A Computer Driven Train Controller
Article 7 – Installation and Operation of the Control Software

This is the seventh article in a series which describes the way I have built a train set which can be completely controlled from a computer. In the previous articles, the theory, the circuit and construction and testing have been presented and this time, I'll describe the steps so that you can configure the software to work for your particular train layout topology. In the ninth and final article, we'll examine the operation of the train controller software in a lot more detail.

As I have indicated in previous articles, I have chosen to implement my software under the Linux operating system. For me, the decision was straightforward because Linux is by far the least expensive and the most comprehensive platform available for PC development applications. The recent releases are very stable, the compilers and tools are excellent and it is freely available to all - as no doubt many of you have found for yourselves already.

Planning your layout

The very first configuration task which lies before you is to design your layout and to plan your points and electrical blocks. I assume that you've already done this but if not, you will be surprised at how much time you will spend drawing and redrawing your ideas with ever greater and greater accuracy - and then changing your mind some time later when you get a new idea. The best advice I could give would be to find a book about layout planning and read it cover to cover. Then talk to others about your ideas. Only when you have completed designing your layout should you come back to the example train controller software and think about starting to configure it. Better still, build your layout first, then configure your software.

With the layout design finalised and an accurate drawing of it in hand, number each of the track blocks and sets of points consecutively starting at 1 as shown in Figure 1. This diagram shows only a simple layout and a single controller card is sufficient to manage the entire example layout on its own.

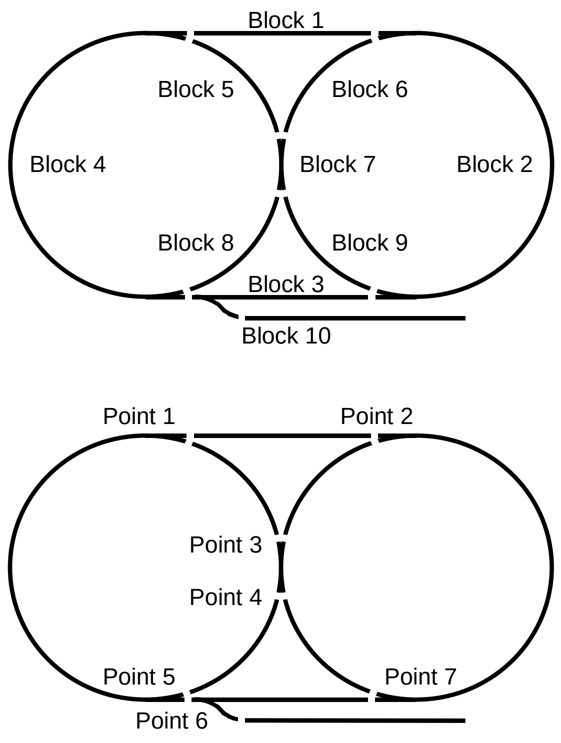

Figure 1 A simple layout showing numbered blocks and points

If you will have more than one train controller card, think about a sensible numeric allocation for track blocks and point numbers: there are three objectives to keep in mind.

The first of the train controller cards in the system will control point numbers 1 to 10 and block numbers 1 to 10. The second card will control points 11 to 20 and blocks 11 to 20 and so on. It doesn’t really matter which train controller card manages which set of points so your first objective should be to minimise the overall length of the point power bus wiring. Locate point numbers 1 to 10 as close as possible to train controller card number 1, point numbers 11 to 20 as close as possible to train controller card number 2 and so on.

For the block supplies, the main objective is to group electrical blocks together on the basis of common routes with a secondary objective to keep the blocks in proximity to the associated train controller card. Grouping blocks on the basis of routes will reduce the number of "handovers" from one power supply to another and will increase the number of trains you can practically run on your layout. Unless you have a huge layout, the length of block wiring does not need to be unreasonably minimised - particularly if you use appropriate heavy gauge wire for the longer runs.

Many hobbyists will find that they are going to have a few "spare" point or block controllers because their particular layout does not have an even multiple of 10 sets of points or an even multiple of 10 sets of blocks. If this applies to your layout, then (at your discretion) you can choose to have some gaps in your allocation of block and point number sequences. In this case, choose a numbering plan which makes best sense for your layout to achieve the objectives described above. But if I were you, and I realised that I was going to have some spare blocks, I’d consider taking some of the longer blocks or more trafficked blocks and dividing them into two or three - you’ll be able to have better control of your trains this way.

