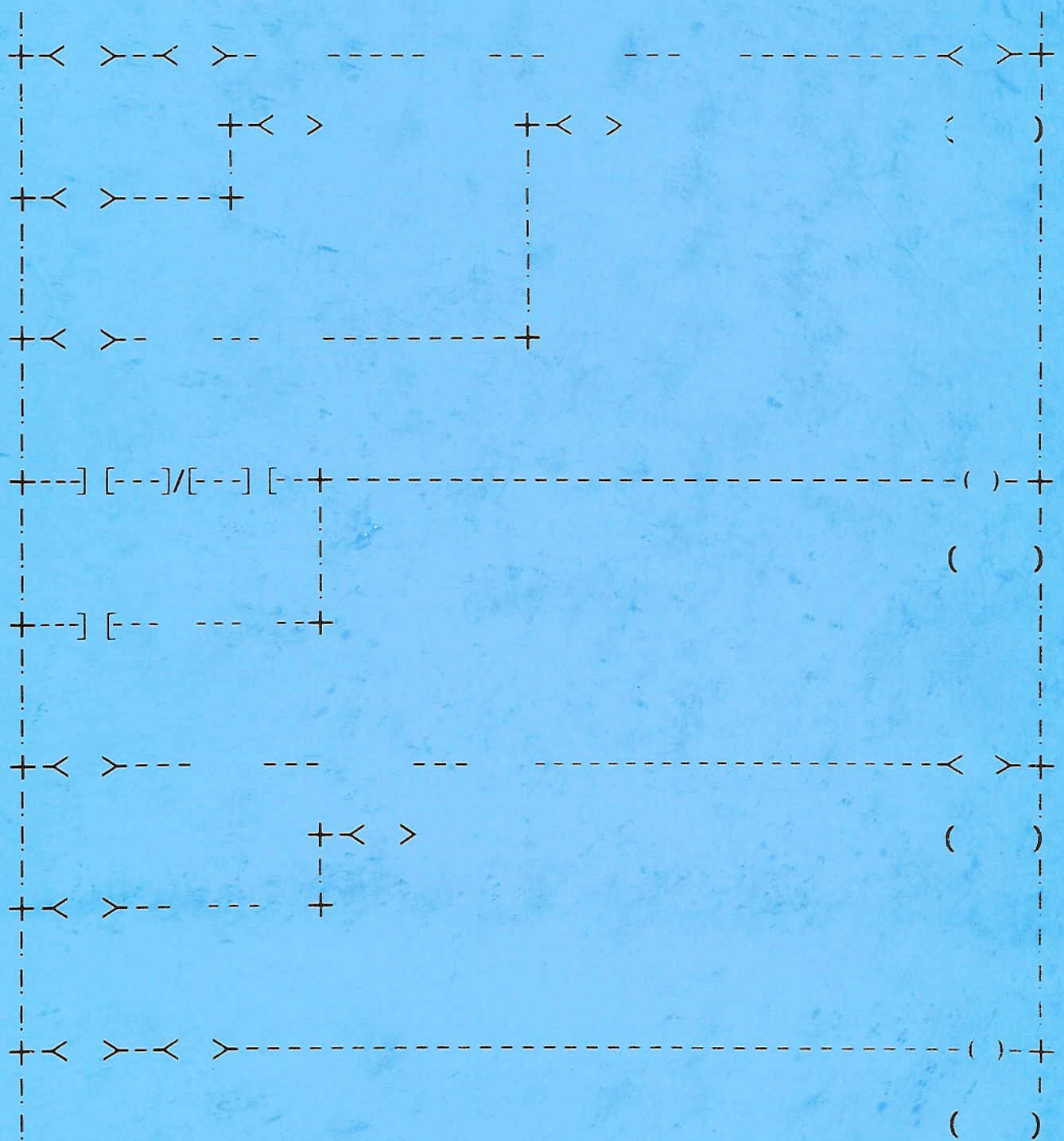
GEC Industrial Controls Limited

GEM 80 Programming Manual



Ladder Diagram Language

GEC

GEM80 Ladder Diagram Language

Programming Manual

Publication No. T391

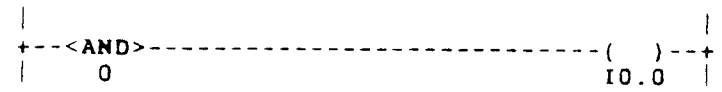
GEM80 LADDER DIAGRAM LANGUAGE - Programming Manual

T391 Issue 4 01.90

Please incorporate the following amendment:-

Page 102

Delete the following rung:-



Please retain this Errata Sheet with the document for reference.

GEM 80 CONTROL LANGUAGE PROGRAMMING REFERENCE MANUAL

FOR USE WITH VDU TYPE PROGRAMMING UNITS

CONTENTS

GEC Industrial Controls Ltd
Kidsgrove, Stoke-on-Trent, Staffordshire ST7 1TW, England

Telephone 0782 783511 Telex 36293 GECICK G

(International 44 782 783511) Fax (Group 3) 0782 776329

Publication T391 Issue 4 01.90

PURPOSE OF THIS MANUAL

This manual is designed primarily for the user who already has some familiarity with programming GEM 80 Controllers, either from having attended GEM 80 Training Courses, or from previous experience.

It therefore attempts to give you:

1. Complete information on the standard elements from which you can construct ladder diagrams, and details of their operation. These are arranged in alphabetical order for easy access. The manual goes into more detail than is possible on GEM 80 Training Courses, and should be able to answer any detail questions that you may have on ladder diagram programming.

2. General guidance on good programming practice, and on how to write programs more efficiently, both as regards improving Controller response time and as regards getting bigger programs into the available memory within the Controller.

One thing that this manual does *not* cover is how to operate a GEM 80 Programmer. This is because there is a range of GEM 80 Programmers of different prices, giving different facilities. However, regardless of what type of Programmer unit you have, you can construct ladder diagrams in the same way from the same elements across *the whole range* of GEM 80 Controllers, from the smallest to the largest. The information appearing in this manual is therefore applicable to *any* GEM 80 Controller programmable in ladder diagram format.

OTHER INFORMATION

Besides this manual, you will require information on the particular model of GEM 80 Controller you are programming, and on the particular type of Programmer being used. This information will consist of:

- Technical Manual for the Controller
- Technical Manual for the Programmer
- Software Data Sheets for all Special Functions available in the Controller

(Note: Some earlier models of Controller have Product Data Sheets and User Information Leaflets instead of Technical Manuals. Where in this manual we refer to a "Technical Manual", you should therefore take this to mean the technical information for the particular model of Controller in whatever form it is published).

THIS PAGE LEFT INTENTIONALLY BLANK

INTRODUCTORY SECTION

CONTENTS PAGE

THIS PAGE LEFT INTENTIONALLY BLANK

RESPONSIBILITY

If you are the system designer, it is ultimately your responsibility if there are any accidents causing injury to personnel or damage to the plant being controlled.

We try to make our Controllers *reliable*; that means that they go wrong very infrequently. But we *don't* (and neither do our competitors) guarantee that our Controllers will *never* go wrong; we would all be fools if we did.

YOUR PROGRAM

What you write in your program has nothing to do with safety. You can put in rungs to take particular courses of action when certain error conditions arise, but these will only work as long as the Controller is up and running correctly. For those conditions where the Controller maloperates, the only way to achieve a safe system is *by hardware*, not software.

HARD WIRED CIRCUITS

Safety circuits must be hard wired and independent of the Controller. The watchdog relay contacts from the Controller must be wired into such safety circuits, to trip them if the Controller itself trips out. All safety circuits must still be operational when the Controller is powered down or tripped out.

This manual is not the place to teach you how to design fail-safe circuits; we expect a qualified system designer to know this, as the principles have been around for decades before programmable controllers came on the scene. Our Customer Support people may be able to give you advice, but, as we said at the start of this section, the responsibility for safety is ultimately your own.

A FEW OBVIOUS POINTS

When a GEM 80 Controller is first powered up, output modules and units can give spurious outputs until the watchdog relay has pulled in (during which time the Controller is doing its start-up checks). It is therefore essential that the plant side power supplies are not switched onto the output modules and units until *after* the watchdog relay has pulled in. An example of how the circuit might be arranged is given in the Technical Manual for every model of GEM 80 Controller.

Some types of output module have a facility for suppressing the output by an electrical signal. This can be very convenient where you have several outputs operating at different voltages, as it saves on the number of contacts you might otherwise require. However, this electrical suppression of outputs again relies on electronic circuitry working correctly. Where you have outputs that, if spuriously switched on, could cause an unsafe condition, then you *must* remove the power to them with a contact of a relay. You should therefore use an auxiliary relay operated by the watchdog with a sufficient quantity of contacts to break all the supplies to output modules that must be off for safety.

If there is a power cut, GEM 80 Controllers will shut down in a controlled fashion, and will start themselves up again when the power comes back on. If, because moving machinery is being controlled, you don't want the machinery to start moving again without intervention by the operator, then you must arrange for the plant side power supplies not to be applied until some external relay circuit has been operated.

Normal start and stop push buttons will generally be connected via input modules and units, but those push buttons and control switches that must stop equipment moving for safety reasons *must* be hard wired. For instance, any switches or push buttons that must guarantee that the equipment doesn't move while maintenance personnel are working on it must *not* depend on the Controller being operational, and must be hard wired. Don't connect them into ordinary input modules and units, or worse still, don't send such command signals down serial links between two Controllers. Use a chain of normally-closed pushbuttons of the stop lock-off type for emergency stop circuits, mounted locally near moving equipment. Any maintenance man must have confidence that, once he

has pushed down his local button, no-one can restart the equipment again until he has reset his local button.

LADDER DIAGRAM PROGRAMS

Most GEM 80 Controllers are designed to be programmed in "GEM 80 Control Language". This uses the ubiquitous ladder diagram approach, allowing you to follow the workings of your program via displays and print-outs in diagramatic form. These displays and print-outs closely resemble relay and switch contact circuit diagrams. However, GEM 80 Control Language cleverly combines the ability to handle ON/OFF signals with that of handling numbers. It includes pre-programmed Special Functions for many applications such as printing text and the control of closed-loop systems.

You will find a list of the Special Functions available in your particular model of Controller in its Technical Manual.

PROGRAM CYCLE

All Controllers using GEM 80 Control Language, the details of which are given in the following pages, operate on a repetitive cycle, servicing all input and output signals at regular time intervals. The language is therefore geared to real-time control, and programs cannot get themselves into endless loops or other types of hang-up as can happen with poorly programmed systems using BASIC or other computer-type languages.

Note that, as this manual is GEM 80 Controller oriented, the terms "input" and "output" are used to mean "input to the GEM 80 Controller" and "output from the GEM 80 Controller".

For older models of GEM 80 Controller, the repetitive program cycle operates as follows:

- Controller reads the states and values of all input signals, and stores these in memory.
- Controller goes through your ladder diagram program, working out what the output signals should be, and stores these states and values in memory.
- When the end of the program is reached, the Controller outputs the states and values now in memory to the output interface modules and units. All these modules and units have their own internal memory latches. They therefore remember the desired outputs, which remain set until the next time that they are serviced by the Controller.
- Controller goes back to the start of the cycle, which then repeats.

For newer models of GEM 80 Controller, the repetitive program cycle is similar, but the work load is shared out among two or more processors. For instance, when a "Ladder Processor" reaches the end of its program, it can start on the next execution of the program almost straight away, since the task of setting the output devices is handled by one or more separate "I/O Processors". As far as you are concerned, however, you don't need to be aware of how the tasks are shared out, and you can still think of the repetitive program cycle as operating as shown above.

MEMORY STRUCTURE

Any GEM 80 Controller contains some memory in which it stores the instructions of your ladder diagram program. Other memory stores preset values used within the program. However, you will see from the description of the program cycle above that yet further memory is used for remembering the states of inputs and the states of the required outputs.

Apart from the memory used for instructions, the remainder of the memory is arranged as *tables*. Each table is defined by a letter, e.g. A-table, D-table, G-table, etc.

Within any of these tables, data is stored as 16-bit words. Each word is held in a *table location*, and the quantity of table locations in any model of GEM 80 Controller is given in its Technical Manual.

To refer to any particular table location, you must quote its *address*. This consists of the table letter followed by a number, e.g. A15, D92, G345, etc.

However, you may also want to refer to an individual bit within the 16 that make up the data in a table location. To do this, you must quote:

- the table location address, followed by:
- a point (full stop or period), followed by:
- a number in the range 0 to 15.

e.g. A15.4, D92.11, G345.0, etc. Such references are generally referred to as *bit addresses*.

(Note that, in common with most computer based equipments, address numbers start from 0 rather than from 1. Not only is the bottom bit in a 16-bit word called Bit 0, but also the bottom table location in any table has address 0, e.g. A0, D0, G0, etc.).

The uses of the various tables within GEM 80 Controllers are given below. Not all these tables appear in any one type of Controller.

A-table : Input data from Basic I/O modules and units.
B-table : Output data to Basic I/O modules and units.
C-table : Input data from Fast I/O modules.
D-table : Output data to Fast I/O modules.
E-table : Timing markers, overflow flags, time values, etc.
F-table : Fault codes and values.
G-table : General internal workspace for user.
H-table : Extension of G-table.
I-table : Data for Serial Link control, or Input data from own I/O on slave controllers.
J-table : Input data from Serial Links.
K-table : Output data to Serial Links.
L-table : Video format control data.
M-table : Maintenance test mode data.
N-table : Extension of G-table.
O-table : Extension of G-table, or Output data to own I/O on slave controllers.
P-table : Preset data for configuration, user constants and messages.
Q-table : Extension of G-table. On some models of Controller, retained floating point data table.
R-table : Retained data table.
S-table* : Single rung input Special Function list on older models of Controller.
Programmer selection, RAM/EPROM selection, etc. on newer models of Controller.
T-table* : Twin rung input Special Function list on older models of Controller.
Not used on newer models of Controller.
U-table : Extension of G-table.
V-table* : Controller data and pointers.
W-table : Retained internal workspace for user.

* Tables whose data is fixed for a particular model of Controller and Software Issue, or determined by the Controller itself. They may be displayed in a data list but must not be used in rungs in a program.

TABLE CHARACTERISTICS

Different tables have different attributes. For the same table letter, the attributes may differ between different models of Controller. Refer to the Technical Manual for your particular model for details of the tables included in it.

Write-enabled/Write-protected

The data in write-protected tables cannot be changed by your program while the Controller is running. If you wish to start your Controller running with certain preset values but to modify them later on, then you may need to copy these values from a write-protected table into a write-enabled one by instructions in your program.

Retained/Non-retained

The data in some tables is retained. After powering down, this data will still be there in memory when the Controller is powered up again. The data in other tables is lost if the Controller is powered down.

Most Controllers also let you define whether the data in certain tables is to be cleared or retained when you command the Controller to run after it was halted with power still on.

Video Access

On models of Controller that include video processors, the video processor cannot necessarily access data from all the tables. You cannot make use of data from those tables not accessible by the video processor in a video display format.

Choice

Given the different table attributes, you must choose the appropriate tables to use for each item of data utilized in your program.

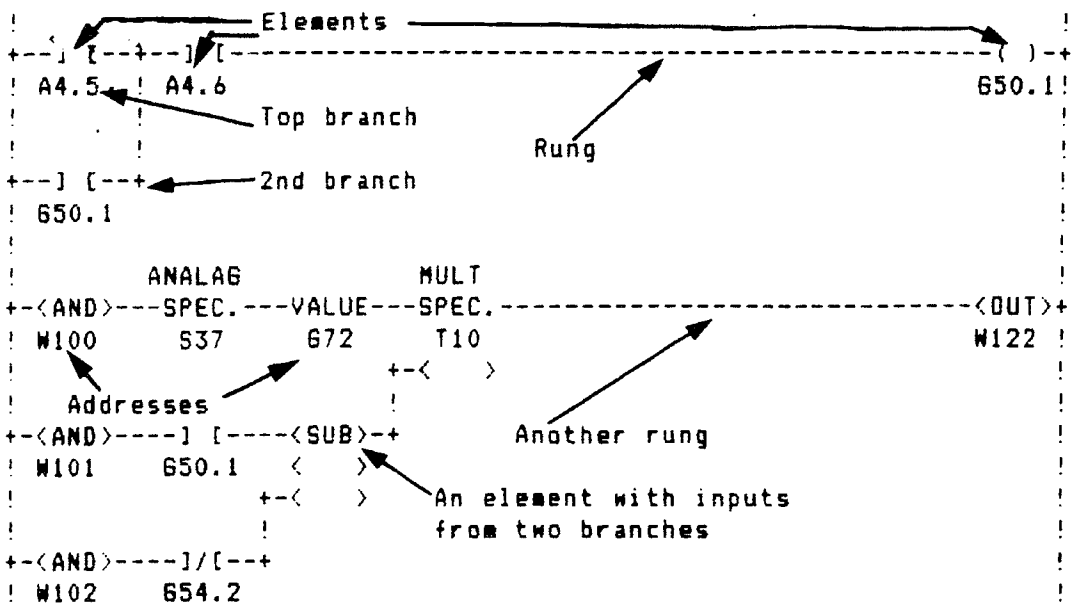
THIS PAGE LEFT INTENTIONALLY BLANK

A FEW DEFINITIONS

A ladder diagram consists of a number of *rungs*. Each rung consists of a number of *elements*, the details of which are given in the Reference Section (Section 2) of this manual.

You can key in these elements from a Programmer keyboard, in most cases by single key-strokes, and they appear on the Programmer screen. You can position the elements by cursor movement, but they will automatically fall on a grid 10 elements wide and 5 elements high.

If a rung contains elements placed in other levels of the grid besides the top row, the rung is said to have *branches*.



The rightmost element of the top branch of any rung must be either a COIL, an OUT, a SEQR or an OBEY BLOCK. Only one element out of these four types is allowed per rung, and this element must always be in the top right position in the grid.

Most elements require an *address* to define where in the GEM 80 Controller memory the data is located that determines the operation of the element or, with an output element, where the result of a rung is to be stored. Depending on the type of element, the address must either define the location of a single bit or of a 16-bit word.

The size of a program is determined by the quantity of *instructions* it takes. One instruction, in a GEM 80 Controller, means an element *and* any associated address (they are not counted separately, as is done with P.C.'s manufactured by some of our competitors).

The Technical Manual for your particular model of Controller will tell you how many instructions and how many data table locations are available. There is usually a trade off between the quantity of instructions and the quantity of preset (P-table) values you can program.

SIGNAL FLOW

The signals that notionally pass from left to right along the branches of rungs consist of *16 bits* that are said to make up 16-bit *words*

For rungs performing relay-type logic only (i.e. using only relay-type elements such as contacts, coils, delays, etc.) the 16 bits in any word passing along the rungs will either be all 1's or all 0's. The left hand vertical line of a ladder diagram, equivalent to the relay supply in relay circuits, provides a word at the start of any branch connected to it consisting of 16 bits all set to 1.

For rungs that include arithmetic elements such as ADD and SUB, the word passing along a rung will usually have its 16 bits not all the same, and will represent a number. Any relay contact elements in these rungs will either allow such numbers to pass along the rung, or stop them passing. When the contact stops them, the word to the right of the contact element will be zero, i.e. a word consisting of 16 0's. The input to a branch from the left hand vertical line of the ladder diagram consisting of 16 bits all set to 1 represents the number -1 (see next item, "Decimal Notation").

Other elements that you can use in the rungs of ladder diagrams are the logic elements AND, OR, XOR and INVert. These elements operate on the 16 bits in words individually, effectively performing 16 parallel bit-level operations. Although you could use them to provide the equivalent of hard wired logic if the words they operated on consisted of all 1's or all 0's (in a similar fashion to relay-circuit rungs), they are generally made use of for their ability to handle several different bits at once. One rung using logic elements can often do as much as several rungs that use relay-type elements.

With some newer Controllers, the number that notionally passes along a rung can on occasions be a 32-bit floating point number, or in other words a pair of 16-bit words. The details of elements given in Section 2 tell you how they behave for both 16-bit and 32-bit numbers.

BINARY

All computers perform arithmetic internally in binary, even though they give outputs as decimal numbers. In a similar way, GEM 80 Controllers perform arithmetic internally in binary, although you can monitor and enter numbers in decimal form.

The 16 bits in a word in a GEM 80 Controller, whether in a word stored in a table in memory or in a word passing along a rung in a ladder diagram, are weighted in powers of 2. That is:

bit 0 represents 1,
bit 1 represents 2,
bit 2 represents 4,
bit 3 represents 8, etc.,
or, mathematically:
bit n represents 2 to the nth power.

The only exception is bit 15. This represents not 2 to the 15th power (32,768) but -32,768. Bit 15 is often referred to as the "sign bit". If bit 15 = 1, then the number represented will always be negative, whereas if bit 15 = 0, the number represented will either be positive or zero.

The maximum positive number that can be represented is 32,767, which has the bit pattern 0111111111111111.

The most negative number that can be represented is -32,768, which has the bit pattern 1000000000000000.

The bit pattern 1111111111111111, being the sum of the two, represents -1. -1 is therefore the decimal representation of the signal at the start of a rung branch provided by the left hand vertical line in a ladder diagram.

EXAMPLES

We take a couple of examples, one a positive and one a negative number.

1st example : 0110100110100101

2nd example : 1001101001010110

Bit 0 :	1 x 1 =	1	1 x 0 =	0
Bit 1 :	2 x 0 =	0	2 x 1 =	2
Bit 2 :	4 x 1 =	4	4 x 1 =	4
Bit 3 :	8 x 0 =	0	8 x 0 =	0
Bit 4 :	16 x 0 =	0	16 x 1 =	16
Bit 5 :	32 x 1 =	32	32 x 0 =	0
Bit 6 :	64 x 0 =	0	64 x 1 =	64
Bit 7 :	128 x 1 =	128	128 x 0 =	0
Bit 8 :	256 x 1 =	256	256 x 0 =	0
Bit 9 :	512 x 0 =	0	512 x 1 =	512
Bit 10 :	1024 x 0 =	0	1024 x 0 =	0
Bit 11 :	2048 x 1 =	2048	2048 x 1 =	2048
Bit 12 :	4096 x 0 =	0	4096 x 1 =	4096
Bit 13 :	8192 x 1 =	8192	8192 x 0 =	0
Bit 14 :	16384 x 1 =	16384	16384 x 0 =	0
Bit 15 :	-32768 x 0 =	0	-32768 x 1 =	-32768
		25045		-26026

Thus, the first example represents 25,045 and the second example represents -26,026.

The above examples may seem a little tedious to work out. Fortunately, in practice the GEM 80 Controller does the conversion for you, so it is very rarely that you need to do such calculations yourself.

FLOATING POINT NUMBERS

Some newer models of GEM 80 Controller include provision for floating point numbers. Each floating point number uses 32-bits, or two words, and only the Q-table can store these numbers. However, inputs and outputs from a GEM 80 controller are normally 16-bit only. Some of the rung elements described in Section 2 are therefore designed to operate with either floating point or integer numbers.

On most GEM 80 Controllers, however, there is no provision for handling floating point numbers in ladder diagrams. All the numbers passing as signals along the rungs in a ladder diagram on these Controllers consist of 16-bit words as given above, and are therefore integers in the range -32,768 to +32,767.

