Chapter 2: Introduction to Software

www.jlcenterprises.net

©2013 Bruce A. Chubb
Chapter 2

Introduction to Software

Software refers to the instructions placed in a computer’s memory to tell the computer what action to take. These instructions can be mathematical or logical. A group of instructions is referred to as a program, and a program loaded into a computer defines, step-by-step, what kind of a machine the computer will be.

When we load a check-balancing program, the computer helps us to balance the checkbook. Load a space-war program and the same computer becomes a space-war game. I will be showing interface programs you can use to communicate with your external hardware using C/MRI. These programs will include test software to make sure your system is operating correctly, as well as sample application programs.

The software examples presented in this C/MRI User’s Manual and in the Railroader’s C/MRI Applications Handbook are written with a primary emphasis of making them easy to understand and apply. After you have worked a little with the many examples provided, you will find that tailoring them to fit your specific railroad becomes a quite straightforward exercise.

Reading and writing program statements can be much like using the English language. For example:

IF BLOCK(15) = OCCUPIED THEN SIGNAL(24) = RED

It is straightforward to write and to understand: basically, if Block 15 is occupied then Signal 24 is set to red. It would be difficult to make this any simpler. To handle the green case, for 2-aspect signaling, add the statement:

IF BLOCK(15) = CLEAR THEN SIGNAL(24) = GREEN

Alternatively, the red and green cases can be handled together using the statement:

IF BLOCK(15) = OCCUPIED THEN SIGNAL(24) = RED ELSE SIGNAL(24) = GREEN

This simply adds the “(or) else if Signal 24 is not red then it is green” statement to cover the situation when Block 15 is clear. It presumes that if Block (15) is not occupied then it must be clear. Once you get the hang of it, understanding and writing such statements becomes a straightforward process and actually is fun to accomplish.

Feedback from hundreds of C/MRI user’s expresses such sentiments as: “I never dreamed programming could be such fun. Every time I add a few lines of code I have more functions running on the railroad. It’s great!”

SOFTWARE REAL-TIME LOOP

Essentially independent of the details making up your C/MRI applications and also of the programming language used, you can be assured that most C/MRI programs will follow the fundamental format illustrated in Fig. 2-1.
Fig. 2-1. Software real-time loop – simplified

Each program starts by performing application initialization, such as defining the types of signals being used and the type of interface hardware connection implemented. Once this initialization is complete, the computer then enters a continuous loop to perform the following steps:

1. Read inputs from the railroad
2. Calculate railroad outputs based upon the inputs
3. Write outputs to the railroad
4. Go back to step 1

This 4-step loop is repeated ad infinitum until you exit the program. It typically executes between 10 and 1000 times a second. At these speeds the computer software is essentially responding in real-time to events on the railroad. As an example, when the computer senses a train has crossed a signal block boundary, this results in the computer changing a signal to red. Because the computer software is forever rapidly repeating itself in a loop, while responding to real-time events on the railroad, the process is frequently referred to as a tight-infinite loop or as a real-time loop. In all that follows I will use the latter name.

Example inputs in signaling applications include reading the status of block occupation detectors used to determine train position, turnout position to know what tracks are aligned, and control panel inputs such as may be found on a dispatcher panel. Example outputs include setting trackside signal aspects, aligning turnouts and setting display LEDs on control panels. The computer software uses the continuously
updated inputs on each repetition, or iteration, of the loop, to re-calculate the changes in outputs and then it writes the updated outputs to the railroad.

**BASIC SOFTWARE EXAMPLES**

At this point I will focus on the logic portion of the program as illustrated by the central box in Fig. 2-1. It is the more creative part of programming. It is where we take the railroad’s inputs and decide, or calculate, what should be the corresponding outputs. How the inputs are read from the railroad and how the outputs are written to the railroad varies a little, based upon the interface hardware, e.g. SMINI, or SUSIC, so I will cover it later when we get more into hardware configurations. Examples of the logic we might need to program are:

- Train approaching grade crossing so start flashers, bell and lower gates
- Pushbutton pressed for yard Track 12 so align all switch motors for Track 12
- Train just passed Signal 34E so set the signal to red
- Block now clear so set signals leading into block to green
- Dispatcher cleared traffic movement out of siding so align turnout for siding, then after a delay, set corresponding dwarf signal to green
- Siding turnout now occupied so prevent it from being thrown
- Train just passed dwarf signal so set it to red

To begin to understand how easy it is to carry out such functions using C/MRI software, let’s look at a few examples.

**Signaling a Loop of Track**

Fig. 2-2 shows a loop of track divided into six signaled blocks labeled BK(1) through BK(6). Each block incorporates an occupation detector, such as a JLC provided OD or DCCOD, to inform the computer which blocks are occupied and which are clear. A signal is located at the end of each block and driven by the computer. The signal informs the train crew if the block ahead is occupied. I have labeled signals in the eastbound direction SE(1) through SE(6) and for the westbound direction SW(1) through SW(6).

![Fig. 2-2. Loop track with signals](image-url)

For this first example, I will start very simply by assuming 2-aspect, red and green, color light signaling. In later examples, and especially in the **Railroader’s C/MRI Applications Handbook**, I will look at
much more complete and realistic signaling applications including implementing full ABS, APB, CTC and interlocking plants with all types of multi-aspect signaling. The C/MRI can do it all, more easily, more prototypically and at less cost than any other signaling method.

However, we need to start someplace and I have chosen Fig. 2-2. Once we generate the logic for handling this simple signaling setup, it becomes a solid baseline from which to build more involved examples. Basically, we need to understand the basics before moving forward. There are many ways to program signal logic. At this point I will lead you through one approach to handle the signaling in Fig. 2-2.

Most intermediate block signals on the prototype, like those in Fig. 2-2, display green unless the block immediately following the signal is occupied and then the signal is red. I will explain adding the yellow aspect a little later. At this point, to program the red-green situation, we will calculate the signal aspects using a two step process. First we will set every signal to red. At this stage, this initial red setting simply defines the signal aspect in the computer’s memory and does not directly affect the trackside setting. As a second step, we will look to see where blocks are clear and set the affected signals to green. Once both steps are completed, we will write the signals to the railroad.

Fig. 2-3 shows the program logic statements. For this example, I am defining a red aspect as RED and a green aspect as GRN. Also, the variable CLR is used to define a block as clear and OCC to define a block as occupied. The lines beginning with REM are remarks used to document or comment on the different sections of a program and they are not executed by the computer. I use them in all my programming to inform people reading the program of the purpose, or function, of the block of code that follows each REMark. Everything immediately following an apostrophe is also a remark.

```
REM**INITIALIZE ALL SIGNALS TO RED
FOR I = 1 TO 6
SE(I) = RED
SW(I) = RED
NEXT I

REM**CHECK IF BLOCK CLEAR THEN SET SIGNALS LEADING INTO BLOCK GREEN
IF BK(1) = CLR THEN SE(6) = GRN: SW(2) = GRN
IF BK(2) = CLR THEN SE(1) = GRN: SW(3) = GRN
IF BK(3) = CLR THEN SE(2) = GRN: SW(4) = GRN
IF BK(4) = CLR THEN SE(3) = GRN: SW(5) = GRN
IF BK(5) = CLR THEN SE(4) = GRN: SW(6) = GRN
IF BK(6) = CLR THEN SE(5) = GRN: SW(1) = GRN
```

Fig. 2-3. Signal logic programming for loop track

I have elected to use a FOR-NEXT loop to initialize the 12 signals to red. The FOR I = 1 to 6 statement sets up a program loop where the first time through the loop variable I = 1, the next time through the loop I = 2, then 3, then 4 and so forth up to the last time through the loop where I = 6. The content of the loop is all statements between the FOR statement and the NEXT I statement. Thus, the first time through the loop, SE(1) and SW(1) are set to red. The second time through the loop SE(2) and SW(2) are set to red and so on up to the last time through the loop where SE(6) and SW(6) are set to red.

Once the loop variable “I” reaches the value of 6, there is no NEXT I to increment so the program stops performing the loop and proceeds to execute the block of statements following the NEXT I statement. This next block of code simply looks at each signaled block and if clear it sets the affected signals to green. For example IF BK(1) is clear THEN SE(6) and SW(2) are set to green. However for the condition where Block1 is occupied, i.e., it is not clear, the statements after the THEN are skipped over keeping SE(6) and SW(2) at red.
Once both blocks of code are executed, the program proceeds to write the outputs to the railroad and then to branch back to read in the detector inputs again and to recalculate the signal outputs. How the reads and writes are actually handled is covered in later chapters where attention is devoted to specific node types, i.e. the SMINI and SUSIC.

