

Mixed Language Linking

Nick Maclaren

Computing Service

nmm1@cam.ac.uk, ext. 34761

February 2007

Overview of Course

Mainly the principles, and where to look

Details vary with system, compiler and versions
Will describe how to select a feasible task

Firstly, what language mixing is possible
Secondly, some other practical issues
Thirdly, **Fortran** and **C** in more detail

Beyond the Course

Email escience-support@ucs for advice

- This is an area where experience really helps

Some references given in context

Look at the Programmer's Guides or similar

For **both** compilers, **and** your system

Few generic documents are worth bothering with

Rule Number 1

- **KISS** – Keep It Simple, Stupid!

If you try to be clever,
you **WILL** shoot yourself in the foot

Simple use very often works, easily

- Even so, there are **NO** safe recipes

This course is about understanding the issue

Why Link Multiple Languages?

- Usually to get access to system interfaces
Very rarely needed in **Python**, **Perl** etc.
Functions are typically very simple

Later, will give **Fortran** to **C** examples

Get high-precision (microsecond) timestamp

Get environment variable, if not in library

Extend Language Features

- Usually just an extra primitive
Like above, such functions are usually simple
- Commonly, using **C** for special I/O
This is how **MPI** etc. are implemented
- Beyond that is typically task for experts
E.g. writing floating-point emulator for Python

Joining Applications Together

- Strongly advise you to avoid this
Always tricky – and can be fiendish
Better to keep them separate processes

[http://www-uxsup.csx.cam.ac.uk/courses/...
.../MultiApplics](http://www-uxsup.csx.cam.ac.uk/courses/.../MultiApplics)

- May need to write special I/O functions
But that is generally easier (see above)!

Language Combinations

- Only some combinations are feasible
Some others are possible with some compilers
Question of how much skill & effort you need
 - Will describe only plausible combinations
Ones **NOT** assuming advanced hacking skills
- Even for these, can be very compiler-dependent
- Portable mixed-language linking can be hard

Masters and Servants

- Some languages insist on being master
Others must follow the **master's** conventions
- This is not always clear in documentation
- There must always be a **single** master
Even for the easy **C + Fortran 77** case

Need for run-time initialisation/termination

Platform mechanisms are now very rare

Sometimes exist when using just one vendor

Microsoft (1)

Don't use them myself, but here are plans
Amusingly, repeat of late 1980s **IBM CEE** ones

Used to be more-or-less assembler interfaces
With **Visual Basic** as primary language

Moving to **CLI** (\equiv **IBM CEE**) in **.NET**
With **C#** as primary interface language
Plus **Visual Basic**, **J#**, **C++/CLI** and others
NONE of which match external standards

Microsoft (2)

IBM CEE failed to take over the world
Partly for extraneous reasons (workstations)
But will Microsoft succeed this time?

Principles of what I say applies to both
Some details apply only to non-Microsoft

- Situation won't settle down before 2010
- I can advise how to minimise problems
But not within scope of this course

Example Masters

Anything with fancy memory management

- Two garbage collectors is **BAD** news

Exception handling, some I/O, etc. are similar

Python, Fortran 90, C++, Java, Perl, C#, Tcl/Tk,
MATLAB, Maple, Mathematica, Excel, . . .

In some cases, can be used for servant code

- Needs lots of experience and skill
- Always very implementation-dependent

Servants

- Easier to list these, as only a few
May be a few other, rarer languages

C90, C99, Fortran 77, almost always
C++, Fortran 90 can be used with care
And, of course, suitable assemblers

Microsoft C# was described earlier

Most other systems use C for interfaces

- Regard most libraries as simple C code

Code Generation (1)

Masters may have a generation option

The **MATLAB Compiler** is an example

<http://www.mathworks.com/products/compiler/>

Also **Mathematica MathCode C++/F90**

[On **Microsoft** systems only]

<http://www.wolfram.com/products/...>

[.../applications/{mathcode,mathcodef90}/](http://www.wolfram.com/products/.../applications/{mathcode,mathcodef90}/)

- Such code may need extensive editing

Code Generation (2)

- May generate 'core' code only, no interfaces
Or interfaces for wrong target language
Need to add them manually and painfully

- Problem if may need to keep updated
It can be done, but needs a **LOT** of skill

Or converse problem, described under **SWIG**

Combinations

Will describe most important combinations
And give indication of how to proceed

- If I don't mention it, investigate first
May be an infeasible combination
Or I may simply not have thought of it

For infeasible combinations, look at:

[http://www-uxsup.csx.cam.ac.uk/courses/...
.../MultiApplics](http://www-uxsup.csx.cam.ac.uk/courses/.../MultiApplics)

The Trivial Cases

C++ is almost a superset of C!

