EMC2 Code Notes

5th September 2004

Contents

Chapter 1

Introduction

1.1 Intended audience

This document is a collection of notes about the internals of EMC2. It is primarily of interest to developers, however much of the information here may also be of interest to system integrators and others who are simply curious about how EMC2 works.

1.2 Organisation

There will be a chapter for each of the major components of EMC2, as well as chapter(s) covering how they work together. This document is very much a work in progress, and it's layout may change in the future.

Chapter 2

Overview of EMC2

2.1 Introduction

This will eventually contain the information that is currently at http://home.att.net/~jmkasunich/EMC_Docs/E
however that needs to be modified and updated before it is added.

2.2 Terms and definitions

AXIS An axis is one of the six degrees of freedom that define a tool position in three dimensional Cartesian space. Those axes are X, Y, Z, A, B, and C, where X, Y, and Z are linear coordinates that determine where the tip of the tool is, and A, B, and C are angular coordinates that determine the tool orientation. Unfortunately "axis" is also sometimes used to mean a degree of freedom of the machine itself, such as the saddle, table, or quill of a Bridgeport type milling machine. On a Bridgeport this causes no confusion, since movement of the table directly corresponds to movement along the X axis. However, the shoulder and elbow joints of a robot arm and the linear actuators of a hexapod do not correspond to movement along any Cartesian axis, and in general it is important to make the distinction between the Cartesian axes and the machine degrees of freedom. In this document, the latter will be called "joints", not axes. (The GUIs and some other parts of the code may not always follow this distinction, but the internals of the motion controller do.)

JOINT A joint is one of the movable parts of the machine. Joints are distinct from axes, although the two terms are sometimes (mis)used to mean the same thing. In EMC2, a joint is a physical thing that can be moved, not a coordinate in space. For example, the quill, knee, saddle, and table of a Bridgeport mill are all joints. The shoulder, elbow, and wrist of a robot arm are joints, as are the linear actuators of a hexapod. Every joint has a motor or actuator of some type associated with it. Joints do not necessarily correspond to the X, Y, and Z axes, although for machines with trivial kinematics that may be the case. Even on those machines, joint position and axis position are fundamentally different things. In this document, the terms "joint" and "axis" are used carefully to respect their distinct meanings. Unfortunately that isn't necessarily true everywhere else. In particular, GUIs for machines with trivial kinematics may gloss over or completely hide the distinction between joints and axes. In addition, the ini file uses the term "axis" for data that would more accurately be described as joint data, such as input and output scaling, etc.

POSE A pose is a fully specified position in 3-D Cartesian space. In the EMC2 motion controller, when we refer to a pose we mean an EmcPose structure, containing three linear coordinates and three angular ones.

Chapter 3

Motion Controller

3.1 Introduction

The motion controller receives commands from user space modules via a shared memory buffer, and executes those commands in realtime. The status of the controller is made available to the user space modules through the same shared memory area. The motion controller interacts with the motors and other hardware using the HAL (Hardware Abstraction Layer). This document assumes that the reader has a basic understanding of the HAL, and uses terms like hal pins, hal signals, etc, without explaining them. For basic information about the HAL, read the "Introduction to HAL" document (available as CVS/documents/lyx/Hal_Introduction.lyx, or CVS/emc2/docs/Hal_Introduction.pdf, or http://linuxcnc.org/Hal_Introduction.pdf). Another chapter of this document will eventually go into the internals of the HAL itself, but in this chapter, we only use the HAL API as defined in emc2/src/hal/hal.h.

3.2 Block diagrams and Data Flow

Figure 3.1 is a block diagram of a joint controller. There is one joint controller per joint. The joint controllers work at a lower level than the kinematics, a level where all joints are completely independent. All the data for a joint is in a single joint structure. Some members of that structure are visible in the block diagram, such as coarse_pos, pos_cmd, and motor_pos_fb.

Figure 3.1 shows five of the seven sets of position information that form the main data flow through the motion controller. The seven forms of position data are as follows:

1. emcmotStatus->carte_pos_cmd - This is the desired position, in Cartesian coordinates. It is updated at the traj rate, not the servo rate. In coord mode, it is determined by the traj planner. In teleop mode, it is determined by the traj planner? In free mode, it is either copied from actualPos, or generated by applying forward kins to (2) or (3).
2. emcmotStatus->joints[n].coarse_pos - This is the desired position, in joint coordinates, but before interpolation. It is updated at the traj rate, not the servo rate. In coord mode, it is generated by applying inverse kins to (1) In teleop mode, it is generated by applying inverse kins to (1) In free mode, it is copied from (3), I think.
3. emcmotStatus->joints[n].pos_cmd - This is the desired position, in joint coords, after interpolation. A new set of these coords is generated every servo period. In coord mode, it is generated