**Approach Lighting Signals**

For another signal logic example, let’s assume a railroad has 50 signaled blocks and it is desired to implement approach lighting for all 100 signals. This means the two signals at each end of each block are dark unless the block in approach to the signals is occupied. Many prototypes do this to prolong bulb life, reduce battery drain during backup operations and now most importantly, to reduce vandalism. For this example, I have labeled the signal blocks with BK(I) and the corresponding signals exiting the block with SE(I) and SW(I) where the variable I, called a subscript because it is in parentheses, takes on the values 1 through 50. To implement approach lighting for all 100 signals and the 50 blocks, we add the 6 program statements listed in Fig. 2-4.

```
FOR I = 1 TO 50
  IF BK(I) = CLR THEN
    SE(I) = DRK
    SW(I) = DRK
  END IF
NEXT I
```

Fig. 2-4. Programming approach lighted signals

The FOR …NEXT statements set up a program loop, between the FOR and the NEXT that is executed 50 times. The first time through the loop the variable I = 1, the next time I = 2, then 3, 4, and so on up to 49 and finally 50. For each value of I, the IF THEN - END IF block of code is executed. For each signal block that is clear, the two statements that follow the IF are executed to set the signals at the east end and the west end of that block to dark. For each block that is occupied, i.e. not clear, the two statements setting the corresponding signals to dark are skipped over, keeping the signals at their previously calculated setting. Written in the form shown in Fig.2-4, the IF statement is referred to as a Block IF statement, which is used to head up a consecutive set of statements all relating to the Block IF condition. The END IF statement is always required to define the end of the Block IF set of statements. The following NEXT I statement defines the end of the FOR I = 1 TO 50 loop.

Fig. 2-5 illustrates adding approach lighting to our previous loop of track. We simply copy the block of code from Fig. 2-4 and paste it at the end of Fig. 2-3. We then take the pasted code and change I = 1 TO 50 to read I = 1 TO 6. That is all there is to implementing approach lighting.

It may seem inefficient that we first set all signals to red then we determine if a particular signal should be set to green and, if so, we set it to green. Then after we calculate the aspects for all the signals we go through them again and see if they should be set to dark. A key point is that we are only manipulating the signal aspect as it is stored in the computer’s memory. These internal-to-memory manipulations are having no immediate impact on the actual aspect of the trackside signal. It does not change until we have completed calculating all of our outputs, then transmitting them to the railroad via the interface.

It is possible to include additional program branching statements and reduce the number of repeated times a given aspect may be calculated. For example, if a signal is going to be dark, skip the steps to determine what the aspect would be if it was not going to be dark.
REM**INITIALIZE ALL SIGNALS TO RED
FOR I = 1 TO 6
    SE(I) = RED
    SW(I) = RED
NEXT I

REM**CHECK IF BLOCK CLEAR THEN SET SIGNALS LEADING INTO BLOCK GREEN
IF BK(1) = CLR THEN SE(6) = GRN: SW(2) = GRN
IF BK(2) = CLR THEN SE(1) = GRN: SW(3) = GRN
IF BK(3) = CLR THEN SE(2) = GRN: SW(4) = GRN
IF BK(4) = CLR THEN SE(3) = GRN: SW(5) = GRN
IF BK(5) = CLR THEN SE(4) = GRN: SW(6) = GRN
IF BK(6) = CLR THEN SE(5) = GRN: SW(1) = GRN

REM**IMPLEMENT APPROACH LIGHTING BY SETTING SIGNALS TO DARK...
REM**      ...IF BLOCK APPROACHING SIGNAL IS CLEAR
FOR I = 1 TO 6
    IF BK(I) = CLR THEN
        SE(I) = DRK
        SW(I) = DRK
    END IF
NEXT I

Fig. 2-5. Programming loop track for approach lighting

However, to take advantage of the computer’s super high processing speeds and to keep things simple it is best to proceed logically straight through every processing step each time through the real-time loop. This keeps programming simple, straightforward and easy to understand. These are the truly important precepts for effective C/MRI programming. Because our railroading applications are fairly low speed, maximum program efficiency and speed of computation are relatively unimportant. For example, the total code in Fig. 2-5 probably executes in a fraction of a microsecond in today’s computers.

Also, in general, IF-THEN branching takes longer than straight in-line calculations. My motto in all C/MRI programming is, “Keep your programming simple, straightforward and easy to understand!” Recalculating your signal aspects for several different conditions is typically much easier to understand than using a number of alternative and perhaps “nested” branching statements. The term “nesting” applies to where branching statements occur within other branching statements, which can lead to quite complex software behavior.

Three-Aspect Signaling

Let’s take another example and program the Automatic Block Signals (ABS) illustrated in Fig. 2-6 for 3-aspects, namely red, yellow and green. As before, the trackage is divided into signal blocks with an occupancy detector in each block. With ABS, and as we did in the previous example, a block signal is placed at the end of each block to inform the engineer if it is safe to proceed into the next block and if so under what conditions. Only eastbound signals are shown because westbound signals are handled in an identical manner. A train is placed in Block 5 to illustrate a typical set of aspects for the signals which are red immediately behind the train, then yellow, with the remainder of the signals being green.

Blocks and signals can be identified anyway you desire and computer software can be written accordingly. To illustrate this point, in this example only, I will use complete names for the variables rather than the more typically used abbreviations. After this example I will change back to using the shorter, i.e. abbreviated, variable names, as they require fewer keystrokes for input and more information can be fitted on a given coding line.
As before, every block and every signal needs to be numbered so they can be addressed individually by the software. We could use BLOCK1, BLOCK2, BLOCK3, etc. but it is often easier to use subscript notation, also referred to as an array. In programming these are written as BLOCK(1), BLOCK(2), BLOCK(3), etc. Although it is not mandatory, it is usually best to set up the numbering as 1, 2, 3, 4, etc. (i.e. starting with 1 and not skipping numbers).

If you are not aware of it already, you will soon observe that programs can be written in many different ways using a multitude of different approaches and languages. The software programs included on the disk provided with this manual and the additional examples presented in the Railroader's C/MRI Applications Handbook cover many railroad examples following all types of approaches and in several languages. For example, the handbook shows four different ways to program ABS signals.

This ABS programming example is set up to execute the block of statements in Fig. 2-7. It does this in a continuous loop many times a second. Therefore, as trains move from block to block, the signals change to reflect the correct aspects similarly to the prototype. To follow the prototype more exactly, we should calculate the aspect of Signal 5 ahead of Signal 4 and 4 ahead of 3 and so forth. This way each signal aspect calculation is using the latest available aspect of the other signals it needs for the calculation. You can arrange the blocks of code so that this is the case. However, at the high speed at which computers operate, the eventual results are identical. If a signal does have an incorrect aspect, such as red instead of yellow, it is corrected in the next iteration through the program, typically occurring a fraction of a second later.

![Diagram of ABS signals and blocks](image)

**Fig. 2-6.** Block signals used for ABS programming

REM**CALCULATING ASPECT FOR SIGNAL 1
SIG1: IF BLOCK(2) = OCCUPIED THEN SIGNAL(1) = RED: GOTO SIG2
    IF SIGNAL(2) = RED THEN SIGNAL(1) = YELLOW ELSE SIGNAL(1) = GREEN

REM**CALCULATING ASPECT FOR SIGNAL 2
SIG2: IF BLOCK(3) = OCCUPIED THEN SIGNAL(2) = RED: GOTO SIG3
    IF SIGNAL(3) = RED THEN SIGNAL(2) = YELLOW ELSE SIGNAL(2) = GREEN

REM**CALCULATING ASPECT FOR SIGNAL 3
SIG3: IF BLOCK(4) = OCCUPIED THEN SIGNAL(3) = RED: GOTO SIG4
    IF SIGNAL(4) = RED THEN SIGNAL(3) = YELLOW ELSE SIGNAL(3) = GREEN

REM**CALCULATING ASPECT FOR SIGNAL 4
SIG4: IF BLOCK(5) = OCCUPIED THEN SIGNAL(4) = RED: GOTO SIG5
    IF SIGNAL(5) = RED THEN SIGNAL(4) = YELLOW ELSE SIGNAL(4) = GREEN

REM**CALCULATING ASPECT FOR SIGNAL 5
SIG5:

**Fig. 2-7.** Programming statements for ABS Signals 1 through 4

Only two lines of code are required per signal. Use the same two lines for each signal on the railroad and simply change the subscript numbers. To save keying in every line, simply copy and paste the same block
of code for as many signals as on the railroad and then go back and change the numbers. To see how the program works, we only need to look at Signal 1 as all the others function identically.

