MULTI-LANGUAGE PROGRAMMING WITH CLARION 3 AND THE TOPSPEED LANGUAGES

SEAN WILSON

TopSpeed Support Engineer
Clarion Software

The Clarion Language

It is usually advisable to develop an application in a single language if it is possible to do so without compromising the quality or performance of the finished product. By using a single language you avoid unnecessary restructuring to compensate in the differences between languages.

Strengths of the Clarion Language

Clarion is a high-level, business oriented language and development environment with support for string data types, dynamically typed parameters, automatic type conversions, screen control, database access, report generation and template controlled source code generation.

Weaknesses of the Clarion Language

Clarion offers limited facilities for low-level programming. It is not possible to write interrupt handlers in Clarion; there is no means of binding a variable to a machine address; address manipulation and pointers are not supported.

The C Language

C is a low-level programming language which was designed with system-level programming in mind. As a result of its popularity, there are a vast number of C utility libraries many of which may be of use to Clarion developers.

Strengths of the C Language

C provides good access to the machine's address space, pointer types and function pointers. TopSpeed C provides facilities for writing interrupt handlers. C is a popular language with a great deal of third-party support in the form of utility libraries and development tools.

Weaknesses of the C language

C is not particularly strongly typed, novice programmers will often fall victim to common programming errors

Using C with Clarion

Much of the Clarion compiler is written in C. The Clarion run-time library, which incorporates the C run-time library, is also written in C. As a result, there are no additional libraries that need to be linked when you write C functions for use from Clarion code. This means there is minimal overhead associated with using C together with Clarion. Since the majority of third party programming libraries are implemented in C, if you determine the need for a particular feature for use within your Clarion application, the chances are that it is available as a C library.

The C++ Language

C++ is a superset of the C language offering Object Oriented programming extensions. C++ is becoming increasingly popular, particularly since the quality of many commercial compilers is approaching that of the popular C compilers. There are a number of C++ class libraries available, though most of the popular libraries are user interface tools making them inappropriate for use within a Clarion application.

Strengths of the C++ Language

C++ offers better type checking than C, good access to the machine's address space, pointer types, and function pointers. TopSpeed C++ provides features for writing interrupt handlers. OOP features include operator and function overloading, multiple inheritance and polymorphism.

Multi-language Programming with Clarion 3 and the TopSpeed Languages

Clarion Database Developer v3.0 is an exceptional development tool. In just a few hours it is possible to generate applications that could take weeks or months to code from scratch using conventional programming tools (such as C or C++). In addition, Clarion's object code generation rivals that of most C compilers. Couple this with Clarion's extensive run-time library and you have an extremely powerful general purpose programming tool.

So why use C, C++, Modula-2 or Pascal code at all in a Clarion application? The answer is, most developers won't. Most Clarion developers will probably develop large, complex applications generated entirely within the Clarion development environment. A small number may resort to hand-written Clarion code to address more complex programming tasks.

There are several reasons why a developer might choose to utilise code within a Clarion application that is written in a different language:

- Some low-level programming tasks cannot be performed in Clarion: Clarion provides no facilities for writing interrupt handlers and very limited support for accessing machine addresses. Also, there is no means to bind the data at a given address to a variable, so there is restricted restrict access to the environment.

- Substantial investment in existing code written in another language: Some companies may have a large amount of code that is written in another language that would take too long to convert to Clarion. If the code is tried and tested it may well be a complete waste of time to attempt the conversion.

- Access to facilities provided by third party libraries: There are many third party libraries, particularly for C and C++, which can give the developer access to facilities not provided by the Clarion environment (e.g., Graphics, Communications, Financial Calculations, Statistics and Device Support). Whilst the Clarion language is powerful enough for some of these areas of functionality, is it worth reinventing the wheel?

For the first time Clarion developers now have a compiler that gives them access to the tools that developers using other languages have taken for granted. There is a wealth of programming libraries available that could significantly cut the development time of any sophisticated application, or give the Clarion developer access to new technology. The majority of the popular libraries are written in C however, many powerful C++, Modula-2 and Pascal libraries are also available.

