

Mach3 Internals

A Plug-In Writers Bible.

by

Art Fenerty

What do I know?

I get accused allot of knowing CNC well, or of being some sort of CNC Guru. Let me assure you nothing could be further from the truth. I once read an article from an education researcher that it takes 10 years to become a Master of any subject. That is to say that with hard work and study anyone can master any subject they pick in 10 years. Likely, that is a bit light for some fields, but I get the feeling it is accurate for most. This includes CNC. That has not to say everyone in CNC for 10 years is a master. Some people have one year of experience and they repeat it 10 times. This would not be mastering the topic.

My introduction to CNC was 6 years ago. Up to then I could not have told you what the letters CNC stood for. (Constantly Nagging Complexities)? Since then, I have worked hard and studied allot, but most of my knowledge is theoretical, gleaned from thousands of posts and emails on various suggestions for added functionality and reports of problems that people have encountered. This does not make me an expert by a long shot. I am constantly surprised by code I see, and by the ways that people use the various machines hooked up to Mach3. True mastery of CNC takes a very long time and a heck of a lot of perseverance. I haven't got there as yet, but when it comes to exactly how Mach3 does its job, I know it pretty well and Ill try to help you understand it to the depth you may (or may not) want to. This document is not a manual, it is not going to be indexed or have links to various subjects. It's more of a discussion of Mach3 and how it works. A plug-in programmer will want to read and re-read several sections of it. A user may want to read it if he's bored, or wants to understand the reasons behind why he is forced to do some things in strange ways. In my experience, the more one understands the way a program works, the better they can use it. Mach3 is not a typical program. Its event driven for the most part, timer driven in several other parts and overall is a compromise between what is possible and what is not in Windows based CNC. Several have asked why I never ported to Linux, the reason is simple. EMC is an excellent program, is in Linux, and those who use that flavor of operating system already have a solution. I dislike programming in Unix, am aware that over 90% of the world uses Windows, and my internal drive to do Mach3 was based more on providing a solution than selling software. It was a case of "If you build it, they will come.". Turned out that way anyhow, and as of my retiring from Mach3 development, over 12000 users had come. They all came with various levels of understanding, and various backgrounds in CNC that defined their expectations. For my part I tried to do what was possible, and the result is a program that will do most jobs within certain limitations. If you actually make it through this document you'll know the limitations and why they exist. If you are a plug-in programmer you'll be able to better understand the flow of the program and how to interface to its internals.

Its all Connected:

To fully understand what Mach3 will do in a given situation, you really have to have a general understanding of how a system all fits together. The better one understands all aspects of the cnc system, the better the end results of their work. This document is

meant to explain a bit about the interactions within Mach3 of the various subsystems, including GCode, ModBus, General inputs, Brains, and whatever else one tends to connect to the software. Its my hope, that after reading this , the user will have a better overall understanding of the complexities involved, and what to consider when trying to setup a machine for a particular purpose. As I stated, don't expect cross references and details as to where to find information on various subjects. This is a discussion on mach3 in which you have to pretend your in a room with me and I'm rambling on. Close your eyes and you will be able to picture you are here with me and I simply refuse to let you get a word in edgewise. Well, OK if you close your eyes Ill go quiet as you wont be able to read, but here again you don't get to tell me that so Ill continue..

EVOLUTION

Mach3 is a product of evolution. Originally designed only for my own use on a router table, it has evolved over time at the request of thousands of people to be all things to all men in terms of cnc operation. That process is likely far from complete and continues with more and more requests each month as to what needs to be added. At the time of Master5's creation (Master5 being Mach3's distant ancestor) it was expensive to do CNC. The costs seemed to me at the time to be much higher then necessary, certainly more than I was willing to pay, so Master5 was written to run my table. Actually, the program was called EZCNC in those days, but was quickly renamed to Master5. (After going through Master1, 2, 3, and 4 behind the scenes). Master5 brought a modicum of motor control to Windows 95 in the form of a reprogrammed motherboard timer which could be bumped to 8Khz to allow for a steady stream of step pulses without worrying too much about Windows interrupting the stream. One of the many reasons CNC was so expensive back then for a Windows user was that Windows is not capable of pulse generation in a timely or smooth manner. Master5 reprogrammed the system timer and ran in Ring0 at interrupt 8 to get a stable stream. This allowed motor control to be done efficiently up to a maximum speed of 8Khz. (Back then that was considered fast enough for the majority of homebuilders. Things have changed now!).

As we progressed from Win95 to Windows XP, a new generation of timer was required. It took months to develop the theory and to force Windows to do something it truly despises, to give reliable step pulses at high speeds. At time of writing this, the maximum is 100K per second, but really, in general operation, 65000 or so per second is the limit for most CPU's. There exist several reasons for this. To understand the timer is to understand why Mach3 operates as it does in many areas of its use. All processes have built in limitations, and Mach3 has several by virtue of its design. It has to be remembered that what Mach3 does was considered for a long time to be impossible, and it relies on some devious trickery to do its job and an intricate game that Mach plays with the Operating System is what creates some of the limitations. Some will wonder why it is then, when an external device is used (smoothstepper, Galil, G100, ncPod..etc..) those limitations still exist. This seems nonsensical on its face, but when you consider Mach3 was designed for the printer port, and its architecture is derived from the treacherous territory of Windows kernel mode, then it will be understood more easily, why the design architecture, which has evolved over

time, must reflect the original printer port operation in terms of its internal limitations. This means that the simple fact you are using an external engine does not mean you will not necessarily have a limitation enforced by the original specifications of the software. Of course, work is always ongoing to separate main core components to provide a level of isolation that provides for removing external limitations on pulse output.

The Timer

The timer in Mach3 is special. Its only real purpose in life is to deal with things required to be done more quickly than Windows OS's will typically allow. For those that wonder how that is done, it is a several step process. The timer, when you startup mach3, analyses a bit about your motherboard to see how it is using various timing systems. It then reprograms the motherboard to take advantage of the apic local timer unused in almost all CPU chips. If no apic chip is present, or if it is used then Mach3 will institute a different method of getting the CPU, interrupt set to the main timer chip used. It redirects the interrupts on your system to take over interrupt 0, the highest interrupt level you can have in a software environment. It does not allow any other software system to interrupt it, and it can stop windows to do its job without windows knowing its there. It shares a block of memory with Mach3 and operates totally as an independent agent of pulsing. It follows commands sent to it via the shared memory block, and processes data in the block to be sent out, as well as takes incoming data from the ports and puts it into the memory block so that Mach3's main application can see it. Mach3's application actually has no knowledge of the hardware involved. It simply has to set memory variables in the shared block and the engine, merrily running on its own in Ring 0 (privileged windows space) at a very high rate of speed, will then deal with those variables , sending pulses, setting output bits, or performing whatever automatic functions it may be programmed to do.

The Driver does have some automatic functions, things that it must do on its own, as the Windows system cannot respond fast enough to help with the job. Jogging is a good example. There is no way windows can handle the complexities of jogging, so the driver must handle it by command from the application. Homing is another. To keep accurate, Windows is not fast enough to stop at the proper position when reading a switch, so the driver must intervene.

For technical reasons, the engine cannot do any floating point. This is because it is a very time critical subsystem. You can imagine what would happen if in the course of events, while Windows is calculating the balance of your checking account and is about to divide two numbers, Mach3's engine stopped it, used the floating point chip, and then returned to Windows. This interaction could cause the wrong result. It is possible to store the floating-point state of your computer, then do the work in the engine, then restore the various stacks, but the time required would make things difficult on slower computers. For that reason, no floating point is allowed.

So all that having been said, lets look at some example functions of the system to see how they are done. This is the best way I think to see the flow and begin to understand how Mach3 looks at the world as an overall synchronized system.

Jog Motion:

Jogging is done using pure integer math. It is automatic in the engine when the application sets certain variables in the memory block. (In future, let us just refer to the shared memory block between mach3 and its engine as the Block, the driver as the Engine, and the Mach3 as the App. It will hopefully make things clearer.)

