Application Development Using Multiple Programming Languages

National Semiconductor
Application Note 590
February 1989

INTRODUCTION

National Semiconductor provides optimizing compilers for software development for *Series 32000* based designs. GNX-Version 3 is the name of the software tools family that includes the optimizing compilers. Languages supported in GNX-Version 3 include compilers that support C, Pascal, FORTRAN-77, and Modula-2. Each of the optimizing compilers share a common optimizer and code generator and intermediate representation. This greatly simplifies the process of mixed-language programming, or combining modules written in different high-level languages in the same application. The ability to use mixed-language programming simplifies the porting of pre-existing applications and code reuse.

Mixed-language programs are frequently used for a 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 by simply making a call to them.

A programmer who wishes to mix several programming languages needs to be aware of subtle differences between the compilation of the various languages. The following sections describe the issues the user needs to be aware of when writing mixed-language programs and then compiling and linking such programs successfully.

WRITING MIXED-LANGUAGE PROGRAMS

The mixed-language programmer should be aware of the following topics:

- Name Sharing—Potential conflicts including permitted name-lengths, legal characters in identifiers, compiler case sensitivity, and high-level to assembly-level name transformations.
- Calling Convention—The way parameters are passed to functions, which registers must be saved, and how values are returned from functions. The application note *Portability Issues and the GNX-Version 3 C Optimizing Compiler* contains a description of parameter passing. This information is also contained in Appendix A of the *GNX-Version 3 compiler reference manuals.*
- Declaration Conventions—The demands that different languages impose when referring to an outside symbol (be it a function or a variable) that is not defined locally in the referring source file. Note that this is also true of references to an outside symbol that is not in the same language as that of the referring source file.

To help the programmer avoid these potential problems, a set of rules for writing mixed-language programs has been devised. Each rule consists of a short mnemonic name (for easy reference), the audience of interest for the rule, and a brief description of the rule.

Table I summarizes all of the rules in the context of each possible cross-language pair.

TABLE I. Cross-Language Pairs

	C	Pascal	FORTRAN 77	Modula-2	Series 32000 Assembly
Series 32000 Assembly	"_" prefix	"_" prefix include ext case sensitivity	"_" prefix "_" suffix ref args case sensitivity	"_" prefix DEF & IMPORT init code	
Modula-2	DEF & IMPORT init code	DEF & IMPORT init code include ext case sensitivity	"_" suffix DEF & IMPORT init code ref args case sensitivity		"_" prefix DEF & IMPORT init code
FORTRAN 77	"_" suffix ref args case sensitivity	"_" suffix include ext ref args		"_" suffix DEF & IMPORT init code ref args case sensitivity	"_" prefix "—" suffix ref args case sensitivity
Pascal	include ext case sensitivity		"_" suffix include ext ref args	DEF & IMPORT init code include ext case sensitivity	"_" prefix include ext case sensitivity
C		include ext case sensitivity	"_" suffix ref args case sensitivity	DEF & IMPORT init code	"_" prefix

 TL/EE10358

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

RULE 1 case sensitivity

This rule is of interest to every programmer who mixes programming languages.

Modula-2, C, and Series 32000 assembly are case sensitive while FORTRAN 77 and Pascal are not (at least according to the standard). Programmers who share identifiers between these two groups of languages must take this into account. To avoid problems with case sensitivity, the programmer can:

1. Take case to use case-identical identifiers in all sources and compile FORTRAN 77 and Pascal sources using the case-sensitive option (CASE_SENSITIVE on VMS, -d on UNIX).
2. Use only lower-case letters for identifiers which are shared with FORTRAN 77 or Pascal, since the FORTRAN 77 and Pascal compilers fold all identifiers to lower-case if not given the case-sensitive option.

RULE 2 "__" prefix

This rule is of interest to those who mix high-level languages with assembly code.

All compilers map high-level identifier names into assembly symbols by prepending these names with an underscore. This ensures that user-defined names are never identical to assembly reserved words. For example, a high-level symbol NAME, which can be a function name, a procedure name, or a global variable name, generates the assembly symbol __NAME.

