Obelix Configuration Tool Back-End, programmers manual[*]

Anders Davidsson
adn99006@student.mdh.se

August 4, 2000

[*]Part of the Asterix frame-work

Contents

1 Introduction

This document is a programmers manual to the Obelix configuration tool back-end. The back-end is a code generator package which consists of routines for generating Asterix Real-Time Kernel initiation files for the Lego Mindstorm RCX micro-controller. An Obelix syntax tree (intermediate representation of Obelix Configuration Tool) representing a specific configuration is the input to the generator. Two files are generated, tasks.c and tasks.h, which contains variable declarations, initiations and a bunch of defines of different constants. The routines should be applied in a specific order described later in this document.

The code generator is implemented in four files: emit.c, emit.h, eutils.c and eutils.h. The code generating routines are present in emit.c and utility routines, called from several code generating routines, are stored in eutils.c.

Each routine in this document are described in the same way. First the syntax are explained followed by the semantics. Each parameter (input) to the routine are stated and also the output. All other routines called are enumerated and finally we make comments, if necessary.

2 Constants

There are some constants defined which are target system dependent.

- `STACKTYPE` - Data type of the stack.
- `CLOCKTYPE` - Data type in which the clock-resolution should be given.
- `BITSINTYPE` - Number of bits in largest data type the processor supports.
- `DEF_ERROR_ROUTINE` - The name of the kernel defined error routine.
- `CFILENAME` - The name of the C-file to create.
- `HFILENAME` - The name of the H-file to create.

These constants holds processor specific items and should be changed if some other hardware is used.

3 Global variables

A few global variables are defined in emit.c.

- `FILE *cFile` - The handle to the C-file.
- `FILE *hFile` - The handle to the H-file.
- `NameListEl *routine_list` - A list of start-routine names that has been used.
- `NameListEl *error_routine_list` - A list of error-routine names that has been used.
- `int no_tasks` - To store the number of tasks in the system.
- `int no_semaphores` - To store the number of semaphores in the system.
- `int error_H_flag` - To settle if the default, kernel defined error-routine is prototyped.
- `int wf_flag` - To settle if there is wait-free communication in the system.

4 Code generating routines

In this section all the code generating routines are enumerated. The routines are described in the same order as they should be called. The first routine described (*createFile*) is the one that should be called from the Bison file. The *createFile* routine then calls the other necessary routines.

4.1 Start

4.1.1 Generate c-files

Syntax: `void createFile(tTree root)`

Semantic: Generates the necessary initialization files for the Asterix real-time kernel on the basis of a syntax tree.

Input: `root` - The syntax tree

Output: – .

Calls:

- `initFiles()`
- `createTestModeSetUp()`
- `mDefines()`
- `createFunctionProtos()`
- `createStacks()`
- `createDispatchList(()`
- `createResolution()`
- `createSynch()`
- `createWaitFree()`
- `createReadyQueue()`
- `createTCBList()`
- `defineInterupts()`
- `completeFiles()`
- `closeFiles()`
- `freeNameList()`

Comments: This is the routine which should be called from the Bison file.

4.2 File initiation

A few files has to be included in the generated initiation files. We make the includes in tasks.h, which in turn is included in tasks.c. The following files are included in tasks.h:

- `<stdlib.h>`
- `<types.h>`
- `<kertypes.h>`
- `<tcb.h>`
- `<t_return.h>`
- `<system.h>`

4.2.1 Initiate the files to create

Syntax: `void initFiles(void)`

Semantic: Open up the files to create, write the header and includes.

Input: – .

Output: – .

Calls: – .

Comments: – .

4.2.2 Initiate for NORMAL or TEST system mode

Syntax: `void createTestModeSetUp(tTree program)`

Semantic: Control if the system should bun in test-mode. If so, initiate for test-mode.

Input: `program` - The syntax tree

Output: – .

Calls: – .

Comments: – .

4.2.3 Generate constants

Syntax: `void mDefines(tTree mode)`

Semantic: Defines the constants NRTCBS and NRTASKS.

Input: `mode` - A syntax tree mode branch.

Output: – .

Calls: `countTasks()`

Comments: – .

4.3 Function prototypes

Each task start- and error-routine are prototyped in tasks.h. An error-routine must take a parameter of STACKTYPE type. STACKTYPE is a constant described earlier in this document. If no error-routine is assigned by the application programmer, a default error-routine, defined in the kernel, is associated with the task. Each start- and error routine are prototyped once only. It is possible for multiple tasks to call the same start- and error routine.