The first line for Signal 1 simply reads – if Block 2 is occupied then set Signal 1 to red followed by a branch to calculate the next signal, which is Signal 2. That is pretty straightforward. If the computer finds that Block 2 is occupied and it sets Signal 1 to red there is no need to check out the other possible aspects for Signal 1. In essence, Signal 1 stays red no matter what else is true. If Block 2 is occupied, then we simply branch to process the next signal. The colon is used to separate the multiple statements placed on a given line.

For the case when Block 2 is not occupied, the statements immediately to the right of THEN are skipped and we proceed with executing the second statement for Signal 1. This reads – if Signal 2 (the next signal down the line) is red then Signal 1 is yellow else Signal 1 is green. That’s about all there is to implementing signal logic for 3-color ABS signaling.

**Turnout Control**

Fig. 2-8 shows a track plan with five staging tracks converging down to a single track main. The same arrangement could represent tracks departing from a passenger terminal, or many other possible situations. Assume that each turnout is controlled by a Tortoise machine that is either wired directly to two C/MRI output pins or to a single C/MRI output pin via a JLC provided Switch Motor Control (SMC12) card. I will cover the actual wiring connections in the next chapter. One turnout control button is used per track, labeled 1, 3, 5, 7 and 9.

The required code to align the turnouts using the software equivalent diode matrix is listed in Fig. 2-9. I have listed only the code for the upper yard, the odd number tracks, because once you see how it works it is easy to apply for any track arrangement. I am using the variable TB( ) for the inputs from the Track alignment pushButtons and the variable SM( ) for the outputs to control the Switch Motors.

Using a computer, turnout alignment is very simple regardless of the complexity of the track arrangement or type of switch machine. The computer simply reads the button pressed and uses its built-in software logic to set the appropriate turnouts. For example, look at the statement for Track 7. Rhetorically, it reads:
If Track Button 7 is pressed then Switch Motor 1 is set reversed, 2 is set reversed, 3 is set normal and 4 is set reversed.

REM** INITIALIZE TURNOUT CONTROL CONSTANTS
TBP = 1     'Turnout button pressed
TUN = 1     'Turnout normal alignment
TUR = 2     'Turnout reverse alignment

REM** SET TURNOUTS BASED UPON WHICH TRACK BUTTON IS PRESSED
IF TB(1) = TBP THEN SM(1) = TUN
IF TB(2) = TBP THEN SM(1) = TUR: SM(2) = TUN
IF TB(3) = TBP THEN SM(1) = TUR: SM(2) = TUR: SM(3) = TUR
IF TB(4) = TBP THEN SM(1) = TUR: SM(2) = TUR: SM(3) = TUN: SM(4) = TUR
IF TB(5) = TBP THEN SM(1) = TUR: SM(2) = TUR: SM(3) = TUN: SM(4) = TUN

Fig. 2-9. Turnout control programming

Note that the definitions of the turnout alignment constants, TUN and TUR, in Fig. 2-9, assume that the switch motor is directly connected to two output lines. Pull one line low, the turnout aligns ‘normal’. Pull the other line low the turnout aligns ‘reversed’. If a single output line is used to control the switchmotor, with a JLC provided SMC12 card, then redefine TUN = 0 and TUR = 1. That’s all there is to having the software control turnouts and the scheme works for any track arrangement.

For example, on the SV Oregon System, we are using the software diode matrix approach to control the turnouts in 5 different double-ended freight classification yards, 2 different passenger terminals, 2 large double-ended staging yards and an entrance-exit interlocking plant. That adds up to a total of 236 track alignment buttons controlling 203 switch motors. The result is that at every location you simply press a single track alignment button and every turnout aligns toward the requested track. You just can’t beat the software diode matrix for simplifying operations – especially at large junctions, staging areas and classification yards. Pressing one button in Eugene Yard aligns the whole ladder to the requested track.

Preventing Switch Throwing Under a Train

However, the advantages of the C/MRI for these applications do not stop here. Let’s say you want to prevent turnouts from being thrown when a staging area throat is occupied. Simply add a couple more statements as shown in Fig. 2-10. It is very straightforward. You only need look to see IF OS(14) = occupied THEN skip over the statements that read the button presses and that set the turnouts! That’s all there is to using software to prevent throwing a turnout under a train.

REM** INITIALIZE TURNOUT CONTROLLING AND BLOCK OCCUPATION CONSTANTS
TBP = 1     'Turnout button pressed
TUN = 1     'Turnout normal alignment
TUR = 2     'Turnout reverse alignment
CLR = 0     'Block clear
OCC = 1     'Block occupied

REM** SET TURNOUTS BASED UPON WHICH TRACK ALIGNMENT BUTTON IS PRESSED
REM**   ...PLUS INHIBIT MOVING SWITCH POINTS IF TURNOUT IS OCCUPIED
IF OS(14) = OCC THEN GOTO EXT 'Skip reading buttons if throat occupied
IF TB(1) = TBP THEN SM(1) = TUN
IF TB(2) = TBP THEN SM(1) = TUR: SM(2) = TUN
IF TB(3) = TBP THEN SM(1) = TUR: SM(2) = TUR: SM(3) = TUR
IF TB(4) = TBP THEN SM(1) = TUR: SM(2) = TUR: SM(3) = TUN: SM(4) = TUR
IF TB(5) = TBP THEN SM(1) = TUR: SM(2) = TUR: SM(3) = TUN: SM(4) = TUN

EXT:

Fig. 2-10. Programming to prevent throwing switches under train
We used this procedure at every dispatcher controlled CTC powered turnout on the previous Sunset Valley as well as on the new SV Oregon System. Doing so emulates exactly what is done on the prototype to prevent accidentally throwing turnouts under a train. Software effectively “locks” the turnout’s position for the time period that the turnout is occupied.

Using Figures 2-9 and 2-10 as examples, try setting up one of your own yard track arrangements to see how easy it is to program the computer to control turnouts and to prevent the switch points from being thrown when the turnout, or group of turnouts are occupied.

**Signaling a Terminal Throat**

It is natural, once turnouts are set and locked, to have the computer look at track occupancy to calculate and set the trackside signals. One way the software can set the exit signals from the terminal tracks is shown in Fig. 2-11. This program assumes there is a signal lever added to the panel; I call it SIGLEV, and that it must be set to LEFT to clear an exit signal. Also, OS(14) and BK(18) must be clear.

The key to making the code easier is to define a track aligned variable. I call it TK and set it up in the calculate turnout alignment section. Basically, if the turnouts are aligned for Track 1 then variable TK = 1, for Track 3 then TK = 3, for Track 5 then TK = 5, for Track 7 then TK = 7 and for Track 9 then TK = 9. Using this approach, the TK variable is readily available during the signal calculation to use as a subscript to pick the correct signal number, i.e. the one to which the track is aligned.

The programming in Fig. 2-11 follows very typical programming practice. Each time through the real-time loop all signals are initialized at red, or to red over red for two headed signals. Then for each signal, multiple conditions are checked in sequence to see if a more favorable condition is possible. As soon as a condition is found that keeps a given signal at stop, the program simply branches to calculate the next signal thus keeping the signal just calculated at its most restrictive indication.

REM**SET TURNOUTS BASED UPON WHICH TRACK BUTTON IS PRESSED...
REM**   ...PLUS INHIBIT THROWING SWITCH POINTS IF T HROAT IS OCCUPIED...
REM**   ...PLUS DEFINE TRACK ALIGNMENT NUMBER

IF OS(14) = OCC THEN GOTO EXT  'Skip reading buttons if throat occupied
IF TB(1) = TBP THEN SM(1) = TUN: TK = 1
IF TB(3) = TBP THEN SM(1) = TUR: SM(2) = TUN: TK = 3
IF TB(5) = TBP THEN SM(1) = TUR: SM(2) = TUR: SM(3) = TUR: TK = 5
IF TB(7) = TBP THEN SM(1) = TUR: SM(2) = TUR: SM(3) = TUN: SM(4) = TUR: TK = 7
IF TB(9) = TBP THEN SM(1) = TUR: SM(2) = TUR: SM(3) = TUN: SM(4) = TUN: TK = 9

REM**INITIALIZE ALL EXIT SIGNALS TO RED OVER RED
EXT: SW(1) = REDRED: SW(3) = REDRED: SW(5) = REDRED
SW(7) = REDRED: SW(9) = REDRED

REM**SET EXIT SIGNAL EQUAL TO RED OVER YELLOW IF TRACK ALIGNED, THROAT CREAR...
REM**   ...AND SIGNAL LEVER IS LEFT
IF OS(14) = OCC OR BK(18) = OCC THEN GOTO SIGEXT
IF SIGLEV < > LEFT THEN GOTO SIGEXT  'Lever not equal left
SW(TK) = REDYEL

SIGEXT:

Fig. 2-11. Programming exit signals at staging yard throat
PROGRAMMING REWARDS

For special situations, I occasionally perform custom programming. For example I performed contract programming for a fellow railroader because he told me, “I want to put in a large C/MRI system, I can do the hardware but I need to hire someone to do the programming.” I was so busy at the time that I tried to talk him into doing the programming but he came back very strongly with, “I can never program, I've tried before...I don't understand it.....never will....It's beyond me.....etc.” I took the job, and working with the person I got part way through and once he saw how straightforward C/MRI programming was, he took over all the programming himself. His comments changed to those like, “This programming my railroad is not bad at all....I still have a little problem now and then but for the most part everything works right away...when I do have to scratch my head for a bit, it's wonderful to see the right performance evolve....I feel better having done it myself and then I really know how it works.” He then went on to program several additional railroads. One of my greatest joys is to see such personality changes brought on by the C/MRI experience. Contagiously, it becomes exciting to key in a few lines and watch things begin to work on the railroad!