However, in order to provide results of sufficient accuracy for control purposes, many of the Special Functions work *internally* in floating point arithmetic, and round the answer to the nearest integer.

Also, those Special Functions that provide output to printers or VDU's allow numbers to be printed or displayed in formats that can include decimal points. There are also provisions in the specification of video formats, on Controllers incorporating video processors, for numbers to be represented in a variety of ways.

NOTATION

Some people are initially frightened by the word "hexadecimal", as it sounds highly technical. It isn't. It is just a very handy shorthand notation for representing a lot of 0's and 1's with only a quarter as many characters.

Since GEM 80 Controllers work with 16-bit words, you often want to know, or else you want to define, the states of all 16 bits. If you do this in 0's and 1's, you would have to read or to key-in something like:

0110100110100101

Try doing that frequently without making an error! Also, if all numbers were represented this way on a Programmer screen, there wouldn't be much room left for anything else.

GROUPS OF FOUR

With hexadecimal notation, what you first do is to separate the 0's and 1's into groups of four. Thus, you would split up the binary number above as:

0110 1001 1010 0101

Within each group, the 4 bits are weighted 8,4,2,1. Thus, the first group of four above is represented by the digit 6; the second group of four is represented by 9; and the last group of four is represented by 5.

However, the third group of four, totalling the weightings, comes to 10, which is not a single character. What we do in hexadecimal is to use the letters A to F for the values 10 to 15 respectively. (We give a table below). The third group of four bits is therefore represented by A.

Hence, in hexadecimal notation, the 16 0's-and-1's we started with are represented by:

69A5

A hexadecimal number does not necessarily contain any letters. As we might therefore get it confused with decimal numbers, we prefix it with a "commercial at", and write it as:

@69A5

That is all there is to hexadecimal - it's just a shorthand. But, like shorthand, if you want to be any good at it, you should learn to do it fast. That is, you should learn the skill to convert 0's and 1's to hexadecimal and to convert hexadecimal back into 0's and 1's rapidly, without stopping to think about it.

HEXADECIMAL CODE TABLE

0000	=	@0	0100	=	@4	1000	=	@8	1100	=	@C
0001	=	@1	0101	=	@5	1001	=	@9	1101	=	@D
0010	=	@2	0110	=	@6	1010	=	@A	1110	=	@E
0011	=	@3	0111	=	@7	1011	=	@B	1111	=	@F

THIS PAGE LEFT INTENTIONALLY BLANK

REFERENCE SECTION

CONTENTS PAGE

RUNG ELEMENTS

THIS PAGE LEFT INTENTIONALLY BLANK

FUNCTION

To add two numbers together, each number being a signed integer in the range -32,768 to +32,767, or a floating point number in the range roughly $\pm 10^{38}$ to $\pm 10^{-38}$.

FORMAT

```
-<ADD>-        or        -<ADD>-
   Y                      <   >
                         -<   >
```

where:

Y = table address of any table location, or numeric constant.

EXAMPLES

```
(1)  -<ADD>-
       150

(2)  -<ADD>-
       619

(3)  -<ADD>-
      <   >
     -<   >
```

OPERATION

In all three examples, the rung input has a second number added to it, which is either specified as:

- a constant below the ADD (Example 1),
- taken from a table (Example 2), or
- taken from a second branch input (Example 3).

Numbers may be positive or negative, and integer or floating point. The allowable number range is -32,768 to +32,767 for integer, and roughly $\pm 10^{38}$ to $\pm 10^{-38}$ in floating point.

For integers only, the rung output is limited to numbers in the range -32,767 (not -32,768) to +32,767. See the Application Notes below for the result of an ADD coming outside the permissible number range.

KEYBOARD ENTRY FROM PROGRAMMER

Press the following sequence of keys:

ADD key (Beware! *Not* the AND key!)
Numeric value, terminated by SPACE, or else
table address, terminated by SPACE, or else
JOIN (for second branch input).

PROGRAMMING RULES

1. ADD cannot be the first element keyed into a rung.

2. When ADD is used as in format example (1) above, the number below the ADD must be a decimal integer, and cannot be in hexadecimal or in floating point.

3. When two integers are added together, the result will be an integer. If either of the numbers is floating point, the result will be floating point.

MONITORING

When monitoring, if the second number is integer and comes from a table location, as in Example 2, then the value of this number will be displayed above ADD, e.g.

```
1357
-<ADD>-
 G19
```

The result of the addition is not displayed.

APPLICATION NOTES

1. The result of an addition between two integers could be a number either more positive than +32,767 or more negative than -32,767. In such cases, the rung output from an ADD will be set to +32,767 or -32,767 as appropriate and the excess ignored.

   If the rung input is -32,768, and if 0 is ADDed to it, the rung output will be -32,767. The same will happen if the rung input is 0 and -32,768 is ADDed to it.

2. When such an overflow occurs, the single bit E0.7 will be set to 1. If overflow represents an abnormal condition in the system operation, you can take appropriate action by putting rungs in your program that utilize this bit.

3. E0.7 will be set back to 0 when a further ADD or SUB is encountered in the program, the result of which does not cause overflow.

4. To prevent overflow, the order in which you perform a number of separate integer calculations can be important.

   For example, the expressions (a + b - c) and (a - c + b) are algebraically equivalent. However, if a, b and c are all positive, then the first expression could cause overflow from the initial addition (a + b). On the other hand, the initial subtraction (a - c) in the second expression would give a smaller or negative number that could not overflow.

   Where you design a rung performing integer arithmetic involving ADD, SUB and the Special Functions MULT and DIV, you should check your design for what will happen at intermediate points in the rung when large numbers are used against possible overflow.

5. You may enter ADD at the start of a rung, if you precede it by LINK, to give subtraction of 1, since the left hand vertical line corresponds to -1 in numeric terms. The LINK will be removed if you subsequently recall the rung for display; see sheet for LINK.

   Thus:

```
+-<ADD>-
  W345
```

   is equivalent to

```
+-<AND>---<SUB>-
  W345       1
```

but saving one instruction. You can use this technique where you require to decrement a value on each execution of a program or BLOCK. Note, however, that a rung beginning in this way is very easily misread, i.e. the ADD at the start of the rung is easily misread as an AND, and so you should avoid this "trick" unless running short of memory space for instructions.

THIS PAGE LEFT INTENTIONALLY BLANK

FUNCTION

Provides an output whose 16 bits are the result of logically ANDing the 16 rung input bits individually with the 16 bits of a numeric constant or table value.

The two most common uses of AND are:

1. To plant a number, or a value from a table, into the start of a rung branch.

2. For "masking", i.e. only allowing certain specified bits to pass along a rung.

FORMAT

```
-<AND>-        or        -<AND>-
   Y                      <   >
                         -<   >
```

where:

Y = table address of any table location, or numeric constant

EXAMPLES

```
(1)  -<AND>-
       @FF

(2)  -<AND>-
       G24

(3)  -<AND>-
      <   >
     -<   >
```

OPERATION

In all three examples, the rung input is ANDed with a second number, which is either specified as:

- a constant below the AND (Example 1),
- taken from a table (Example 2), or
- taken from a second branch input (Example 3).

The truth table below of a single bit shows that there will be a bit set to 1 in the output from the AND only if there is a 1 in the rung input and a corresponding 1 in the second number:

Rung input	2nd number	AND output
0	0	0
0	1	0
1	0	0
1	1	1

Example of ANDing two 16-bit numbers:

```
1001011010100101   Rung input
0000000011111111   2nd number
0000000010100101   AND output
```

KEYBOARD ENTRY FROM PROGRAMMER

Press the following sequence of keys:

AND key (Beware! *Not* the ADD key!)
Numeric value, terminated by SPACE, or else
table address, terminated by SPACE, or else
JOIN (for second branch input).

PROGRAMMING RULES

When AND is used as in format example (1) above, the number below the AND may be either a decimal integer or in hexadecimal.

MONITORING

When monitoring, if the second number is integer and comes from a table location, as in Example 2, then the value of this number will be displayed above AND. If you monitor with the display in hexadecimal, this might appear as below:

```
@1357
-<AND>-
 G19
```

The result of the AND operation is not displayed.

APPLICATION NOTES

1. AND is meant for dealing with bits, not numeric variables. Its use with decimal numbers is generally meaningless or at least confusing. For instance, the example given in "Operation" above would, in decimal, be -26971 AND 255 equals 85.

2. The only case where you will find AND useful with numbers is to plant a number into the start of a rung, e.g.:

```
!
+-<AND>-----
!  100
```

   This works because the vertical line at the left hand side of a ladder diagram corresponds to 16 bits that are all 1. When ANDed with any number, this will cause the number itself to be passed along the rung.

3. You can similarly use AND to plant a number into the middle of a rung, but only where all functions to the left of it are contacts that pass or block all 16 1's from the left hand vertical.

4. With floating point numbers, you can use AND to plant a number into the start of a rung branch from the left hand vertical line, but nowhere else. Any attempt to use AND with floating point numbers elsewhere will give you a compiler error message, and your program won't run.

5. AND is most frequently used, as a logic operation, for "Masking", to allow only a defined set of bits to pass along a rung, and to block the rest. In the example given in "Operation" above, you can consider the second number as a "Mask" that allows the right hand eight bits to pass unchanged, but "masks out" the left hand eight bits.

6. To enter a mask, it is convenient to define it in hexadecimal. Thus, in format example (1) above, the mask is given as @FF, which is equivalent to 0000000011111111 in binary. With masking operations, it is also best to monitor with numbers displayed in hexadecimal.

7. Common masks that you often need are:

@FF	:	Masks out top eight bits
@FF00	:	Masks out bottom eight bits
@8000	:	Masks out everything except bit 15 (bit 15 is the sign bit with numeric values)
@7FFF	:	Masks out bit 15.

8. For rungs that begin with AND followed by INV, see application note for XOR on how to save one instruction.

THIS PAGE LEFT INTENTIONALLY BLANK

FUNCTION

To define the start and end of a set of rungs in a program that are to be obeyed only if a given condition is met. If the condition is not met, then the set of rungs will be skipped, and program execution will continue at the next rung after the set of rungs.

FORMAT

```
(1)
                                                                  -OBEY-+
                                                                   BLOCK
                                                                     *
     ********************* START OF BLOCK 7 ********************

(2)

     + ********************* END OF BLOCK 7 ************************ +
```

EXAMPLE

```
+--] [-----------------------------------------------------------OBEY-+
 A12.14                                                            BLOCK
                                                                     *
                                                                     *
  ************************ START OF BLOCK 4 ********************
```

OPERATION

The block of program rungs following an OBEY BLOCK up to the corresponding END OF BLOCK will only be obeyed when the rung input to the OBEY is ON (i.e. any non-zero integer). If the rung input is OFF (i.e. the number zero), the block of rungs will be skipped, and program execution will continue at the next rung after the END OF BLOCK.

In the Example above, when A12.14 is ON, making the N/O contact closed, the block of lines following this rung will be obeyed. If A12.14 is OFF, the block of lines following will not be obeyed, but program operation will skip to the next rung after the end of the block. The end of the block is defined by a rung as in Format (2) above.

KEYBOARD ENTRY

To start a block, begin a rung with the elements that determine the condition under which the block is to be obeyed, then press BLOCK. This will automatically cause the OBEY BLOCK caption to appear at the end of the rung, with a line drawn from the last of the condition elements. The line of asterisks and the START OF BLOCK caption will also be displayed.

To end a block, simply press BLOCK as the first (and only) element in a rung, and the END OF BLOCK will be displayed as in Format (2) above.

PROGRAMMING RULES

To start a block, you must enter some other element before pressing BLOCK; otherwise, the Programmer will take the pressing of the BLOCK key to mean that you require an end of block.

The number passing along the rung to the OBEY BLOCK must be an integer of 16-bits. If you attempt to operate a BLOCK with a floating point number, then you will get a compiler error message and your program won't run.

A block may be contained within a larger block. Blocks within blocks are said to be "nested". In the same way that, in an algebraic expression containing brackets, there must be a closing bracket for every opening bracket, similarly in programs there must be an END OF BLOCK for every OBEY BLOCK. There is one exception to this on older models of Controller only - see Application Notes below.

The maximum depth to which blocks may be nested in any given GEM 80 Controller is given in its Technical Manual.

MONITORING

When monitored rungs appear to be giving wrong answers, it is usually because they are inside a block not being obeyed at the time.

See sheets for individual elements as to the effect on the display on a Programmer screen when a block is or is not being obeyed for the rungs contained within the block.

PRINT-OUTS

Print-outs produced by Programmers generally include BLOCK No.'s. (Print-outs produced by older versions of Programmer may not). The first OBEY BLOCK encountered will give a print-out of:

```
                                                        -OBEY-+
                                                         BLOCK
                                                           *
********************** START OF BLOCK 1 **********************
```

and the corresponding end of the block will give:

```
+ ********************** END OF BLOCK 1 ************************ +
```

The Programmer will remember the block numbers so that, where blocks are nested, it will provide the correct numbers for corresponding starts and ends of blocks.

APPLICATION NOTES

1. Block numbers shown in the "Format" and "Example" above appear only on print-outs. They do not appear on Programmer screen displays.

2. Modern types of GEM 80 Controller check that every START OF BLOCK has a corresponding END OF BLOCK. If you program in more STARTs than ENDs or vice versa, you will get a compiler error message, and your program will not run.

3. For older models of GEM 80 Controller, although you should generally program an END OF BLOCK for every START OF BLOCK, any superfluous END OF BLOCK that has no related START OF BLOCK will be ignored.

4. With some older types of GEM 80 Controller, you can reduce the program cycle time by putting a START OF BLOCK that is permanently not obeyed (e.g. with the condition AND 0) at the very end of a program, with no corresponding END OF BLOCK.

   As explained above in Note 2, you can't use this method with modern models of GEM 80 Controller. In any case, it wouldn't give any time saving with such Controllers, even if the compiler allowed you to write your program this way.

FUNCTION

To set a single bit in a data table location to 1 if the rung input is ON; otherwise, if the rung input is OFF, the bit will be set to 0.

FORMAT

```
--( )-+
   Y
```

where:

Y = bit address

EXAMPLE

```
--( )-+
B21.15
```

OPERATION

If the rung input to the COIL is ON (i.e. any non-zero integer), then the particular bit given by the address below the coil symbol is set to 1. If the rung input is OFF (i.e. the number on the rung input is zero), then the bit is set to 0.

The remaining 15 bits in the table location are not affected, and remain in whatever state they were previously. Thus, in the example, of the 16 bits in the table location B21, only bit 15 is set; bits 0 to 14 in B21 are unaltered.

You cannot operate a COIL with a floating point number.

KEYBOARD ENTRY

Press the following sequence of keys:

-( )-key (this will automatically place the symbol at the end of the rung, and join it to the rest of the rung by a line)
Bit address,terminated by SPACE.

PROGRAMMING RULES

The address shown as Y in the "Format" above must define a bit, i.e. it must include a point followed by a figure generally between 0 and 15.

This address, however, must not contain more than 7 characters. You can therefore enter a bit address of W999.15, but not W1000.15.

For addresses with 4-digits before the point, the Programmer will automatically switch to expecting a hexadecimal representation for any bit number in the range 10 to 15. For example, you must key in W1000.15 as W1000.F.

A COIL can only be the last element in a rung. It may also only be placed in the top branch of a rung that has parallel branches. Only one COIL can therefore appear in any rung.

A COIL can be the only element in a rung. Such a rung would set the bit defined by the address Y to a fixed value of 1. See also Application Notes below.

A COIL cannot be operated by a floating point number. Any attempt to do so will give you a compiler error message, and your program won't run.

MONITORING

A COIL that is "energized", i.e. with its rung input ON, is displayed:

```
--( )-+
```

A COIL that is "de-energized", i.e. with its rung input OFF, is displayed:

```
..( ).+
```

OPERATION WITHIN BLOCKS

Any COIL contained in a BLOCK that is not being obeyed will remain in whatever state it was in, even though its rung input may be displayed when monitoring as though it were trying to drive it to the opposite state.

PRINT-OUTS

When displayed on the screen of a Programmer, the COIL joins to a vertical line at the right hand side of the screen. When a ladder diagram is printed as a hard copy record, older versions of Programmer omit the right hand vertical line.

APPLICATION NOTES

1. If the COIL sets a bit in a table location of an output table (e.g. B-table), the information is not read from this table to the physical output devices (e.g. output modules, serial links, LCD displays, etc.) until the end of the complete ladder diagram program has been reached.

2. There is no restriction on you including COILs and OUTs more than once in a program. You can set bits or words in the same table location different ways as the program execution proceeds. In that case, the way the bits will be set when they are read out to the output devices depends on what state they are in when the end of the program is reached, even though they may have been altered back and forth several times previously in the program.

3. We do not generally recommend the above practice, however, as it makes programs difficult to understand, and increases the likelihood of you generating errors in the logic of your program.

4. It may be useful during commissioning, however, to insert a rung or rungs at the very end of a program to override the way that a COIL has been set earlier in a program. Such rungs should be removed when the end of commissioning is reached.

   See "Forcing Outputs" in the Item "Testing" in Section 3 of this manual.

5. There is also no restriction on you including COILs in a program that set bits in table locations in input tables (e.g. A-table, C-table). The data in these input tables is normally set at the beginning of each program cycle by the Controller when it reads the data from the input modules and units. If you includes COILs in your program that set bits in input tables, then the input data will be overwritten for these bits.

6. We do not generally recommend the above practice, as it makes programs difficult to understand. You may find it useful during commissioning, however, to insert a rung or rungs at the very start of a program to override the data read in from the plant. See "Forcing Inputs" in the Item "Testing" in Section 3 of this manual. You should remove such rungs when you reach the end of commissioning.

7. It is possible on some older models of Controller to use COILs to write bits into input tables that are not being used (e.g. if output table location B12 is being used, then input table location A12 will not be used for input data). You cannot do this on modern Controllers, as you will get a compilation error message if you try this.

   However, the use of spare input tables on older Controllers is generally not to be recommended. You should only use them in this way where you are short of memory/instruction space. First check your configuration data in P-table to make sure that any such spare input table locations do not get set back to zero at the beginning of a program cycle, if you don't want this to occur.

THIS PAGE LEFT INTENTIONALLY BLANK

FUNCTION

Provides an output after the input has changed from OFF to ON a specified number of times.

FORMAT

```
-COUNT---VALUE-
   Y         Z
-RESET
```

where:

Y = table address of a write-enabled table location
Z = numeric constant or table address

EXAMPLES

```
(1)  -COUNT---VALUE-
       G12      500
     -RESET

(2)  -COUNT---VALUE-
       G12      W40
     -RESET
```

OPERATION

The VALUE gives the number of OFF-to-ON transitions required before COUNT gives an output.

In Example (1), there must be 500 such transitions before there is an output.

In Example (2), the number of transitions to be counted is whatever number is contained in table location W40.

An ON signal to the RESET input stops the count and resets it to zero. RESET overrides any transitions on the normal rung input. When the RESET is removed, the COUNT will start again on the next OFF-to-ON rung input transition. It does not matter if, when the RESET is removed, the normal rung input is OFF or ON; only subsequent transitions from OFF-to-ON are counted.

KEYBOARD ENTRY FROM PROGRAMMER

Press the following sequence of keys:

COUNT key
Table address, terminated by SPACE
VALUE key
Numeric value, terminated by SPACE, or else
table address, terminated by SPACE.

PROGRAMMING RULES

1. COUNT must always be followed by VALUE.

2. The data table location given below the COUNT must be one in write-enabled memory.

3. The maximum number that you can use with COUNT is 32,767 decimal, or @7FFF hexadecimal.

4. If a negative number is given for VALUE then the COUNT will be zero. (The equivalent in hexadecimal is if numbers are in the range between @8000 and @FFFF).

5. COUNT cannot be the first instruction in a rung.

6. The rung inputs to COUNT must be 16-bit integer, as must the number used for the VALUE. If you attempt to use floating point numbers, you will get a compiler error message and your program won't run.

7. For COUNT and DELAY elements used in your program, you must use a separate table location for each element. You will get spurious operation if you use the same table location for more than one COUNT and/or DELAY.

DYNAMIC CHANGE OF VALUE

If the VALUE for the COUNT is read from a table, as in Example (2), and if the number held in this table location changes while the COUNT is operating, then if the value counted up so far is greater than the new VALUE number, the COUNT will immediately set its output ON. If the number held in the table location is larger than the value counted so far, then the COUNT will set its output ON when the new VALUE figure is reached.

DATA TABLE LOCATION USED WITH COUNT

The table location below COUNT (shown as G12 in both Examples 1 and 2 above) is used by COUNT for two purposes:

1. Bits 0 to 14 hold the number of transitions of the rung input from OFF to ON since COUNT was last reset, or since the table itself was last cleared, whichever was the more recent event.

2. Bit 15 is held at 1 if, on the last program scan, the rung input to COUNT was ON, or is held at 0 if the last input was OFF. COUNT makes use of this bit internally when establishing whether there has been an OFF to ON transition of the rung input.

OPERATION WITHIN BLOCKS

COUNTs within BLOCKs only operate while the BLOCK is being obeyed. The number of transitions counted so far will be held when a BLOCK ceases being obeyed, and will start again from this value if the BLOCK is obeyed again subsequently.

MONITORING

Using Example (1) above:

Input to COUNT is OFF, and the number of transitions so far is 498:

```
  498
.COUNT.--VALUE-
  612      500
.RESET
```

Input to COUNT goes ON, giving an OFF to ON transition, and increasing the count by 1:

```
  499
.COUNT.--VALUE-
  612      500
.RESET
```

After the input goes OFF and ON again, the COUNT reaches the VALUE of 500, and provides an output:

```
      500
   -COUNT---VALUE-
     G12      500
   .RESET
```

Any further transitions of the rung input make no change to the figure displayed above COUNT, which remains at 500.

If RESET is switched ON, then the COUNT is zeroed:

```
      0
   .COUNT.--VALUE-
     G12      500
   -RESET
```

APPLICATION NOTES

1. When you monitor a COUNT, the figure displayed above it is equal to the number held in the table location whose address entered below the word COUNT, but ignoring Bit 15. See section above on "Data Table Location used with Count".

2. If you need to make use of the COUNT number held in the data table location defined below COUNT (G12 in the monitoring examples above), then you must mask off Bit 15, e.g.

```
-(AND)---(AND)------
  G12     @7FFF
```

   Refer to the sheet for the AND function if you are unfamiliar with masking.

3. If, however, you monitor the contents of this table address anywhere else in your ladder diagram, or as part of a datalist, Bit 15 will *not* be ignored. The number displayed will therefore alternate between positive and negative numbers as the COUNT is counting up. To obtain a display where the negative numbers are suppressed, Bit 15 should be masked out (as in Note 2 above) and the resultant number put into another table location using OUT.

THIS PAGE LEFT INTENTIONALLY BLANK

FUNCTION

Provides an output after a time delay from the input being switched ON.

FORMAT

```
-DELAY---VALUE-
   Y       Z
```

where:

Y = table address of a write-enabled table location
Z = numeric constant or table address

EXAMPLES

```
(1)  -DELAY---VALUE-
       G11     20

(2)  -DELAY---VALUE-
       G11     G70
```

OPERATION

The VALUE gives the delay in increments that, depending on the model of GEM 80 Controller, will either be 0.1 s or 0.01 s. See the Technical Manual for the particular Controller. In the description that follows, it is assumed that the delay increment for the Controller is 0.1 s unless otherwise stated.