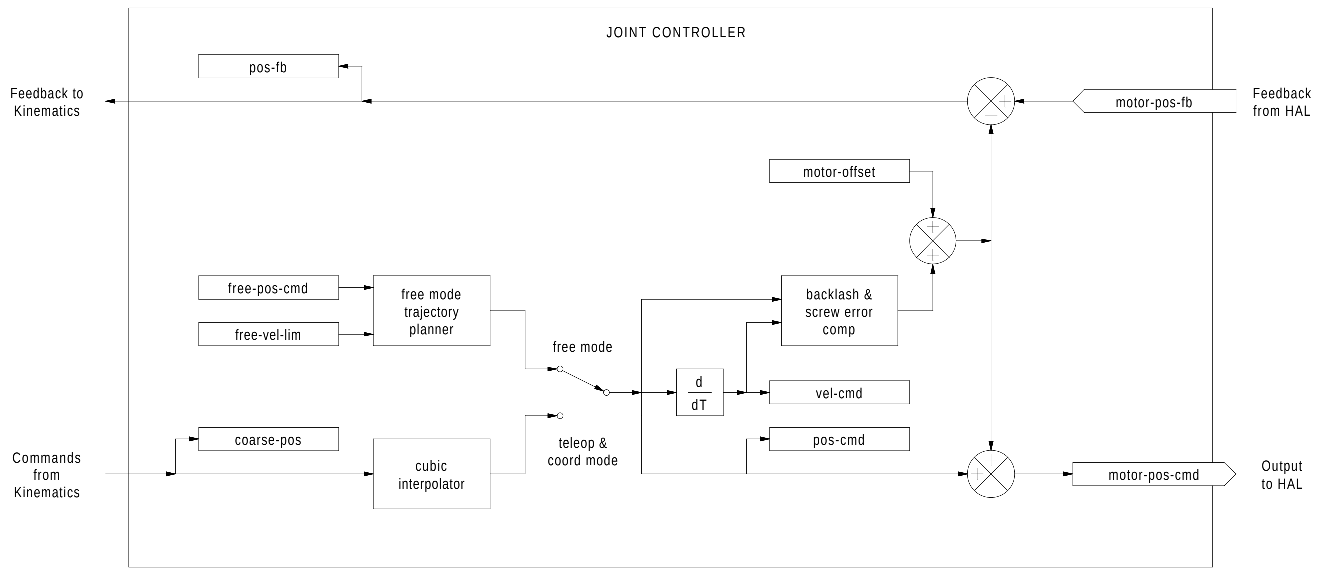


Figure 3.1: Joint Controller Block Diagram

from (2) by the interpolator. In teleop mode, it is generated from (2) by the interpolator. In free mode, it is generated by the free mode traj planner.

4. emcmotStatus->joints[n].motor_pos_cmd - This is the desired position, in motor coords. Motor coords are generated by adding backlash compensation, lead screw error compensation, and offset (for homing) to (3). It is generated the same way regardless of the mode, and is the output to the PID loop or other position loop.

5. emcmotStatus->joints[n].motor_pos_fb - This is the actual position, in motor coords. It is the input from encoders or other feedback device (or from virtual encoders on open loop machines). It is "generated" by reading the feedback device.

6. emcmotStatus->joints[n].pos_fb - This is the actual position, in joint coordinates. It is generated by subtracting offset, lead screw error compensation, and backlash compensation from (5). It is generated the same way regardless of the operating mode.

7. emcmotStatus->carte_pos_fb - This is the actual position, in Cartesian coordinates. It is updated at the traj rate, not the servo rate. Ideally, actualPos would always be calculated by applying forward kinematics to (6). However, forward kinematics may not be available, or they may be unusable because one or more axes aren't homed. In that case, the options are: A) fake it by copying (1), or B) admit that we don't really know the Cartesian coordinates, and simply don't update actualPos. Whatever approach is used, I can see no reason not to do it the same way regardless of the operating mode. I would propose the following: If there are forward kins, use them, unless they don't work because of unhomed axes or other problems, in which case do (B). If no forward kins, do (A), since otherwise actualPos would _never_ get updated.

3.3 Commands

This section simply lists all of the commands that can be sent to the motion module, along with detailed explanations of what they do. The command names are defined in a large typedef enum in emc2/src/emc/motion/motion.h, called cmd_code_t. (Note that in the code, each command name starts with "EMCMOT_", which is omitted here.)

The commands are implemented by a large switch statement in the function emcmotCommandHandler(), which is called at the servo rate. More on that function later.

There are approximately 44 commands - this list is still under construction.

ABORT

The ABORT command simply stops all motion. It can be issued at any time, and will always be accepted. It does not disable the motion controller or change any state information, it simply cancels any motion that is currently in progress.[1]

Requirements

None. The command is alway accepted and acted on immediately.

[1]It seems that the higher level code (TASK and above) also use ABORT to clear faults. Whenever there is a persistent fault (such as being outside the hardware limit switches), the higher level code sends a constant stream of ABORTs to the motion controller trying to make the fault go away. Thousands of 'em.... That means that the motion controller should avoid persistent faults. This needs looked into.

Results

In free mode, the free mode trajectory planners are disabled. That results in each joint stopping as fast as it's accel (decel) limit allows. The stop is not coordinated. In teleop mode, the commanded Cartesian velocity is set to zero. I don't know exactly what kind of stop results (coordinated, uncoordinated, etc), but will figure it out eventually. In coord mode, the coord mode trajectory planner is told to abort the current move. Again, I don't know the exact result of this, but will document it when I figure it out.

FREE

The FREE command puts the motion controller in free mode. Free mode means that each joint is independent of all the other joints. Cartesian coordinates, poses, and kinematics are ignored when in free mode. In essence, each joint has it's own simple trajectory planner, and each joint completely ignores the other joints. Some commands (like JOG) only work in free mode. Other commands, including anything that deals with Cartesian coordinates, do not work at all in free mode.

Requirements

The command handler applies no requirements to the FREE command, it will always be accepted. However, if any joint is in motion (GET_MOTION_INPOS_FLAG() == FALSE), then the command will be ignored. This behaviour is controlled by code that is now located in the function "set_operating_mode()" in control.c, that code needs to be cleaned up. I believe the command should not be silently ignored, instead the command handler should determine whether it can be executed and return an error if it cannot.

Results

If the machine is already in free mode, nothing. Otherwise, the machine is placed in free mode. Each joint's free mode trajectory planner is initialised to the current location of the joint, but the planners are not enabled and the joints are stationary.

TELEOP

The TELEOP command places the machine in teleoperating mode. In teleop mode, movement of the machine is based on Cartesian coordinates using kinematics, rather than on individual joints as in free mode. However the trajectory planner per se is not used, instead movement is controlled by a velocity vector. Movement in teleop mode is much like jogging, except that it is done in Cartesian space instead of joint space. On a machine with trivial kinematics, there is little difference between teleop mode and free mode, and GUIs for those machines might never even issue this command. However for non-trivial machines like robots and hexapods, teleop mode is used for most user commanded jog type movements.

Requirements

The command handler will reject the TELEOP command with an error message if the kinematics cannot be activated because the one or more axes have not been homed. In addition, if any joint is in motion (GET_MOTION_INPOS_FLAG() == FALSE), then the command will be ignored (with no error message). This behaviour is controlled by code that is now located in the function "set_operating_mode()" in control.c. I believe the command should not be silently ignored, instead the command handler should determine whether it can be executed and return an error if it cannot.