Assembly written code which refers to a name defined in any high-level language should, therefore, prepend an underscore to the high-level name. Stated from a high-level language user viewpoint, assembly symbols are not accessible from high-level code unless they start with an underscore.

RULE 3 "__" suffix

This rule is of interest to those who mix FORTRAN 77 with C, Pascal, Modula-2, or assembly code.

The FORTRAN 77 compiler appends an underscore to each high-level identifier name (in addition to the action described in RULE 1). The reason for an appended underscore is to avoid clashes with standard-library functions that are considered part of the language, e.g., the FORTRAN 77 WRITE instruction. For example, a FORTRAN 77 identifier NAME is mapped into the assembly symbol __NAME__.

Therefore, a C, Pascal, Modula-2, or assembly program that refers to a FORTRAN 77 identifier name should append an underscore to that name. Stated from a FORTRAN 77 user viewpoint, it is impossible to refer to an existing C, Pascal, Modula-2, or assembly symbol from FORTRAN 77 unless the symbol terminates with an underscore.

RULE 4 ref args

This rule is of interest to those who mix FORTRAN 77 with other languages.

Any language which passes an argument to a FORTRAN 77 routine must pass its address. This is because a FORTRAN 77 argument is always passed by reference, i.e., a routine written in FORTRAN 77 always expects addresses as arguments.

Routines not written in FORTRAN 77 cannot be called from a FORTRAN 77 program if the called routines expect any of their arguments to be passed by value. Only routines which expect all their arguments to be passed by reference can be called from FORTRAN 77.

Pascal and Modula-2 programs must declare all FORTRAN 77 routine arguments using var. C programs must prepend the address operator & to FORTRAN 77 routine arguments in the call.

The C, Pascal, or Modula-2 programmer who wants to pass an unaddressable expression (such as a constant) to a FORTRAN 77 routine, must assign the expression to a variable and pass the variable, by reference, as the argument.

RULE 5 include ext

This rule is of interest to Pascal programmers who want to share variables between different source files which may or may not be written in Pascal.

Pascal sources which share global variables must define these variables exactly once in an external header (include) file. The external header file has to be included in all Pascal source files which access the shared global variable, and its name must have a .h extension.

RULE 6 DEF and IMPORT

This rule is of interest to those who mix Modula-2 with other languages.

Modula-2 modules which access external symbols must import external symbols. If external symbols are not defined in Modula-2 modules but defined in other languages, the programmer must export these symbols to conform with the strict checks of the Modula-2 compiler.

External symbols can be exported by writing a "dummy" DEFINITION MODULE which exports all of the foreign language symbols, making them available to Modula-2 programs.

This export must be nonqualified to prevent the module name from being prepended to the symbol name.

RULE 7 init code

This rule is of interest to those who mix Modula-2 with other languages.

Modula-2 modules which import from external modules activate the initialization code of the imported modules before they start executing. The initialization code entry-point is identical to the imported module name.

To avoid getting an "Undefined symbol" message from the linker, the programmer should define a possibly empty, initialization function for every imported module. This is in case the implementation part of that module is not written in Modula-2. It should be noted that the initialization code is not necessarily called during run-time. Initialization code is executed if, and only if, the following two conditions hold true:

1. The main program code is written in Modula-2.
2. The Modula-2 routines which are supposed to activate the initialization part are not called indirectly through some non-Modula-2 code.

In addition to these rules, a few points should be noted. First, GNX Version 3 FORTRAN 77 allows identifiers longer than the six character maximum of traditional FORTRAN compilers. Second, the family of GNX Version 3 compilers allows the use of underscores in identifiers. Both of these enhancements simplify name sharing.

