opLINUXCNC源程序学习

源程序的树结构如下：

Overview of the emc2 directory (generated by tree -I .git -d) :

.

|-- app-defaults

|-- bin (user mode binaries)

|-- configs

| |-- 5axis

| |-- boss

| |-- common

| |-- dallur-thc

| |-- demo\_mazak (sample mazak config files)

| |-- demo\_sim\_cl (sample sim with ladder IO)

| |-- demo\_step\_cl (sample stepper with ladder IO)

| |-- etch-servo

| |-- halui\_halvcp

| |-- hexapod-sim

| |-- lathe-pluto

| |-- m5i20 (sample servo using Mesa PCI)

| |-- max

| |-- motenc (sample servo using Vital PCI)

| |-- nist-lathe

| |-- plasma-thc

| |-- plasma-thc-sim

| |-- ppmc

| |-- puma

| |-- scara

| |-- sim (simulated motion and IO)

| |-- stepper (parport stepper driver)

| |-- stepper-gantry

| |-- stepper-xyza

| |-- stg (sample servo using STG ISA)

| |-- univpwm (sample PICO servo generator)

| |-- univstep (sample PICO stepper generator)

| `-- vti

|-- debian (files needed to build deb packages)

| |-- extras-Ubuntu-5.10 (extra files for Ubuntu 5.10)

| | |-- etc

| | | |-- modprobe.d

| | | |-- udev

| | | | |-- rules.d

| | | | `-- scripts

| | | `-- xdg

| | | `-- menus

| | | `-- applications-merged

| | `-- usr

| | `-- share

| | |-- applications

| | |-- desktop-directories

| | `-- pixmaps

| |-- extras-Ubuntu-6.06 (extra files for building on Ubuntu 6.06 Dapper Drake)

| | |-- etc

| | | |-- modprobe.d

| | | |-- udev

| | | | |-- rules.d

| | | | `-- scripts

| | | `-- xdg

| | | `-- menus

| | | `-- applications-merged

| | `-- usr

| | `-- share

| | |-- applications

| | |-- desktop-directories

| | `-- pixmaps

| |-- extras-Ubuntu-7.10 (extra files for building on Ubuntu 7.10)

| | |-- etc

| | | |-- modprobe.d

| | | |-- udev

| | | | `-- rules.d

| | | `-- xdg

| | | `-- menus

| | | `-- applications-merged

| | `-- usr

| | `-- share

| | |-- applications

| | |-- desktop-directories

| | `-- pixmaps

| |-- extras-Ubuntu-8.04 (extra files for building on Ubuntu 8.04 Hardy Heron)

| | |-- etc

| | | |-- modprobe.d

| | | `-- xdg

| | | `-- menus

| | | `-- applications-merged

| | `-- usr

| | `-- share

| | |-- applications

| | |-- desktop-directories

| | `-- pixmaps

| |-- extras-sim-Ubuntu-5.10 (extra files for sim package for Ubuntu 5.10)

| | |-- etc

| | | `-- xdg

| | | `-- menus

| | | `-- applications-merged

| | `-- usr

| | `-- share

| | |-- applications

| | |-- desktop-directories

| | `-- pixmaps

| `-- extras-sim-Ubuntu-6.06 (extra files for sim package for Ubuntu 6.06 Dapper Drake)

| |-- etc

| | `-- xdg

| | `-- menus

| | `-- applications-merged

| `-- usr

| `-- share

| |-- applications

| |-- desktop-directories

| `-- pixmaps

|-- docs (All the important and relevant Docs.)

| |-- help

| |-- html (html version of docs - some generated from lyx)

| |-- man (man pages)

| | |-- man1

| | |-- man3

| | `-- man9

| `-- src (API and source notes should be in each src dir.) (source for the handbooks) (top level .lyx files)

| |-- code

| |-- common (shared .lyx files and images, such as glossary, GPLD license)

| |-- config

| |-- gcode (.lyx files, images, etc. for G-Code documentation)

| |-- gui (.lyx files, images, etc. for GUI documentation)

| |-- hal (.lyx files, images, etc. for HAL documentation)

| |-- install

| |-- ladder

| |-- motion

| `-- quickstart

|-- include (headers installed here - originals in src/xxx/)

|-- lib (user mode object files)

| `-- python

| |-- rs274

| `-- yapps

|-- nc\_files (Sample NC files)

|-- rtlib (kernel mode object files, only after a successfull compile)

|-- scripts(bash scripts like linuxcnc, realtime, rip-environment, etc.)

|-- share

| |-- axis

| | |-- images

| | `-- tcl

| |-- emc

| `-- locale

| |-- de

| | `-- LC\_MESSAGES

| |-- es

| | `-- LC\_MESSAGES

| |-- fr

| | `-- LC\_MESSAGES

| |-- hu

| | `-- LC\_MESSAGES

| |-- it

| | `-- LC\_MESSAGES

| |-- pt\_BR

| | `-- LC\_MESSAGES

| |-- ro

| | `-- LC\_MESSAGES

| |-- ru

| | `-- LC\_MESSAGES

| |-- se

| | `-- LC\_MESSAGES

| |-- sr

| | `-- LC\_MESSAGES

| `-- zh\_CN

| `-- LC\_MESSAGES

|-- src (source tree - configure script, top level makefile, Makefile.inc, etc)

| |

| |-- depends (generated dependency tree)

| |

| |-- emc (actual LinuxCNC code)

| | |-- canterp (interpreter for canonical commands)

| | |-- ini (inifile related operations)

| | |-- iotask (IO interface, lots of HAL pins)

| | |-- kinematics (trajectory planner and kinematics轨迹规划器和运动学)

| | |-- motion (motion controller, talks through SHM to the rest of EMC)

| | |-- nml\_intf (emc specific NML implementation, all messages sent in emc)

| | |-- rs274ngc (the rs274 g-code interpreter)

| | |-- sai

| | |-- task (core component in emc, dispatches actions to other parts)

| | `-- usr\_intf (interfaces for some GUIs, and other interfaces: AXIS, halui, stepconf GUI界面)

| | |-- axis

| | | |-- etc

| | | |-- extensions

| | | `-- scripts

| | `-- stepconf

| |-- hal (the Hardware Abstraction Layer, provides a unified interface across all hardware

| 硬件抽象层，提供跨所有硬件的统一接口)

| | |-- classicladder

| | | `-- projects\_examples

| | |-- components

| | |-- drivers

| | | |-- m5i20

| | | | |-- bit

| | | | `-- hostmot5\_src

| | | |-- mesa-hostmot2

| | | | |-- doc

| | | | `-- firmware

| | | | |-- 5i20

| | | | | `-- SOURCE

| | | | `-- 7i43

| | | | |-- CPLD

| | | | | `-- SOURCE

| | | | `-- SOURCE

| | | |-- mesa7i43-firmware

| | | | |-- gpio

| | | | | `-- source

| | | | |-- hostmot2

| | | | | `-- src

| | | | `-- source

| | | |-- mesa\_5i2x

| | | | `-- firmware

| | | |-- pluto\_servo\_firmware

| | | `-- pluto\_step\_firmware

| | |-- user\_comps

| | | |-- devices

| | | `-- vcp

| | `-- utils

| | `-- halgui

| |-- libnml (a clean implementation of RCSLIB)