Results

If the machine is already in teleop mode, nothing. Otherwise the machine is placed in teleop mode. The kinematics code is activated, interpolators are drained and flushed, and the Cartesian velocity commands are set to zero.

COORD

The COORD command places the machine in coordinated mode. In coord mode, movement of the machine is based on Cartesian coordinates using kinematics, rather than on individual joints as in free mode. In addition, the main trajectory planner is used to generate motion, based on queued LINE, CIRCLE, and/or PROBE commands. Coord mode is the mode that is used when executing a G-code program.

Requirements

The command handler will reject the COORD command with an error message if the kinematics cannot be activated because the one or more axes have not been homed. In addition, if any joint is in motion (GET_MOTION_INPOS_FLAG() == FALSE), then the command will be ignored (with no error message). This behaviour is controlled by code that is now located in the function "set_operating_mode()" in control.c. I believe the command should not be silently ignored, instead the command handler should determine whether it can be executed and return an error if it cannot.

Results

If the machine is already in coord mode, nothing. Otherwise, the machine is placed in coord mode. The kinematics code is activated, interpolators are drained and flushed, and the trajectory planner queues are empty. The trajectory planner is active and awaiting a LINE, CIRCLE, or PROBE command.

ENABLE

The ENABLE command enables the motion controller.

Requirements

None. The command can be issued at any time, and will always be accepted.

Results

If the controller is already enabled, nothing. If not, the controller is enabled. Queues and interpolators are flushed. Any movement or homing operations are terminated. The amp-enable outputs associated with active joints are turned on. If forward kinematics are not available, the machine is switched to free mode.

DISABLE

The DISABLE command disables the motion controller.

Requirements

None. The command can be issued at any time, and will always be accepted.

Results

If the controller is already disabled, nothing. If not, the controller is disabled. Queues and interpolators are flushed. Any movement or homing operations are terminated. The amp-enable outputs associated with active joints are turned off. If forward kinematics are not available, the machine is switched to free mode.

ENABLE_AMPLIFIER

The ENABLE_AMPLIFIER command turns on the amp enable output for a single output amplifier, without changing anything else. Can be used to enable a spindle speed controller.

Requirements

None. The command can be issued at any time, and will always be accepted.

Results

Currently, nothing. (A call to the old extAmpEnable function is currently commented out.) Eventually it will set the amp enable HAL pin true.

DISABLE_AMPLIFIER

The DISABLE_AMPLIFIER command turns off the amp enable output for a single amplifier, without changing anything else. Again, useful for spindle speed controllers.

Requirements

None. The command can be issued at any time, and will always be accepted.

Results

Currently, nothing. (A call to the old extAmpEnable function is currently commented out.) Eventually it will set the amp enable HAL pin false.

ACTIVATE_JOINT

The ACTIVATE_JOINT command turns on all the calculations associated with a single joint, but does not change the joint's amp enable output pin.

Requirements

None. The command can be issued at any time, and will always be accepted.

Results

Calculations for the specified joint are enabled. The amp enable pin is not changed, however, any subsequent ENABLE or DISABLE commands will modify the joint's amp enable pin.

DEACTIVATE_JOINT

The DEACTIVATE_JOINT command turns off all the calculations associated with a single joint, but does not change the joint's amp enable output pin.

Requirements

None. The command can be issued at any time, and will always be accepted.

Results

Calculations for the specified joint are enabled. The amp enable pin is not changed, and subsequent ENABLE or DISABLE commands will not modify the joint's amp enable pin.

ENABLE_WATCHDOG

The ENABLE_WATCHDOG command enables a hardware based watchdog (if present).

Requirements

None. The command can be issued at any time, and will always be accepted.

Results

Currently nothing. The old watchdog was a strange thing that used a specific sound card. A new watchdog interface may be designed in the future.

DISABLE_WATCHDOG

The DISABLE_WATCHDOG command disables a hardware based watchdog (if present).

Requirements

None. The command can be issued at any time, and will always be accepted.

Results

Currently nothing. The old watchdog was a strange thing that used a specific sound card. A new watchdog interface may be designed in the future.

PAUSE

The PAUSE command stops the trajectory planner. It has no effect in free or teleop mode. At this point I don't know if it pauses all motion immediately, or if it completes the current move and then pauses before pulling another move from the queue.

Requirements

None. The command can be issued at any time, and will always be accepted.

Results

The trajectory planner pauses.

RESUME

The RESUME command restarts the trajectory planner if it is paused. It has no effect in free or teleop mode, or if the planner is not paused.

Requirements

None. The command can be issued at any time, and will always be accepted.

Results

The trajectory planner resumes.

STEP

The STEP command restarts the trajectory planner if it is paused, and tells the planner to stop again when it reaches a specific point. It has no effect in free or teleop mode. At this point I don't know exactly how this works. I'll add more documentation here when I dig deeper into the trajectory planner.

Requirements

None. The command can be issued at any time, and will always be accepted.

Results

The trajectory planner resumes, and later pauses when it reaches a specific point.

OPEN_LOG, START_LOG, STOP_LOG, CLOSE_LOG

These commands are used to control the logging feature of the motion controller. Logging will probably be changed in the near future, so I'm not attempting to document these commands in detail right now. Parts of the logging feature may be replaced by halscope and other tools, other parts will be retained.

Requirements

To be documented later.

Results

To be documented later.

SCALE

The SCALE command scales all velocity limits and commands by a specified amount. It is used to implement feedrate override and other similar functions. The scaling works in free, teleop, and coord modes, and affects everything, including homing velocities, etc. However, individual joint velocity limits are unaffected.

Requirements

None. The command can be issued at any time, and will always be accepted.

Results

All velocity commands are scaled by the specified constant.

OVERRIDE_LIMITS

The OVERRIDE_LIMITS command prevents limits from tripping until the end of the next JOG command. It is normally used to allow a machine to be jogged off of a limit switch after tripping. (The command can actually be used to override limits, or to cancel a previous override.)

