C/C++ Coding Guide

G. Giannopoulou, P. Huang, N. Stoimenov, L. Thiele
April 15, 2014

This document describes how to program DOL-Critical applications using C/C++ as programming language.

To be able to achieve a good performance of applications running on the MPPA-256 architecture platform, a certain coding style is required to program these applications. This coding style has been devised for the CERTAINTY project in which application code is expected to be programmed once, but can be used without any modification for different tool chains, e.g. functional simulation, hardware dependent software generation, and final system synthesis. Another key aspect of the CERTAINTY project is the design space exploration (DSE) used for generating a mapping of the application onto the MPPA architecture platform. To enable automatic DSE, an efficient extraction of key parameters of the application is required. Both aspects necessitate consistent code and a well-defined application programming interface.

In the CERTAINTY project, an application is defined as a network of independent processes with the following features:

- Each process is characterized by a criticality level (safety levels DAL-A to DAL-E).
- Each process is associated with a controller process, which is responsible for activating the former according to a *periodic*, *pseudo-periodic*, *asynchronous* or *restartable* pattern, and conveying control information from other processes or from the system scheduler to it.
- A process can be executed in different modes, depending on the availability of input data or the run-time scheduling behavior. The different modes correspond to different, well-defined functionalities that need to be performed in each case.
- Each process is defined according to the superblock model of execution. Namely, it is defined as a sequence of communication and computation phases, where either accesses to a shared memory system or local

computation is performed. Based on static analysis, the characterization of these phases (lower and upper bounds on memory accesses and CPU execution cycles) is included in the application specification. The superblock model enables efficient timing/interference analysis.

- Processes can communicate directly through data channels, implemented as mailboxes or blackboards. Also, they can communicate with controller processes by transmitting event messages through control channels.

The processes define the application's functionality. This document describes how to program these processes and their controllers as well as the communication between processes and between processes and controllers within the DOL-Critical framework. The key properties of the coding style are:

- The internal state of a process is stored in structure `DOLCProcess`. The details about this structure are given in Section 1.
- The behavior of a process is abstracted into three methods, namely `init()`, `fire()`, and `terminate()`. Execution modes and superblock phases are explicitly defined within `fire()` by a switch statement and the `PHASE()` function, respectively. The details about these functions are given in Section 2.
- Communication among processes is achieved by two main functions i.e., the non-blocking `DOLC_read()` and `DOLC_write()`. Communication between a process and a controller process is achieved by the two functions `DOLC_send_event()` and `DOLC_yield()`, respectively. The details about these functions are given in Section 3.

1 Process Structure

The interface to the internal state and the behavioral functions of a process is abstracted in the structure `DOLCProcess`. This structure `DOLCProcess` can be seen as an interface in the object-oriented sense that allows a unified access to any process, independent of the actual process-specific implementation.

`DOLCProcess` contains five pointers, namely `local`, `init`, `fire`, `terminate`, and `wptr`, as shown in Listing 1:

- The `local` member, line 13, is a pointer to a structure that will store local information of a certain process. "Local" means that the data of the referenced structure is visible for just one process instance.

- The `init`, `fire`, and `terminate` function pointers, lines 14, 15, 16, point to the corresponding behavior functions which are described in Section 2. These three functions differentiate the behavior of each process.
- The `wptr` pointer, line 17, is a place holder for tool-chain specific process information. In the DOL-Critical functional simulation, for instance, `wptr` is used to point to an instance of the C++ process class wrapper.

```
01 #ifndef DOLCRITICAL_H
02 #define DOLCRITICAL_H
03
04 struct _dolcprocess;
05 typedef struct _dolc_local_states *DOLCLocalState;
06 typedef void *DOLCWPTR;
07 typedef void (*DOLCProcessInit)(struct _dolcprocess*);
08 typedef void (*DOLCProcessTerminate)(struct _dolcprocess*);
09 typedef int  (*DOLCProcessFire)(struct _dolcprocess*, int);
10
11 // process handler
12 typedef struct _dolcprocess {
13     DOLCLocalState          local;
14     DOLCProcessInit         init;
15     DOLCProcessFire         fire;
16     DOLCProcessTerminate    terminate;
17     DOLCWPTR                cwptr;
18 } DOLCProcess;
19
20 #define PHASE(var) {};
21
22 #endif
```