Fortran 2003 is a superset of Fortran 77

- Easiest to use ‘higher’ compiler for both

Can often mix code from different compilers
But see later about issues with that

The Simple Combinations

Using a higher language (Python etc.)
Its implementation language as servant

- Nowadays, latter is almost always C
Rarely, may be C++ – see later
On Microsoft, may be Visual Basic or C#

Don't underestimate the learning needed

- Errors in C etc. often cause CHAOS

Fortran and C

Nowadays, Fortran is the higher language
Its library is almost always based on C

- Treat it as master, link using Fortran
Rarely will need to fiddle libraries etc.
Usually easiest to use Fortran main program
Not needed for simple Fortran 77 procedures
- Will come back to this at length later

SWIG (1)

A semi-generic C/C++ interface builder
<http://www.swig.org/>

Not used it for real, but it looks sound
Also under active development by a team

The Web pages are rather full of hype
The manual is a LOT better – looks OK

SWIG (2)

- Generated code is not maintainable
Generator is compiler – **NOT** intelligent
Not a highly optimising compiler, either

Lots of unnecessary code and actions

- You should maintain original source only
Use **SWIG** as black-box pre-processor

- Universal problem with generic converters
Exact converse of one mentioned above

SWIG (3)

It lowers the effort, but that is all
Trivial uses are trivial, but . . .

- You will **HAVE** to customise interfaces
I didn't seriously try out such aspects

More detail about the underlying problem later

The Tradeoff

- Complete generator is much easier to use
- Much better if need to keep source updated
- Limited use for generating ‘proper’ code

- Core-only generator much more effort
- Much easier for generating ‘proper’ code
- Pain in the neck if source keeps changing

Manual conversion is like core-only generator

Python and Java

May be a fully documented mechanism and API
All (!) you have to do is to obey its rules

<http://docs.python.org/ext/>

<http://java.sun.com/j2se/1.4.2/docs/guide/jni/>

I am currently doing this with **Python**

No major problems for even advanced work

- **MUST** use its recommended conventions

Need discipline for practical debugging

MATLAB

Web pages have information and examples

[http://www.mathworks.com/access/helpdesk/...
.../help/techdoc/matlab_external/](http://www.mathworks.com/access/helpdesk/.../help/techdoc/matlab_external/)

MATLAB can call **C** and **Fortran**

Can start and use **MATLAB** from those, too

Mathematica

Mathematica MathLink allows calling C and C++

<http://support.wolfram.com/mathematica/mathlink/>

Mathematica J/Link allows linking to Java

<http://www.wolfram.com/solutions/mathlink/>

There is also some .NET integration

[http://documents.wolfram.com/mathematica/...
.../Add-onsLinks/NETLink/](http://documents.wolfram.com/mathematica/.../Add-onsLinks/NETLink/)

Tcl/Tk and Perl

Tcl/Tk has a documented interface library

<http://www.tcl.tk/man/tcl8.4/>

A zillion (unmaintained?) Tcl/C++ interfaces

Would guess that using SWIG is better

Perl was a nightmare, even for hackers

There is now a book that maps the minefield

[Extending and Embedding Perl,](#)

Jenness & Cozens

Others

Maple to C is **not** well documented

Oracle and similar are also possible

Particular Issues

This is a miscellaneous set of tips

- **NOT** a complete checklist of problems

The restrictions are not usually 'hard'

- Bypassing them may need advanced hacking

Please ask if you have problems

Compiler Compatibility

Very much like **Fortran** and **C** issues

- Two **C** compilers need not be compatible

Anywhere I say **usually** is a risk, and more

- But there are problems beyond data passing

Don't trust versions to be compatible

Not just compiler, but libraries, too

Intel has a particularly poor record

Basic Interfaces

At bottom level, may use different registers

- Only assembler programmers can handle that

Assume basic calling sequence is compatible

- Check for documented compiler options

Make sure both are in 32- or 64-bit mode!

Make sure **IEEE 754** modes are compatible

Name munging (**Fortran** and **C++**) may vary

Very often options to control that

Compilation and Linking

Compile all servant code without linking

- Link using master compiler or script

May need extra libraries or to hack script

Look at documentation first but, if not:

Usually option to display command expansion

`-v`, `-#`, `-dryrun` etc.

