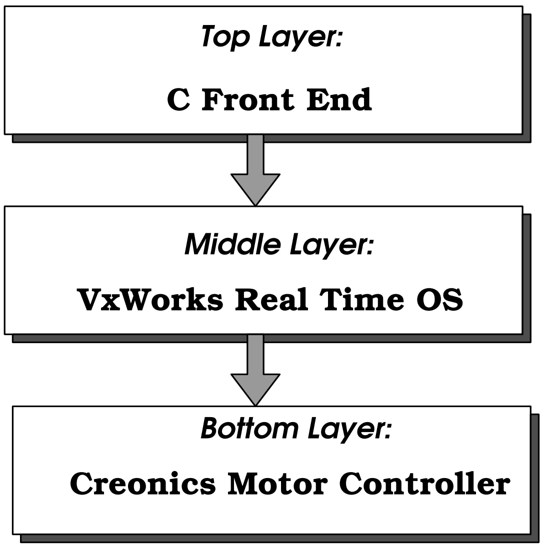
3. Software

In this section we briefly describe the structure of the software for the ED I robot, and explain some of the basic principles underlying the MOSAIC/ED I system. The system is composed of two components: (a) **frontend**: comprising of a C programming system with access to C library routines that perform various actuation and sensing operations; and (b) **backend**: a `VxWorks`^{tm} based real-time system that implements the actuation and sensing operations. In the future, we plan to implement a graphical interface that will operate above the current frontend and make the man/machine interface substantially friendlier.



The software for the ED I robot is organized in a layered manner. It consists of a top layer of C environment, visible to the users, where the programming of ED I is done by calls to routines in a C library. The middle level translates these calls to low-level commands for the Creonics controllers and is accomplieshed by the VxWorks real time systems. The lowest level executes these commands. All but the top layer is completely invisible to everyone except the systems developers.

Figure 6: **The software organization of the ED I system**

3.1 Programming the MOSAIC/ED1 System

The MOSAIC/ED1 system is usually programmed in a high level language such as C or Pascal, where the low-level actuation and sensing operations are accomplished by a set of "library" calls. In this way, the intermediate level details of real time constraints and the very low-level servo parameters remain completely hidden from the novice users. The real time operating system `VxWorks`^{tm} handles the synchronization of and mutual exclusion among various low-level operations via internal queues and semaphores and thus provides the user the illusion of smooth concurrent operations of the axis actuators and the sensors; additionally, it ensures that these operations are executed in a timely manner. A detailed discussion of the intermediate level is discussed in details later in this section. The *low level axes driver interpreter* (see figure 3.2.4) then translates the commands dispatched by the `VxWorks`tm into still lower-level commands that are executed by the Creonics motor controller card (see the preceding section for more details).

We must warn that the above picture of the software organization is occasionally violated inside the intermediate level of `VxWorks`tm real time systems, primarily for the sake of occasional efficiency improvement. However, for the present document, we shall ignore these details in order to avoid confusion.

3.1.1 The Library Routines

Currently, executing a high-level robotic task on ED I (e.g. pick-and-place, tracking or peg-in-the-hole) involves simply writing C code sprinkled with library calls. Thus, the user accomplishes all of the numerical computation, user interface, error handling, data structure organization and temporal ordering of the actuation and sensing using the primitives provided by the C language, a set of standardized data structures, and some related libraries (e.g. `math.h`). The compiled and linked code then automatically runs the robot in the intended manner.

In this subsection, we describe in substantial details, the library facilities available and illustrate their usage by a practical example. The example is taken from a student-project written at a preliminary stage of the software development. (See appendix A, for the full listing.) Complete details of the library will be provided in a separate Programmer's manual and will also be available on-line.

- **The `AXIS_STRUCT` Data Structure and Related Routines:** The most important data structure used is the one defining the characteristics of an *axis*. Its definition, shown below, is logically divided into four parts:

1. *Identification and general axis information*;
2. *Limits and hardwired constants for the axis structure* (These are read from configuration files);
3. *Physical status information* (e.g. current position, velocity, etc.);
4. *Software status information* (e.g. semaphores, flags etc.).

```
typedef struct axis_struct {
  /* Identification and general info. */
  char Id[IDENT_LENGTH];    /* The full name of the axis */
  char Name;                /* A single character identifier for */
                            /* the same purpose as Id[]*/


  /* Limits and constants */
  /* ...omitted... */

  /* Physical status */
  /* ...omitted... */
  double Current_Origin;    /* Origin of the current coordinate */
                            /* system, with respect to the 'original'. */
  double Current_Pos_Limit;
  double Current_Neg_Limit;

  /* Software status */
  SEM_ID Busy;              /* True if axis is ready for a command */
  char Status;              /* Byte encoding of various info */
  char Note[MAX_MSG_SIZE];  /* Used for communication with driver */
} AXIS_STRUCT;
```

One important field of the data structure above is the `Note` field, which serves as a buffer where low level commands are placed before being fed to the Creonics motion control card driver and where results are read from. How exactly this is accomplished by the `VxWorks`tm will be explained further in the next subsection.