Which Language Should I Use?

Since Clarion 3.0 started shipping, a common question from developers wishing to develop add-on Clarion libraries has been *"Which language should I use?"*. In order to determine the answer to this question it is useful to understand the relative strengths of the available languages:

Using Modula-2 with Clarion

Some Clarion programmers will find that Modula-2 source code has a familiarity about it, this may be because some Clarion features are based on Modula-2 features. The naming and parameter passing conventions used by the Clarion compiler are similar to those used in TopSpeed Modula-2, which can simplify interface coding. In addition, the language is very strict which means that novice programmers are likely to make fewer mistakes. The only downside is that if you make use of the Modula-2 run-time libraries, they must be linked, which may duplicate functionality in the Clarion run-time library and lead to some overhead.

Assembly Language

Assembly programming is the resort of programmers wishing to produce the smallest, fastest code possible. With modern optimising compilers, it is usual for the compiler to achieve better results than an assembler programmer, simply because the compiler is usually capable of balancing the various tradeoffs better than a human programmer. However, in some circumstances it is advantageous or even necessary to resort to assembler.

Strengths of Assembly Language

Assembly programming may result in better results for small functions and offers the best possible access to the machine address space.

Weaknesses of Assembly Language

Assembly code may have to be completely rewritten for each new environment or memory model. There are no checks to prevent you writing incorrect code. It can be difficult to write optimal assembler code for any but the smallest functions. A very good understanding of the hardware, operating system and calling conventions is required.

Using Assembly with Clarion

Assembly code may be useful for optimising small, frequently-called functions. However, a good understanding of the Clarion memory models and calling conventions is required. It is suggested that any assembly re-coding should be left as the final task in application development and only used to speed up critical functions which are causing a bottleneck. Clarion can be linked with code produced by Microsoft MASM and Borland TASM, alternatively the TopSpeed Assembler may be integrated into the Clarion environment.

Utilising a TopSpeed Compiler with the Clarion Development Environment

The Clarion Environment is capable of recognising and using any of the TopSpeed Compilers to compile source files in C, C++, Modula-2, Pascal or Assembler. In order to utilise a TopSpeed Compiler from the Clarion Environment the following steps must be taken:

- Install the TopSpeed Compiler, Environment, and optionally the Extender ToolKit, in the C:\TS sub directory. Ensure that the Extra Large, Overlay and DynaLink model libraries are installed.

- Ensure that the system path lists your C:\CLARION3 sub directory followed by C:\TS\XTD_SYS;C:\OS2_SYS;C:\SYS

Weaknesses of the C++ Language

C++ is not an easy language to learn. As a result, many C++ programmers use their C++ compiler as no more than a strict C compiler and do not make full use of C++'s major features.

Using C++ with Clarion

Because of its strictness and the fact that the C library is incorporated into the Clarion run-time library, the TopSpeed C++ compiler is a good choice of programming tool for use with Clarion. However the programmer needs to be aware of the fact that C++ normally performs name mangling that must be disabled where access from Clarion is required. In addition, although OOP features may be used within the C++ code to enhance the readability or usefulness of the code, Clarion has no access to OOP features and so a simple C 'glue-code' layer must be developed.

The Pascal Language

Pascal was originally designed as a simple language for teaching programming. Standard Pascal is intended to offer machine and operating system independent features and is consequently a comparatively limited language. As a result, many variations have been implemented which differ drastically from one-another.

Strengths of the Pascal Language

Pascal is a simple language offering array and string types, high-level string operations, and good compile-time and run-time error checking.

Weaknesses of the Pascal Language

Limited support for modern programming methods has lead to drastically differing extended implementations.

Using Pascal with Clarion

Standard Pascal provides few features that are not available within Clarion, as a result using Pascal and Clarion together is probably practical only where there is a significant investment in Pascal source code. It would in general be advisable to write new code in Clarion rather than Pascal.