Run for servant and select libraries/options

Add carefully to master link command

Termination etc.

- Start and terminate in master language
Can be done other way, but gets much trickier

Don't rely on the servant language cleaning up

- Close all servant I/O streams before exit
Also free all space, if continuing in master

- Don't **longjmp** across other languages
Same applies to **C++** exceptions etc.

Name Clashes

Avoid **Fortran** and **C** externals of same name
And that means even when case is ignored

- Including **ALL** names in **EITHER** library

For example, **Fortran SQRT** \neq **C sqrt**

- Name munging only sometimes protects you
Internals, statics etc. are not a problem

Can get name clashes within libraries

All solutions to that are advanced hacking

I/O

Can usually write to standard output/error

- **ALWAYS** call **flush** after doing so

Fortran 2003 has a **FLUSH** subroutine

Almost all **Fortran** systems have one

Don't do any other form of I/O mixing

Don't reposition standard output/error

Can often be done, but is minefield

C and Fortran

What many people assume by mixed-language

Will go into some details of simpler cases

Will **NOT** go into the arcane details

- Please ask if you have or hit problems

Data Model

All bets off for **fancy** interfaces

Must read API specification or language guide

Or reverse engineer implementation's interface

Basic interfaces are semi-portable

Used for most **Fortran** and **C** interfaces

Will start with describing interface design

C and C++ Args and Results

Arguments are by value, like a sort of structure

- Alignment rules may be very different

Structures etc. **usually** passed inline

float **usually** promoted to **double**

char, **short** **usually** promoted to **int**

Results are also returned by value

Similar, even less defined, promotion rules

Structures returned in several different ways

C and C++ Recommendation

Args and results use **int**, **double** and pointers

⇒ no **complex** results

Relevant only to **C99** and **C++**, of course

Pointers to **char**, **short**, **float**, **complex** are fine

No problem with any type of array argument

Or returning pointer to anything

Fortran Arguments and Results

Almost always passed as pointers

- May be pointer to cell containing a copy

CHARACTER lengths **usually** elsewhere

- Must use fixed, known-length strings

Occasionally may be extra argument

- Stick to **INTEGER** and **D.P.** results

CHARACTER and **COMPLEX** are problems

Fortran External Names

Usually lower-cased and suffixed with ‘_’

Many other rules exist – use `nm` to detect

- Sometimes options, otherwise fix up in C

Fortran external procedures \equiv C extern functions

Fortran COMMON \approx C extern struct

Do not assume padding rules are the same

- Avoid unaligned data like the plague

Fortran COMMON and C

```
REAL(KIND=DP) :: A(5,10,3)
INTEGER :: N(20)
COMPLEX(KIND=DP) :: C(5,10)
COMMON /FRED/ A, C, N
```

```
extern struct {
    double a[3][10][5];    ← Note!
    complex double c[10][5];
    int n[20];
} fred_;
```

Fortran Calls and C (1)

```
SUBROUTINE FRED (A, B, C)  
REAL(KIND=DP) :: A  
INTEGER :: B  
COMPLEX(KIND=DP) :: C
```

```
extern void fred_ (double *a,  
                 int *b,   complex double *c);
```

Fortran Calls and C (2)

```
INTEGER FUNCTION FRED (A, B, C)  
DOUBLE PRECISION :: A(5,10,3)  
CHARACTER(LEN=15) :: B(15)  
INTEGER :: C(20)
```

```
extern int fred_  
    (double a[3][10][5],  
     char b[15], int c[20]);
```

C Datatypes

What most compilers do, not what is required

- The basic types everything is mapped onto

Anything not mentioned likely to be a trap

C99 introduced a LOT of pitfalls

- Most systems don't use them by default

Integer Types

Almost always, **short** is 16-bit, **int** is 32-bit
long is 32- or 64-bit, depending on system
unsigned affects only arithmetic, not data

Only one representation – twos' complement

- Endianness does not vary within a system
 - Almost every integer mapped to one of those
- May not be the same mapping for every compiler
- Ask if you want guidelines on what is likely

Floating Types

`float` & `double` are 32- & 64-bit **IEEE 754**

- Don't use options selecting **Intel** format

Watch out for hard vs soft underflow

- **MUST** use consistently through program

[http://www-uxsup.csx.cam.ac.uk/courses/...
.../Arithmetic](http://www-uxsup.csx.cam.ac.uk/courses/.../Arithmetic)

