# Programming the Zynq Based System (v2.1)

**This Application Note describes the software interfaces for programming the Zynq based Epiphany computers. This is a draft of the final API, and should be considered as such. No guarantee is provided for inclusion or exclusion of any part of this document in the final API versions.**

## The `e-host` and `e-loader` Libraries

### The `e-host` library API

The `e-host` API encapsulates the low-level Epiphany functionality. This can be seen as the Epiphany device driver. It provides the interface for initializing a device context, reading from and writing to the device and attached eDRAM, as well as resetting and initiating the loaded program.

```
#include <e-host.h>

typedef struct {} Epiphany_t;
typedef struct {} DRAM_t;
```

The e_open() and e_close() create and destroys a device context containing definitions for the chip SRAM memory banks and system registers:

```
int e_open(Epiphany_t *dev);
int e_close(Epiphany_t *dev);
```

The e_read() and e_write() are the standard interface to the chip SRAM memories. The functions get an internal address and the size in bytes of memory to read to or write from a buffer:

```
ssize_t e_read(Epiphany_t *dev, int corenum, const off_t from_addr,
             void *buf, size_t count);
ssize_t e_write(Epiphany_t *dev, int corenum, off_t to_addr,
             const void *buf, size_t count);
```

In addition to these following access functions are provided:

```
ssize_t e_read_buf(Epiphany_t *dev, int corenum,
             const off_t from_addr, void *buf, size_t count);
ssize_t e_write_buf(Epiphany_t *dev, int corenum, off_t to_addr,
             const void *buf, size_t count);

int     e_read_word(Epiphany_t *dev, int corenum,
             const off_t from_addr);
```

```
ssize_t e_write_word(Epiphany_t *dev, int corenum, off_t to_addr,
             int data);

int     e_read_reg(Epiphany_t *dev, int corenum,
             const off_t from_addr);
ssize_t e_write_reg(Epiphany_t *dev, int corenum, off_t to_addr,
             int data);
```

Where:

**e_*_buf()** - read/write a buffer.
**e_*_word()** - read/write a single word.
**e_*_reg()** - read/write a from/to an eCore register (address given as a 16-bit value).

**corenum** - the number of the core to access: 0-15 in the "raster scan" order.
**from_addr** - the Epiphany internal core address to read from.
**to_addr** - the Epiphany internal core address to write to.
**buf** - host memory space data buffer.
**count** - length of data buffer in bytes.
**data** - single word data to write.

A set of functions is provided for accessing the eDRAM. The e_alloc() and e_free() are used to allocate and free a buffer anywhere on the eDRAM space. This buffer can later be accessed with an offset relative to its start address:

```
int e_alloc(DRAM_t *dram, off_t mbase, size_t msize);
int e_free(DRAM_t *dram);
```

Where:

**mbase** - offset from the base address of eDRAM (modulo Linux page size (4KB)).
**msize** - size of the allocated eDRAM (in multiples of Linux page size (4KB)).

Note that in the current implementation, The e_mread() and e_mwrite() are the standard interface to the eDRAM memories. The functions get an offset relative to the buffer defined bye_alloc() and the size in bytes of memory to read to or write from a buffer in the host space:

```
ssize_t e_mread(DRAM_t *dram, const off_t from_addr, void *buf,
             size_t count);
ssize_t e_mwrite(DRAM_t *dram, off_t to_addr, const void *buf,
             size_t count);
```

In addition to these following access functions are provided:

```
ssize_t e_mread_buf(DRAM_t *dram, const off_t from_addr, void *buf,
             size_t count);
```

```
ssize_t e_mwrite_buf(DRAM_t *dram, off_t to_addr, const void *buf,
            size_t count);

int     e_mread_word(DRAM_t *dram, const off_t from_addr);
ssize_t e_mwrite_word (DRAM_t *dram, off_t to_addr, int data);
```

Where:

**from_addr** - offset in the allocated buffer to read from.
**to_addr** - offset in the allocated buffer to write to.
**data** - single word data to write.

The eDRAM and Epiphany mapping data is stored in the respective `DRAM_t` and `Epiphany_t` objects. Normally, there should be one `Epiphany_t` object hanging around, but there may be more than one `DRAM_t` objects. This is done with using allocating multiple eDRAM segments with different `mbase` offsets.