The Modula-2 Language

Modula-2 was designed to supersede the Pascal programming language. It provides much better support for modern modular and object-oriented programming methods. Surprisingly, there are few commercial Modula-2 compilers and programming tools.

Strengths of the Modula-2 Language

Modula-2 is a simple but powerful language, offering good support for array types, good access to the machine's address space, including binding variables to specific addresses and facilities for writing interrupt handlers. TopSpeed Modula-2 provides OOP extensions and excellent compile-time and run-time diagnostics.

Weaknesses of the Modula-2 Language

There are few third-party libraries for Modula-2

Data Type Equivalence

The Clarion language defines the data types BYTE, SHORT, USHORT, LONG, ULONG, SREAL, REAL, and STRING which map fairly easily to C, C++, Pascal, and Modula- 2 equivalents. Clarion also defines DATE and TIME data types, and GROUP structures, which may be mapped to structured types in each language. CSTRING and PSTRING data types are specifically provided by Clarion to simplify interfacing with external functions using C or Pascal conventions.

The DECIMAL, PDECIMAL, BFLOAT4, and BFLOAT8 types have no direct equivalents in any of the TopSpeed Languages but values of these types may be converted to values of type REAL which may be passed to TopSpeed code.

The following table describes the data type equivalencies between the Clarion and TopSpeed languages. To simplify the development of external source modules, the ***Library API*** includes interface files that define equivalent data types for each of the TopSpeed Languages.

<table>
<tr><th>Clarion</th><th>C/C++</th><th>Modula-2</th><th>Pascal</th></tr>
<tr><td>BYTE</td><td>unsigned char</td><td>BYTE</td><td>BYTE</td></tr>
<tr><td>SHORT</td><td>short</td><td>INTEGER</td><td>INT16</td></tr>
<tr><td>USHORT</td><td>unsigned short</td><td>CARDINAL</td><td>WORD</td></tr>
<tr><td>LONG</td><td>long</td><td>LONGINT</td><td>INTEGER</td></tr>
<tr><td>ULONG</td><td>unsigned long</td><td>LONGCARD</td><td>INTEGER</td></tr>
<tr><td>SREAL</td><td>float</td><td>REAL</td><td>SHORTREAL</td></tr>
<tr><td>REAL</td><td>double</td><td>LONGREAL</td><td>REAL</td></tr>
<tr><td>DECIMAL
PDECIMAL
BFLOAT4
BFLOAT8</td><td colspan="3">Convert to REAL if passing to non-Clarion code, be aware of range limitations</td></tr>
<tr><td>STRING
CSTRING
PSTRING</td><td>char *
(See below)</td><td>ARRAY OF CHAR
(See below)</td><td>STRING
(See below)</td></tr>
<tr><td>DATE</td><td>passed as a long, use the following union to decode the fields:

typedef union {
long n;
struct {
unsigned char Day;
unsigned char Month;
unsigned short Year;
} s;
} CLADATE;</td><td>passed as a LONGINT use the following RECORD to decode the fields:

DATE = RECORD
CASE : BOOLEAN OF
| TRUE:
l : LONGINT;
ELSE
Day : BYTE;
Month : BYTE;
Year : CARDINAL;
END
END;</td><td>passed as an INTEGER use the following RECORD to decode the fields:

DATE = RECORD
CASE BOOLEAN OF
TRUE:
(n : INTEGER);
FALSE:
(Day : BYTE;
Month : BYTE;
Year : WORD);
END;</td></tr>
<tr><td>TIME</td><td>passed as a long, use the following union to decode the fields:

typedef union {
long n;
struct {
unsigned char Hund;
unsigned char Second;
unsigned char Minute;
unsigned char Hour;
} s;
} CLATIME;</td><td>passed as a LONG, use the following RECORD to decode the fields:

TIME = RECORD
CASE : BOOLEAN OF
| TRUE:
l : LONGINT;
ELSE
Hund : BYTE;
Second : BYTE;
Minute : BYTE;
Hour : BYTE;
END
END;</td><td>passed as an INTEGER, use the following record to decode the fields:

TIME = RECORD
CASE BOOLEAN OF
TRUE:
(n : INTEGER);
FALSE:
(Hund : BYTE;
Second : BYTE;
Minute : BYTE;
Hour : BYTE);
END;</td></tr>
<tr><td>*GROUP</td><td>struct *
(See below)</td><td>VAR RECORD
(See below)</td><td>VAR RECORD
(See below)</td></tr>
</table>

- Edit your CLARION.RED file to include the paths for the *.LIB, *.OBJ files and the source and interface files for your chosen language. These paths are specified in the TS.RED file for the TopSpeed Environment.

If the above steps are taken, the appropriate TopSpeed compiler will be invoked to compile any external source modules that are specified in the application's project file.

Interface considerations

When calling code from Clarion which has been written in another language, it is necessary to be aware of the way in which the two languages communicate. That is we need to be aware of:

- The way in which data and function names are handled by each compiler
- The way in which data of various types may be represented
- How parameters of different types are passed to called procedures
- How values may be returned to the calling procedure
- How global data may be accessed

Each of these issues is examined briefly below.

Naming Conventions

Different programming languages and compilers encode names differently when generating object code. The following table summarises briefly the naming conventions employed by the Clarion and TopSpeed Compilers

Language	CODE	DATA
Clarion	Names are upper-cased or as specified in the `NAME()` attribute.	Use the `NAME()` attribute for data that must be accessed from a different language.
C	Case sensitive, prepended by an underscore (_). Use: `#pragma name(prefix=>"", upper_case=>on)` to simulate Clarion naming conventions	

C++	Names are encoded with type information in a process called name-mangling. It is necessary to use the "Pascal" linkage specifier in order to generate Clarion compatible names:
	`extern "Pascal" {`
	`// Declarations for Clarion accessible`
	`// functions & data`
	`};`	

Modula-2	Case sensitive, of the form: `ModuleName$ProcName.` Use `(*# name(prefix=>windows) *)` to simulate Clarion naming conventions.	Case sensitive, of the form: `ModuleName@VarName.`
Pascal	Upper-cased, of the form: `UNITNAME$PROCNAME.` Use `(*# name(prefix=>windows) *)` to simulate Clarion naming conventions.	Upper-cased, of the form: `UNITNAME@VARNAME.`

```
DEFINITION MODULE TopSpeed;

(*# save, name(prefix=>windows) *)
(*# call(o_a_copy=>off) *)
  PROCEDURE TypeString1(VAR String: ARRAY OF CHAR);
(*# call(o_a_copy=>off, o_a_size=>off) *)
  PROCEDURE TypeString2(VAR String: ARRAY OF CHAR);
(*# restore *)
END TopSpeed.
```

The corresponding Pascal definition module would contain:

```
INTERFACE UNIT TopSpeed;

(*# save, name(prefix=>windows) *)
(*# call(s_copy=>off) *)
  PROCEDURE TypePString(VAR Str: STRING[high]);
(*# call(t_l_size=>off, t_l_copy=>off) *)
  PROCEDURE TypeString2(VAR Str);
(*# restore *)
END.
```

GROUP Types

Clarion GROUPs may be passed to TopSpeed language functions by address only (i.e. the Clarion code must declare the parameter as *GROUP). A group is roughly equivalent to a C or C++ struct, or a RECORD in Modula-2 or Pascal. When passed as a parameter to a procedure, GROUPs are normally passed as three parameters: first, a USHORT is passed which contains the size of the GROUP; second, the address of the GROUP structure; and third, the address of a buffer containing a type descriptor for the GROUP. The contents of the type descriptor are not discussed here and are subject to change in future versions of Clarion. Usually the RAW attribute is specified on the Clarion prototype which causes the compiler to pass only the address of the GROUP. A suitable struct or RECORD type must be defined in the TopSpeed code to map the values that are passed in using the GROUP.