In Example (1), the delay is therefore 2 s.

In Example (2), the delay is whatever number is found in table location G70, in units of 0.1 s. Thus, if G70 contains the number 47, the delay will be 4.7 s.

When the input to the DELAY is OFF, the output from the DELAY is always OFF.

When the input to the DELAY changes to ON (i.e. any non-zero integer), the output of the DELAY only changes to ON after the time specified by the VALUE.

If, while the DELAY is timing out, its input changes back to OFF, then the DELAY is reset. The full delay time will still be used the next time that the input changes to ON.

KEYBOARD ENTRY FROM PROGRAMMER

Press the following sequence of keys:

DELAY key
Table address, terminated by SPACE
VALUE key
Numeric value, terminated by SPACE, or else
table address, terminated by SPACE

PROGRAMMING RULES

1. DELAY must always be followed by VALUE.

2. The data table location given below the DELAY must be one in write-enabled memory.

3. The maximum number that can be used with VALUE is 32,767 decimal, or @7FFF hexadecimal. This gives 3276.7 seconds (approx. 54 minutes) for Controllers with a 0.1 s delay increment, or 327.67 seconds (approx. 5.4 minutes) for Controllers with a 0.01 s delay increment.

4. If a negative number is given for VALUE then the DELAY time will be zero. (The equivalent in hexadecimal is if numbers are in the range between @8000 and @FFFF).

5. You must not use floating point numbers with DELAY. If you attempt to do this, you will get a compiler error message and your program won't run.

6. For COUNT and DELAY elements used in your program, you must use a separate table location for each element. You will get spurious operation if you use the same table location for more than one COUNT and/or DELAY.

DYNAMIC CHANGE OF VALUE

If the VALUE for the DELAY is read from a table, as in Example (2), and if the number held in this table location changes while the DELAY is timing out, no change is made to the delay time; the value used is that which held good at the initiation of the DELAY. The new value will not be used till the next time the DELAY is operated.

OPERATION WITHIN BLOCKS

DELAYs within BLOCKs only operate while the BLOCK is being obeyed. The DELAY count-down freezes when a BLOCK ceases being obeyed, and will start again from this value if the BLOCK is obeyed again subsequently.

MONITORING

Using Example (1) above for a Controller with a 0.1 s delay increment:

Input to DELAY is OFF; hence output also OFF:

```
    0
.DELAY.--VALUE-
  G11      20
```

Input to DELAY is ON; DELAY is timing out, and output is still OFF:

```
   15                    Number above DELAY changes,
.DELAY.--VALUE-          counting down from 20 to 0.
  G11      20            Indicates units of 0.1_s yet to go.
```

Input to DELAY is ON; DELAY has now timed out, and so output is ON:

```
    0
-DELAY---VALUE-
  G11      20
```

OPERATION WHEN SINGLE CYCLING

When a Controller is set to work in Single Cycle mode, the number held in the table location given below DELAY will decrement by 1 each cycle, and will not operate in real time. You can verify this when monitoring.

TIMING ACCURACY

Delay times cannot be to a better resolution than the program cycle time set in the Controller configuration data (or the natural program cycle time, if the Controller is free running).

Delay times will, as a result, tend to be fractionally longer than those set by the VALUE. For instance, suppose that the program cycle time is 0.08 s (=80 ms). The actual delay time can therefore only be some multiple of 0.08 s, such as 0.08 s, 0.16 s, 0.24 s and so on. If you set a delay to 0.3 s, then the first multiple of 0.08 s longer than this is 0.32 s; in practice your 0.3 s delay would therefore become 0.32 s. Similarly, if you set a delay to 0.5 s, then the first multiple of 0.08 s longer than this is 0.56 s: in practice your 0.5 s delay would therefore become 0.56 s.

If the program cycle time exceeds 0.1 s, then the count-down values observable when monitoring will be decremented by more than one on some program cycles, so as to maintain the correct delay times. The resolution of delay times will still be no better than the program cycle time, however.

APPLICATION NOTES

1. When a DELAY output is ON, it gives an output where all 16 bits are 1's (equivalent to -1 in decimal or @FFFF in hexadecimal). You can therefore follow a DELAY-and-VALUE with a logic INVert to obtain an output that will be complementary to that of the DELAY (see sheet for INV).

2. DELAY gives a delay on the input changing from OFF to ON, but no delay on a change from ON to OFF. To get a delay off, use the following:

```
-<INV>---DELAY---VALUE---<INV>-
              Y       Z
```

   where Y and Z are as defined in "Format" above.

3. To get a delay ON and a delay OFF, use the following:

```
-DELAY---VALUE---<INV>---DELAY---VALUE---<INV>-
  Y1      Z1              Y2      Z2
```

   where the first delay gives the delay ON, and the second delay give the delay OFF.

4. To get timing pulses that are ON for a single program cycle, use a delay that resets itself, e.g.:

```
---]/[----DELAY---VALUE-------------( )-
   G14.0     G15      50              G14.0
```

   The above would cause G14.0 to be ON for one program cycle after a delay of 5 s. The DELAY would not be reset till the next program cycle by the contact of G14.0 opening, as the Controller evaluates each program rung only once in any program cycle.

5. If you require times larger than 54 minutes, you can use several techniques. The simplest is to cascade two DELAYs in one rung (this would give up to 1 hour 49 minutes). For longer delays, timing pulses generated as in Application Note 4 above can be fed into a COUNT function.

THIS PAGE LEFT INTENTIONALLY BLANK

FUNCTION

Provides an output whose 16 bits are the inverse of those in the input, i.e. all 0's in the input produce 1's in the output, and all 1's in the input produce 0's in the output.

FORMAT

-<INV>-

OPERATION

As described above under "Function"; e.g. if the input is:

1010010110010110

then the output is:

0101101001101001

If represented in hexadecimal, the equivalent is:

If the input is: @A596

Then the output is: @5A69

where you will see that each hexadecimal digit in the output is 15 minus the corresponding hexadecimal digit in the input (A being the hex. equivalent of 10).

Note that INV works on individual bits. If you consider a rung input that is ON (non-zero) or OFF (zero), then, although INV operating on OFF gives the output of ON, INV operating on ON does not necessarily give an output of OFF. In the example given above of the operation of INV, neither the input nor the output of INV are zero, and therefore both the input and the output are ON. INV operating on ON will only give an output of OFF if all 16 bits in the input are 1's.

KEYBOARD ENTRY FROM PROGRAMMER

There is no single key for INV; press the following sequence:

LOGIC key
Letter I

PROGRAMMING RULES

You cannot use INV with floating point numbers. Any attempt to do so will result in a compiler error message, and your program won't run.

APPLICATION NOTES

INV is intended for logic operations, not for numbers. However, if you do use INV on a number, the result will be the negative of the number -1, e.g. the inverse of 3 will be -4, and vice versa.

THIS PAGE LEFT INTENTIONALLY BLANK

FUNCTION

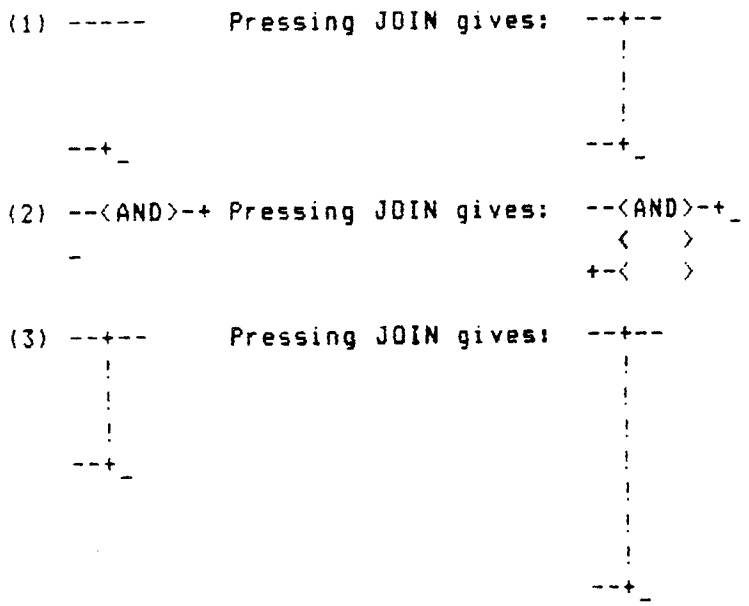
There are three different functions given by JOIN:

1. To close a branch.
2. To declare that a logic function (AND, OR or XOR) or ADD or SUB will have its second input from a rung branch.
3. To vertically space out the elements displayed on a programmer screen, allowing you to insert extra branches.

FORMAT

Varies, depending on the cursor position when you press JOIN. See Examples below.

EXAMPLES



OPERATION

In Example 1, a lower branch of a rung is incomplete, with the cursor displayed to the right of its end. Pressing JOIN causes the vertical line to join the lower branch to the one above.

In Example 2, a logic AND is the last element that you have keyed in while constructing a branch of a rung. The cursor has moved below the AND, allowing you to enter a constant or table location address, if you so desire. However, if you instead press JOIN, the Programmer will take this as an instruction that the value to be ANDed will come from another branch in the rung. It prepares the screen display for this by displaying the start of the second input as shown.

In Example 3, you have previously completed the lower branch shown with a JOIN to the upper branch. Pressing JOIN with the cursor positioned as shown causes the lower branch to be moved downwards by one vertical increment, and the display will be re-drawn as shown.

APPLICATION NOTES

1. JOIN used to close a branch takes 1 instruction.

2. As the screen of a Programmer allows a maximum of 5 parallel branches (limited by the height of the screen), you cannot use JOIN for Functions 2 or 3 with the cursor at bottom branch level.

3. After editing, if a rung contains superfluous JOINs just to space the rung branches out vertically, then these superfluous JOINs will be removed when you recall the rung for display at a later time.

4. Where you use a JOIN to close a branch to a higher branch in a rung, the rung output to the right of the join is effectively a logical OR function on the inputs from the two branches. It is sometimes referred to as a "Wired OR" function from analogy to hardware logic design. See page 55 for OR.

5. You do not need to use JOIN to start a new branch half way along a rung.

6. You must not use JOIN as in Example 1 if either of the branches being joined uses floating point numbers. If you attempt to do so, you will get a compiler error message, and your program won't run. You can only merge two branches that use floating point numbers using 2-input elements or Special Functions whose specifications state that they are suitable for floating point inputs.

FUNCTION

To space out elements displayed on a Programmer screen.

FORMAT

OPERATION

If you press LINK at the end of an incomplete branch of a rung, the line will be extended by a length equivalent to one element. Note that you don't need to press LINK to complete the top branch of a rung, as the rest of the line will automatically be drawn when you press the key for the output element (BLOCK, COIL, OUT or SEQR).

When you edit a rung, if you position the cursor at the beginning of an element, then pressing LINK will cause all elements to the right of the cursor to move one element to the right (in all branches of the rung), allowing you to insert an extra element. (This is assuming, of course, that there is space for an extra element).

APPLICATION NOTES

1. A LINK does not take up any instructions within the GEM 80 Controller.

2. A rung that you have edited or written in such a way that it contains superfluous LINKs will, when you recall it for display at a later time, have these superfluous LINKs removed, and be compacted towards the left.

3. You can use a LINK at the start of a rung to allow entry of functions that the Programmer will not accept connected to the left hand vertical line; you can then enter such functions as the second element of the rung. When you recall the rung for display, the LINK will be removed, as in Note 2 above. You should use this "trick" with care, as it could lead to peculiar program operation.

THIS PAGE LEFT INTENTIONALLY BLANK

FUNCTION

The LOGIC key is used in conjunction with I, O or X to provide the logic functions of inversion, inclusive-or or exclusive-or.

FORMAT

See INV, OR and XOR.

PROGRAMMING RULES

After you have pressed the LOGIC key, you may only enter I, O or X (apart from RUB OUT to cancel it).

NOTES

Logical AND has its own dedicated key. You mustn't press the LOGIC key before AND.

THIS PAGE LEFT INTENTIONALLY BLANK

FUNCTION

To transmit a number along a rung when a particular bit is set OFF in data tables; to give zero out when this bit is set ON.

FORMAT

```
--]/[--
   Y
```

where:

Y = table address of the bit controlling the contact.

EXAMPLE

```
--]/[--
A27.14
```

OPERATION

In the Example, when bit A27.14 is OFF, then whatever number appears on the left of the contact symbol will be transmitted to the right hand side of the symbol. If A27.14 is ON, then the number appearing at the right hand side of the symbol will be zero.

N/C contacts work with both integer and floating point numbers as rung inputs.

KEYBOARD ENTRY FROM PROGRAMMER

Press the following sequence of keys:

-]/[Key
Bit address, terminated by SPACE

PROGRAMMING RULES

The address shown as Y in the "Format" above must define a bit, i.e. it must include a point followed by a figure generally between 0 and 15.

This address, however, must not contain more than 7 characters. You can therefore enter a bit address of W999.15, but not W1000.15.

For addresses with 4-digits before the point, the Programmer will automatically switch to expecting a hexadecimal representation for any bit number in the range 10 to 15. For example, you must key in W1000.15 as W1000.F.

If a N/C contact is the first element of a rung, connecting to the vertical line at the left hand edge of the screen, the input number at the left of the contact symbol is -1 decimal, or @FFFF hexadecimal, i.e. 16 bits that are all 1's.

MONITORING

When the bit defined by the address below the contact symbol is ON, and the output is zero:

```
..]/[..
A27.14
```

When the bit is OFF, and the output equals the input:

```
--]/[--
A27.14
```

OPERATION WITHIN BLOCKS

N/C contacts within BLOCKs only operate while the BLOCK is being obeyed.

When you are monitoring rungs, N/C contacts within BLOCKs are *displayed* as though they were operating, regardless of whether the BLOCK is being obeyed or not. However, when the BLOCK is not being obeyed, the last element in the rung (BLOCK, COIL, OUT, or SEQR) will not operate, even though a closed path to that element may be displayed on the screen.

FUNCTION

To transmit a number along a rung when a particular bit is set ON in data tables; to give zero out when this bit is set OFF.

FORMAT

```
--] [--
   Y
```

where:

Y = table address of the bit controlling the contact.

EXAMPLE

```
--] [--
A27.14
```

OPERATION

In the Example, when bit A27.14 is ON, then whatever number appears on the left of the contact symbol will be transmitted to the right hand side of the symbol. If A27.14 is OFF, then the number appearing at the right hand side of the symbol will be zero.

N/O contacts work with both integer and floating point numbers as rung inputs.

KEYBOARD ENTRY FROM PROGRAMMER

Press the following sequence of keys:

-] [- Key
Bit address, terminated by SPACE.

PROGRAMMING RULES

The address shown as Y in the "Format" above must define a bit, i.e. it must include a point followed by a figure generally between 0 and 15.

This address, however, must not contain more than 7 characters. You can therefore enter a bit address of W999.15, but not W1000.15.

For addresses with 4-digits before the point, the Programmer will automatically switch to expecting a hexadecimal representation for any bit number in the range 10 to 15. For example, you must key in W1000.15 as W1000.F.

If a N/O contact is the first element of a rung, connecting to the vertical lineat the left hand edge of the screen, the input number at the left of the contact symbol is -1 decimal, or @FFFF hexadecimal, i.e. 16 bits that are all 1's.

MONITORING

When the bit defined by the address below the contact symbol is OFF, and the output is zero:

```
..] [..
A27.14
```

When the bit is ON, and the output equals the input:

```
--] [--
A27.14
```

OPERATION WITHIN BLOCKS

N/O contacts within BLOCKs only operate while the BLOCK is being obeyed.

When you are monitoring rungs, N/O contacts within BLOCKs are *displayed* as though they were operating, regardless of whether the BLOCK is being obeyed or not. However, when the BLOCK is not being obeyed, the last element in the rung (BLOCK, COIL, OUT, or SEQR) will not operate, even though a closed path to that element may be displayed on the screen.

FUNCTION

Provides an output whose 16 bits are the result of performing the logical operation of Inclusive Or between each bit of the rung input and each bit of a numeric constant, table value, or second branch input.

FORMAT

```
-<OR >-        or        -<OR >-
   Y                      <   >
                          -<   >
```

where:

Y = table address of any table location, or numeric constant

EXAMPLES

```
(1)  -<OR >-
       #55AA

(2)  -<OR >-
       W410

(3)  -<OR >-
       <   >
     -<    >
```

OPERATION

In all three examples, the rung input is ORed with a second number, which is either:

- specified as a constant below the OR (Example 1),
- taken from a table (Example 2), or
- taken from a second branch input (Example 3).

The truth table below of a single bit shows that there will be a bit set to 1 in the output from the OR when there is either a 1 in the rung input or a 1 in the second number or a 1 in both:

Rung input	2nd number	OR output
0	0	0
0	1	1
1	0	1
1	1	1

It is because an output is provided when both inputs are 1 that the function is called "inclusive or"; compare with the truth table for "exclusive or" given on the sheet for XOR.

Example of ORing two 16-bit numbers:

```
1001011010100101   Rung input
0000000011111111   2nd number
1001011011111111   OR output
```

KEYBOARD ENTRY FROM PROGRAMMER

There is no single key for OR; press the following sequence of keys:

LOGIC
Letter O (*not* figure 0)
Numeric value, terminated by SPACE, or else
table address, terminated by SPACE, or else
JOIN (for second branch input).

PROGRAMMING RULES

1. OR cannot be the first element keyed into a rung.

2. When you use OR as in format example (1) above, the number below the OR must be in hexadecimal, and cannot be in decimal.

3. You cannot use OR with floating point numbers. If you attempt to do so, you will get a compiler error message and your program won't run.

APPLICATION NOTES

1. OR is meant for dealing with bits, not numeric variables. Its use with decimal numbers is generally meaningless, or at least confusing. For instance, the example given in "Operation" above would, in decimal, be -26971 OR 255 equals -27136.

2. There is little point in using the two-branch version of OR (Example 3) as you can obtain the same result by closing the lower branch with JOIN; see sheet for JOIN, where it is pointed out that a JOIN gives the equivalent of a "Wired OR". Both a two-branch OR and a JOIN require 1 instruction. The JOIN, however, does not occupy the space in the rung on a Programmer screen that could be utilized by another element, whereas OR does.

3. Because OR is meant for dealing with bits, it is best if you monitor OR's with figures displayed in hexadecimal.

FUNCTION

To set a word of 16 bits in a data table location to the number or bit pattern present on the rung input, or to set a pair of 16-bit words in a pair of consecutive data table locations if the number present on the rung input is in floating point.

FORMAT

```
-<OUT>+
   Y
```

where:

Y = table address of a write-enabled table location, or the table address of the first of a pair of data tables for floating point.

EXAMPLE

```
-<OUT>+
 G101
```

OPERATION

Data in the table location (G101 in the example above) is set to whatever pattern of bits is present on the rung input to the OUT. Depending on the application, this pattern of bits may represent individual logic states or may represent a number; if the OUT was for a serial link, it could represent a pair of ASCII characters.

Similarly, with 32-bit floating point data, the data in two adjacent Q-table locations is set to whatever pattern of bits is present on the rung input to the OUT.

KEYBOARD ENTRY

Press the following sequence of keys:

OUT key (this will automatically place the symbol at the end of the rung, and join it to the rest of the rung by a line)
Table address, terminated by SPACE

PROGRAMMING RULES

The address shown as Y in the "Format" above must define a word, not a bit, and therefore must not include a point.

Where the data you want to store in table is 32-bit floating point, then the table you use must be the Q-table. Also, the Q-table location number must be zero or an even number (e.g. Q0, Q2, etc). If you try using an odd numbered location or a table other than the Q-table, then you will get a compiler error message.

An OUT can only be the last element in a rung. It can also be placed only in the top branch of a rung that has parallel branches. Only one OUT can therefore appear in any rung.

You can use an OUT as the only element in a rung. Such a rung would set the word defined by the address Y to a fixed pattern of all 16 bits being 1, equivalent to a number of -1 decimal or @FFFF hexadecimal. See also the Application Notes below.

MONITORING

For 16-bit words or numbers, the value of the word contained in the table location will be shown above the OUT when monitoring, e.g.

```
3207
-<OUT>+
6101
```

If hexadecimal values are selected when monitoring, then the hexadecimal equivalent will be shown, e.g.

```
#C87
-<OUT>+
6101
```

Hexadecimal is more useful where you are treating the data as a set of bits rather than as a number.

No values are displayed where the number held in a pair of table locations is floating point.

OPERATION WITHIN BLOCKS

Any OUT contained in a BLOCK that is not being obeyed will remain in whatever state it is in, even though its rung input may be displayed when monitoring as though it were trying to drive the OUT to a different number or bit pattern.

PRINT-OUTS

When displayed on the screen of a Programmer, an OUT joins to a vertical line at the right hand side of the screen. When you print out a ladder diagram as a hard copy record, older versions of Programmer omit the right hand vertical line.

APPLICATION NOTES

1. If an OUT sets the 16 bits in a table location of an output table (e.g. B-table or D-table), the information is not read out from this table to the physical output devices (e.g. output modules, serial links, etc.) until the end of the complete ladder diagram program has been reached.

2. There is no restriction on you including OUTs and COILs with the same reference more than once in a program, setting bits or words in the same table location different ways. In that case, the way the bits will be set when they are read out to the output devices will be whichever way they were set when the end of the ladder diagram program was reached, even though they may have changed back and forth several times in the course of the program.

3. We do not generally recommend the above practice, however, as it makes programs difficult to understand, and increases the likelihood of you generating errors in the logic of your program.

4. It may be useful during commissioning, however, to insert a rung or rungs at the very end of a program to override the way that an OUT has been set earlier in a program, or that individual bits in the table location have been set by COILs earlier in the program. Such rungs should be removed when the end of commissioning is reached.

   See "Forcing Outputs" in the Item "Testing" in Section 3 of this manual.

5. There is also no restriction on you including OUTs in a program that set words in input table locations (e.g. A-table, C-table). The data in these input tables is normally set at the beginning of each program cycle by the Controller when it reads the data from the input modules and units. If you include OUTs in your program that set words in input tables, then the input data will be overwritten for these words.

6. We do not generally recommend the above practice, as it makes programs difficult to understand. You may find it useful during commissioning, however, to insert a rung or rungs at the very start of a program to override the data read in from the plant. See "Forcing Inputs" in the Item "Testing" in Section 3 of this manual. You should remove such rungs when you reach the end of commissioning.

7. On some older models of Controller, it is possible to use OUTs to write bits or words into input tables that are not being used (e.g. if output table location B12 is being used, then input table location A12 will not be used for input data). You cannot do this on modern Controllers, as you will get a compilation error message if you try this.

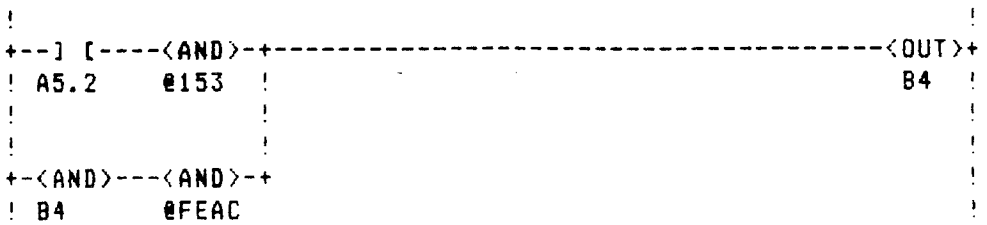
   However, the use of spare input tables on older Controllers is generally not to be recommended. You should only use them in this way where you are short of memory/instruction space. First check your configuration data in P-table to make sure that any such spare input table locations do not get set back to zero at the beginning of a program cycle, if you don't want this to occur.