Two other functions that manipulate the `AXIS_STRUCT` are `axis_cmd` and `axis_open`. They mask the operations on the semaphore `Busy`. The first one does a V operation on the semaphore, allowing other commands to "jump in", while the second one does a P operation, eventually blocking out other commands on the axis.

Since the axes of the robot are stored in a table a third function is useful, `find_axis`, which retrieves an "axis identifier" given a character mnemonic for it.

- **Other Major Routines:** In the current implementation there are a number of routines that can be used to write 'higher level' commands, or that can serve as a minimal protocol for the functioning of ED I robot arm. Following is a list of a few of the currently available low-level routines:

Routine	*Classification*
`joint_move` `home`	Movements related
`on` `off`	System control

▶ <u>`home` routine:</u> takes an axis identifier (a single character) and performs the *homing sequence* for it. After the homing sequence is completed (i.e. the axis has hit its limit switches), we have a firm reference point starting from which the axis movements can be accurately computed.

▶ <u>`joint_move` routine:</u> This routine takes four parameters

1. A (double) float value that specifies the "new location" of the first joint of the robotic arm (θ_1),
2. Another double that specifies the new location of the second joint (θ_2),
3. A (double) float that specifies how much "time" must be used by the movements,
4. A flag that specifies whether the movements must be *absolute* or *relative*.

As an example, if the robotic arm is in Joint Space position $\langle \theta_1, \theta_2 \rangle = \langle 0.0, 0.0 \rangle$ then the call

```
joint_move(45.0, 30.0, 5.0, 0);
```

will cause the joints to move to the (absolute) position $\langle 45.0, 30.0 \rangle$ in the current coordinate reference in Joint Space, in 5 seconds. The subsequent call

```
joint_move(45.0, -15.0, 5.0, 1);
```

will move the joints to move to the (Joint Space position) $\langle 90.0, 15.0 \rangle$.

There is an identical procedure for the other two distal joints, called `last_joint_move`.

▶ <u>`on` routine:</u> is used to power on a specified axis.

- ▶ <u>`off` routine:</u> is used to power off a specified axis.

The `on` and `off` routines are therefore viable means to operate the robot safely under program control.

3.1.2 An Example Program: the `stroke` Routine

In this subsection, we illustrate the natural style of C programming for the ED I robot, written using the library routines discussed earlier. The example, we have chosen for this purpose, is a rather simple routine that moves the end effector in a *straight line* in the physical space. The full listing appears in the appendix A.

The routine uses the `joint_move` routine and produces a simple "stop-and-go" movement through the *via points* (see [7] for terminology). The via points are generated by a version of Taylor's *bounded deviation* algorithm (see [25] for details).

```
/* stroke --
 * Moves the manipulator from the current position (x-start, y-start)
 * to the position x, y in the specified amount of time.
 */

int stroke(x, y, time)
     double x;
     double y;
     double time;
{
 int i = 0;

 double init_theta1 = Axis[0].Last_Pos_In_Joint;
 double init_theta2 = Axis[1].Last_Pos_In_Joint;

 double final_theta1 = 0.0; /* Dummy final values */
 double final_theta2 = 0.0; /* Dummy final values */
 /* ...omitted... */
 int knot_count = 0;
 knot_struct knot_sequence[MAX_KNOTS_NUMBER]; /* knot_struct, defined elsewhere */

 /* Functions used */
 double compute_deviation();
 double taylor_bdev();
 int    count_knots();
 int    joint_move();
 /* ...omitted... */

 /* First compute the Joint Space position of the final point. */

 arm_inverse_kinematics(&final_theta1, &final_theta2, x, y);



 /* Now compute the sequence of knots needed to */
 /* approximate the straight line in cartesian space. */