Every interrupt, the driver checks to see if it should be jogging. The engine is a point checker. It checks several different ways to see if it should take a step. If it is not running a program then on every interrupt, it will check to see if it is jogging. When I say interrupt, I mean every check at the speed of your kernel mode. If you have 25000 selected, and I'll assume you do through the rest of these discussions, then that is every 1/25000 or 40us of time. Therefore, every 40us the engine looks at six sets of variables, one for each axis. In the variable Engine->Axis[n].Jogging the system sees if indeed the axis should be considered in the jog processor. If it is set to true the driver looks to additional variable to see what speed we are doing, and what acceleration should be used to increase the speed, or finally to see if we are decelerating and if so, if we should stop. All of these variables are in the engine as a set of structures called the Axis structures.

```
struct AxisInfo
```

```
{
    int Index; // Current Count
        int MasterIndex; //Unused at this point
        char StepPin; // Pin for step pulse
        char DirPin; // Pin for Direction Pulse
        char StepPort; // Port # for step
        char DirPort; // Port # for Direction
        bool StepNegate; // low active step?
        bool DirNegate; // low active Direction?
        int CurVelocity; // Current Velocity
        int MaxVelocity; // Current Max Velocity (jogging )
        int MasterVelocity; // Master Velocity...Maximum Velocity in all
circumstances.
        int Acceleration; // Acceleration
        bool AtSpeed; // At Speed Currently
        bool Acc; // Accelerating?
        bool Dec; // Decelerating?
        bool Enable; // Axis Enabled?
        bool Jogging; // Jogging On?
        int JogDir; // Direction of Jog;
        bool Homing; // is this a homing jog?
        bool DeRef; // Dereferencing Move?
        int Memory[6];
        int ActiveMemory;
        int Color;
        int TripCount; // when to stop a probe
        int DepthCount; //where a probe actually hit
```

```

bool Probing;    // if we're probing or not.
bool Slave;      // if this axis is a slave or not.
int  SlaveAxis;  // if so, who the heck are we slaved to?

```

```
};
```

There are actually 7 axis structures, 6 motion axis and one spindle. However, the spindle is special and not processed the same way. Things like Jog variables and such are ignored in the spindle processing.

Say the user has hit the Jog Key. To Jog ,the App gets information you've told it in motor tuning, and calculates a maximum jog speed, puts it in MaxVelocity in the Axis structure for that axis, it sets the acceleration in Acc, sets false in the Dec variable to indicate we are not currently decelerating, and finally sets Jogging to true. The App must also set the Dir variable to indicate what direction we wish to jog. From there it is all auto magic. The driver will see the current velocity is zero; it will take the acceleration and add it to the current velocity. It will then use the current velocity to keep a stream of step pulses coming out at that speed. Remember that this happens 25000 times a second, so the acceleration of the steps coming out will have a very tight granularity of change. This makes it very smooth and accurate in terms of its ramping. The Acc value and the MaxVelocity values are calculated numbers. They are not in steps/second or anything that simplistic. In reality, they are the number of steps per second divided by the kernel frequency and multiplied by 2048. The engine uses these numbers as mentioned to automatically calculate on each interrupt the current speed. It then uses that speed to sense if a pulse should be put out or not during that interrupt. If the current speed is equal to the kernel speed multiplied by 2048 then a pulse will be put out on each interrupt, so maximum system speed is sent to the motor. The MaxVelocity setting keeps the speed limited to the maximum set in the motor tuning, scaled by the Jog% setting in the App and applied to the formulas explained above. To stop a jogging on any axis, the App needs only to set the Dec (decelerate) variable to true. The engine will automatically decelerate the axis interrupt by interrupt until it hits zero and which time the driver will set the Jogging variable in the axis structure to false on its own. The application now knows through simply checking the Jogging variable if an axis is jogging, what direction, and what speed at any time, even though its not actually doing the work. This is why Mach3 is possible. It usually is not doing much of the work, the Engine is the workhorse and it runs very fast.

There are variations on the Jogging process. Probing for example is just a jog. TripCount is simply set to the step count when the engine will set the Dec variable to true on its own, probing is set to true, and the rest of the jogging parameters are set as above. The engine will then automatically stop the jog when the probe input goes active or when the axis Index value hits the setting in TripCount. Index is a very important variable. It is the count of steps taken. It is incremented or decremented each time the engine puts out a step. This way Mach3 always knows where it is at and very accurately. Nothing sets the index except steps going out, or a homing operation that zeros the Index. Otherwise, it remains untouched by any code to ensure accuracy. If the DRO does not agree with motor position, the motor lost steps. There can be no question about that, and it was planned that way as a main system diagnostic value.

For Homing, the App simply sets the Homing variable in the axis structure to true, and then starts a jog in the direction set by the user in the homing config. The engine knows to stop when the axis hits a switch, move off, and then zero the Index register all on its own. The App is not involved other than to wait for all this to occur. In fact, most of the time the applications main job is to wait for something to be done or to occur. It is usually waiting on something because it looks at the world in a larger period as opposed to the engine view, which is very short.

When the engine finishes an operation that has changed the position of a machine, and it knows it has no further commands to change the world's position, it signals the App of that by setting a Block variable called Sync. If the engine sets Engine->Sync to true, the App will then synchronize its internal knowledge of the world to the current settings of the Index registers in the axis structures. This means if you initiated a jog and stopped at an X of 10.5 inches, the driver on seeing the jog has stopped will issue a Sync command, which the App will then use to trigger an update of its trajectory planner to the X position of 10.5 so we remain in sync with the engine. Any process can issue a Sync command by setting Engine->Sync = true. The variable will be reset to false when the App resyncs itself, usually within a 1/10 second time.

Planned Motion:

The only other type of motion is a planned motion. This works quite differently from Jogging and is a source of the main limitations within Mach3, but also the reason Mach3 can work in Windows. It is the primary reasoning behind the rather strange architecture of the programs internals.

To initiate planned motion within the context of the Engine requires that the engine be told that motion is available. The Engine, as explained earlier is not capable of floating point operations, and in any event runs much too quickly to plan any upcoming motion. Planning motion is a complex task and uses much of the CPU time available. First, let us consider planned motion for the point of view of the Engine, which is only concerned with output of the step pulses required. The Engine uses a ring buffer of 4096 commands. Each of the commands in the ring comprises index locations, and some other information to help in smoothing the stream of steps. As you will recall, the main system for tracking a motor movement is the Index variable for each axis, it's a count of actual steps put out. In the ring buffer, we divide time by kernel interrupt time multiplied by 5. For example, if the kernel speed is 25K, then the interrupt time is 40us, and multiplying by 5 gives us 200us. Therefore, the ring buffers entries actually represent 4096 time intervals of 200us. Therefore, the ring buffer contains entries signifying where the respective axis should be 200us from the last entry. This ensures that any entry in the ring buffer is no more than five steps from the previous entry and that the next entry will be no more than five steps from the current one. Some of you are scratching your heads here, so let us look deeper. First, a Ring Buffer is a loop, like a series of locations where the last location points back to the first one. SO let's consider a smaller ring of 10 locations, but remember that Mach3 uses 4096 positions in its ring buffer.

Two pointers are used for all this, TrajHead and TrajIndex. For the programmers among you those variables are found in Engine->TrajHead and Engine->TrajIndex. Let's say they are both set to zero. The Engine, seeing they are equal will put out no

steps. It's the App that fills the buffer. Each entry of the ring buffer comprises one copy of the following structure.

```
struct TrajPoint
{
    int Points[6];
    int Command;
    int ID;
    char shifter[6];
};
```

As you can see it's pretty small, but then that are 4096 of them used. The points[6] variable is related to the 6 axis and contains simply a count of the number of total steps in machine coordinates that the axis should be at when the buffer is sitting at that location. The Command variable is special, usually unused and tells the driver to trigger an event like special output types or special actions that can happen while it proceeds through the entries in the ring. As it hits that entry, it will perform that action prior to moving to the index locations in that entry. The ID is simply the GCode line number the entry is a part of. A simple move like G0X10 may contains tens of thousands of entries in the ring, so the ID changes only when the line number of the GCode from which the entry was derived changes. Shifter is a bit special, it tells the engine how to space the maximum of five steps the move represents to keep them as well anti-aliased as possible in time. Smoothness is the result. This "Shifter" variable is what you are turning off or on when you select "Enhanced Pulsing" in the App configurations. So let's consider our 10 position ring buffer, and make a move. When we start, both TrajHead and TrajIndex are zero; Mach 3 sets them that way on startup. Therefore, the driver looks only at Loc 0.