IMPORTING ROUTINES AND VARIABLES

The general conventions of all languages must be kept in mixed-language programs. In particular, externals must be declared in those program sections which import them. The following are examples of declarations of external (imported) functions/procedures and external (imported) variables in each language. The examples are in the form:

Caller Language: *external (imported) functions/procedures or external (imported) variables*

```
C: extern int func_( );
    or
   extern int var_name_;
```

Note that the strict reference C model (draft-proposed ANSI C standard) is assumed. If the model is relaxed, then the external declarations are not mandatory.

```
FORTRAN 77: INTEGER func
             or
            COMMON /var_name/ local_name

   Pascal:  function func_: integer;
            external;
            procedure proc_; external;
             or
            #include "var_def.h"
```

where the file var__def.h contains the following declaration:

```
             var
              var_name_: integer;
```

as explained in RULE 5 (include ext).

```
Modula-2:    FROM modula_name IMPORT func_
              or
             FROM module_name IMPORT
             var_name_

Series 32000: .globl _func_
  assembly     or
             .globl _var_name_
```

USING THE ASM KEYWORD

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

asm *(constant-string)*;

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

Asm can be used inside of functions as a statement and out of functions in the scope of global declarations. A newline character will be appended to the given string in the assembly code.

Example: if for the C source:

```
i++;
j+ = 2;
```

the assembly code generated is:

```
addqd $1, _i
addqd $2, _j
```

then the assembly code generated for:

```
i++;
asm ("movd _i, r0");
j+ = 2;
```

will be:

```
addqd $1, _i
movd _i, r0
addqd $2, _j
```

Note: The word asm is a reserved keyword. Using asm as an identifier is a syntax error. Existing programs using such identifiers must be modified.

In support of mixed-language programming, the compiler also recognizes and compiles appropriate files written in other programming languages. Files with a .s suffix are assembly source programs and may be assembled (to produce .o files) and linked. Pascal, FORTRAN 77, and Modula-2 source files are also recognized, and compile appropriately if your system includes the National Semiconductor GNX Version 3 compiler for those languages. The suffixes for these files are listed in Table II.

TABLE II. Filename Conventions

File Name Suffix	File Type
.c	C Source File
.i	Preprocessed C Source File
.f, .for	FORTRAN 77 Source File
.F, .FOR	FORTRAN 77 Source with cpp Directives
.m, .mod	Modula-2 Source File
.M, .MOD	Modula-2 Source with cpp Directives
.def	Modula-2 Definition Module Source File
.DEF	Modula-2 Definition Module Source with cpp Directives
.p, .pas	Pascal Source File
.P, .PAS	Pascal Source with cpp Directives
.s	Assembly Source File
.o	Object Code
.a	Library Archive File

COMPILING MIXED-LANGUAGE PROGRAMS

After writing different program parts in different languages, keeping in mind the rules previously mentioned, the mixed-language programmer must still link and load these parts to make them run successfully. Three points should be mentioned in conjunction with the successful linking and loading of programs. These are as follows:

- External library (standard or nonstandard) routines must be bound with the user-written code that calls them.
- Initialization code which arranges to pass program parameters to the main program and then calls the main program, sometimes has to be bound with user-written code.
- The entry point of the code, i.e., the location where the program starts executing, should be determined.

In some cases, a standard is not so widely accepted as with Modula-2. In these cases, the user must be aware of the libraries that are available and the calling conventions of the main program used by the operating system.

LIBRARIES

Table III lists libraries associated with each compiler. When programming with mixed-languages, the libraries associated with the languages used must be bound with the program during the link phase of compilation.

TABLE III. Compilers and their Associated Libraries

Compiler (Driver) Name	Libraries
cc (Cross nmcc)	libc
f77 (Cross nm77)	libF77, libI77, libm, libc
pc (Cross nmpc)	libpas, libm, libc
m2c (Cross nm2c)	libmod2, libm, libc

INITIALIZATION CODE AND ENTRY-POINTS

Normally, the entry point of the final executable file is called *start*. The code that follows this entry-point is initialization code that prepares the run-time environment and arranges parameters to be passed to the user-written main program. The initialization object file which contains *start* is linked in by default is called *crt0.o*. The *crt0.o* file always calls main.

The assembly-symbol that starts the user main program in the C language is __main (the underscore is prepended by the C compiler) and is called __MAIN____ in Pascal, FORTRAN 77, or Modula-2 programs.

Note that the last three compilers completely ignore the user's main program name. Therefore, in C, the user-written code is called directly from *crt0.o*. In Pascal, FORTRAN 77, and Modula-2, __main is located in the respective standard library which performs additional initializations before calling the user entry-point __MAIN____.