```

```
  /* Initialize the values for the points already obtained. */
  /* ...omitted... */

  taylor_bdev(init_theta1, init_theta2,
              final_theta1, final_theta2,
              knot_sequence,
              0,
              MAX_KNOTS_NUMBER - 1);

  knot_count = count_knots(knot_sequence);
  new_time   = time / knot_count;
  for (i = 0; i < knot_count; i++) {
    joint_move(knot_sequence[i].x, knot_sequence[i].y, new_time,
               ABSOLUTE_MOVE);
    taskDelay(iround((time / knot_count) * TICKS_SEC * 1.5));
    /* Delay to be sure that move has ended! The 1.5 factor is just */
    /* a safety measure. Note, ´taskDelay´ is a VxWorks primitive */
  }
  return OK;
}
```

The `stroke` routine is straightforward C code[2]. The only routines that need some explanations are `arm_inverse_kinematics` and `taylor_bdev`. The routine `count_knots` is obvious.

The routine `arm_inverse_kinematics` computes the Joint Space position of the manipulator, given two values in Cartesian Space. The values are returned in the first two arguments, which are passed 'by-reference.'

The routine `taylor_bdev` computes the via points in Joint Space (by calling also `arm_inverse_kinematics`) using the Taylor's algorithm, mentioned earlier. The resulting via points are stored in the array `knot_sequence` and are eventually fed to `joint_move`.

This is just a very simple example which produces a "stop'n'go" movement of the manipulator. The better version also produces values for velocities and accelerations and needs to set up a special communication link with the low level hardware.

3.1.3 How Programs Get Executed

Once the programs are written and compiled into an object file[3], they can be loaded into the `VxWorks`tm address space. A simple machinery (explained in the next section) links them to a parser that can be used to invoke them.

Alternatively they can be invoked directly from the `VxWorks`tm shell (see [37] for details). This turns out to be very useful for debugging purposes.

- **The `VxWorks`^{tm} Real Time Operating System:** The core of the MOSAIC/ED-I architecture is the `VxWorks`^{tm} real time operating system manufactured and marketed by Wind River System [37].

[2] The comments of the form `/* ...omitted... */` indicate omitted code and is done to avoid clutter. E.g. most error checking code is left out from what is shown here.

[3] In reality, the programs are cross-compiled for the 68020 architecture.

VxWorks^{tm} RT OS Features	
Manufacturer	Wind River System
Version	5.02b
Memory Model	shared space
RT Feature	Multitasking Preemptive Priority Scheduling and Priority Inversion
IPC Features	Semaphores, Message Queues
Connectivity	TCP/IP

Table 9: `VxWorks`^{tm} **real time operating system features**

Its characteristics as a RT OS are listed in table 3.1.3. An advantage of `VxWorks`tm is its tight integration with a UNIX workstation. In our case `VxWorks`tm runs on a Motorola MVME147s board and communicates with the UNIX host through a standard Ethernet connection. Programs are developed and compiled (but *not* linked) on the UNIX host and linked/loaded into the `VxWorks`tm memory through the network.

The `VxWorks`tm uses a shared memory model. The basic unit of execution is the *task*. Each task runs a C function as a separate entity in a shared addressing space. Synchronization primitives as semaphores and message queues are provided for basic IPC (Inter Process Communication) programming.

When `VxWorks`tm is first loaded (i.e. at the end of its boot phase) it starts a special session of its own *shell* on the system console or on a remote terminal[4] (most likely an `xterm` on the UNIX host running X-Windows).

The `VxWorks`tm shell acts almost like a C 'interpreter.' The object files produced in the cross-compilation phase must contain the symbol table for them. Once these object files are loaded into the `VxWorks`tm workspace, a function can be simply called by typing its name and arguments at the shell. `VxWorks`tm spawns a task for the function that executes to completion or until an error occurs. Of course the function may call the OS primitives to spawn other tasks (e.g. `taskSpawn`).

As an example, the function `stroke` defined in the beginning of this section could be invoked as

```
-> stroke(45.0, 0.0, 5.0, 0)
```

where `->` is the shell prompt.

Built in commands are available for directory inspection and, above all for debugging purposes. It is in fact possible to inspect the execution stack of the running tasks and to actually block them via breakpoints. As such the `VxWorks`tm shell also works as a limited debugging environment.

- **Available Functions:** In this section we provide a quick reference to all the functions currently available to a programmer. Table 3.1.3 contains a list of such functions. Each function will be described in more detail in the following. Argument types will be given in an ANSI C style.

  ▶ <u>on routine:</u>

```
void on(char c)
```

[4] There are ways to bypass this step and launch directly an application using the EPROM chips on the CPU board running `VxWorks`tm.