I hope by studying the above examples you will begin to see how easy it is to program the C/MRI to handle any signaling requirement including aligning and locking turnouts.

SELECTING A PROGRAMMING LANGUAGE

Programs can be written in many different languages, but one of the most common and easy to understand languages for personal computers is BASIC. It is easy to learn and for many years a BASIC interpreter was included as an integral part of nearly every personal computer. On the other hand, earlier versions of BASIC have limitations for real-time operation with external hardware, which is, interacting with external hardware as it runs. Because many versions of Basic are strictly interpreted languages (not separately compiled into digital machine code), these versions of BASIC can be very slow in executing instructions. If you run into applications that demand higher program operational speed, it is best to move beyond an interpreted version of BASIC, but it is more than adequate for getting started.

To speed up program development and operation, I recommend using one of the more advanced Basics. The two I like especially well are Microsoft QuickBASIC and Microsoft Visual Basic. QuickBASIC preceded Visual Basic so I will cover it first. The most universally applied version of Quick Basic is Version 4.5 and it is very suitable for C/MRI. QuickBASIC has a user-friendly editor that makes programming easier, it is well suited to real-time applications, capable of structured programming, no statement numbers are required and meaningful variable names can be applied. Fundamentally, I find it works very well for all real-time C/MRI programming applications not requiring a high level of graphic user interface coupled with extensive mouse and keyboard input. For those latter applications, Visual Basic is much better suited.

Microsoft has dropped QuickBASIC as a product but this is not a problem because copies are readily available from a multitude of users and via the internet for zero cost. Likewise, the restriction that QuickBASIC must operate under DOS is not much of a disadvantage as most C/MRI users tend to dedicate somewhat older computers to the railroad rather than tying up their most powerful business or family computer. Alternatively, it is possible to use third-party DOS on more modern computers. Typically, high computer power is not important to most C/MRI applications.

Although more expensive and slightly harder to learn, Visual Basic (VB) is an extremely popular language for C/MRI applications. VB operates under all versions of Windows and has many great graphics attributes. Its major advantages shine for applications requiring extensive mouse and keyboard interfacing with pull down menus, dialog boxes, control buttons and extensive color graphics. Such
capability is extremely important for modeling applications that emulate modern dispatching centers with computer generated track-train status monitoring displays. Actually, once you become familiar with using the Visual Basic structure, programming in Visual Basic is just as easy, or even easier, than programming in QuickBASIC.

Another useful language, PowerBASIC, operates under Windows. It is powerful, very fast and although it does not seem to have all the fancy bells and whistles of VB it seems well suited for more general C/MRI applications. I have no personal experience with Power Basic but if you are interested you can find more about it at www.powerbasic.com. Alternatively, as covered in the Railroader’s Application Handbook, there are many other versions of Basic that can be used for C/MRI applications. These include Liberty Basic, Just Basic, REAL Basic, Chipmunk Basic and Free BASIC.

QuickBASIC should not be confused with QBasic, an interpreted Basic that came bundled with DOS 5.0 and higher. QBasic can be used for operating C/MRI, but it lacks the programming power and speed provided by higher-level compiler-based programs like QuickBASIC, Visual Basic, Power Basic, Pascal, Turbo Pascal, C, Turbo C, C++, Visual C++ and JAVA to name just a few. All work with the C/MRI.

If you have a favorite programming language, then use it with the C/MRI. Any programming language providing access to serial I/O ports and basic IF-THEN type logic statements is usable with the C/MRI. If you are not familiar with programming, then BASIC is a good place to start. If BasicA, GW Basic or QBasic is installed on your machine then use it. If not, then I recommend you obtain a copy of Microsoft QuickBASIC Version 4.5 and/or a copy of Microsoft Visual Basic Version 6.0 or newer. Both work well for developing C/MRI applications. However, Visual Basic is definitely the better choice for applications seeking operations in the Windows environment where a high level of graphics coupled with mouse and keyboard input is desired. One nice feature is that if you do start out using QuickBASIC, and then find you have a desire to move up to Visual Basic, you will discover that almost all of your programming remains identical. In fact, in Chapter 16 I will show how easy it is to convert QuickBASIC programs to Visual Basic!

As with any programming language, BASIC instructions vary a bit from one manufacturer to another. The two BASIC languages I use in this manual are Microsoft QuickBASIC V4.5 and Microsoft Visual Basic V6.0. The motto for all my programming is: Keep programming simple, straightforward and easy to understand. All my programs include extensive commenting so that reading the resulting code is much like reading the English language. The result is that these programs should be easy to convert to your favorite language, if this has not already been accomplished for you via the enclosed disk or via the C/MRI User’s Group.

I cannot take the space to teach BASIC, QuickBASIC or Visual Basic or any other programming language in this manual, however most statements that we will need are easy to understand. Most languages you purchase come with a manual and computer stores have a wide assortment of books on most every programming language. One title I have always liked is BASIC BASIC. You can hardly get more basic than that!

If you desire more information on Microsoft QuickBASIC, I highly recommend The Waite Group’s, MICROSOFT QuickBASIC BIBLE, published by Microsoft Press. Also, BASICS for DOS, by Gary Cornell, McGraw-Hill Publishing., provides excellent coverage of GW-BASIC, BasicA and Microsoft QBasic. Both books are out of print, but copies have been found through www.amazon.com and www.ebay.com. There are a hundreds of books available on Visual Basic. A good introductory book to get started with is Visual Basic 6 - Weekend Crash Course by Richard Mansfield (ISBN 0-7645-4679-1). Additional VB references are included in this manual at the end of Chapter 15.
All Visual Basic examples operate with the C/MRI under a Windows environment. The QuickBASIC examples are written to operate under DOS. A software disk, enclosed with the User’s Manual, includes all the software presented in this manual using both QuickBASIC and Visual Basic. Be sure to start with the README file for a more detailed explanation of the disk’s content. Also make a working copy of the disk and save the original in a safe place prior to tailoring the programs to best fit your own needs.

**DEFINING PROGRAM VARIABLES**

Independent of the language selected, every railroad device that is interfaced to the computer needs to be given a unique name for use by the software. Such unique names, within software, are typically referred to as symbol names or variables. In the C/MRI documentation, including this manual, I use the term variable.

I like the term variable as it best represents the variable nature of the different railroad devices. For example, signals change between red, yellow and green. Turnouts change between normal and reverse. The signal levers on dispatcher CTC panels vary in position between left, stop and right.

A key to making programming easy is selecting meaningful variable names to represent each railroad device. This was rather difficult using original versions of BASIC which limited you to two character variable names. However, most programming languages are much more lenient. For example, QuickBASIC lets you pick variable names with up to 40 alphanumeric characters in length. The first character must be a letter and you cannot include blank spaces but the alphabetical characters can be any combination of upper or lower case. The result provides very wide latitude in naming railroad devices. For example, you can use SIGNAL12, Signal12, SalemTurnout9, Turnout17, TURNOUT21, SignalLever192, or almost whatever else you desire.

For convenience if you wish, you can elect to use subscripted variables like SignalEast(1), SignalEast(2), SignalEast(3) and so forth up to some defined maximum such as SignalEast(35). The number enclosed within the parentheses is referred to as the subscript or index. A DIMension statement is required to define the maximum range of each subscripted variable. In this case we would need to add the statement DIM SignalEast(35). When an error message “Subscript out of range” appears, it means that your subscript usage is larger than the size of the DIMension. If you use a subscripted variable without a corresponding DIMension statement, QuickBASIC defaults to a dimension of 10.

