The Epiphany x-lib

Version 0.2

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

Table of Contents

Introduction

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 *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.

New in version 0.2

- Diagnostic routines for dumping the contents of x-lib data structures and Epiphany registers and address space in human-readable form.
- Functions to support the Epiphany system-call interface: testing for traps, extraction of system call parameters, handling of the system calls, and return of the results to the Epiphany program. These functions can be used with any Epiphany application, and are not dependent on the x-lib application, task or communication libraries.
- Example programs

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 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 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 102 cycles (170ns at 600MHz) of which 67 are consumed by synchronisation. The remainder of the overhead is in transfer speed optimisation code that tests the data alignment and size to determine the most appropriate transfer method.
- 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`.
- Comparisons of C-coded and DMA transfer rates can therefore be conveniently expressed in millions of units transferred per second, where the transfer unit is a byte, word or doubleword as determined by the data alignment and size.
- On the 600MHz Epiphany-III, preparation of the C-coded transfer takes about 57ns (34 clock cycles), while the overhead associated with `e_dma_copy` is approximately 280 cycles, i.e. 470ns.
- Once preparation has been done, the C-coded transfer moves 130 to 140 million data items per second, while e_dma_copy moves 200 million items per second.
- Taking these overheads into account, once more than 150 items are to be transferred `e_dma_copy` is the better transfer method.

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

This example is based on the `test_controller.c` file, with the sanity-checking tests stripped out to keep it as simple as possible. This little program is a generic test driver that loads an x-lib task onto every Epiphany core, configures wraparound connections between the tasks (in other words every core is provided with sending and receiving connections to its four neighbours), then launches them and monitors progress of the application until its completion. The name of the Epiphany executable is received as first argument, and the remaining arguments are passed to the newly created tasks.

The program must run with supervisor rights in order to access the Epiphany hardware – for example using a wrapper script. The `test.sh` script can be used for this purpose: after setting up the environment its arguments are simply passed to the test_controller program (if no Epiphany executable is specified, it defaults to running the `x_hello` program described in the next section).

```
#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   = 0;

    x_initialize_application (NULL, &workgroup_rows, &workgroup_columns,
                                    &host_task_slots);
    x_prepare_mesh_application (argv[1], X_WRAPAROUND_MESH);
    x_launch_application (argc-2, argv+2);
    x_monitor_application (X_STATUS_DISPLAY | X_LIVE_DISPLAY, 1);
    return (x_finalize_application ());
}
```

Understanding the status display

First, an example of the status display when the `x_hello` core program described in the next section is run on a 16-core Epiphany.

```
Running  WG 4x4  Virgin:0  Init:0  Act:6  Success:10  Fail:0
Core/PID  Co Ro ST   HB CN Cmd                Status
      0    0  0  3    6  8 x_hello.srec Sending...
      0    1  0  3    6  8 x_hello.srec Receiving...
      0    2  0  0    6  8 x_hello.srec Goodnight and goodbye from (2, 0)
      0    3  0  0    7  8 x_hello.srec Goodnight and goodbye from (3, 0)
      0    0  1  0    4  8 x_hello.srec Goodnight and goodbye from (0, 1)
      0    1  1  3    4  8 x_hello.srec Received [Hello from (0,1)] from left
      0    2  1  0    3  8 x_hello.srec Goodnight and goodbye from (2, 1)
      0    3  1  0    3  8 x_hello.srec Goodnight and goodbye from (3, 1)
      0    0  2  0    4  8 x_hello.srec Goodnight and goodbye from (0, 2)
      0    1  2  3    4  8 x_hello.srec Received [Hello from (0,2)] from left
      0    2  2  0    3  8 x_hello.srec Goodnight and goodbye from (2, 2)
      0    3  2  0    3  8 x_hello.srec Goodnight and goodbye from (3, 2)
      0    0  3  3    2  8 x_hello.srec Howzit from task 12 at (0, 3) in workgroup
of 4 by 4
      0    1  3  3    3  8 x_hello.srec Receiving...
      0    2  3  0    3  8 x_hello.srec Goodnight and goodbye from (2, 3)
      0    3  3  0    3  8 x_hello.srec Goodnight and goodbye from (3, 3)
```

The first line of the display contains the overall application status, the workgroup size, and the number of tasks in various states. Control data structures exist for **Virgin** tasks but their execution has not yet started, while the **Init** state indicates tasks where execution has begun but control has not yet been transferred to user code.

This is followed by a table of per-task details:

- Core ID (in the case of Epiphany tasks) or Unix PID (for host tasks). Not implemented yet!
- Coordinates – columns and row - within the workgroup

- Task state, one of the following:
  - -1 Failed
  - 0 Successfully completed
  - 1 Virgin
  - 2 Initializing
  - 3 Running
- Heartbeat, a simple counter that can be used to help spot tasks that have stalled.
- CN is the number of connections
- Cmd is the name of the file being executed.
- The Status field contains the last status message set by the task.