Once you’ve decided the numbering for track blocks and points, you can start to configure the train controller software. But as I mentioned above, it is still best to wait until construction of your layout has progressed to the stage that you know that what you designed is exactly what you built.

Configuring the source code to work with your particular layout

If you have not already obtained the archive of the example software for this project from ftp://xxx.xxx.xxx/xxx/train.tgz and extracted the files as described in the last article, now is the time to do so. There are four main files within the archive which contain the source code for the train controller software: train.c, train.h, controllerN.h and config.h.

Before you jump in at the deep end and start to modify the configuration files, make yourself a backup of all of the original files. Next month, when we go step by step through the compiling and testing process, we’ll firstly compile the original code to make sure that it works.

When you look at all the files you’ve extracted, you will see three called controller1.h, controller2.h and controller3.h. The first configuration task is to choose the correct controllerN.h file and make a symbolic link from it to a dummy file called "controller.h". Depending on the number of train controller cards you plan to have, you will need to choose the right one - for example, if you have two train controller cards, you will create the link by typing the Linux command:

```
ln -s controller2.h controller.h
```

If you plan to have more than three train controller cards or if you vary the electrical design of the ring, you’ll need to look inside the various controllerN.h files, see the pattern in how they differ and edit yourself a new controllerN.h file which reflects your layout - but for most of you, this will not be necessary.

The rest of the task of configuration centres around the config.h file. Using one of the many text editors available under Linux, edit config.h and start reading from the top.

For Linux novices unfamiliar with the numerous Linux editors, I suggest that you try the more intuitive application called "Notepad" under the Windows operating system or perhaps a word processor such as "Word". There is no problem in editing any of the files under Windows and copying the edited files over to Linux because the compiler doesn’t fuss over the differences between Linux’s and DOS’s CR/LF formats.

If you don’t have any common disk partitions which both operating systems can access, use a floppy disk to transfer from Linux to Windows (and back) using the Linux command "mcopy" as you would use the MSDOS command "copy". (Try the Linux command "man mcopy" if you’re having trouble.) But in the end, you’re going to want to know how to use a Linux editor and so it is probably best to choose one and start to learn about it.

Looking now at config.h, you will probably not need to make any changes to the first section of parameters. It looks as follows:

```
/* #define BASIC_DEBUG            set to display basic debug messages */
/* #define MESSAGE_TRACE          debug msgs (not always printed in order)*/
/* #define SEM_TRACE              set to debug semaphores */
/* #define SHM_TRACE              set to debug shared memory */
/* #define PORT_DEBUG             set to debug port controller */
/* #define STATUS_SCAN_DEBUG      set to debug status scanning process */
/* #define TIMER_DEBUG            set to debug timekeeper process */
/* #define DISPATCHER_DEBUG       set to debug dispatcher process */
/* #define POINTS_DEBUG           set to debug the points process */
/* #define POWER_DEBUG            set to debug the power process */
/* #define SIG_HANDLER_DEBUG      set to debug signal handlers */
/* #define CONTROLLER_DEBUG       set to debug the controller process */
/* #define ENGINEER_DEBUG         set to debug the engineer process */
#define PARANOID               /* set to enable pedantic error checking */
```

This section consists of a block of mainly commented debugging flags. (A comment in the C language is enclosed in ’/*’ and ’*/’ tokens). If for any reason in the future you find that some part of the train controller software is not working properly (for example you modified and broke the code), any or all of these debugging flags can be turned on (by removing the comment tokens) to help you find out what is going wrong. But for the time being, it is best to leave them exactly as they are.

The PARANOID flag is particularly useful at this stage because it turns on run time sanity checking which will among other things, verify the internal consistency and validity of the information you are about to specify and change in config.h. Only once you are sure that the software works with your configuration, you should recompile it with the PARANOID flag turned off.

Just below the debugging flags you will find a special debugging flag:

```
#define SIMULATE_PORT          /* set if controllers not connected in ring */
```

The SIMULATE_PORT flag can be used to let you run the train controller software on a PC which is not connected to any train set controllers. If you ever decide to modify the train controller software, you may find this flag useful while you are trying to debug your new code. But normally, this should be commented out because normally, you’ll have your train set connected to your PC, right?