Every variable is automatically allocated its own unique memory location within the computer. For example, every time QuickBASIC or Visual Basic encounters a new variable name within an application program, it assigns to it a unique memory location. Likewise, when QuickBASIC or Visual Basic encounters a DIM SignalEast(35) statement it reserves 35 memory locations to contain SignalEast(). When an application program executes a statement containing SignalEast(I) for example, it is the value of I, the subscript, enclosed within the parentheses that determines which SignalEast( ) is being addressed.

Numerous programming languages let you include underscores within a variable name, such as Signal_East_12. Many programmers tend to use such very long concatenated variable names, for example, Dunsmuir_Signal_No._12_East. I personally find using such long names laborious to input via the keyboard, they are prone to making entry errors, they do not let you include much logic on a given coding line and the overall impact tends to clog up the program listing with a lot of unnecessary characters. I prefer using shorter more abbreviated, yet still meaningful, variable names. I typically select my variable names between 2 and 10 characters in length and usually all upper case. Some of the typical variable names I like to use are CLR for clear, OCC for occupied, RED for red, YEL for yellow, GRN for green, REDRED for red over red, SE( ) for signals east, SW( ) for signals west, BK( ) for block occupancy, SM( ) for switch motors, MAXTRIES for maximum tries before aborting inputs, and so forth.
Where you have a whole bunch of similar devices such as signals, block occupancy detectors, switch motors, and so forth it is often easiest to simply number them consecutively as 1, 2, 3, 4 etc. and make use of subscripted variables such as SIG( ), BK( ) and SM( ). On the other hand, if your railroad already happens to be documented using non-contiguous numbering, such as found on a dispatcher’s CTC machine, or on prototype intermediate block signal number plates (that are based upon milepost nomenclature), then subscript notation does not work very well. In these cases, you are better off using a unique variable name for each device.

For example, reading from the SV’s Oregon System US&S CTC machine, selected variable names for the trackside signals can be written as; TS50RAB, TS50LA, TS50LB for the triad of signals at a typical prototype OS section, where the TS stands for Trackside Signal, the R and L denote right or left facing signals on the CTC panel, and the A and B denote the signal head arrangement. The AB denotes a dual-head signal leading into the facing point end of a passing siding turnout. The A-only denotes a single-head mast signal at the frog end main track and the B-only denotes a dwarf leading out of the siding.

Fundamentally, what I am trying to get across is that you can name your software variables representing each railroad device in any manner that is the most comfortable for you. Picking software variable names that correspond to your railroad’s wiring documentation, prototype layout and/or your control panel designations can provide significant advantages. I am biased, I must admit, but I recommend where feasible to stay close to those variable names I use in my numerous C/MRI example programs. Many additional recommended variable naming conventions are presented in the Railroader’s C/MRI Applications Handbook. In summary, choose a system and use it faithfully. The C/MRI software can easily handle whatever variable naming convention you prefer.

STRUCTURED PROGRAMMING

Similarly to keeping your wiring neat and well documented, there is a lot to be said for doing the same with your software. Well documented programs, with plenty of meaningful remarks spread throughout the code, are well worth the extra effort.

Whole books are written on structured programming which from our view might reflect on how best to set up a program's organization. Typically however, for real-time applications, the best – from both a processing time as well ease in understanding perspective – is straight line code. Thus even with modular programming (covered in Chapter 13), it is still best to employ straight line programming within each module.

In my opinion, many professional programmers and some C/MRI programmers, complicate their approach in the pursuit of optimized Structured Programming. In doing so, they tend to include a large variety of different branching-type statements such as DO WHILE, DO UNTIL, EXIT DO, IF-THEN-ELSEIF-THEN-ELSE-ELSE-END-IF, ON…GOSUB, ON...GOTO, CHAIN, WHILE...WEND, FOR EACH...NEXT and a whole bunch more. I recommend taking the exact opposite approach preparing C/MRI programs. The guidelines I like to follow are:

1. Use a very small subset of basic instructions and then use these same statements over-and-over again with very minimal changes like substituting different numbers and variable names.
2. Use only the IF-THEN and the IF-THEN-ELSE statements for conditional branching.
3. Use only the GOTO statement for unconditional branching and then only for making short forward jumps to branch around short sections of code.
4. Use abbreviated yet still meaningful variable names and statement labels.

5. Include one or more documentation lines at the beginning of every short block of statements to describe the codes purpose and action being taken. In addition, for added clarity, append further documentation to subsequent statements.

6. Use a single global area to define all variables that must be accessible by two or more modules. This item applies only when using CALL statements (covered in Chapter 13 and further used in Chapters 14 and 15) which are encouraged for use within most medium to large application programs.

In summary you might say that, “I limit myself to using a minimum subset of statements that gets the job done.” Likewise, I purposely stay away from using alternate statements that might appear as being more elegant or a “better way to execute code.” My recommended procedures may not take advantage of the most powerful aspects of a given language or reduce the number of coding lines but they do provide programs that are easy to generate, to understand and that work exceptionally well. This approach is valid independent of whether we are programming in QuickBASIC or Visual Basic. My motto in all C/MRI programming is, “Keep your programming simple, straightforward and easy to understand!”

It is my belief that the vast majority of C/MRI interested readers have never been exposed to programming. By demonstrating that a user need only learn a very small set of statements and then to use these repeatedly, with minimal changes, is the best approach to helping more railroaders take advantage of everything a computer can offer to increase hobby enjoyment. My underlying goal is to make every program come as close as I can to read like reading the English language.

I spent a majority of my 35 year engineering career creating and supervising the creation of real-time software systems for the aerospace industry. Keeping programming simple, straightforward and easy to understand while at the same time doing the job that needs to be done is a worthy and achievable goal.

---

**Important Point**

New C/MRI users simply need to learn a few basic programming statements. Using these statements repeatedly – with very minimal changes – new users (or any user) can generate almost any desired C/MRI application program.

---

The above is true if all you want to do is monitor train location and light a few signals, control grade crossing protection devices, or if you want to prototypically signal a vast network of trackage using multiple head signaling. It applies equally well to the implementation of Computer Cab Control with interfacing to a lever-type CTC machine and an Entrance-Exit type interlocking plant.

To me, being able to accomplish all the above while at the same time keeping the program simple, straightforward and easy to understand is what C/MRI programming is all about!

**ESTABLISHING PROGRAM FLOW**

With all variables named, and their memory location automatically defined by QuickBASIC or by Visual Basic, (or by whatever language you are using), the C/MRI application program merely executes the program statements one statement at a time. Unless directed otherwise the application program executes the statements in a linear fashion starting from the top and sequentially working through to the bottom of the program. For example, Fig. 2-12 includes a grouping of typical programming type statements.
**Sample Code Lines from Initialization Section**

- **DRK** = 0
- **GRN** = 1
- **YEL** = 3
- **RED** = 2
- **REDRED** = 10
- **REDEYELRED** = 46

**Sample Code from Real-Time Loop Section**

- **TS52LAB** = **REDRED**
- **TS52RA** = **GRN**
- **TS52RC** = **RED**
- **TS72RA** = **RED**
- **TS72RB** = **RED**
- **TS72LABC** = **REDEYELRED**

---

**Fig. 2-12.** Sequentially stepping through typical programming statements

Proceeding sequentially through the initialization statements, the PC program takes the value 0 and places it into the memory location named **DRK**. Likewise it places the value 1 in the memory location named **GRN**, 3 in the location named **YEL**, 2 in location named **RED**, 10 in location named **REDRED** and 46 in location named **REDEYELRED**. Later on, I will show how we arrive at these numbers. In this case they happen to be the standard signal aspect constants for searchlight signals. Because the values do not change throughout the program’s operation, it is important to note that variables can hold constants just as well as variable numbers. For this reason I will frequently refer to non-changing variables such as **DRK**, **GRN** and **RED** as constants, in this case signal aspect constants, rather than variables.

While executing the statement examples within the real-time loop, the first statement sets Trackside Signal 52LAB to red over red. It does this by copying the value stored in memory location named **REDRED**, which is decimal 10, into the memory location for **TS52LAB**. It should be noted that executing this statement changes only the signal’s value as stored in the computer’s memory and does not change the actual trackside setting. The latter only occurs when the signal values are transmitted or written to the railroad.

Taking the statements in sequential order, as the PC executes the program, the next statement sets **TS52RA** to green, followed by setting **TS52RC** to red, etc., through to setting the 3-headed Trackside Signal **TS72LABC** to red over yellow over red.