| | |-- buffer

| | |-- cms

| | |-- inifile

| | |-- linklist

| | |-- nml

| | |-- os\_intf

| | |-- posemath

| | `-- rcs

| |-- module\_helper

| |-- objects

| | |-- emc

| | | |-- canterp

| | | |-- ini

| | | |-- iotask

| | | |-- motion

| | | |-- nml\_intf

| | | |-- rs274ngc

| | | |-- sai

| | | |-- task

| | | `-- usr\_intf

| | | `-- axis

| | | `-- extensions

| | |-- hal

| | | |-- classicladder

| | | |-- components

| | | |-- drivers

| | | |-- user\_comps

| | | | |-- devices

| | | | `-- vcp

| | | `-- utils

| | |-- libnml

| | | |-- buffer

| | | |-- cms

| | | |-- inifile

| | | |-- linklist

| | | |-- nml

| | | |-- os\_intf

| | | |-- posemath

| | | `-- rcs

| | |-- rtapi

| | |-- rtemc

| | | |-- kinematics

| | | `-- motion

| | |-- rthal

| | | |-- classicladder

| | | |-- components

| | | `-- utils

| | |-- rtlibnml

| | | `-- posemath

| | `-- rtobjects

| | `-- hal

| | `-- components

| |-- po (Translation files for i18n support)

| |-- rtapi (\*.c and \*.h for RTAPI) (unified RT API, wraps over RTLinux, RTAI and sim)

| | `-- examples (testing examples for the RTAPI)

| | |-- extint

| | |-- fifo

| | |-- semaphore

| | |-- shmem

| | `-- timer

| `-- tests

|-- tcl (tkemc.tcl, mini.tcl - the tcl GUI's)

| |-- bin

| `-- scripts

`-- tests

|-- abs.0

|-- and-or-not-mux.0

|-- basic

|-- ccomp

| |-- lathe-comp

| |-- mill-g90g91g92

| |-- mill-line-arc-entry

| `-- mill-zchanges

|-- counter-encoder.0

|-- flipflop.0

|-- interp

| |-- cam-nisley

| `-- flowsnake

|-- limit3.0

|-- limit3.1

|-- limit3.2

|-- modparam.0

|-- overrun

|-- oword

| `-- sub.0

|-- save.0

|-- save.1

|-- source.0

|-- stepgen.0

|-- stepgen.1

|-- stepgen.2

|-- threads.0

`-- timedelay.0

Overview of the emc2 install dirs:

/ (file system root)

|

|

|--- etc/

| |

| \--- init.d/

| |

| \--- realtime (realtime start/stop script)

|

|

|--- $(prefix) (default: /usr/local)

| | (all the following $xxxxdir are configurable through autoconf

| |

| |--- bin/ (known as $bindir, files from EMC2/bin)

| |

| |--- sbin/ (known as $sbindir, )

| |

| |--- etc/ (known as $sysconfdir, config files and subdirs from configs/)

| |

| |--- lib/ (known as $libdir, lib files \*.so from libs/)

| |

| \--- share/

| |

| \--- emc/

| |

| |

| |--- docs/

| | |

| | \--- (All the important and relevant Docs.)

| |

| |--- handbooks/

| | |

| | \--- (PDF versions of handbooks)

| |

| \--- nc\_files/

| |

| \--- (Sample NC files)

|

|--- $MAN\_DIR/ (default: /usr/local/man)

| |

| |--- man1/

| | |

| |

| |

| \--- man3/

| |

|

|

\--- $moduledir/ (based on where ./configure finds the RTOS modules)

根据这个结构树，我们可以清楚的知道每一个文件夹，每一个子文件的作用，包括了接口，硬件抽象层，实时内核等等部分。当然，在这里面我们需要深入研究的是linuxcnc-dev/src/emc文件夹中的各个文件，也是该数控系统的核心所在。我们再次列出linuxcnc-dev/src/emc中的子结构树：

linuxcnc-dev/src/emc中的子结构树：

| |-- emc (actual LinuxCNC code)

| | |-- canterp (interpreter for canonical commands)

| | |-- ini (inifile related operations)

| | |-- iotask (IO interface, lots of HAL pins)

| | |-- kinematics (trajectory planner and kinematics)

| | |-- motion (motion controller, talks through SHM to the rest of EMC)

| | |-- nml\_intf (emc specific NML implementation, all messages sent in emc)

| | |-- rs274ngc (the rs274 g-code interpreter)

| | |-- sai

| | |-- task (core component in emc, dispatches actions to other parts)

| | `-- usr\_intf (interfaces for some GUIs, and other interfaces: AXIS, halui, stepconf)

| | |-- axis

| | | |-- etc

| | | |-- extensions

| | | `-- scripts

| | `-- stepconf

根据该结构树，我们一个一个的再一次进行分析：

（1）canterp (interpreter for canonical commands，即规范命令的解释器)

#include <stdio.h> // FILE, fopen(), fclose()

#include <string.h> // strcpy()

#include <ctype.h> // isspace()

#include <limits.h>

#include <algorithm>

#include "config.h"

#include "emc/nml\_intf/interp\_return.hh"

#include "emc/nml\_intf/canon.hh"

#include "emc/rs274ngc/interp\_base.hh"

其中包括了语法解析，读取，执行等函数，对刀具信息、主轴转速、开始、停止等信息进行判断和执行工作。

（2）ini (inifile related operations，初始化相关的文件)

同样，列出该文件夹的结构树：

| |-- ini (inifile related operations)

| | |-- emcIniFile.cc/hh

| | |-- iniaxis.cc/hh

| | |-- inihal.cc/hh

| | |-- initraj.cc/hh

| | |-- initool.cc /hh

(2.1) emcIniFile.cc/hh

#include <math.h> // M\_PI.

#include "emcIniFile.hh"

该文件定义了一些INI文件中字符的赋值，包括 一个TURE或者FAULSE，单位mm ,m，角度，直线，旋转等等。

(2.2) iniaxis.cc/hh

#include <unistd.h>

#include <stdio.h> // NULL

#include <stdlib.h> // atol(), \_itoa()

#include <string.h> // strcmp()

#include <ctype.h> // isdigit()

#include <sys/types.h>

#include <sys/stat.h>

#include "emc.hh"

#include "rcs\_print.hh"

#include "emcIniFile.hh"

#include "iniaxis.hh" // these decls

#include "emcglb.h" // EMC\_DEBUG

#include "emccfg.h" // default values for globals

#include "inihal.hh"

该文件主要用到了两个函数：loadaxis以及iniaxis。*前一个函数loadaixs是对INI文件中的轴类型，轴单元，最大最小位置限值，误差限值，最大速度，加速度，加加速度等初始化参数进行读取并存储在&axisIniFile中，这些参数对应于INI文件中[AXIS]一栏中的参数，并将部分参数赋值给old\_inihal\_data。*除此之外，iniaxis函数主要是读取INI文件中[TRAJ]部分的AXIS轴数，然后调用loadaxis进行初始化加载。

除去两个主要函数，还有一个精度判断函数iniGetFloatPrec以及状态函数（可有可无）。

（2.3）initraj.cc/hh

该文件主要包含以下几个函数：loadTraj，用于加载INI文件中[TRAJ]中的配置参数，然后将读取的参数值进行保存，主要赋值给old\_inihal\_data以及traj\_max\_velocity等变量，供后续其他文件调用。同时该文件的后面还有一个坐标轴返回原点的判断程序，从而保证[TRAJ]中的原点坐标。其实，该文件类似于iniaxis.cc/hh。

（2.4）inihal.cc/hh

#include "rcs\_print.hh"

#include "emc.hh"

#include <stdio.h>

#include "hal.h"

#include "rtapi.h"

#include "inihal.hh"