8. Although OUTs are normally used for setting all 16 bits in a table location, you can use them in combination with AND for setting selected bits only. Using this technique, you can save on the number of instructions required in a program compared with outputting bits individually with COILs. For instance, if you need to set bits B4.0, B4.1, B4.4, B4.6 and B4.8 when A5.2 is ON, you can do this with the rung shown below. To set just the bits listed, you need a "mask" word of:

   0000 0001 0101 0011

   which can be represented in hexidecimal (see Section 1) as:

   0 1 5 3



   In this rung, the mask in the first branch permits only the desired bits to be set by A5.2. The second branch is included to retain all the other bits in B4 in their previous states; if you omitted this branch, they would be set to 0.

   This rung takes 6 instructions. Using individual rungs to set each of the 5 bits, using COILs, you would need 10 instructions.

   The masks with the two right hand ANDs must be complementary, i.e. where 1's occur in the first mask, there must be 0's in the second mask, and vice versa. See sheet for AND if you are unfamiliar with its use for masking.

   You can quickly work out the second mask by subtracting each digit of the first mask from 15. that is:

   Least significant digit = 15 - 3 = 12 = C in hex
   next digit = 15 - 5 = 10 = A in hex
   next digit = 15 - 1 = 14 = E in hex
   Most significant digit = 15 - 0 = 15 = F in hex

9. If you use an OUT to a Q-table pair of locations but the rung input to it is just a 16-bit integer, then the number is converted to 32-bit floating point format first and then stored.

You cannot, however, do the inverse of this - if you try storing a floating point 32-bit number in a single table location, even though it evaluates to an integer, then you will get a compiler error message. See Programming Rules above.

FUNCTION

To step through a sequence of up to 16 steps. Provides the software equivalent of a stepping relay or uniselector.

FORMAT

```
-SEQ.-+
  Y
-RESET
```

where:

Y = table address of a write-enabled table location.

EXAMPLE

```
-SEQ.-+
  G67
-RESET
```

OPERATION

The SEQR begins with a value of 1 in the table location you specify (G67 in the example above) whenever it is reset by the lower branch input being ON (i.e. any non-zero number).

On every program cycle during which the top branch input is ON while the RESET is OFF, the bit that is set to 1 in the table location moves one place to the left. Thus, with the SEQR in the example:

first operation:	G67.0 will be set to zero G67.1 will be set to 1
next operation:	G67.1 will return to 0 G67.2 will be set to 1 and so on, until
15th operation:	G67.14 will return to 0 G67.15 will be set to 1

After this, the sixteenth operation causes the sequence to re-cycle, with G67.15 being set back to 0 and G67.0 being set to 1.

If, at any time during the sequence, RESET is switched ON, the SEQR. returns to its starting condition with just the first bit (G67.0 in the example) being 1 and all other bits being 0. The RESET overrides any input that may be present on the top branch during the same program cycle.

KEYBOARD ENTRY FROM PROGRAMMER

Press the following sequence of keys:

SEQR key (this will automatically place the symbol at the end of the rung, and join it to the rest of the rung by a line)
Table address, terminated by SPACE.

PROGRAMMING RULES

1. The data table location given below SEQR must be one in write-enabled memory.

2. The rung inputs to a SEQR must not be floating point. If they are, you will get a compiler error message, and your program won't run.

OPERATION WITHIN BLOCKS

SEQRs within BLOCKs only operate while the BLOCK is being obeyed. The step that the SEQR has reached will remain frozen until the BLOCK starts to be obeyed, again after which it will carry on to the next step in the sequence.

MONITORING

During monitoring, the value contained in the table location entered below SEQR will be displayed. This is not the step number in the sequence, but will normally be a power of 2, e.g. as the SEQR steps, it will go through the values 1, 2, 4, 8, 16, etc., up to 16384 on the 14th step, then -32768 on the 15th step, after which it reverts to 1 on the 16th step. It is generally easier to follow if you display in hexadecimal, when the values will go through the series @1, @2, @4, @8, @10, @20, @40, etc., up to @4000 on the 14th step, @8000 on the 15th step, and back to @1 on the 16th step.

PRINT-OUTS

When displayed on the screen of a Programmer, a SEQR joins to a vertical line at the right hand side of the screen. When you print out a ladder diagram as a hard copy record, older versions of Programmer omit the right hand vertical line.

APPLICATION NOTES

1. To find the step number, the Special Function S3 (HISTATE) may be used. See the Software Data Sheet for this function.

2. Remember that SEQR moves on one step on every program cycle during which its input is ON. When you need a SEQR to move on just one step, then you must arrange for the input to remain ON for just one program cycle, and then be removed on subsequent program cycles.

3. If the table location you use with SEQR is one that is cleared to zero when the Controller is started up, the SEQR must be executed once to get Bit 0 set to 1. If you use the HISTATE Special Function in your program in an earlier rung than the SEQR, then after start up you will get an output of -1 on the first program cycle, as no bit will yet be set in the SEQR table location.

FUNCTION

To invoke any of the Special Functions available in the instruction set of the GEM 80 Controller.

FORMAT

```
      mmmmm                          mmmmm
(1a) -SPEC.-                   (1b) -SPEC.---VALUE-
       SXX                            SXX       Y

      mmmmm                          mmmmm
(2a) -SPEC.-                   (2b) -SPEC.---VALUE-
       TXX                            TXX       Y
     -<   >                         -<   >
```

where:

mmmmm	=	mnemonic of the Special Function (if available)
SXX	=	Single-branch input Special Function number
TXX	=	Two-branch input Special Function number
Y	=	numeric constant or write-enabled table location

EXAMPLES

```
 BCDIN                          LINCON
-SPEC.-                        -SPEC.---VALUE-
  S1                             S11      M75

 MULT                           MOVE
-SPEC.-                        -SPEC.---VALUE-
  T10                            T20       4
-<   >                         -<   >
```

NOTES

1. The range of Special Functions available in a given GEM 80 Controller is given in the Technical Manual for that Controller.

2. Details of the operation of Special Functions are given in separate Software Data Sheets.

3. Special Functions whose numbers begin with S have single branch inputs; those whose numbers begin with T have two branch inputs.

4. Some Special Functions need to be followed by a VALUE (as in Format 1b and 2b above) whereas others do not (Formats 1a and 2a above). Refer to the appropriate Software Data Sheet.

5. Again, some Special Functions can accept floating point inputs whereas others can not. Refer to the appropriate Software Data Sheet.

MNEMONICS

When you enter a Special Function in a rung via your Programmer, you can normally enter it by keying in either the Special Function No. (e.g. S11) or its mnemonic (e.g. LINCON). Whichever method of keying in you use, the Programmer will display both items, as in the Examples.

In a few cases, a mnemonic may not be recognized by the Programmer. This may be because:

(a) the Special Function is one developed more recently than the Programmer software version, or
(b) the Special Function is a custom designed one included in a Controller with specially modified firmware

In either case, you must enter the Special Function by giving its number and not its mnemonic.

KEYBOARD ENTRY

If the particular Special Function does not require to be followed by a VALUE, press the following sequence of keys:

SPEC. key
Special Function Number, terminated by SPACE, or else mnemonic, terminated by SPACE

If the particular Special Function requires a VALUE, press the following sequence of keys:

SPEC. key
Special Function Number, terminated by SPACE, or else mnemonic, terminated by SPACE
VALUE key
Numeric constant, terminated by SPACE, or else Table location, terminated by SPACE

PROGRAMMING RULES

1. Any number you give for the Special Function must be that of one contained in the particular Controller you are programming.

2. If the Special Function is not contained in the particular Controller, the action taken will depend on the version of Controller and on the version of the Programmer.

   In most cases, you will not be able to enter such a Special Function into the Controller without some error indication being given to you.

   In a few cases, with older versions of Controller, you may be able to enter a Special Function that is not in the Controller, but, although you can display the function, the Controller will ignore it (i.e. its rung output will equal its rung input, and any second branch will also be ignored).

MONITORING

Special Functions do not display any change when monitoring. However, if the Special Function is one that is followed by VALUE, and if the VALUE has a table location beneath it (see second Example), then the number in this table location will be displayed.

For the meaning of this number, refer to the Software Data Sheet for the particular function. In many cases, it is a status word indicating whether numbers in other table locations used by the Special Function are within the correct range of values, etc.; in such cases, the conditions indicated by the number obtained when monitoring should be looked up in the appropriate Table in the Software Data Sheet.

FUNCTION

To subtract one number from another, each number being a signed integer in the range -32,768 to +32,767, or a floating point number in the range roughly $\pm 10^{38}$ to $\pm 10^{-38}$.

FORMAT

```
-<SUB>-        or        -<SUB>-
   Y                      <   >
                         -<   >
```

where:

Y = table address of any table location, or numeric constant.

EXAMPLES

```
(1)  -<SUB>-
       245

(2)  -<SUB>-
       W17

(3)  -<SUB>-
      <   >
     -<   >
```

OPERATION

In all three examples, the rung input has a second number subtracted from it, which is either specified as:

- a constant below the SUB (Example 1),
- taken from a table (Example 2), or
- taken from a second branch input (Example 3).

Numbers may be positive or negative, and integer or floating point. The allowable number range is -32,768 to +32,767 for integer, and roughly $\pm 10^{38}$ to $\pm 10^{-38}$ in floating point.

For integers only, the rung output is limited to numbers in the range -32,767 (not -32,768) to +32,767. See the Application Notes below for the result of a SUB coming outside the permissible number range.

KEYBOARD ENTRY FROM PROGRAMMER

Press the following sequence of keys:

SUB key
Numeric value, terminated by SPACE, or else
table address, terminated by SPACE, or else
JOIN (for second branch input).

PROGRAMMING RULES

1. SUB cannot be the first element keyed into a rung.

2. When SUB is used as in format example (1) above, the number below the SUB must be a positive decimal number, and cannot be in hexadecimal or in floating print.

3. When two integers are subtracted, the result will be an integer. If either of the numbers is floating point, the result will be floating point.

MONITORING

When monitoring, if the second number is integer and comes from a table location, as in Example 2, then the value of this number will be displayed above SUB, e.g.

```
1357
-<SUB>-
W17
```

The result of the subtraction is not displayed.

APPLICATION NOTES

1. The result of the subtraction of one integer from another could be a number either more positive than +32,767 or more negative than -32,767. In such cases, the rung output from a SUB will be set to +32,767 or -32,767 as appropriate and the excess ignored.

   If the rung input is -32,768, and if 0 is SUBtracted from it, the rung output will be -32,767.

2. When such overflow occurs, the single bit E0.7 will be set to 1. If overflow represents an abnormal condition in the system operation, you can take appropriate action by putting rungs in your program that utilize this bit.

3. E0.7 will be set back to 0 when a further SUB or ADD is encountered in the program, the result of which does not cause overflow.

4. To prevent overflow, the order in which you perform a number of separate calculations can be important.

   For example, the expressions (a + b - c) and (a - c + b) are algebraically equivalent. However, if a, b and c are all negative, then the first expression could cause overflow from the initial addition (a + b). On the other hand, the initial subtraction (a - c) in the second expression would give a positive or less negative number that could not overflow.

   Where you design a rung performing integer arithmetic involving ADD, SUB and the Special Functions MULT and DIV, you should check your design for what will happen at intermediate points in the rung when large numbers are used against possible overflow.

FUNCTION

To allow entry of a numeric constant or of a table location containing data for use by a DELAY, a COUNT, or a Special Function.

FORMAT AND EXAMPLES

See COUNT, DELAY, and SPEC.

PROGRAMMING RULES

VALUE cannot be used on its own. It can only be used following COUNT, DELAY or SPEC.

THIS PAGE LEFT INTENTIONALLY BLANK

FUNCTION

Provides an output whose 16 bits are the result of performing the logical operation of Exclusive Or between each bit of the rung input and each bit of a numeric constant, table value, or second branch input.

Usually used for comparison between two words to detect which bits are different.

FORMAT

```
-<XOR>-        or        -<XOR>-
   Y                      <   >
                         -<   >
```

where:

Y = table address of any table location, or numeric constant

EXAMPLES

```
(1)  -<XOR>-
       #55AA

(2)  -<XOR>-
       M410

(3)  -<XOR>-
       <   >
      -<   >
```

OPERATION

In all three examples, the rung input is XORed with a second number, which is either specified as:

- a constant below the XOR (Example 1),
- is taken from a table (Example 2), or
- is taken from a second branch input (Example 3).

The truth table below of a single bit shows that there will be a bit set to 1 in the output from the XOR when there is either a 1 in the rung input or a 1 in the second input but not a 1 in both at the same time.

Rung input	2nd number	XOR output
0	0	0
0	1	1
1	0	1
1	1	0

It is because there is no output when both inputs are 1 that the function is called "exclusive or"; compare with the truth table for the normal or "inclusive or" given on the sheet for OR.

Example of XORing two 16-bit numbers:

1001011010100101	Rung input
0000000011111111	2nd number
1001011001011010	XOR output

KEYBOARD ENTRY FROM PROGRAMMER

There is no single key for XOR; press the following sequence of keys:

LOGIC
Letter X
Numeric value, terminated by SPACE, or else
table address, terminated by SPACE, or else
JOIN (for second branch input).

PROGRAMMING RULES

1. XOR cannot be the first element keyed into a rung.

2. When you use XOR as in format example (1) above, the number below the XOR must be in hexadecimal, and cannot be in decimal.

3. XOR will not work with floating point inputs. If you attempt to make either input floating point, you will get a compiler error message, and your program will not run.

APPLICATION NOTES

1. XOR is meant for dealing with bits, not numeric variables. Its use with decimal numbers is generally meaningless, or at least confusing. For instance, the example given in "Operation" above would, in decimal, be -26971 XOR 255 equals -27046.

2. XOR is most frequently used, as a logic operation, for comparing two 16-bit numbers to detect which bits are different. You can think of it as a "Not Equal" function, as it gives output bits set to 1 where there is a difference between the rung input and the second number. See the example given in "Operation" above.

3. Because XOR is meant for dealing with bits, it is best if you monitor XOR's with figures displayed in hexadecimal.

4. You may enter an XOR at the start of a rung, if you precede it with LINK, to give the equivalent of AND followed by INV, thereby saving one instruction. The LINK will be removed if you recall the rung for display later on; see sheet for LINK. Thus:

```
+-<XOR>-
  6214
```

is equivalent to:

```
+-<AND>---<INV>-
  6214
```

GENERAL GUIDANCE SECTION

CONTENTS PAGE

SECTION 3

CONTENTS PAGE

We can't give you a comprehensive list of all the mistakes that users of GEM 80 Controllers are capable of making when programming them. The notes below are therefore given as guidelines to good programming practice, and show how to avoid some of the commoner mistakes.

PLANNING YOUR PROGRAM

Don't just power up your Programmer and start keying in rungs as though it were a personal computer. Plan your program first. Get a large stack of paper, and start sketching your program out. Don't stop there, but carry on and fill in the detail. You should take 90% of your time working out your program and checking its logic, and then the last 10% - keying the program into the Controller and program testing - will come easy.

Prior to this last 10%, the only time that you might need to get your hands on the Programmer is for checking the operation of some element that we haven't explained adequately in this manual. Drop us a line, using the form provided at the rear of this manual, and we will try to make sure that our next version of this manual covers the point you had difficulty with.

HOUSEKEEPING RECORDS

Keep a record of every table location you have used, and what you used it for. In some cases, you may also need to keep a record of what individual bits were used for.

We have included some forms we find useful for keeping records on in Section 4. Feel free to photocopy these.

Be particularly careful about recording *table locations whose addresses do not appear explicitly in the ladder diagram.* For instance, many Special Functions require a number of preset values. You will often enter these values in Data Address mode, but only the first one of their addresses will appear in the ladder diagram. In particular, you may use the Special Functions LOCATE (S20) and MOVE (T20) to move a block of data from one table to another; only the address of the first word of data in each table will appear explicitly in the ladder diagram.

If you use the Search facility on the Programmer to try to find where you have used a table location, any addresses that the Programmer cannot find in your ladder diagram *could have been used* by one of these Special Functions. So don't fool yourself; keep records, and check your records, not what you can see on a Search.

PROGRAMS WHERE SECTIONS ARE WRITTEN BY DIFFERENT PEOPLE

Where, with a large program, different sections are written by different people, the need for housekeeping records is absolutely essential. The different people involved in programming should meet, and decide what table locations will be allocated to each person, so that one person does not start overwriting table locations used by another person in another section of the program. Further, some data written into tables by one person in his section of program may need to be accessed by another person in *his* section of program, and so you need to mutually agree what table locations will be common to both sections of program.

When working out such allocations of table locations, remember that only those with three-figure numbers (e.g. W547) can always have the bit address entered in decimal (e.g. W547.9) when used with CONTACTs and COILs. With four-figure numbered table addresses, you may have to use hexadecimal to represent the bit number. You may therefore find it desirable to give one person W0 to W499 and W1000 to W1499, while a second person is allocated W500 to W999 and W1500 to W1999, say, if you want to avoid using hexadecimal numbers for bit addresses.

PLAY IT SAFE - BE PESSIMISTIC

When you write a program, assume that everything that can possibly go wrong *will* go wrong. Assume that your program will have bugs in. Assume that the hardware in the Controller will go wrong. Look on the black side, and keep on asking yourself "What if such and such a thing happened?"

If you haven't come across this approach before, it is the way that professional Data Processing people write their software. What they do is to write "error traps" into their programs. Then, if something does go wrong, the program doesn't do anything stupid. Instead, the traps take some appropriate action, like stopping the program from carrying on running before it bills a million dollars when it should have been 50 cents.

Adopting the D.P. professionals' approach, you should therefore build error traps into your GEM 80 programs. You may only require some of these error traps, where they are to trap software bugs you've inadvertently written, until you are fully confident that there are no conditions under which a section of program can give wrong answers, and you can remove them later after program testing. You should leave in error traps for hardware failure.

ERROR TRAPPING - VALIDITY CHECKING OF INPUT DATA

The cardinal rule, both in D.P. and with GEM 80 Controllers, is that you should not make use of input data until you have checked that it looks reasonable.

Input data may be wrong for several reasons. First, where data comes from a keyboard, thumbwheel switches, etc., the operator could input a value that is inappropriate (probably too big). Second, analog signals could be out of range, either through an equipment fault or from something simple like a broken wire. Third, where input consists of single bits, it may be that these should switch on and off in a fixed sequence (e.g. photocells on a conveyor line), in which case you can detect a fault if they don't. And so on. The list is endless. Keep asking yourself the "What if..?" question, and put in extra rungs to check input data where necessary.

Another type of input data is that transmitted to one GEM 80 Controller by another, via a serial link. Don't necessarily trust this data; especially, don't trust it if the program for the other Controller was written by somebody else! Because of the automatic checking on GEM 80 serial links, the data received by one GEM 80 Controller will be the same as that transmitted by the other Controller, but that doesn't prevent the data being wrong in the first place.

ERROR TRAPPING - DATA MANIPITATION

Once you have got the input data checked for validity, generally towards the start of the program, then you can write the data manipulation part of the program. Initially, this may contain bugs that are due to faulty logic in your program writing.

If there are places in the program where you are a little unsure about whether figures that result from arithmetic or data manipulation could under unusual circumstances get outside the desired range, you could put in the LIMIT Special Function S32. Not only will this prevent numbers going outside the desired range, but it will also give you an error code in the first table location (whose address is given with the VALUE) if the program ever calculates a number larger or smaller than the limit values.

The most serious circumstance in which an incorrect number will give trouble is where you are moving data about with the MOVE Special Function. With this function, you can copy data from one table to another (maybe from one Controller to another via a serial link), or from one portion of a table to another portion of the same table. If the numbers you use in LOCATE and MOVE Special Functions S20 and T20 are calculated wrongly, data will be transferred to wrong tables, and will very likely overwrite good data.

AVOID CHECKING FOR EQUALITY

In many programs, there are sections of program split off as BLOCKs that have to be operated on a set number of program cycles, and not obeyed on subsequent program cycles (or at least, not until some further condition is met). Normally, you will keep a tally of the number of program cycles in which the BLOCK has operated by using a totalizing rung within the BLOCK such as:

```
!                                                                    !
+-<AND>---<ADD>-----------------------------------------------<OUT>+
! W1340      1                                                W1340!
```

What you now need to check is the value of W1340 at the start of the BLOCK, to determine whether it should be obeyed or not. Suppose you want the block to be obeyed 10 times. Then a possible rung is

```
!                                                                    !
+-(AND)---(SUB)-----------------------------------------------OBEY-+
  W1340     10                                                BLOCK
                                                                 *
************************ START OF BLOCK ************************
```

Since a BLOCK is obeyed whenever the input is ON (non-zero), then when W1340 contains precisely 10, the BLOCK will no longer be obeyed. However, if, because of a bug elsewhere in the program, the contents of W1340 skipped to 11, the BLOCK would again be obeyed - and forever after. It is therefore safer not to check whether the contents of W1340 are equal to 10, but whether they are greater than 9. At the cost of one extra instruction, you can do this by modifying the rung as below:

```
!                                                                    !
+-(AND)---(SUB)---(AND)---------------------------------------OBEY-+
  W1340     10      @8000                                     BLOCK
                                                                 *
************************ START OF BLOCK ************************
```

where, as long as the contents of W1340 are less than 10, the result of the subtraction is negative, and the sign bit will be set. Using the @8000 mask, only this sign bit is passed along the rung, causing the BLOCK to be obeyed. Once the result of the subtraction is zero or is positive, the sign bit will become zero and the BLOCK will no longer be obeyed.

Although the example above is fairly trivial, and describes a situation where there is little likelihood of the program going wrong, you will have got the general idea. It is much more likely that checking for equality will go wrong where input data is suspect (assuming that, by keeping good housekeeping records, you have no program bugs causing overwriting of data within the program itself).

AVOID THE CLEVER-CLEVER

At the end of the day, your program when completed should be "readable". That is, what the rungs do should not need a lot of comments to explain how they do it. You should only need comments to say *what* a rung or a block does, not how.

You should therefore beware of writing rungs that are too clever, and whose operation is difficult to understand. Often, the presence of terribly clever rungs in a program indicates that the programmer, rather than being a whiz kid, has not thought enough; there will probably be a much simpler solution to the problem that would produce readable rungs and require fewer instructions.

BUGS UPON BUGS

Be careful about how you remove bugs from a faulty program. How to eliminate some bugs is fairly obvious, but with more obscure bugs it has been known for the user to patch in an extra rung or two that cure the known bug - but introduce two new ones that weren't there before!

Think first. Don't try curing a bug by the most obvious solution, until you've checked out whether it produces any unwanted side effects.

This applies both at the design stage and at the installation stage of your program. If you feel a certain section of your program is getting in a bit of a mess, it may be worthwhile making a second start. You can probably make a much tidier job of that messy section of program a second time round, rather than carrying on patching up a poor first version.

NAUGHTIES

Most people are curious. They like to try out using instructions in ways that we haven't published anything about. They come up with tricks by which they can save on the number of instructions in a program, for instance.

