Introducing the Epiphany x-lib

Mark Honman (mark.honman@gmail.com)
October 2013

The x-lib is a set of eXperimental eXtensions to the Epiphany* SDK. Due to its experimental nature it is likely to change rapidly (that includes the signatures and semantics of API functions), and until a decent library of unit tests has been constructed there will sometimes be regressions.

On the other hand its experimental nature means that backwards-compatibility takes second place to functional improvement, and everyone is encouraged to suggest changes. It is envisaged that stable and popular features would be available to become part of an official API.

*Epiphany is a trademark of Adapteva Inc.

Why Synchronous message passing?

While the Epiphany hardware provides an easy way to transfer information between cores, and the SDK provides useful wrappers around this functionality, there is no standard mechanism for local synchronisation.

Synchronous message passing is useful during early stages in the development of parallel applications, when it helps the programmer to reason about where two communicating processes are relative to each other in their individual control flows. In production code it may also be useful where memory is tight and there is little to spare for intermediate buffers – synchronous message passing allows the data to be transferred without buffering because the transfer only occurs at a time when the receiver is ready to accept it, and the sender can proceed to re-use those locations once the transfer has completed.

Synchronisation can be used to ensure that race conditions do not occur, but introduces the possibility of deadlock. In most cases deadlock is the lesser of the two evils because it is a stable state (some would say excessively so) and its origin is thus easier to determine.

Future extensions to the x-lib will permit loosely synchronised message passing, in which the sender and receiver can indicate their willingness to communicate without being forced to rendezvous with the peer process. Thus two or more processes can exchange information with each other without any need to explicitly order the communications in the code.

Why *another* new API?

Apart from the freedom to experiment and seek out techniques that are particularly appropriate to the Epiphany architecture, design tradeoffs in the x-lib API generally favour minimisation of e-core internal memory use. These techniques may become useful in the implementation of message-passing standards such as MPI.

Acknowledgements

- to Flemming and Ben at Sundance for access to their Zedboard with Epiphany-III.
- to the Occam and Transputer architects of long ago, for setting a remarkable standard of conceptual elegance in software and hardware for parallel processing.
- To Yaniv Sapir and the Parallela forum members who have provided many helpful (and patient) suggestions along the way.

x-lib concepts

Application

An application is a collection of cooperating tasks running on the host and a workgroup within the attached Epiphany processor. The maximum number of tasks is defined at the time the application is created, with no more than one Epiphany task per workgroup core.

The tasks and the connections between them are defined by an application coordinator program. For some common communication topologies – e.g. a SPMD mesh-connected configuration mapped to a rectangular workgroup – there is an application-level utility routine that can be used by the coordinator to create the necessary tasks and the connections between them.

Utility routines provide a way for the coordinator process to provide information on the structure and state of the application. A “monitor” function periodically displays some or all of this status until all tasks have completed. The overall application state is derived from the individual task statuses.

Once all launched tasks have terminated, the application is considered to have terminated but its resources remain assigned until they are explicitly released. This allows the coordinator to read back data from core memory and assists in post-mortem debugging of a failed application.

Notes:

- It is not necessary to assign a task to each workgroup core: cores with no assigned task are not started.

Limitations as at 21/10/2013:

- Host based tasks are not yet implemented.
- Once the first task has been launched, it is no longer possible to add connections.

Task

Tasks are the basic unit of computation, corresponding to Epiphany cores and host processes. Unlike operating system level processes they are considered to be part of the application and each one must be prepared before the application can be launched.

All tasks have access to task-local memory, private areas within the DRAM that is shared between host and Epiphany, and globally shared memory that is managed by the master process.

Tasks receive Unix-like argument lists from the master process – the first (zeroth) argument being the name of the file from which the task's code was loaded. In order to achieve this the main() program is part of the x-lib library code, and the main program of the task is a function `task_main (int argc, char * argv[]);`

A small task-oriented API enables the task to obtain information on its place in the application, access endpoints for the inter-task connections defined by the controller, and report status back to the controller.

While task support code is efficient and has a very small memory footprint, the name “task” has been deliberately chosen to emphasis their persistence and the fact that a task has complete control over the Epiphany core on which it has been launched.

Limitations as at 21/10/2013

- Host based tasks are not yet implemented
- Task argument passing is not yet implemented
- Management of the globally shared memory area is not implemented (waiting on host tasks and host-side messaging endpoints)

Connection and Endpoint

In this version of x-lib, communication can only occur where a connection between the tasks has been defined by the application controller. The connection is identified to each task by integer-valued key the bottom 16 bits of which can be used for application-defined keys.

The connection can be identified to the connected tasks by different keys at each end, as long as no two connections involving a task are given the same key for that task's endpoint. This enables tasks to be coded in generic terms, for example in a mesh-connected topology the `X_TO_LEFT` key of each task identifies the same connection as the `X_FROM_RIGHT` key of its left-hand neighbour.

Connections are uni-directional, one end is an input and the other an output. See notes for the reasoning behind this choice.

Since almost all communication in the Epiphany architecture occurs in the form of memory-to-memory transfers, it is necessary for the connection to have a small amount of control data local to the cores at each end (this enables a core to busy-wait on local rather than off-core memory).

These local data structures are called Endpoints, and are all that a task needs to communicate with its peer once the peers have exchanged endpoint addresses. This exchange is performed as part of the x-lib task initialisation that occurs before control is transferred to the user's `task_main`.