Given the Clarion code:

```
                MAP; MODULE('TopSpeed')      ! TopSpeed code module
                  Proc(*GROUP), RAW          !   TopSpeed procedure
                . .                          ! .

Struct          GROUP, EXTERNAL              ! Struct type definition,
Byte              BYTE                       ! the EXTERNAL attribute
Short             SHORT                      ! ensures that no space is
Ushort            USHORT                     ! reserved, allowing Struct
Long              LONG                       ! to be used as a user
Ulong             ULONG                      ! defined type.  Struct may
Sreal             SREAL                      ! be used in a LIKE()
Real              REAL                       ! statement to reserve
Str               CSTRING(20)                ! space for a variable.
                .

Parm             LIKE(Struct), PRE(Parm)     ! Parm is of type Struct

  CODE
  ...
  Proc(Parm)
```

STRING Types

Clarion STRING, CSTRING and PSTRING variables are normally passed as two parameters: first, a USHORT is passed which contains the length of the string data buffer; second, the address of the buffer. The RAW attribute may be used in the Clarion procedure prototype to pass only the address of the string data to external functions. Clarion STRING variables are normally padded with spaces to the full length of the buffer, CSTRING variables contain no space padding and are terminated with a NULL character (0C), and PSTRING variables store the number of characters in the string as the first byte in the buffer.

When passing string data to C or C++ functions, it is suggested that you prototype the C/C++ procedures in the Clarion MAP as taking a *CSTRING parameter and use the RAW attribute to suppress the length parameter. This is consistent with the usual C conventions.

When passing string data to Modula-2 code, it is suggested that you prototype the Modula-2 procedures in the Clarion MAP as taking a *CSTRING parameter. Clarion CSTRING parameters are passed in the same manner as Modula-2 ARRAY OF CHAR parameters with the call(o_a_copy=>off) pragma in effect (the length and the address of the string are passed). If the RAW attribute is used on the Clarion prototype, then the Modula-2 call(o_a_size=>off, o_a_copy=>off) pragmas must both be used. Note that the Modula-2 HIGH() and LOW() operators may not be used when the call(o_a_size=>off) pragma is in effect, so it is necessary to test for the terminating NULL.

When passing string data to Pascal code, it is suggested that you prototype the Pascal procedures in the Clarion MAP as taking a *PSTRING parameter. Clarion PSTRING parameters are passed in the same manner as Pascal STRING parameters with the call(s_copy=>off) pragma in effect (the length and the address of the string are passed). If the RAW attribute is used, the Pascal procedure must declare the parameter as an anonymous VAR type (note that in this case it may be necessary to pass a CSTRING and determine the length of the string by testing for the terminating NULL).

Given the following Clarion prototypes:

```
MAP;  MODULE('TopSpeed')
  TypeString1(*CSTRING)          ! C or Modula-2 compatible function
  TypeString2(*CSTRING), RAW     ! C, Modula-2 & Pascal compatible
  TypePString(*PSTRING)          ! Pascal compatible function
. .
```

The equivalent C/C++ declarations would be:

```
#ifdef __cplusplus
extern "Pascal" {
#else
 #pragma save, name(prefix=>"", uppercase=>on)
#endif
extern void TypeString1(unsigned short Len, char *String);
extern void TypeString2(char *String);
#ifndef __cplusplus
 #pragma restore
#else
}
#endif
```

The corresponding Modula-2 definition module would contain:

Parameter Passing

Clarion, in common with many other languages, offers two distinct methods of passing parameters. Parameters may be 'passed by value,' or 'passed by address.'

Unless otherwise specified, parameters are 'passed by value,' a copy of a data is passed to the called procedure or function which may operate on the data, and may even change the value. Any changes made to the callees copy of the data will not affect the value of the data in the calling procedure. Parameters passed in this way are called 'value parameters.'

When a parameter is 'passed by address,' the address of the data is passed to the called procedure or function. Any changes made to the data in the callee will affect the data in the calling procedure. Parameters to be passed by address are specified by prefixing the parameters data type with an asterisk (*) in the Clarion prototype. Parameters passed in this way are called variable parameters.'

