"Hello World" in Micro-C

This lab walk-through tells you how to run a "Hello World" program using a Netronome NIC and command line tools.

There are 96 MEs on the NFP-4xxx chip and 120 MEs on the NFP-6xxx chip, collected into a number of "islands". Each ME has its own instruction store and local memory. Each island also contains additional shared memory and special-purpose execution units that operate on data in that shared memory. Islands are not all identical. Some islands contain mostly MEs; some contain a few MEs plus other functionality such as PCIe interfaces or bulk-crypto; some islands contain only shared memory and special-purpose execution units.

It is traditional to start with a program that prints "Hello World", or on embedded systems to flash an LED, or something similar. But we are writing code for a "Micro-Engine" processor (ME) which has no direct access to I/O facilities, so the traditional "Hello World" program for the ME merely reads integers from one initialized array in memory and writes them in reverse to another array. (This example was introduced in IXP2400/2800 Programming by Erik J. Johnson & Aaron R. Kunze, Intel Press 2003.) Of course, to be convinced that the program has actually done something, you need to be able to inspect the memory before and after, to see what has happened. So we will also learn how to do that.

Steps

1. Check out the repository containing these labs and the source code.
2. Open a terminal window, and enter the directory of the repository. At this point it may be worth browsing the structure of the repository - or that can be saved for later.
3. Enter the 'apps' directory.

```
> cd apps
```

4. Create a new ME application by copying the 'lab_template' directory.

```
> cp -r lab_template lab_hello_world
```

5. Enter the new directory, so the new application can be created.

```
> cd lab_hello_world
```

6. Enter the following code into a new '.c' file, 'hello_world.c'. Use the editor of your choice.

```
#include <nfp.h>
__declspec(ctm) int old[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
__declspec(ctm) int new[sizeof(old)/sizeof(int)];
int
main(void)
{
    if (__ctx() == 0) {
        int i, size;
        size = sizeof(old)/sizeof(int);
        for (i = 0; i < size; i++) {
           new[i] = old[size - i - 1];
        }
    }
    return 0;
}
```

This is a simple program that copies data from one array, `old` in to a new array `new`, using the inner for loop. This loop is conditional on `__ctx()` being zero. The `__ctx()` function is defined in `nfp.h`, and it returns the thread number (or context number) of the executing thread on the microengine. Hence the copying of the array is only performed by the thread 0 for the microengine.

When the execution completes, for each thread, the function `main` returns; using the standard run-time library for micro-C this causes the execution of that thread to stop, while other threads may continue.

It is worth noting at this point the `__declspec(ctm)` attribute provided for the arrays `old` and `new`. This is a micro-C attribute that informs the toolchain as to which

memory in the NFP should be used to hold the associated structure - in this case the arrays. This program places the arrays in the CTM that is closest to the ME running the code.

7. Now add the new source file to the Makefile for compilation. This involves editing the Makefile and adding a few lines. Find the section of the Makefile with the comment 'Application definition starts here'. Add in the line shown below to make the new code be compiled as a program.

```
#
# Application definition starts here
#
$(eval $(call micro_c.compile_with_rtl,hello_world_obj,hello_world.c))
```

Note that spaces are critical in Makefiles; do not put extra spaces anywhere.

8. Now assign the new code to a set of MEs. This involves adding another line in the Makefile, after the first additional line. The code is to be assigned to two MEs: i32.me0 and i32.me1

```
#
# Application definition starts here
#
$(eval $(call micro_c.compile_with_rtl,hello_world_obj,hello_world.c))
$(eval $(call fw.add_obj,hello_world,hello_world_obj,i32.me0 i32.me1))
```

Note that spaces are critical in Makefiles; do not put extra spaces anywhere.

9. Complete the Makefile with an invocation to make the firmware. This is the third line added to the Makefile, and it causes the firmware to be built using the description provided in previous lines. Particularly, the new line makes the firmware be built with run-time symbols.

```
#
# Application definition starts here
#
$(eval $(call micro_c.compile_with_rtl,hello_world_obj,hello_world.c))
$(eval $(call fw.add_obj,hello_world,hello_world_obj,i32.me0 i32.me1))
$(eval $(call fw.link_with_rtsyms,hello_world))
```

Summary notes:

- The first line (with `micro_c.compile_with_rtl`) specifies a C file to be compiled (`hello_world.c`) with the run-time library, and to create an object `hello_world_obj`. This object is a program that can run on an ME, or more than one ME.
- The second line adds the object to some firmware: it adds `hello_world_obj` to the firmware that is called `hello_world`; it does this such that the object program is installed on two MEs, `i32.me0` and `i32.me1`. These are two MEs in island number 32.
- The third line completes the firmware description, indicating that the firmware `hello_world` should be linked, and run-time symbols should be included in

10. The firmware is now ready to be compiled, using make:

```
> make
```

11. A minor diversion can be taken now: look at the output link map file, which shows the placement of the symbols used in the program.

```
> cat hello_world.map
Memory Map file: ./hello_world.map
Date: Thu Nov 12 17:37:53 2015

nfld version: 5.2.0.0-devel,  NFFW: hello_world.fw

Address       Region     ByteSize        Symbol
=================================================
0x0000000000800000    i24.emem      108             .mip
0x0000000000000000    i32.ctm       704             i32.me0.ctm_40$tls
0x00000000000002c0    i32.ctm       704             i32.me1.ctm_40$tls

ImportVar           Uninitialized Value
=================================================
```

There are three informative lines in the link map file. The first indicates that the `.mip`

is placed in `i24.emem`. The `.mip` is the data structure that contains the run-time symbols, and the toolchain will always place this at the same place.