A little further on in config.h, you’ll encounter a pair of definitions which look like:

```
#define CHANGE_PERSISTENCE   2  /* # additional reads before recognising chng */
#define KEEPALIVE_INTERVAL   10 /* seconds. To refresh hardware ring reset */
```

The first of these specifies how many times the train controller software needs to consecutively scan a change on any of the input ring bits before it recognises the change - this is the "debouncing" setting and the correct value will depend upon your layout. If your layout is "clean" and smooth and the engines make good contact with the tracks at all times, a value of 2 for the CHANGE_PERSISTENCE is probably conservative (but OK). If your trains arc a lot as they travel around, you may find that there are some false triggers from time to time which causes the controller programme to complain about trains disappearing from blocks or being on a block that they shouldn’t be on. If so, you should try to clean or replace your track and locomotive wheels but to fix the problem in software, increase the number from 2 to 3 or 4 or more. For the time being, leave this at 2 and come back to it if you have problems.

The second definition is the "keep alive interval" that ensures that the reset function for the point and relay driver ICs is kept from tripping when the train controller is running. Normally, a value of 10 will be fine (using the components specified in the suggested circuit) but if you feel need to change the value of the timeout, because your trains suddenly do crazy things like change direction or speed, this is the place to do it.

Now you are about to configure the tricky bit. The next (and last) section of config.h is where you must declare a "software description" of your layout design. Because we are using the ’C’ language, this software description must be written in a way that the ’C’ compiler can understand. As an example, we will study the details of how you would configure the layout drawn in Figure 1 but you will definitely need to draw your own layout diagram, invent a numbering scheme for your blocks and points and then create a similar configuration structure of your own.

OK. Take a deep breath and scan a little further down the file. The first parameter is not so tricky.

```
#define MAX_BLOCK_LENGTH 120        /* used in cross checking */
```

The train controller software needs to know the "length" of each of the track blocks. It uses this information to help it decide how to manage situations where trains are reversing or need to be diverted or delayed to avoid a collision. The specification for the actual measured lengths of each block will be discussed below but the constant MAX_BLOCK_LENGTH is used during sanity checking of the real information.

For the time being, identify which of your electrical blocks has the longest run and estimate its length in a relatively fine unit (like centimetres) and then add a little. If your longest block is approximately 100 centimetres for example, specify MAX_BLOCK_LENGTH to be 120 (no fractional part). Notice that there are no units for the length measurement and you could choose centimetres, millimetres, tenths or sixteenths of an inch or maybe even scale metres or scale feet.

Next comes the layout specification itself. This is defined in a ’C’ array called Block[] and it contains a number of sections called structures. There is one structure for each track block and it has the format:

```
{
  "Any text name you care to choose for this block",
  {
    {description of the path from block centre to boundary of the first exit},
    {description of the path from block centre to boundary of the second exit},
      …
    {description of the path from block centre to boundary of the last exit}
  }
},
```

You will notice that there are many sets of curly braces in this structure and you’ll need to edit carefully to make sure that you don’t inadvertently end up with a few too many or a few missing. Commas are important too and will cause the ’C’ compiler to complain if you add or delete one accidentally.

If a particular track block has more than one exit, the order in which you describe the exits is not important, you can choose any order you like. But each description is itself a ’C’ structure and has this format:

```
{Number of the adjacent segment,
  Length from the centre of this block to the adjacent segment exit boundary,
    Power supply polarity needed to reach the adjacent block from this block,
      {list of each point number and setting from centre to boundary}
},
```

All right! Settle down! These structures within structures are getting confusing. Let’s just go straight to the example layout and look at its first entry. (Refer to Figure 1 for a diagram of this example layout)

```
BLOCK_STRUCT Block[] =
{
  {
    "Block 1",
    {
      {2,   40, NORMAL_POLARITY,   {BOUNDARY}},
      {4,   40, OPPOSITE_POLARITY, {BOUNDARY}}
    }
  },
```

The name of this block is rather boring. Just as in Figure 1, it's called "Block 1" but you could just as easily choose something more elaborate like "Marylebone Station" or "Everley Street Shops". There are also two exits from this block, an exit to block number 2 and another to block number 4. To get from block 1 to block 2, you set the direction relay to NORMAL_POLARITY and to get from block 1 to block 4, you set the direction relay to OPPOSITE_POLARITY.

The choice of whether you specify NORMAL or OPPOSITE polarities is entirely up to you. If you connect the two block wires one way, the polarity needed to reach a particular exit will be NORMAL and if you connect the wires the other way, the polarity will be OPPOSITE. In all likelihood, when you wire your layout to your train controller cards, you'll forget which polarity is which and get a few of the entries in config.h wrong. So the first time you run your trains, you'll encounter some "bugs" and trains will head off in unexpected directions or will suddenly trip the controller's overcurrent protection as the locomotive moves from one block onto the next.

