables on the frame of a different function (such as the caller of this function) by complex pointer arithmetic may also cease to work.

- Using asm statements

The code inserted by asm statements may cease to work because the surrounding code produced by the GNX-Version 3 C compiler will normally differ from another compiler's code. (See section 6.6.)

4.5 Using Non-Standard Library Routines

The GNX-Version 3 C compiler assumes by default that all the C standard mathematical library routines listed in Table 4-1 are available as a standard run-time library. These library routines have absolutely no access to global variables. Therefore, calls to these routines are specially recognized and marked as calls that do not disturb optimizations of global variables. This is normally a safe assumption since it is unusual for a program to redefine (and thereby hide) these standard routines. In addition, the functions *abs*, *fabs*, and *ffabs* actually compile into in-line code and do not generate a procedure call at all.

The compiler generates a warning message whenever it compiles a program which does redefine one of these routines. In this case, the user must decide whether the redefined behavior of the routine is consistent with the assumption of the optimizer that it will not affect the optimization of global variables. If it does affect global-variable optimizations, the user has the choice of:

- renaming the redefined routine (so that calls to it are not specially recognized), or
- using the /NOSTANDARD_LIBRARY flag to turn off the recognition of all library routines.

TABLE 4-1. Recognized Library Routines

abs	erf	fceil	fhypot	fsinh	jn	sqrt
acos	erfc	fcos	flog	fsqrt	ldexp	tan
asin	exp	fcosh	flog10	ftan	log	tanh
atan	fabs	ferf	fmod	ftanh	log10	y0
atan2	facos	ferfc	fmodf	gamma	modf	y1
cabs	fasin	fexp	fpow	hypot	pow	yn
ceil	fatan	ffabs	frexp	j0	sin	
cos	fatan2	ffmod	fsin	j1	sinh	
cosh	fcabs	ffmodf				

4.6 Reliance on Naive Algebraic Relations

The optimizer performs floating-point constant folding. That is, it rearranges expressions to evaluate constant subexpressions at compile time. As a result, some naive algebraic expressions are folded away.

```
Example:  do {
                  a = a*2;
          }
          while ((a + 1.0) - 1.0 = a);
  is optimized to
          do {
                  a = a*2;
          }
          while (1);
  which was not the programmer's intention.
```

To maintain the program and keep the programmer's original intention, the programmer should use the /NOFLOAT_FOLD flag to suppress the folding optimization.

5.0 DEBUGGING OF OPTIMIZED CODE

Most of the time, the user should not need to debug an optimized program. The majority of all bugs can be found before optimization is turned on. However, there are some very rare bugs which make their appearance only when the optimizer is introduced. These bugs are difficult to find without a debugger.

The problem is that code motion optimizations and register allocation make most of the symbolic debugging information generated by the compiler obsolete. With this in mind, special care must be used when reviewing assembly code generated by the compiler. The following ''rules of thumb'' can be employed when using symbolic debug information together with the optimizer:

- Line number information is correct, but the code performed at the specified lines may be different from non-optimized code. This is a result of various code motion optimizations, such as moving loop invariant expressions out of loops.
- Symbolic information for global variables is normally correct, since global variables are rarely put in registers. In particular, if a global variable is not referenced within the current procedure, the value in memory is valid and the symbolic information is correct.

- Symbolic information for parameters is correct except in the following two cases:
  1. When a parameter is allocated a register and there is an assignment to that parameter, the symbolic information is incorrect.
  2. When a parameter of a local procedure is passed in a register as a result of an optimization, the symbolic information is incorrect. In this case, the symbolic information of all other parameters is incorrect because their offset within the procedure's frame has been changed.
- Symbolic information of local variables is likely to be incorrect because most of the local variables are put in registers; the rest of the local variables are reordered into new frame locations.
- Note that if symbolic information is requested, then slightly different code is generated. This happens because the /FIXED__FRAME optimizing flag is automatically disabled when the /DEBUG qualifier is used. Specifically, the ENTER instruction is always generated at the entry of procedures, and frame variables are referenced by FP-relative rather than SP-relative addressing mode. Without disabling this flag, symbolic debugging is almost impossible.

It is helpful to have an assembly listing of the program in question which has been compiled with the /ASM and the /ANNOTATE qualifiers. Such a listing contains comments from the optimizer regarding its actions.

6.0 ADDITIONAL GUIDELINES FOR IMPROVING CODE QUALITY

The following programming guidelines take advantage of the GNX-Version 3 C compiler optimizations to further improve the quality of compiled code.

6.1 Static Functions

It is not only good software engineering practice, but also good optimization practice to declare all functions not called from outside the file as ''static.'' This allows the optimizer to use a more efficient internal calling sequence to call such routines. This internal calling sequence uses the JSR instruction instead of the HSR or CXP instruction and also passes parameters in registers rather than on the stack.

Note: If a program consists of a single file, and compilation and linking is indicated in one step, then all functions within that file are automatically considered as static by the compiler.