Remember that, if there is an unpublished trick way of doing something, we don't guarantee that it will always work. What may have worked on one model of Controller may not work on another model, or even on a later version of the same model, if it is not published in our data sheets. For instance:

- Don't use absolute addressing of memory with Special Functions where we recommend using LOCATE.
- Don't use S, T or V-tables in rungs. By all means look at their contents in a datalist display on your Programmer, but don't put them in rungs.

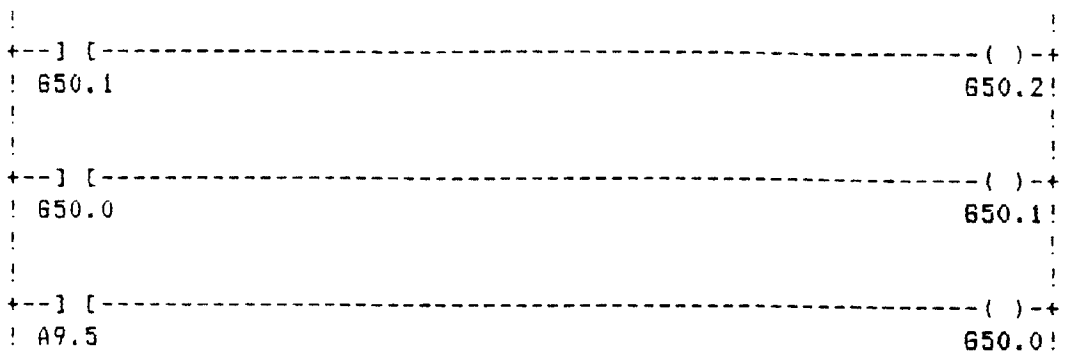
MATHEMATICS

Unless your particular model of Controller caters for floating point numbers, then the simple mathematics functions ADD, SUB, MULT and DIV will only work in integer arithmetic. However, even on Controllers that won't handle floating point numbers in your program, many of the Special Functions work in floating point arithmetic internally, although giving answers rounded to the nearest integer.

Where you need to multiply a number by a factor close to 1, e.g. by 1.048, or other numbers including decimal fractions, then you should use the Special Function LINCON (S11), if available in your model of Controller. For interpolation between two output values, you can use the Special Function FGEN (S30) with just a single segment defined.

ORDER OF RUNGS

When a GEM 80 Controller runs through the program you have entered into it, it performs the execution of the program in order, starting from the first rung and working through until it reaches the last rung (but skipping any rungs in non-obeyed blocks). The order of rungs is therefore important; you must not think of a ladder diagram program as being the exact equivalent of a hard wired relay circuit, because it isn't, even if there are similarities. For instance, consider the three rungs shown below:

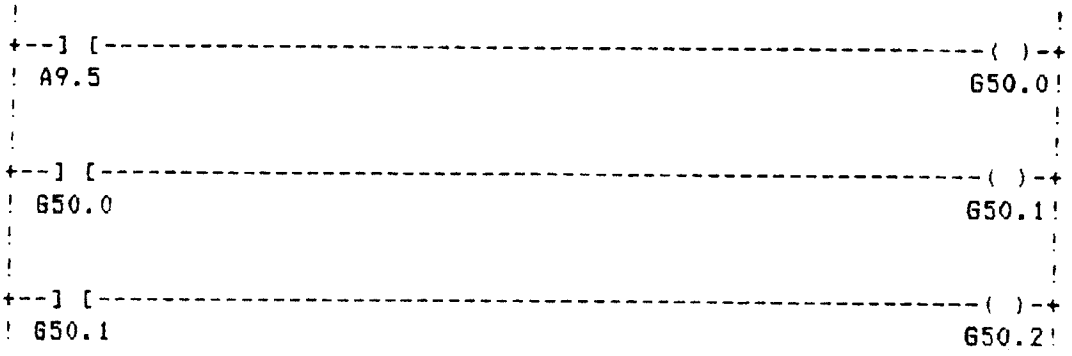


Assume that G50.0, G50.1 and G50.2 are all initially OFF (zero). Suppose that A9.5 is ON the first time the program is executed. Because the rungs are executed in order, the first rung makes no change to G50.2, which remains OFF. The second rung similarly makes no change to G50.1, which remains OFF. The third rung, however, sets G50.0 to ON. G50.0 going ON makes *no difference* to G50.1, because the second rung has already been executed.

On the second cycle through the program, however, with G50.0 now being ON, the second rung will set G50.1 ON. Since the first rung will already have been executed by the time this happens, G50.2 will remain OFF.

It is therefore only on the third cycle through the program that G50.2 goes ON. In fact, with the rungs in the order shown, the designer has effectively programmed a two-program-cycle delay between A9.5 operating and G50.2 coil energizing.

Now suppose that you reversed the order of the rungs, i.e.:



In this case, when A9.5 goes ON, G50.0 is set ON as before, but this immediately sets G50.1 ON in the second rung; and G50.1 being ON immediately sets G50.2 ON in the third rung. All the changes take place on a single cycle through the program, as a "ripple through" action from one rung to the next, without any program-cycle delays.

ORDER OF ELEMENTS WITHIN RUNGS

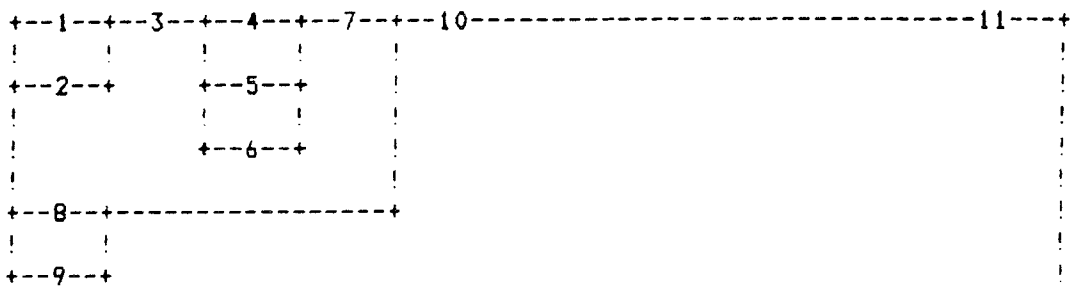
At first sight, you might think that you can have only one output in any rung, because the rung must end with a COIL, OUT or SEQR (or a BLOCK, which is not really an output) on the top branch of the rung, and because you cannot put these particular elements anywhere else in a rung.

However, a rung can produce an output halfway along the rung where it contains an element followed by a VALUE. Standard Functions requiring a VALUE are COUNT and DELAY. Several Special Functions also require a VALUE, as defined on the individual Software Data Sheets. For these functions, you have to specify a table location for the Controller to store intermediate results in memory.

For instance, with a COUNT, the table location stores how far the COUNT has got. With a DELAY, it stores how far the DELAY has yet to go when timing out. With a Special Function, it may store fault or status flags (e.g. signal in limit with LIMIT Special Function S32). Consequently, the contents of these table locations can change halfway through the execution of a rung, and it is not just the output element at the end of the rung that will change.

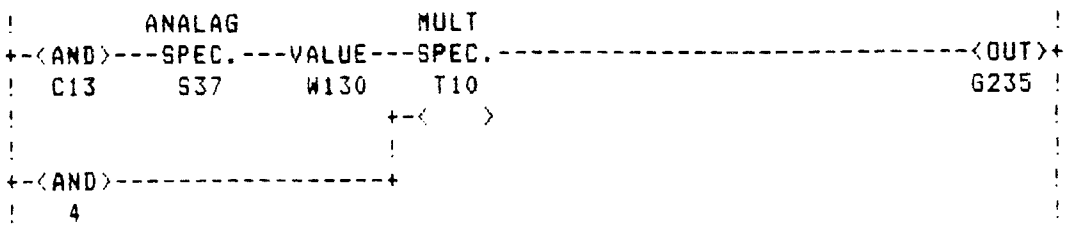
In most cases, these changes to the contents of certain tables halfway through a rung don't matter. However, in a few cases, you may write a rung where you make use of these table values within the *same rung* as where they get changed. In this case, the order in which elements in a rung get executed can be of significance.

The order of execution is as follows. The top branch of any rung will be executed from left to right until a branch close is encountered. At this point, execution will continue with the lower branch until the two branches close, when execution will continue along the top branch again. If another branch close is encountered, again execution will switch to the lower branch until it meets the upper branch. As an example, the rung shown diagrammatically below would be executed in the order shown by the figures:

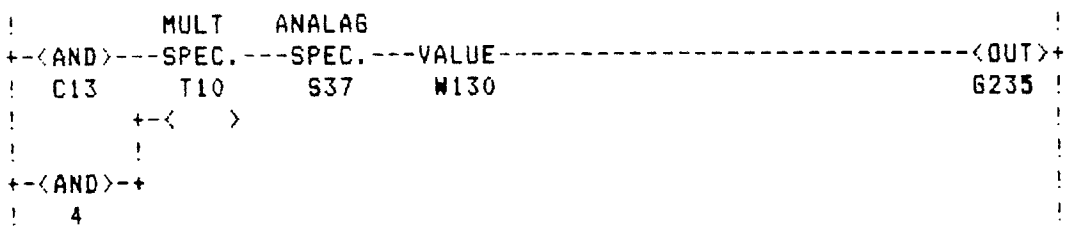


ORDER OF ELEMENTS - ARITHMETIC

Be careful about putting arithmetic and Special Functions in the best order, unless your model of Controller has floating point number facilities. At first sight, you might expect different orders to give the same result, but they may not do so with integer arithmetic. For instance, suppose you have an analog input signal that you wish to smooth out using the ANALAG Special Function (S37) and scale up by multiplying by 4. Then you could use the rung shown below:



However, this has one minor disadvantage. Since the ANALAG Special Function gives an output that is an integer, then the output of the MULT Special Function can only move in steps of 4. If, on the other hand, you put the MULT Special Function first, the ANALAG Special Function would smooth out the stepping by 4, so that the output from the OUT could move in steps of 1 :



Note that, if you replaced the MULT with the LINCON Special Function, you would get no improvement compared with either of these rungs where the multiplying factor was 4. LINCON works in floating point internally, but produces an output that is rounded to the nearest integer, and so would give exactly the same result in this case as MULT.

You would need to use LINCON, however, where the multiplying factor was a fractional number (e.g. 3.9 or 4.1). Regardless of this, you would still get the output number jumping if you did not put the ANALAG following it, rather than before it, in the rung.

Other comments about the best order for arithmetic operations are included in the sheets for ADD and SUB in Section 2 of this manual.

ORDER OF ELEMENTS - FLOATING POINT ARITHMETIC

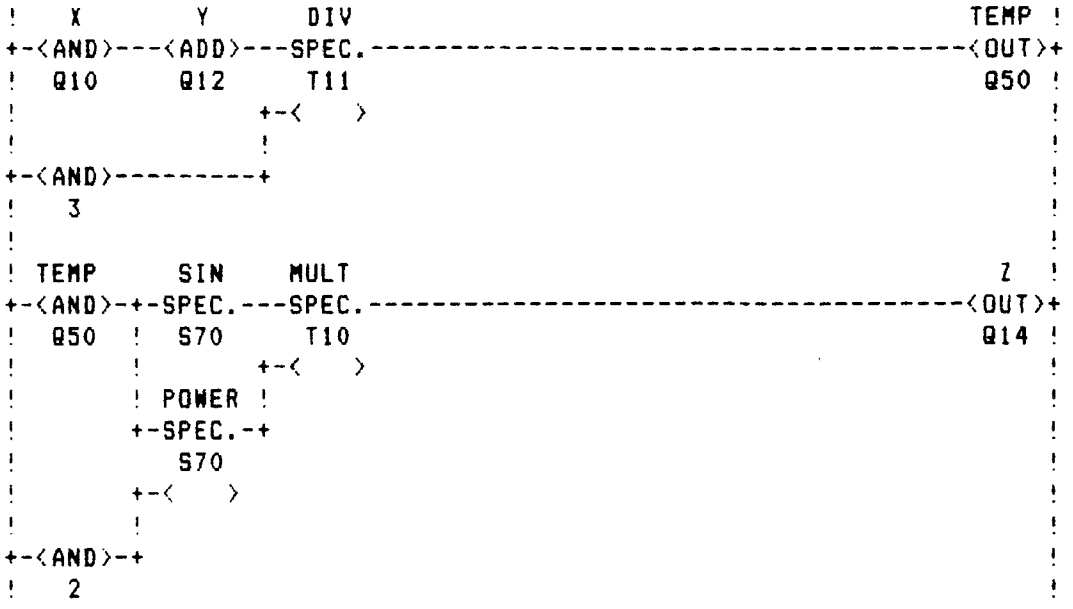
If your Controller includes floating point mathematics facilities, then it can use numbers up to 10^{38} in size, to 6-figure accuracy.

Although this is generally sufficient, the coprocessor in fact works to 9-figure accuracy. If you are worried about gettig the best accuracy from your ladder diagram program, then you should put as much floating point calculation as you can in each rung, and avoid splitting the calculation into a number of separate rungs.

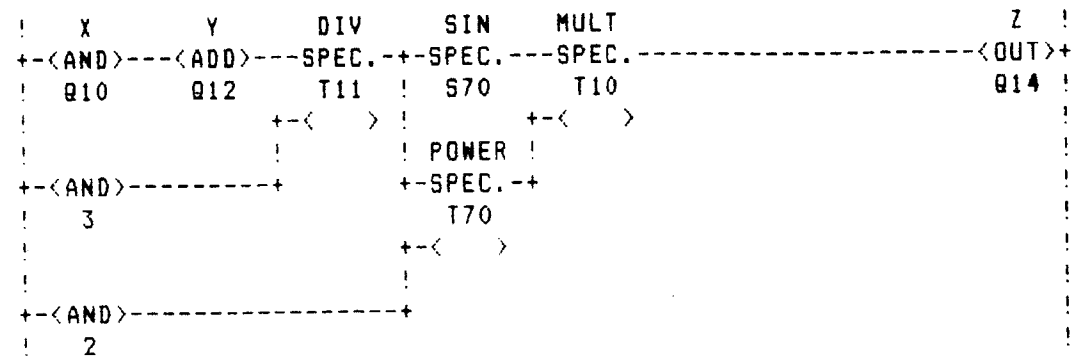
For instance, suppose you want to evaluate:

$$Z = ((X+Y)/3)^2 \times \sin((X+Y)/3)$$

Because of the common expression (X+Y)/3, it is quite likely that you would write the following program to evaluate Z:



However, you would get more accurate answers by combining the two rungs into the single rung:



Every time a floating point value is outputted to a table, it is rounded to 6-figure accuracy. In the first example above with two separate rungs, this occurs when the first stage of the evaluation is outputted to Q50 and Q51. In the second example with only one rung, the figure can retain its full 9-figure accuracy right till the end of the calculation, when the result is rounded to 6-figures as it is placed in Q14 and Q15.

CONFIGURATION DATA

Some types of GEM 80 Controller need you to give them all the facts and figures about the number of inputs and outputs that are connected to the Controller by way of I/O modules and units. You have to feed this data into specific P-table locations, as given in the Technical Manual for the particular model of Controller. Once you have fed this information in, there is no need for you to change it ever again (unless you add further I/O modules to the hardware at a later date, when you will have to amend it in line with the amended hardware). Consequently, this P-table data is not normally set by rungs in the program, but instead you directly input it into P-table from the Programmer as a data list.

Modern types of GEM 80 Controller, however, save you this chore by deducing from your program what inputs and outputs you are using. If the Controller sees that you have got a rung containing, say, an AND with table address C45, then it "knows" that you have got some Fast I/O input module whose input data needs to be read and placed into table location C45. You will find from the Technical Manual for your particular model of Controller what data you need to feed into P-table and what data the Controller can deduce for itself.

What is important, however, with the modern type of Controller, is that you include somewhere in your program a rung containing the *highest* table address used in each table being used. It is common practice to use the Special Functions LOCATE (S20) and MOVE (T20), for instance, to copy data from one table into another (see Item "Space Hints" later in this Section). If you do this, then you are likely to have only the *lowest* table addresses of the blocks being transferred appearing in your program. We also pointed out earlier in this Section that you needed to keep records of what table locations you had used, because many of these locations would not have addresses appearing explicitly in the ladder diagrams. Knowing from your records what the highest address is that you have used in any table, you may therefore need to include a "do nothing" rung such as:

```
!                                                                  TRASH!
+-(AND)---(AND)---(AND)---(AND)-------------------------------(OUT)+
!  C95      D93     A115     B101                                G100 !
```

which, in this instance, declares the highest A, B, C and D-tables used. The number obtained from ANDing them all together has to go somewhere, so it is put into table location G100. This table location is not used anywhere else in the program, except maybe for dumping other nonsense values to.

New users of GEM 80 Controllers sometimes complain that they can't get all the program in that they wanted, and that they have run out of instruction space. This usually turns out to be because they have not programmed their Controller very efficiently. The notes below give some ideas on how to get more into the available memory.

FRILLS

Begin by programming in what you intended originally, and leave any frills till the end. Many users, on discovering the power of a GEM 80 Controller, start to put in extra bits of program that weren't part of what was anticipated when the order for the Controller was placed. Confine yourself to programming the controls originally envisioned, and put in any fancy bits at the end in what memory space is left over.

TEXT

If you are using a printer, or a VDU connected to a printer port, remember that wordy-wordy messages can take up an awful lot of table space, and reduce the amount of memory available for ladder diagram instructions. Keep messages short, using recognized abbreviations, and string sub-messages together rather than duplicating whole messages, until you are certain that you've got room for more words.

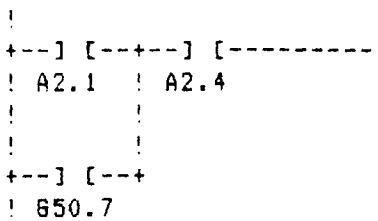
On older models of Controller that include paged or logging video, note that you can couple a printer to this processor rather than to one of the Controller serial ports. Any text printed will then come from the video format memory, and will not use up any of the P.C.'s memory. Since the text will be the same as that displayed in part of the video monitor screen area, you need to take this into account when you design your video formats. For instance, you might reserve the bottom couple of lines on the screen for alarm messages that you also need to print.

USE OF WORDS RATHER THAN BITS

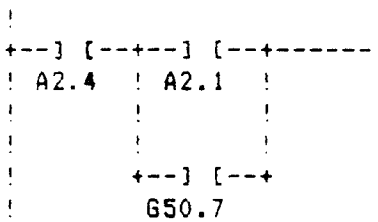
Remember that the logic instructions AND, INV, OR and XOR operate on 16 separate bits in parallel. It is therefore often possible to replace a number of separate rungs that use CONTACTs and COILs by a single rung that uses logic operations. An example of this is given in the sheet for OUT earlier in this manual.

ORDER OF ELEMENTS IN RUNGS

Opening a branch halfway along a rung takes 2 instructions, whereas opening a branch from the left hand vertical line takes no instructions. Thus:



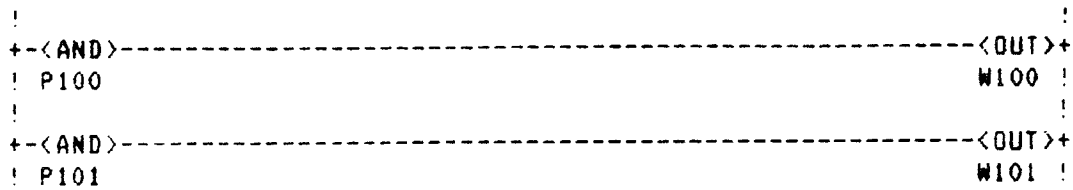
takes 2 instructions fewer than:



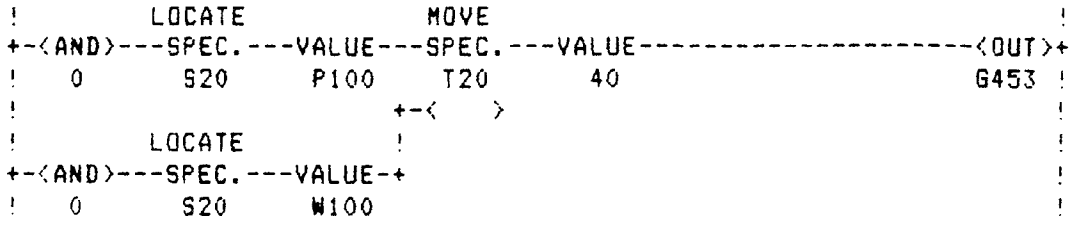
even though they are logically equivalent.

TRANSFER OF DATA USING LOCATE AND MOVE

There are many cases where you will want to move data from one table location to another. One of the commonest cases is where preset data in P-table needs copying into a write-enabled table such as G-table or W-table. You could do this with a number of separate rungs such as:



and so on, but if there is a lot of data to transfer, this elementary method will take up a lot of rungs and a lot of instructions. Consequently, it becomes easier to use the Special Functions LOCATE (S20) and MOVE (T20) as follows:



which copies 40 P-table words of data (from P100 to P139) into 40 W-table locations (W100 and W139). For more details, refer to the Software Data Sheets for the particular Special Functions.

Note, however, that the use of LOCATE and MOVE does affect how you lay out your tables. You can only use LOCATE and MOVE to copy data from and to a block of contiguous table locations, without any gaps in the middle.

Note also that, in a practical program, you should put any section like those above that copy data out of P-table (whether using separate rungs or using LOCATE and MOVE) in a BLOCK that operates only once, when the Controller is powered up; there is no point in recopying the data on every program cycle.

HARDWARE LAYOUT

The arrangement of input and output modules and units affects how easy or difficult it will be to write your program. Consequently, it affects how many instructions you will need. Your hardware layout and your program should be considered together, not independently.

For instance, the arrangement of I/O modules on the basic I/O highway determines what A-tables and B-tables you will use. If you want to use LOCATE and MOVE, you can only do this if the relevant table locations are contiguous, as pointed out above. Ease of programming may also depend on which bit within a word does what, which in turn depends on which signal goes to which module terminal.

LOOK-UP TABLES

The Special Function FGEN (S30) is generally thought of as being the software equivalent of a hardware function generator, used for such things as linearizing an analog input or output with a suitable straight line segmented approximation to a desired curve.

However, if you use FGEN with the rung input incrementing in steps of 1, then, for each different value of input, you can preset a corresponding value of output. FGEN can therefore provide you with the equivalent of a look-up table in a single rung, e.g.:

```
!          FGEN                                                    !
+-(AND)---SPEC.---VALUE-----------------------------------------(OUT)+
!  A17      S30      P154                                        W150 !
```

where, in the P-table data from P158 onwards, the look-up values are stored. For more details, refer to the Software Data Sheet for FGEN (S30).

If you attempted doing the equivalent in other ways, you could end up with numerous rungs and quite complicated logic.

CONDITIONAL ADD, etc.

Suppose you need to add one number to another only if a condition is met. Probably the most obvious way to do this is as follows:

```
!                                                                   !
+-(AND)-----------(ADD)-------------------------------------------(OUT)+
!  G1               (    )                                         G3  !
!                 +-(    )                                             !
!                 !                                                   !
+-(AND)----] [--+                                                     !
!  G2      A2.5                                                       !
```

where G2 is added to G1, and the result placed in G3, only if A2.5 is ON, otherwise nothing will be added to G1, and G3 will then equal G1. A simpler implementation is shown below:

```
!                                                                   !
+-(AND)----] [----(ADD)-------------------------------------------(OUT)+
!  G2      A2.5      G1                                            G3  !
```

which accomplishes the same thing, but uses one fewer instruction.