C or C++ functions that take variable parameters should be prototyped (in the C or C++ source) taking a parameter which is a pointer to the equivalent type, by affixing an asterisk (*) to the type name in the parameter list.

Modula-2 and Pascal procedures and functions that take variable parameters should be prototyped taking a VAR parameter of the appropriate type.

Returned Values

Only the simple types BYTE, SHORT, USHORT, LONG, ULONG, REAL and SREAL may be returned from TopSpeed functions to Clarion. Iin order to return STRING data to a Clarion function, it is usually necessary to pass a string parameter variable to the TopSpeed function and fill the string in the TopSpeed code. With Clarion 3.005 however, it is possible for C and C++ functions to return NULL terminated strings to Clarion if the Clarion prototype lists the return type as a CSTRING:

```
                MAP; MODULE('TopSpeed')
                  GetErrMsg(SHORT), CSTRING, NAME('_GetErrMsg')
                . .

ErrMsg           CSTRING(80)
ErrNum           SHORT

  CODE
  ...
  ErrMsg = GetErrMsg(ErrNum)
  TYPE('Error: ' & ErrMsg & '(' & ErrNum & ')' & @LF)
```

The appropriate C code would be:

```
#include <string.h>

char *GetErrMsg(short ErrNum)
{    return strerror(ErrNum);
}
```

The Corresponding C/C++ interface would contain:

```
typedef struct Struct {
    unsigned char   Byte;
    short           Short;
    unsigned short  Ushort;
    long            Long;
    unsigned long   Ulong;
    float           Sreal;
    double          Real;
    char            Str[20];
}                   Struct;

#ifdef __cplusplus
extern "Pascal" {
#endif
#pragma save, name(prefix=>"", uppercase=>on)
extern void Proc(Struct *Parm);
#pragma restore
#ifdef __cplusplus
}
#endif
```

The corresponding Modula-2 definition module would contain:

```
DEFINITION MODULE TopSpeed;

  TYPE
    Struct = RECORD
      Byte    : BYTE;
      Short   : INTEGER;
      Ushort  : CARDINAL;
      Long    : LONGINT;
      Ulong   : LONGCARD;
      Sreal   : REAL;
      Real    : LONGREAL;
      Str     : ARRAY[0..19] OF CHAR;
    END;

(*# save, name(prefix=>windows) *)
  PROCEDURE Proc(VAR Parm: Struct);
(*# restore *)
END TopSpeed.
```

The corresponding Pascal definition module would contain:

```
INTERFACE UNIT TopSpeed;

  TYPE
    Struct = RECORD
      Byte    : BYTE;
      Short   : INT16;
      Ushort  : WORD;
      Long    : INTEGER;
      Ulong   : INTEGER;
      Sreal   : SHORTREAL;
      Real    : REAL;
      Str     : ARRAY[0..19] OF CHAR;
    END;

(*# save, name(prefix=>windows) *)
  PROCEDURE Proc(VAR Parm: Struct);
(*# restore *)
END.
```

```
(*#restore*)
END TopSpeed;
```

Pascal is similar to Modula-2 in the organisation and behaviour of its units, the following Pascal definition units serve the same purpose as the Modula-2 code above:

```
(*# module(implementation=>off, init_code=>off) *)
INTERFACE UNIT clamod;
(*# save, name(prefix=>windows) *)
  VAR
    ErrNum  : INT16;
(*#restore*)
END.

INTERFACE UNIT TopSpeed;
(*# save, name(prefix=>windows) *)
  VAR
    ErrMsg  : ARRAY[0..79] OF CHAR;
(*#restore*)
END;
```

Module Initialisation

Many Modula-2 and Pascal modules contain an initialisation section which is executed prior to executing the main program module. If you are calling Modula-2 or Pascal procedures in modules that require initialisation, it is essential to ensure that each module is initialised prior to use. The InitModules procedure is provided to facilitate Modula-2 and Pascal module initialisation:

```
    MAP; MODULE('InitMod')
      InitModules(LONG, LONG, <LONG>, <LONG>, <LONG>, <LONG>), |
        C, RAW, NAME('_InitModules');
    . .

MyModule    BYTE, DIM(0), EXTERNAL, NAME('MyModule$')

  CODE
  InitModules(ADDRESS(MyModule), 0)
```