COMPILATION ON UNIX OPERATING SYSTEMS

National Semiconductor's GNX tools (assembler, linker, etc.,) on systems relieve a user's concern about external libraries, initialization code, and entry-points. This is due to the coherency and consistency of the GNX-Version 3 compilers and their integration through the use of a common driver.

When using a GNX Version 3 compiler on a UNIX system, the user does not directly call the compiler front end, optimizer, code generator, assembler or linker. Instead, the calls are indirectly made through the driver program.

The driver program accepts a variable number of filename arguments and options and knows how to identify language-specific options. The driver also identifies the languages in which its filename arguments are written by the names of these arguments. Therefore, the driver can arrange to compile and bind the programs with the needed libraries in order to run the program successfully.

As mentioned earlier, the driver program used by C, Pascal, FORTRAN 77, and Modula-2 programmers is exactly the same program on UNIX systems. The respective driver names are cc, pc, f77, and m2c on native systems such as the SYS32/20 or SYS32/30 and nmcc, nmpc, nf77, and nm2c on cross-support systems such as VAX/VMS or a VAX running Berkley UNIX.

The driver program looks at its own name in order to determine the libraries that are bound with the program. In addition, the driver links additional libraries according to the name extensions of any of its filename arguments. For instance, cc also links libm and libpas when one of the filename arguments is a Pascal source (recognized by the .p extension).

The -v (verbose) option of the driver verbosely outputs all driver actions. With this option, the interested user can track problems that might arise (such as undefined symbols from the linker).

As mentioned in the previous section, different languages use different initialization code that resides in language-specific standard libraries. It is necessary that the correct language initialization code be linked with a mixed-language program. The driver program helps do this, but it needs to know in which language the main program is written.

To ensure that the correct initialization code is linked with a mixed-language program, the user should call the driver that corresponds to the language of the main program module within the mixed-language program.

For example, suppose there are five source modules written in five different languages (c__utils.c written in C, f__utils.f written in FORTRAN 77, p__tuils.p written in Pascal, m__utils.m written in Modula-2, and s__utils.s written in assembly), and there is a sixth module that has already been compiled separately (obj.o, an object module). Assuming there is a main program written in FORTRAN 77, the f77 driver should be used.

f77 main.f c__utils.c f__utils.f p__utils.p m__utils.m s__utils.s obj.o

If the main program is written in C, cc is used, and so on.

COMPILATION ON VMS OPERATING SYSTEMS

When using the GNX tools on VMS systems, the linking phase is separate from the compilation phase; therefore, it demands separate actions from the user.

The interested user should refer to the language tools manuals (assembler, linker, etc.) for a complete description of how to use them on VMS systems.

COMPILING A MIXED-LANGUAGE EXAMPLE

The example listed in Appendix A consists of a number of program modules written in languages different from the main program, which is written in C.

COMPILING THE EXAMPLE ON A UNIX SYSTEM

To compile the program modules on a Berkeley UNIX system, type the command:

```
nmcc c_main.c\
     c_fun.c dmod_fun.def dummy.def
     f77_fun.f\
     imod_fun.m pas_fun.p asm_fun.s
```

This assumes that all the program modules are in the same directory. If the program compiles and links successfully, the result is an executable file that, when run on a Series 32000 CPU, prints the line "Passed OK!!!".

APPENDIX A
PROGRAM MODULE LISTINGS

The different program modules are listed in this section.

c_main.c

```
/*-------------------------------------------------
 * Example of a C program which communicates with C, Pascal,
 * Fortran 77, Modula-2 and Assembly external functions, via
 * direct calls as well as via a global variable.
 * Parameter passing by reference is accomplished by passing the
 * addresses of the characters variables "a', "b', "c', "d' and "e'.
 *-------------------------------------------------*/
char str_[ ] = "Passed OK!!!\n''; /* global ('exported') string*/
main () {
   char a, b, c, d, e;
   int three = 3; /* FORTRAN must get its parameters by reference
                  *So we put this constant into a variable . . .
                  */
   if (c_func (&a, 0)  && /* in C arrays start with 0*/
    pas_func (&b,2)  && /* in Pascal they start at 1*/
    f77_func_(&c,&three)&& /*in f77, at 1*/
    mod_func (&d, 3)  && /* in Modula-2, at 0*/
    asm_func (&e, 4)) /*in assembly, at 0*/
     printf ("%c%c%c%c%c%s'', a, b, c, d, e, str_ +5);
     /*Should print "Passed OK!!!''*/
}
/* dummy initialization function for Modula-2*/
dummy ()
{
}
```