该文件中主要包含以下几个函数：ini\_hal\_init，用于定义HAL中的新的管脚，包括了各轴中的INI参数管脚，以及TRAJ中的INI参数管脚；ini\_hal\_init\_pins，用于对之前定义好的新管脚进行赋值，主要是将old\_inihal\_data中的参数值存储到\*(the\_inihal\_data->NAME)中去。check\_ini\_hal\_items()，用于对各个管脚的值进行更新检查。

备注：old\_inihal\_data中的值都是在iniaxis.cc以及initraj.cc中赋值了的。

（3） iotask

该文件在这里不做详细介绍，主要是创建“iocontrol”组件，并对一些接口文件进行配置。

（4）Kinamitcs

该文件夹中包含了不同的运动控制模块，例如五轴、三轴、六轴等等。除此之外，还包含了插补以及速度控制的轨迹规划文件。

（4.1）运动控制模块

（4.2）轨迹规划模块

TC.C文件——主要是基于轨迹规划的简单判别文件，由TP.C调用。下面我们分别说明该文件中的一些函数。

#include "rtapi.h" /\* rtapi\_print\_msg \*/

#include "posemath.h"

#include "emcpos.h"

#include "tc.h"

（1）tcGetStartingUnitVector 主要是获取运动轨迹的起始点的起始单位向量，如果是直线或者攻丝，那么直接计算起始与终止点的位移大小。如果不是，那么就是圆形运动，计算曲率，半径，起始点等等。

（2）tcGetEndingUnitVector 主要是获取运动轨迹的终止点的终止单位向量，如果是直线，那么直接计算终点坐标减去起点坐标的大小。如果是攻丝，那么计算起点坐标减去终点坐标的大小。对于其他运动，即曲线运动，则根据转过的角度，计算终点坐标，然后计算曲率半径，并将数据保存到指定位置V中。

（3）tcGetPos/tcGetEndpoint 两个都是通过调用函数tcGetPosReal从而得到位置信息。

（4） tcGetPosReal 根据传入的参数判断是计算当前位移值还是目标位移值；其次判断运动类型，如果是攻丝，判断是正转还是反转的，再进行坐标值计算（具体算法先省略）。如果是直线运动，因为只是涉及到了xyz，其他轴uvw，abc等只需要进行配合运动即可，或者是XYZ不动，或者只有ABC再动等等。如果是曲线运动，同样配合计算其他各轴位移。判断完毕之后，返回各个轴的位移。

（5）tcqCreate() 建立TC结构体的新队列。

（6）tcqDelete() 删除TC结构体的队列。

（7）tcqInit

（8）tcqPut()/tcqRemove()/tcqLen()/tcqItem() 数据存放，先入先出；tcqLen()数据队列的大小；tcqItem()读取数据队列的第N的元素。

（9）tcqFull() 判断堆栈是否已满，或者快要满了。

TP.C文件——主要实现轨迹规划。

#ifdef RTAPI

#define assert(args...) do {} while(0)

#else

// SIM

#include <stdio.h>

#include <stdlib.h>

#include <assert.h>

#include <stdint.h>

#endif

#include "rtapi.h" /\* rtapi\_print\_msg \*/

#include "rtapi\_string.h" /\* NULL \*/

#include "posemath.h"

#include "tc.h"

#include "tp.h"

#include "rtapi\_math.h"

#include "../motion/motion.h"

#include "hal.h"

#include "../motion/mot\_priv.h"

#include "motion\_debug.h"

（1）tpCreate（）创建队列，供后续调用。

（2）tpClearDIOs() 如果需要改变的数字IO的存放队列，该函数将会清除之前的IO触发器队列供下次使用。

（3）tpClear() 某种程度而言，该函数属于“软件初始化”，结构体TP\_STRUCT中的参数将会被保留下来，而队列将会被清空。

（4）tpInit（），初始化TP结构体中的参数值。

（5）tpSetCycleTime（），设置程序循环时间。

（6）tpSetVmax（）设置最大速度。

（7）tpSetVlimit（）/ tpSetAmax() 同上，设置速度限制以及最大速度。

（8）tpSetId（）/tpGetExecId（） 分别设置下一个运行周期的ID号，或获取当前的ID号。

（9）pSetTermCond(tp, cond) 设置后续运动队列移动的终止条件，返回终止条件和公差。

（10）tpSetPos（）设置当前位置以及目标位置。

备注：（4-10）所返回的参数值都是TP结构的成员，可以理解为TP中都是当前运动模式下的参数值。

（11）tpAddRigidTap（）/ tpAddLine()/tpAddCircle() 这三个函数是用于判断在原有的运动上所需要加入的新的运动模式，根据下一步的运动模式不同，设置不同的参数保存在结构体TC中，可以理解为TC结构体中存入了下一个运动模式时的参数值。

（12）tcRunCycle（） 该函数通过传入结构体参数TP以及TC，进行运动段的速度控制，源程序用到的是T形速度控制。根据判断速度和加速度的状态控制速度变化。最后返回新的速度，以及最终速度变化量（以当前的速度与最大限速进行比较，得到需要加速或者减速的多少）。

（13）tpToggleDIOs（） 控制数字或者模拟IO口的开或者断，保证下一次输入或者输出正确。

（14）tpRunCycle（） 该函数是运动控制的核心部分。首先进行emcmotStatus的初始化；判断TC是否空队列；判断当前运动是否同步进行，执行当前运动，完毕之后删除，载入下一个运动；判断下一个运动的可行性，如果出现异常中断（abording），那么TP队列中的值全部初始化。判断所等待的ID号是否与TC中的ID号一致，如果不一致，则输出错误信息，并给所等待的ID赋空值；对于TC值得第一次读取进行一系列状态判断，保证所有动作符合要求；

对于攻丝动作，需要进行特殊处理。即当状态tc->motion\_type == TC\_RIGIDTAP的时候，先保证主轴正转；其次判断攻丝动作所处的状态：TAPPING，该状态表示正在攻丝，如果当前位移量以及超过了或者等于所需要攻丝的长度，那么将状态切换到REVERSING，主轴反向，并设置反向运动的参数，如果当前位移量超过了攻丝长度，那么切换状态到FINAL\_REVERSAL；主轴最后一个反向，然后回位；

判断下一次的运动状态，判断是否TC是第一次被读取；

判断TC结构体中的同步参数，然后判断运动中的速度以及加速度是否满足要求。其中一部分算法属于轨迹规划中的加速度判断法；

在做完上述一些列判断之后，进行轨迹规划的重要部分：

primary\_before = tcGetPos(tc);

tcRunCycle(tp, tc, &primary\_vel, &on\_final\_decel);

primary\_after = tcGetPos(tc);

pmCartCartSub(primary\_after.tran, primary\_before.tran,

&primary\_displacement.tran);

primary\_displacement.a = primary\_after.a - primary\_before.a;

primary\_displacement.b = primary\_after.b - primary\_before.b;

primary\_displacement.c = primary\_after.c - primary\_before.c;

primary\_displacement.u = primary\_after.u - primary\_before.u;

primary\_displacement.v = primary\_after.v - primary\_before.v;

primary\_displacement.w = primary\_after.w - primary\_before.w;

