Using the Epiphany SDK for Host Side Application

Rev 4.13.03.30

Using the Epiphany SDK for Host Side Application

What's New in ESDK 4.13.03.30

This Application Note describes the software interfaces for programming an Epiphany accelerated computer, emphasizing the major changes of the new eSDK e-HAL Host driver from the previous release.

With the 4.13.03.30 release of the eSDK we completed the transition from the EMEK legacy to the new, univrsal and portable Epiphany SDK host driver. While the old driver was called e-host, the new driver's name is e-hal. The actual driver file can be found at:

```
${EPIPHANY_HOME}/tools/host/lib/libe-hal.so
```

The header file, required for developing a C or C++ host program are located at:

```
${EPIPHANY_HOME}/tools/host/include/e-hal.h
```

The previous API was described in the "Programming the Zynq Based System (v2.1)" document, where the functions for initializing the device and communicating with cores were described. The basic unit was a single eCore, referred to by a core-number ID.

The new API takes "eCore workgroups" as a main theme. A workgroup is a rectangular mesh of eCores, as small as a single core or as large as the whole chip(s), usually (but not necessarily) allocated for concurrently performing a computational task.

Data Types

The major data types that are used system-wide are:

`e_platform_t` - defining an Epiphany platform object
`e_epiphany_t` - defining an eCore workgroup object
`e_mem_t` - defining an external memory object

Initializing the Epiphany System

The host appication should initialize the e-hal using the e_init() call. This function accepts a HDF (Hardware Description File) file name, that describes the physical attributes of the Epiphany system, mainly the various Epiphany chips and external memory segments. The default HDF's are located at:

```
${EPIPHANY_HOME}/bsps/current/xyz.hdf
```


When a e-hal is initialized, the physical available eCore mesh is calculated as the minimal bounding box around all installed chips. The eCore with the lowest CoreID now becomes the platform's origin. The subsequent workgroups are allocated relative to this origin.

The system parameters can be retrieved using the e_get_platform_info() API (please refer to the SDK Reference manual for additional limitations on using this API).

When the application ends, use e_finialize() to release any allocated resources.

Function prototypes:

```
int e_init(char *hdf);
int e_get_platform_info(e_platform_t *platform);
int e_finalize();
```

Allocation of Workgroups and External Memory

A workgroup is allocated using the e_open() API. It requires a reference to a workgroup object, its origin coordinates, relative to the platform origin, and the required size. Note that no check is made for preventing re-allocation of cores that are already allocated for a previous workgroup!

When not required anymore, a workgroup can be released using the e_close() API.

Similarly, an external memory block is allocated using the e_alloc() API. It requires a reference to an external-memory object, its starting address, given as an offset from the Epiphany external memory segment, and its size. Note that no check is made for preventing re-allocation of memory blocks that are already allocated with previous calls to e_alloc()!

When not required anymore, a memory block can be released with the e_free() API.

Function prototypes:

```
int e_open(e_epiphany_t *dev, unsigned row, unsigned col,
    unsigned rows, unsigned cols);
int e_close(e_epiphany_t *dev);
int e_alloc(e_mem_t *mbuf, off_t base, size_t size);
int e_free(e_mem_t *mbuf);
```

Accessing Workgroups and External Memory

The universal API's for data transfer to and from a workgroup or an external memory block are e_read() and e_write(). Although similar interface, there is a slight variation in usage for workgroups and external memory blocks.

For accessing an eCore, the functions receive a reference to a workgroup object, the core's coordinates relative to the workgroup's origin, the address to read from or write to, as an offset in an eCore's local memory space (1MB), a reference to a data buffer and the size of transfer. In order to access an eCore register, predefined register symbols are provided. Please refer to the SDK Reference manual for more details.

In order to access an external memory block, the functions receive a reference to an external-memory object, the address to read from or write to, as an offset from the block's origin and the size of transfer. The coordinate parameters, although required for the function calls, are ignored and should be called with 0.

Function prototypes:

```
ssize_t e_read(void *dev, unsigned row, unsigned col,
    off_t from_addr, void *buf, size_t size);
ssize_t e_write(void *dev, unsigned row, unsigned col,
    off_t to_addr, const void *buf, size_t size);
```

System and Core Control

A few API's are provided to control system and eCore operations.

The e_reset_system() API resets the whole Epiphany system, including the Epiphany chips and the associated FPGA logic. The e_reset_core() API performs a soft reset on a single eCore. It accepts a reference to a workgroup and the coordinates of the eCore to reset, relative to the group's origin.

The e_start() API is used to launch a loaded program on an eCore. The e_signal() API generates a Software Interrupt on an eCore. Both accept a reference to a workgroup and the core's coordinates.

Function prototypes:

```
int e_reset_system();
int e_reset_core(e_epiphany_t *dev, unsigned row, unsigned col);
int e_start(e_epiphany_t *dev, unsigned row, unsigned col);
int e_signal(e_epiphany_t *dev, unsigned row, unsigned col);
```

Program Load and Start

The e-loader is now an integral part of the e-hal driver. This means that no reference to libe-loader.so library is required at compile time. The e-loader processes an Epiphany program image given as an SREC format file. Contrary to the previous driver, the target core for loading is not read from the SREC file itself. Thus, the same image can be loaded on multiple cores w/o re-processing and concatenation of files. Optionally, the loaded programs can be launched immediately after the group load is finished.

Two functions are provided for loading a program on an eCore or on a workgroup. The e_load() API loads an executable image on a single eCore. It receives the image filename, a reference to a workgroup, the coordinates of the core to be loaded, and an indication for launching the loaded program. The e_load_group() API receives the group's coordinate, relative to the platform, and its size.

Function prototypes:

```
int e_load(char *executable, e_epiphany_t *dev, unsigned row,
    unsigned col, e_bool_t start);
int e_load_group(char *executable, e_epiphany_t *dev, unsigned row,
    unsigned col, unsigned rows, unsigned cols, e_bool_t start);
```

Platform Usage Information and Examples

For driver usage example, please refer to the matmul-16 or matmul-64 example applications that are provided with the eSDK release.

Please refer to the "**External (shared) memory architecture**" section in the eHAL chapter of the SDK Reference manual for details and limitations on using the external memory segment from the Host and from the Epiphany.

Future development and improvements

In future releases, we plan on revising the e-loader such that it will process Epiphany ELF binary images, as generated by the compiler, rather than the pre-processed SREC files.

Additionally, a unification of the e_epiphany_t and the e_mem_t object types is planned. However, current programs using these objects will probably not need modifications, as the change will be backward compatible.