The eDRAM base physical address is `0x1e000000` for the ZedBoard system and `0x3e000000` for the ZynqBoard system. There are currently 32MB of dedicated eDRAM available. This can be increased if required. From the Epiphany side, this same memory is aliased at `0x8e000000` (see below).

The following functions are used to control the device. The e_send_reset() function sends a reset signal to the device. A few reset options are defined, supporting different reset patterns:

```
typedef enum {
      E_RESET_CORES,
      E_RESET_CHIP,
      E_RESET_ESYS,
} e_resetid_t;

int e_send_reset(Epiphany_t *pEpiphany, e_resetid_t resetid);
```

Currently, only the E_RESET_CORES method is supported. This reset type initializes each core in the device individually. Some device-wide functions may not be reset by this reset type.

The **`e_send_ILAT()`** function send an ILAT signal to the given core:

```
int e_send_ILAT(Epiphany_t *pEpiphany, unsigned int coreid);
```

Normally, these two functions are invoked during the program load, but are provided here for improved process controllability.

The e_set_host_verbosity() function sets the diagnostics level of the e-host library:

```
void e_set_host_verbosity(int verbose);
```

## The `e-loader` library API

The `e-loader` library provides the functionality for loading programs (in the form of SREC files) on the Epiphany device. Normally, this should be done at the beginning of the host program.

```
#include <e-loader.h>

typedef int bool;
```

The e_load() API loads a program, represented as a SREC file, to the device. Optionally, the device may be reset prior to the load and the ILAT signal may be sent after the load to start the program. For multicore project where the binary is different for each core, the SREC is the concatenation of the respective per-core SREC files. Alternatively, if all cores share the exact same binary image, then a single core image can be broadcasted across all the chip cores:

```
int e_load(char *srecFile, bool reset_target, bool broadcast,
           bool run_target);
```

**`srecFile`** - file name for the Epiphany SREC image.
**`reset_target`** - reset the Epiphany cores.
**`broadcast`** - broadcast the single-core SREC to all cores.
**`run_target`** - start running the programs immediately after loading the chip.

The e_set_loader_verbosity() function sets the diagnostics level of the e-loader library:

```
void e_set_loader_verbosity(int verbose);
```

**`verbose`** - print diagnostics (levels 0-3)

Note that due to the current eDRAM architecture and e-loader implementation, there is no support for loading programs to external memory (eDRAM), so you should an internal.ldf based linker script.

## Library packages

The two libraries are provided as static libraries, as well as shared objects. We recommend dynamic linking of your host program with the e-host and e-loader libraries, thus providing easier support for future update.

Installation instructions and the files will be available in a separate archive.

## Programming The Epiphany for Zynq

When programming the Epiphany for using the eDRAM, one should note a few limitations. In the current system implementation there is a limitation on accessing some eDRAM physical addresses (`0x3e000000` to `0x3fffffff` -or- `0x1e000000` to `0x1fffffff`) from the eCores. In order to bypass this limitation, we implemented a remapping logic to make this range contiguous and available for all eCores on the chip. The remapped address range is `0x8e000000` to `0x8fffffff`.

Transferring data between eCores and between eCore and eDRAM can be done using memcopy or eDMA. Some DMA configuration are not yet completely verified, especially the reading of data from eDRAM to SRAM. As far as we can tell, using the eLib's e-DMA API's should work as expected. However, should one encounter problems, the fallback should be reverting to memcpy() transfers.

In the provided example matmul project, we used eDMA for transferring data between eCores and for writing results from eCore to eDRAM. But, memcopy (the `subcpy()` function) was used for reading the operand matrices data.

On the current systems, the chip origin is at Core ID 0x808 (32,08). However, this information should normally not be required in the host application, as it is encapsulated inside th e-host library.

## Future development and improvement

We plan on modifying the definition and addressing of core groups in a project. Currently, cores arereferenced using their absolute Core ID, or thrugh the `corenum` variable. In the future, a scheme based on global chip origin Core ID and a group origin coordinates, will enable referencing a core with the relative coordinates in a group. This will enable a better scalability and portability of projects across platforms and configurations.

The name of the e-host module will change to something that better describes it as a device HAL or device driver. This should be the only piece of code required to be ported to a new system, where all the toolchain functionality is built on top of it.