Requirements

None. The command can be issued at any time, and will always be accepted. (I think it should only work in free mode.)

Results

Limits on all joints are over-ridden until the end of the next JOG command. (This is currently broken... once an OVERRIDE_LIMITS command is received, limits are ignored until another OVERRIDE_LIMITS command re-enables them.)

HOME

The HOME command initiates a homing sequence on a specified joint. The actual homing sequence is determined by a number of configuration parameters, and can range from simply setting the current position to zero, to a multi-stage search for a home switch and index pulse, followed by a move to an arbitrary home location. For more information about the homing sequence, see section 3.4 of this document.

Requirements

The command will be ignored silently unless the machine is in free mode.

Results

Any jog or other joint motion is aborted, and the homing sequence starts.

JOG_CONT

The JOG_CONT command initiates a continuous jog on a single joint. A continuous jog is generated by setting the free mode trajectory planner's target position to a point beyond the end of the joint's range of travel. This ensures that the planner will move constantly until it is stopped by either the joint limits or an ABORT command. Normally, a GUI sends a JOG_CONT command when the user presses a jog button, and ABORT when the button is released.

Requirements

The command handler will reject the JOG_CONT command with an error message if machine is not in free mode, or if any joint is in motion (GET_MOTION_INPOS_FLAG() == FALSE), or if motion is not enabled. It will also silently ignore the command if the joint is already at or beyond it's limit and the commanded jog would make it worse.

Results

The free mode trajectory planner for the joint identified by emcmotCommand->axis is activated, with a target position beyond the end of joint travel, and a velocity limit of emcmotCommand->vel. This starts the joint moving, and the move will continue until stopped by an ABORT command or by hitting a limit. The free mode planner accelerates at the joint accel limit at the beginning of the move, and will decelerate at the joint accel limit when it stops.

JOG_INCR

The JOG_INCR command initiates an incremental jog on a single joint. Incremental jogs are cumulative, in other words, issuing two JOG_INCR commands that each ask for 0.100 inches of movement will result in 0.200 inches of travel, even if the second command is issued before the first one finishes. Normally incremental jogs stop when they have travelled the desired distance, however they also stop when they hit a limit, or on an ABORT command.

Requirements

The command handler will silently reject the JOG_INCR command if machine is not in free mode, or if any joint is in motion (GET_MOTION_INPOS_FLAG() == FALSE), or if motion is not enabled. It will also silently ignore the command if the joint is already at or beyond it's limit and the commanded jog would make it worse.

Results

The free mode trajectory planner for the joint identified by emcmotCommand->axis is activated, the target position is incremented/decremented by emcmotCommand->offset, and the velocity limit is set to emcmotCommand->vel. The free mode trajectory planner will generate a smooth trapezoidal move from the present position to the target position. The planner can correctly handle changes in the target position that happen while the move is in progress, so multiple JOG_INCR commands can be issued in quick succession. The free mode planner accelerates at the joint accel limit at the beginning of the move, and will decelerate at the joint accel limit to stop at the target position.

JOG_ABS

The JOG_ABS command initiates an absolute jog on a single joint. An absolute jog is a simple move to a specific location, in joint coordinates. Normally absolute jogs stop when they reach the desired location, however they also stop when they hit a limit, or on an ABORT command.

Requirements

The command handler will silently reject the JOG_ABS command if machine is not in free mode, or if any joint is in motion (GET_MOTION_INPOS_FLAG() == FALSE), or if motion is not enabled. It will also silently ignore the command if the joint is already at or beyond it's limit and the commanded jog would make it worse.

Results

The free mode trajectory planner for the joint identified by emcmotCommand->axis is activated, the target position is set to emcmotCommand->offset, and the velocity limit is set to emcmotCommand->vel. The free mode trajectory planner will generate a smooth trapezoidal move from the present position to the target position. The planner can correctly handle changes in the target position that happen while the move is in progress. If multiple JOG_ABS commands are issued in quick succession, each new command changes the target position and the machine goes to the final commanded position. The free mode planner accelerates at the joint accel limit at the beginning of the move, and will decelerate at the joint accel limit to stop at the target position.

SET_LINE

The SET_LINE command adds a straight line to the trajectory planner queue.
(More later)

SET_CIRCLE

The SET_CIRCLE command adds a circular move to the trajectory planner queue.
(More later)

SET_TELEOP_VECTOR

The SET_TELEOP_VECTOR command instructs the motion controller to move along a specific vector in Cartesian space.
(More later)

PROBE

The PROBE command instructs the motion controller to move toward a specific point in Carte Sean space, stopping and recording it's position if the probe input is triggered.

(More later)

CLEAR_PROBE_FLAG

The CLEAR_PROBE_FLAG command is used to reset the probe input in preparation for a PROBE command. (Question: why shouldn't the PROBE command automatically reset the input?)

(More later)

SET_xix

There are approximately 15 SET_xxx commands, where xxx is the name of some configuration parameter. It is anticipated that there will be several more SET commands as more parameters are added. I would like to find a cleaner way of setting and reading configuration parameters. The existing methods require many lines of code to be added to multiple files each time a parameter is added. Much of that code is identical or nearly identical for every parameter.

3.4 Homing

Overview

Homing seems simple enough - just move each joint to a known location, and set EMC's internal variables accordingly. However, different machines have different requirements, and homing is actually quite complicated.

In EMC2, homing is done in free mode. The core of the homing algorithm is a state machine contained in the function "do_homing()", which in turn makes use of the free mode trajectory planner. Homing does not use kinematics or the coordinated trajectory planner.

Homing Sequence

Figure 3.2 shows four possible homing sequences, along with the associated configuration parameters. For a more detailed description of what each configuration parameter does, see the following section.

