A Computer Driven Train Controller
Article 9 – The train controller software

This is the final article in a series which has been exploring the implementation of a computer controlled model train set. In the previous eight months, we've been covering a lot of ground related to model trains and how to control them electronically but the one important topic upon which we've yet to touch is how the software does its job.

Well lastly but not least, we're going to spend some time looking at the example train controller programme. Indeed, some of you are bound to have your main interest in computers and software and for you, this final article and the source code itself are likely to be of most interest.

For many reasons, I chose to write my train controller software using the 'C' programming language and the Linux operating system. But there's no reason that the software, be it either your own code or the actual example code, cannot be adapted to work in a different environment - such as an old personal computer which has remained idle since some time in the last century. If you're considering a port to a different platform, I'd still suggest that you still plan on starting with the provided software and using Linux. This will surely be the fastest way for you to confirm that your hardware is working and to see your train set running under computer control.

Right at the outset of this article, I'd like you to understand that I'm encouraging you to modify or rewrite the software for yourselves. Although you may not consider it necessary to delve into the source code and change it, you're certainly going to learn a lot more about real world control and computer operating systems if you put on a brave face, take risks and just start to do it. Hey, I can almost guarantee you'll have great times as you introduce different types of bugs and then witness spectacular model train collisions! Of course, you don't need to start writing code immediately and that's the whole purpose of having provided the example programme.

A birds eye view of the train control software

In order to manage a train set, a computer needs to control the power supply for each engine and at the same time, follow each train's position, from front to rear, as it moves about the train set layout. The computer must make sure that points are correctly set in advance of each moving train and that trains which are travelling on paths which would otherwise meet (i.e. crash) are directed to slow down, stop or take an alternate route in order to avoid the disaster. To be able to do all of these things at precisely the right times, the software needs to be carefully structured.

The good thing about computer software is that there are always many different ways of achieving the same end, but this is also the bad thing too. If before reading further, you were asked to design your own programme, I'll bet that you'd each come up with different ideas and methods. How could you know whether your particular choice would be the best without seeing and understanding the others? You never could!

No matter what your natural approach would have been, I'll argue that the simplest way of writing reliable, maintainable and flexible train controller software is to divide the overall complexity into many different stand alone chunks. I'm talking about multi tasking.

When I first found the inspiration for commencing this project, I was an undergraduate student at the University of New South Wales and I had taken an Operating Systems subject in which the lecturer used a computer controlled train set to teach what we needed to learn. If you're a "hobbyist

programmer" or if you're still young in your software education, you may be like I was when I started that subject. I had never really encountered multi-tasking from a programmer's perspective and I had no idea about how to divide and structure my code into "tasks" – I hadn't trained myself to think like that.

The way I had previously written software, even very complex programmes, was to lay out the complete problem as a single series of steps and let the logic flow sequentially. In a multitasking system such as Linux, this would be like writing the entire programme as a single task. The Operating Systems subject helped me learn a better approach.

Taking a complex software problem and breaking it down into many autonomous functions is almost like dividing a programme into different subroutines. But rather than execute the subroutines sequentially and in a particular fixed order, multitasking effectively runs them all at the same time.

To prevent the seemingly independent tasks (or "processes" as they're sometimes called) from falling into a state of uncoordinated chaos, good operating systems support several efficient ways for tasks to communicate with their peers. Only by meticulously communicating between themselves can all the tasks in a complex system ward off pandemonium. But believe it or not, even in the most complicated systems and even though many different things may be happening at the same time, it's relatively simple to keep all under control and to ensure that important business gets attended to with higher priority. However co-ordination happens by careful design, not by accident.