4.3.1 Generate function prototypes

Syntax: `void createFunctionProtos(tTree tasks)`

Semantic: Generates function prototypes for each task start and error routine

Input: `tasks` - The tasks branch of a mode in the syntax tree.

Output: – .

Calls: `createPrototype()`

Comments: – .

4.3.2 Generate a specific function prototype

Syntax: `void createPrototype(tTree task)`

Semantic: Generates the start and error function prototype for *task*, if it does not already exist.

Input: `task` - A task in the syntax tree

Output: – .

Calls:

- `isInList()`
- `createPrototype()`

Comments: The function is recursive and goes through the whole list of tasks in the parameter.

4.4 Stacks

Each task in the system must have a own stack. The stacks are declared in tasks.c and made external in tasks.h. The kernel requires, in addition to the application programmer assigned stack size, two stack elements to store kernel overhead.

4.4.1 Generate user stack declarations

Syntax: `void createStacks(tTree stacks)`

Semantic: Generates the user stack declarations.

Input: `tasks` - The task branch of a mode in the syntax tree.

Output: – .

Calls: `createTaskStack()`

Comments: – .

4.4.2 Generate one task stack

Syntax: `void createTaskStack(tTree task)`

Semantic: Generates code for a task stack and calls itself with next task as argument. A stack is named by *stack* followed by the current task priority (which is unique). For example *stack22*.

Input: `task` - A task in the syntax tree.

Output: – .

Calls: `createTaskStack()`

Comments: The function is recursive and goes through the whole list of tasks in the parameter. This part of Obelix is very target-system dependent. The type of the stacks are declared to be STACKTYPE, which is a constant described earlier in this document.

4.5 dispatch-list

The dispatch-list is an array of *dispatch_t* elements. The list is declared and initiated in tasks.c, and made external in tasks.h.

4.5.1 Generate the dispatch-list declaration

Syntax: `void createDispatchList(tTree tasks)`

Semantic: Generates code for the kernel dispatch-list.

Input: `tasks` - The tasks branch of a mode in the syntax tree.

Output: – .

Calls:

- `countTasks()`
- `createTaskInDispatchList()`

Comments: – .

4.5.2 Generate the code for a specific task in the dispatch-list

Syntax: `void createTaskInDispatchList(tTree task)`

Semantic: Generates code for a task in the kernel dispatch-list. A task is specified by: *state, left-to-deadline, stack-pointer, start-address, CCR, R0...R6 (R0 holds optional arguments), offset.*

Input: `task` - A task in the syntax tree

Output: – .

Calls: `createTaskInDispatchList()`

Comments: The function is recursive and goes through the whole list of tasks in the parameter.

4.6 Clock resolution

Each mode in an Asterix system has a clock resolution that is set by this function. The resolution is declared and initiated in tasks.c and made external in tasks.h.

4.6.1 Set the clock resolution

Syntax: `void createResolution(int res)`

Semantic: Sets the clock resolution of a mode.

Input: `res` - Microseconds between timer interrupts, must be at least 2 and even.

Output: – .

Calls: – .

Comments: – .

4.7 Synchronization and task-list

The following routines generates code for task synchronization, by semaphores and signals, and code for the kernel task-list. Semaphores, signals and the task-list are declared and initiated in tasks.c and made external in task.h.

The task-list is an array of all tasks and semaphores in the system. A task is represented by it's index in the tcb-list. Semaphores are represented by 0, this means that task 0 (the idle-task) in the tcb list cannot access semaphores. The task-list is ordered by priority and ceiling. The higher priority/ceiling, the higher index in the task-list.

A semaphore is defined by an array of tasks that are authorized to access the semaphore, the number of tasks that are authorized to access the semaphore and the semaphore task-list index. A task is represented by the index in the task-list.The number of semaphores in the system has to be stated. To identify the semaphore, the application programmer assigned identity is mapped to an unique integer.

A signal is defined by an array of tasks that are authorized to access the signal and the number of them, the tasks that are triggered by the signal and the number of them. A task is represented by the index in the task-list. To identify the signal, the application programmer assigned identity is mapped to an unique integer.

4.7.1 Create code for semaphores and signals

Syntax: `void createSynch(tTree mode)`

Semantic: Generates the code for semaphores and signals, it also builds up a representation of the task-list and generates code for the task-list.