c_fun.c

```
/*
 * Declaration of the public character string 'str[]' and definition
 * of the C function 'c_func()'.
 * Note the appending of an underscore to the external symbol 'str_'
 * which is shared with FORTRAN 77.
 */
 extern char str_[];
int c_func (c_ptr, index)
char  *c_ptr;
int   index;
{
   *c_ptr = str_[index];
   return 1;
}
```

```
          f77_fun.f
C
C  The FORTRAN 77 function:
C
C  All parameters are passed by reference
C  The COMMON statement aliases the external array 'str' as 'text'
C
   LOGICAL FUNCTION f77_func(c, index)
   CHARACTER c
   INTEGER  index
   COMMON /str/text
   CHARACTER text(15)
   c = text(index)
   f77_func = .TRUE.
   RETURN
   END
          dmod_fun.def
DEFINITION MODULE mfunc_module;
  EXPORT mod_func;
  PROCEDURE mod_func(VAR c: CHAR; index: INTEGER): BOOLEAN;
END mfunc_module.
          dummy.def
 (*
  *  This definition module was written in order to 'satisfy' Modula-2
  *  strict conformance checks regarding the foreign language functions
  *  and in order to define the global character array 'str[]'.
  *  The external functions are called from the Modula-2 main program,
  *  so they must be exported from somewhere . . .
  *)
DEFINITION MODULE dummy;
 EXPORT
  str_, c_func, pas_func, f77_func_, asm_func;

  (*external function declarations*)
  PROCEDURE c_func (VAR c: CHAR; index: INTEGER): BOOLEAN;
  PROCEDURE pas_func (VAR c: CHAR; index: INTEGER): BOOLEAN;
  PROCEDURE f77_func (VAR c: CHAR; VAR index: INTEGER): BOOLEAN;
  PROCEDURE asm_func (VAR c: CHAR; index: INTEGER): BOOLEAN;
  VAR
  str_: ARRAY [0. . 14] OF CHAR;
  END dummy.
```

imod_fun.m

```
(*
 * Definition of the Modula-2 function 'mod_func ()'
 *)
IMPLEMENTATION MODULE mfunc_module;
  FROM dummy IMPORT str_;
  PROCEDURE mod_func(VAR c: CHAR; index: INTEGER): BOOLEAN;
  BEGIN
   c: = str_[index];
   RETURN (TRUE);
  END mod_func;
 END mfunc_module.
```

pas_fun.p

```
(*
 * The Pascal function 'pas_func ()'
 *)
(* 'str[] character-array declaration *)
#include 'str_pas.h'';
(* make this function visible to outsiders ('export')*)
function pas_func(var c: char; index: integer): boolean; external;

function pas_func ();
begin
  c: = str_[index];
  pas_func: = TRUE;
end;
```

str_pas.h

```
(* 'str[]' character-array declaration for Pascal*)
var
  str_: packed array [1. . 15] of char;
```

```
          asm_fun.s
#
# The 32000 Assembly Language Function 'asm_func'
#
#  The function includes an artificial use of r7, to demonstrate the
#  need to save it upon entry and restore upon exit, as opposed to
#  r0, r1 and r2; f0, f1, f2 and f3 which can be used freely without
#  saving or restoring. This is according to the Series 32000
#  standard calling convention.
#  The function return value is placed in r0, also according to the
#  standard calling convention.
#
   .globl _str_ #Import the global str[] array.
   .globl _asm_func #Export (make visible) the assembly function.
   .align 4

_asm_func:
   enter  [r7],0     #Set frame, demonstrate saving of r7
   movb  _str_+0(12(fp)),0(8(fp)) # argument_1 ← str[argument_2]
   movqd  $(1), r7      #artificial use of r7
   movd  r7, r0      #return_value  ←  TRUE
   exit  [r7]      #Unwind frame, restore r7
   ret  $(0)      #Return to caller
```

Lit. # 100590

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