By taking note about which block boundaries caused mishap, you can go back and correct config.h section by section. But for the time being, don't worry too much.

The length from block 1 to either block 2 or to block 4 is 40 arbitrary units. This does not mean that block 1 is 40 units long - it is actually 80 units. The trick is that from the point of view of the software, the length is measured from the 'nominal centre' towards the block boundary rather than from end to end.

The 'centre' of the block does not need to be the 'middle' of the block even though it is in the example above.

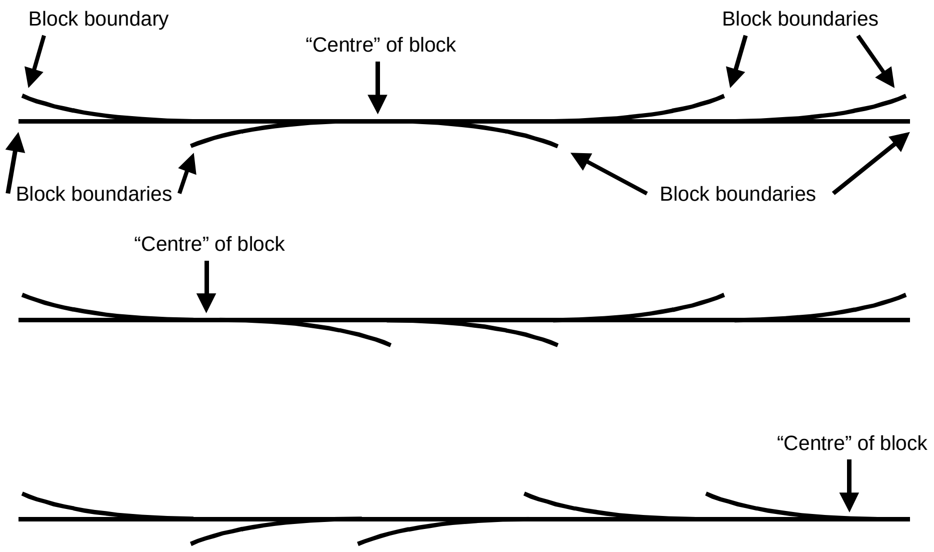

Figure 2 The "centre" of a block doesn't need to be at the middle

The 'centre' of a block, as shown in Figure 2, is that place within the block from which you can move to any of that block's exits without needing to stop and change directions. Another way of saying this is that the centre is the place, where no matter what direction you look, all sets of points within the block diverge. The block boundary is the location of the electrical gap which separates one block from the adjacent block.

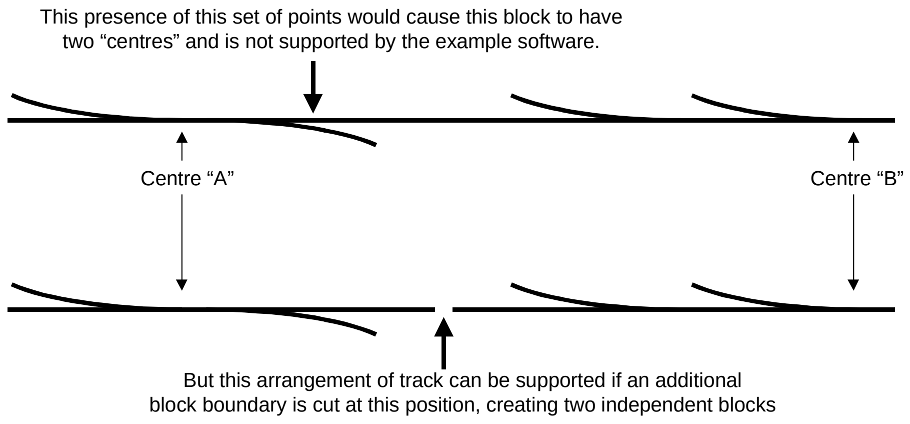
This presence of this set of points would cause this block to have two “centres” and is not supported by the example software.
Centre “A”
Centre “B”
But this arrangement of track can be supported if an additional block boundary is cut at this position, creating two independent blocks

Figure 3 A block can’t have "two" centres

But in Figure 3, a track arrangement is shown which could lead to ambiguity about where the real "centre" of the block is located. In general, the example train controller software only supports block arrangements where, if you start at the logical centre, you can progress to any of the boundaries without needing to stop and change direction.