Without specific statements to alter the program flow, the PC simply executes the statements in sequence from top to bottom. Although it makes no difference how program statements are executed, multiple statements can be placed on the same line. For example the bottom four statements in Fig. 2-12 could be written as follows and will provide the exact same results.

- **TS52RC** = **RED**: **TS72RA** = **RED**: **TS72RB** = **RED**: **TS72LABC** = **REDEYELRED**

The colon is use to separate multiple statements on a given line.

**IF-THEN Conditional Branching**

Using the simple power of IF-THEN logic, what can be accomplished by a new C/MRI user is truly amazing. The application of this statement is so easy to understand. We live with the logic of it throughout every day of our lives:

- **IF Janet comes home before noon THEN I am taking her to the movies**
• IF Billy comes over to visit THEN we are going to run some trains
• IF there is a “special” on apples today THEN I am going to buy a peck

Application of the IF-THEN-ELSE statement is almost as easy to understand.

• IF Billy comes over THEN we will run trains ELSE I will clean the garage
  (As a side note, I sure hope Billy comes over)
• IF it rains today THEN I will get ready for that op session ELSE I will mow the yard

Applying the same logic to programming our model railroads we have:

• IF Track Alignment Button 7 is pressed THEN Switch Motor 5 is set normal and 9 is set reversed
• IF Signal 24 is red THEN Signal 32 is yellow ELSE Signal 32 is green

All a new C/MRI user needs to do is to combine the logic power of the IF-THEN statement with a few other statements like FOR-NEXT, GOTO and GOSUB and creation of some very effective C/MRI application programs is now possible!

The IF-THEN statement is a natural step for altering program flow based upon different situations. For example let’s assume we need to set SE(3) = GRN if the switch motor for Turnout 9, denoted as say SM(9), is normal, or alternatively to set SE(3) = RED if SM(9) is reversed. The way we will most frequently alter program flow is using the IF-THEN statement. One way to handle this situation is to write:

\[
\begin{align*}
SE(3) &= \text{RED} \\
\text{IF } SM(9) &= \text{TUN } \text{THEN } SE(3) = \text{GRN} \\
&\vdots
\end{align*}
\]

Firstly, Signal SE(3) is initialized to red, followed by executing the next statement, which is the IF-THEN statement. Here IF SM(9) = TUN, i.e. if the turnout is in its normal position, the IF statement is true so the program alters its flow to execute the statement immediately to the right of THEN. This sets SE(3) to green followed by executing the next statements below the IF, represented by the three dots. IF SM(9) is not equal to TUN, i.e. the turnout is reversed, the IF statement is false and the program does not execute the statement to the right of THEN and proceeds directly to the statements represented by the three dots.

As an alternative we could use the IF-THEN-ELSE statement to achieve the same results:

\[
\begin{align*}
\text{IF } SM(9) &= \text{TUN } \text{THEN } SE(3) = \text{GRN } \text{ELSE } SE(3) = \text{RED} \\
&\vdots
\end{align*}
\]

In this case, if SM(9) = TUN, i.e. the turnout is normal, then the statement SE(3) = GRN is executed prior to moving on to the statements represented by the 3 dots. Else if SM(9) is not normal, i.e. it is reversed, the statement SE(3) = RED is executed before continuing with the statements represented by the 3 dots. Here we have altered the program flow through two different paths dependent upon the state of the switch motor and its corresponding turnout position.

It is quite easy to run out of line space when using IF-THEN and IF-THEN-ELSE statements. For these situations the Block IF format comes to the rescue, for example as with the following typical block of software diode matrix statements:
IF TB(16) = TBP  THEN
  SM(1) = TUN; SM(2) = TUR; SM(3) = TUN; SM(4) = TUN; SM(5) = TUR
  SM(6) = TUR; SM(9) = TUN; SM(11) = TUR; SM(14) = TUN
END IF

If Turnout Pushbutton 16, named with variable TB(16), is pressed, denoted by variable TBP for Turnout Button Pressed, then the block of statements between the IF-THEN and the END IF statements is executed. However, if TB(16) is not pressed the block within the IF statements is skipped.

The Block IF-THEN-ELSE works much the same way, as for example:

IF {expression} THEN
  {'A' block of statements}
ELSE
  {'B' block of statements}
END IF

IF the expression between the IF and the THEN is true, then only the A block of statements is executed. However, if the expression is false only the B block of statements is executed. Once either the A block or the B block of statements is executed the program continues with the statements represented by the 2 dots.

It is important to note how selected statements are indented in the above examples. Following this procedure makes reading easier and is a recommended programming practice. For example, indenting all the statements between the IF-THEN statement and the corresponding END-IF statement clearly indicates which statements are included within the use of the Block IF format.

FOR-NEXT Loops

The FOR-NEXT loop, introduced previously, also alters program flow. For example:

FOR IJ = 1 to 100
  Signal(IJ) = RED
NEXT IJ

Here the program keeps looping through these 3 statements starting with IJ = 1, and with each iteration of the loop the value of IJ is incremented by 1. Once the NEXT IJ statement is reached with IJ = 100 there is no more incrementing to perform. Thus, the program flow continues by executing whatever statements exist below the NEXT IJ statement as represented by the 3 dots. The above statements could also be written on one line as:

FOR IJ = 1 to 100: Signal(IJ) = RED: NEXT IJ

There are many other types of conditional logical branching, such as the ELSE-IF, the DO UNTIL, and the DO WHILE statements to name a few. However, making use of a bunch of additional statements really is not necessary to our C/MRI programming. Therefore, in support of my philosophy to keep programming simple, straightforward and easy to understand, the examples in this manual do not use these additional statements.
### Important Point

All you need to learn is a few basic statements and to use these statements repeatedly – with minimal changes – you can generate any desired C/MRI application program.

---

**GOTO Statements**

The GOTO statement creates what is called an unconditional branch. It is the most direct method of altering program flow. Each use of GOTO requires a “branch-to” label immediately to the right of the GOTO command. A frequent example we will be using is GOTO BRTL. Each time the program execution reaches the statement GOTO BRTL, the program diverts from its sequential execution of statements and immediately branches to execute whatever statement is located after the BRTL label. For example, consider the following programming situation:

```
BRTL: {label for next statement executed after GOTO BRTL statement is executed}

戈

GOTO BRTL
```

The addition of the colon after the BRTL label is required to indicate to the basic compiler that the characters preceding the colon form a label. It should be noted that the label can be placed anywhere within the program. When the program execution reaches the GOTO BRTL statement, an unconditional branch is executed to the label BRTL regardless of where the BRTL is located. The above coding illustrates what I call a “backward branch”. This is where a branch is created back into the program, at a point before the GOTO instruction.

Care should be taken to avoid backward branching. Why? When executing backward branching several times within a program it is extremely easy to lose track of how the program flows. Also, with backward branching it is easy to put your program into an infinite loop, where you branch back, move forward, branch back, move forward and branch back ad infinitum – resulting in a program “lock up” and gets you nowhere. As dictated by good programming practice, the only place I will be using backward GOTO branching in our C/MRI application programs is at the end of our real-time loop. That is where we need to return to the beginning of our real-time loop, i.e. GOTO BRTL, in order to keep our program looping.

By contrast, short forward GOTO branching is extremely useful in C/MRI programming. Most signal programming involves initializing signals to RED and then checking a number of conditions to see if signals can change to a more favorable aspect. Anytime a condition is found whereby a signal needs to be retained at RED, you simply branch directly to the starting point for determining the next signal – keeping the previous signal at RED. Such programming becomes second nature once we complete a few signaling examples in Chapter 8.

The major ground rule to effectively use forward GOTO branching is to keep multiple branches short and where possible to the same forward point. This is the procedure I will follow in all our application program examples.

**GOSUB Statements**

The GOSUB statement performs much like the GOTO statement with one major difference. The GOSUB label statement causes a branch to the label but in doing so the software saves the location of the program at the statement immediately following the GOSUB statement. This way, once the program continues...
with executing the statements following the label and it comes to a RETURN statement, the program returns (using the saved location) to execute the statement immediately below the GOSUB.

Basically, incorporating a GOSUB statement enables the program to branch to another block of statements, called a subroutine, and then to RETURN to the statement immediately following the GOSUB. Fig. 2-13 illustrates the situation.

![Diagram of GOSUB function](image)

**Fig. 2-13.** How a GOSUB functions

Note that the branch-to label contained within the GOSUB statement coincides with the label used to start the subroutine. In this case, I have used the name INIT to correspond to the name given to the subroutine within the GOSUB version of our standard Serial Protocol Subroutine package (SPSBG). Subroutine INIT is invoked each time we want to initialize a serial node.