There are many similar cases where, with a little thought, you can save instructions with little effect on the readability of the program. Look out for rungs having more than one branch that starts with an AND just planting a number into the branch without doing anything else; these are the rungs that are most likely to offer scope for simplification in a similar way to the example above.

GROUP INSTRUCTIONS

Some models of Controller include a number of Special Functions that operate on groups of table locations. For instance, you can use the Special Function XORGRP in place of several XORs, probably reducing the number of rungs required as well as the number of instructions. One application is in alarm schemes, where you may want to check whether any bit has changed in, say, 200 inputs. Since 16 input bits make up one word, you would need at least 13 XORs, using single word instructions, but only one XORGRP, using group instructions.

THIS PAGE LEFT INTENTIONALLY BLANK

The longer you make your program, the more instructions it will contain. Consequently, the time it takes to execute a cycle through the program will also get longer. Inputs from and outputs to the plant being controlled would then get serviced less frequently. The notes below therefore give some ideas on how to keep the program cycle time short in spite of the program being long and complex.

USE OF BLOCKS - SPREADING THE LOAD

Before the start of a program cycle, the GEM 80 Controller reads the states or values of all its inputs. This action is automatic, under firmware control, and you cannot affect it with your ladder diagram program. In a similar way, after it reaches the end of a program cycle, the GEM 80 Controller reads out new values or states to the physical output modules and units, again automatically, under firmware control.

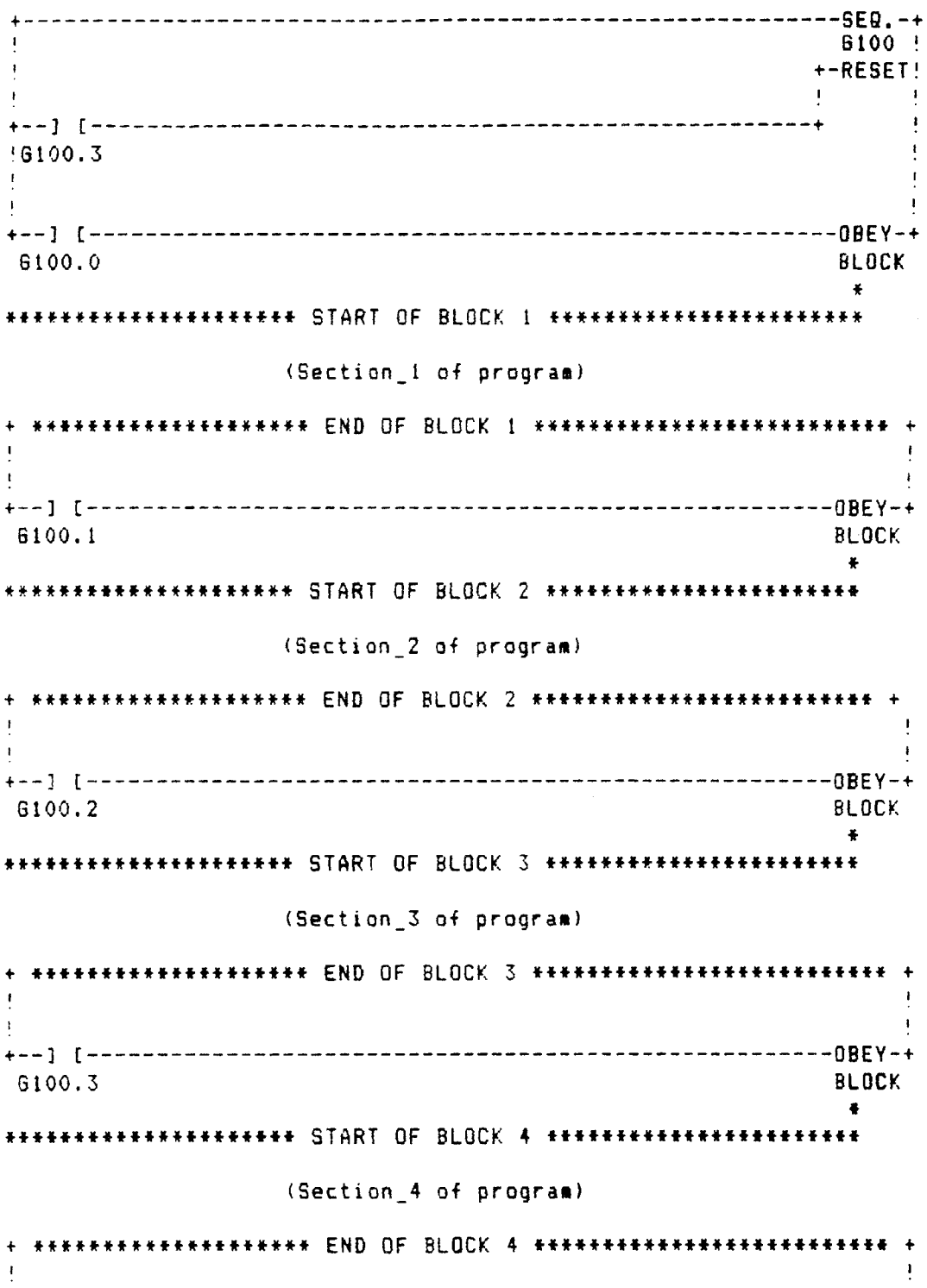
However, you don't have to make use of revised input values, nor update output values, if you don't want to. For instance, on many flow control loops, if you updated the value of an analog signal to a valve actuator on every program cycle, the valve actuator probably wouldn't be able to keep the valve in step with such a rapidly fluctuating signal, but, because of all the dithering about, mechanical wear might be increased.

When designing your program, you should therefore differentiate between:

- parts of the program that must operate on every program cycle (where rapid response is required), and:
- those parts of the program that only need to operate every few program cycles (where such a rapid response is unnecessary).

Using the BLOCK function, you can then arrange to operate only certain sections of your program on certain program cycles where rapid response is not essential. You should attempt to equalize the times on different program cycles, such that no one program cycle will take significantly longer than the rest.

As an example, you could operate four sections of program in turn on different program cycles using the SEQR function, in a repetitive sequence, as shown below:



To make sure that the different sections of program within the BLOCKs take roughly the same times to execute, you can begin by trying to make the BLOCKs contain roughly the same number of instructions as each other. However, especially if some of the rungs include Special Functions, you will get a more accurate answer by counting up the execution times of the individual instructions. You will find the execution time of each type of instruction given in the Technical Manual for your particular model of Controller.

USE OF BLOCKS - REACTING TO CHANGES ONLY

Another technique using BLOCKs is to check whether there have been any input *changes* since the previous program cycle, and only to obey a section of program when there has in fact been a change. For example:

```
!                                                                    !
+-(AND)---(XOR)-----------------------------------------------OBEY-+
   A8      G150                                                BLOCK
                                                                *
************************ START OF BLOCK ************************

                        (Section_of Program)

!                                                                    !
+-(AND)-------------------------------------------------------(OUT)+
!  A8                                                          G150 !
!                                                                    !
+ ************************ END OF BLOCK ************************** +
```

In this example, the BLOCK is obeyed only if there is a difference between the bits in A8 and those in G150. At the end of the BLOCK, G150 is updated to the latest value of A8 (this rung being included within the block since, if A8 hasn't changed, you don't need to update it).

This technique only reduces the program cycle time when the BLOCK is not being obeyed. It will therefore only speed things up if the program contains several such BLOCKs and where the likelihood of more than a small fraction of them ever needing to be obeyed on the same program cycle is low.

USE OF BLOCKS - FINAL NON-OBEYED BLOCK

On some older models of GEM 80 Controller, you can increase speed by putting in, as the final rung of the program, a START OF BLOCK that is never obeyed. This method is given in the sheet for BLOCK earlier in this manual. It works by causing the program to jump to the end, rather than checking through unused locations in case there were any more instructions. It therefore only gives much time saving on the program cycle time if the program does not use up all the available instruction space and where, as a result, a section of the memory is area not being used. If you have used just about all the possible number of instructions, then you won't obtain much speed improvement.

This technique works with older models of GEM 80 Controller only. It will not work on modern models, which use memory rather differently.

NUMBER OF INSTRUCTIONS

The sheets in this manual on "Space Hints" give methods of economizing on the number of instructions you need to perform certain program functions. Normally, reducing the number of instructions will also reduce the execution time of the program.

Note, however, that some Special Functions can give long execution times, e.g. if you used the Special Function MOVE (T20) to copy large amounts of data from one table to another in a single program cycle. In such cases, you should arrange for the function to copy only a limited amount of data on any one program cycle.

THIS PAGE LEFT INTENTIONALLY BLANK

WALK THROUGH

Before you reach the stage of installing your program in the GEM 80 Controller, you should have thoroughly checked it through on paper first, especially any "clever" bits where the logic gets rather involved.

Note also that you will expect some sections of program to work only on rare occasions, or else fleetingly. Sometimes, it is rather difficult to monitor the actual program operation because of the speed at which things are happening, so you need to be certain that these sections do in fact work as you intended.

You should therefore "walk through" the program, checking out on paper whether it will do what you want it to do under all circumstances. It is generally a good idea to get someone other than yourself to do this as well; we seldom pick up our own mistakes.

SECTIONAL TESTING

For obvious reasons, it is not a good idea to try entering a large program into a Controller and expecting it all to work correctly first time. Instead, you should check it out in sections.

If you have a number of Controllers, you may be able to check sections of program on a spare Controller rather than on the target Controller. However, be a little careful about doing this; some Controllers offer more facilities than others, so you cannot be 100% certain that the program will work the same unless you test it on *identical* hardware. Similarly, if you are checking a program section that handles serial link messages between two Controllers, the hardware at *both* ends of the link must be the same as the target hardware for 100% confidence.

If you are entering a program into the actual target hardware, then you can do this a section at a time, building up until the complete program has been entered, and testing each section as you go. Otherwise, you can enter it all at once but temporarily disable sections that you have not yet tested by writing in extra START OF BLOCKs and END OF BLOCKs. You can remove these extra BLOCK rungs afterwards as program testing proceeds.

MONITORING

When using a Programmer to monitor program rungs or the contents of table locations, it is important to remember that the display on the Programmer does not update as rapidly as the program is running in the Controller. With the Controller set to run normally, you won't be able to observe every change that occurs in a rung. You may not notice any contacts that only close for single scans (such as E0 table timing markers). Even if the Programmer were fast enough to show all the changes, you would still not be able to observe them all as they would be too fast for the eye to register.

For some sections of program involving a sequence, you can set the Controller to Single Cycle mode, to allow you to observe the effect on rungs whenever you command another program cycle via the Programmer. However, it can take several program cycles to operate COUNTs with large number VALUEs, or DELAYs with long times set by large VALUEs. You may therefore find it convenient to temporarily edit such VALUEs to smaller numbers while program testing, and edit them back to their proper numbers when you have finished testing. Note that DELAYs do not operate in real time when the Controller is in Single Cycle mode; see sheet for DELAY earlier in this manual.

FORCING INPUTS

Especially when commissioning a plant, it is useful to be able to override certain inputs, and to be able to set certain outputs to values that override what the program decides they should be. This is generally referred to as "forcing" the given inputs and outputs.

To force inputs, what you need to do is to temporarily edit the program by putting in additional rungs at the very beginning of the program. Remember that the normal GEM 80 Controller program cycle consists of:

1. Reading the states and values of inputs into memory

2. Performing the program once, working out what the outputs should be, holding these in memory

3. Writing these output states and values to the physical output modules and units

Consequently, if you put in additional rungs at the very start of the program, you overwrite what the Controller has read for the states and values of inputs, and the program will operate on the values you have given in the rungs. For instance, if you want to overwrite A4.5 and force it to be always ON, you can put in a rung at the start of the program as follows:

```
!                                                                    !
+--------------------------------------------------------------( )-+
!                                                              A4.5 !
```

or if you want it always to be off, then:

```
!                                                                    !
+-<AND>-------------------------------------------------------( )-+
!   0                                                        A4.5 !
```

If you want to force several bits in the same table location, then you can do this with logic elements. For instance, if you want A4.0, A4.1, A4.5, A4.7 and A4.8 to be set ON regardless of the way they are read from the plant, then you can do this in one rung, as shown below:

```
!                                                                    !
+-<AND>---<OR >-----------------------------------------------<OUT>+
!  A4      @1A3                                                 A4   !
```

In this rung, the contents of A4 as read from the plant is ORed with the required pattern of bits.

If, on the other hand, you wanted to force the same set of bits OFF, you could use the following rung:

```
!                                                                    !
+-<AND>---<AND>-----------------------------------------------<OUT>+
!  A4      @FE5C                                                A4   !
```

which only allows through those bits of A4 as read from the plant where there is a 1 in the mask, the remaining bits being forced to 0; that is, where there is a 0 in the mask, the corresponding bit in A4 will be forced to 0. If you have a condition where you want to force some inputs to 0 and others to 1, then the two types of rung can be combined as follows:

```
!         FORCE1  FORCE0                                             !
+-<AND>---<OR >---<AND>-------------------------------------<OUT>+
!  A4     @3186   @BDB6                                        A4   !
```

which you can check forces bits 1, 2, 7, 8, 12 and 13 to 1, and bits 0, 3, 6, 9 and 14 to 0, leaving bits 4, 5, 10, 11 and 15 to remain as read from the plant inputs.

Note that, in the example rung above, any forcing of a bit to 0 will override any forcing by the previous OR to 1. If you prefer forcing to 1 to override forcing to 0, then the second and third elements in the rung, complete with their masks, should be interchanged. If you have specified your masks so that no bit is trying to be forced both ways at once, then the order of the second and third element in the rung is immaterial.

If you have got some spare P-table locations, then it will be easier to use these for the masks, rather than defining them literally in the rungs. The masks can then be checked as actual bit patterns by using the Programmer to display them in binary in a datalist.

FORCING OUTPUTS

For forcing outputs, you can adopt exactly the same approach as for inputs, using identical looking rungs, except that the table locations given with OUTs or COILs will be those of output tables (e.g. B-table), but inserting these rungs at the very end of the program. This will cause the outputs to be forced to states regardless of what the previous program rungs have worked out that they should be.

With output forcing rungs on older models of controller, beware of just one thing - if you have put in a non-obeyed START OF BLOCK at the very end of the program to skip unused instruction space in memory (see the Item on Speed Hints), then these forcing rungs must come just before this START OF BLOCK rung, and not at the very end of the program, otherwise they will not operate.

THIS PAGE LEFT INTENTIONALLY BLANK

PROGRAM CYCLE TIME

For a given GEM 80 Controller and I/O configuration, and for a given program, there is a natural minimum time that it will take for one complete program cycle. This cycle consists primarily of reading the inputs, running the program to determine the required output settings, and then latching these settings into the physical output modules and units.

If the Controller is configured to free run, then the natural cycle time is what will occur in practice.

If you have configured the Controller to have a cycle time *longer* than this, then the Controller will wait towards the end of the cycle before starting the next cycle, so as to achieve the given cycle time.

If you have configured the Controller so as to call for a cycle time *shorter* than the natural cycle time, then the Controller will not be able to match the cycle time you require and, by default, will free run.

TIME DEPENDENT FUNCTIONS

There are a number of Special Functions, such as RAMP and three-term control functions, that must operate at regular intervals if they are to perform correctly. Further, this repetition interval must be of a known value as it affects what values you need to enter to achieve the correct settings. The relation between the numbers entered into data tables and the repetition interval is given in the Software Data Sheets for these functions.

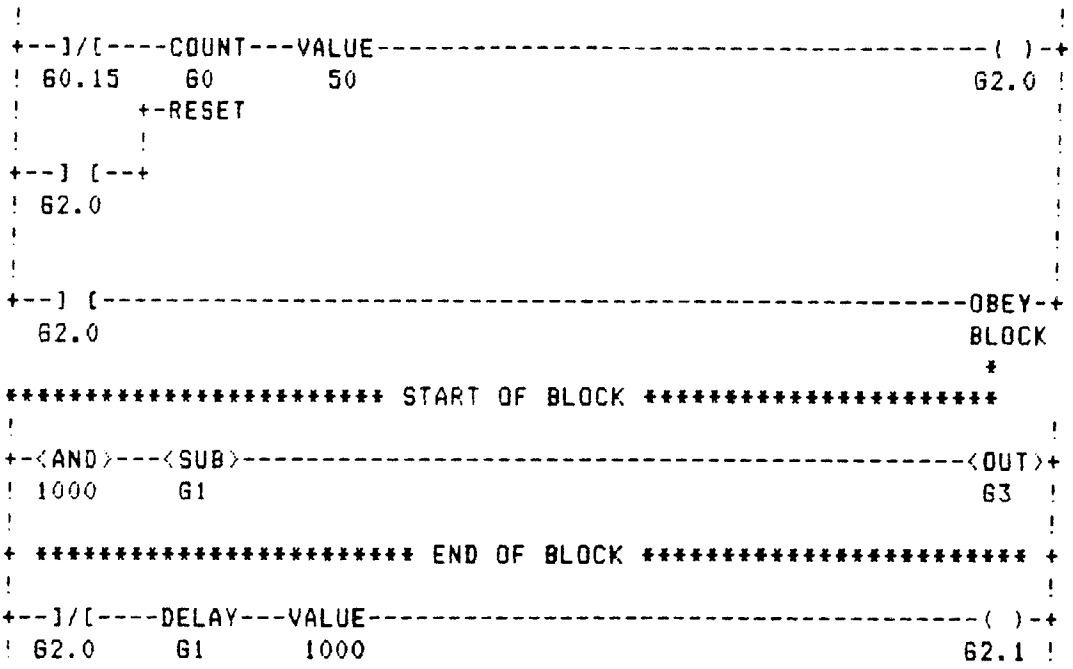
It is therefore important that you know, when configuring the Controller to have a given cycle time, whether this cycle time is being achieved since, if the program cycle time exceeds this value, you won't obtain the characteristics you desire from the Special Functions.

MEASUREMENT OF PROGRAM CYCLE TIME

Modern models of GEM 80 Controller include table location E7 in which the Controller stores the program cycle time to the nearest 10 ms. On older models of Controller, you can use the section of program given overleaf to measure the average cycle time that the Controller uses to get through 100 cycles of program.

Remember that, if your program contains BLOCKs of rungs that may conditionally be skipped, then every cycle through the program will not take the same length of time (if the Controller were free running), so that a spot check on the time for just one program cycle would not be representative.

The section of program is intended to be inserted temporarily at the end of your program. It uses four arbitrary table locations G0 to G3; you should substitute any suitable locations not used in your program.



The first rung increments the COUNT on every other scan. This is because G0.15, used for the input to the COUNT, is the bit used by the COUNT to remember the state of the input on the previous scan (see sheet for COUNT). With a VALUE of 50, G2.0 COIL will therefore be set ON after 100 program cycles.

When G2.0 goes ON, it resets the COUNT back to zero, and also causes the BLOCK to be obeyed.

However, prior to G2.0 going ON, the DELAY in the last rung operates, and carries on timing out all the time until G2.0 goes ON. The VALUE associated with the DELAY has been set arbitrarily at 1000, this number being made sufficiently large for the DELAY not to have timed out in 100 program cycles. The number in G1 therefore counts down from 1000 until such time as G2.0 goes ON, when the DELAY is reset, and the contents of G1 will start again from 1000 the next time G2.0 goes OFF. Consequently, there should never be any output from the DELAY, and G2.1 never operates - it is just a dummy output that has to be included to complete the rung.

After 100 program cycles when G2.0 goes ON, the BLOCK is obeyed. The rung within the BLOCK subtracts G1 from 1000, to obtain the number of delay increments that G1 has counted down by from 1000, and places this in G3 where it can be monitored. G2.0 being ON also causes the DELAY to be reset, and the operation of the program then repeats.

For Controllers with a delay increment of 0.1 s (100 ms), then G3 will contain the number of 100 ms increments for 100 scans, i.e. it will give the average cycle time in milliseconds directly. For Controllers with a delay increment of 0.01 s (10 ms), the figure in G3 will be the average cycle time in ms multiplied by 10. Instead, you could change the VALUE in the first rung to 5, giving an average over 10 program cycles only, when the figure in G3 would again be directly in milliseconds.

MONITORING

For a Controller that has been set to free run, the cycle time is increased fractionally when a Programmer is plugged in and set to monitor the Controller operation, either when monitoring rungs or when monitoring a list of data table locations. This increase in cycle time is due to activity on the serial link between the Controller and Programmer.

If the cycle time is measured, either by looking at table location E7 or by using the program given earlier, then the additional time for monitoring will be included in the measured figure. Although the Controller will run fractionally faster when the Programmer is disconnected from the Controller, you should design on the basis of the measured cycle time since you will almost certainly want to monitor the Controller at some time in the future when the plant is running.

PROGRAM CYCLE TIME TOO LONG

If the program cycle time works out to be longer than what you require, some improvement in speed may be possible. See the Item "Speed Hints" in this Section.

There will, however, be a hardware limit on how fast a program can run for a given Controller and a given I/O configuration. Figures of the times it takes to handle different types of I/O are given in the Technical Manual for the particular Controller. The manual also gives times for the execution of the various program instructions.

Note two particular problems that can arise if the program cycle time is too long. First, if you have any fleeting input signals that are present for a length of time shorter than the program cycle time, the possibility exists that these signals will not be seen by the Controller at all. Second, if you are using any counter input modules or units where the input pulse rate is sufficiently high, you may fill the counters up before the end of the program cycle (i.e. an overflow condition). Obviously, this is more likely on 8-bit counters than on 16-bit ones.

SERIAL LINKS

Remember that it takes a certain length of time for data to be sent over a serial link from one Controller to another. See the Item in this Section devoted to serial links.

TIMING MARKERS

GEM 80 Controllers include timing markers in E-table. For most controllers, these are:

E0.0 : 0.1 s marker
E0.1 : 1 s marker

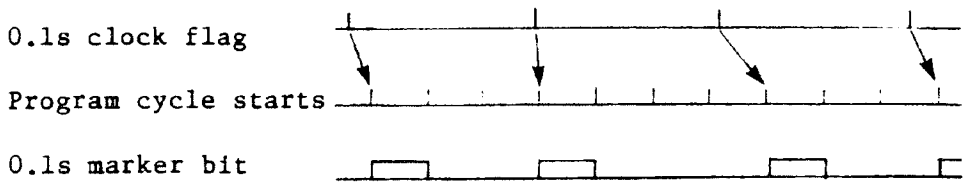
For other Controllers with faster cycle times, they may be:

E0.0 : 0.01 s marker
E0.1 : 0.1 s marker
E0.2 : 1 s marker

Details will be given in the Technical Manual for the particular Controller.