If you find that your layout will contain an arrangement which has two "centres", you will definitely need to add an electrical gap and make two blocks of it. Alternately, if you don’t like this limitation you can modify the example train controller software or write your own! But for the life of me, I can’t think about how you could do this without introducing considerable software complexity.

There are some other fields in the description of Block 1 which are more easily explained by first looking at the description of Block 2.

```
{
  "Block 2",
  {
    {3,  75, NORMAL_POLARITY,   {{7,   OPEN}, BOUNDARY}},
    {9,  72, NORMAL_POLARITY,   {{7, CLOSED}, BOUNDARY}},
    {1,  75, OPPOSITE_POLARITY, {{2,   OPEN}, BOUNDARY}},
    {6,  72, OPPOSITE_POLARITY, {{2, CLOSED}, BOUNDARY}}
  }
},
```

Looking at Figure 1 and the structure description, you can see that Block 2 connects to four other blocks: Block 3, Block 9, Block 1 and Block 6. From the perspective of block 2, you reach blocks 3 and 9 by setting the polarity to NORMAL and you reach blocks 1 and 6 by setting it to OPPOSITE. The nominal distances from the ’block centre’ to each of the boundaries are 75 or 72 centimetres respectively.

Block 2 is different from Block 1 because it contains two sets of points, one set on either side of the block, and you can see how these points are described by the curly bracket enclosed list at the end of each line. For the first line of the Block 2 structure, which describes the exit from Block 2 to Block 3, you will notice that point number 7 needs to be set to the OPEN state in order to reach Block 3 from the nominal centre. Alternately, to reach Block 9 (the second line), you need to set point number 7 to CLOSED. Likewise in the other direction, Block 2 connects to Block 1 by setting point number 2 to OPEN and it connects to block 6 by setting point number 2 to CLOSED.

Just as before, it is up to you how you choose to use OPEN and CLOSED. For the time being, choose one combination and wire your layout which ever way seems easiest. Then, when it comes to running your trains for the first time, "debug" your configuration by observing where your trains turn the wrong way. You can edit and correct the definitions in train.h when you recognise the "bugs".

That wasn't too complicated, was it? Yes OK. Perhaps it was. Go over it once again and maybe it will become a little clearer.

When you've understood the configuration of the first two blocks, the rest of the Block[] structure will be a little more intuitive. Let's now look at the remainder of the description for this layout.

```
  {
    "Block 3",
    {
      {4,   5, NORMAL_POLARITY,   {{6,   OPEN}, BOUNDARY}},
      {2,  75, OPPOSITE_POLARITY, {            BOUNDARY}},
      {10, 10, OPPOSITE_POLARITY, {{6, CLOSED}, BOUNDARY}}
    }
  },

  {
    "Block 4",
    {
      {1,  75, NORMAL_POLARITY,   {{1,   OPEN}, BOUNDARY}},
      {5,  72, NORMAL_POLARITY,   {{1, CLOSED}, BOUNDARY}},
      {3,  75, OPPOSITE_POLARITY, {{5,   OPEN}, BOUNDARY}},
      {8,  72, OPPOSITE_POLARITY, {{5, CLOSED}, BOUNDARY}}
    }
  },

  {
    "Block 5",
    {
      {4,  40, NORMAL_POLARITY,   {BOUNDARY}},
      {7,  40, OPPOSITE_POLARITY, {BOUNDARY}}
    }
  },

  {
    "Block 6",
    {
      {2,  40, NORMAL_POLARITY,   {BOUNDARY}},
      {7,  40, OPPOSITE_POLARITY, {BOUNDARY}}
    }
  },

  {
    "Block 7",
    {
      {5,  15, NORMAL_POLARITY,   {{3,   OPEN}, BOUNDARY}},
      {6,  15, NORMAL_POLARITY,   {{3, CLOSED}, BOUNDARY}},
      {8,  15, OPPOSITE_POLARITY, {{4,   OPEN}, BOUNDARY}},
      {9,  15, OPPOSITE_POLARITY, {{4, CLOSED}, BOUNDARY}}
    }
  },

  {
    "Block 8",
    {
      {7,  40, NORMAL_POLARITY,   {BOUNDARY}},
      {4,  40, OPPOSITE_POLARITY, {BOUNDARY}}
    }
  },

  {
    "Block 9",
    {
      {7,  40, NORMAL_POLARITY,   {BOUNDARY}},
      {2,  40, OPPOSITE_POLARITY, {BOUNDARY}}
    }
  },

  {
    "Block 10",
    {
      {3, 100, NORMAL_POLARITY,   {BOUNDARY}},
    }
  },
} ;
```

I'll leave it to you to go through this array of structures to satisfy yourselves that it corresponds with the layout shown in Figure 1.