Configuration

There are six pieces of information that determine exactly how the home sequence behaves. They are stored in the joint structure, and each joint is configured independently.

home_search_vel

'home_search_vel' is a member of the joint structure (as defined in motion.h). The default value is zero. A value of zero causes EMC to assume that there is no home switch. The search and latch

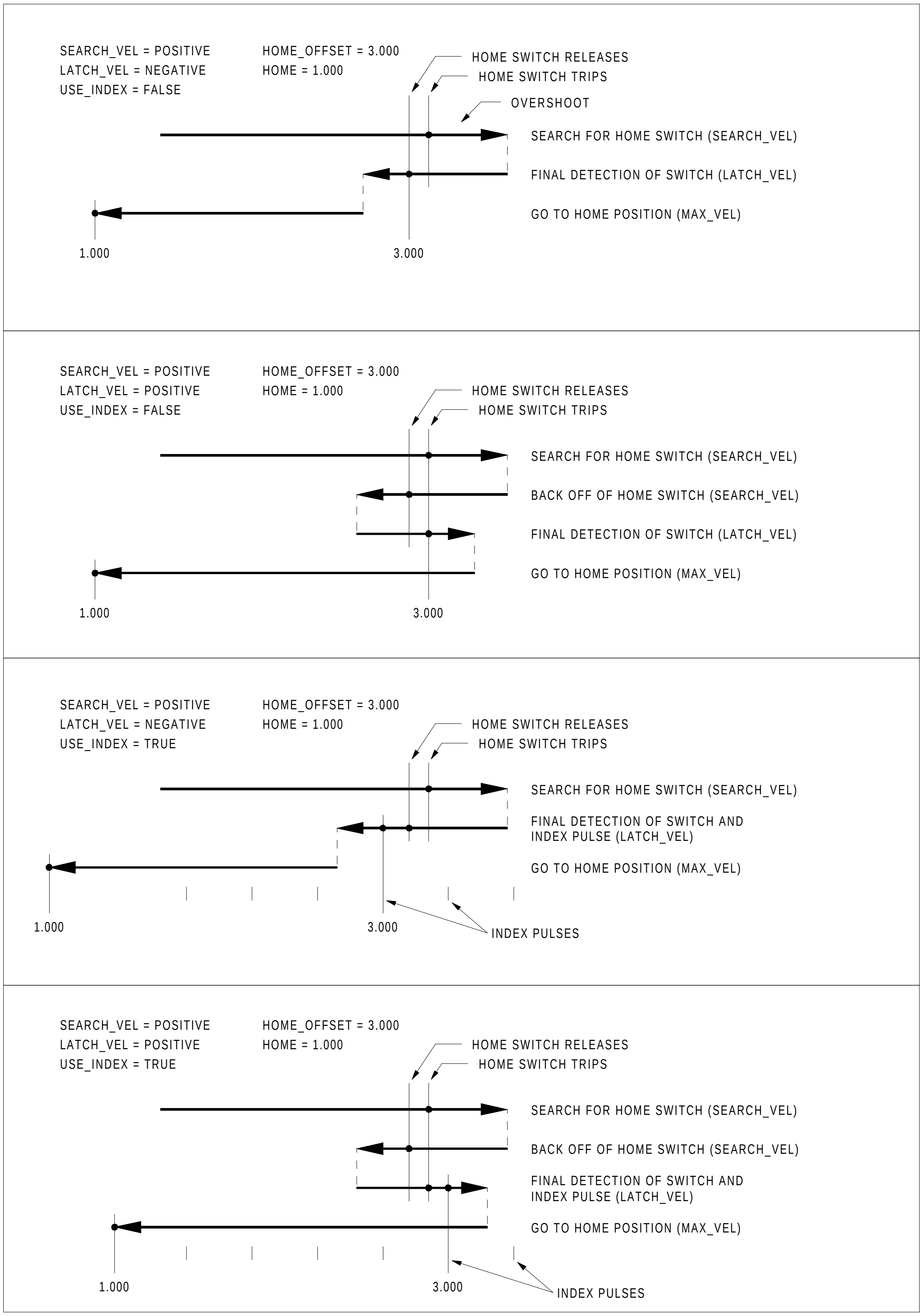


Figure 3.2: Homing Sequences

stages of homing are skipped, EMC declares the current position to be “home_offset”, and does a rapid to “home” if “home” is not equal to “home_offset”.

If ’home_search_vel’ is non-zero, then EMC assumes that there is a home switch. It begins searching for the home switch by moving in the direction specified by the sign of ’home_search_vel’, at a speed determined by its absolute value. When the home switch is detected, the joint will stop as fast as possible, but there will always be some overshoot. The amount of overshoot depends on the speed. If it is too high, the joint might overshoot enough to hit a limit switch or crash into the end of travel. On the other hand, if ’home_search_vel’ is too low, homing can take a long time.

home_latch_vel

’home_latch_vel’ is also a member of the joint structure. It specifies the speed and direction that EMC uses when it makes its final accurate determination of the home switch and index pulse location. It will usually be slower than the search velocity to maximise accuracy. If search_vel and latch_vel have the same sign, then the latch phase is done while moving in the same direction as the search phase. (In that case, EMC first backs off the switch, before moving towards it again at the latch velocity.) If search_vel and latch_vel have opposite signs, the latch phase is done while moving in the opposite direction from the search phase. That means EMC will latch the first pulse after it moves off the switch. If ’search_vel’ is zero, the latch phase is skipped and this parameter is ignored. If ’search_vel’ is non-zero and this parameter is zero, it is an error and the homing operation will fail. The default value is zero.

home_ignore_limits

’home_ignore_limits’ is a single bit within the joint structure member ’home_flags’. This flag determines whether EMC will ignore the limit switch inputs. Some machine configurations do not use a separate home switch, instead they route one of the limit switch signals to the home switch input as well. In this case, EMC needs to ignore that limit during homing. The default value for this parameter is OFF.

home_use_index

’home_use_index’ is a single bit within the joint structure member ’home_flags’. It specifies whether or not there is an index pulse. If the flag is true, EMC will latch on the rising edge of the index pulse. If false, EMC will latch on either the rising or falling edge of the home switch (depending on the signs of search_vel and latch_vel). If ’search_vel’ is zero, the latch phase is skipped and this parameter is ignored. The default value is OFF.

home_offset

’home_offset’ is a member of the joint structure. It contains the location of the home switch or index pulse, in joint coordinates. It can also be treated as the distance between the point where the switch or index pulse is latched and the zero point of the joint. After detecting the index pulse, EMC sets the joint coordinate of that point to “home_offset”. The default value is zero.

home

’home’ is a member of the joint structure. It is the position that the joint will go to upon completion of the homing sequence. After detecting the index pulse, and setting the coordinate of that point to “home_offset”, EMC makes a move to "home" as the final step of the homing process. The default value is zero. Note that even if this parameter is the same as “home_offset”, the axis will slightly overshoot the latched position as it stops. Therefore there will always be a small move at this time (unless search_vel is zero, and the entire search/latch stage was skipped). This final move will be made at the joint’s maximum velocity. Since the axis is now homed, there should be no risk of crashing the machine, and a rapid move is the quickest way to finish the homing sequence. [2]

[2]The distinction between ’home’ and ’home_offset’ is not as clear as I would like. I intend to make a small drawing and example to help clarify it.

