KRoC — Calling C Functions from occam

David C. Wood, Computing Laboratory, University of Kent at Canterbury
D.C.Wood@ukc.ac.uk

1. Introduction

This document describes the mechanism for calling C functions from occam running on x86 Linux KRoC.

2. The Problem

The parameter-passing convention used by occam on the transputer, and hence by KRoC, is in general different from that used by C (and other languages). The mechanism described here for converting between these conventions is designed so that KRoC needs to know very little about those used by C; as far as possible, this is left to the local C compiler.

3. Mechanism

The parameters of an occam PROC are passed in consecutive locations at the start of the occam workspace of the process making the call. To access a C function, these parameters have to be re-organised according to the conventions of the C compiler. Fortunately, this can be done in a fairly transparent way, using the C compiler itself, so that neither KRoC nor the programmer need to know those conventions.

Suppose we wish to call a C function with the prototype:

```
int foo_bar (float this, float that);
```

then we need an occam 'prototype' to make the call from occam. Although we could model this with an occam FUNCTION, functions in C cannot be trusted to be free from side-effect. Therefore, we model all C 'functions' by occam PROCs that include an extra result parameter (or parameters). The occam 'prototype' is introduced using the #PRAGMA EXTERNAL mechanism, since its implementation is 'external' to the occam system. For the above C function, we need:

```
#PRAGMA EXTERNAL "PROC C.foo.bar (INT result, VAL REAL32 this, that) = 0"
```

The initial 'C.' is a naming convention used by KRoC so that it can generate the special calling sequence for a C function. Please note that this means that KRoC occam programs must not declare *normal* PROCs (or FUNCTIONs) with names starting 'C.'

The zero at the end of this #PRAGMA declaration is the number of words of occam workspace needed to execute the PROC. However, C functions create new stack frames for their workspace and these do not live in the occam world.

The KRoC system cannot compile a call of C.foo.bar directly into a call of the C function *foo_bar*, since it doesn't know the full parameter passing conventions required by the C compiler. Instead, KRoC compiles it into a call of a C 'interface function' (with the name *__foo_bar*) and which the programmer has to supply. Its name is derived from the occam #PRAGMA name, substituting '__' (double underscore) for the opening 'C.' and changing any other dots into underscores.

KRoC C interface functions always have the same signature for their prototypes: they return *void* and take only one parameter – a pointer to the occam workspace where the actual parameters have been set up. For our example, the interface prototype is:

```
void __foo_bar (word w[]);
```

where *word* is a C data type corresponding to an occam INT. For the x86 Linux system, this is:

```
typedef int word;
```

Note that the occam workspace is simply represented as an array of *word*s. Note also that KRoC has to know just a little about the parameter passing convention required by C in order to pass this single pointer.

It is the responsibility of the C interface function to make the actual call of the C target function, supplying it with parameters extracted from the occam workspace. To write this function, the programmer needs to know how occam has placed the arguments in its workspace and how to convert between occam data types and parameter modes and those in C.

This is not as hard as it sounds. For our example, the interface function is:

```
void __foo_bar (word w[]) {
  *INT(w[0]) = foo_bar (VAL_REAL32(w[1]), VAL_REAL32(w[2]));
}
```

Comparing this with the occam `C.foo.bar`, we see that occam parameters appear on *w[]* in ascending order from element zero, with the leftmost parameter at *w[0]*. All occam parameters occupy just one word, except for open arrays which are passed in two words (a pointer to their start and their actual size). occam data structures occupying more than one word are always passed by pointer (even if they are VAL parameters).

To convert between the basic occam data types and parameter modes and those in C, a set of macros (e.g. *INT*, *VAL_REAL32*, ...) is provided. These are needed to keep the C type-checker happy and generate no run-time code. A full list is given in Section 4.

If the local C compiler provides a mechanism for calling other languages (e.g. Fortran), there is nothing to stop these C interface functions directly making such calls. In this way, access to routines in other languages is automatically inherited by KRoC occam.

Similarly, the interface function may be written in native assembler, allowing direct access to the x86 instructions from KRoC. [For information, the single parameter to this function (which points to the occam workspace and, hence, the occam parameters) is passed on the top of the stack. However, programmers will need to obey the Intel 386/486/Pentium calling Convetions.

4. MAPPING BETWEEN THE BASIC occam AND C TYPES

Mappings between the basic occam and C types and given in Figure 1. Equivalent headings between an example occam PROC (which would appear in a #PRAGMA EXTERNAL declaration) and the target C function are given in Figure 2. The interface function that connects the occam call with the target C function is given in Figure 3. The interface function uses a number of C macros which are defined in the file `callc\callc.h` (in the KRoC release directory). For information, a listing of these is given in Figure 4. Use of these macros is not compulsory, but they simplify the writing of interface functions and make them portable to KRoC systems running on other processors.

occam	C
BYTE	*char*
BOOL	*char*
INT16	*short int*
INT	*int*
INT32	*int*
INT64	*long long int*
REAL32	*float*
REAL64	*double*

Figure 1: Mapping between occam and C basic types

occam PROC heading	C function prototype

`PROC C.basic (VAL BYTE v.c,`
`  VAL BOOL v.b,`
`  VAL INT16 v.s,`
`  VAL INT v.i,`
`  VAL INT32 v.j,`
`  VAL INT64 v.l,`
`  VAL REAL32 v.f,`
`  VAL REAL64 v.d,`
`  BYTE c,`
`  BOOL b,`
`  INT16 s,`
`  INT i,`
`  INT32 j,`
`  INT64 l,`
`  REAL32 f,`
`  REAL64 d)`
	`void basic (char v_c,`
	`  char v_b,`
	`  short int v_s,`
	`  int v_i,`
	`  int v_j,`
	`  long long int v_l,`
	`  float v_f,`
	`  double v_d,`
	`  char *c,`
	`  char *b,`
	`  int *s,`
	`  int *i,`
	`  int *j,`
	`  long long int *l,`
	`  float *f,`
	`  double *d);`