The second and third lines indicate that `i32.ctm` is usd to contain two 704-byte structures, one for ME 0 and one for ME1, whose run-time symbols are `i32.me0.ctm_40$tls` and `i32.me1.ctm_40$tls`. These are "thread local storage"; with eight threads in each ME, and with the program declaring an array for each thread (effectively), memory has to be allocated for one array per thread (i.e. there are eight copies of the `old` array per ME, and eight copies of the `new` array).

Each thread, then, has 704/8 or 88 bytes of memory associated with it, within its "thread local storage"; the start of this contains the `old` array, and starting 48 bytes further in is the thread's `new` array.

12. It is now time to load the firmware that has been created onto the NFP.

```
> make load_hello_world
```

13. With the firmware loaded, the memory contents of the program can be examined.

```
> nfp-rtsym --len 176 i32.me0.ctm_40\$tls:0
0x0000000000:  0x00000001 0x00000002 0x00000003 0x00000004
0x0000000010:  0x00000005 0x00000006 0x00000007 0x00000008
0x0000000020:  0x00000009 0x0000000a 0x00000000 0x00000000
0x0000000030:  0x00000000 0x00000000 0x00000000 0x00000000
*
0x0000000050:  0x00000000 0x00000000 0x00000001 0x00000002
0x0000000060:  0x00000003 0x00000004 0x00000005 0x00000006
0x0000000070:  0x00000007 0x00000008 0x00000009 0x0000000a
0x0000000080:  0x00000000 0x00000000 0x00000000 0x00000000
*
```

Note the `\` before the `$`, to escape it for the shell.

This shows the first 176 bytes of the contents of the thread local storage for ME 0 in island 32; this will be for all the threads.

Note that the `old` array has the data 1, 2, 3, etc; the `new` array has data of 0.

The lines with `*` indicate that the *previous line contents* are repeated.

14. The code can now run

```
> make fw_start
nfp-nffw start
```

All the microengine threads have now started - and the threads will all have completed execution.

15. Inspect the memory again

```
> nfp-rtsym --len 176 i32.me0.ctm_40\$tls:0
0x0000000000:  0x00000001 0x00000002 0x00000003 0x00000004
0x0000000010:  0x00000005 0x00000006 0x00000007 0x00000008
0x0000000020:  0x00000009 0x0000000a 0x00000000 0x00000000
0x0000000030:  0x0000000a 0x00000009 0x00000008 0x00000007
0x0000000040:  0x00000006 0x00000005 0x00000004 0x00000003
0x0000000050:  0x00000002 0x00000001 0x00000001 0x00000002
0x0000000060:  0x00000003 0x00000004 0x00000005 0x00000006
0x0000000070:  0x00000007 0x00000008 0x00000009 0x0000000a
0x0000000080:  0x00000000 0x00000000 0x00000000 0x00000000
*
```

The data starting at 0x30 is now 10, 9, 8, ... 1 - but only for thread 0. The for loop has been executed, only for thread 0, and the data has been copied in reverse - as expected.

16. IMPORTANT - unload the firmware

```
> make fw_unload
```

17. Optional stopping point

18. Edit the hello_world.c code, and add `scope(island)` to the attributes for the array declarations

```
__declspec(ctm export scope(island)) int old[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 1(
__declspec(ctm export scope(island)) int new[sizeof(old)/sizeof(int)];
```

19. Make the code again and look at the map file.

```
> make
```

20. Look at the output link map file, which shows the placement of the symbols used in the program.

```
> cat hello_world.map
Memory Map file: ./hello_world.map

Date: Thu Nov 12 15:11:27 2015

nfld version: 5.2.0.0,  NFFW: ./hello_world.fw

Address       Region     ByteSize        Symbol
=================================================
0x0000000000800000    i24.emem      108             .mip
0x0000000000000000    i32.ctm       40              i32._old
0x0000000000000030    i32.ctm       40              i32._new

ImportVar           Uninitialized Value
=================================================
```

There are again three lines of information; the first is the same.

The second and third lines are quite different though. There is now a single copy of `i32.old` and a single copy of `i32.new` - before there we copies for 2 microengines and 8 threads per microengine. The invocation of `scope(island)` means that the data is shared by all microengines in the same island - so both microengines and all the threads share the same arrays.

21. Edit the hello_world.c code, and make the scope to be `shared` in the attributes for the array declarations

```
__declspec(ctm shared) int old[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
__declspec(ctm shared) int new[sizeof(old)/sizeof(int)];
```

22. Make the code again and look at the map file.

```
> make
```

23. Look at the output link map file, which shows the placement of the symbols used in the program.

```
> cat hello_world.map
Memory Map file: ./hello_world.map

Date: Thu Nov 12 15:11:27 2015

nfld version: 5.2.0.0,  NFFW: ./hello_world.fw

Address       Region     ByteSize        Symbol
=================================================
0x0000000000800000    i24.emem      108             .mip
0x0000000000000000    i32.ctm       40              i32.me0._old
0x0000000000000060    i32.ctm       40              i32.me0._new
0x0000000000000030    i32.ctm       40              i32.me1._old
0x0000000000000090    i32.ctm       40              i32.me1._new

ImportVar           Uninitialized Value
=================================================
```

There are now copies of `old` and `new` that are separate for the different microengines in the firmware - remember that the firmware has the same code on ME0 and ME1. The scope for the arrays is now shared between the ME threads, but is not shared between microengines.