There are just a couple more points to note before we wind up this article for the month however.

In the Figure 1 layout, all of the blocks except for Block 10 have at least one exit on either side. Block 10 though is different because it is only connected to one other block, Block 3. At the other end of Block 10, the track finishes in some "buffers" (which is the technical name for the stack of wood, sand or whatever that the rail company has hopefully placed at the end of the line to stop its trucks rolling off into somebody's back yard). In this example, the length of the block is 100 centimetres which is in fact the total length from end to end. The absence of a definition of an exit with one or the other polarities tells the train control software that there is no exit at that side of the block.

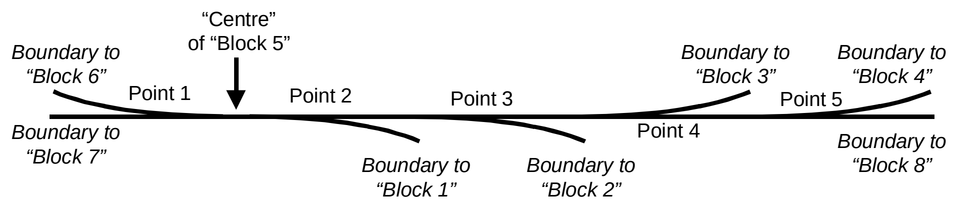

Figure 4 A more complicated block boundary example

If you design yourself a more complicated layout, you are bound to want to create a block which has more than one set of points at one or the other end of it. An example of such a more complicated block is shown in Figure 4. The 'C' structure describing this arrangement would look something like this:

```
{ /* This is BLOCK 5 and these are all its exits */
  "Main Line Interchange",
  {
    {1, 35, NORMAL_POLARITY,   {{2, CLOSED}, BOUNDARY}},
    {2, 50, NORMAL_POLARITY,   {{2, OPEN}, {3, CLOSED}, BOUNDARY}},
    {3, 65, NORMAL_POLARITY,   {{2, OPEN}, {3, OPEN}, {4, CLOSED}, BOUNDARY}},
    {4, 83, NORMAL_POLARITY,   {{2, OPEN}, {3, OPEN}, {4, OPEN}, {5, CLOSED}, BOUNDARY}},
    {8, 85, NORMAL_POLARITY,   {{2, OPEN}, {3, OPEN}, {4, OPEN}, {5, OPEN}, BOUNDARY}},
    {6, 23, OPPOSITE_POLARITY, {{1, CLOSED}, BOUNDARY}},
    {7, 25, OPPOSITE_POLARITY, {{1, OPEN}, BOUNDARY}}
  }
},
```

You can see that when more than one set of points are cascaded to reach a particular exit, they are simply listed in order. Theoretically, you can cascade as many points as you like and list them all on the same line. Practically, you'll rarely need more than a few in series. The software includes a compile time definition of the maximum number of cascaded points per block in controllerN.h in the symbol MAX_CASCADED_POINTS. If you get compile time errors or warnings relating to excess elements in an array initialiser, this is likely to be your problem and you can correct it by increasing the default.

If you have decided to attempt construction but do not feel confident that you have completely understood the software concepts described in this article, you should speak with another hobbyist who is more familiar with the ’C’ language and have them help you through your questions. While the example train controller software should certainly be capable of getting you started in controlling any arbitrarily shaped layout (undiscovered bugs not withstanding), it is important for you to design electrical block boundaries in compatible locations and equally important that you be able to express your layout design in the ’C’ language as I have described. It really would be best that you have your questions answered before you commence nailing or gluing track to timber and cork.

Alas, we have reached the end of the seventh article in this series. For those of you who have decided to attempt this project, you should now have enough basic information to design a layout of your own, reproduce the train controller circuit and to configure the example train controller software to work with your design.

In the next article, we’ll take a look at how to compile and use the example train controller software and I’ll provide a couple of tips to improve the operation of your train set. In the final article in the series, we’ll look at the innards of the train programme and at how a multitasking control application is structured. Until then, happy configuring!

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.