Loc	0	1	2	3	4	5	6	7	8	9	10
Index	0	1	1	2	3	4	4	5	5	5	0

Since TrajIndex and TrajHead are set to zero, the engine will not move. Current Axis Indexes are zero. The App seeing that a move request is in the planner for five steps of movement will begin to fill the locations in the ring buffer with data in the index locations as above. In this

Example the App will fill 10 locations and then increment the TrajHeadPointer to 10. It does this one step at a time, to start a move, it puts in a index location based on acceleration and velocity into the Loc pointed to by TrajHead and then increments the head to 1, it repeats this till it gets to the end of the move or the ring buffer is full. In this, case the move ends just as the buffer gets to the end. You have to remember that the engine is merrily running along. When it sees that the TrajIndex (which is zero) is no longer equal to the TrajHead, it grabs the location pointed to by TrajIndex, (in this case zero) and compares the current Axis index to that. If they differ (they can only differ by a max of 5), it creates a structure, which over the next five interrupts will put out the number of steps necessary to have the MasterIndex for that axis match the ring buffer index. In this case, zero steps will go out over the next five interrupts. (200us). The engine ,on completion of that 200us period will then bump the TrajIndex to the next position (1) and will see that now there is a 1 step differential and over the next 200us will output one step. (The shifter is an 8-bit byte in which only 5 bits are used

and indicate the engine which of the five interrupts should output that step.). When the TrajIndex reaches 10, it will be equal to the TrajHead and movement will stop. If, in the mythical case of our 10 segment ring buffer, the App fills so fast that it reaches 10 while the TrajIndex is still at zero, the App will simply go into a wait state until it sees that the distance from the TrajHead to the TrajIndex is more than a few entries, it would then start to fill starting at TrajHead again. In this way, the buffer fills and empties automatically. When TrajHead or TrajIndex is 10 and gets incremented, it Will be reset to zero and start again from there.

As the Engine knows only about 200us of data at a time, it obviously does not deal with acceleration or ramping, the Application takes care of that by filling the appropriate number of steps into each location in the ring buffer.

Ring Index's 0 1 1 2 3 4 4 5 5 5 0

In our example, you can see the move slowly move from 0 – 1, then more quickly to 2, then 3 then slowly from 4 to 5 than stop at 5. This is a shrunken example of a ramped move from 0 to 5 for that axis. Therefore, as you can see the ring buffer knows nothing of speeds or accels, the resultant speeds from the port are simply a timed output

of the ring buffers information. This is why stopping can be a problem, you cannot simply tell it to ramp down, you have to start adding a ramped down motion. Since we have 4096 entries, each being 200us long, we have $4096 * .0002 = 819\text{ms}$ of motion that may be stored at any time. In faster kernel speeds, the time is less, and at 100 KHz, the time is only 40ms. This is why you need a very fast CPU to allow for 100 KHz, because the application will have trouble filling faster than the engine can empty the data. You do have to keep in mind that as the App is filling the ring, at the exact same time the Engine is hard at work emptying it. This necessitates a very fast filling routine in the application. If the application cannot fill fast enough you'll find the Buffer Load% on the Diags screen shows 100% quite a bit during a long move. That is your clue to slow the kernel.

Jogging and other motion is not permitted when the TrajHead is not equal to the TrajIndex. Only ring buffer movement will occur if there is data in the ring, and as explained above, its all auto magic from the point of view of the application, it only fills. It empties as if by magic.

FLOW

As explained, Jogging and such are simply Engine responses to keyboard or commands put to the Application. The application is simply filling in structures and massaging data to keep the ring filled and the commands flowing, waiting when necessary. So lets look at how that all flows. Lets look at the example of what happens from the time you load a program to running it. This may be unimportant to most, but I suspect its important for some to get a grasp on the actual program flow.

Mach3, when you load it , loads all your settings, and fills hundreds of variables with flags to indicate to it what to do in various situations. When the loading is done it doesn't appear to be doing anything, but in truth Mach3 never sleeps. Its always at

work on one or more of hundreds of subtasks. First lets consider what's going on when no motion is being commanded and the program is sitting "Idle".

Mach runs in 1/10 second intervals. That is to say that every 1/10th of a second the main system timer is told to start an update process which does the following.

1) First, it looks to all controls , each of which are windows of their own. DRO's, Buttons, LED's Tickers..ect.. are all told one by one to update themselves. This goes much faster than you'd think on a screen with over 800 windows running because only visible controls are updated. This is to save time, so the LED's and such on the Diags screen when your on the program screen are not updated, they are ignored. But any visible control is told to update. This means every LED or DRO is told to check its value, and if it differs from the last one displayed, update its display. This is done for all visible controls every 1/10 second.

2) GCode windows and Toolpaths are told to update. They know internally if positions have changed, or the current Gcode line has changed and will redraw to reflect that. The GCode display whenever the trajhead is no equal to the trajindex pointer will display whatever the current ring buffers ID variable tells it to. This is why the GCode display automatically shows the current line.

3) Indexes and Inputs are checked to see if they changed, and if they reflect the final stage of any known running sequence or wait condition. A macro for example may have called SystemWaitFor(INPUT1) , so INPUT1 will be checked to see if that wait object may have been canceled by the driver since the last 1/10 interval had passed. Its actually the engine that stops the wait condition, so waits are cancelled within 40us in most systems in PP mode.

4) The ring buffer will be checked to see if there was a wait in effect for filling it more. In this case motion is in progress, but the system is "idle" while it waits for the engine to use up some if its ring buffer data. No sense in filling just one entry so we wait till there is at least 500 free slots in the ring.

5) Any commands requiring processing will be processed.

Commands can stack up waiting for the Ring buffer to empty for example, or waiting for other events to clear prior to allowing them to trigger their actions. Commands such as spindle/on or off, waiting macro calls etc, all fall into that category or waiting events.

6) The trajectory planner is checked to see if it has remaining movement. If so, and the ringbuffer has room, more moves at 200us intervals are calculated and the ring is refilled.

So when a program is run, the process is to fill the planner with moves up to the number of lines set by the lookahead configuration value, OR to the next wait event type of command. For example :

```
G1X10
G1Y10
```

G1Z10

M5

G1X0Y0Z0

Will fill the trajectory planner only up to the M5. When the M5 is interpreted, the planner stops getting lines as the "wait event" of a command is added to stop the spindle output. Processing of the next line (G1X0Y0Z0) will not even get read from the input file until all previous movement has stopped, and then after all commands are executed. (M5) , at that point the last line would get read and executed. As each of the first three lines are read, their mixed feedrate is calculated as a function of the acceleration settings, max velocities and Feedrate commanded for each axis in the mix. Since mixed distances, max velocities, and accelerations of the various axis are often at odds with each other, this causes variations in the pulse stream which are quite normal, but makes planned motion not as smooth as the Jogging. Since Jogging doesn't have to respect the other axis settings, its movement is more pure and often leads to the comment that jogging is superior to GCode motion, unless the various axis are well tuned. As each moves motion distance and accel/vel calcs are complete they are added to the planner. The planner controls the CV'ing of the lines, as well as taking care of making each output equal to 200us , or 5 times the kernel interrupt time in faster modes.

Each pass through the Applications 1/10 second loop, the ring buffer is filled from the planner, if it is not empty, and the Engine automatically performs the motion required by the ring buffer. While filling the ring buffer the Application recalculates the pulse distance averages to further smooth the output timing to make the binary frequency stream as mathematically pure as it can be on that interrupt levels time base.

The effect of the time base calculations means that using a higher frequency for lower speeds is actually advantageous to the user, BUT has to be traded off to processor speed and the power required to fill the buffer faster than the engine can empty the buffer. For this reason, and the users pathological drive to always use the fastest speed he can thus creating support problems on slower computers, my recommendation has always been to use the lowest kernel speed possible. Now that computers are faster and I suspect most people use 2Ghz or better coming in, I don't mind admitting that a 60Khz run even with motors that don't move beyond 20khz is the smoothest option to use. Not necessarily the best option, as system criticality has to be considered. (A 4096 buffer at 60Khz holds only 341ms as opposed to the 819ms buffer of 25Khz mode.). Since things like feedhold need to start adding a ramped down move to the ring buffer, 25Khz mode will take up to a max of 819ms to start to slow, while 60Khz will take only 341ms. Almost three times as fast. So things such as that need to be factored into the decision as to what settings to use. The faster you select the better your response, but the easier it will be to lose steps by switching screens or starting your email program and such.