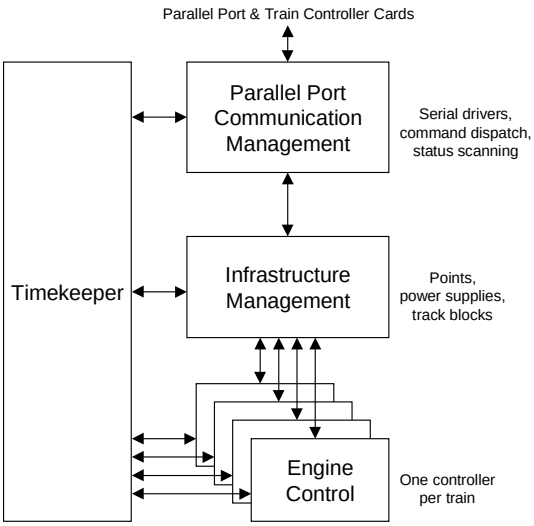

Figure 1 High level structure of the train scheduler software

The high level break down of the train software is drawn in Figure 1 and shows the birds eye view of the structure. The boxes represent major groupings of tasks and the lines of communication between them are shown with arrows. With a little imagination, you could liken the construction to an organisation of people in an office or more specifically, an organisation of people at a railway.

Each train is represented by its own task (Engine Control). The engine control boxes are like train engineers – one engineer drives each train and makes the step by step decisions about whether to speed up, slow down or keep going.

Train drivers cannot set points however, this job is left for "signal operators", or in Figure 1, the Infrastructure Management group. In fact, it is also up to the signal operators to ensure that trains do not collide. Particularly, when two trains intend to head in opposite directions on the same length of track, an Infrastructure Management task ensures that one of the trains is temporarily diverted or delayed so that the other can pass.

In order to communicate with the train controller cards, a few special functions (called Parallel Port Communication Management) are provided so that the Infrastructure Management group is not burdened with this.

Finally, along side all tasks, a Timekeeper is included to “remind” the others to do certain things on a regular basis. In fact, the timekeeper has one of the most important managerial roles in any real-world software control system because it keeps things happening at the right times.

A more detailed look at the train controller software

Sooner or later when your curiosity gets the better of you or when you find a bug that absolutely needs to get fixed as quickly as possible, you're going to need to become much more intimate with the source code. Why postpone the inevitable? Let’s become acquainted now!

If you haven't downloaded the software yet, you'll find it at ftp://xxx.xxx.xxx/xxx/train.tgz and when you decompress the archive, you'll see a number of different components. "train.c" is the heart of the code, "train.h" contains the major declarations, "config.h" is for the parameters which pertain to your layout, "controllerN.h" contains definitions which relate to the number of train controller cards you have. Some of the earlier articles in this series describe the configurations and definitions. This month, we're looking at the software in train.c.

At first pass through train.c, it's going to be hard to recognise the structure in Figure 1. But with the “organisation” of the train controller programme fresh in your mind, we can redraw its structure showing more detail (Figure 2) and the software might start to make a little more sense. From this point on, I'm going to assume that you're confident with the 'C' language, comfortable with the concept of multitasking and have the source code in view in front of you.

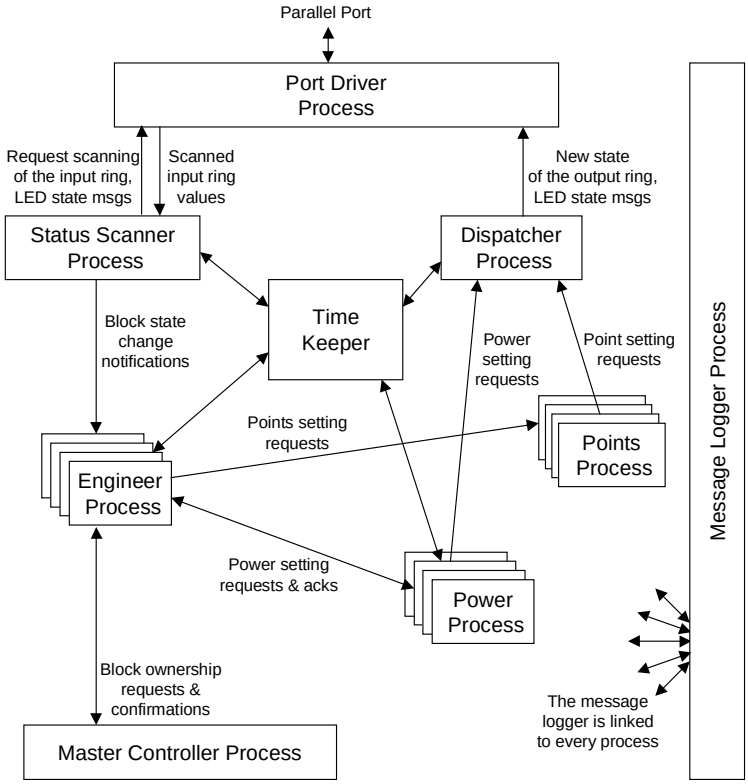

Figure 2 Detailed software architecture of the train controller programme

Imagine each of the boxes in Figure 2 to be a completely separate programme on your Linux PC and all running at the same time. In reality, this is very close to the truth because if you type the Linux command "ps" (to list all processes), you'll see more than a dozen separate tasks all named "train", (type it in a different shell window when the train programme is running, dummy).