Listing 1: Header File dolc.h (DOLCProcess)

2 Behavior Function

The functional behavior of a process is split into three phases, an *initialization* phase, an *execution* phase, and a *termination* phase, which are defined in the three corresponding functions `init()`, `fire()`, and `terminate()`. Users are, however, free to add other methods, which can be called within `init()`, `fire()`, and `terminate()`.

- The `init()` function of each process is called once only. The purpose of this function is to initialize a process before running.
- The `fire(int mode)` function will be called repeatedly during runtime according to the activation pattern of a process. By calling `fire()`,

the runtime behavior of a process for a particular mode is activated. The execution modes of a process are defined as in Section 2.1.

- The `terminate()` function cleans up the runtime context of a process when this process is not to be executed (fired) again. It will be called once by the runtime scheduler before the process is terminated.

The `init()`, `fire()`, and `terminate()` must *not* contain infinite loops. For inter-process and controller-process communication, the communication functions described in Section 3 can be used inside the `init()`, `fire()`, and `terminate()` functions.

2.1 Superblock Structure and Execution Modes

Each process in the DOL-Critical XML application specification is defined as a sequence of non-preemptable execution units, known as superblocks. Each superblock is divided into phases with known computation and memory accessing requirements (upper and lower bounds on execution cycles and memory accesses). Additionally, we assume that each defined process can be executed in several modes, depending on input data availability, criticality mode, etc. The different implementations of a process for the different execution modes as well as the division of each implementation into superblock phases have to be clearly and consistently defined in the process code.

For this purpose, function `fire()` has an input parameter `mode`, defining the execution mode of a process that needs to be triggered in every occasion. The `mode` parameter must comply with the following rules:

- For each execution mode of a process, a mode macro must be declared in the header file of the process. This declaration must take the following form:

  ```
  01 #define MODE_X mode_id
  ```

  The identifier of the mode `mode_id` is an integer and it must be the same as defined in the application XML file (e.g., `0` for degraded mode, `1` for normal mode, etc.). The name of the defined macro must have the prefix `MODE_` and an arbitrarily selected suffix. It is prohibited to define any other macro having "`MODE_`" as its prefix.

- Upon calling the `fire()` function, only the defined macro may be used, not the integer identifier itself:

  ```
  01 fire(p, MODE_DEGRADED);
  ```

- In the body of the `fire()` function, a `switch(mode)` structure defines the behavior of a process under the different execution modes, as shown, for instance, for the `square` process in Listing 4.

  For each execution mode of a process, the corresponding code is divided into superblock phases by using the pre-defined macro `PHASE(phase_id)` at the beginning of every phase. The identifier `phase_id` of each phase is a string (contained in "") and it must be the same as the corresponding phase `name` defined in the application XML file.

3 Communication Interface

There are two sets of communication interface, (i) one for transmitting data over blackboard and mailbox data channels, and (ii) one for transmitting events among processes and process controllers, e.g., control data from a process to the controller of another process, dictating the restart of the latter. The primitives of the two interface sets are presented in the following sections.