TIMING

Mach3 uses Interrupt 0, which is a very powerful interrupt, but the one enemy of this is the DMA cycle. DMA , when it occurs to get Hard disk data, or to send audio data, CAN stop the timer because the processor doesn't see DMA transfers, the CPU simply goes to sleep for a bit and Windows doesn't see that anything has occurred,

nor does the Engine. Small disk transfers will have no effect as they are typically less than the period of one interrupt, but large transfers can cause lost steps as the pulse timings get ragged. So pick a speed for your system that takes all the above into account, and at all times refrain from doing other work while cutting, specifically those that cause a lot of disk traffic.

Loading:

When you select a program to load, mach will clear any buffers it has. It then sets flags to tell the engine any movement is fake and not to send it out. The loader then simply sets the current line to zero and tells the program to run. It does make a copy of the current state engine as well just in case any dangling states are left set. It restores the state back to start conditions after the initial load/run. When the planner sees that the Engine is set to loading mode, it runs very fast. Instead of 200us per move, the planner will run though moves at 6ms per move. This means a file typically runs at 30 times faster than normal. This allows it to run the file quickly both as a way to generate a toolpath as well as a way to check for errors. If any line triggers an error during a load run, the Gcode display stop motion, the program is halted and the error is put out on the status line.

During the run, a database file "display.dat" is created with a list of moves in it, this list is used to display the toolpath and to create offset information for the G41/G42 offsets in there are any in the file. In fact, if G41/G42 is used, the file is run twice, once to see the original moves, and a second time to calculate offsets from them to display in white on the toolpath.

The addition toolpath display of the comped moves is stored in a file called CompDisplay.dat .

Once run through, the file is rewound even if not told to so the display will start at zero after a load. You are then ready to press cycle start as long as the unit is not in EStop mode.

EStop & Safe Mode:

In the interests of safety, Mach3 always starts in EStop mode with the flashing EStop button. There are 6 enable outputs that are tied to this operation. They are programmed to turn on , one by one with a couple hundred ms of time between each one. This is to stop excessive current being drawn from your power supplies as all the drivers get enabled at once. Many don't use that feature but it is there for that purpose. Having the drive power up from the enable lines can be problematic for some as the power is removed from them whenever an Estop occurs, but in terms of safety it is a good idea to have a main relay kill all motor mower anyway when an Estop occurs. So enabling each drive with an Enable signal is a good way to stop the high current at startup of the system.

Allot of things will trigger an Estop, limit switches, Soft limits and a few other things. The Engine is responsible for most of them as it can respond very quickly within one interrupt period. The Debounce setting in the config simply tells the engine that any signal is to be ignored unless it is seen to be active in a Debounce number of

consecutive interrupts. So a debounce of 1000 at 25000 mode is $40\mu s * 1000$ or 40ms of time.

Pressing the Rest makes you go into "Safe" mode where the EStop stops flashing and your ready to go. Of course the EStop button checks to make sure its allowed to reset before doing so. Warnings are sent to tell you why it wont reset if there is a problem.

INPUTS & OUTPUTS

The engine handles inputs and outputs as simple semaphore triggered events. There is a memory structure called InSigs in the Engine and one called OutSigs. There are 53 input signals, and 30 output signals. They are based on the following two structures.

// output signal structure

```
struct OutputInfo
{
    bool active;
    char OutPin;
    char OutPort;
    bool Negated;
    bool Activated;
};
```

// input signal structure

```
struct InputInfo
{
    bool Active;    // signal active?
    char InPin;    // which Pin
    char InPort;    // which Port
    bool Negated;
    int ReqState;  // Required State on Activation
    bool Activated;
    bool Emulated;
    int EmulationKey;
};
```

Each input and output is named, but that's a simple defined number from 0 to maxsignals for either array of structures. If the Active variable is set to true, the system considers that signal to be turned on. The Port and pin tells the engine to fill or trigger that signal depending on the port number. If the port is 1 or 2, then the Engine will handle the signal, if it is not, then a plug-in is considered to be in charge of that signal. The Activated variable tells the engine if the signal is on or off, and the LowActive tells the engine to trigger a high or low output on the requested pin, or to see an incoming high or low as "active" to set the activated variable in an input signal structure. For input signals keyboard emulation can be used and that's the other variables in the insig structure. If the port is not port 1 or 2 though, the engine will do nothing with that input or output. A Plug-in will then be expected to fill those variables. Mach3 will check the inputs each 1/10 second to see if anything needs to be

done once a signal triggers, but for time critical inputs such as EStop , limits or probing, the Engine will handle what needs to be done as only it can respond fast enough.

MACROS

Mach3 uses the Cypress VB engine to accept scripts and user macros. This is done by a Macro thread, which responds whenever it sees that a CString variable "Macro" has been set. Any script macro requested is first held till all movement stops and the ring buffer is empty, then the macro command is translated to the proper file which is then loaded and the text placed in the Macro variable. This has a maximum of 65000 characters. When the variable contains a text input, the Macro thread will automatically run that script, then empty the variable, and the program will continue. This is why a macro cannot call another macro. The threader only looks to one variable for text and there is no secondary storage for another script. The code is a bit weak there and will likely be strengthened over time as it's a complex connection to get just right. When a script is loaded certain constants are loaded with it such as Sleep , the names of the signals, (INPUT1 , OUTPUT1 ..ect.),and the scripts may avail themselves of those constant declarations.

Encoders & MPG's

On each interrupt the engine will also count encoder positions and track the counts. These counts are put in the following structures of which 7 copies are maintained. Four for simple encoders and 3 for MPG's.

```
struct Encoder
{
    bool Active; // is the encoder turned on?
    int APin; // A Pin designations
    int BPin; // B Pin designations
    int APort; // A Pin Ports
    int BPort; // B Pin Ports
    bool LastStateA;
    bool LastStateB; // storage for the last known state
    int Count[2]; // Index counts for each encoder
    int Memory[16];
    int ActiveMemory;
    int Color;
};
```

The same rules apply as for inputs and outputs. If the port number is not 1 or 2, then the engine will not fill those variables and a plug-in or Brain is expected to do so if they are enabled. On each loop of the applications run it will check the MPG and encoder values, update any DRO's involved and , if the jog mode is set to MPG will move the axis in correspondence with the MPG mode in effect. The only reason MPG's must be switched on is that when the MPG stops and its velocity is zero the DEC variable of the jog registers is set to true to auto stop any jog. A rewrite of this

will likely allow for Jogging via MPG at any time and will probably be done in future. When doing an MPG jog the current velocity of the MPG must be calculated by the application if in printer port mode, or sent in via brain or plug-in if using plugins or modbus devices to set the mpg. Examples of the timing clock interface are in the current example brains showing a g100 using a MPG from its encoder inputs.

MODBUS

Modbus connections to Mach3 can be made as Serial or TCP. Each selection uses its own thread with an update timer of 25ms to control its IO. Its best to think of ModBus simply as a way to fill certain memory blocks in Mach3 with data. In the case of Serial mode ModBus, there are two data structures (MasterInputs[1024] and MasterOutputs[1024] which are filled with data from a modbus device, or used to send data to a ModBus device in a timed way. In the case of TCP ModBus or Serial Plug-in Modbus, two different structures are used as the ModCon block.

```
struct ModCfg
{
    short Data[128];
    char Comment[80];
    bool Enabled;
    int ModAdd; //modbus address
    int nReg;
    int Input;
    int Port;
    int Slave;
    int Refresh;
    int timref;
    int status;
    char msgstatus[40];
};
```

```
struct ModCon
{
    ModCfg cgf[65];
};
```

The ModCon structure then contains 65 copies of the ModCFG structure. Every 25ms or so , the ModBus facility looks at each of the 65 cgf's in the ModCon and determines from its variables if data needs to be transfered into or out of the Data[128] block of that cgf. The end result is all the 65 Data[128] structures are sent or received as programmed by the various variables setup in the config/modbus settings in those modes. The data are then used by Brains to effect logical operations which may otherwise have been impossible. For example a Brain could be told to look in the ModCon at cgf[10] in data[12] and if bit #1 is set turn on the Estop. I wont explain the Brain interface to that level, but one should keep in mind the only true function of the ModBus interface is to get and send data from a plc type of device. From there the logic is based purely on script work or Brains work. In the case of Serial ModBus macros are typically used, though Brains can also be used, while in the

case of TCP ModBus or Serial PlugIn ModBus the Brain subsystem is the only way to deal with the data.