Functions available to the programmer	
`on`	Turns on the power for the ED-I motors
`off`	Turns off the power for the motors
`pos`	Returns the position of a specified joint
`axis_status`	Prints out a summary of various values for a specified axis
`coord`	Reset the Coordinate frame to the current position as origin
`arm_forward_kinematics`	Computes the forward kinematics for the two main joints
`arm_inverse_kinematics`	Computes the inverse kinematics for the two main joints
`joint_move`	Move the two main joints
`stroke`	Moves the joints from the current position to a specified one interpolating a straight line
`home`	Performs the predefined *homing sequence*

Table 10: **Functions available for programming**

This function turns on the power for the motor of the axis specified by the mnemonic character passed to it.

- `off` routine:

```
void off(char c)
```

This function turns off the power for the motor of the axis specified by the mnemonic character passed to it (as in case of `on`).

- `pos` routine:

```
void pos(char c)
```

This function works by side effect. It prints on the console the actual position (in encoder edges with respect to the origin of the current coordinate frame) of the axis specified by the mnemonic character. Moreover, it makes sure that such value is recorded in the field `Target_Pos_In_Joint` of the relevant `AXIS_STRUCT`.

- `axis_status` routine:

```
void axis\_status(char c)
```

This function prints on the console in a human readable format the contents of the `AXIS_STRUCT` relative to the mnemonic character passed to it.

- `coord` routine:

```
void coord()
```

This function resets the origin of the coordinate frame to the current position in Joint Space of the manipulator joints.

- `arm_forward_kinematics` routine:

```
int arm_forward_kinematics(double theta1,
                           double theta2,
                           double* x,
                           double* y)
```

Given the point in Joint Space ⟨theta1, theta2⟩, the function returns in `x` and `y` the coordinates of corresponding point in Cartesian Space.

▶ `arm_inverse_kinematics` routine:

```
int arm_inverse_kinematics(double* theta1,
                           double* theta2,
                           double x,
                           double y)
```

Given the point in Cartesian Space ⟨x, y⟩, the function returns in `theta1` and `theta2` the coordinates of (one of) the corresponding point(s) in Cartesian Space.

`ERROR` is returned if the computation did not terminated correctly, i.e. the point was not in the workspace of the manipulator. Otherwise `OK` is returned.

▶ `joint_move` routine:

```
int joint_move(double theta1,
               double theta2,
               double time,
               int mode)
```

This functions moves the two main joints (θ_1 and θ_2) from their current positions to the one specified by the parameters `theta1` and `theta2` in time `time`. If the last integer parameter (`mode`) is equal to `ABSOLUTE_MOVE` (0), then the two joints will move to the position specified with respect to the current frame. If the `mode` parameter is `RELATIVE_MOVE` (1) then the values of `theta1` and `theta2` are used as increments to the current position.

The value returned is a code that specifies whether the operation completed correctly or not. A value equal to `OK` indicates correct termination. Otherwise `ERROR` is returned.

A return value of `ERROR` usually indicates that the kinematics computations could not be completed, i.e. the selected point is outside the manipulator workspace.

▶ `stroke` routine:

```
int stroke(double pt1,
           double pt2,
           double time,
           int mode)
```

This function generates a sequence of *via points* (see [7]) in order to move the end effector of the manipulator on a straight line in the XY plane. The movement is done in a duration specified by `time`. If the `mode` parameter is equal to `CARTESIAN_SPACE` (1) then the values of `pt1` and `pt2` are intended as a point in the physical space XY plane, otherwise (`mode` is equal to `JOINT_SPACE`) they are intended as a final position in joint space. Kinematics computations are executed accordingly.

The value returned is `OK` if all the computations terminated correctly, `ERROR` otherwise. Once again, if `ERROR` is returned, most likely the end point ⟨pt1, pt2⟩ or one of the generated via points is outside the manipulator workspace and the kinematics routines failed to complete. See the error message printed on the console for more information.

▶ `home` routine:

```
void home(char c)
```

This function performs the *homing sequence* specified at a very low level by the motion control hardware[5]. The argument `c` is used to specify for which axis the homing sequence is to be performed. No value is returned.

3.2 Low Level Software Architecture: MOSAIC/ED I

The ED I system is based on the architecture of the MOSAIC open architecture manufacturing control system. As explained earlier, its main components are a UNIX workstation and a `VxWorks`tm Real-Time subsystem, which are directly in charge of the special Creonics hardware cards that control the robot axes. The UNIX workstation is used mainly for cross-development, but it could also be used as a *client* for a `VxWorks`tm *server*.