Pointers

- Pointers are address of first byte
No information on type or length
- Arrays are pointer to first element
Always contiguous (i.e. no gaps)
- **LAST** subscript varies fastest

Function pointers are just addresses, too

C99 Arrays

C99 now comparable to Fortran 77

Argument array bounds can be variable

```
SUBROUTINE FRED (L, M, N, A)
```

```
INTEGER :: L, M, N
```

```
DOUBLE PRECISION :: A(N,M,L)
```

```
extern void fred_ (  
    int *l, *m, *n,  
    double a[*l][*m][*n]);
```

Structures

Structures are in order, with natural alignment

Sometimes there are options to vary this

- Avoid unaligned data if at all possible

`struct{int a; double d;}` will be:

Bytes 0–3:	a	OR	a
Bytes 4–7:	unused		d'
Bytes 8–11:	d'		d''
Bytes 12–15:	d''		

Other C Datatypes

char is generally 8-bit **ASCII**

- Strings are NUL-terminated arrays of **char**
As expected, are stored as pointers to **char**
Length is passed separately or scanned for

complex is structure: **real**, **imaginary**

Can usually be treated as array of length 2

union is whichever member is selected

Some systems have other types, but rarely

C++ Classes

Simplest **class** is like **struct**

- Static members are omitted
- Not if virtual functions, virtual base classes
- Nor if it uses **public** or **private**

Class data, member functions passed implicitly

- Class of object known at compile time

Fortran Datatypes

INTEGER \approx **C** int

Sometimes an option to use 64 bits for it

REAL and **D.P.** \equiv **C** float and double

Both can be varied with **KIND**

In memory, **COMPLEX** \approx **C99/C++** complex

- Argument and result handling may differ
- Default (not recommended) is **REAL**

Fortran CHARACTER

Generally 8-bit **ASCII**, like **C**

- An extra dimension of array, varying fastest
- **NO** termination, NUL or otherwise

Length is explicit in most declarations

- Length is **implicit** for arguments

See above for use in arguments

- Don't return **CHARACTER** results

Fortran Arrays

Fortran 77 arrays \approx C arrays, transposed
I.e. explicit-shape and assumed-size arrays

`DIMENSION A(5,10,3), B(20,*)`

- **FIRST** subscript varies fastest
- Regard other sorts of array as fancy types
Allocatable, assumed-shape etc.

`DIMENSION C(5:), D(:,,:)`

`REAL, ALLOCATABLE :: E(20), F(:)`

Other Fortran Datatypes

Derived types are fancy – but see below

- Regard pointers as fancy types, too
May be **fat pointers** – not just addresses

Procedures are just addresses, like **C**

Fortran I/O units are **NOT POSIX** descriptors

Fortran 2003

It specifies some limited interoperability
Not yet generally available, but coming

Simple derived types can match **struct**
No pointers, or allocatable objects
Several, more obscure, restrictions

In theory, need to declare as **BIND**
Definitely need to for external variables

High-Precision Timestamp

```
/* Return high-precision timestamp. */  
#include <stddef.h>  
#include <sys/time.h>  
double gettime_ (void) {  
    struct timeval timer;  
    if (gettimeofday(&timer, NULL))  
        return -1.0;  
    return timer.tv_sec +  
        1.0e-6*timer.tv_usec;  
}
```

Using the Timestamp

```
program main
  real(kind=kind(0.0d0)), &
    external :: gettime
  write (*, '(f20.6)') gettime()
end program main
```

Environment Variable (1)

```
#include <string.h>
#include <stdlib.h>
int getenvir_ (int *len, char *text) {
    char *ptr;
    if ((ptr = memchr(text, ' ', *len))
        == NULL) return -2;
    *ptr = '\0';
    if ((ptr = getenv(text)) == NULL)
        return -1;
```

Environment Variable (2)

```
if (strlen(ptr) < *len) {  
    memset(text, ' ', *len);  
    memcpy(text, ptr, strlen(ptr));  
    return 0;  
} else {  
    memcpy(text, ptr, *len);  
    return 1;  
}  
}
```

Using Environment Variable

```
program main
  integer, external :: getenvir
  integer :: n
  character(len=15) :: c
  read (*,'(a15)') c
  n = getenvir(15,c)
  write (*,*) n, c
end program main
```

Rule Number 1

- **KISS** – Keep It Simple, Stupid!

Simple use very often works, easily
Ask for advice if you have problems