There are various Doc files and Videos that explain the scripting and Brain process in fairly good detail so we won't go into small details here. Suffice it to say that Brains are dealt with every 1/10 second and are seriously fast. Much faster than VB Scripts and can be considered instantaneous in most instances taking only 1 – 4µs to do their job. They aren't meant for huge logic projects but rather as nice simple routines to allow for minor IO flexibility in terms of what a signal from a plc can control.

PLUGINS

Ahh, to the meat of the matter. Most readers of this document will probably have plugins in mind. A Plug-In can do virtually anything, it has a lot of power over how mach3 responds to the user. It can modify present capability or add functionality not in the core program. It has a steep learning curve however, but I've been surprised by a few who have done amazing things in plugins without a single question to me as to how to do it. Those individuals are truly at the top of their game to be able to do so. Even I stumble in Plug-in work at times. We will examine plugins closely to aid in not only their use, but in their creation in terms of what can and cannot be done.

Many examples are around for plugins. We'll use one here that controls a mythical device called the "ServoRun" Device. The "ServoRun" was invented by Einstein, it has a perfect command set, never loses a step, and is very fast. Its interface is simpler than most devices (Being perfect) and so we should have no trouble writing a plug-in for it here.

The files for a plug-in comprise only a few, but a large includes folder. The includes folder allows access to many if not most of the underlying variables in Mach3. This includes the Block of shared memory, the App, and the Engine itself. The Engine is still used in external device plugins as they are simply expected to emulate the engines use of the printer port in many instances. Let's develop one from scratch with more explanation than has been given before so one can see exactly how to write a plug-in. Keep in mind that other types of Plugins are easier to write, simple IO or video don't need to interface quite as tightly as a motion interface, but the theory is the same, think of a plug-in as simply dealing with all the shared memory of Mach3, while the App deals with the vast majority of the logic to keep things running..

Required File: MachDevice.cpp, MachDevice.h, (and folder MachIncludes)

These files contain all the routines that Mach3 will call. It's good programming practice to have a

File called MachDevImplementation.cpp which contains calls referenced by the MachDevice.cpp calls. This makes the MachDevice.app a file which needs no modification when you write a plug-in. Or at the very most only minor modifications. In MachDevice you will see the Mach3 hooks and pointers that are used. It's my experience that many C++ programmers are not really comfortable with pointers and pointer recasting, so the MachDevice.cpp can, for some, be considered a black box operating in a magical mode where its only purpose is to make sure the routines in

MachDevImplementation.cpp at the proper times. We'll take a look here though at exactly what MachDevice.cpp does..

We'll start with the .h file which defines many things used by MachDevice.cpp. Here is a current MachDevice.h's internals from top to bottom. We start with some pointer type declarations.

```
typedef void (CALLBACK *NoParms) ();
typedef void (_cdecl *OneShort) ( short );
typedef double (_cdecl *DoubleShort) ( short );
typedef void (_cdecl *VoidShortDouble) ( short , double );
typedef bool (_cdecl *BoolShort) ( short );
typedef void (_cdecl *CSTRret) ( CString );
typedef void (_cdecl *VoidLPCSTR) ( LPCTSTR );
typedef void (_cdecl *VoidShortBool) ( short, bool );
typedef int (_cdecl *IntShort) ( short );
```

These declarations tell the compiler what Mach3 will expect for some types of functions. DLL's , as a rule are meant to contain code to be called by an application. They are not generally meant to allow for the DLL to call the main application on their own. This made it necessary to redirect some functions and create backward hooks for them In the list above for example, the VoidShortBool declaration is a way of telling the compiler that any function declared as VoidShortBool is actually a function declared in Mach3 as

Void Function(short, bool); , so though it looks complex to programmers not used to pointer indirection, its really just syntax necessary due to using the dll to call the main application..

The only other thing in the h file is
class CMachDeviceApp : public CWinApp

```
{
public:
    CMachDeviceApp();
    ~CMachDeviceApp();

// Overrides
public:
    virtual BOOL InitInstance();

    DECLARE_MESSAGE_MAP()

};
```

which simply declares the CMachDeviceApp as an application of its own, with full power to create and maintain windows of its own if it so desires.

Now if we look to the .cpp implementation of the device much more will become clear as to the actual interface.

We start with the normal defines of course, and then define as external any routines we wish the device to be able to call. This means if you add a routine to the device,

you will need to then create a new routine in the MachDevImplementation.cpp and add an external reference here. Heres what is in most MachDevice.cpp files at this point..

```
extern CString myProfileInit (CString, CXMLProfile*);      // initial access to
Mach profile

// when enumerating available plugins
extern void myInitControl ();                             // called during Mach initialisation
// you can influence subsequent init by actions
here

// **** Not used in typical device plugin
extern void myPostInitControl ();                         // called when mach fully set up
extern void myConfig (CXMLProfile*);                     // Called to configure the
device

extern void myUpdate ();                                 // 10Hz update loop
extern void myHighSpeedUpdate ();                       // called at 40Hz

extern void myCleanup();                                //Destrucion routine for
clenaup.

extern void myNotify(int message);                       // you can influence subsequent
init by actions here
extern void MyJogOn( short axis, short dir, double speed); //Routines to start
and stop Jogging..
extern void MyJogOff(short axis);
extern void myDwell( double time);
extern void myProbe( );
extern void myHome( short axis );
extern void myPurge( short flags );                     //for purging the G100
extern void myReset();
```

Just remember these are only references to allow the MachDevice interface to do its job outside of this file. In actual fact a plugin could be all done within MachDevice.cpp , the reason we call out to another file is just to keep things cleaner, and allow for a standard interface class of MachDevice.cpp.

Next we define a few variables that some plugins use.

```
CXMLProfile      *DevProf;
OneShort         DoButton;  // void DoButton( code )
DoubleShort      GetDRO;    // Double GetDRO( code )
VoidShortDouble  SetDRO;    // void SetDRO( short code, double value);
BoolShort        GetLED;    // bool GetLED( short code );
VoidShortBool    SetLED;    //SetLED Fucntion
CSTRret          GetProName; // CString GetProName()
VoidLPCSTR       Code;      // void Code( "G0X10Y10" );
IntShort         GetMenuRange;
```

The first one, DevProf , is a pointer to a XML processor. Mach3 controls an XML processor to give access to system variables. When it has control it is exclusive

control, but at specific times in the life of a plugin, Mach3 will relinquish control of the XML, and allow the plugin to modify system variables. You MUST only use this variable when Mach3 has given up control of it though. Ill note that in the routines we discuss when that occurs. The rest of the declarations we see are derived from the h files typing of certain pointer types. For example, DoButton is defined above as begin of type OnShort, which basically just translates to a declaration of

```
Void DoButton( short var );
```

As you can see it isn't as mysterious as it looks on first glance. Its just that Mach3 has to see them defined this way to be able to call the routines in the plug-in. Next we see...

```
HANDLE m_timerHandle;
bool KickTimer = false; //this will be a special case variable. See Update loop;
bool TimerOn = false; // this tells us the timer is not yet running. (See Update Loop
);
// CXMLProfile
CXMLProfile *AppProf;
CString ProfileName;
```

These are variables used in the for handling timers and the Profile name to be returned back to Mach3 to be displayed as a version number and plugin name. Now we come to more important matters.

```
TrajectoryControl *MainPlanner; //used for most planner functions and program
control
CMach4View *MachView; //used for most framework and configuration calls.
TrajBuffer *Engine; //Ring0 memory for printer port control and other device
synchronization
setup *_setup; //Trajectory planners setup block. Always in effect
```

These variables are pointers to Mach3's main blocks which we've been discussing in this document. MainPlanner is the App in our discussions, or most of the App. MachView is also part of the App, but is not required as much by a plugin writer. Engine is the "Shared Memory Block" we have been discussing. _setup is the one you haven't heard of till now, and it's the planner variable block. The main Gcode processors variables. It can give you access to Gcode variables, states, modes, and errors.