3.1 Data Channel Communication Interface

All processes use uniform communication primitives, which are summarized below:

- `void DOLC_read(void *port, DOLCData *buf, int len, DOLCProcess *p)` attempts to read `len DOLCData` elements from port descriptor `port` into the buffer starting at `buf`. Reading is non-blocking (the function returns immediately if no data are available, with `buf->valid = false`) and semaphore-protected (data of a channel cannot be read and written simultaneously). It is non-destructive if `port` is an input port from a data channel of type *blackboard* and destructive for a data channel of type *mailbox*.
- `int DOLC_write(void *port, DOLCData *buf, int len, DOLCProcess *p)` attempts to write `len` elements from the buffer starting at `buf` to port descriptor `port`. Writing is non-blocking and semaphore-protected (only one writer can update a channel's contents at a time and a channel cannot be read and written simultaneously). It is overwriting if `port` is an output

port to a data channel of type *blackboard* and non-overwriting (elements are appended) for a data channel of type *mailbox*. As a special case, writing invalid data (`buf->valid = false`) to a data channel of type mailbox can be used to effectively *erase* its previous contents (overwriting operation). If one attempts to exceed the capacity of a mailbox (by setting `len` to a value greater than the "free" elements of the mailbox), `DOLC_write` returns `1`. On the other hand, a return value of `0` indicates that writing was completed successfully.

Note that the parameter `DOLCProcess *p` of the `DOLC_read()` and `DOLC_write()` primitives is used to store the address of the caller process.

3.1.1 Transmitted Data

The data that can be read from or written to data channels (either blackboards or mailboxes) are defined by the data structure shown in Listing 2. The `DOLCData` structure includes a boolean variable `valid`, indicating the validity of the data, along with a pointer `ptr` to the data values that are to be read or written and an integer variable `size`, defining the size of a single channel element (token) in bytes. The data values may be of any type, integrated or used-defined. Therefore, type casting from the `void*` pointer to the pointer of the corresponding data type is necessary upon every assignment to `ptr`.

New variables of type `DOLCData` can be declared and properly initialized using macro `DOLCDataVar`.

```
01 #ifndef DOLCRITICAL_H
02 #define DOLCRITICAL_H
03
04 typedef struct _dolcdata {
05    bool valid;  // validity bit
06    void* ptr;   // pointer to actual data
07    unsigned int size; // size of one element in bytes
08 } DOLCData;
09
10 #define DOLCDataVar(type, var)  type _data_##var; \
11     DOLCData var; var.valid=false; \
12     var.size=sizeof(type); var.ptr = &_data_##var
13
14 #endif
```

Listing 2: Header File dol.h (DOLCData)

Note that the third argument of all communication primitives is `DOLCData *buf`.

3.1.2 Port Access

The first argument of the communication primitives is `void *port`. The following rules apply for this parameter.

- For each port which is accessed within a process, a port macro must be declared in the header file of the process. This declaration must take the following form:

  ```
  01 #define PORT_X "portname"
  ```

  The name of the port must be the same as defined in the application XML file. If the name of the port is an integer, `portname` is simply the corresponding integer. If the name of the port is a string, `portname` is the string in quotes. The name of the defined macro must have the prefix `PORT_` and an arbitrarily selected suffix. It is prohibited to define any other macro having `PORT_` as its prefix.
- Within a call to a communication primitive, only the defined macro may be used, not the string itself:

  ```
  01 DOLC_read((void*)PORT_X, &buffer, 1, p);
  ```

3.2 Control Channel Communication Interface

This communication interface is used to exchange control data among processes and controllers through the XML-defined control channels. we define two primitives that can be called within the `fire()` function of a process, namely `DOLC_send_event()` and `DOLC_yield()`.

- `void DOLC_send_event(void *port, void *event, DOLCCProcess *p)` attempts to send an `event` encoded as a string to port descriptor `port`. The event is sent asynchronously, in zero time to a controller process other than the process's own controller, e.g., in order to change the execution mode or restart another process of the application. `port` specifies the control channel, through which the event is sent.

– `void* DOLC_yield(void *event, DOLCProcess *p)` interrupts the execution of function `fire()` of process `p` in order to check if its controller has received any new event. For example, the controller process may have to restart process `p` because of an event that it received from another process of the application or from the system scheduler. Process `p` will be informed about this decision upon calling `DOLC_yield` with the `event` argument being set to `null`. If process `p` needs to also send an event to its controller, the `event` argument should be set accordingly. The "response" from the controller is encoded by the appropriate event identifier and returned by `DOLC_yield` (`null` if no action needs to be taken).