The `VxWorks`tm real time system loads and runs (see [37]) a set of concurrent routines, which are synchronized by means of traditional (i) *semaphores* and (ii) *message passing*. The real-time nature of the system is obtained by means of a prioritization scheme (i.e. privileges) and by supporting a wide variety of styles of command executions. We will defer the discussion of how the MOSAIC/ED I system takes advantage of these structures, while keeping the details hidden from the user. We begin by looking at how the complete MOSAIC/ED I system is put together: namely, its main components, their software representation and how these components interact within the `VxWorks`tm kernel.

3.2.1 Main Hardware Components

The ED I system is composed of the following abstractions of hardware components:

$$\textbf{A Machine} \implies \begin{cases} \textbf{A Set of Axes} \\ \textbf{A Set of Sensors} \end{cases}$$

Axes and sensors must be controlled by special *hardware cards*[6], which run special *software drivers*. Therefore we can see a machine from another viewpoint as

$$\textbf{A Machine} \implies \textbf{A Set of Cards} \implies \textbf{A Set of Software Drivers}$$

Using the above model, we can now explain the architecture of the ED I system as it is implemented on the `VxWorks`tm platform. Our exposition also closely follows the actual programming structure and can be viewed as a *code-walk*.

3.2.2 The Configuration File

We begin with the description of the initialization process, i.e., how the software starts up and builds for itself a *representation* of the diverse components. The initialization routine (and the entry point for the whole "program") begins by reading a *configuration file* that contains a description of various

[5] Basically the axis is moved until it hits the hardware limit switches and it is slowly moved off them before stopping. This functionality and its various settings are provided by the Creonics Motion Control Card.

[6] The current set up has two Creonics cards capable of controlling two axes; the first controls the x and y axes, while the second will control the z and the ρ (rotational) axes.

components of the system and creates their representation. At the end of this process, the software is ready to operate in its normal mode and in a safe state.

The configuration file, `ed1.cfg`, for the ED I robot system is divided into the following four sections:

- *Card Declaration-definition Section.*
- *Machine/axes Declaration-definition Section,*
- *Sensors Declaration Section.*
- *Commands Declaration Section.*

• **Card Declaration Section:** The *card section* has entries (one per line) denoting all the card that are actually lined up on the VME bus. Each entry has the following format:

card–number	*accesses–number*	*SW driver name*

The actual content of this section is

```
1       #
2       # Card declaration/definition section
3       #
4       0 99 Null_Driver
5       1 1 mcc_driver.o
6       -1 99 Null_Driver
```

The card having *card–number* equal to `0` denotes the ‘main processor card’ (line `4`); in our case a card containing a MC68020 running the `VxWorks`tm kernel. The name `Null_Driver` tells us that no software driver is actually running on the card (since this card actually runs the OS). This particular card has also an *accesses–number* equal to `99`, telling us that the driver associated (none in this case) will accept up to `99` concurrent “accesses” by different “tasks” (in the `VxWorks`tm terminology).

The line `5` denotes the card numbered `1` on the VME rack and it is actually used to run the axes of the machine. The *accesses–number* of `1` tells us that only one task will be performed at once by the card. The “performer” will actually be the routine `mcc_driver`, which is in charge of the very low-level details of communicating directly with the hardware.

The last line(`6`) is just a place holder. Its *card–number* equal to `-1` tells the initialization routine that the card declaration section is finished. As long as the *accesses–number* and the *driver name* “fields” are filled consistently (i.e. the reader routine does not signal an error) their content is nevertheless ignored. If the reading of the configuration file fails then the system does not start.

• **Machine/Axes Declaration Section:** The *machine and axes section* contains a line containing the “name” of the machine followed by the declaration of the axes. The declaration of the axes present on the machine is done on lines having the following format:

Field	**Description**
initial status	A bit-vector integer used to encode various state informations regarding the card
control	
position limit	The *positive* limit reachable by the axis (expressed in degrees)
negative pos. limit	As above but in the "negative direction".
velocity	Velocity
acceleration	Acceleration
deceleration	Deceleration
last position	Initialized to a "home position" of `0.0`
target position	As above
axis coordinate	A single character denoting the actual axis coordinate
identifier	The name actually given to the axis (used mainly for printouts)

The actual entries in our sample configuration file is

```
1       #
2       # Axis declaration/definition section
3       # EDUCATION_ROBOT
4       00 0 180.0 -180.0 180.00 150.0 150.0 0.000 0.000 x theta_1
5       00 0 180.0 -180.0 180.00 150.0 150.0 0.000 0.000 y theta_2
6       00 0 0.00 0.00 0.00 0.00 0.00 0.000 0.000 - END
```

Our particular configuration file contains two axes called `theta_1` and `theta_2` (lines `4` and `5`). The last line (`6`) with a fictitious axis named `END` signals the end of the section (unfortunately all the previous entries must be filled in).