Using GOSUB comes in handy when you want to branch to do something special, especially if you want to do this repeatedly or from several different locations within a program. Each time the GOSUB is executed you always return back to continue where you were before branching. When we begin serial programming examples, I will frequently use the GOSUB statement each time we initialize a node, or write outputs to a node or read inputs from a node.

The subroutine code using GOSUB, although it may appear to be separate from the main program, actually remains a part of the mainline program. This means that all variables are equally available to the main program code and to the subroutine code. Therefore, for example, if you happen to set SE(45) = REDRED prior to the GOSUB execution and you added a PRINT SE(45) statement in the subroutine code, it would print out the red over red status of Signal East 45. Likewise any variables defined in the subroutine are available in the main body of the program.

For entry level programming and for programming small to medium systems, the availability of all variables across subroutine boundaries can be of significant advantage. You only have one overall set of variables and they are all available no matter what particular area of the program is being written.

As your programming becomes more advanced and/or you tackle larger system applications, it is often desirable to use CALL statements rather than GOSUB statements. When using a CALL statement the subroutine being called actually becomes a separate program module. I will cover modular programming and the use of the CALL statement in Chapter 13 and then use the techniques in Chapters 14 and 16.

**SERIAL PROTOCOL SUBROUTINES**

Serial interfacing requires the ability to convert the data format from parallel to serial and vice versa, coupled with the ability to send card addresses and data in serial form over the same wire. Special
hardware built into PCs and into the Microcontroller chips used with the SMINI, SUSIC and USIC automatically performs the required parallel-serial conversions. We will make use of special software, typically referred to as **Serial Protocol Subroutines**, to interact with the hardware.

The protocol subroutines we will use with the C/MRI automatically form the required packets of information, called messages, to be transmitted to the external hardware. It handles the messages received from the interface hardware. The software inserts special control characters to mark the beginning and end of transmissions, as well as define which bytes are addresses and which are pure data. This may sound complicated, but I have done the hard work for you by providing standardized routines to carry out the C/MRI protocol. All you need to do is to initialize a few variables and then just invoke the appropriate protocol subroutine.

To meet different programming desires, I provided three new and different serial protocol packages with the release of the Version 3.0 C/MRI User’s Manual. Table 2-1 summarizes these packages including the package name along with a brief description and example applications.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Example Application Areas</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPSQBG</td>
<td>Serial Protocol Subroutine Package using QuickBASIC’s GOSUB Option</td>
<td>Typically used for smaller applications programmed in such languages as Basica, GW Basic, QBASIC, QuickBASIC and Power Basic operating under DOS</td>
</tr>
<tr>
<td>SPSQBC</td>
<td>Serial Protocol Subprogram Package using QuickBASIC’s CALL Option</td>
<td>Typically used for larger applications programmed in QBASIC and QuickBASIC operating under DOS and Power Basic operating under DOS or Windows</td>
</tr>
<tr>
<td>SPSVBM</td>
<td>Serial Protocol Subprogram Package using Visual Basic’s CALL Option with Microsoft provided MSComm</td>
<td>Typically used for Visual Basic applications operating under Windows using Microsoft provided MSComm for serial communications</td>
</tr>
</tbody>
</table>

Although not specifically provided at this time, it is reasonable to expect that an equivalent standard Serial Protocol Subroutine package will become available to support of the C-based family of languages.

Using one of the standard Serial Protocol Subroutine packages offers a big advantage in that all the serial data communications between the PC and the C/MRI hardware is handled for you automatically. You do not need to be concerned about any of the details. At a point in your program where you want to initialize a node you add the statement GOSUB INIT, or CALL INIT. At the point where you want to transmit outputs to a C/MRI node you add the statement GOSUB OUTPUTS or CALL OUTPUTS. Finally, when you want to receive inputs back from a node you include a GOSUB INPUTS or a CALL INPUTS. Table 2-2 summarizes the situation.

<table>
<thead>
<tr>
<th>Statements used with GOSUB option</th>
<th>Statements used with CALL option</th>
<th>Function performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>GOSUB INIT</td>
<td>CALL INIT</td>
<td>Initializes a node (USIC, SUSIC or SMINI)</td>
</tr>
<tr>
<td>GOSUB OUTPUTS</td>
<td>CALL OUTPUTS</td>
<td>Transmits all output bytes to a node</td>
</tr>
<tr>
<td>GOSUB INPUTS</td>
<td>CALL INPUTS</td>
<td>Receives all input bytes from a node</td>
</tr>
</tbody>
</table>
Understanding the internal details of the different Serial Protocol Subroutines is not important to their successful application. However, for readers interested in the details concerning these packages and how they operate, such material is covered in Appendix B for the QuickBASIC version and in Appendix C for the VB version. Also, the source code for each package is provided on the disk enclosed with this manual. Making effective use of INIT, OUTPUTS and INPUTS subroutines becomes second nature once we get into Basic Programming Examples in Chapter 7 and into the numerous application examples covered in Chapters 9, 12, 13, 14 and 16.

****Important Point****

Reader’s desiring to update existing C/MRI programs, so that they make use of the updated Serial Protocol Subroutines presented in this manual, should consult the last sections of Appendix B for a list of the steps required to make the conversion.

**BINNARY NUMBER SYSTEM**

When we talk to a computer, we use the decimal number system for the most part: the digits 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. The computer, however, uses a binary number system which consists of only two digits, 0 and 1, to do everything we do with 0 through 9.

Computers use the binary system, because it is very easy to handle this system electronically. Inside a computer are millions of electronic switches, transistors that store and control the flow of electrical signals. For example, a midrange Pentium chip has over three million transistors and that is only a small fraction of the total in a Pentium-based system. You can think of these transistors as if they were switches, like those in Fig. 2-14.

Each switch is separately controlled and can be either on or off, representing a single binary digit. When the transistor is on, then the value is 1 and the value is 0 when the transistor is off. A single switch can represent only two codes, but a pair can represent four, equivalent to decimal numbers 0 through 3. Each time a switch is added, the number of codes that can be represented doubles. Three switches can represent eight binary codes, four switches can represent 16, five switches can represent 32, six switches can represent 64, and so on.

The abbreviation for a single binary digit is **bit**. A bit can have only two states, but they may be referred to as on or off, high or low, true or false, +5V or 0V, as well as 1 or 0. The personal computers you will want to devote to C/MRI applications are either eight-, sixteen-, thirty two- or sixty four-bit machines, which means that their electronic switches are arranged in groups of 8, 16, 32 or 64. A group of eight contiguous bits is called a **byte**, and as Fig. 2-14 shows, it takes eight binary switches to represent, or store, one byte of information.

If all eight switches are on, they indicate binary 11111111, which is equivalent to decimal 255. The decimal equivalent is the sum of the decimal equivalent values of each switch position that is turned on. The value is read from right to left and each switch doubles in decimal equivalent value. Each switch that is off equals zero, therefore, one byte can represent all decimal numbers between 0 and 255, a total of 256 binary codes.
Table 2-3 shows the relationship between the number of address bits and the size of memory that a computer can address. The entries are simply powers of two. For example, a 16-bit bus supports an address space of $2^{16}$ or 65,536 decimal locations that can be individually addressed.

In computer jargon, 65,536 is referred to as 64K bytes, or 64KB for short. In the general sciences, the symbol K typically stands for Kilo, Greek for 1000, however, in the computer industry K stands for $2^{10}$ or 1024. This difference can be confusing, but in this manual K, or Kilo, will always be 1024.
When dealing with computers, the 1024 interpretation is handiest since computers “think” in powers of two, i.e. the binary number system. Likewise, in the computer field, M (standing for Mega or million), is equated with $2^{20}$ or 1,048,576 locations and G (standing for Giga meaning giant in Greek or billion), is equated to $2^{30}$ or 1,073,741,824. Note that M = K times K and G = K times K times K or more simply M times K. It is also interesting that each time you add a single bit you double the address space.

In Table 2-3 numbers, for instance 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, and 1024, coincide with numbers we will be using throughout this manual. Virtually every number we use with computers has its roots in this powers-of-two table. We’ll use these relationships extensively as we proceed through this manual so you might find it handy to refer back to Table 2-3 when needed.

Binary is good for computers, but cumbersome for humans, so I will not deal in binary any more than absolutely necessary. One reason binary is cumbersome is that it takes a combination of eight 1’s and 0’s to represent a byte. A decimal number from 0 to 255 can also define the contents of a byte, but it is difficult to see the correspondence between the decimal number and the binary bit pattern. This is where the hexadecimal number system comes to the rescue.