调用之前的函数，计算当前值，然后运行一个周期，再次获取新的位置，然后分别计算各轴位移量。

下一步计算属于混合模式下（即将几段相似的程序进行合并）的运动控制：同样进行条件判断，之后计算前后两段运动的速度，位移等等，最后输出前后两端的位移大小，实现合并。

（15）最后对一些错误信息进行判断和验证。保证正确输出。

备注：在tp.c中轨迹规划是如何完成的：

（1）解释调用tpAddLine还是tpAddCircle?主要是依据MDI或者G代码的输入；

（2）tpRunCycle在每一个TRAJ\_PERIOD周期内进行调用，从而更新currentPos；

（3）currentPos最终反馈回HAL对应的是axis.X.motor-pos-cmd HAL pins。

在tpruncycle中：

* 判断tc队列是否为空，if(!tc)；运动队列为空就表示已经到了程序的终点或者说队列处于饥饿状态，需要存入数据：

此时，采取以下动作：（1）初始化一个空的运动队列；（2）通过tp->goalPos = tp->currentPos and tp->done = 1来终止机器的动作；（3）调用tpResume(tp)然后等待队列tc中写入东西。

* 判断是否有异常终止信息if(tp->aborting)；

此时，采取以下动作：(1) 将当前的速度设置为0；（2）通过tp->goalPos = tp->currentPos and tp->done = 1来终止机器的动作；

* 判断是否当前命令行执行完毕if (tc->target == tc->progress)：

此时，采取以下动作：(1) 将当前的命令行从队列中删除；（2）将下一个命令载入到队列中。

（4.3）cubic.c三次样条插值

（5）Motion

Motion部分主要是

（5.1）command.c

#include <linux/types.h>

#include <float.h>

#include "posemath.h"

#include "rtapi.h"

#include "hal.h"

#include "motion.h"

#include "motion\_debug.h"

#include "motion\_struct.h"

#include "emcmotglb.h"

#include "mot\_priv.h"

#include "rtapi\_math.h"

#include "motion\_types.h"

该函数主要是通过用户空间获取命令，执行不同的动作，这些命令结构体为emcmotCommand->command，而全部动作值都保存为EMCMOT\_COMMAND在 motion.h中。

如果想要进行添加新的信号参数，你就需要做以下的工作：

（1）在motion.h中添加新的配置或者轴参数'foo'。

（2）在motion.h中添加'EMCMOT\_SET\_FOO"命令，在cmd\_code\_t中。

（3）在command\_t struct中添加一个类a field

（4）在command.c中加一个大的选择开关从而控制'EMCMOT\_SET\_FOO' command。

（5）添加一个函数emcSetFoo()在taskintf.cc中。

（6）在emc.hh中添加规范emcSetFoo()。

（7）iniaxis.cc或者inixxx.cc中加入代码从而获取INI file中的值，以及调用emcSetFoo().注意，参量一定不要重复。

（8）在iniaxis.cc中写入新值。

这说明你可以在INI文件中读取数据保存到共享内存，然后在运动控制器中调用。如果你想通过NML实现数据操作，还应该做以下：

（9）定义#define EMC\_SET\_FOO\_TYPE 在 emc.hh中。

（10）定义一个类EMC\_SET\_FOO 枣emc.hh中。

（11）在emctaskmain.cc中添加一个大的状态控制开关，从而在接收到NML信息之后调用emcSetFoo()。

（12）在emc.cc,中添加两个与查找和设计新的命令相关的大的状态开关控制。

好的，说完这些我们进行逐一介绍该函数的主程序。

（1）checkAllHomed(void) 根据名字就很好猜到，这是检查各轴是否处于原点。

（2）limits\_ok()/jog\_ok() 检查限位以及各轴是否准备好。

（3）refresh\_jog\_limits（） 重新检查限位；如果是从原点开始的，则直接重新给限位赋值，如果不是，则在原有位置信息的基础上加上限位大小；

（4）inRange() 检查轴是否在限定范围内运行。如果是，返回非零值，如果超过，返回零。

（5）clearHomes() 清零回位标志位；

（6）emcmotDioWrite()/emcmotAioWrite 用于管脚写入值，对于数字IO，写入0或者1；对于模拟IO，输出实际值。

（7）is\_feed\_type（） 用于判断运动类型。对于不同的运动类型返回不同的值；

（8）emcmotCommandHandler() 在主循环中进行调用，用于读取共享内存中的数据。在进行命令判断之前，先判断命令的读取以及数目是否正确。并且一定要将emcmotCommand中的axis与joint\_num对应起来。

之后进行各个命令的判断：

EMCMOT\_ABORT：不管机器运行在何种模式，所有轴都停止动作，并清除错误标志位。

EMCMOT\_AXIS\_ABORT：只是某一个轴出现错误，停止该轴动作，并清除标志位。

EMCMOT\_FREE:切换当前模式到自由运行模式；这段代码并非是真正实现模式转换的代码，只是将部分标志位清除，并且重置emcmotDebug->coordinating，从而推迟该周期的转换。

EMCMOT\_COORD: 切换当前模式到坐标轴运行模式；同样，这段代码并非是真正实现模式转换的代码，只是将部分标志位清除，并且重置emcmotDebug->coordinating，从而推迟该周期的转换。

EMCMOT\_TELEOP: 切换当前模式到遥控运行模式（teleop motion）；

EMCMOT\_SET\_NUM\_AXES:其实应该叫EMCMOT\_SET\_NUM\_JOINTS。

EMCMOT\_SET\_WORLD\_HOME:将当前位置信息设置为世界坐标系的原点。

EMCMOT\_SET\_HOMING\_PARAMS:将emcmotCommand中的成员赋值给Joint。

EMCMOT\_OVERRIDE\_LIMITS:

EMCMOT\_SET\_MOTOR\_OFFSET: 将emcmotCommand中的刀具偏置成员赋值给Joint。

EMCMOT\_SET\_POSITION\_LIMITS: 将emcmotCommand中的限位信息成员赋值给Joint。

EMCMOT\_SET\_BACKLASH:

EMCMOT\_SET\_MAX\_FERROR:

EMCMOT\_JOG\_CONT:做连续的JOG运动，直到极限。当然只限于自由运动模式，并且被激活的情况下。至于回位，坐标轴运行模式，以及进给保持或者速度倍率为0时都是不可以的，进行完限制之后，设定自由模式下的速度等信息。

EMCMOT\_JOG\_INCR:做增量式JOG运动，同上。

EMCMOT\_JOG\_ABS: 做绝对式JOG运动，同上。

EMCMOT\_SET\_TERM\_COND:调用TP.C中的函数，设置终止条件。

EMCMOT\_SET\_SPINDLESYNC: 调用TP.C中的函数，设置刀具同步信息。

EMCMOT\_SET\_LINE: emcmotDebug->queue做线性队列，要求是在坐标轴模式下运行，使能关闭，没有限制。

EMCMOT\_SET\_CIRCLE:同上。

EMCMOT\_SET\_VEL:设置下一个运动的速度。将emcmotCommand中的速度值赋值给joint。

EMCMOT\_SET\_VEL\_LIMIT: 设置所有后续行动的绝对最大速度。将emcmotCommand中的速度值赋值给joint。

EMCMOT\_SET\_JOINT\_VEL\_LIMIT:设置轴的最大速度。将emcmotCommand中的速度值赋值给joint。

EMCMOT\_SET\_JOINT\_ACC\_LIMIT: 设置轴的最大加速度。将emcmotCommand中的速度值赋值给joint。

EMCMOT\_SET\_ACC: 设置所有后续行动的绝对最大加速度。将emcmotCommand中的加速度值赋值给joint。

EMCMOT\_PAUSE:调用tpPause进行暂停。

EMCMOT\_RESUME: 调用tpResume关闭暂停。

EMCMOT\_STEP:单步运行。

EMCMOT\_FEED\_SCALE: 设置速度倍率。将emcmotCommand中的速度倍率值赋值给emcmotStatus。

EMCMOT\_FS\_ENABLE:打开或者关闭速度倍率设置。

EMCMOT\_FH\_ENABLE: 打开或者关闭速度保持。

EMCMOT\_SPINDLE\_SCALE:设置主轴转速的倍率。

EMCMOT\_SS\_ENABLE: 打开或者关闭速度倍率。

EMCMOT\_AF\_ENABLE:打开或者关闭来自于HAL管脚的自适应速度。

EMCMOT\_DISABLE:重置emcmotDebug->enabling标志位，

EMCMOT\_ENABLE:

EMCMOT\_ACTIVATE\_JOINT:打开轴动作。

EMCMOT\_DEACTIVATE\_JOINT:关闭轴动作。

EMCMOT\_ENABLE\_AMPLIFIER:不用计算，直接使用放大器。

EMCMOT\_DISABLE\_AMPLIFIER:

EMCMOT\_HOME:使得某个特定的轴回位。

EMCMOT\_ENABLE\_WATCHDOG:打开看门狗。

EMCMOT\_UNHOME:不让某个特定的轴回位。

EMCMOT\_DISABLE\_WATCHDOG:

EMCMOT\_CLEAR\_PROBE\_FLAGS:对刀信息标志位清除。

EMCMOT\_PROBE:在坐标系模式下进行对刀。

EMCMOT\_RIGID\_TAP:

EMCMOT\_SET\_TELEOP\_VECTOR:

EMCMOT\_SET\_DEBUG:

EMCMOT\_SET\_AOUT:调用tpSetAout函数。

EMCMOT\_SET\_DOUT: 调用tpSetDout函数。

EMCMOT\_SPINDLE\_ON:将emcCommand中的参数值赋值给emcmotStatus。

EMCMOT\_SPINDLE\_OFF:

EMCMOT\_SPINDLE\_ORIENT:

EMCMOT\_SPINDLE\_INCREASE:以100为基础进行加。

EMCMOT\_SPINDLE\_DECREASE:

EMCMOT\_SPINDLE\_BRAKE\_ENGAGE:打开制动。

EMCMOT\_SPINDLE\_BRAKE\_RELEASE:关闭制动。

EMCMOT\_SET\_JOINT\_COMP:轴补偿。

EMCMOT\_SET\_OFFSET:刀具偏置设置。

（5.2）control.c

#include "posemath.h"

#include "rtapi.h"

#include "hal.h"

#include "emcmotglb.h"

#include "motion.h" 很多参数由此而来

#include "mot\_priv.h"

#include "rtapi\_math.h"

#include "tp.h"

#include "tc.h"

#include "motion\_debug.h"

#include "config.h"

该部分是伺服周期内的主要部分，调用各种状态逻辑以及轨迹计算。

（1）本地函数定义LOCAL FUNCTION PROTOTYPES

process\_inputs()——用于读取HAL中通过硬件层传进来的信号；

do\_forward\_kins(void)——获取轴的坐标位置反馈，通过正解计算出笛卡尔坐标系下的位置反馈，还有部分是用来控制机器不是正向运动或者是没有回到原点的情况。

process\_probe\_inputs(void)——在探测器被触发的时候笛卡尔坐标位置被锁定，并且只有在正向运动过程中探测器输入才会有效。

check\_for\_faults()——用于检测是否超过限位，是否存在跟随误差等等，并且只检测正在动作的轴。

set\_operating\_mode()——用于切换free, coordinated, and teleop.三种模式。

handle\_jogwheels(void)——控制旋钮或者按钮的开关。

do\_homing\_sequence()——根据emcmotStatus->homingSequenceState来决定做什么。如果是多个轴的，则在homing.c和mot\_priv.h中。

do\_homing()——根据每个轴结构的home\_state来决定做什么。Homing作为机器的一种状态进行执行。同上，在homing.c和mot\_priv.h中。

get\_pos\_cmds()——产生位置的设置点，通过调用轨迹规划和插补。

compute\_screw\_comp()——用于计算间隙和导程误差补偿。Leadscrew的误差补偿比间隙更加精密，通过使用emcmotStatus->joint\_vel\_cmd中的速度信息决定轴的运动方式，通过emcmotStatus->joint\_pos\_cmd中的位置信息决定轴在什么位置。这些信息都会被用来计算补偿值，从而将补偿值加入到joint\_pos\_cmd中生成motor\_pos\_cmd，或者从motor\_pos\_fb中减去从而得到joint\_pos\_fb值。最基本的补偿值保存在backlash\_corr中，当然，如果方向反转，则会使得下一步发生变化。backlash\_filt作为一个加强版。

**output\_to\_hal()——控制函数的结果输出到HAL中，将内部变量输出到HAL中以便在HAL的示波器和采集卡中显示。**

update\_status()——输出各种状态信息到共享内存中，从而保证更高级的代码调用。

（2）公共函数PUBLIC FUNCTION CODE

emcmotController()——每个周期的轨迹规划和插补。即使是不动作的轴也是在计算，只是不进行输出而已。该函数开始进行周期的设定，然后依次调用前面介绍的这些函数进行计算。最后通过判断emcmotStatus的状态来确定是否运行完毕。

（3）本地函数代码LOCAL FUNCTION CODE

process\_probe\_inputs（）——emcmotStatus->probeVal

process\_inputs(void)——

（5.3）emcmotcfg.h

编译时参数的默认值；

（5.4）emcmotglb.c/h

#include "emcmotglb.h" /\* these decls \*/

#include "emcmotcfg.h" /\* initial values \*/

编译时的配置参数。主要是将emcmotcfg.h中的默认值赋给emcmotglb.h中的参数。

（5.5）motion.c

#include <stdarg.h>

#include "rtapi.h" /\* RTAPI realtime OS API \*/

#include "rtapi\_app.h" /\* RTAPI realtime module decls \*/

#include "rtapi\_string.h" /\* memset \*/

#include "hal.h" /\* decls for HAL implementation \*/

#include "emcmotglb.h"

#include "motion.h"

#include "motion\_debug.h"

#include "motion\_struct.h"

#include "mot\_priv.h"

#include "rtapi\_math.h"

// Mark strings for translation, but defer translation to userspace

#define \_(s) (s)

主模块初始化以及例程清理。

（1）内核模块参数KERNEL MODULE PARAMETERS

（2）定义全局变量GLOBAL VARIABLE DEFINITIONS

\*emcmot\_hal\_data \*joints joint\_array[EMCMOT\_MAX\_JOINTS] \*emcmotStruct

\*emcmotCommand \*emcmotStatus \*emcmotConfig \*emcmotDebug \*emcmotInternal

\*emcmotError

注意，参数的传递无非就是从一个变量赋值到另一个变量；对于Linux/RT-Linux是在映射内存中对于Unixes是在OS共享内存中。emcmotStruct就是在该内存中。

而各个指针的方向如下：

emcmotCommand points to emcmotStruct->command,

emcmotStatus points to emcmotStruct->status,

emcmotError points to emcmotStruct->error

（3）本地函数类型定义LOCAL FUNCTION PROTOTYPES

init\_hal\_io——从实时控制模块输出HAL管脚以及参数到实际设备上。如果想要在HAL的示波器上看到该参数，就要在此函数进行设置。

export\_joint——被init\_hal\_io调用，同样，如果想要进从HAL的示波器上观察到某些参数，就需要再次进行设置。主要是一些轴的信息。

init\_comm\_buffers()——分配和初始化命令、状态和错误缓冲区。用于和用户空间部分进行信息交换。

emcmotCommand = &emcmotStruct->command;

emcmotStatus = &emcmotStruct->status;

emcmotConfig = &emcmotStruct->config;

emcmotDebug = &emcmotStruct->debug;

emcmotInternal = &emcmotStruct->internal;

emcmotError = &emcmotStruct->error;