• **Sensor Declaration Section:** This section describes each sensor in the system. Its entries are of the following form:

initial status	*sensor identifier*

The actual section is

```
#
# Sensor Declaration Section
#
00 END
```

I.e. no sensors are currently attached to the ED I robot. The dummy sensor identifier `END` signals the end of the section. Thus at present the user has access to the joint position informations only.

• **Command Declaration Section:** The last section of the configuration file contains a declaration of the commands "acceptable" by the system. The commands are divided into three categories:

- *Prioritized commands* (or *privileged*)

- *Ordered commands*
- *Immediate commands*

The distinction among the different kinds of commands is made depending on the particular roles they play within the real time environment.

Prioritized commands: are commands that must be actually executed at the highest priority. They usually include the initialization routines for the various drivers running on the cards of the VME bus. Moreover, they also declare "generic" library routines that have to be loaded in the `VxWorks`tm kernel in order to run the system[7].

Ordered commands: are those commands whose execution should be terminated (if no error occurs) before other ordered commands are started. "Move" commands are typically in this category. The bottom line is that this kind of commands is *queued* for execution.

Immediate commands: are commands that can be executed immediately by the real time kernel upon "completion" of the current task. I.e. they can bypass the usual queue. Information reading commands can usually be run in this way.

The standard format for an entry in this section is:

command type	*command name*	*arguments*

The *command type* field can be one of `PRI`, `ORD`, `IMD` or `END`. The actual form of this section is listed hereafter.

```
#
# Acceptable Commands declaration section
#
# Truly "prioritized" commands
#
PRI creonics_init
PRI mcc_axis_driver
PRI mcc_axis_init
#
# Actually libraries dynamic loading
#
PRI kinematics
PRI queuer_util
PRI queuer
#
# Ordered commands
#
ORD on  %A;#Aa:
ORD off  %A;#Aa:
ORD pos  %A;#Aa:
ORD coord :
ORD abort %A;#Aa:
ORD tune %A%c:
ORD joint_move %f%f%f:
ORD joint22_move %f%f%f:
```

[7] This is actually a trick in the current state of the implementation. A new category (possibly **library commands**) should be introduced.

```
ORD cartesian_move %f%f%f:
ORD home A;#Aa:
ORD joint_trajectory :
#
# Immediate commands
#
IMD at  taborttune 30
END
```

3.2.3 Starting the System

The whole system must now be started from the `VxWorks`tm *shell*[8] This is done in two steps[9]. First a script is loaded into the shell and afterward a specialized command interpreter is started (the `queuer` prioritized command of the previous section). The initialization file is listed hereafter.

```
ld < vme_util.o
ld < mail_util.o
ld < machine_util.o
ld < command_util.o
ld < mosaic_init.o
#####################################
# Version 3.0 Of MOSAIC use caution #
#      for EDUCATION ROBOT 1        #
#####################################
mosaic_init("ed1.cfg")
creonics_init(1,0xffffff00)
mcc_axis_init("theta1_axis")
mcc_axis_init("theta2_axis")
```

The lines containing with `ld <` ⟨*object file*⟩ tell the `VxWorks`tm linker/loader to place the exported symbols in the systemwide symbol table.

The last four lines in the script file launch different tasks that run concurrently in the `VxWorks`tm environment (which is composed of different cards on the VME bus; each of the cards potentially running concurrently a different task).

Each of the routines spawns a task that sets up various pieces of the whole environment, starts the “interpreter loops” and, most importantly, creates the queues and the semaphores used for synchronization.

- **Starting the Command Interpreter:** The commands can be given to the system by starting a simple command interpreter (as in the configuration file) by hand. The “programs” `queuer_util` and `queuer` must be started by typing directly to the `VxWorks`tm shell:

```
-> queuer
```

At this point all the “immediate” and “ordered” commands declared in the configuration file can be used.

[8] The `VxWorks`tm “shell” act almost as a C language interpreter. Refer to the manuals for its inner working.

[9] All the steps are done by the `VxWorks`tm; no work is required on the part of UNIX.

3.2.4 The Internal Organization

The commands in the initialization script cooperate in getting the system up and running. Here is a brief description of what it is done by them.