InitModules must be the first call from the main program module of any Clarion program that uses Modula-2 or Pascal modules that have initialisation code. Up to 5 modules may be specified, the last parameter passed in the list must be a 0. The parameters are the addresses of the initialisation records generated by the compiler for each Pascal or Modula-2 module (if the module(init_code=>of) pragma is not specified). You must declare an initialisation record for any Pascal or Modula-2 modules you write. The name of the initialisation record that appears in the NAME() attribute is the name of the module with a '$' affixed.

Note that calling InitModules() to initialise a specific Modula-2 or Pascal module automatically initialises any imported modules.

Third-Party Compilers

Clarion 3.0 uses the TopSpeed object code generator, so it uses the same efficient register-based parameter passing mechanism employed by all TopSpeed languages. Clarion language features

→ 2 pages

Accessing Global Data

Global data may be accessed from program modules written in any language, provided the correct naming conventions are employed and the declared type of the variable matches the definition. Because the naming conventions used by Clarion for data are a little more complex than those used for function and procedure names, it is suggested that the NAME() attribute be applied to data names that must be accessed from code written in another language.

Given the following Clarion code:

```
ErrMsg       CSTRING(80), NAME('ERRMSG')          ! Clarion data
ErrNum       SHORT, EXTERNAL, NAME('ERRNUM')      ! External data

  CODE
  ...
  IF ErrNum <> 0 THEN
    TYPE('Error: ' & ErrMsg & '(' & ErrNum & ')' & @LF)
  .
```

The corresponding C/C++ declarations would be:

```
#pragma save, name(prefix=>"", upper_case=>on)
extern short    ErrNum;             /* Defined in Clarion code */
char            ErrMsg[80];         /* C data                  */
#pragma restore
```

The example becomes a little more complex for Modula-2. Typically, a definition module is created which will contain declarations of all data and procedures defined in the Clarion code which must be accessed from the Modula-2 code. Any Modula-2 procedures and data which must be accessed from Clarion must be also be declared in some definition module, since Modula-2 names are only globally available if they are exported. In the following example code the definition module clamod corresponds to a Clarion source module and provides Modula-2 code with access to the code and data within the Clarion module. Note that the module(implementation=>off) pragma prevents the Modula-2 compiler from looking for a Modula-2 implementation module associated with the definition module. The module(init_code=>off) prevents the Modula-2 compiler from generating code to initialise the Clarion module.

```
(*# module(implementation=>off, init_code=>off) *)
DEFINITION MODULE clamod;

(*# save, name(prefix=>windows) *)
  VAR
    ErrNum   : INTEGER;
(*#restore*)
END clamod.
```

The TopSpeed definition module corresponds to a source module written in TopSpeed Modula-2 that provides procedures and data that will be accessed from Clarion code. Note that all procedures and data which may be used in Clarion code must appear in some definition module.

```
DEFINITION MODULE TopSpeed;

(*# save, name(prefix=>windows) *)
  VAR
    ErrMsg   : ARRAY[0..79] OF CHAR;
```

Summary

- C++, Modula-2 or Pascal compilers may be used to give your Clarion applications access to low-level functions or third-party libraries. C and C++ offer better integration with the Clarion run-time library. Modula-2 will seem familiar to Clarion programmers and is a strict language offering robustness and security.

- Most Clarion data types are easily mapped to standard types in other languages which makes it easy for routines written in another language to operate on Clarion data.

- TopSpeed compilers can be seamlessly integrated into the Clarion environment, use compatible calling conventions and can use Clarion naming conventions. Clarion can use calling conventions and generate names compatible with third-party compilers though TopSpeed compilers are recommended.

- For simplicity and efficiency, STRING and GROUP parameters should usually be passed by address with the RAW attribute to ensure only the address is passed.