6.2 Integer Variables

Many operators, including index calculations, are defined in C to operate on integers and imply a conversion when given non-integer operands. Therefore, to avoid frequent run-time conversions from **char** or **short** to **int**, integer variables should be defined as type int and not short or char. This is particularly important for integer variables that serve as array indices.

6.3 Local Variables

Since local variables have a better chance of being placed in registers, they should be used as much as possible, particularly when they are employed as loop counters or array indices.

6.4 Floating-Point Computations

In programs which do not require double-precision floating-point computations, a significant run-time improvement can be achieved by using the following guidelines:

- All functions should be defined as returning type, **float** not **double**.
- All constants should be defined to be **float** using the f suffix or cast expressions explicitly to **float**.
- The single-precision version of the standard floating-point routines should be used. For example, ffabs( ) should be used instead of abs( ), fsin( ) instead of sin( ), etc.

6.5 Using Pointers

6.5.1 Terminology

The following terms are used throughout this section.

- Potential definition

A statement potentially defines a memory location if the execution of the statement may change the contents of that memory location.

Example: A call to a function potentially defines all global variables because their values may change during the execution of that function. Imagine the following code fragment:

```
extern int *p, *q;
:
*p = 8;
:
```

The assignment statement potentially defines the memory location *q because q may point to the same memory location as p. The location *p is defined (i.e., given a new value) by the assignment. Location *q may be changed; therefore, it has the potential definition.

- Potential use.

A statement ''potentially uses'' a memory location if it may reference (read from) that memory location.

- Address taken variable.

A variable is considered ''address taken'' if the address operator (&) is applied to it within the file or if the variable is a global variable that is visible by other files.

- Volatile/nonvolatile registers.

By convention, the registers are divided into volatile registers (registers R0 through R2 and F0 through F3) and nonvolatile registers (registers R3 through R7 and F4 through F7). Volatile registers may be changed by a procedure call, whereas nonvolatile registers are guaranteed to retain their value across procedure calls. Therefore, all nonvolatile registers used within a procedure must be saved at the entry and restored at the exit of that procedure.

6.5.2 Potential Difference Assumptions

The optimizer does not keep track of the contents of pointers. Therefore, it cannot tell, for any given location in the program, where each pointer is pointing. Since a pointer can point to any memory location, the optimizer makes the following assumptions concerning pointer usage:

1. Every assignment to a pointer dereference (the location pointed to by a pointer) potentially defines all other pointer dereferences and all address-taken variables.
2. Every use of a pointer dereference (i.e., a value read through a pointer) potentially uses all other pointer dereferences and all address-taken variables. This is because any accessible memory location is potentially read.
3. Every function call potentially defines and potentially uses all pointer dereferences, all address taken-variables, and all global variables. Therefore, using pointers, the function's code may read and/or write any accessible memory location. Of course, any global variable may be used and/or changed.

When working with pointers, these assumptions should be considered. For example, using arrays is preferable to using pointers. The following example illustrates this point. Assume a is an array of char and p is a pointer to char. The two program segments perform the same function.

```
Example:  program segment 1
          for (i = 0; i != 10; i++){
               a[i] = global_var;
               a[i+1] = global_var + 1;
          }
          program segment 2
          for (p = &a[0]; p != &a[10]; p++){
               *p = global_var;
               *(p+1) = global_var + 1;
          }
```

In program segment 1, global__var can be put in a register. In program segment 2, however, p may point to global__var. The first statement (*p = global__var) potentially defines global__var; therefore, it cannot be put in a register.

6.5.3 Common Subexpressions

Another aspect of this same issue is that of common subexpressions. The optimizer normally recognizes multiple uses of the same expression and saves that expression in a temporary variable (usually a register). This cannot be performed when worst-case assumptions are made about potential definition of expressions (as described above). Expressions that contain pointer dereferences or global variables are vulnerable. Therefore, if many uses of the same expression span across procedure calls, it is advisable to save them in local variables. Consider the example:

```
foo1(p → x);
foo2(p → x);
```

Here, the expression p → x cannot be recognized by the optimizer as a common subexpression because foo1( ) may change its value. In this case, the following hand optimization may help:

```
t = p → x; /* t is local, therefore */
foo1(t);       /* not potentially defined by foo1( ) */
foo2(t);       /* so its value is still valid for foo2( ) */
```

The programmer can make this optimization by using the knowledge that p → x is not changed by foo1( ). The optimizer cannot do the same because it assumes the worst case.

6.6 asm Statements

The keyword **asm** is recognized to enable insertion of assembly instructions directly into the assembly file generated. The syntax for its use is:

asm (constant-string);

where constant-string is a double-quoted character string.

Extreme care should be taken if **asm** statements are used. The following guidelines should be observed:

- The optimizer is not aware of the contents of an **asm** statement. Therefore, it assumes that an **asm** statement potentially defines and potentially uses all of the variables (including local variables). This means that no common subexpressions can be recognized across an **asm** statement.
- In order to allow an **asm** statement to use a specific register (e.g., **asm** (''save [r0, r1, r2]'');), the optimizer de-allocates all the registers.
- The compiler usually generates code which differs from the code generated by other compilers. This applies particularly to allocation of local variables and parameters of static procedures.
- The code surrounding the **asm** statement may change as a result of changes in other parts of the procedure.
- An **asm** statement that contains a branch instruction or a branch target (label) may cause the optimizer to generate wrong code.