• **Behavior of `mosaic_init`:** After reading the configuration file, the procedure `mosaic_init` starts three tasks that actually take care of the internal workings of the system.

- *The* `scheduler` *task,*
- *The* `dispatcher` *task, and*
- *The* `error` *task.*

The `scheduler` task is in charge of taking commands from the *command message queue*, checking that they fall in the "ordered" category, parsing them and "executing" them[10].

The `dispatcher` task is in charge of actually communicating with the cards on the VME bus. Its main loop checks whether there are commands of the different kinds and executes them accordingly, provided that semaphores conditions do not prevent it to do so.

The `error` task is presently only a message dumping routine that waits for messages on a specialized queue.

• **Behavior of `creonics_init`:** The `creonics_init` routine is in charge of initializing the Creonics card with some hardware dependent parameters. The whole process is achieved by sending an 'internal' immediate command to the dispatcher over the appropriate queue. The meaning of the command is to *initialize the driver*.

• **Behavior of `mcc_axis_init`:** The routine `mcc_axis_init` does two things. First it reads an initialization file for the axis it is initializing and then it spawns a task that executes the specialized driver for the axis itself, accordingly with the declaration contained in the configuration file.

The initialization file for the axis contain data that sets some of the default parameters of the driver (see [8] for a complete reference). For example, the file `theta1_axis` contains the following: (Two of these files are loaded; one for each axis.)

```
1       ttheta1_axis 0 1 0x11 102400
2         9 0x11 0x07 0x00 0x2d 0x00 0x00 0x00 0x00 0x05
3         7 0x11 0x07 0x01 0x27 0xfa 0xff 0xff
4                           ...
5         0
```

The line `1` contains the name to be given to the task in charge of driving the axis[11], followed by the axis number, the slot/card number, the `0x11` Creonics internal axis selector (...) and a parameter denoting the *initial position* of the axis.

The name, the axis identifier and the card identifier will be eventually used by `mcc_axis_init` to spawn the correct axis driver (which was declared in the configuration file: that is the routine `mcc_axis_driver`).

Each of the following lines is read and queues up in the immediate commands queue for direct interpretation by the card (via the `dispatcher` routine). The meaning of the first such line is (as interpreted from the Creonics manual [8]).

[10] Each of the commands actually must be aware of some conventions that are used in the system. E.g. the commands operating on the axes should always start their actions with the functions calls `axis_cmd` and `axis_open`, etc.

[11] The double `t` is a `VxWorks`tm convention.

Code	*Description*
9	Number of data entries on this line (each to be read and later interpreted as a single byte)
`0x11`	Card axis identifier
`0x07`	Creonics code for *direct commands*
`0x00`	Code for setting the *servo loops gains*
`0x2d 0x00`	Proportional Gain
`0x00 0x00`	Integral Gain
`0x00 0x05`	Velocity Gain

The second line is similar. It is just used to set the initial values of the *home position* of the axis (codes `0x07` and `0x01`. The format of this entry is

Direct Command Code	Command Type	Four Bytes (units in edges [8])
`0x07`	`0x01`	`0x27 0xfa 0xff 0xff`

The remaining commands in the axis initialization file set the following parameters:

- *Home* and *Overtravel* mode,
- *Maximum Travel* limits (two lines),
- *Maximum Velocity*, *Acceleration* and *Deceleration*,
- *Position Error Tolerance*,
- *Deadband Compensation*,
- *Feedforward Gain*,
- *Encoder Mode*,
- *Axis/Drive type Specification*,
- *In-Position Tolerance*,
- *Velocity Error Tolerance*,
- *Unwind Constant*,
- *Servo Output* limit and *PAC Constants*.

A line containing only a `0` marks the end of the *usable* section of the file.