Input: `mode` - A mode branch of the syntax tree

Output: – .

Calls:

- `buildTaskList()`

- `createSemaphores()`

- `createSignals()`

- `createTaskList()`

Comments: – .

4.7.2 Build an internal task-list

Syntax: `void buildTaskList(tTree mode, tTree task_list[])`

Semantic: Build up an internal (internal) representation of the task-list, the task-list consists of tasks (tcb's) and semaphores ordered by priority and ceiling, lowest first.

Input:

- `mode` - A mode branch of the syntax tree

- `task_list` - An empty array of tTree that will hold the task-list representation

Output: – .

Calls: – .

Comments: – .

4.7.3 Create semaphores

Syntax: `void createSemaphores(tTree in_sem, tTree task_list[])`

Semantic: Generates the semaphore code

Input:

- `in_sem` - The semaphore branch in the syntax tree

- `task_list` - The internal representation of the task-list

Output: – .

Calls:

- `createSemUserArray()`
- `countUsers()`
- `getSemaphoreTaskListIndex()`

Comments: – .

4.7.4 Array of semaphore users

Syntax: `void createSemUserArray(tTree semaphores, int no, tTree task_list[])`

Semantic: Creates the array of users for a semaphore (the first semaphore in the semaphore input list).

Input:

- `semaphores` - The semaphore to work on.
- `no` - Name information.
- `task_list` - The internal representation of the task-list.

Output: – .

Calls: `getUserTaskListIndex()`

Comments: The identity of the array is the string *sem* followed by an unique integer, for example *sem0*.

4.7.5 Create signals

Syntax: `void createSignals(tTree in_sig, rTree task_list[])`

Semantic: Generates the signal code.

Input:

- `in_sig` - The signal branch of a mode in the syntax tree.
- `task_list` - The internal representation of the task-list.

Output: – .

Calls:

- `createSigUserArray()`
- `createSigBlockedArray()`
- `countUsers()`

Comments: – .

4.7.6 Array of signal users

Syntax: `void createSigUserArray(tTree signals, int no, tTree task_list[])`

Semantic: Creates the array of users for a signal (the first signal in the list).

Input:

- `signals` - The signal to work on.
- `no` - Name information.

- `task_list` - The internal representation of the task-list.

Output: – .

Calls:

- `getUserTaskListIndex()`

- `countUsers()`

Comments: The identity of the array is made up by the string *sig* followed by a unique integer and *_1*, for example *sig0_1*.

4.7.7 Array of tasks that is blocked by a signal

Syntax: `int createSigBlockedArray(tTree signals, int no, tTree task_list[])`

Semantic: Generates an array of tasks that is blocked by the signal and returns the number of them. A task is expressed by its index in the task-list.

Input:

- `signals` - The signal to work on.
- `no` - Name information.
- `task_list` - The internal representation of the task-list.

Output: The number of tasks that is blocked by the signal

Calls:

- `getUserTaskListIndex()`

- `countUsers()`

Comments: The identity of the array is made up by the string *sig* followed by a unique integer and *_2*, for example *sig0_2*.

4.7.8 Create the task-list

Syntax: `void createTaskList(tTree task_list[])`

Semantic: Generates the code for the task-list, that is all tasks and semaphores in the system.

Input: `task_list` - The internal representation of the task-list.

Output: – .

Calls: – .

Comments: – .

4.8 Communication

The routines in this section generates code for the inter-process communication in the system. In this version of Obelix, only wait-free IPC is supported. A wait-free channel is specified by three parts:

- A *message-type* array of length *no_buffers*. This is the physical channel where the communication takes place. *message-type* is the application programmer assigned type of the wait-free channel and *no_buffers* is the number of buffers in the wait-free channel. This variable is named by *wfbuf*X*buf*, where X is a unique integer. All elements in the array are initiated to be empty.

- A *uint16* (for the RCX target-system) array of length *no_buffers*. This array keep track of which buffers, in the channel, that are currently accessed, a kind of semaphore array. This one is named by *wfbuf*X*bufsem*, where X has the same value as in the *wfbufXbuf* variable. All elements in the array are initiated to 0.
- A *llnode_t* array, also of length *no_buffers*, which will keep track of which buffer that stores the oldest value. The name is *wfbuf*X*oldest*, where X, again, has the same value as in the *wfbufXbuf* variable. Each *llnode_t* element consists of it's own index in the *oldest* array and the address of next element in the array.