Note: For these reasons, looking at the generated assembly code is strongly recommended before and after inserting **asm** statements into a program.

6.7 Register Allocation

The C language is unique in that it allows the programmer to specify (or rather, recommend) that some variables be allocated to machine registers. The optimizer normally ignores these recommendations, since in most cases the optimizer's own register allocation algorithms are as good as or superior to the programmer's recommendations. There are several reasons for this:

- The user can use a register for one variable only. The optimizer, however, allocates a register along live ranges of variables, making it possible for several variables with non-conflicting live ranges to use the same register.
- The user can declare as a register only local variables whose addresses are not taken; whereas, the optimizer allocates global variables as well as variables whose addresses are taken (where possible).
- The user can allocate variables in safe registers only. Therefore, every register used has to be saved/restored at the entry/exit of the procedure. The optimizer allocates variables that do not live across procedure calls in unsafe registers. Therefore, these registers need not be saved/restored.
- Because of code motion optimizations, the number of references of variables may be changed. Therefore, the choice of register variables may not be optimal. This is illustrated in the following example:

```
Example:  int j;
          register int i;
          i = j;
          if (i == 3 || i == 4 || i == 5)
```

In this example, undesired effects result if optimized with the /USER__REGISTERS flag. The reason is that j is copy-propagated and replaces all occurrences of i. As a result, i occupies a register but is not used. If the ordinary register allocation of the optimizer is not invoked, or if there are no registers left, j will be placed in memory.

6.8 setjmp( )

Calls to setjmp( ) are specially recognized by the compiler. Procedures that contain calls to setjmp( ) are only partially optimized because procedure calls may end up in a call to longjmp( ). Code motion optimizations are performed only within linear code sequences (those sequences not containing branches or branch targets). Register allocation is limited to optimizer-generated temporary variables, register-declared variables, and variables whose live ranges do not contain function calls.

6.9 Optimizing for Space

The default behavior of the GNX-Version 3 C compiler optimizes for optimal speed. However, there are several things that can be done to improve code density:

- Optimize with the /NOSPEED__OVER__SPACE turned on.
- Squeeze the data area by using /TARGET = (BUS = 1) for smaller alignment between variables.
- Squeeze all record definitions by using the /ALIGN = 1 switch.

7.0 COMPILATION TIME REQUIREMENTS

Using the optimizer slows down the compilation process. Therefore, it is recommended that the optimizer be used only on final production versions of a program. The amounts of resources (time and memory) vary strongly from program to program and actually depend on the size of the routines in the compiled program file. The larger a routine, the more time and memory needed to optimize it. This behavior is more or less quadratic. That is, the optimizer needs about four times the resources to optimize a routine of 1000 lines than to optimize a routine of 500 lines.

If time or memory requirements are unacceptable and routines cannot be reduced to a reasonable size of about 500 lines, it is possible to turn off some optimizations using the /NOCODE__MOTION and/or the /NOREGISTER__ALLOCATION flags.

8.0 TARGET MACHINE SPECIFICATION

The GNX-Version 3 C Optimizing Compiler provides a way to tune the code for a specific target machine by specifying its CPU, FPU, and buswidth. The values for the CPU and FPU can either be the complete device name (e.g., NS32332 or NS32081) or the last three digits of the device name (e.g., 332 or 081). The buswidth is specified in bytes. This tuning is performed by specifying /TARGET on the command line. Table 8-1 lists the flags and the possible settings.

Example: The following example specifies an NS32332 CPU, an NS32081 FPU, and a buswidth of 4 bytes.

NMCC /TARGET = (CPU = 332, FPU = 081, BUS = 4) TEMP.C

TABLE 8-1. Target Selection Parameters

CPU (C)	FPU (F)	Buswidth (B)
[NS32]008	[NS32]081	1
[NS32]016	[NS32]381	2
[NS32]cg16	[NS32]580	4
[NS32]032		
[NS32]332		
[NS32]532		

Lit. # 100606

LIFE SUPPORT POLICY

NATIONAL'S PRODUCTS ARE NOT AUTHORIZED FOR USE AS CRITICAL COMPONENTS IN LIFE SUPPORT DEVICES OR SYSTEMS WITHOUT THE EXPRESS WRITTEN APPROVAL OF THE PRESIDENT OF NATIONAL SEMICONDUCTOR CORPORATION. As used herein:

1. Life support devices or systems are devices or systems which, (a) are intended for surgical implant into the body, or (b) support or sustain life, and whose failure to perform, when properly used in accordance with instructions for use provided in the labeling, can be reasonably expected to result in a significant injury to the user.
2. A critical component is any component of a life support device or system whose failure to perform can be reasonably expected to cause the failure of the life support device or system, or to affect its safety or effectiveness.

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