3.5 Backlash and Screw Error Compensation

Chapter 4

libnml

4.1 Introduction

libnml is derived from the NIST rcslib without all the multi-platform support. Many of the wrappers around platform specific code has been removed along with much of the code that is not required by emc2. It is hoped that sufficient compatibility remains with rcslib so that applications can be implemented on non-Linux platforms and still be able to communicate with emc2.

This chapter is not intended to be a definitive guide to using libnml (or rcslib), instead, it will eventually provide an overview of each C++ class and their member functions. Initially, most of these notes will be random comments added as the code scrutinised and modified.

4.2 LinkedList

Base class to maintain a linked list. This is one of the core building blocks used in passing NML messages and assorted internal data structures.

4.3 LinkedListNode

Base class for producing a linked list - Purpose, to hold pointers to the previous and next nodes, pointer to the data, and the size of the data.

No memory for data storage is allocated.

4.4 SharedMemory

Provides a block of shared memory along with a semaphore (inherited from the Semaphore class). Creation and destruction of the semaphore is handled by the SharedMemory constructor and destructor.

4.5 ShmBuffer

Class for passing NML messages between local processes using a shared memory buffer. Much of internal workings are inherited from the CMS class.

4.6 Timer

The Timer class provides a periodic timer limited only by the resolution of the system clock. If, for example, a process needs to be run every 5 seconds regardless of the time taken to run the process, the following code snippet demonstrates how :

```
main()
{
    timer = new Timer(5.0);    /* Initialise a timer with a 5 sec-
ond loop */
    while(0) {
        /* Do some process */
        timer.wait();    /* Wait till the next 5 second interval */
    }
    delete timer;
}
```

4.7 Semaphore

The Semaphore class provides a method of mutual exclusions for accessing a shared resource. The function to get a semaphore can either block until access is available, return after a timeout, or return immediately with or without gaining the semaphore. The constructor will create a semaphore or attach to an existing one if the ID is already in use.

The Semaphore::destroy() must be called by the last process only.

4.8 CMS

At the heart of libnml is the CMS class, it contains most of the functions used by libnml and ultimately NML. Many of the internal functions are overloaded to allow for specific hardware dependant methods of data passing. Ultimately, everything revolves around a central block of memory (referred to as the “message buffer” or just “buffer”). This buffer may exist as a shared memory block accessed by other CMS/NML processes, or a local and private buffer for data being transfered by network or serial interfaces.

The buffer is dynamically allocated at run time to allow for greater flexibility of the CMS/NML subsystem. The buffer size must be large enough to accommodate the largest message, a small amount for internal use and allow for the message to be encoded if this option is chosen (encoded data will be covered later). Figure 4.1 is an internal view of the buffer space.

The CMS base class is primarily responsible for creating the communications pathways and interfacing to the O.S.

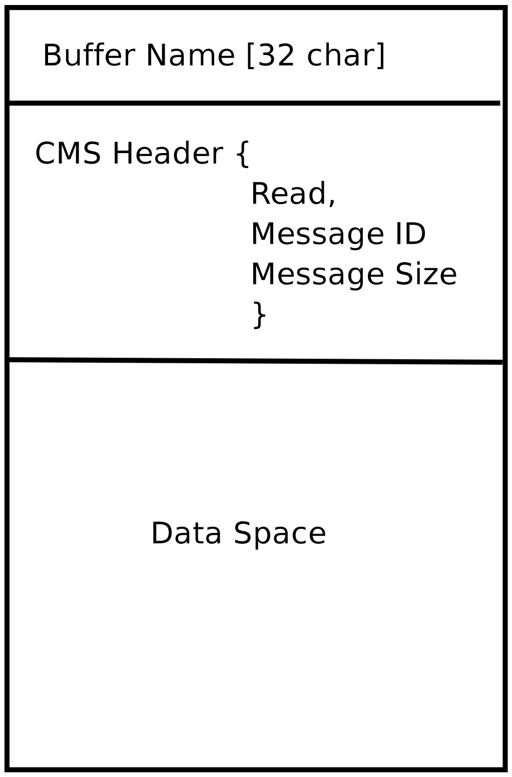


Figure 4.1: CMS buffer

Chapter 5

NML Notes /* FIX ME */

A collection of random notes and thought whilst studying the libnml and rcslib code.
Much of this needs to be edited and re-written in a coherent manner before publication.

5.1 Configuration file format

NML configuration consists of two types of line formats. One for Buffers, and a second for Processes that connect to the buffers.

Buffer line

The original NIST format of the buffer line is:
B name type host size neut RPC# buffer# max_procs key [type specific configs]

- B identifies this line as a Buffer configuration.
- name is the identifier of the buffer.
- type describes the buffer type - SHMEM, LOCMEM, FILEMEM, PHANTOM, or GLOBMEM.
- host is either an IP address or host name for the NML server
- size is the size of the buffer
- neut a boolean to indicate if the data in the buffer is encoded in a machine independent format, or raw.
- RPC# Obsolete - Place holder retained for backward compatibility only.
- buffer# A unique ID number used if a server controls multiple buffers.
- max_procs is the maximum processes allowed to connect to this buffer.
- key is a numerical identifier for a shared memory buffer

Type specific configs

The buffer type implies additional configuration options whilst the host operating system precludes certain combinations. In an attempt to distill published documentation in to a coherent format, only the SHMEM buffer type will be covered.