When there is wait-free communication in the system, each task is assigned a control buffer that will store information on the tasks wait-free communication. The control buffer is an array of *bufctrl_t* of length $task\_no\_of\_wf +1$, where *task_no_of_wf* is the number of wait-free channels used by the task. A *bufctrl_t* is defined by:

- Wait-free channel index in the wait-free channel array (*buffer*).
- The constant WF_READER if the task is reader of the channel or the constant WR_WRITER if the task is writer.
- The last position is always set to 0.

The first position in the *bufctrl_t* array is special. It describes the number of wait-free communication channels used by the task. The first position in the element settles that and the other two is always set to 0. The following elements describes the wait-free channels used.

The code for the wait-free channels is a *buffer_t* (perhaps not so good type name) array of length *no_of_wf*. *no_of_wf* is the number of wait-free communication channels in the system. I there is no wait-free channels at all, no wait-free code will be written. Each *buffer_t* consists of four addresses:

- address to the first position in the physical channel array (*wfbufXbuf*)
- address to the first position in the semaphore array (*wfbufXbufsem*).
- address to the first position in the oldest array (*wfbufXoldest*).
- address to the last position in the oldest array (*wfbufXoldest*).

All these variables are declared and initiated in tasks.c and made external in tasks.h. For further information about the data-structures used in the kernel see [].

4.8.1 Create code for the wait-free communication

Syntax: `void createWaitFree(tTree mode)`

Semantic: Generates the wait-free communication code.

Input: `mode` - A mode branch of the syntax tree

Output: – .

Calls: `createTaskBufCtrl()`

Comments: – .

4.8.2 Wait-free control buffers

Syntax: `void createTaskBufCtrl(rTree wait_free, tTree tasks)`

Semantic: Generates the code for all tasks control buffer

Input:

- `wait_free` - The wait-free branch in a mode.
- `tasks` - A task branch in a mode, could be hard or soft.

Output: – .

Calls:

- `getWfEl()`
- `checkWf()`
- `isWriter()`
- `freeWfElList()`

Comments: Elements of type *wfel* are used to build up an internal representation of each tasks control buffer.

4.9 Ready queue

The ready queue is declared and initiated in tasks.c and made external in tasks.h. The ready-queue is defined by an array of the largest data type on the target system. The size of the array are decided by the number of tasks in the system (tcb's and semaphores). There must be, at least, as many bits in the array as there are tasks in the system. For example, it the system consists of 20 tasks and the largest type has 16 bits, the ready queue array has to contain 2 elements of this type (that is 32 bits).

4.9.1 Ready-queue

Syntax: `void createReadyQueue(void)`

Semantic: Declares and initialize the ready-queue.

Input: – .

Output: – .

Calls: – .

Comments: – .

4.10 TCB-list

The TCB-list is declared and initiated in tasks.c and made external in tasks.h. The TCB-list is an array of all tasks (application programmer designed) in the system and it is static during execution. The dispatch-list is the dynamic part of the TCB-list.

4.10.1 TCB-list

Syntax: `void createTCBList(tTree tasks)`

Semantic: Declares and initialize the TCB-list. A task in the list is represented by *period, offset, deadline, start-address, argument, (errorfunc_t)errorHandler, CCR default value, task wait-free bufctrl.*

Input: `tasks` - A task branch in a mode, could be hard or soft.

Output: – .

Calls:

- `countTasks()`
- `createTaskInTCBList()`

Comments: – .

4.10.2 Tasks in the TCB-list

Syntax: `void createTaskInTCBList(tTree task)`

Semantic: Generates code, declares and initialize, for all tasks in the TCB-list. A task is defined by... ?????? The function is recursive.

Input: `task` - A task in the syntax tree, which, in fact, is a list of tasks

Output: – .

Calls: `createTaskInTCBList()`

Comments: – .

4.11 End of files

This section describes the routines that finish off tasks.c and tasks.h.

4.11.1 Interrupts

Syntax: `void defineInterupts(void)`

Semantic: Defines interrupts.

Input: – .

Output: – .

Calls: – .

Comments: – .

4.11.2 End of initiation files

Syntax: `void completeFiles(void)`

Semantic: Writes the end of file...

Input: – .

Output: – .

Calls: – .

Comments: – .

4.11.3 Close files

Syntax: `void closeFiles(void)`

Semantic: Closes the files.