这里就是各个指针所指向的内存位置。

并且进行各个指针参数的初始化：例如：

init command struct；

init status struct

init more stuff

joint->type = 0;

joint->max\_pos\_limit = 1.0;

joint->min\_pos\_limit = -1.0;

joint->vel\_limit = 1.0;

joint->acc\_limit = 1.0;

joint->min\_ferror = 0.01;

joint->max\_ferror = 1.0;

joint->home\_search\_vel = 0.0;

joint->home\_latch\_vel = 0.0;

emcmotDebug->tMin = 0.0;

emcmotDebug->tMax = 0.0;

emcmotDebug->tAvg = 0.0;

emcmotDebug->sMin = 0.0;

emcmotDebug->sMax = 0.0;

emcmotDebug->sAvg = 0.0;

emcmotDebug->nMin = 0.0;

tpInit(&emcmotDebug->queue); // tpInit called from tpCreate

tpSetCycleTime(&emcmotDebug->queue, emcmotConfig->trajCycleTime);

tpSetPos(&emcmotDebug->queue, emcmotStatus->carte\_pos\_cmd);

tpSetVmax(&emcmotDebug->queue, emcmotStatus->vel, emcmotStatus->vel);

tpSetAmax(&emcmotDebug->queue, emcmotStatus->acc);

所以基本一些参数都要在这里进行配置。

init\_threads()——被init\_comm\_buffers()调用，创建实时线程，输出函数到实时控制部分，并且添加到线程。

（4）公共函数代码PUBLIC FUNCTION CODE

emcmot\_config\_change(void)——

reportError(const char \*fmt, ..——

emc\_message\_handler（）——

rtapi\_app\_main(void)——调用init\_hal\_io();init\_comm\_buffers()；init\_threads();hal\_ready(mot\_comp\_id);进行线程配置。

rtapi\_app\_exit(void)——释放内存，断开HAL以及RTAPI连接。

（5）本地函数代码LOCAL FUNCTION CODE

（5.6）usrmotintf.cc

#include "config.h" /\* LINELEN definition \*/

#include <stdlib.h> /\* exit() \*/

#include <sys/stat.h>

#include <string.h> /\* memcpy() \*/

#include <float.h> /\* DBL\_MIN \*/

#include "motion.h" /\* emcmot\_status\_t,CMD \*/

#include "motion\_debug.h" /\* emcmot\_debug\_t \*/

#include "motion\_struct.h" /\* emcmot\_struct\_t \*/

#include "emcmotcfg.h" /\* EMCMOT\_ERROR\_NUM,LEN \*/

#include "emcmotglb.h" /\* SHMEM\_KEY \*/

#include "usrmotintf.h" /\* these decls \*/

#include "\_timer.h"

#include "rcs\_print.hh"

#include "inifile.hh"

#define READ\_TIMEOUT\_SEC 0 /\* seconds for timeout \*/

#define READ\_TIMEOUT\_USEC 100000 /\* microseconds for timeout \*/

#include "rtapi.h"

#include "dbuf.h"

#include "stashf.h"

用户进程中与emcmot.c中的实时运动控制器进行数据交换的接口函数定义(init, exit, read, write)。

usrmotIniLoad()——从INI文件中加载参数(SHMEM\_KEY, COMM\_TIMEOUT, COMM\_WAIT)

usrmotWriteEmcmotCommand(emcmot\_command\_t \* c)——从指针C中写入命令。并且将整个命令结构体保存到共享内存中\*emcmotCommand = \*c;

usrmotReadEmcmotStatus(emcmot\_status\_t \* s)——将状态结构体从共享内存拷贝到本地内存，即参数S中。采用如下函数：memcpy(s, emcmotStatus, sizeof(emcmot\_status\_t));

usrmotReadEmcmotConfig(emcmot\_config\_t \* s)——将参数配置结构体从共享内存拷贝到本地内存，即参数S中。memcpy(s, emcmotConfig, sizeof(emcmot\_config\_t));

usrmotReadEmcmotDebug(emcmot\_debug\_t \* s)——将DEBUG结构体从共享内存拷贝到本地内存，即参数S中。memcpy(s, emcmotDebug, sizeof(emcmot\_debug\_t));

usrmotReadEmcmotError(char \*e)——将错误信息结构体从共享内存拷贝到本地内存，即参数S中。memcpy(s, emcmotDebug, sizeof(emcmot\_debug\_t));

printEmcPose(EmcPose \* pose)——输出坐标轴的位姿值；

printTPstruct(TP\_STRUCT \* tp)——输出TP结构体中的所有参数；

usrmotPrintEmcmotDebug(emcmot\_debug\_t \*d, int which)——输出各轴调试中的速度，加速度信息。

usrmotPrintEmcmotConfig(emcmot\_config\_t c, int which)——输出配置信息。

usrmotPrintEmcmotStatus(emcmot\_status\_t \*s, int which)——状态量输出，即S中的参数。

usrmotInit(const char \*modname)——从实时接口（RTAPI）中获取共享内存模块，然后获取内存模块的地址，随后获取所需要的参数信息。

emcmotCommand = &(emcmotStruct->command);

emcmotStatus = &(emcmotStruct->status);

emcmotDebug = &(emcmotStruct->debug);

emcmotConfig = &(emcmotStruct->config);

emcmotError = &(emcmotStruct->error);

usrmotExit(void)——清楚上一函数所使用的内存，然后置零。

emcmotStruct = 0;

emcmotCommand = 0;

emcmotStatus = 0;

emcmotError = 0;

usrmotLoadComp(int joint, const char \*file, int type)——从补偿文件中获取一组补偿值，然后重新计算标准模式，前进后退模式下的补偿计算。

usrmotPrintComp(int joint)——无

（6）nml\_intf

该文件夹中的函数都是对于NML消息机制传递和更新的一种定义。

（6.1）canon.hh

该头文件是所有使用标准命令的的应用程序都应该用到的，包含了正交的三个轴XYZ，除此之外还有旋转坐标ABC。

（6.2）emc.cc/hh

#include "rcs.hh"

// Include command and status message definitions

#include "canon.hh"

#include "emc.hh"

#include "emc\_nml.hh"

#include "emcglb.h"

#include "emcpos.h"

#include "cms.hh"

一个用来处理函数更新和消息路径的自动生成的文件。

（1）emcFormat(NMLTYPE type, void \*buffer, CMS \* cms)

NML/CMS格式函数。由NML CodeGen Java Applet. 自动生成。例如：

case EMC\_TRAJ\_STEP\_TYPE:

((EMC\_TRAJ\_STEP \*) buffer)->update(cms);

（2）\*emc\_symbol\_lookup(long type)

NML符号查找函数。

例如：case EMC\_ABORT\_TYPE:

return "EMC\_ABORT";

（3）EMC\_TASK\_PLAN\_CLOSE::update(CMS \* cms)