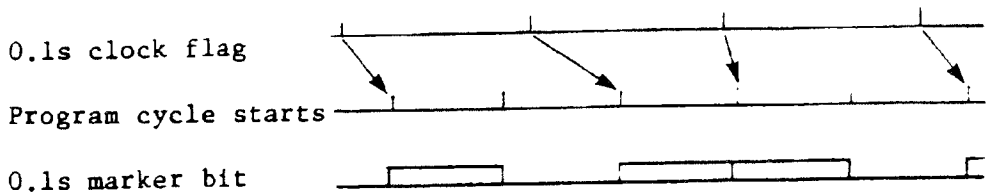
Within the Controller, there is effectively a clock that sets internal flags whenever another 1 s, 0.1 s, and maybe 0.01 s, has passed. These internal flags are checked at the start of any program cycle to see whether they have been set. If any of them have, they are cleared (ready for the next time round on the clock) and the appropriate E0 bit set for use in the program. At the end of the program cycle, these E0 bits are cleared again.

This will be made clearer by a timing diagram. Suppose that the program cycle time is 30 ms (0.03 s) and that we consider the 0.1 s timing marker; then the timing will be as shown below:



Note that, with a 30 ms program cycle time, the 0.1 s markers cannot be equally spaced, but that they will be at 0.1 s intervals on average. With the same program cycle time, the 1 s markers would be spread much more evenly.

However, now consider what happens to the 0.1 s markers if the program cycle time exceeds half of 0.1 s, i.e. if is is greater than 50 ms. The timing diagram for a 60 ms program cycle time for the 0.1 s marker bit in E0 would be:



You will see that, in this case, the 0.1 s marker bit is sometimes set ON on successive program cycles. Only if the program cycle time is less than half of 0.1 s can it be guaranteed that the marker will not remain ON for two successive cycles.

Exactly the same type of effect would occur with the 1 s marker if the program cycle time were longer than 0.5 s, and with the 0.01 s marker (if available in the particular Controller) if the program cycle time were longer than 5 ms.

Where the program cycle time is such that a timing marker may be ON for successive program cycles, then you must be careful not to use it for anything depending on transitions from OFF to ON or vice versa. A particular instance is the COUNT element (see sheet for COUNT earlier in this manual). You cannot count timing markers to determine an elapsed time if the interval between timing markers is less than twice the program cycle time. In such cases, you should redesign your program rungs to use DELAYs rather than timing markers and COUNTs, or else you should use timing markers of the next longer interval.

INPUT STAGGER

At the start of any program cycle, the Controller reads the state of all the inputs. It doesn't look at them again till the next program cycle. We sometimes think of this as though the Controller were taking a snapshot of the inputs, at the start of each cycle.

However, this is not an entirely accurate explanation of the way the Controller operates. Although it takes the Controller a very short time to read the state of the inputs and load this data into input tables (e.g. A-table, C-table, etc.), this time is finite, and the data is not read all at once but from a module or unit at a time.

Thus, if input signals are changing at the start of a program cycle while their states are being read, some modules will have their states read before others. If all the signals happened to change simultaneously, some might get read before the change, others after the change. We must not fall into the trap of assuming that all input data read at the start of any cycle necessarily represents the state the inputs from the plant were in "at a particular time", especially if we are wanting to know which of two events happened first.

Second, if we consider an input module such as an 8-way Input on the Basic I/O Highway, where the Controller reads all 8 bits (truly) simultaneously, the 8 bits read may not necessarily represent the state of the inputs. This is because, with logic input modules, each input has a filter circuit to eliminate electrical noise and cross talk, and even if the highest precision components were used, it would not be possible to guarantee that 8 different filters had precisely equal characteristics. If all 8 ways were to change, on the plant side, simultaneously, then it is possible that if the Controller read the data from the 8-way input module at precisely the right instant, it might register that some bits were ON and some were OFF.

Note, however, that input modules specifically designed to provide numbers to the Controller (e.g. analog and counter modules) are designed so that the number cannot change in the middle of being read. It is not therefore possible to read numbers that are spurious transiently; they will either be numbers representing the state before or after a change, but not some spurious intermediate value.

Where one must be particularly careful is therefore in using logic modules, designed for inputting separate bits, for inputting numeric data. Because of the small difference in filter time constants, it is possible for the bits not to change all at once, so that if the input changes, say, from 00000111 to 00001000, then it is possible to read a transiently spurious value of 00001111 or maybe 00000000, or other intermediate values. Therefore, where absolute shaft encoders are used, we recommend that they should be in Gray code and not pure binary, so that on transitions only one bit changes on any number change. Gray codes are also often referred to as unit-distance codes.

THIS PAGE LEFT INTENTIONALLY BLANK

GEM 80 Controllers, especially newer models, provide a wide range of serial communications facilities, all of which we can't possibly cover in a manual on programming. We therefore give below some of the simpler ways of using serial ports, and leave you to read the Serial Communications Manual (publication T457) for more details.

SERIAL PORTS

The usual way for two GEM 80 Controllers to communicate with each other is via serial links. Different models of Controller have different numbers of serial ports. The quantity of Controllers that you can connect to any one serial port also varies with the model. Refer to the Technical Manuals for your particular model or models of Controller. Also refer to the Serial Communications Manual.

An extension to Serial Links is the CORONET Local Area Network (LAN) system. Details of this are given later, on Page 104, after we have dealt with simple serial links. The maximum quantity of Controllers you can connect to a CORONET LAN is 32 using a single LAN, although this number can be extended with subnetworks.

J/K-TABLE MESSAGE EXCHANGE

Ignoring serial ports provided by Serial Comms modules on a Fast I/O highway, messages between Controllers are transmitted from the K-table in the first Controller and received into the J-table in the second Controller. Similarly, messages passing along the serial link in the reverse direction are transmitted from the K-table in the second Controller and received into J-table in the first Controller.

If you set a serial port to free running mode (the simplest mode), then, to send a message, all you need to do is to get your program to put data into the appropriate K-table locations. You can do this using COILs, OUTs or, if available, the Special Functions LOCATE and MOVE. The Controller firmware will then look after the transmission of the data. You can therefore treat the K-table in a similar way to other output tables, such as the B-table or D-table.

Similarly, messages received go into J-table, which you can use like other input tables such as the A-table or C-table, by putting contacts, ANDs, Special Functions, etc. in your program with the appropriate J-table addresses.

However, in most applications, it takes longer than a single program cycle to send and receive messages. Consequently, the contents of the J-table will not get updated like the other input tables before every program cycle, and what you put into K-table will not get transmitted on every program cycle. This means that, although you can often treat the J-table and K-table as normal input and output tables respectively, there are differences from the other input and output tables in the way that they operate.

J-TABLE AND K-TABLE LOCATIONS

In any Controller with more than one serial port, certain J- and K-table locations are allocated to each port. Further, where any of the ports can talk to more than one other Controller (i.e. on a multi-drop link or CORONET LAN), certain J- and K-table locations may be allocated to messages for each Controller on that link.

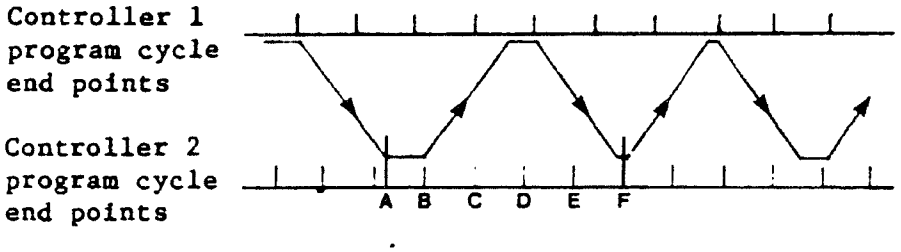
The quantity of J- and K-table locations set aside for each message route or channel determines the maximum message length that can be handled. You don't have to use messages of maximum length if you don't want to; in fact, because longer messages take longer to transmit, you may well find it better to keep messages short, but to transmit more of them. In this case, you will not be making use of all the available J- and K-table locations.

In the descriptions below of how messages are exchanged, you must take it as understood that, where we talk about a message being received into J-table or being transmitted out of K-table, this refers not necessarily to the whole of the J-table or K-table, but only to the relevant locations being used for a particular message route or channel.

FREE RUNNING MESSAGE TRANSFER

Some models of Controller have certain flag bits that enable you, if you so wish, to control by program when messages are sent, and other flag bits to indicate when messages have been received. If you don't make use of this facility, or if it is not available on your particular model of Controller, then the message traffic along the serial link will be controlled solely by the firmware in the Controller. We refer to this as "free running" message transfer.

For a point-to-point link between two Controllers, how automatic message transfer works is best understood by reference to a typical timing diagram, as below:



We begin by considering Controller 2 receiving a message from Controller 1. The characters coming down the serial link do not directly go into J-table, but first go into a buffer memory area. The Controller firmware checks when the message is complete and then checks it out for no corruptions. (If there are any, Controller 2 will send a negative acknowledgement back to Controller 1, calling for the message to be retransmitted).

Assuming that Controller 2 receives a valid message that is complete and checked out at time A, the message does not get transferred to J-table till the end of the program cycle, at time B. Therefore, just as with other input tables, the contents of the J-table cannot change midway through the execution of a program cycle.

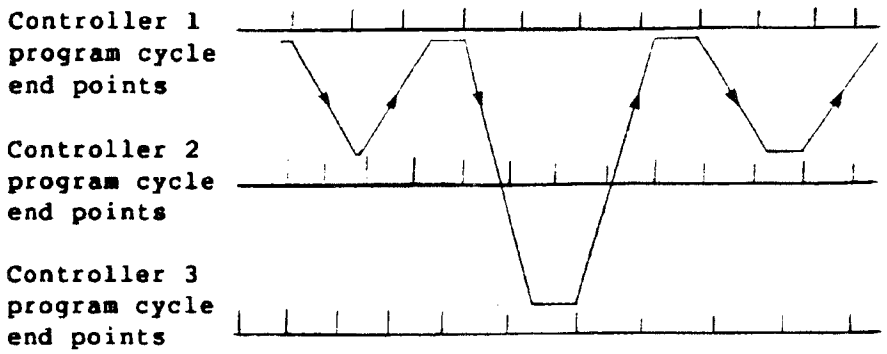
Under free running message transfer, the receipt of a message into J-table initiates the sending of a message back from Controller 2 to Controller 1, also at time B. Controller 2 copies whatever happens to be in K-table at the end of the program cycle into an output buffer memory area, from which it transmits the actual message.

There are two important points to note about free running message exchange. First, any message transmitted consists of what was in K-table *at the particular moment* (e.g. at B) when the input message got transferred to J-table. If you keep on altering the K-table contents on every program cycle (e.g. during the program cycles ending at C, D or E in the example diagram), then this data will not get transmitted as messages. It is only at time F that another message gets transmitted, and this will consist of what you have left set in K-table at the end of the program cycle between E and F.

Second, the output of a message is initiated *immediately* following receipt of a message into J-table, with no intervening program execution cycle. This is like letters crossing in the mail; the message you are sending cannot be influenced by the message just received, because you haven't yet "read" the incoming message. For those applications where input messages constitute, in coded form, a request for particular table contents to be transmitted back to Controller 1, then a reply message containing this data cannot be sent until the *next* input message is received (e.g. at time F rather than at time B).

MULTIDROP LINKS

So far, we have only considered a serial link of the point-to-point type, with one Controller on either end of the link. If you have more than two Controllers on one link, then one of the Controllers must be given the task of controlling the message traffic on the link. In its configuration data in P-table, you must have set up this Controller's serial link port as a "control" port; the other Controllers must be set up as "tributaries". For a link with three Controllers, where Controller 1 provides the message traffic control, a typical timing diagram is as given below:



Note that all messages travel to or from the Controller that controls the message traffic; no messages can pass directly between Controller 2 and Controller 3. Controller 1 sends messages alternately to each of the two tributary Controllers. (If there were more tributaries, it would send messages to them in rotation).

Note that, compared with a point-to-point link, the frequency of any particular Controller receiving messages is less (apart from the one directing the traffic), assuming that everything else is the same (i.e. message length, data rate).

Where you use free running message transfer, the method of operation is the same as for point-to-point links, i.e. initiation of message transmission from K-table occurs immediately after a received message gets put into J-table. The only difference is at the Controller providing the control port, where receipt of a message from one tributary initiates the sending of a message to the next tributary.

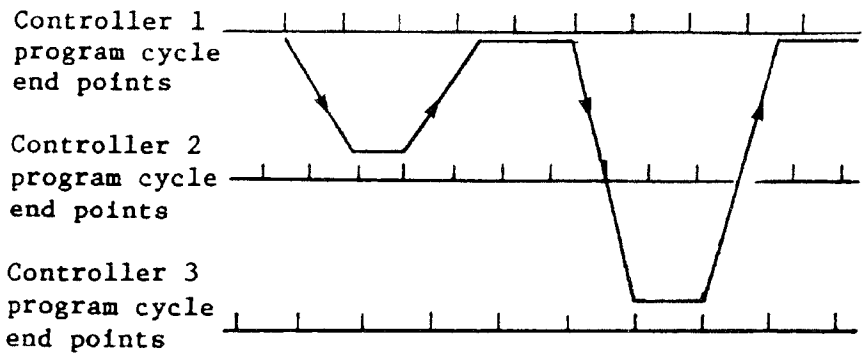
USER CONTROL OF MESSAGE TRANSFER

On those models of Controller that include message control flag bits, you can delay messages being sent until you set the appropriate flag bit. This allows you to send reply messages to incoming messages one program cycle later, if you so wish, instead of having to wait for the next incoming message. Instead of messages "crossing in the mail" (as happened with free running message transfer), you can delay your reply by one program cycle. During this cycle, your program can work out the required reply message, put this into K-table, and then set the control flag bit to tell the Controller to send this message at the end of the program execution.

Of course, you could delay messages by more than one program cycle if you wanted to. For instance, you might not want to send a reply until some particular input condition arose. However, note that, while you hold up giving a reply, no other messages can travel along the serial link, i.e. all message traffic comes to a halt. It is therefore usually better to delay any reply by a single program cycle only, and to give a "nothing" reply if there is nothing to report. This is particularly important on multi-drop links, as no other tributary Controllers that have important data to send can do this if message traffic is halted.

To prevent you inadvertently halting message traffic in your program, and to guard against equipment faults, the Controller includes a preset P-table timeout value. When this is reached, the Controller providing the control port moves on to the next Controller on the link. Timeout periods are given in the Serial Communications Manual.

A typical timing diagram where user-control of message transfers is used to delay replies by one program cycle is given below for a serial link with two tributaries; compare with the previous timing diagram:



Note that the frequency of messages is slightly reduced because of the program cycles intervening between message receipt and message transmission. If all that the tributary Controllers have to do is to send the latest values of a fixed set of table locations, there is no advantage in user-control over free running message exchange. If, on the other hand, Controller 1 needs to request different information on different occasions from Controllers 2 and 3, then it can get its information from them more rapidly if the programs for the Controllers make use of the message control flag bits; Controller 1 gets its information on the first messages back from them, not the following ones, avoiding the "crossing in the mail" problem.

Supposing that a Controller has user-control flag bits (for the particular message route or channel) I0.8 to indicate receipt of a message, and I0.0 to initiate the output of a message. Then the section of program controlling message transfer might be written something like that shown below:

```
!                                                                    !
+-(AND)-------------------------------------------------------( )-+
!   0                                                          I0.0 !
!                                                                    !
+--] [-------------------------------------------------------OBEY-+
  I0.8                                                       BLOCK
                                                              *
************************ START OF BLOCK ************************
!                                                                    !
+-(AND)-------------------------------------------------------( )-+
!   0                                                          I0.8 !
!                                                                    !
!        LOCATE          MOVE                                        !
+-(AND)---SPEC.---VALUE---SPEC.---VALUE----------------------(OUT)+
!   0      S20     W220     T20      8                         G102 !
!                        +-(   )                                     !
!        LOCATE          !                                           !
+-(AND)---SPEC.---VALUE-+                                           !
!   0      S20     K0                                               !
!                                                                    !
!                                                                    !
+-------------------------------------------------------------( )-+
!                                                              I0.0 !
!                                                                    !
+ ************************ END OF BLOCK ************************ +
```

This section of program first sets I0.0 to zero, to prevent any message being sent until we are ready. If a message has been received, I0.8 will have been set to 1 at the end of the previous program cycle, in which case the BLOCK will be obeyed. Once in the BLOCK, we immediately set I0.8 back to zero, to make sure that the BLOCK will not be obeyed again until such time as another message is received.

Next, output data is put into the relevant K-table locations. (In this simple example, we have simply put 8 words of data from W220 to W227 into K0 to K7; in practice, you might have some decision -making rungs deciding what to load into K-table depending on what was in J-table). At the end of the BLOCK, we set the flag bit I0.0 to 1, to signal that a message is ready for transmission. At the end of the program cycle, this flag bit will cause the K-table contents to be transferred to the output buffer, from where it will get transmitted. Note that this last rung in the BLOCK overrides the way that I0.0 was set prior to entering the BLOCK.

CONTROLLERS WITH NO USER CONTROL FACILITY

If your Controller does not include the user-control facility, you may still wish to know when a message has been received. This might occur where you need to send a sequence of data, or where you need to send what data another Controller requests, because, as discussed earlier, there is no point in updating the K-table contents on every program cycle. All you need to do is update it once after a message has been received, and not update it again until the next message is received.

Depending on the type of messages being received, there are a number of ways of doing this. First, you could compare the contents of a J-table location with its contents on the previous program cycle, e.g.:

```
!                                                                    !
+-(AND)---(XOR)-----------------------------------------------OBEY-+
  W100     J0                                                 BLOCK
                                                               *
************************ START OF BLOCK ************************
                                    •
                        (BLOCK of program)

!                                                                    !
+-(AND)-------------------------------------------------------(OUT)+
!  J0                                                          W100  !
!                                                                    !
+ ************************** END OF BLOCK ************************** +
```

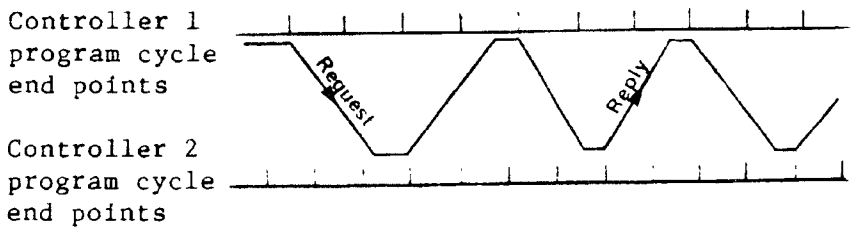
This will work all right as long as every message provides a contents into J0 different from the contents provided on a previous message. If this is not so, then you could use:

```
!                                                                    !
+-(AND)-----------------------------------------------------OBEY-+
   J0                                                        BLOCK
                                                              *
************************ START OF BLOCK ************************

                        (BLOCK of program)

!                                                                    !
+-(AND)-------------------------------------------------------(OUT)+
!    0                                                          J0   !
!                                                                    !
+ ************************** END OF BLOCK ************************** +
```

This second version will work all right, even if successive messages put the same data into J0, because J0 is set back to zero each time the BLOCK is executed. However, even this second version will not work if the message itself can put zero data into J0. If this is a situation you could encounter in practice, then you may need to make at least one bit equal to 1 in every message, to act as a flag bit, just to make sure that receipt of a new message can be detected.

With any of the above schemes, it is important to note that you have no facility to delay the outputting of messages, even though you can detect when a message has been received. A message will be sent immediately after a message is received, before the program cycle in which you detect receipt of a message begins.

Consequently, if an incoming message requests the Controller to output certain data, it can't do this till the following message. For a point-to-point link between two Controllers, the timing diagram would then be as shown below:



CORONET LOCAL AREA NETWORK

The CORONET Local Area Network system is unlike the simple serial links we have been describing in that no Controller is a master. With CORONET, any Controller can initiate a message to any other Controller. The Controllers do not have to talk to each other in a fixed order, and all messages are sent under user-control. As for normal serial links, you use K-tables for sending output messages and J-tables for receiving input messages.

Since no Controller is the master on a LAN of the CORONET type, then two or more Controllers can attempt to send messages at once. This is referred to as a "collision". When a collision occurs, all the Controllers attempting to send messages get off the line and have another go later.

To avoid collisions, any Controller wanting to send a message first listens in on the network bus to make sure that no other Controller is already talking. It is only when two Controllers try talking at precisely the same instant that a collision occurs, so collisions are relatively infrequent.

A Controller may therefore be held up in sending a message because the network is already busy. The length of time it may have to wait for the network to become free depends on how many other Controllers want to talk at the same time. As a result, message transmission times cannot be guaranteed but only quoted on a statistical basis.

Any message you send using CORONET must contain the destination address. That is, you must put in the "phone number" of the Controller you are sending the message to. Your program must put this number into I-table. Any other Controller on the network will ignore messages being sent unless they include its own address number.

The CORONET interface automatically puts the number of the sender in the message on the network. The Controller at the receiving end can therefore tell from where any message is being received.

For further details of CORONET, refer to the Serial Communications Manual, T457.

MESSAGE TRANSMISSION TIMES

You may need to estimate how long it takes for a message to be transmitted from one Controller to another.

Messages consist of a certain number of bytes, or groups of 8 bits. One byte is therefore half the length of the amount of data held in one table location. If you decide on a message length of n table locations, then this is equivalent to 2n bytes.

In addition to the actual data, each message includes 7 other bytes on a serial link, or 8 other bytes on CORONET. The functions of these extra bytes are given in the Serial Communications Manual. The message length is therefore 2n + 7 bytes on a serial link, or 2n + 8 bytes on CORONET.

Also, each byte in the message is "framed" by a start bit and a stop bit. Each character in the message is therefore transmitted as 1 + 8 + 1 = 10 bits. The total number of bits is consequently (2n + 7) x 10 for serial links, or (2n + 8) x 10 for CORONET.

Knowing the data rate that you have configured a serial link to run at in bits/second, you can then work out the transmission time. Thus, if you have a message length of 4 table locations, the normal number of bits in a message will be 15 x 10, or 150 bits. If your data rate is 2400 bits/s, then the tranmission time would normally be 150/2400 or about 63 ms. It is not quite so simple to work out the timings on CORONET because, as mentioned earlier, times can only be given on a statistical basis. See the Serial Communications Manual for details.

Since the data you are transmitting can consist of any pattern of bits you like, it is possible that on odd occasions one of the data bytes corresponds to one of the characters normally used to tell the Controller that a message is starting or ending. So that the Controller at the receiving end cannot be misled by such data, the Controller at the transmitting end stuffs in extra preceding characters, which inform the receiving Controller not to treat any following byte in the message as being a start or end of message but to treat it as data. Consequently, although in most cases messages will consist of 2n + 7 or 2n + 8 bytes, on odd occasions they can be a few bytes longer.

If you are looking at how long it takes to send a message and get a reply back over a serial link, then you must not simply double the figure but allow for the time it takes to complete a program cycle on each of the Controllers. An incoming message could be completed just before the end of a program cycle, in which case there would be very little delay in its being transferred into J-table. On the other hand it could be completed just after the end of a program cycle, in which case the transfer would not occur till almost a whole program cycle later. All you can therefore use in calculations is a mean figure of half a program cycle.

Note that the program cycle times on different Controllers will generally not be the same. If you look at the timing diagrams shown earlier, you will see that, to make them typical, the program cycle times were not shown as being equal. Even if you have Controllers configured to have identical program cycle times, there will be nothing to make sure that the program cycle end points are synchronized. Consequently, you can never guarantee precisely how long any individual message will take, but only work out an average figure.

If message transfer is not free running but under user-control, then you will have to add the times for any additional program cycle delays you have written into your program.