These variables are filled automatically during load to set them to point to Mach3. So for example is the coder wants access to the FeedRate variable in Mach3, he'd use MainPlanner->FeedRate. If he wants a signal level he'd use Engine->InSigs[INPUT1].Activated. In this way the plugin author can get access to almost any system variable. These variable then, are your hook into the logic of Mach3.

```
#include " ServoRun.h" // add includes for code you call here
extern ServoRun *sr;
```

These are our mythical device class declarations. If your device is called "MyDoorOpener", you'd write a class that knows how to control your door opener and declare that class and an pointer to it to be used. You'll notice the pointer to the device "sr" is external, this is because its expected to be created in the MachDevImplementations. Doesn't have to be, but often is..

```
CMachDeviceApp::CMachDeviceApp()
```

```
{
    // TODO: add construction code here,
    // Place all significant initialization in InitInstance
}
```

```
CMachDeviceApp::~CMachDeviceApp()
{

}
```

As you'll notice the constructor and destructor are empty. They aren't used by the system at all, but you are free to use them if required. I make no pretence that the structure I used was the best to use, I was learning as I went so today I'd probably do it somewhat differently, but this format does work well.

```
CMachDeviceApp theApp;
```

Is used by the system to control the application object, this is just a declaration you won't use personally.

```
BOOL CMachDeviceApp::InitInstance()
{
    CWinApp::InitInstance();

    if (!AfxSocketInit())
    {
        AfxMessageBox("Sockets Init Failed");
        return FALSE;
    }

    // Register all OLE server (factories) as running. This enables the
    // OLE libraries to create objects from other applications.
    COleObjectFactory::RegisterAll();

    return TRUE;
}
```

Here's our first routine, and it's a simple one. It's called as the plugin is loaded, before Mach3 is actually running full out, so you won't interface to mach3 here, you'll just use this to tell the system you intend to use Sockets in this case, if you don't know what Sockets are then you probably need not worry about it. If you do know then no further explanation is required.

// DllGetClassObject - Returns class factory

```
STDAPI DllGetClassObject(REFCLSID rclsid, REFIID riid, LPVOID* ppv)
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());
    return AfxDllGetClassObject(rclsid, riid, ppv);
}
```

// DllCanUnloadNow - Allows COM to unload DLL

```

STDAPI DllCanUnloadNow(void)
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());
    return AfxDllCanUnloadNow();
}

// DllRegisterServer - Adds entries to the system registry

STDAPI DllRegisterServer(void)
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());

    if (!AfxOleRegisterTypeLib(AfxGetInstanceHandle(), _tlid))
        return SELFREG_E_TYPELIB;

    if (!COleObjectFactory::UpdateRegistryAll())
        return SELFREG_E_CLASS;

    return S_OK;
}

// DllUnregisterServer - Removes entries from the system registry

STDAPI DllUnregisterServer(void)
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());

    if (!AfxOleUnregisterTypeLib(_tlid, _wVerMajor, _wVerMinor))
        return SELFREG_E_TYPELIB;

    if (!COleObjectFactory::UpdateRegistryAll(FALSE))
        return SELFREG_E_CLASS;

    return S_OK;
}

The above are all boilerplate, you neednt worry about them, they are used by the
system only to allow some DLL functions to operate. Leave them alone for the most
part unless you have a deeper understanding of what they are doing.
extern "C" __declspec(dllexport) void Notify( int ID )      //void DoButton( short
code );
{
    //here ID is the menu item just clicked..
    AFX_MANAGE_STATE(AfxGetStaticModuleState( ));
    myNotify( ID );
}

```

}

Heres out first declaration of importance. And its best if you understand exactly what's going on here so Ill explain this one in depth, most of the following work the same way in terms of how they are declared. You'll notice its declared as extern "C" __declspec , this is so that Mach3 can find it. During startup of a plugin, Mach3 looks for specific routines, basically all the ones with extern, if it find one it expects, it adds it to a list of calls that Mach3 will make. For example, if the routine Notify is not in the machdevice, mach3 will not trip an error, it just wont try to use it in the course of running the program. In this way some plugins can be shrunk down just by removing functions unused. But since the call mechanism is very fast, you really don't save much time by removing these functions, just leave their implementation blank if you don't use them. The Notify routine is called by Mach3 in the case of certain events happening. There is a list of define in the TrajectoryControl file that dictate what events can be used. They are

//Plugin Notifications and defines..

```
enum { EX_DDA , EX_VMS, EX_COMMAND, EX_SPINON, EX_SPINOFF,
EX_SPINSPEED, EX_MOTORTUNED
      , EX_SETUP, EX_FEEDHOLD, EX_RUN, EX_ESTOP ,EX_CONFIG,
EX_REWIND , EX_RESET };
```

These ID's tell the plugin what has occurred. If MyNotify is called with ID = 3, then the user has commanded the spindle to turn on. (EX_SPINON = 3), and if the user turns off the spindle, then ID will be equal to EX_SPINOFF. More notifications will likely get added over time, but the list includes most things the plugin should know about. The code we're on though is just a header declaration to tell mach3 that such a call does exist. More such declarations follow in the machdevice.cpp file. As you can see the code just calls MyNotify(ID) which means its simply passing the call to the MachDeviceImplementation.cpp file for processing. As mentioned before you don't really need to pass the call on, but to keep things clean and allow you to start a plugin easily, its usually best to keep the machdevice.cpp as generic as possible. A typical MyNotify function for the ServoRun object may look like this..

```
void myNotify(int message)
{
    if( sr == NULL ) return; //return if there is no sr object

    if( Engine->EStop && sr->curSEQ != SNOSEQ ) //if we are in EStop, kill all motion
    {
        sr->StopAllMotion();
        sr->curSEQ = SSTOPSEQ; //just in case;
        Engine->DwellTime = 0;
    }

    if( message == 0xffff ) //unsigned message, so check in hex..
    {
        //General Stop command called
        Engine->DwellTime = 0; //stop any dwell in progress..
        Engine->State = STOP; //stop any ring buffer calcs in Mach3.
        MainPlanner->SoftWait = 0; //stop any waiting in Mach3.
        sr->StopAnyJogging(); //to kill the Jog processor..
    }

    if( message == EX_CONFIG )
    {
        if( sr != NULL )
        {
            sr->Initialize(); //reinit the device, the user has just reconfigured.
        }
    }
}
```

```

        Engine->DwellTime = 0;
        return;
    }

    if( message == EX_MOTORTUNED )
    {
        if( sr != NULL )
        {
            sr->Initialize();
            Engine->DwellTime = 0;
        }
        return;
    }

    if( message == EX_SPINON || message == EX_SPINSPEED ) //this is Spindle ON;..
    {
        if( sr != NULL && Engine->Axis[6].Jogging )
        {
            sr->SetSpindle( MainPlanner->Spindle.ratio);
        }
        return;
    }

    if( message == EX_SPINOFF ) //this is Spindle OFF;..
    {
        if( sr != NULL ) //meaning the ServoRun is running
        {
            sr->SetSpindleOff(0);
        }
    }
}

```

You may have more messages you wish to process, the point is here is where you will notify the sr device to take appropriate action for whatever notification is incoming from mach3. You'll see in the general update function where you can do more asynchronous types of processing.

Next we see a few more rather small entries..

```

extern "C" __declspec(dllexport) void SetDoButton(OneShort pFunc)    //void
DoButton( short code );
{
    DoButton = pFunc;
}

extern "C" __declspec(dllexport) void SetGetMenuRange(IntShort pFunc)    //void
DoButton( short code );
{
    GetMenuRange = pFunc;
}

extern "C" __declspec(dllexport) void SetSetDRO(VoidShortDouble pFunc) //void
SetDRO( short code, double value );
{
    SetDRO = pFunc;
}

```

```
extern "C" __declspec(dllexport) void SetGetDRO(DoubleShort pFunc)    // double
GetDRO( short code );
{
    GetDRO = pFunc;
}
```

```
extern "C" __declspec(dllexport) void SetGetLED(BoolShort pFunc)    // bool
GetLED( short code );
{
    GetLED = pFunc;
}
```

```
extern "C" __declspec(dllexport) void SetSetLED(VoidShortBool pFunc)    //
bool GetLED( short code );
{
    SetLED = pFunc;
}
```

```
extern "C" __declspec(dllexport) void SetCode(VoidLPCSTR pFunc)    // bool
GetLED( short code );
{
    Code = pFunc;
}
```

These retrieve from Mach3 the addresses of the Mach3 functions used to do calls like SetLED() , they are only used by the plugin as it starts up as when Mach3 see's them, it calls them, passing the addresses of the actual Mach3 functions. SO one the call sets SetLed = pFunc , the plugin may from then on call SetLed(100, true) for example. As I say, this is done all automatically, so all you need to know is that the plugin may use Mach3 functions of those names. SetLed(led,state) Code("text"), DoButton(button) etc..