- If a function takes a pointer or var parameter, the corresponding parameter in the Clarion prototype for that function should be declared as 'passed by address' by prefixing the data type with an asterisk (*). If a function takes a pointer to a structure or a var record as a parameter, the corresponding parameter in the Clarion prototype should be *GROUP.

- Use the interface (*.H, *.DEF, *.ITF) files as a template for developing a Clarion interface to code in another language.

- Use the NAME attribute on the Clarion declaration for external data in a module written in another language.

- Don't try to access C++ objects of class type or their members from your Clarion code.

Where you can obtain more information

Additional information on the topics discussed is available from the following sources:

Library API: This new Clarion product documents many of the issues discussed in this article and contains useful example code, a Clarion interface to useful portions of the C library and Clarion internal run-time library.

Advanced Programming Guide: This manual ships with the TopSpeed TechKit which also contains the TopSpeed Assembler and contains a good discussion of the calling conventions used by the Clarion and TopSpeed Compilers and documents considerations when implementing assembly code for use with the Overlay and DynaLink memory models.

TopSpeed Developer's Guide: This manual ships with the TopSpeed Environment and provides a good overview of the project system language, standard memory models, multi-language programming, and TopSpeed Compiler pragmas.

CPP_CLA.ZIP, M2_CLA.ZIP & PAS_CLA.ZIP: These are available from Clarionet and the Clarion forum on CompuServe, they contain documentation on interfacing each of the TopSpeed Languages with Clarion.

3rd Party

are provided to facilitate calling code written for use with third-party compilers, however, there are a number of restrictions.

The calling conventions used by the Clarion compiler to call external procedures must match the calling conventions used by the compiler generating the code. Most third-party compilers don't use Clarion-compatible parameter passing conventions, but do provide 'standard' C and Pascal stack based parameter passing mechanisms. Clarion 3.0 provides the C and PASCAL procedure prototype attributes to specify stack-based parameter passing.

Most third-party C and C++ compilers use the 'cdecl' calling convention where parameters are pushed onto the stack from right to left (as read from the parameter list). The Clarion C procedure attribute may be used to specify that a procedure must be called using this convention.

Many C and C++ compilers also offer a 'pascal' calling convention where parameters are pushed left to right from the parameter list. This convention is also utilised by most other languages on the PC. The Clarion PASCAL procedure attribute may be used to specify that a procedure must be called using this convention.

There is no standard of compatibility between compilers regarding the way in which floating point values are returned from a function. For example, Microsoft C returns floating-point values in a global variable while Borland C returns them on the stack. Therefore, any functions from non-TopSpeed compilers which must reference floating point values and modify them should receive them 'passed by address' and directly modify the value. Do not have the function return the value.

Most compilers generate object code which calls 'helper functions' to perform common tasks such as stack overflow checking, long division or multiplication and pointer arithmetic. The Clarion 3.0 run-time library provides helper functions which are compatible with Microsoft and Borland helper functions for use when linking code compiled with these compilers.

The biggest obstacle to using third-party compiled code with Clarion 3.0 are the supported memory models. Clarion 3.0 offers a choice of Extra Large, Overlay, DynaLink, Extended & ExtenDLL memory models. Each of these memory models carry a specific set of conventions and restrictions. Code that is linked in Extended or ExtenDLL model programs must run in protected mode which means that there cannot be any physical memory addressing and certain forms of pointer manipulation are outlawed. In addition, segment registers may only contain valid selector values. The overlay manager used in Overlay and DynaLink model programs requires that procedures create a valid stack frame, that bp is valid at the point of a far call, and imposes other restrictions due to the fact that segments may be relocated during the running of the program. In general, code compiled with a third-party compiler will not observe the restrictions imposed for these memory models. However, the Large and Huge memory models supported by many third-party compilers generally produces code that may be used in Extra Large model programs.

In spite of the restrictions there are a wide range of libraries which can be used with Clarion. Many of these are available with C source code which may be recompiled with TopSpeed C for use in Clarion programs.