Note that undefined events will be discarded by the receiving process. Port access is defined as described in Section 3.1.2.

3.2.1 Event Access

The second (first) argument of `DOLC_send_event` (`DOLC_yield()` is `void *event`. The following rules apply for this parameter.

– For each event which is accessed within a process, a event macro must be declared in the header file of the process. This declaration must take the following form:

```
01 #define EVENT_X "eventname"
```

The name of the event must be the same as defined in the application XML file (e.g., `restart`) and has to be surrounded by quotes. The name of the defined macro must have the prefix `EVENT_`. It is prohibited to define any other macro having `EVENT_` as its prefix.

– Within a call to a communication primitive, only the defined macro may be used, not the string itself:

```
01 DOLC_send_event((void*)PORT_P1, (void*)EVENT_RESTART, p);
```

4 Further Rules

Besides the guidelines described above, there are some additional rules concerning program code that should be executed either using func-

tional simulation or on the MPPA platform. Some of them are summarized below:

- The structure for saving the user-defined local data of a process needs to be named complying with the following naming convention: `Processname_State`, that is, the name of the process with the first letter capitalized followed by `_State`.
- When a function has a pointer to a `DOLCProcess` structure as an argument, such as `square_init()` and `square_fire()` in the example of Listing 4, the argument name needs to be "`p`".

5 Example

An example implementation of a process complying with the described coding style is shown in Listings 3 and 4. The listings present the header and the C file of process `square`, which, in normal mode, reads a floating-point value from a data channel, computes the square value of it and then, writes it to a data channel. In degraded mode, `square` writes invalid data to the output data channel. In the header file, the ports and the execution modes of this process, the local data structure, and the behavior functions are declared. In the corresponding C file, the behavior functions are defined.

```
01 #ifndef SQUARE_H
02 #define SQUARE_H
03
04 #include<dolc.h>
05
06 #define PORT_INPUT "p1"
07 #define PORT_OUTPUT "p2"
08 #define MODE_NORMAL 1
09 #define MODE_DEGRADED 0
10 #define EVENT_FINISH "finish"
11
12 typedef struct _dolc_local_states{
13     int index;
14 } Square_State;
15
16 void square_init(DOLCProcess *);
17 void square_fire(DOLCProcess *, int);
18 void square_terminate(DOLCProcess *);
19
20 #endif
```

Listing 3: Header File of *square* Process

```
01 #include <stdio.h>
02 #include "square.h"
03
04 void square_init(DOLCProcess *p) {
05     p->local->index = 0;
06 }
07 void square_fire(DOLCProcess *p, int mode) {
08     float i;
09     DOLCDataVar(float, idata);
10
11     switch(mode){
12         case MODE_NORMAL:
13             PHASE("mode_normal_phase1");
14             DOLC_read(((void*)PORT_INPUT, &idata, 1, p);
15             if (idata.valid) {
16                 i = *((float*) idata.ptr);
17                 *((float*) idata.ptr) = i * i;
18                 idata.valid = true;
19             }
20             DOLC_write((void*)PORT_OUTPUT, &idata, 1, p);
21
22             PHASE("mode_normal_phase2");
23             p->local->index++;
24             printf("Process square has been fired %d times.", p->local->index);
25
26             DOLC_yield((void*)EVENT_FINISH, p);
27             return;
28
29         case MODE_DEGRADED:
30             PHASE("mode_degraded_phase1");
31             idata.valid = false;
32             DOLC_write((void*)PORT_OUTPUT, &idata, 1, p);
33             DOLC_yield((void*)EVENT_FINISH, p);
34             return;
35         }
36 }
37 void square_terminate(DOLCProcess *p) {
38     // freeing pointers of Square_State, erasing data from mailboxes
39     // would happen here (not needed in this simple example)
40 }
```

Listing 4: C File of *square* Process