对于EMC\_TASK\_PLAN\_CLOSE的NML/CMS更新函数。

（4）EMC\_IO\_STAT::update(CMS \* cms) 同上；

EMC\_SPINDLE\_STAT\_MSG::update(CMS \* cms)；同上；

…………………………………………………………

(5)emc.hh——定义了

EMC error

和EMC\_AXIS types

和 EMC\_TRAJ types

和EMC\_MOTION

和EMC\_TOOL types

和 EMC\_AUX types

和EMC\_SPINDLE types

和EMC\_COOLANT types

和EMC\_LUBE types

和EMC\_IO types

和EMC aggregate types 和其他。

（6.3）emcargs.cc——emccfg.h中值得全局化。

（6.4）emccfg.h——EMC中编译时的的默认值。

（6.5）emcglb.c

#include <string.h> /\* strcpy() \*/

#include "emcglb.h" /\* these decls \*/

#include "emccfg.h" /\* their initial values \*/

#include "emcpos.h" /\* EmcPose \*/

emccfg.h中值得全局化。

（6.6）emc\_nml——EMC NML词汇的申明

（6.7） emcops.cc/hh

对于NML而言，进行初始化以及其他函数功能。这会补充自动生成的emc.cc，其中emc.cc包含了所有消息更新机制。例如：

EMC\_AXIS\_STAT::EMC\_AXIS\_STAT():

EMC\_TRAJ\_STAT::EMC\_TRAJ\_STAT():

EMC\_MOTION\_STAT::EMC\_MOTION\_STAT():

EMC\_TASK\_STAT::EMC\_TASK\_STAT():

EMC\_TOOL\_STAT::EMC\_TOOL\_STAT():

EMC\_AUX\_STAT::EMC\_AUX\_STAT():

EMC\_SPINDLE\_STAT::EMC\_SPINDLE\_STAT():

EMC\_LUBE\_STAT::EMC\_LUBE\_STAT():

EMC\_TOOL\_STAT EMC\_TOOL\_STAT:

EMC\_STAT::EMC\_STAT():

（6.8）interpl.cc

NML消息的队列机制，被翻译器和规范接口用来报告NML的状态，主要是通过程序文件。

#include <string.h> /\* memcpy() \*/

#include "rcs.hh" // LinkedList

#include "interpl.hh" // these decls

#include "emc.hh"

#include "emcglb.h"

#include "linklist.hh"

#include "nmlmsg.hh" /\* class NMLmsg \*/

#include "rcs\_print.hh"

NML\_INTERP\_LIST interp\_list;——NML单元，对于控制器而言；

NML\_INTERP\_LIST::set\_line\_number(int line)——对于后续要加的东西进行行号设置；

（7）task

（7.1）emccanon.cc——3轴NC应用的规范定义。

（7.2）emcsvr.cc——EMC NML的网络服务器

（7.3）emctask.cc——为EMC\_TASK类模式和状态管理，主要是针对G代码和M代码

#include <stdlib.h>

#include <string.h> // strncpy()

#include <sys/stat.h> // struct stat

#include <unistd.h> // stat()

#include <limits.h> // PATH\_MAX

#include <dlfcn.h>

#include "rcs.hh" // INIFILE

#include "emc.hh" // EMC NML

#include "emc\_nml.hh"

#include "emcglb.h" // EMC\_INIFILE

#include "interpl.hh" // NML\_INTERP\_LIST, interp\_list

#include "canon.hh" // CANON\_VECTOR, GET\_PROGRAM\_ORIGIN()

#include "rs274ngc\_interp.hh" // the interpreter

#include "interp\_return.hh" // INTERP\_FILE\_NOT\_OPEN

#include "inifile.hh"

#include "rcs\_print.hh"

#include "task.hh" // emcTaskCommand etc

#include "python\_plugin.hh"

#include "taskclass.hh"

emcTaskInit()——

emcTaskHalt()——

emcTaskAbort()——意外终止，则清除各种状态信息；

emcTaskSetMode(int mode)——设置运行模式。例如：

EMC\_TASK\_MODE\_MANUAL:

EMC\_TASK\_MODE\_MDI:

EMC\_TASK\_MODE\_AUTO:

并且根据不同的模式调用不同的函数：

emcTaskSetState(int state)——设置工作状态。例如:

EMC\_TASK\_STATE\_OFF:

EMC\_TASK\_STATE\_ON:

EMC\_TASK\_STATE\_ESTOP\_RESET:

EMC\_TASK\_STATE\_ESTOP:

并且根据不同状态调用不同函数：

determineMode()——根据子系统的模式返回相应的模式，主要是根据traj模式和mdiOrAuto的标志位来决定。

emcTaskPlanInit()——

emcTaskPlanSetWait()——

emcTaskPlanIsWait()——

emcTaskPlanClearWait()——

emcTaskPlanSetOptionalStop(bool state)——

emcTaskPlanSetBlockDelete(bool state)——

emcTaskPlanSynch()——

emcTaskPlanExit()——

emcTaskPlanOpen(const char \*file)——

emcTaskPlanRead()——

emcTaskPlanExecute(const char \*command)——

（7.4）emctaskmain.cc

周期性的调用emcTaskPlan() 和 emcTaskExecute()；

（1）emcTaskPlan()读取新的命令，根据机器的状态state (estop, on)和模式(manual, auto, mdi)决定下一步要做什么，许多命令只是直接走出子系统。**在自动模式下，调用了翻译器，最终interp\_list附加了NML命令。**

（2）emcTaskExecute()执行一个大的状态开关。如果是，就获取interp\_list的下一个项目，并且设置execState进行预处理。该预处理包括了运动的等待和IO的等待等等，一旦满足这些条件，就发出命令，并设置execState进行预处理，满足该条件之后再读取下一个项目，依次继续。

（3）预处理和后处理都是结合interp\_list中的命令来看的，直接命令就不需要结合他们来看。

（4）此刻，没有任何东西会添加到interp\_list. 但是在定义JOG和HOME命令的预处理和后处理时，可能会改变，如果有所改变，那么一定要确保清理掉相应的interp\_list 终止信息。

（5）在checkPreconditions() 中单步运行作为第一种条件来控制，如果是在单步运行模式，主要是通过变量”stepping”，我们会设置状态waiting-for-step，……

#include <stdio.h> // vsprintf()

#include <string.h> // strcpy()

#include <stdarg.h> // va\_start()

#include <stdlib.h> // exit()

#include <signal.h> // signal(), SIGINT

#include <float.h> // DBL\_MAX

#include <sys/types.h> // pid\_t

#include <unistd.h> // fork()

#include <sys/wait.h> // waitpid(), WNOHANG, WIFEXITED

#include <ctype.h> // isspace()

#include <libintl.h>

#include <locale.h>

#include "rcs.hh" // NML classes, nmlErrorFormat()

#include "emc.hh" // EMC NML

#include "emc\_nml.hh"

#include "canon.hh" // CANON\_TOOL\_TABLE stuff

#include "inifile.hh" // INIFILE

#include "interpl.hh" // NML\_INTERP\_LIST, interp\_list

#include "emcglb.h" // EMC\_INIFILE,NMLFILE, EMC\_TASK\_CYCLE\_TIME

#include "interp\_return.hh" // public interpreter return values

#include "interp\_internal.hh" // interpreter private definitions

#include "rcs\_print.hh"

#include "timer.hh"

#include "nml\_oi.hh"

#include "task.hh" // emcTaskCommand etc

#include "taskclass.hh"

#include "motion.h" // EMCMOT\_ORIENT\_\*

#include "inihal.hh"

checkInterpList(NML\_INTERP\_LIST \*il, EMC\_STAT \*stat)——使用一个指针指向解释器列表，一个指针指向EMC状态，将每一个NML信息退出队列（堆栈），然后检查状态中的极限，资源可用性等等。