Input: – .

Output: – .

Calls: – .

Comments: – .

5 Utility routines

This section describes utility routines used by the code generating routines.

5.1 Counters

5.1.1 Count tasks

Syntax: `int countTasks(tTree tasks)`

Semantic: Counts the number of tasks in a tasks branch.

Input: `tasks` - A task branch in the syntax tree.

Output: Number of tasks in the tasks branch.

Calls: – .

Comments: – .

5.1.2 Count users

Syntax: `int countUsers(tTree users)`

Semantic: Count the number of users in a user branch of the syntax tree.

Input: `users` - A user branch of the syntax tree.

Output: The number of users in the branch.

Calls: – .

Comments: – .

5.2 Represent a task control buf

This section describes routines for representing tasks wait-free communication channels in order to initiate the task control buffer.

5.2.1 Allocate memory for a *wfel*

Syntax: `wfel *getWfEl(int index, int is_writer)`

Semantic: Allocates memory for a *wfel* and initiate it.

Input:

- `index` - For initiation.
- `is_writer` - For initiation.

Output: A pointer to the new element.

Calls: – .

Comments: – .

5.2.2 Free a *wfel* list

Syntax: `void freeWfElList(wfel **list)`

Semantic: Free space.

Input: `list` - A list of *wfel*.

Output: – .

Calls: – .

Comments: – .

5.2.3 Control if a task is a user of a wait-free channel

Syntax: `int checkWf(tTree task, rTree wf)`

Semantic: Settle if a task is user of a wait-free channel.

Input:

- `task` - A task in the syntax tree.
- `wf` - A wait-free channel in the syntax tree.

Output: 1 if the task is a user of the wait-free channel, otherwise 0.

Calls: – .

Comments: – .

5.2.4 Control if a task is writer of a wait-free channel

Syntax: `int isWriter(tTree task, tTree wf)`

Semantic: Settle if a task is writer of a wait-free channel.

Input:

- `task` - A task in the syntax tree.
- `wf` - A wait-free channel in the syntax tree.

Output: 1 if the task is writer of the wait-free channel, otherwise 0.

Calls: – .

Comments: – .

5.3 Keep track of the function prototypes

This section describes routines for controlling the function prototyping. If several tasks uses the same start- or error-routine, the routine should be prototyped once only.

5.3.1 Control if a function is prototyped

Syntax: `int isInList(char name[], NameListEl **list)`

Semantic: Settle if a name is in the *NameListEl*. If the name is not in the list, a new list element is created and appended to the list.

Input:

- `name` - The name to look for.
- `list` - The list to look in.

Output: 1 if the name is in the list, otherwise 0.

Calls: `getListEl()`

Comments: – .

5.3.2 Allocate memory for a *NameListEl*

Syntax: `NameListEl *getListEl(char name[])`

Semantic: Allocates memory for a *NameListEl* and initialize it.

Input: `name` - For initiation.

Output: A pointer to a new *NameListEl*.

Calls: – .

Comments: – .

5.3.3 Clean up a *NameListEl* list

Syntax: `void freeNameList(NameListEl **list)`

Semantic: Clean up.

Input: `list` - The list to free.

Output: – .

Calls: – .

Comments: – .

5.4 Find out indexes in the task-list

5.4.1 Get a semaphore task-list index

Syntax: `int getSemaphoreTaskListIndex(tTree semaphore, tTree task_list[], int no_tasks, int no_semaphores)`

Semantic: Finds out which index in the task-list a semaphore has.

Input:

- `semaphore` - A semaphore branch in the syntax tree.
- `task_list` - The task-list representation.
- `no_tasks` - Number of tasks in the system.
- `no_semaphores` - Number of semaphores in the system.

Output: The task-list index of the semaphore if the semaphore is found, otherwise -1.

Calls: – .

Comments: – .

5.4.2 Get a task index in the task-list

Syntax: `int getUserTaskListIndex(tTree users, tTree task_list[], int no_tasks, int no_semaphores)`

Semantic: Finds out which index in the task-list a task, expressed as a user, has.

Input:

- `users` - A user branch in the syntax tree.
- `task_list` - The task-list representation.
- `no_tasks` - Number of tasks in the system.
- `no_semaphores` - Number of semaphores in the system.

Output: The task-list index of the task (user) if the task is found, otherwise -1.

Calls: – .

Comments: – .