Limitations as at 21/10/2013:

- While it is possible to define a connection from a task back to itself, attempts to synchronise using that connection will block indefinitely.

Why have different keys for sender and receiver?

The keys are a substitute for compile-time channel names. They provide a way for the communication paths to be given names that make sense in the context of the sending or receiving process, without any need for the names to be globally unique.

Why explicit and asymmetrical communication paths?

If the full set of possible communication peers is known at task startup, it is possible to allocate exactly the right amount of e-core memory for the endpoint data structures.

The original reason for the asymmetry was to support an synchronisation model that used an absolute minimum of off-core writes and which was necessarily asymmetric. The synchronisation method currently in use is symmetric, so if there is a strong case for bi-directional communication paths it would be possible to change over to that approach.

It is likely that slave-mode DMA could be used to implement a N:1 message passing system, which would support a paradigm such as MPI that does not depend on pre-configuration of communication paths.

Notes on Communication Costs

In the x-lib release of 21 October 2013

- The `x_sync` intertask synchronisation call takes 29 cycles to synchronise with the task running on an adjacent core of an Epiphany-III; at a clock speed of 600MHz this equates to 49ns.
- There is a very small possibility of a race condition in `x_sync` if the core is interrupted between two specific instructions and its peer attempts synchronisation before control is transferred back to the point where it was interrupted. This can be made bulletproof by disabling and then re-enabling interrupts at the right points in the code.
- The `x_sync_send` and `x_sync_receive` routines have a much higher overhead which can probably be reduced by careful examination of the code which performs the memory-to-memory copy. The total overhead on the Epiphany-III is about 100 cycles (170ns at 600MHz) which is about equally divided between synchronisation and transfer speed optimisation.

- Both coded and DMA transfer speeds are affected by data aligment: doubleword-aligned transfers that are a whole number of doublewords in size move 8 times the data per cycle as can be transferred when source or destination is byte-aligned. More information on this can be found in the source files `x_sync.c` and `e_messaging_test.c`.

The above transfer times are obviously best-case: communication between adjacent tasks that are already running in lockstep.

The essentials of an x-lib application

```
#include <x_application.h>
#include <x_application_display.h>

int main(int argc, char *argv[])
{
    int workgroup_rows    = 0,
        workgroup_columns = 0,
        host_task_slots   = 1;

    x_initialize_application (argv[0],
                              &workgroup_rows, &workgroup_columns,
                              &host_task_slots);
    x_prepare_mesh_application ("x_hello.srec", X_WRAPAROUND_MESH);
    x_launch_application (argc-1, argv+1);
    x_monitor_application (X_STATUS_DISPLAY | X_LIVE_DISPLAY, 1);
    return (x_finalize_application ());
}
```

Understanding the status display

```
Running  WG 4x4  Virgin:1  Init:0  Act:14  Success:2  Fail:0
Core/PID  Co Ro ST   HB CN Cmd                 Status
      0    0  0  3    5  8 e_messaging_test.s Hello from core 0x808 (0,0) taskpos 0
0 in 4 4
      0    1  0  3    5  8 e_messaging_test.s Hello from core 0x809 (1,0) taskpos 1
0 in 4 4
      0    2  0  3    5  8 e_messaging_test.s Hello from core 0x80a (2,0) taskpos 2
0 in 4 4
      0    3  0  3    6  8 e_messaging_test.s Hello from core 0x80b (3,0) taskpos 3
0 in 4 4
      0    0  1  0    6  8 e_messaging_test.s Second receive result 512
      0    1  1  0    6  8 e_messaging_test.s Second send result 512
      0    2  1  3    2  8 e_messaging_test.s Hello from core 0x84a (2,1) taskpos 2
1 in 4 4
      0    3  1  3    2  8 e_messaging_test.s Hello from core 0x84b (3,1) taskpos 3
1 in 4 4
```

Writing Tasks with x-lib

```
#include <string.h>
#include <x_task.h>
#include <x_sleep.h>
#include <x_endpoint.h>
#include <x_application.h>
#include <x_sync.h>

int task_main(int argc, const char *argv[])
{
    int                 wg_rows, wg_cols, my_row, my_col;
    x_endpoint_handle_t ep;
    char                message[80];
    int                 received;

    x_get_task_environment (&wg_rows, &wg_cols, &my_row, &my_col);
    x_set_task_status ("Howzit from task %d at (%d, %d) in workgroup of %d by %d”,
                       x_get_task_id(), my_col, my_row, wg_cols, wg_rows);
    x_sleep(10);
    if (my_col == 0) {
          ep = x_get_endpoint (X_TO_RIGHT);
          snprintf (message, sizeof(message), "Hello from (%d,%d)", my_col, my_row);
          x_set_task_status ("Sending...");
          x_sync_send(ep, message, strlen(message)+1);
        }
        else if (my_col == 1) {
          ep = x_get_endpoint (X_FROM_LEFT);
          x_sleep(2);
          received = x_sync_receive(ep, message, sizeof(message));
          x_set_task_status ("Received [%s] from left", message);
          x_sleep(3);
        }
        x_set_task_status ("Goodnight and goodbye from (%d, %d)", my_col, my_row);
        return X_SUCCESSFUL_TASK;
    }
```

Utility routines

x_sleep.h

Configuring x-lib tunables

Diagnostic tools

Overview of the internals of x-lib

The Application global data structure

NOT YET DOCUMENTED

Task private data structures

NOT YET DOCUMENTED