先选择list中的状态类型，然后对应于不同的类型选择判断条件。

readahead\_reading(void)——调用emcTaskPlanIsWait()检查是否就绪，可以进行读取。如果就绪，则通过emcTaskPlanLine()和emcTaskPlanCommand进行命令读取，进行执行emcTaskPlanExecute。然后进行其他状态判断。

mdi\_execute\_hook(void)——首先，也是判断各项是都就绪，当一个MDI程序正常执行完毕之后 ，检查在emcTaskPlan()中的完成的MDI命令是否退出队列。

readahead\_waiting(void)——进行逻辑调用，检查子程序运行完成，如果完成，则emcTaskPlanClose()和emcTaskQueueCommand(&taskPlanSynchCmd);

emcTaskPlan()——NC代码或者自由操作模式下进行规划。如果emcCommand和 emcStatus返回的编码不一致，说明是新的命令。然后根据emcStatus->task.state进行选择，调用emcTaskIssueCommand(emcCommand)得到返回值。或调用emcTaskQueueCommand得到返回值。

emcTaskIssueCommand——立即发出命令。根据cmd->type状态进行不同命令的发送。

emcTaskQueueCommand(NMLmsg \* cmd)——将命令放入解释器列表。

emcTaskCheckPreconditions(NMLmsg \* cmd)——被解释器列表中的命令调用，通过对命令类型cmd->type进行判断，返回emcTaskExecute()用于等待的状态。

emcTaskCheckPostconditions(NMLmsg \* cmd)——被解释器列表中的命令调用，通过对命令类型cmd->type进行判断，

emcTaskExecute(void)——

emctask\_startup()——进行初始化，和资源分配

emctask\_shutdown(void)——释放资源

iniLoad(const char \*filename)——

main(int argc, char \*argv[])——

（7.4）iotaskintf.cc——NML的IO接口函数

主要是讲命令赋值给emcCommand.Command,同时赋值给emcCommand.NAME相应的其他参数。

#include <math.h> // fabs()

#include <float.h> // DBL\_MAX

#include <string.h> // memcpy() strncpy()

#include <stdlib.h> // malloc()

#include "rcs.hh" // RCS\_CMD\_CHANNEL, etc.

#include "rcs\_print.hh"

#include "timer.hh" // esleep, etc.

#include "emc.hh" // EMC NML

#include "emc\_nml.hh"

#include "emcglb.h" // EMC\_INIFILE

#include "initool.hh"

sendCommand()——一直等到当前执行的命令执行完毕，再写入给定的命令。首先检查emcIoCommandBuffer和emcIoStatusBuffer是否不为空。然后判断是否需要等待或者开始发送命令。

forceCommand()——写入指定命令，不管之前在执行的是什么样的命令。

emcIoInit()——NML commands接口初始化，在emcio中得到NML缓冲区。

emcIoHalt()——

emcIoAbort(int reason)——

（7.5）taskintf.cc——运动模块的接口函数

该程序主要是给emcmotCommand.NAME中进行赋值。

#include <math.h> // isnan()

#include <float.h> // DBL\_MAX

#include <string.h> // memcpy() strncpy()

#include <unistd.h> // unlink()

#include "usrmotintf.h" // usrmotInit(), usrmotReadEmcmotStatus(),

// etc.

#include "motion.h" // emcmot\_command\_t,STATUS, etc.

#include "motion\_debug.h"

#include "emc.hh"

#include "emcglb.h" // EMC\_INIFILE

#include "emc\_nml.hh"

#include "rcs\_print.hh"

#include "timer.hh"

#include "inifile.hh"

#include "iniaxis.hh"

#include "initraj.hh"

#include "inihal.hh"

用于进行初始化emcCommand.NAME

emcAxisSetAxis：

emcAxisSetUnits：

emcAxisSetBacklash：

emcAxisSetMinPositionLimit：

emcAxisSetMaxPositionLimit：

emcAxisSetMotorOffset：

emcAxisSetFerror：

emcAxisSetMinFerror：

emcAxisSetHomingParams：

emcAxisSetMaxVelocity：

emcAxisSetMaxAcceleration：

用于检查是否都进行了初始化，如果存在没有初始化的，则调用usrmotIniLoad and usrmotInit；如果都初始化了，则调用usrmotExit。

AxisOrTrajInited(void)：

emcAxisInit(int axis)：

emcAxisHalt(int axis)：

emcAxisAbort(int axis)：

emcAxisActivate(int axis)：

emcAxisDeactivate(int axis)：

emcAxisOverrideLimits(int axis)：

emcAxisEnable(int axis)：

emcAxisDisable(int axis)：

emcAxisHome(int axis)：

emcAxisUnhome(int axis)：

emcAxisJog(int axis, double vel)：

emcAxisIncrJog：

emcAxisAbsJog(int axis, double pos, double vel)：

emcAxisLoadComp(int axis, const char \*file, int type)：

emcAxisUpdate(EMC\_AXIS\_STAT stat[], int numAxes)：将joint指针赋值到 stat[axis].NAME。这些全局变量的设定都在emcMotionUpdate()以及emcAxisUpdate(), emcTrajUpdate()，保存及调用usrmotReadEmcmotStatus。

当地状态数据EMC\_TRAJ functions，这些状态数据不是由emcmot提供。主要是赋值给emcmotCommand.NAME

emcTrajSetAxes(int axes, int axismask)

emcTrajSetUnits(double linearUnits, double angularUnits)

emcTrajSetMode(int mode)

emcTrajSetTeleopVector(EmcPose vel)

emcTrajSetVelocity(double vel, double ini\_maxvel)

emcTrajSetAcceleration(double acc)

emcTrajSetMaxVelocity(double vel)

emcTrajSetMaxAcceleration(double acc)

emcTrajSetHome(EmcPose home)

emcTrajSetScale(double scale)

emcTrajSetSpindleScale(double scale)

emcTrajSetFOEnable(unsigned char mode)

emcTrajSetSOEnable(unsigned char mode)

emcTrajSetAFEnable(unsigned char enable)

emcTrajSetMotionId(int id)

emcTrajInit()

emcTrajHalt()

emcTrajEnable()

emcTrajDisable()

emcTrajAbort()

emcTrajPause()

emcTrajStep()

emcTrajResume()

emcTrajDelay(double delay)

emcTrajGetLinearUnits()

emcTrajGetAngularUnits()

emcTrajSetOffset(EmcPose tool\_offset)

emcTrajSetSpindleSync(double fpr, bool wait\_for\_index)

emcTrajSetTermCond(int cond, double tolerance)

emcTrajLinearMove(EmcPose end, int type, double vel, double ini\_maxvel, double acc,int indexrotary)

emcTrajCircularMove(EmcPose end, PM\_CARTESIAN center,PM\_CARTESIAN normal, int turn, int type, double vel, double ini\_maxvel, double acc)

emcTrajClearProbeTrippedFlag()

emcTrajProbe(EmcPose pos, int type, double vel, double ini\_maxvel, double acc, unsigned char probe\_type)

emcTrajRigidTap(EmcPose pos, double vel, double ini\_maxvel, double acc)

emcTrajUpdate(EMC\_TRAJ\_STAT \* stat)：将所有emc的状态量赋值到stat。

emcPositionLoad()

emcPositionSave()

emcMotionInit()

emcMotionHalt()

emcMotionSetDebug(int debug)

emcMotionSetAout()——发送EMCMOT\_SET\_AOUT信息到运动控制器，在运动开始或者当前状态设计一个AOUT命令。

function emcMotionSetDout()——发送EMCMOT\_SET\_DOUT信息到运动控制器，在运动开始或者当前状态设计一个AOUT命令。

emcSpindleOrient(double orientation, int mode)——

emcSpindleOn(double speed, double css\_factor, double offset)——

emcMotionUpdate(EMC\_MOTION\_STAT \* stat)——读取emcmot的状态，保存状态到stat中。