If you start parsing through train.c, the independence of each task is not as apparent as the diagrams may have led you to believe. train.c is a large file and if you've printed it, you might want to consider indexing it in a folder to make it easier to flip through and find sections.

Looking from the top down, you'll first notice a preamble of comments and declarations and then you'll encounter some common routines for performing housekeeping functions such as initialising Linux message queues, shared memory and semaphores. These housekeeping routines aren't tasks at all, they're subroutines which are called by the tasks that we'll meet a little further down. Rather than spend time looking at the details of these subroutines now, lets wait until later because their full meaning will become more apparent when we study the tasks.

In any multi tasking operating system, the three main methods for inter process communication are messaging, shared memory and semaphores and particularly the last two are often closely interrelated.

Imagine messaging to be like email. The operating system's primary job when passing messages is to be the postman and to deliver them correctly. Tasks send short "notes" to each other and after a note has been sent, the originator can usually no longer see, change or retract it. What gets written in these messages and how the recipient uses the information is entirely up to each task and the recipients usually read these notes in their own good time - meaning, not necessarily immediately.

Shared memory is like a white board: when one task writes something, all other tasks sharing the memory can immediately see the changes and may freely erase or alter what has been written. But being able to freely and immediately overwrite another task's information is a recipe for chaos so semaphores are often brought into play to prevent this.

Through the use of semaphores, the operating system usually performs or helps perform the job of a meeting moderator so that different tasks access the shared memory in an orderly fashion and one task don't scribble on another's stuff before it's finished being written. Semaphores can be used in various ways but for the majority of time, it's best to think of them like the batten that a team of relay runners passes around: only the athlete holding the batten is allowed to run and in the same way, only the process "holding the semaphore" can access the shared memory. But if you know what you're doing, you don't have to use semaphores all the time when accessing shared memory: some of train's shared memory is controlled by semaphores and some is not.

Now that we've introduced them, it will be easier to understand the concepts of messages, shared memory and semaphores by seeing how they're used in the code so with the housekeeping in order, let's continue through the source listing. Further down, you will encounter the first of the tasks shown in Figure 2.

The Log Process

Generally, messages in a multitasking programme can be simple or complicated and usually contain one or more data structures. In the case of the Log Process, each message is actually a simple text string which is to be printed on the screen or logged to a file. The aim of the Log Process is relatively straightforward: it knows whether to write messages to the terminal, to a file or to both and all other processes within the train programme can depend on the log process to reliably do the right thing with any text message.

Whenever a programme has any I/O (Input / Output) to perform, such as writing debugging or informational messages, you can be almost certain that the operating system will force the tasks which do the outputting to slow down to the pace at which information can be written to the I/O device. Practically, this means that the operating system will stop the "I/O bound" tasks for periods as short as milliseconds but usually longer - up to as long as seconds while the I/O device itself catches up and flushes the information which the task had written.

In a real world control programme like train.c, it would not be good if any of the major tasks were blocked (stopped) by the operating system. If any of the processes directly controlling the locomotives or the power supplies or points went to sleep waiting for a message to be logged, a train could crash while the instructions which would have prevented the catastrophe were pending.

The nice thing about having a centralised message logger is that any task can asynchronously send messages through the logging process rather than directly to the output device itself without the burden of trying to accommodate the risk of being delayed by the operating system. As long as messages are relatively short and tasks don't prolifically flood them to the logger, most operating systems won't cause the sender to ever become blocked.