### HEXADECIMAL NUMBER SYSTEM

I have pretty well eliminated the use of hexadecimal numbers in this version of the C/MRI User’s Manual and in the Railroader’s C/MRI Applications Handbook. Therefore you may want to skip this section and come back to it later if a need arises. However, for those interested, I will go ahead and explain hexadecimal in case you run across its usage as part of the numerous discussions and application examples found on the C/MRI User’s Group at http://groups.yahoo.com/group/cmri_users.

The hexadecimal number system has 16 digits, or characters: 0 through 9 plus A through F. The letters are symbols used as digits, with A = 10, B = 11, C = 12, D = 13, E = 14, and F = 15. These 16 digits, 0 through F, can represent all possible bit combinations for a 4-bit group, sometimes called a nibble. A nibble represents half a byte. Fig. 2-15 shows corresponding hexadecimal, decimal, and 8-bit binary-coded numbers. A complete table would take 256 rows, 0 through 255, so I have eliminated many higher order rows.
Fig. 2-15. Hexadecimal, decimal and binary numbers

Being able to eliminate these rows, yet still have all the information we need, is one reason hexadecimal is so handy. It takes only two hexadecimal characters to represent any eight-bit binary code, a byte. That is convenient, but the main advantage of hexadecimal is the ease of converting binary to hex and hex to binary – much easier than with decimal numbers.

You can ignore all entries in Fig. 2-15 except those shown in bold type face. These are the hexadecimal numbers 0 through 9 followed by A through F, the corresponding decimal numbers 0 through 15, and the rightmost four zeros and ones in the binary column corresponding to decimal numbers 0 through 15.

This small portion, highlighted with a heavy outline border, is the only part we need to understand, because everything else can be derived from it. For example, a hexadecimal 11 is binary 0001|0001, a hex BB is 1011|1011, and a hex 1B is 0001|1011. The left hex character defines the left four binary bits, and the right hex character defines the right four bits. This can be very useful when you write application software to use with your C/MRI. For example, users employing languages such as Pascal and C typically make extensive use of hexadecimal. Whenever I use hexadecimal numbers, I either append a lower case h, so hex number D9 will appear as D9h or I write the number in the format 0xD9.

COUNTING CARDS/NODES USING NUMBER ZERO

Writing the original Model Railroader C/MRI series I debated whether to illustrate counting nodes, and I/O cards, starting with the numeral 0 or the numeral 1. In the computer world there are advantages to start counting with 0. However, almost since the beginning of time people have always counted 1, 2, 3,... versus starting counts as 0, 1, 2, 3,... Therefore, figuring it would be easier to understand, I elected to stick with 1, 2, 3,... when counting nodes and I/O cards.
Over time though, I have found that counting cards as 1, 2, 3,.... has created confusion when setting 
C/MRI card address DIP switches that require counting cards as 0, 1, 2, 3... The result was that the user 
always needed to subtract 1 from the card number when setting the card’s DIP switch. This got to be such 
a nuisance that starting with Version 3.0 of the C/MRI User's Manual, the counting sequence for nodes 
and cards is changed to 0, 1, 2, 3,... that is the first node in a distributed serial system is now called Node 
0 and the first I/O card within any given node is now called Card 0. This makes it easy because the node 
and card numbers are in exact agreement with their respective DIP switch settings.

Although certainly not essential to correctly count cards and set DIP switches, it can be interesting to look 
a little further into counting procedures. The decimal number system is based upon using 10 numerals or 
digits. Although many people may think of the 10 numerals as being the numbers 1 through 10, in 
actually, they are the digits 0 through 9. Every decimal number is constructed from these 10 digits 0 
through 9. Thus, you could count on your fingers saying 0, 1, 2, 3,... that is your 10 fingers are numbered 
0 through 9. When you increment 1 above the maximum digit count of 9, you get a 10 (a one-zero), which 
in reality is a carry of 1 with the base digit starting again at 0.

The same is true for the binary number system. Every binary number is constructed from its 2 numerals 0 
and 1. When you increment 1 above the maximum base count of 1 you get 10 (one-zero), which in reality, 
is a carry of 1 with the base digit starting again at 0.

With the above in mind, take a look back at Fig. 2-15 to see how counting accumulates in binary, decimal 
and hexadecimal. In decimal, each time we increment a 9 we carry over 1 to add the next column and 
return the base column to 0. We do the same in binary, where each time we increment 1 we carry over 1 
to add the next column and return the base column to 0. As a reinforcing experience, and without using 
Fig. 2-15 as a crutch, start at 0000 and try counting in binary. Take a piece of scrap paper and write down 
your counts in a vertical column as:

```
0000
0001
0010

```

Generating the rest of the sequence is up to you. Just remember, for each column, anytime you add 1 to 1 
you get 0 with a carry over of 1 to the next column. When you are done, check your results to make sure 
they agree with the rightmost column in Fig. 2-15.

If you get as far as 1111, which corresponds to the decimal number 15, you should have counted 16 
different entries numbered 0 through 15. These entries could be, for example, 16 I/O cards numbered 0 
through 15. Or they could be 16 C/MRI system nodes numbered 0 through 15.

In Fig. 2-15, the decimal column denotes the card, or node, number while the binary column is used to set 
the corresponding card address DIP switch segments, where 0 equates to a switch segment turned off and 
1 equates to switch segment turned on. Thus, Card 0 corresponds to having all switch segments off. Card 
1 then corresponds to having the rightmost segment turned on. Card 2 corresponds to having the second 
from the right switch segment on, and so forth.

Using this starting from zero card/node numbering procedure, it is always best to ignore DIP switch 
setting tables found in earlier C/MRI and UCIS publications and stick with the updated numbering system 
introduced with V3.0 of the C/MRI User’s Manual and V3.0 of the Railroader’s C/MRI Applications 
Handbook. That way, everything is consistent between card number and DIP switch setting.
If you ever need to convert from binary to decimal, it is quite straightforward following the procedure illustrated in Fig. 2-14. Each binary bit position has a weighted decimal value equal to, from right to left, 1, 2, 4, 8, 16, 32, 64 and 128.

For readers inclined toward a deeper mathematical understanding, the weighted decimal value of each bit position is simply the number 2 raised to the power denoted by the bit position. For example, bit position 0 has the weighted decimal value of $2^0$ or 1. Bit position 1 has the weighted decimal value of $2^1$ or 2 while bit position 2 has the weighted decimal value of $2^2$ or 4, bit position 3 has the weighted decimal value of $2^3$ or 8, and so forth up to bit position 7, which has the weighted decimal value of $2^7$ or 128.

Table 2-4 summarizes the weighted decimal value of each bit position within an 8-bit byte. To obtain the decimal equivalent for any 8-bit byte, simply add up the weighted decimal values for each bit that is turned on.

<table>
<thead>
<tr>
<th>Bit Position</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decimal Value</td>
<td>128</td>
<td>64</td>
<td>32</td>
<td>16</td>
<td>8</td>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

For example, with all 8 bits turned on, a binary 11111111, the decimal value of the byte is calculated as $128 + 64 + 32 + 16 + 8 + 4 + 2 + 1$ which equals 255. With all 8 bits turned off, binary 00000000 is calculated as the sum of all zeros for a decimal value of 0. This confirms what we said earlier that an 8-bit byte can store 256 different values 0 through 255.

Selecting an in between binary number of say 01001011 corresponds to a decimal equivalent value of $0 + 64 + 0 + 0 + 8 + 0 + 2 + 1$ which adds up to be 75. If you are interested in perfecting your skills in this area, pick a few of the binary entries in Fig. 2-15 and compute the corresponding decimal equivalent. Check your results against those shown in Fig. 2-15 to make sure you are doing the math correctly.

When DIP switch settings are necessary, I’ll present tables that illustrate actual switch setting for different card numbers. Therefore, doing the math is seldom required. However, once you have the above math understood, with a little practice, you will easily be able to read DIP switch settings as decimal numbers as well as binary.

**NODE AND I/O CARD ADDRESS DIP SWITCH SETTINGS**

Fig. 2-16, contained on the following page, pictorially shows DIP switch settings for different card and node numbers (addresses). Using this circumvents the need to do the mathematics for most addresses. Keep it handy to use over and over again each time there is a need to set a card or node address DIP switch.
Fig. 2-16. DIP switch settings for different card/node numbers (addresses)

Note: There are 128 possible nodes addresses (0 – 127) and only 64 possible I/O card addresses (0 – 63)