Notes:

- It is envisaged that the list of task states will be expanded to include Sending, Receiving, and I/O wait.
- In the example listing, the tasks in row 0 of the workgroup have higher Heartbeat values because they were started first and needed to wait longest for the exchange of endpoint details to complete.
- However these tasks are also slower at updating their statuses in external memory, so it may be that this behaviour is systematic.
- Setting task status is quite costly, not only because it is written to external RAM, but also because parts of the string and stdio libraries are needed by the core program.

Example programs

The following examples accompany the x-lib software in the `/src` directory:

- `messaging_test.c` (host) and `e_messaging_test.c` (Epiphany) – use `run.sh` to run this example.
- `test_controller.c` and its wrapper script `test.sh` are intended for use with with a variety of Epiphany-side example programs that do not need host code.
  - `x_hello.c` – hello world, x-lib style
  - `x_naive_matmul.c` – test code for comparison of naïve non-parallel matrix multiplication code against host processors. Will probably grow with time to demonstrate different optimisation techniques.
  - `x_syscall_demo.c` – workout for testing system-call handlers
  - `simon.c` – Simple IO monitor, useful in its own way but also serves as an example of how to use the system call and trap handling functions.

Writing Tasks with x-lib

This tutorial is based on the source code to the x_hello example (`x_hello.c`). This is an SPMD core program in which the tasks use `x_set_task_status` to communicate with the outside world. The left-most tasks in the mesh send a “hello” message to the middle tasks which then set it as their status. The timing of the sends and receives is offset to demonstrates the synchronous nature of the transfer (offsets are +row*2 seconds for sends, -row*2 for receive operations). The tasks in the remainder of the workgroup simply set a "hello" status, wait 10 seconds, and then set a "goodbye" status.

Although `x_sleep` has been used to slow the demonstration down, a similar `x_usleep` function is available for sub-second pauses.

Note that the example assumes that the connections from left-column tasks to second-column tasks have been created with `X_TO_RIGHT` and `X_FROM_LEFT` keys for their endpoints as is done by the host-side application setup function `x_prepare_mesh_application`.

The example should work for any workgroup size from 1 row x 2 columns upwards.

```
#include <stdio.h>
#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) {
      x_sleep (my_row*2);
      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_set_task_status ("Receiving...");
      x_sleep((wg_rows - (my_row+1))*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;
}
```

x-lib Support Programs

Simon – the Simple I/O Monitor

A basic subset of Unix system calls are supported in the C library distributed with the Epiphany SDK. These are implemented via TRAP instructions, and the present implementation is partially described in the Epiphany Architecture Reference description of TRAP (there are some errors and omissions which are described in the trap and system call handling library documentation).

The standard Epiphany SDK supports these calls with the assistance of gdb. An instance of the debugger connects to the e-server which monitores the cores for halt states, and passes the parameters of the system call traps through to the connected instance of gdb.

The simple I/O monitor `simon` is a first step towards providing trap-handling functionality outside of the debugging environment. It is intended to provide I/O performance close to that of a host program, and is designed to run alongside the user program.

It can be run as a process dedicated to monitoring a specific core – if a row and column number are specified on the command line – or in the absence of command-line parameters it polls all cores of the Epiphany in a round-robin fashion.

Usage:

```
simon [<row> <col>]
```

If a core is not specified, every core in the attached Epiphany device is polled for trap statuses.

Limitations as at 12/11/2013:

- The polling interval of 10 microseconds results in continuous background traffic on the e-mesk and especially the external e-links.
- Direct accesses are made to Epiphany memory and registers as well as external RAM, and thus the program must run as root.
- Polling must be stopped before the Epiphany device is reset.
  When the Epiphany device is reset (e.g. by a call to `e_reset_system()`) it is inaccessible for a significant fraction of a second – `e_reset_system()` sleeps for 0.2s after issuing a reset. If the I/O monitor attempts to access the Epiphany during that time, the host CPU will lock up.