As with all of the tasks in the train control programme, the Log Process is implemented as an infinite loop. The basic idea of the loop is that the process waits for a message, looks at what the message is saying and on this basis, reacts in one way or another. Sometimes, this form of construct is called a

"state machine" which in simple terms, means that the task is either in a "waiting for a message" state or in one of possibly several "processing a message" states.

In high level terms, the loop is set out like this:

```
Log_Process()
{
    for (;;) /* this is an infinite loop */
    {
         wait_for_a_message() ;

         if ("message is the first type")
             process_type_1() ;

         else if ("message is the second type")
             process_type_2() ;
    }
}        /* etc */
```

This basic structure is very important in multitasking software and you'll see it and variations in all sorts of real time and non real time applications.

Although I haven't described explicitly what may happen in the loop as each message is processed, it is common that this task will send one or more follow on messages to other processes. As the other tasks receive the secondary messages, they in turn process them and send further follow on messages. You can almost imagine that messages are sometimes exchanged in a sort of chain reaction like a line of falling dominoes. Message ping pong is normal in a multitasking application and it is the responsibility of the operating system to give each task an opportunity to send and receive its messages in a fair and timely manner.

Actually, all tasks also have a responsibility to be "well behaved". In order to be "well behaved", a task needs to tell the operating system when it's finished processing each message, in effect volunteering to stand aside to be temporarily put to sleep. If there happens to be another task waiting to process a message, the operating system will schedule that other task until it too volunteers to stand aside.

The Port Driver, Status Scanner and Dispatcher

A little further down in the code, you will encounter three more processes, the Port Driver, Status Scanner and Dispatcher, which form the basic communications centre of the train programme. In Figure 1, these three tasks are described as "Parallel Port Communication Management". Let's start by looking at the Port driver which is the only task in train.c which is permitted to read from and write to the ring.

When many tasks run together but need to share resources like disk drives, network interfaces and the screen, the chances are high that two tasks might want to access the same resource at the same instant in time. If an operating system allowed this, then one or more programmes would crash, lock up or fail in some other way.

Implementing a supervisor task (such as the Port Driver) to control each "critical resource" (like the ring) is one of the several ways that programmes like train.c and also operating systems solve this problem. The idea is that a supervisor task is the only one which is permitted to directly access a particular resource and so all other tasks wanting to use the resource need to line up behind each

other to have the supervisor do things for them on their behalf. This technique is an important multitasking tool and the train programme uses the same approach to manage the point power supplies and the train power supplies.

Parts of the logic in the Port Driver code are similar to parts of the programme test_ring.c which was presented in a previous article. Importantly though, in train.c the Port Driver algorithm has been modified to work in a multitasking environment by the addition of that "infinite loop and wait for a message" construct shown above. The Port Driver can receive many different types of message: a request to send a string of bits to the output ring, a request to read a string of bits from the input ring or requests to turn the parallel port interface card's LEDs or power control line on or off. Most importantly though, the Port Driver will only process one ring request at a time.

Looking further down train.c, you will reach the Status Scanner which is responsible for periodically asking the Port Driver to read the input ring so that it can determine whether any of the input status bits have changed. This is an important part of how the train programme keeps track of each engine's progress around the layout.

Every time the Status Scanner receives a wake up alarm from the Timekeeper, it asks the Port Driver to read the input ring for it. After the Port Driver replies, the Status Scanner checks to see if any of the ring status bits have changed and if so, it notifies the appropriate Engineer or Controller Process (which we'll meet below).

Next, you'll meet the Dispatcher Process which co-ordinates and formats sequences of bits to shift out onto the ring. A new sequence of bits need to be sent onto the ring each time one of the programmable power supplies, relays or point motors needs reconfiguration. Requests to change a ring setting could originate from any of the various Engineers, Points and Power controllers and the Dispatcher receives these requests in the form of commands like "throw points number 5 to 'set'".

The Dispatcher is the type of process which sometimes sits idle for a long period of time (seconds) and then suddenly becomes busy when several ring commands need to be sent simultaneously. If the Dispatcher simply relayed such a burst of commands sequentially, the Port Driver (who's task is relatively complex) may be inundated with work which in turn could bring the train controller software to its knees. So the Dispatcher needs to do something to avoid causing this problem.