Next we can see

```
extern "C" __declspec(dllexport) char* SetProName( CString name)    // CString
GetProfName();
{
    static CString strPlgName, strVer;
    ProfileName = name;
    DevProf = new CXMLProfile(); //start up the Profile class for XML usage.
    strVer = myProfileInit (name, DevProf); // call implementers code
    strPlgName = AfxGetApp()->m_pszExeName + strVer;
    delete DevProf;
    return (char*)(LPCTSTR)strPlgName;
}
```

This is used to open the Mach3 current XML. Mach3 calls this routine at startup of the plugin to give the plugin the opportunity to get data from Mach3's current XML.

You'll notice it calls myProfileInit() . This is usually where the actual work is done. A typical myProfileInit would look like.

```
CString DefDir;
void myProfileInit(CString name, CXMLProfile *DevPro)
{

    CString s;
    //this gets the default directory DefDir in which Mach3 is located. And the profile
    //name ex. "Mach3Mill"
    DefDir = DevProf->GetProfileString("Preferences","DefDir","C:\\Mach3\\");
    Profile = DevProf->GetProfileString("Preferences","Profile","Mach3Mill");
    // The values stored in the XML file should represent text of the actual quantity
    involved.
    ControllerFreq = DevProf->GetProfileString(MyDeviceName, "ControllerFreq", "");
    return "Version 1.0";
}
```

In this variation, if the plugin is named " ServoRun.dll" then the SetProName will return the string ServoRun Version 1.0" to Mach3 as the plugins official name and the name that appears in the device selection dialog in Mach3 to select this device.

Next in the code of MachDevice.cpp we see :

```
extern "C" __declspec(dllexport) void StopPlug(void)
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState( ));
    DevProf = new CXMLProfile(); //start up the Profile class for XML usage.
    myCleanup();
    delete DevProf;
}
```

Since the plugin has opened a profile before, during the init, it can now open a profile in this routine with just a DevProf = new CXMLProfile(); This is because it already knows the name of the profile. This routine is used to close the plugin. The function myCleanup() should be used to delete any variables you created from the heap, or any classes you made. Typically , since you would have created a class object for whatever device your using, you'd delete that device in myCleanup as in this one for the ServoRun.

void myCleanup() //used for destruction of variables prior to exit.. Called as MACH3 shuts down.

```
{

    sr->ShutDown = true;

    if( sr != NULL )
    {
```

```

myReset();

DevProf->WriteProfileString(MyDeviceName, "Variable0", Vars[0]);
DevProf->WriteProfileString(MyDeviceName, "Variable1", Vars[1]);
...etc...
delete sr;
sr = NULL;
}

}

```

Here we are simply deleting the ServoRun object "sr" after setting it to shutdown, a Boolean used perhaps in a timer routine. Then using myReset to set the device into a mode where its outputs would shut down, then deleting the sr object, setting it to NULL and returning. We also saved some variables that the mythical sr object may have used for internal parameters. All of this of course, depends on your device or object your controlling.

As you can see, the way the flow works is Mach3 calls the MAchDevice.cpp when it needs to, and the MachDevImplementation.cpp holds the device specific routines that MachDevice.cpp sends to command up to. When Mach3 gets told to shutdown, this process then helps you shut it all down..

Next in the MachDevice we have:

```

extern "C" __declspec(dllexport) void DoDwell(double time)    // bool GetLED(
short code );
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState( ));
    myDwell(time);
}

```

This obviously gets called when Mach3 processes a G04 command. In this case the myDwell command would send a command to the ServoRun device and it would do a dwell process. Perhaps a routine such as this in the implementation file..

```

void myDwell( double time)
{
    if( sr == NULL ) return;

    Engine->DwellTime = (int) time * 1000;
    sr->DoADwell( time );
}

```

The sr device in this case is sent a command to start timing a dwell. The Engine->DwellTime is set to the time required. Mach3 wont continue until Engine->DwellTime is zeroed. It's the responsibility of

the sr device object to clear the dwelltime variable once it senses the device has timed out on that dwell.

The next function is only called if the plugin is a External motion engine and was selected for use in this profile. You can use it to create your device object in this case. You cannot use XML profiles at this point.

```
extern "C" __declspec(dllexport) void PostInitControl()
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState( ));
    //this routine is called after Mach3 has initialised. Use it as
    Init, BUT no usage of the XML files at all here. Only in Init.
    //this routine is handy for changing variables that Mach3 has
    loaded at startup. Usually, Mach3 will permanently save any var's you
    change here..
```

```
    myPostInitControl ();
    return;
}
```

A typical myPostInitControl function would look like this..

```
void myPostInitControl ()
{
    // The user has selected the ServoRun as the motion device
    // Create an instance of the ServoRun Class
    sr = new CServoRun();
    if (sr == NULL)
    {
        AfxMessageBox("Unable to create ServoRun Class");
        return;
    }

    sr->Initialise(); // Init the device
}
```

Here we have simply created a device object "sr" and initialized it. The details of your initialization and what to do if it fails is left to the plugin author of course.

Next in the Device class we have the InitControl..

```
extern "C" __declspec(dllexport) bool InitControl( void *oEngine , void *oSetup , void
*oMainPlanner, void *oView)
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState( ));
    Engine = (TrajBuffer*)oEngine;
    _setup = (setup*) oSetup;
    MachView = (CMach4View*)oView;
    MainPlanner = (TrajectoryControl *) oMainPlanner;
    TimerOn = false;
    KickTimer = false; //see update loop
    myInitControl ();
    return true; //start the printer port.. Use this for all devices that need the
printer port. For now..thats all of them..
}
```

Mach3 calls this routine during startup and uses it to setup the Block, App and planner variables for use in the plugin. KickTimer is used to startup a 40hz timer for fast processing work if it is required. Then the myInitControl is called. Heres a typical one for our ServoRun object..

```

void    myInitControl ()
// called during Mach initialisation. You can influence subsequent
init by actions here
// **** Not used in typical device plugin
{
    MachView->m_PrinterOn = false; //This signifies to MACH that this
is an external Control device driver.
    //This doesn't mean this device will run, the user will decide
what movement device to run in each profile.
    //use this only for a driver that actually controls pulsing..

    RangeStart  = GetMenuRange( 2 ); // request 2 menu IDs to use

    return;
} //myInitControl

```

Since this is an external motion plugin, the variable MachView->m_PrinterOn is set to false. This tells Mach3 that this plugin CAN be used to control motors, it will then appear in the device selection list. If it is actually selected for use, the PostInitControl function will be called. Otherwise postinit will not be called. SO this routine really only needs to notify Mach3 that the printer will not be used for output. The other call is to mach3 to reserve 2 menu items for this plugin. From that point forward, if the menu item is selected in Mach3, the myNotify will be called with the menu item ID as its message. This means the myNotify will have a message handler such as ..

```

if( message == (RangeStart + 1) )
{
    if( sr != NULL )
    {
        sr->DoFunctionForMenuItem#2
    }
}

```

This presupposes that you have a menu handler in the plugin. Not all do, but you may add one. The example plugins show a menuhandler class that adds in menu items To Mach3's menu system. The class uses the RangeStart variable to determine the ID's that added menu components will have in the menu system. The code for menu Additions is fairly simple, and the MenuHandler Class files are pretty intuitive to use. You can see them in the ncPod example series. The MeneHandler is normally implemented in the myPostInit function. Since myPostInit doesn't get called unless the user selects the device, the menu wont appear unless the device is actually in use. Heres a typical way of adding a menu without a menuhandler function just by adding the following in the myPostInit function.

```

static bool menued = false;
if( !menued)
{
    //" Startup of Menu handler"
    menued = true;
    CFrameWnd *MachFrame = MachView->MachFrame;
    CMenu *menu = MachFrame->GetMenu();
    HMENU hSubmenu = CreatePopupMenu();

```

```

    int pos = FindMenuItem(menu,"PlugIn Control");
    //here we can add menu items to Mach3's menu..
    int x = pos;
    HMENU control = GetSubMenu( menu->m_hMenu, pos);
    InsertMenu ( control, -1, MF_BYPOSITION, RangeStart , _T(MyDeviceName + " " +
SSVerString + " Config") );
    InsertMenu ( control, -1, MF_BYPOSITION, RangeStart+1 , _T("ServoRun Data
Monitoring") );
    MachFrame->DrawMenuBar();

}

```

Obviously if you do this you need to also handle the notification messages appropriately in the myNotify function.