The IO monitor performs all I/O synchronously, thus if one instance of the program is serving the whole Epiphany device, cores may block waiting for the I/O monitor to interact with a slow device on behalf of another core.

x-lib API Reference

Configuring x-lib tunables

X-lib tunable values are consolidated into `x_lib_configuration.h`. It is likely that some of this information can be extracted from eSDK data structures, configuration files, and LDF files – this set of tunables will change as opportunities arise to obtain the information from eSDK sources.

```
#define X_EPIPHANY_FREQUENCY (600)
```

Core clock speed of the Epiphany system, needed for timing functions.

```
#define X_ESTIMATED_CORES (64)
#define X_APPLICATION_WORKING_MEMORY_SIZE (X_ESTIMATED_CORES*1024)
```

These definitions affect the amount of space allocated in external RAM for x-lib internal data structures. Application-specific data structures such as task and connection lists are allocated from the working memory, and thus `X_ESTIMATED_CORES` is not a hard limit.

```
#define X_HOST_PROCESS_SHARED_DRAM_BASE (0x00000000)
#define X_EPIPHANY_SHARED_DRAM_BASE     (0x8e000000)
```

These definitions are platform-specific and must match the HDF file (they are an example of information that can probably be obtained from the eSDK). They represent the start address of the external shared DRAM in e_alloc() terms and Epiphany address space respectively.

```
#define X_LIB_SECTION_OFFSET (0x00800000)
```

x-lib applications must be built using a special LDF file that introduces an xlib_shared_dram section for xlib data. The `X_LIB_SECTION_OFFSET` must match the offset of this section from the start of the first shared DRAM segment.

```
#define X_MESSAGING_MIN_DMA_ITEMS (152)
```

DMA transfers have a significant setup overhead (see ***Notes on Communication Costs*** above) and in the case of blocking inter-core transfers, load+store loops are more efficient for small amounts of data. The `X_MESSAGING_MIN_DMA_ITEMS` value indicates the transfer size at which DMA becomes more efficient. Since alignment rules force both types of transfer to use the same kind of unit (byte, halfword, word, or double-word), the changeover point depends on the number of units transferred rather than the total number of bytes being transferred.

Unix Compatibility Functions

x_sleep () - Sleep for the specified number of seconds

Synopsis

```
#include "x_sleep.h"
```

```
unsigned int x_sleep (unsigned int seconds);
```

Description

Pauses program (thread) execution for the specified number of seconds. In host programs the function will return early if a signal handler is called.

Available to both host and Epiphany programs.

Return value

The number of seconds remaining from the original request if interrupted by a signal (host only), otherwise 0.

Bugs

This function uses `x_usleep()` which in its Epiphany implementation uses one of the on-core timers. It therefore suffers from the same limitations as `x_usleep()` - see the description of that function for details.

x_usleep () - Sleep for the specified number of microseconds

Synopsis

```
#include "x_sleep.h"
```

```
int x_usleep (unsigned int usec);
```

Description

Pauses program execution (thread execution in host programs) for the specified number of microseconds, maximum 1000000.

Available to both host and Epiphany programs.

Return value

0 if successful, -1 on error.

Sets errno to `EINVAL` if usec is greater than 1000000; errno may be set to `EINTR` if the function is interrupted by a signal (host only).

Bugs

In Epiphany programs, `x_usleep()` is implemented using `e_wait()` (an undocumented SDK function) called with core timer 0 and will interfere with other functions that use that timer. Interrupts do not affect the function, and thus `EINTR` is never set, but if the interrupt routine modifies this timer the behaviour of `x_usleep()` is undefined.

In host programs, `x_usleep()` simply calls the POSIX `usleep()` function.

Data dump functions

Epiphany Programming Notes

ESDK Limitations as of 5.13.09

- No relocation is performed when programs are loaded by `e_load()` and `e_load_group()`. Code is loaded according to the section layout in the selected LDF file, which means that when using the default LDF every core has an external code section starting at the beginning of the external RAM. This has two important conseqences:
  - Two different core programs can only be used if their external-RAM code sections have an identical layout – if not, the first program's calls to its external library routines will reference entry points that have been overwritten by the code of the most recently loaded program.
  - All library functions loaded into external RAM must be thread-safe because two or more cores may be calling that function at the same time.
- While some examples uses the `shared_dram` section as a convenient place to locate globally shared data structures, if the program uses a heap that has been placed in external RAM, the heap of Epiphany core (0,0) will be using the same area. Since the eSDK functions do “fix up” the heap locations to give each core an individual external heap, it should be possible to work around this limitation using a custom LDF. But in that case it is best to create a section dedicated to that global area.

General Caveats

- Static variables – both local and global – in C programs are pre-initialised (defaulting to zero) and form part of the executable file produced by the linker. Thus loading an Epiphany program containing a structure that has been placed in a shared data area will initialise that area. It is generally advisable (especially if the host is responsible for populating the area with data) to access that area through a pointer, either pre-initialised or passed from the host.
- Reading e-core general purpose registers from the host while the core is running causes the host to crash, as least in the case of the ZedBoard. Reading the other registers (config, status, PC, etc. does not seem to cause problems).

Overview of the internals of x-lib

The Application global data structure

NOT YET DOCUMENTED

Task private data structures

NOT YET DOCUMENTED