- mutex=os_sem - default mode for providing semaphore locking of the buffer memory.
- mutex=none - Not used
- mutex=no_interrupts - not applicable on a Linux system
- mutex=no_switching - not applicable on a Linux system
- mutex=mao split - Splits the buffer in to half (or more) and allows one process to access part of the buffer whilst a second process is writing to another part.
- TCP=(port number) - Specifies which network port to use.
- UDP=(port number) - ditto
- STCP=(port number) - ditto
- serialPortDevName=(serial port) - Undocumented.
- passwd=file_name.pwd - Adds a layer of security to the buffer by requiring each process to provide a password.
- bsem - NIST documentation implies a key for a blocking semaphore, and if bsem=-1, blocking reads are prevented.
- queue - Enables queued message passing.
- ascii - Encode messages in a plain text format
- disp - Encode messages in a format suitable for display (???)
- xdr - Encode messages in External Data Representation. (see rpc/xdr.h for details).
- diag - Enables diagnostics stored in the buffer (timings and byte counts ?)

Process line

The original NIST format of the process line is:
P name buffer type host ops server timeout master c_num [type specific configs]

- P identifies this line as a Process configuration.
- name is the identifier of the process.
- buffer is one of the buffers defined elsewhere in the config file.
- type defines whether this process is local or remote relative to the buffer.
- host specifies where on the network this process is running.
- ops gives the process read only, write only, or read/write access to the buffer.

- server specifies if this process will running a server for this buffer.
- timeout sets the timeout characteristics for accesses to the buffer.
- master indicates if this process is responsible for creating and destroying the buffer.
- c_num an integer between zero and (max_procs -1)

Configuration Comments

Some of the configuration combinations are invalid, whilst others imply certain constraints. On a Linux system, GLOBMEM is obsolete, whilst PHANTOM is only really useful in the testing stage of an application, likewise for FILEMEM. LOCMEM is of little use for a multi-process application, and only offers limited performance advantages over SHMEM. This leaves SHMEM as the only buffer type to use with emc2.

The neut option is only of use in a multi-processor system where different (and incompatible) architectures are sharing a block of memory. The likelihood of seeing a system of this type outside of a museum or research establishment is remote and is only relevant to GLOBMEM buffers.

The RPC number is documented as being obsolete and is retained only for compatibility reasons.

With a unique buffer name, having a numerical identity seems to be pointless. Need to review the code to identify the logic. Likewise, the key field at first appears to be redundant, and it could be derived from the buffer name.

The purpose of limiting the number of processes allowed to connect to any one buffer is unclear from existing documentation and from the original source code. Allowing unspecified multiple processes to connect to a buffer is no more difficult to implement.

The mutex types boil down to one of two, the default "os_sem" or "mao split". Most of the NML messages are relatively short and can be copied to or from the buffer with minimal delays, so split reads are not essential.

Data encoding is only relevant when transmitted to a remote process - Using TCP or UDP implies XDR encoding. Whilst ascii encoding may have some use in diagnostics or for passing data to an embedded system that does not implement NML.

UDP protocols have fewer checks on data and allows a percentage of packets to be dropped. TCP is more reliable, but is marginally slower.

If emc2 is to be connected to a network, one would hope that it is local and behind a firewall. About the only reason to allow access to emc2 via the Internet would be for remote diagnostics - This can be achieved far more securely using other means, perhaps by a web interface.

The exact behaviour when timeout is set to zero or a negative value is unclear from the NIST documents. Only INF and positive values are mentioned. However, buried in the source code of rcslib, it is apparent that the following applies:

timeout > 0 Blocking access until the timeout interval is reached or access to the buffer is available.

timeout = 0 Access to the buffer is only possible if no other process is reading or writing at the time.

timeout < 0 or INF Access is blocked until the buffer is available.

5.2 Proposed Configuration Format

Buffer, Host, Size, Key [TCP/UDP port, queued]

Buffer Name remains - Limited to 32 chars.

Host implies a port number and encoding method.

Size must be defined

key is as well to be user defined.

Port number and queueing option remain, whilst encoding method is implied by the port number - Alternative encodings may return.

Several options exist for the format of the configuration file.

- Retain the existing format.
- Use the emc.ini style and re-use some of the code in the process.
- Adopt the NIST Ver. 2.0 format.
- Adopt an XML style.

Provisional configuration file format

Buffer line

Buffer BufferName BufferHost Size Key [queue]

- Buffer Indicates this line specifies a buffer - May be abbreviated to b or B as the config parser only reads the first char.
- BufferName Specifies the name of the buffer - Limited to 32 char
- BufferHost Specifies where the buffer is located - If it is not on this machine, a port number should be given. The format of the BufferHost may be IP address or FQDN followed by a port number separated by a colon or forward slash.
- Size The maximum amount of memory to be allocated to the buffer.
- Key An integer between 0 and 32767
- Process line
- queue is an optional parameter to define whether messages should be held in a FIFO - Default is no queue.

Process Line

Process ProcessName BufferName [ops timeout host]

- Process Indicates this line specifies a process - May be abbreviated to p or P as the config parser only reads the first char.
- ProcessName Specifies the name of the process - Limited to 32 char
- BufferName Specifies the name of the buffer to connect to - It MUST be declared within the same config file.
- ops Either R, W, or RW.

- timeout Specifies how long the process should wait to send or receive a message - INF (or -1), wait indefinitely. Zero, do not wait. Positive value, wait for no more than the specified period in seconds. It is important to note that if a timeout occurs, the message will not be sent or received.
- host Specifies where the process is running if it is not on this machine. The name or address should take the same form as BufferHost on the Buffer line and the port numbers (if given) must be the same although, different buffers may use other ports.

If any buffers or processes are defined as remote by virtue of a port number or the Hostname not resolving to the local machine, TCP/IP protocols and neutral encoding will be used. If no port number is given a default of 5000 will be used.

5.3 NML base class /* FIX ME */

Expand on the lists and the relationship between NML, NMLmsg, and the lower level cms classes.