To prevent overloading the Port Driver, the Dispatcher uses the Timekeeper to help it. When a ring configuration request is received by the Dispatcher, it starts a timer and waits for a while before sending the ring command to the Port Driver. If any other requests arrive at the Dispatcher before the timer has expired, it can amalgamate all of the requests and ultimately send just a single Port Driver message which will trigger just a single ring update. Even though this introduces a small delay, the Engineers, Points Processes and Power Controllers won't really notice if their instructions sometimes take a quarter of a second longer to reach the hardware and for the minor price of this small delay, the Port Driver will never be swamped.

Normally, you do not need to think about changing the Port Driver, Status Scanner and Dispatcher. But if you modify the structure of the ring or modify the way the PC communicates with the train controller cards, it is mainly these three processes you'll need to adapt.

The Timekeeper

The next process in train.c is the Timekeeper Process which we've already mentioned. The need for a programme to have a Timekeeper is an aspect which differentiates "real time" software from any other software systems but despite their critical importance, timekeeper tasks in most cases are straightforward to read and understand. Timers are used for all sorts of things within train.c: to remind the Status Scanner that it's time to poll the block detectors again, to help the Dispatcher pace the rate at which it sends commands to the ring, to help the Power Management processes simulate locomotive inertia correctly and so on.

The Timekeeper in train.c works something like an alarm clock. Any task in the train.c programme can ask the Timekeeper to start a timer for it and there is no practical limit to the number of simultaneous timers that any task can activate. Timers could be as short as a hundred milliseconds or as long as hours (but I've not thought of a use for any hour long timers yet!).

In order to start a timer, a task only needs to send a "start timer" message to the Timekeeper describing the interval. After the interval has expired, the Timekeeper returns a message to the originating task identifying which timer expired. If the originating task was sleeping, waiting to receive a message, then the operating system will "wake it up" again and it can continue. If necessary, a task can also cancel or restart a timer that it had previously started and you can probably guess timer cancellation or restarting is handled with different sorts of messages.

In a real time control system, if the Timekeeper is poorly written or if the operating system is not fast or clever enough to prioritise tasks adequately, you could witness some spectacular faults. Crashes on a model train set might be amusing but mistiming in complex pieces of massive industrial machinery or safety critical systems may not be so benign. For this reason, programmers and system designers need to be particularly attentive and knowledgeable when understanding timekeeper requirements and be ruthlessly methodical when testing.

To ensure that the Timekeeper is never blocked by another less important task, it is assigned the highest absolute priority of all the tasks in train.c. No matter what other activities are going on in the system at any time, after each occurrence of the Timekeeper's clock tick, it will "preempt" all lower priority tasks and be immediately processed. But the Timekeeper is also very quick to execute, so although it will interject and temporarily stop any other activity that's going on, it is finished with its business quickly so that things can keep on going as normal.

The Points and Power Supply Processes

In the birds eye view of the software, the function I described as Infrastructure Management encompassed the Points, Power and Controller Processes. In reality, there could be multiple instances of each of these running at the same time and the actual number of tasks which are launched when you first start train.c depends on how many train controller cards there are.

This is an important concept in multitasking software. When you need to control many similar objects independently, such as the different power supplies on the train controller card or different trains moving about the layout, you only need to write the generic algorithm once but you can "launch" the same code multiple times, once for each object and each as an independent task. You can think of this a little bit like road traffic: all drivers follow the same road rules and control their steering, acceleration and braking in largely the same way but they're all going to different places and they're all at various different stages of their journeys. In the same spirit, each instance of a duplicated task will follow the same "rules" (programming steps and logic) but at all times, each instance will be

independent from the others and could be running a completely different part of the overall sequence of instructions.

In a train set which has two train controller cards, there will be four train power supplies, two points supplies and up to three different locomotives. The train controller software only contains one copy of the Power Process routine, one copy of the Points Process routine and one copy of the Engineer Process routine. But by launching as many different instances of the same code as required (creating multiple tasks from the one copy of the routine), train.c can scale to work with many different controller card and locomotive combinations.

When the Points Process receives a message to throw a particular set of points, it follows a strict sequence of events. Firstly, it sends a ring command to the Dispatcher to activate the right solenoid. After a short time, the Points process removes the current from the solenoid by issuing a second ring command. Finally, the Points Process waits for the monster capacitor (C16) to recharge and only then can it check to see if another points request message has arrived and process another set of points.