Last, if your serial link is not point-to-point but is multi-drop, then you have got to allow for the time taken by message traffic to and from other Controllers on the link. Where you need short intervals between messages, you may need to keep message lengths short if you have got several Controllers on one link.

RE-TRIES

All the above description of how to work out message times was based on all message traffic operating normally. However, there are two cases where message traffic may be disrupted, and transmission times get longer than those calculated.

First, if there were strong outside electrical interference, a message might get corrupted. If the receiving Controller finds that a message does not check out with its CRC check code, it will give a negative acknowledgement and ask for the message to be retransmitted. This will cause the message transmission time to double. If the second version of the message is still corrupt, the transmitting Controller will repeat it a third time, and so on, but only up to a limited number of times. The actual number of re-tries at getting a message through is given, for the particular model of Controller, in the Technical Manual. When this number is exhausted, the Controller providing the control port to the link (i.e. the Controller in charge of the message traffic) will move on to the next tributary Controller on a multi-drop link. In this case, the message never gets through at all.

Second, if any tributary Controller is taken out of service on a multi-drop link, the Controller providing the control port will still continue trying to talk to it. It will therefore spend some time performing its allotted timeout period and number of re-tries before moving on to the next tributary. This, again, can increase message transmission times.

To monitor the performance of serial links, certain F-table locations are provided indicating the number of message re-tries. If these statistics indicate that an abnormally high number of messages are not getting through first time, and if you have not had Controllers out of service, you should check on whether serial links have been cabled in accordance with the practice recommended in your Controller's Technical Manual.

SERIAL COMMS MODULES

If your model of Controller includes Serial Comms modules on a Fast I/O highway, then you can have additional serial ports. The input and output messages will, however, go into and come out of C-table and D-table locations, as given in the Technical Manual for the particular type of module, rather than J- and K-tables. The same general principles of message transfer and message timing still apply, as for standard serial ports.

If your Controller has the programmable type of Serial Comms module, however, there are certain things you can alter, such as talking to a number of tributaries on a multi-drop link in a variable, rather than a fixed, order. Details of how to program such modules, which use a language called COMMUNE (a special adaptation of BASIC) are outside the scope of this manual. Facilities are also included for communicating with types of system other than GEM 80.

This Section wasn't in the first issue of the GEM 80 Programming Manual. We didn't think it was necessary. We thought that everything written below was so obvious that it didn't need saying. However, visits to the sites of a number of customers have proved us wrong. Full marks, then, if you are doing everything we describe below on your installation.

SPARES

All GEM 80 hardware is designed to be put back on stream by replacing faulty modules with operational ones. To help you identify which module has failed, a GEM 80 Controller gives you fault codes and messages. You can then replace the faulty module with a spare - or can you ? Make sure that:

- You have got adequate spare modules.
- They are of the correct types (read the small print!).
- They are readily accessible, and don't take wasted minutes to locate.
- You are certain that these spare modules work. (Have you ever tried them out ?)

Next, you need your faulty module repaired. Make sure that you mail it to us *immediately*. Don't put it back into your stock of spare modules, otherwise it will get mixed up with your operational spares, and you won't know which ones work and which ones don't. Make sure that you also enclose a note, or tie a label to the module, telling us what the fault you found was.

All the above comments on spares may seem plain common sense. But you'd be surprised how many customers we have come across who don't follow these simple rules !

EVENT LOG

Occasionally, a customer will report to us some fault regarding hardware or firmware. As we are very jealous of our reputation for equipment reliability, we obviously want to sort out his complaint and, if it is our fault, to put the design of the equipment right.

What we often find is that our customer can't tell us under what conditions the fault occurred. We get different stories from different people, and nobody is quite sure what precisely happened. Why ? Because nobody wrote it down *at the time*.

We therefore recommend that you maintain an event log. This could be just for the GEM 80 equipment, but you will probably find it more useful if you record faults and events on what the GEM 80 is controlling as well. If you get a GEM 80 fault, you can then present us with hard-and-fast facts about the conditions under which the fault occurred.

We have included a blank Event Log Form in the next Section of this manual, and a fictitious example of how it might be filled in. You can use this form yourselves or modify it to suit your own needs. Note that you should record the times at which things happened. If you get a GEM 80 fault, you should log the fault code it gave you.

Don't wait till the plant is commissioned before starting your event log. Write down events happening during commissioning, including when you take recordings of programs, etc.

EVENT LOG PRINT-OUTS

You can automate some of your event logging if you have a printer permanently connected to one of your serial ports, and if you write your program to print out event messages when particular conditions arise, together with the date and time, of course.

Note, however, that how good this print-out will be depends on what conditions you have programmed the GEM 80 to record. There may be other fault conditions you haven't considered. Further, if you get a fault that trips the GEM 80 Controller, then you have lost the ability to print. Therefore you *still* need the handwritten log, even with a printer.

CONTROLLERS WITH RAM MEMORY

Most customers use GEM 80 Controllers in which the program and P-table are stored in battery-backed up RAM. Further recipe data (e.g. W-tables) is also normally held in RAM. Yet other data that may be stored in RAM is IMAGEM video format and function definitions. Only if your programs, preset and recipe values are completely firmed up is it worth transferring this data to EPROM memory. During commissioning, you are almost certainly going to use RAM memory.

Now, what happens if you get a RAM memory module fault ? You obviously replace the faulty module, but you have also got to reload the program and P-tables, or video programs, or recipe data. So, besides spare modules, have you also got an up-to-date cassette or disk readily accessible, along with your spare module ?

Surprising as it may seem, we have come across customers who have only got so-called "nearly up-to-date" recordings, customers with several recordings that aren't labelled so that they don't know which one is which, and even customers with no recordings at all !

RECORDING ROUTINE

As a matter of routine, you should make a new recording after *every* modification you make to a program or P-table after the plant has been commissioned. But don't just overwrite your previous recording - use a fresh cassette or disk (or else a very old one you definitely no longer need). *Keep* your previous recording so that, if your new modification doesn't work very well, you can go back to your previous version of program.

Also, use a separate cassette or disk for each program. With cassettes, don't use one side of the tape for one program and the other side for an entirely different program. If you do this, then, with modifications, the program issues will get out of step, and you will have difficulty filing your cassettes. It may sound expensive using separate cassettes, but it's cheaper in the long run.

You should always keep an *absolute minimum of three* recordings - the current version, the version one modification ago, and the version before that.

Keeping back versions like this also means that, if you are ever unfortunate enough to get a cassette screwed up or a disk head crash that destroys the recording, you have still got the previous version. It is a lot less effort, and considerably quicker, to reconstruct your program from the previous version than starting from scratch.

During commissioning when you are doing lots of modifications to the program, you should take a fresh recording at least once a day. Always take a recording last thing before you go home or back to the hotel. Stop work earlier than you might otherwise have done, to give yourself time to go through the routine of making a recording of the program and P-table, with the modifications you have made during the day, and, of course, verifying the recording. After that, free from worry about anything happening during the night, you'll find you sleep peacefully.

As often as possible, get a print-out off your programs, of the *same versions* as those you recorded. It can be very confusing if the only print-outs you have got are half-way between two versions of recording, as you never know which modifications shown in the print-out are included in your recording and which are not.

Use your event log to note down when you took recordings of the programs, always listing the date and time as well.

Don't forget to label your cassettes or disks so that you can immediately see what they are for and when they were last updated. If you have re-recorded them so often that there's no more space on the label, then GEC can supply you with additional labels.

SOFTWARE

By software, we mean your ladder diagram program. We have tried to give you guidance in this manual, and in the Software Data Sheets for Special Functions, on how to write your programs. If you write programs with bugs in them, then it is generally your own job to sort them out.

However, if you have come unstuck because of some point we haven't explained adequately or have failed to explain at all, then please draw this to our attention. We have provided a form at the back of this manual for reporting any problems on documentation back to us.

THIS PAGE LEFT INTENTIONALLY BLANK

PROGRAMMING AIDS

CONTENTS

GEC Industrial Controls Limited

REF.

SERIAL LINK ROUTE No.

GEM 80/SERIAL LINK ROUTE
TABLE LOCATION USAGE RECORD

NOTES:-

FROM			TO	
PORT			PORT	
MNEMONIC	OUTPUT ADDRESS	DESCRIPTION/REMARKS	INPUT ADDRESS	MNEMONIC

ISSUE	A	B	C	D	
DATE					
AUTHOR					

GEC Industrial Controls Limited

REF	
ADDRESS RANGE	

GEM 80/TABLE LOCATION BIT USAGE RECORD

MNEMONIC	ADDRESS		DESCRIPTION/REMARKS
		.0	
		.1	
		.2	
		.3	
		.4	
		.5	
		.6	
		.7	
		.8	
		.9	
		.10	
		.11	
		.12	
		.13	
		.14	
		.15	
		.0	
		.1	
		.2	
		.3	
		.4	
		.5	
		.6	
		.7	
		.8	
		.9	
		.10	
		.11	
		.12	
		.13	
		.14	
		.15	
		.0	
		.1	
		.2	
		.3	
		.4	
		.5	
		.6	
		.7	
		.8	
		.9	
		.10	
		.11	
		.12	
		.13	
		.14	
		.15	

ISSUE DATE AUTHOR	A	B	C	D	

 06.88 (Electronic Master)

GEC Industrial Controls Limited

REF.

ADDRESS RANGE

GEM 80/TABLE LOCATION USAGE RECORD

MNEMONIC	ADDRESS		DESCRIPTION/REMARKS	* BIT DEF
		0		
		1		
		2		
		3		
		4		
		5		
		6		
		7		
		8		
		9		
		0		
		1		
		2		
		3		
		4		
		5		
		6		
		7		
		8		
		9		
		0		
		1		
		2		
		3		
		4		
		5		
		6		
		7		
		8		
		9		
		0		
		1		
		2		
		3		
		4		
		5		
		6		
		7		
		8		
		9		
		0		
		1		
		2		
		3		
		4		
		5		
		6		
		7		
		8		
		9		

* TICK IN THIS COLUMN IF THE USAGE OF BITS IS DEFINED ON A SEPARATE SHEET

ISSUE DATE AUTHOR	A	B	C	D	

 06.88 (Electronic Master) **ICE 095**

GEC Industrial Controls Limited

REF.

ADDRESS RANGE G400 - G449

GEM 80/TABLE LOCATION USAGE RECORD

MNEMONIC	ADDRESS	DESCRIPTION/REMARKS	* BIT DEF.
TK1DIST	G400	TRACK 1 DISTANCE TRAVELLED SINCE REC 1	
TK2DIST	G401	TRACK 2 " " " "	
TK3DIST	G402	TRACK 3 " " " "	
TK4DIST	G403	TRACK 4 " " " "	
TK5DIST	G404	TRACK 5 " " " "	
TK6DIST	G405	TRACK 6 " " " "	
TK7DIST	G406	TRACK 7 " " " "	
TK8DIST	G407	TRACK 8 " " " "	
TK9DIST	G408	TRACK 9 " " " "	
TK10DST	G409	TRACK 10 " " " "	
	0		
	1		
TK1TARG	G412	NEXT TARGET DISTANCE, TRACK 1	
TK1BITS	G413	CONTROL BITS FROM CONTROL STORE, TRACK 1	✓
TK2TARG	G414	} DITTO, ~~TRACK 2~~	
TK2BITS	G415	}	✓
TK3TARG	G416	} DITTO, ~~TRACK 3~~	
TK3BITS	G417	}	✓
TK4TARG	G418	} DITTO, ~~TRACK 4~~	
TK4BITS	G419	}	✓
TK5TARG	G420	} DITTO, ~~TRACK 5~~	
TK5BITS	G421	}	✓
TK6TARG	G422	} DITTO, ~~TRACK 6~~	
TK6BITS	G423	}	✓
TK7TARG	G424	} DITTO, ~~TRACK 7~~	
TK7BITS	G425	}	✓
TK8TARG	G426	} DITTO, ~~TRACK 8~~	
TK8BITS	G427	}	✓
TK9TARG	G428	} DITTO, ~~TRACK 9~~	
TK9BITS	G429	}	✓
TK10TRG	G430	} DITTO, ~~TRACK 10~~	
TK10BTS	G431	}	✓
	2		
	3		
	4		
	5		
TK1IDSR	G436	TRACK 1 IDENT CODE	
TK2IDSR	G437	TRACK 2 " "	
TK3IDSR	G438	TRACK 3 " "	
TK4IDSR	G439	TRACK 4 " "	
TK5IDSR	G440	TRACK 5 " "	
TK6IDSR	G441	TRACK 6 " "	
TK7IDSR	G442	TRACK 7 " "	
TK8IDSR	G443	TRACK 8 " "	
TK9IDSR	G444	TRACK 9 " "	
TK10IDS	G445	TRACK 10 " "	
	6		
	7		
TK1REPS	G448	REPAIR/OPTIONS, TRACK 1	
TK2REPS	G449	" " TRACK 2	

* TICK IN THIS COLUMN IF THE USAGE OF BITS IS DEFINED ON A SEPARATE SHEET

ISSUE	A	B	C	D	
DATE					
AUTHOR					

EVENT LOG FOR _____ DATE ____/____/____

TIME	DETAILS OF FAULT OR EVENT	FAULT CODE	OPERATING CONDITIONS	REMEDIAL ACTION TAKEN	TIME

EVENT LOG FOR PRIMER LINE DATE 07/08/88

TIME	DETAILS OF FAULT OR EVENT	FAULT CODE	OPERATING CONDITIONS	REMEDIAL ACTION TAKEN	TIME
12-27	Ultimate limit switch trip	–	In auto	Air in hydraulics – leak	
	on robot track #2			in T. joint. System bled	
				and refilled	12-51
13-49	Wrong detection by photocell #7	–	In auto	Photocell lens cleaned	13-59
14-30	Photocell #7 failed again	–	In auto	Switch to manual	14-32
16-20	GEM80 #5 trip	#719	In manual	Faulty 8213 module slot	
				#10. Module replaced	16-25
17-00	Meal break	–	Halted	Photocell #7 replaced.	
				Restarted in auto	17-30
19-10	Overtemperature trip on	F207	In auto	Fan motor burnt out.	
	conveyor motor #2			Replaced	19-45
20-00	End of shift	–			

GEC Industrial Controls Limited

REF

GEM 80/SERIAL LINK ROUTES
TABLE BLOCK USAGE RECORD

CONTROL PORT

TITLE AUXILIARY 235

LINK No 1

PORT No 2

TRIBUTARIES

Tributary title	Blocks of table locations (control port)	Control port locations	Number of words per message in each direction	Direction	Tributary locations	Blocks of table locations (tributary)	Trib address	Port No.
MASTER 1	OUTPUT	K32-K43	12	→	J0-J11	INPUT	1	1
	INPUT	J32-J34	3	←	K0-K2	OUTPUT		
MASTER 2	OUTPUT	K64-K75	12	→	J0-J11	INPUT	2	1
	INPUT	J64-J66	3	←	K0-K2	OUTPUT		
MASTER 3	OUTPUT	K96-K125	30	→	J0-J29	INPUT	1	3
	INPUT	J96-J125	30	←	K0-K29	OUTPUT		
MASTER 4	OUTPUT	K128-K139	12	→	J0-J11	INPUT	1	4
	INPUT	J128-J130	3	←	K0-K2	OUTPUT		
	OUTPUT			→		INPUT		
	INPUT			←		OUTPUT		
	OUTPUT			→		INPUT		
	INPUT			←		OUTPUT		
	OUTPUT			→		INPUT		
	INPUT			←		OUTPUT		
	OUTPUT			→		INPUT		
	INPUT			←		OUTPUT		

ISSUE	A	B	C	D
DATE				
AUTHOR				

GEC Industrial Controls Limited

REF

ADDRESS RANGE G413 - G417

GEM 80/TABLE LOCATION BIT USAGE RECORD

MNEMONIC	ADDRESS		DESCRIPTION/REMARKS
TK1COL1	G413	.0	ON FOR COLOR SELECT 1, TRACK 1
		.1	
		.2	
		.3	
		.4	
		.5	
TK1DTR1	G413	.6	ON FOR DATA TRANSFER REQUEST TO ROBOT1, TRACK 1
TK1DTR2	G413	.7	" " " " " " ROBOT2, TRACK 1
TK1DTR3	G413	.8	" " " " " " ROBOT3, TRACK 1
TK1DTR4	G413	.9	" " " " " " ROBOT4, TRACK 1
		.10	
		.11	
		.12	
		.13	
		.14	
		.15	
TK2COL1	G415	.0	TRACK 2, AS FOR TRACK 1
		.1	
		.2	
		.3	
		.4	
		.5	
TK2DTR1	G415	.6	
TK2DTR2	G415	.7	
TK2DTR3	G415	.8	
TK2DTR4	G415	.9	
		.10	
		.11	
		.12	
		.13	
		.14	
		.15	
TK3COL1		.0	TRACK 3, AS FOR TRACK 1
		.1	
		.2	
		.3	
		.4	
		.5	
TK3DTR1	G417	.6	
TK3DTR2	G417	.7	
TK3DTR3	G417	.8	
TK3DTR4	G417	.9	
		.10	
		.11	
		.12	
		.13	
		.14	
		.15	

ISSUE DATE AUTHOR	A	B	C	D	

 06.88 (Electronic Master) **ICE 094**

GEC Industrial Controls Limited

REF.

SERIAL LINK ROUTE No.

GEM 80/SERIAL LINK ROUTE
TABLE LOCATION USAGE RECORD

NOTES:-

FROM			TO		
313 A PORT i			163/3 PORT i		
MNEMONIC	OUTPUT ADDRESS	DESCRIPTION/REMARKS	INPUT ADDRESS	MNEMONIC	
MESSNUM	K096	MESSAGE NUMBER	J00	MESSNC	
SSI1P	K097	SIDE SPRAY #11 PRESSURE	J01	SSPR11P	
SSI1C	K098	" " #11 COLOR	J02	SSPR11C	
SSI2P	K099	" " #12 PRESSURE	J03	SSPR12P	
SSI2C	K100	" " #12 COLOR	J04	SSPR12C	
SSI3P	K101	" " #13 PRESSURE	J05	SSPR13P	
SSI3C	K102	" " #13 COLOR	J06	SSPR13C	
SS21P	K103	" " #21 PRESSURE	J07	SSPR21P	
SS21C	K104	" " #21 COLOR	J08	SSPR21C	
SS22P	K105	" " #22 PRESSURE	J09	SSPR22P	
SS22C	K106	" " #22 COLOR	J10	SSPR22C	
SS23P	K107	" " #23 PRESSURE	J11	SSPR23P	
SS23C	K108	" " #23 COLOR	J12	SSPR23C	
SS31P	K109	" " #31 PRESSURE	J13	SSPR31P	
SS31C	K110	" " #31 COLOR	J14	SSPR31C	
SS32P	K111	" " #32 PRESSURE	J15	SSPR32P	
SS32C	K112	" " #32 COLOR	J16	SSPR32C	
SS33P	K113	" " #33 PRESSURE	J17	SSPR33P	
SS33C	K114	" " #33 COLOR	J18	SSPR33C	
RSP	K115	ROOF SPRAY PRESSURE	J19	RSPRP	
RSC	K116	" " COLOR	J20	RSPRCOL	
RSPOSSP	K117	" " POSN SETPOINT	J21	RPOSSP	

ISSUE	A	B	C	D	
DATE					
AUTHOR					

GEC Industrial Controls Limited

REF

GEM 80/SERIAL LINK ROUTES
TABLE BLOCK USAGE RECORD

CONTROL PORT			NUMBER OF WORDS PER MESSAGE IN EACH DIRECTION		TRIBUTARIES			
TITLE								
LINK No								
PORT No								
					TITLE			TRIB ADDRESS
BLOCKS OF TABLE LOCATIONS	OUTPUT			→		INPUT	BLOCKS OF TABLE LOCATIONS	
	INPUT			←		OUTPUT		PORT No.
					TITLE			TRIB ADDRESS
BLOCKS OF TABLE LOCATIONS	OUTPUT			→		INPUT	BLOCKS OF TABLE LOCATIONS	
	INPUT			←		OUTPUT		PORT No.
					TITLE			TRIB ADDRESS
BLOCKS OF TABLE LOCATIONS	OUTPUT			→		INPUT	BLOCKS OF TABLE LOCATIONS	
	INPUT			←		OUTPUT		PORT No.
					TITLE			TRIB ADDRESS
BLOCKS OF TABLE LOCATIONS	OUTPUT			→		INPUT	BLOCKS OF TABLE LOCATIONS	
	INPUT			←		OUTPUT		PORT No
					TITLE			TRIB ADDRESS
BLOCKS OF TABLE LOCATIONS	OUTPUT			→		INPUT	BLOCKS OF TABLE LOCATIONS	
	INPUT			←		OUTPUT		PORT No.
					TITLE			TRIB ADDRESS
BLOCKS OF TABLE LOCATIONS	OUTPUT			→		INPUT	BLOCKS OF TABLE LOCATIONS	
	INPUT			←		OUTPUT		PORT No
					TITLE			TRIB ADDRESS
BLOCKS OF TABLE LOCATIONS	OUTPUT			→		INPUT	BLOCKS OF TABLE LOCATIONS	
	INPUT			←		OUTPUT		PORT No
					TITLE			TRIB ADDRESS
BLOCKS OF TABLE LOCATIONS	OUTPUT			→		INPUT	BLOCKS OF TABLE LOCATIONS	
	INPUT			←		OUTPUT		PORT No.

ISSUE DATE AUTHOR	A	B	C	D

Main Items on any topic have the page numbers *underlined.*

f means "and the following page"; ff means "and the following pages".

THIS PAGE LEFT INTENTIONALLY BLANK

We try to make the information provided in our manuals as accurate and as easy to understand as possible.

If, however, you have found any shortcomings in this manual, then please jot them down on the page overleaf, and mail to:

Dick Wilson
Head of Control Systems Engineering
GEC Industrial Controls Ltd
Kidsgrove
Stoke-on-Trent
ST7 1TW
England

or else pass on to your local GEM 80 agent.

GEM 80 CONTROL LANGUAGE PROGRAMMING REFERENCE MANUAL
Publication T391 Issue 3

Feedback from:
Name .....
Position .....
Company .....
Town or City .....
County or State .....
Postcode/ZIP .....
Country .....

Items that ought to have been included:-

Items inadequately or confusingly explained (please quote page Nos.): -

Items you disagreed with (please quote page Nos.):-

Items you couldn't locate via the Index:-

Any other comments:-

*** Many thanks for your comments above ***

ALSTOM Power Conversion

France
9, rue Ampère
91345 Massy Cedex
Sales Tel: +33 (0) 8 25 02 11 02
Support Tel (International):
+33 (0) 3 84 55 33 33
Support Tel. (National):
08 25 02 11 02

Germany
Culemeyerstraße 1
D-12277 Berlin
Sales Tel: +49 (0) 30 74 96 27 27
Support Tel (International):
+49 (0) 69 66 99 831
Support Tel (National):
01 80 3 23 45 72

UK
Boughton Road, Rugby
Warwickshire, CV21 1BU
Sales Tel: +44 (0) 1788 563625
Support Tel: +44 (0) 1788 547490

USA
610 Epsilon Drive
Pittsburgh, PA 15238
Sales Tel: +1 412 967 0765
Support Tel: +1 800 800 5290

ALSTOM