Not to be confused with NMLmsg, RCS_STAT_MSG, or RCS_CMD_MSG.

NML is responsible for parsing the config file, configuring the cms buffers and is the mechanism for routing messages to the correct buffer(s). To do this, NML creates several lists for:

- cms buffers created or connected to.
- processes and the buffers they connect to
- a long list of format functions for each message type

This last item is probably the nub of much of the malignment of libnml/rcslib and NML in general. Each message that is passed via NML requires a certain amount of information to be attached in addition to the actual data. To do this, several formatting functions are called in sequence to assemble fragments of the overall message. The format functions will include NML_TYPE, MSG_TYPE, in addition to the data declared in derived NMLmsg classes. Changes to the order in which the formatting functions are called and also the variables passed will break compatability with rcslib if messed with - *There are reasons for maintaining rcslib compatability, and good reasons for messing with the code. The question is, which set of reasons are overriding ?*

NML internals

NML constructor

NML::NML() parses the config file and stores it in a linked list to be passed to cms constructors in single lines. It is the function of the NML constructor to call the relevant cms constructor for each buffer and maintain a list of the cms objects and the processes associated with each buffer.

It is from the pointers stored in the lists that NML can interact with cms and why Doxygen fails to show the real relationships involved.

(side note) The config is stored in memory before passing a pointer to a specific line to the cms constructor. The cms constructor then parses the line again to extract a couple of variables... It would make more sense to do ALL the parsing and save the variables in a struct that is passed to the cms constructor - This would eliminate string handling and reduce duplicate code in cms....

NML read/write

Calls to NML::read and NML::write both perform similar tasks in so much as processing the message - The only real variation is in the direction of data flow.

A call to the read function first gets data from the buffer, then calls format_output(), whilst a write function would call format_input() before passing the data to the buffer. It is in format_xxx() that the work of constructing or deconstructing the message takes place. A list of assorted functions are called in turn to place various parts of the NML header (not to be confused with the cms header) in the right order - The last function called is emcFormat() in emc.cc.

NMLmsg and NML relationships

NMLmsg is the base class from which all message classes are derived. Each message class must have a unique ID defined (and passed to the constructor) and also an update(*cms) function. The update() will be called by the NML read/write functions when the NML formatter is called - The pointer to the formatter will have been declared in the NML constructor at some point. By virtue of the linked lists NML creates, it is able to select cms pointer that is passed to the formatter and therefor which buffer is to be used.

Q. Does NML need to maintain a linked list of all the cms objects created ?

A (part 1). No - Each instance of NML will only have one buffer associated with it.

A (part two). Yes - In an application that uses

Chapter 6

Documenting Source Code

Doxygen is useful in generating documentation from the source code. It has the power to extract comments based on predefined tags. In addition, it can be configured to show inheritance of C++ classes along with call graphs of functions. To aid in producing worthwhile documentation from the emc2 source code, some guidelines on formatting tags and where to place them are suggested.

The final output depends ultimately on the config file used with doxygen. In time, one will be committed to the CVS repository.

Header files

These should contain concise descriptions above each function without any formatting tags. Variables contained within class definitions should have a short description on the same line - For example:

```
int timeout;        ///< timeout for blocking semaphore
```

The '///' informs Doxygen that this comment is to appear in the generated docs, and the '<' indicates the comment refers to the variable or function to the left.

Source files

A brief description of each function, parameters, and return values aid other programmers who may work on the code at a later stage. The formatting tags are fairly simple and should appear before the function definition (i.e. Above it). For example:

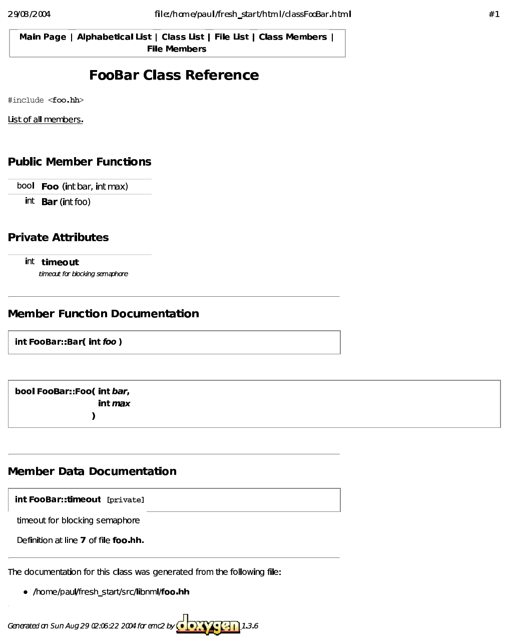
```
/*! This gives a short description of Foo(bar, max)
\param bar is the first value
\param max provides a second value
\returns bool TRUE on success or FALSE on failure
bool Foo(int bar, int max)
*/
{
  if (max > bar) {
    return true;
  }
```

```
    return false;
}
```

An alternative is:

```
///! This gives a short description of Bar(foo)
/*!
\param foo is the only value
\returns the result, or -1 on failure
*/
int Foo(int foo)
{
    if (max == 0) {
        return -1;
    }
    return foo;
}
```

The resulting document will look like:



29/08/2004 file:/home/paul/fresh_start/html/classFooBar.html #1

Main Page | Alphabetical List | Class List | File List | Class Members | File Members

FooBar Class Reference

`#include <foo.hh>`

List of all members.

Public Member Functions

bool	**Foo** (int bar, int max)
int	**Bar** (int foo)

Private Attributes

int	**timeout** *timeout for blocking semaphore*

Member Function Documentation

int FooBar::Bar(int *foo*)

```
bool FooBar::Foo( int bar,
                  int max
                )
```

Member Data Documentation

int FooBar::timeout [private]

timeout for blocking semaphore

Definition at line **7** of file **foo.hh.**

The documentation for this class was generated from the following file:

- /home/paul/fresh_start/src/libnml/**foo.hh**

Generated on Sun Aug 29 02:06:22 2004 for emc2 by doxygen 1.3.6

Figure 6.1: Sample HTML page