Although the time it takes to throw a set of points and to recharge the monster capacitor is short, when there are several sets of points in series, it can take several seconds to arrange all points correctly. If two or more Engineers simultaneously try to arrange their respective points, the Engineer which sent its requests last may need to wait a while before its request is processed and if its train is moving quickly, the delay might lead to an interesting crash.

In order to reduce the likelihood of such a mishap, when the Points Process receives a command to change a set of points, it first checks to see if those points are already in the required state. If they are, the Points Process will simply disregard the message immediately rather than spending the time to throw them again. On average, this tends to save a lot of time and reduces the noise of points being unnecessarily thrown. But even so, this trick is not foolproof and on occasion, I’ve been able to broadside trains on my layout for want of arranging the points more quickly.

The sole purpose of the Power Processes is to make the life for the Engineers simpler by taking over the responsibility for reprogramming power supply voltages. Each instance of the Power Process translates requests such as “Set speed to level 14” to the corresponding PWM and DC levels and writes them to the train controller cards through the Dispatcher. But perhaps the most important responsibility of the Power Processes is to simulate train inertia by gracefully ramping rather than stepping the power supply voltages.

When an Engineer sends a message to a Power Process to set a new speed, it specifies the target speed and the desired acceleration. The Power Process checks the train's current speed setting and decides whether it needs to make the train travel faster or slower in order to reach the target. From the initial settings, the Power Process steps the values of the DC and PWM supplies through a sequence which increment by increment, increases or decreases the average voltage supplied to the locomotive.

After each DC and PWM supply change, the Power Process asks the Timekeeper to wake it after an interval related to the specified acceleration and when awakened, it programmes the next step. If a faster acceleration is specified, the DC and PWM settings are stepped more quickly than for a slower acceleration.

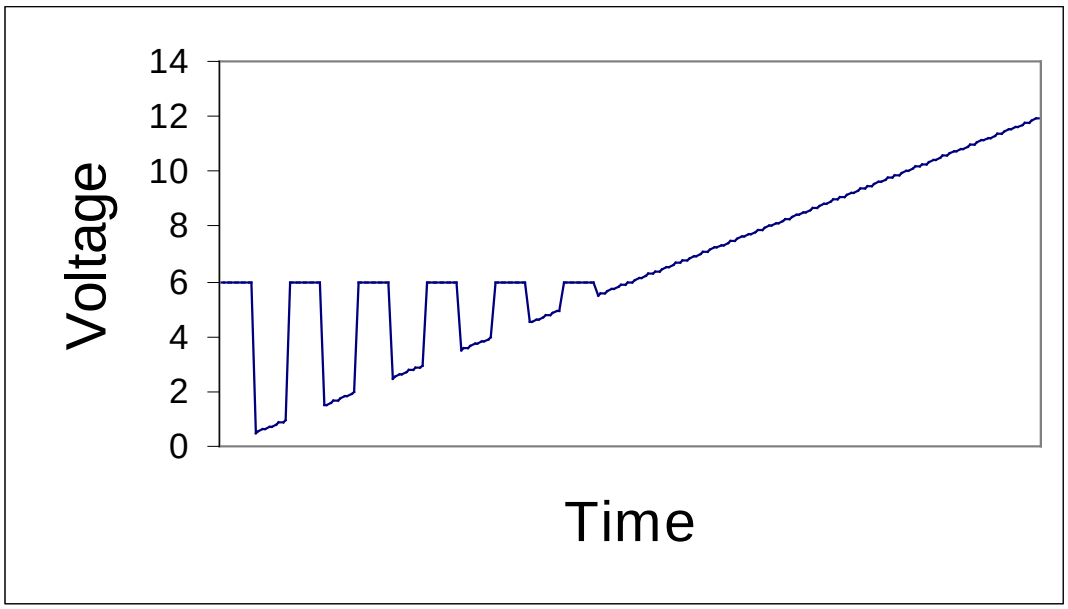

Figure 3 An example of the output voltage waveform as the speed is increased

Figure 3 shows an oversimplified example of how the output voltage might look as it changes from zero through to maximum. The diagram is oversimplified because the acceleration has been drawn much faster than it occurs in practice. In the diagram, you can see how the DC voltage convolutes with the PWM voltage to appear as pulses over a "DC floor" and because the PWM delivers only about 6 Volt to the track, the PWM becomes completely buried under the DC signal as it rises between 6 Volt and 12 Volt. Even at the fastest acceleration, the actual rate at which the Power Process updates the PWM and DC settings his much slower than implied in Figure 3 and practically, there will be many PWM pulses occurring between each setting change.