Figure 2: Equivalent occam and C prototypes (basic types)

```
void __basic (int w[]) {
  basic
   (VAL_BYTE   (w[0]),
    VAL_BOOL   (w[1]),
    VAL_INT16  (w[2]),
    VAL_INT    (w[3]),
    VAL_INT32  (w[4]),
    VAL_INT64  (w[5]),
    VAL_REAL32 (w[6]),
    VAL_REAL64 (w[7]),
    BYTE       (w[8]),
    BOOL       (w[9]),
    INT16      (w[10]),
    INT        (w[11]),
    INT32      (w[12]),
    INT64      (w[13]),
    REAL32     (w[14]),
    REAL64     (w[15]));
}
```

Figure 3: Interface function (basic types)

Summary: when an occam process makes a call to `C.basic`, KRoC implements this as a call to the interface function *`__basic`*, which makes the actual call to the target function *`basic`*. The interface function effectively maps the parameters from the structure set up by occam to whatever is needed by C.

```
typedef int word;

#define VAL_BYTE(w)   (*(char *)&(w))
#define BYTE(w)       ((char *)(w))

#define VAL_BOOL(w)   (*(char *)&(w))
#define BOOL(w)       ((char *)(w))

#define VAL_INT16(w)  (*(short int *)&(w))
#define INT16(w)      ((short int *)(w))

#define VAL_INT(w)    (w)
#define INT(w)        ((int *)(w))

#define VAL_INT32(w)  (w)
#define INT32(w)      ((int *)(w))

#define VAL_INT64(w)  (*(long long int *)(w))
#define INT64(w)      ((long long int *)(w))

#define VAL_REAL32(w) (*(float *)&(w))
#define REAL32(w)     ((float *)(w))

#define VAL_REAL64(w) (*(double *)(w))
#define REAL64(w)     ((double *)(w))
```

Figure 4: Interface macros (basic types)

5. MAPPING BETWEEN ARRAY TYPES

When an occam formal parameter array is declared as VAL, the corresponding C formal parameter should be declared as `const`. However, both occam and C pass arrays by reference, regardless of VAL or `const` decoration. In fact, C does not distinguish between arrays and pointers, so that `int p[]` and `int *p` are equivalent as parameters – both are passed simply as addresses. Hence, in the interface function, arrays can be treated as reference parameters of their base type and we just use those macros whose names are the basic occam types.

It is important to know that an occam formal *open* array parameter (e.g. `[]REAL64`, `[][]BYTE`) is supplied as the pointer to the start of the array, followed by the sizes for all missing dimensions. A formal *sized* array parameter (e.g. `[42]REAL64`, `[768][1024]BYTE`) is just supplied as the pointer to the start of the array.

Figures 5 and 6 show equivalent occam and C headers for an example containing various array parameters. Not all basic types are illustrated. For others, simply substitute the occam names in the occam header and interface function macros and match the target function type according to the table in Figure 1.

6. MAPPING BETWEEN OTHER TYPES

There are no equivalents in occam to C unsigned `int` types. They can be passed as the corresponding signed types; but if there is a need to do arithmetic on them in the occam world, the unchecked operators `PLUS` and `MINUS` should be used.

`RECORD`s in occam correspond to `struct`s in C. However, the occam compiler may layout `RECORD` fields in a different order to the way the C compiler lays out its `struct` fields. For example, occam `RECORD` fields are packed in ascending order of size. No guarantees are, therefore, given that these data structures can be directly mapped.

occam PROC heading	C function prototype

`PROC C.array (VAL [42]REAL32 a,`
`  VAL []REAL32 b,`
 	
`  [42]REAL32 c,`
`  []REAL32 d,`
 	
`  VAL [768][1024]BYTE e,`
`  VAL [][]BYTE f,`
 	
`  [768][1024]BYTE g,`
`  [][]BYTE h)`
	`void array (const float a[42],`
	`  const float b[],`
	`  int b_size,`
	`  float c[42],`
	`  float d[],`
	`  int d_size,`
	`  const char e[768][1024],`
	`  const char f[][],`
	`  int f_size_0, int f_size_1,`
	`  char g[768][1024],`
	`  char h[][],`
	`  int h_size_0, int h_size_1);`

Figure 5: Equivalent occam and C prototypes (array types)

```
void __array (int w[]) {
  array
   (REAL32 (w[0]),
    REAL32 (w[1]),  VAL_INT (w[2]),
    REAL32 (w[3]),
    REAL32 (w[4]),  VAL_INT (w[5]),
    BYTE   (w[6]),
    BYTE   (w[7]),  VAL_INT (w[8]),  VAL_INT (w[9]),
    BYTE   (w[10]),
    BYTE   (w[11]), VAL_INT (w[12]), VAL_INT (w[13]));
}
```

Figure 6: Interface function (array types)

On the other hand, if we only need occam to declare and hold data structures (including C *union*s) whose values are initialised and used by C functions, then `INT` arrays (of the correct size) may represent the C structures. [Note theat occam `INT` arrays will be aligned on word (i.e. 32-bit) boundaries in memory. If it is necessary that the structures should be double-word (i.e. 64-bit) aligned, `INT64` arrays should be used.]

Similarly, occam `INT`s may be used to hold C pointers-to-pointers or pointers-to-functions; but they only have meaning in the C world.

Finally, occam `CHAN`s may be turned into `INT` values by the `RETYPES` mechanism and, hence, passed to C – but this is getting a little exotic!