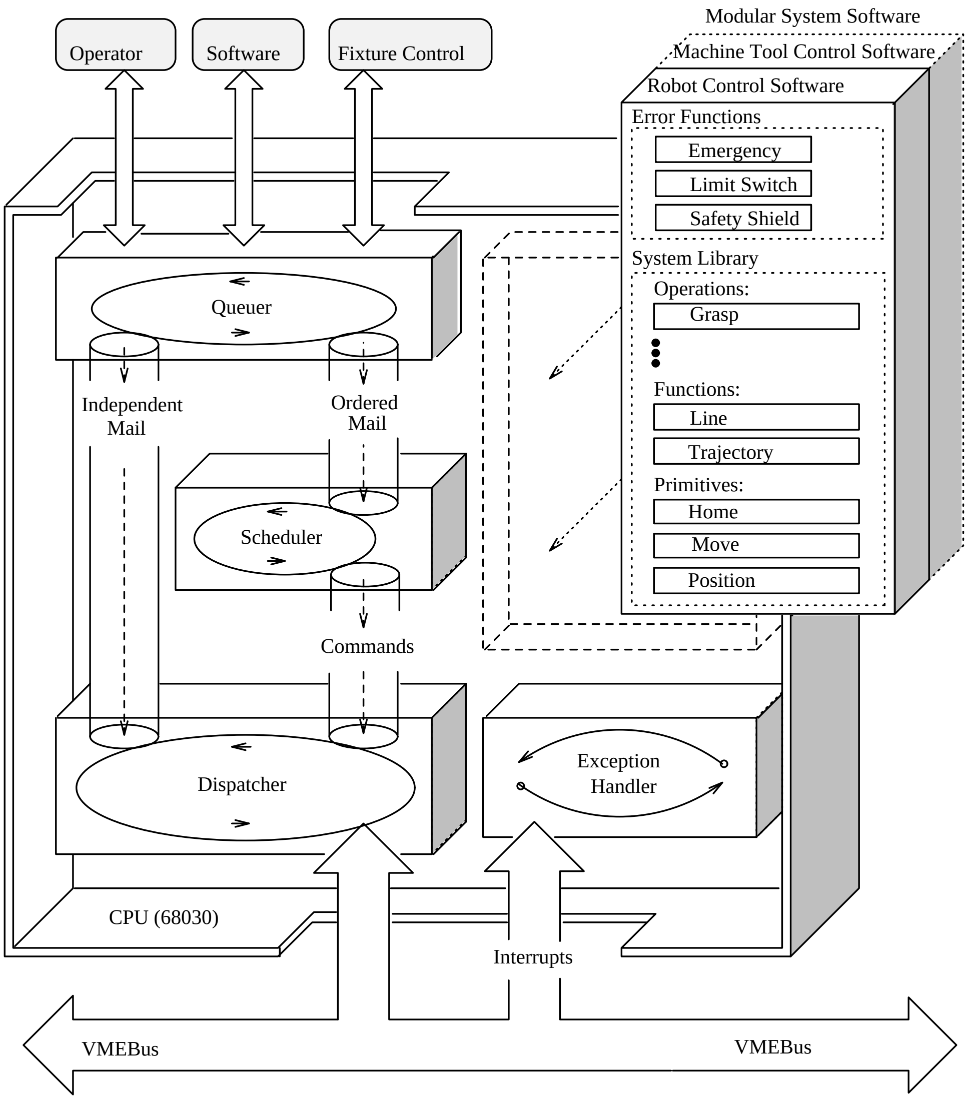


Figure 7: **The software organization of the MOSAIC/ED I system**

- **The Internal Queues and Semaphores:** The system relies its correct functioning on four queues and one "general" semaphore, plus a semaphore for each card and each axes. Both queues and semaphores serve to synchronize access to 'critical' resources. Moreover there is one basic assumption that must be made explicit:

> *The speed of the cards and of the* `VxWorks`tm *real time operating system, coupled with a* careful hand made *serialization of the tasks, is sufficient to avoid race conditions and disastrous events.*
>
> *Moreover, the actions the robot (or a machine tool) can perform are usually not constrained by* deadlines *as in a more "standard" real time scenario.*

This is taken for granted in the implementation of the system.

The interaction of the queues and of the semaphores is best represented graphicallly. See the figure above.

The lines in the schema show the flow of commands as they are queued by the listener (`queuer` and where they may be stopped by the semaphores.

The synchronization among the various components and commands is done by the three message queues (each one for the three kinds of commands) and by a set of semaphores. Currently there are the following semaphores[12].

- A **dispatch semaphore** used by the dispatcher mainly as a wait primitive until the next command is in. Also the commands (and the task executing them) will have to wait if this semaphore is not green.
- A **axis semaphore** for each axes. The axis driver is not allowed to execute more than one command at a time.
- A **card semaphore** for each card on the VME Bus. Since there may be more than one tasks requesting the services of the card, this semaphore is used to regulate the access. The number of concurrent users is declarable in the configuration files.

The primary responsibility of the `VxWorks`tm is thus to dispatch the commands from the generic command queue (coming from the frontend) to the axes driver while ensuring that the semantics associated with *immediate*, *ordered* and *privileged* commands are correctly obeyed. The figure 6 graphically describes how this is achieved by semaphores and queues together with a simple scheduler.

[12] Some are binary MUTEX; some are counting semaphores.