It is probably easier to imagine the track voltage in terms of its average level and the relationship between speed setting and average voltage is shown in Figure 4.

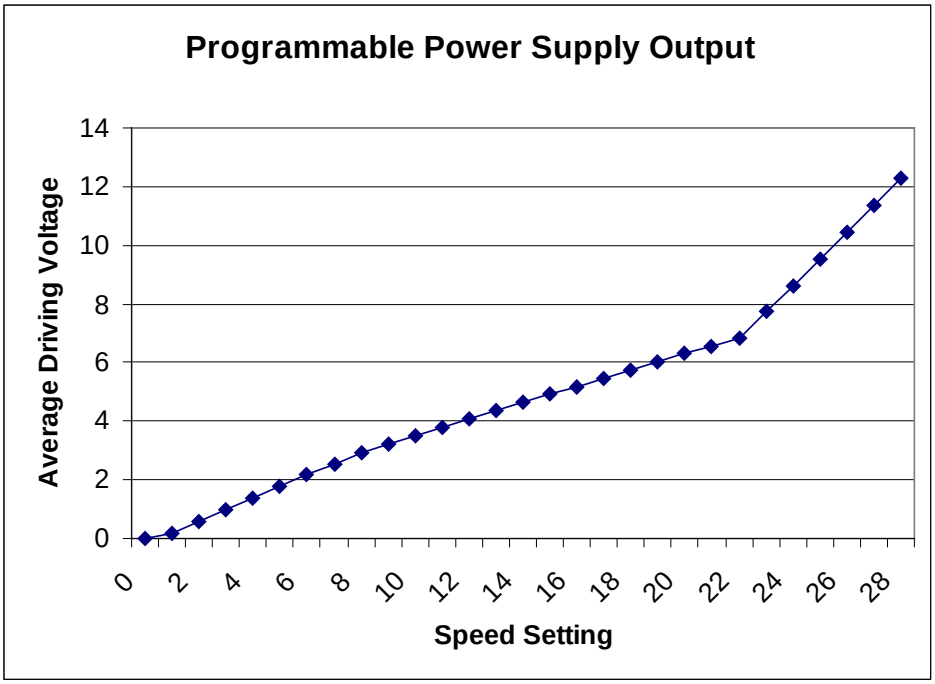

Figure 4 Configuring the PWM and DC programmable power supplies

As the Power Process ramps speed up or down, it obtains the PWM and DC settings from an array called Speed_Table[] in the file config.h. Speed_Table[] contains 29 pairs of entries representing the smallest speed [0] through to the fastest [28]. The first value in each pair is a PWM level and the second is the corresponding DC level.

It is a coincidence that I was able to derive 29 useful speed values using the PWM and DC supplies because most DCC controllers and decoders also support this number of speed steps. I have previously mentioned that the electrical design of the circuit is easy to adapt if a finer speed control is desired, but when you watch your trains accelerate through the 28 default levels, you're likely to agree that this number is more than adequate.

If you are curious, you can tinker with the settings in Speed_Table[] to change the characteristics of the inertia. You could keep the same number of entries and just change individual values, or you could add and delete entries in the table (making sure to update the MAX_SPEED symbol to reflect the new table size). Because it is virtually impossible to work out the average voltage by inspection alone, you're likely to find it very helpful to build yourself a small spreadsheet to calculate and plot the settings just as shown in Figure 4.

The Controller and Engineer Processes

The Controller Process fulfils a role which is something like that of an air traffic controller in an airport and so it is responsible for rationing access to track blocks as the different engines circulate around the layout. Continuing the airport analogy, the Engineer Processes described in Figure 2 manage one train each and behave more like aircraft pilots than drivers on trains. The Engineers use predetermined route files, like flight plans, to structure their paths, inform the Controller of the steps of the route and request permission block by block to move around the train set.