If the user selects config from the config/plugins dialog in Mach3, the following function is called. It is not always implemented as the menu system works better for most users. But should the user select config from that dialog, this routine is called..

```

extern "C" __declspec(dllexport) void Config( )           // CString GetProfName();
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState( ));
    DevProf = new CXMLProfile(); //start up the Profile class for XML usage. Same as
Mach3's.

    //XML reading and writing can occur here..

    myConfig (DevProf);

    delete DevProf;
}

```

In most modern plugins, the myConfig routine simply puts out a message saying "Please use menu system for configuring this device".

Next we have a few functions called by Mach3 for special events.

```

extern "C" __declspec(dllexport) void Reset() //////////////// Called when Reset
is pressed.
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState( ));
    myReset();
    //Called when reset is pressed, at end of actual reset command in Mach3.
    //Check the Engine.Estop variable to see if we have reset or not..
}

```

The above is called when the user hits the reset button. It allows the ServoRun device to gracefully stop, or reset itself when required by a reset button press. The plugin in the myReset can check to see if the Engine->Estop is set or not to see what the end state of the reset buttons function was, to Estop or move out of Estop.

```

extern "C" __declspec(dllexport) void JogOn( short axis, short dir, double speed )
//////////////// Called when Reset is pressed.
{
    //Called when Jog is commanded. 0 for speed is Jog% jog, otherwise it is a new
jog%
    AFX_MANAGE_STATE(AfxGetStaticModuleState( ));
    MyJogOn( axis, dir, speed);
}

```

```
extern "C" __declspec(dllexport) void JogOff( short axis ) //////////////////////////////////////////////////
Called when Reset is pressed.
{
    //Called when jog should stop on a particular axis
    AFX_MANAGE_STATE(AfxGetStaticModuleState( ));
    MyJogOff(axis);
}
```

The above routines are called when Jogging is commanded. Normally in JogOn, the speed variable is zero, and the speed is controlled by the MainPlanner->JogPercent variable to scale the rapid speed of the axis to the right jog speed, however, when things like a shuttle pro are used and the speed is variable, then the speed variable will not be zero and indicated the required speed. The logic required for the device to jog will vary from device to device, but the plugin should check to ensure that the ring buffer is empty and that jogging is allowed in the current mode.

```
extern "C" __declspec(dllexport) void Purge( short flags ) //////////////////////////////////////////////////
Called when Reset is pressed.
{
    //Called when jog should stop on a particular axis
    AFX_MANAGE_STATE(AfxGetStaticModuleState( ));
    myPurge(flags);
}
```

```
extern "C" __declspec(dllexport) void Probe( ) ////////////////////////////////////////////////// Called when Reset
is pressed.
{
    //Called when jog should stop on a particular axis
    AFX_MANAGE_STATE(AfxGetStaticModuleState( ));
    myProbe();
}
```

```
extern "C" __declspec(dllexport) void Home( short axis ) ////////////////////////////////////////////////// Called
when Reset is pressed.
{
    //Called when jog should stop on a particular axis
    AFX_MANAGE_STATE(AfxGetStaticModuleState( ));
    myHome( axis );
}
```

The Purge routine is seldom used, if at all, but in a device like the G100 it is used to tell the device to kill any moves it currently is running. The Probe and Home commands are used to initiate those functions. Homing should be automatic to the plugin, Mach3 simply waits for it to be done. See the section on Sequencing for more details on homing and probing.

```
void CALLBACK MYProc(HWND hwnd, UINT uMsg, UINT_PTR idEvent, DWORD dwTime)
{
    static int ticks = 0;
    //If the update loop doesn't turn this back on in 20 ticks. ( 1/2 second)
    then kill the timer, user
    // has shut us down.
    ticks++;
    if( ticks > 20 && !KickTimer )
    {
        KillTimer( hwnd, idEvent );
        TimerOn = false;
        return;
    }
    //Alternately , lets make sure the user has us selected, if so, keep the
    timer running
    // in an indefinite loop of 40hz.
    if( KickTimer )
```

```

    {
        KickTimer = false;
        ticks = 0; //reset the test
        TimerOn = true;
    }
    //We get here at 40hz only if the user has selected us as ON, and the Mach3
    programs update loops are running correctly
    //Any interruption of Mach3 will operate as a safety watchdog and we wont get
    here in that case.

    if( sr != NULL) sr->myHighSpeedUpdate();
}

```

This procedure is a 40hz timer used by some devices as an asynchronous method of processing. The normal update is synchronized to Mach3's 1/10 second loop, this timer is used to have a 40hz callback which is not related to Mach3's execution timers. TO use it you need only turn it on in the next routine, which is the main update loop. If the hi speed timer senses that the updates from mach3 have stopped (Mach3 has shut down) then the kicktimer variable will stop the hi speed timer on its own..

In Mach3's 1/10 second loop, the system will look at a list of all plugins current enabled, and will call the Update() function in the MachDevice.cpp file that belongs to the plugin. It is this Update() routine that tells the plugin to check itself, update any system variables that are in use, and do whatever else it needs to do for housekeeping. Any plugin should keep its time short in this routine. Its accepted that certain plugins are more complex than others, and generally even hard working code can be run in this routine , but things like displaying a Dialog and holding the routine in a DoModal() call are illegal. The system will , while waiting for the DoModal continue to call Update() and weird things can happen. In general keep the calls to the Update routine as simple as possible, and with no hard coded holdups.

```

extern "C" __declspec(dllexport) void Update() //////////////// UPDATE LOOP 10
Times a Second.
{
    // This is your main update loop. Approx 10hz or so..
    // We'll get smoother control at a higher frequency, so start a timer that
    // runs at 40hz. This Update loop will start the timer once and it will
    // run periodically from that point on. If the timer routine sees that this
    routine
    // fails to run for a half second or so, it will shut down the timer. If this
    // routine starts operating again, it can start the timer again if it sees it has
    AFX_MANAGE_STATE(AfxGetStaticModuleState( ));
    KickTimer = true;
    if( !TimerOn ) SetTimer(NULL, 1, 25, MYProc);
    if( sr != NULL) sr->myUpdate();
} // Update

```

TimerOn is a global Boolean used to tell the system the timer is running. Initially it is set to false and you'll find it at the top of the declarations in the machdevice.cpp file.

That completes the set of functions that Mach3 uses to interface to a plugin. The only thing you need other than the above is your own sr object that implements the logic required to actually run the device. The plugin examples such as ncPod, and the G100 plugin show the devices using the above functions, and the logic of course will vary widely depending on the device used. Simple IO devices will have the easiest

time. They typically only look to an external device, joystick, HID , or transmission, get a status word and use that to set input signal in mach3 or to take mach3's output signals and send them to a device. Heres an example of getting a IO status word from your device and sending them to the Mach3 signals . Its usually done in the myUpdate routine for your device.

```
int inword = sr->GetInputWord();
for (int x = 0; x < nSigs; x++)
{
    if (Engine->InSigs[x].Active && Engine->InSigs[x].InPort == 3 )
    {
        int pin_state = (inword >> 1) && 0x01;
        if (pin_state == 1)
            Engine->InSigs[x].Activated = Engine->InSigs[x].Negated;
        else
            Engine->InSigs[x].Activated = ! Engine->InSigs[x].Negated;
    }
}
```

As you can see the inword retrieved form the device is simply checked to see if it is Set, and if so, the input pin is set to the input signals level. Of course various logic can be applied here. Similar logic is used to create an output word for a plugin to send using the Engine->OutSigs[n] variable list . If an output signal is "activated" for example, typically a word bit is set to a one and then sent to the device.

By examining the headers in the MachIncludes folder, you'll find most structures you need to use in a plugin. The Engine-> variables are found in the engine.h file, the MainPlanner-> variables are found in the TrajectoryControl.h file, and are the typical variables one needs to use for most operations you'd like to attempt.

As time permits Ill try to add more to this document in terms of what variable does what and what you can use to bypass certain logic in Mach3, but hopefully all this rambling has given you a start to understanding how a plugin does its job, and how Mach3 provides an interface to allow for plugins to use Mach3s data in various ways. Reading the plugin examples is the best way to see how I , or others, have decided to massage the system into doing as we wish in the plugin environment.

Well, that's it for now, I'm talked out. Have fun, remember that in the world of plugins, anything goes, you'll easily lock up mach3 if the logic of a plugin defies what Mach3 expects. There's allot more to say, and over time I'm sure it will be said, but I hope this document helps to explain some of the limitations and mysteries surrounding Mach3's internals.

Art Fenerty
January 28th, 2008