The Engineers contain all of the logic to physically move a model locomotive around the computer controlled train set in accordance with the Controller's directives. Every Engineer uses its route file to plan its next move and block by block as the trains move about the layout, the corresponding Engineer process asks the Controller for permission to move to the next track segment.

As the Controller responds to each Engineer with permission, the Engineers send messages to the Points Processes to throw sets of points and other messages to the Power Processes to set speed and direction. The Engineers also receive messages from the Status Scanner as the engines move from one block to another and in turn, they must notify the Controller that they have moved onto the next track block or left the last.

The Controller is the only task which has full view of the train set and thus it has the sole and most important responsibility of recognising potential deadlocks – situations where two or more trains will cross paths and potentially face each other off. The logic for recognising deadlock situations before they occur is non trivial and even more so is the algorithm which tries to find a way to avoid the deadlock.

Whenever two trains do plan to enter the same block, one of the trains needs to either be delayed or diverted so as to let the other train pass. The controller does this by agreeing to grant one of the deadlocked Engineers access to the block and telling the other Engineer to wait a while or to temporarily take a different path. It is the Controller's job to determine this alternate path and this is where the procedure becomes very complicated.

Because trains must have a finite length, they will occupy at least two blocks as the train passes across each block boundary. In planning alternate routes, the Controller must take the physical lengths of each train into account to ensure that the alternate route is actually feasible to completely contain the diverted train. This becomes particularly complicated when a train needs to stop and reverse along a different route than the one it came on because the Controller needs to calculate the best time to stop the train and reverse it.

There are many ways that an alternate path could be chosen. For example, the Controller could propose an alternate route which will result in the least delay, or in the least diversion from the originally proposed route, or that avoids requiring one of the Engineers to stop and reverse. Each of these approaches would lead to a different Controller algorithm. For those that would enjoy a software challenge, set yourself the task of writing your own Controller process and who knows, you may even make a significantly better job of it than I have! (And I'd be very happy to concede defeat and exchange your algorithm for mine if you propose a better approach.)

Modifying and reusing the software

If you've followed and understood this explanation, you are certainly ready to start to think about modifying train.c. However, understanding the structure of the software is by no means a prerequisite for continuing. Confused or enlightened, please don't be discouraged by the complexity: I'm sure that the majority of you can come to complete grips with it if you take it a piece at a time - and after all, piece by piece is exactly how multitasking programmes are themselves structured.

I've made all of the downloadable software available to you under the Free Software Foundation's "GNU General Public License" (GPL). This license broadly allows free distribution, modification of and reuse of the source code in your own projects but there are also some important conditions. I

encourage you all to read and understand these conditions but particularly if you're thinking of commercial applications for even a part of the code (or a modifications of part), you should carefully read the GPL and refer to http://www.gnu.org/copyleft/gpl.html. It is not my intention for anybody to use any of this code for private and closed profit.

Le Finis

At last, we have reached the terminus for this series of articles describing the design and construction steps I took to build my computer controlled train set. If you've come this far with me, I hope that you've discovered new ideas and rekindled or strengthened your interest in the hobby of model trains.

When I first 'caught the bug' which was to drive me to design my own train set controller, I had no idea I was starting a project that would keep me occupied for years. Then when I first decided to write about it, I'd no idea that it would take me nine articles before I'd feel satisfied that I'd told you enough so that you could reproduce it yourselves. I guess that I'd better not get into the forecasting business!

Even if you haven't decided to attempt the electronics, I trust that the series has been engaging in other ways and that you've learned a thing or two along the way. Who knows, perhaps one day you'll consider using your home computer for something non traditional such as a control application. Congratulations on reaching the end of this journey and fare well! It's been a long route we've travelled together.

Stefan Keller-Tuberg BE (Hons) MEM

Stefan Keller-Tuberg is a professional engineer with an honours degree in Electrical Engineering from the University of New South Wales and a masters degree in Engineering Management from the University of Technology, Sydney. He and his family are presently living in the United States where Stefan works in the telecommunications industry planning service architectures and product evolutions for high speed xDSL and Optical Internet access products. They are enjoying life in the USA but are looking forward to their return to Australia some time in the coming year.