

TABLE memory and Job Storage

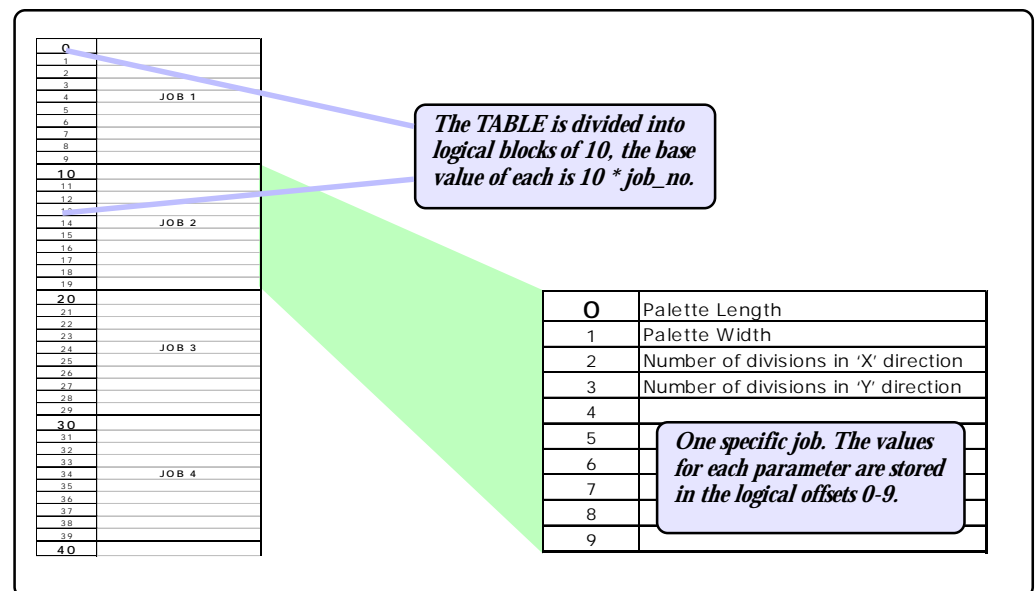
It is often a requirement to allow a users to store and select a machines jobs settings from a menu of stored jobs. On the Motion Coordinator this is quite simple to implement using the controllers **TABLE** memory.

The TABLE comprises of a large single-dimensional array of up to 16000 floating point values. Although typically used to store profiles for software cams, they can be used for anything.

Example

A user program requires the operator to select from a number predefined jobs. Each job contains the following information:

- Palette Length
- Palette Width
- Number of divisions in 'X' direction
- Number of divisions in 'Y' direction
- Up to 6 other values



We have a total of 10 possible values to be used in each job, so we need to divide our table up into blocks of 10, each block will store all the values used in one job, thus:

Referencing the stored values

To reference any particular value in the table, we need to consider two factors, the job_number and the value within the job. If we multiply the job number by the number of values in each block (in this case 10) and then add the offset to the specific value (0-9) then we get an absolute table reference. If our job storage does not reside at the beginning of the table we will also need to add a further offset to the first job in the table, thus

value = TABLE(first_block+(job_number * 10)+value_offset)

To make the code a little more general-purpose, we could assign constants for the first block (**jobs**) and block size (**bsize**) and this is the format used in the following example.

Lets look at a couple of routines which could be used to store and retrieve the values of the first four parameters in our job. We'll call these **plength**, **pwidth**, **xdivs** and **ydivs**. Just for good measure we also have a subroutine called **show** to print the values to a Motion Perfect terminal on channel 5.

```
○ . store: ○  
○ . TABLE(jobs+(bsize*job_no),plength) ○  
○ . TABLE(jobs+(bsize*job_no)+1,pwidth) ○  
○ . TABLE(jobs+(bsize*job_no)+2,xdivs) ○  
○ . TABLE((jobs+(bsize*job_no)+3, ydivs) ○  
○ . return ○  
○ . ○  
○ . fetch: ○  
○ . plength=TABLE(jobs+(bsize*job_no)) ○  
○ . pwidth=TABLE(jobs+(bsize*job_no)+1) ○  
○ . xdivs=TABLE(jobs+(bsize*job_no)+2) ○  
○ . ydivs=TABLE(jobs+(bsize*job_no)+3) ○  
○ . RETURN ○  
○ . ○  
○ . show: ○  
○ . PRINT #5,"Job #";job_no[0] ○  
○ . PRINT #5,"Plength ";plength[0] ○  
○ . PRINT #5,"Pwidth ";pwidth[0] ○  
○ . PRINT #5,"X Divs ";xdivs[0] ○  
○ . PRINT #5,"Y Divs ";ydivs[0] ○  
○ . RETURN ○  
○ . ○
```

fig 2. Simple Job Storage Subroutines

This method can of course be used for blocks of information of any size and with a little thought you can expand the idea to include more complex information.

What if you want to use specific product codes?

Rather than referring to each job by its offset in the table as above, we make one of the values in each block our product-specific code, then we can find the job by scanning through the table until we find a match. Remember, we don't need to read every value, only those holding the job code value.

Take a look at the next code sample, not that we initialise the value of **job_no** to -1 before we scan the table. This gives us a simple check before fetching the information. If the **job_no** returned is -1, no job was found, otherwise we can call our "fetch" routine to get the data.

e.g

```

○ ` job we are looking for is stored in "job_ref" ○
○ find_it: ○
○   job_no=-1 ○
○   FOR a = 0 TO 49 ` 50 jobs in total ○
○     IF TABLE(jobs+(bsize*a)+5)=job_ref THEN job_no=a ○
○   next a ○
○ return ○

```

Using this idea complicates the mechanism for storing jobs slightly as we are no longer directly concerned with the physical table location in which the data is stored.

To store a job we first need to find an empty block. If we have a routine to clear and initialise all the table data to a known value (-1 for example) before use, then to find an empty slot need to test the `job_code` location of each in turn, as soon as we find a value of -1 then we know that is an empty block. To delete a job, we simply need an 'initialise' routine to clear all the value in that block to -1.

Here are a couple more sample routines to initialise the whole table (assuming a maximum of 50 jobs), to find a blank job, and to delete a known job. Note that they all use our master `job_no` variable. The existing routines `store`, `find_it` and `fetch` can be used to load and save jobs.

```

○ init_table: ○
○   FOR job_no=0 TO 49 ○
○     GOSUB delete ○
○   NEXT job_no ○
○ RETURN ○
○ delete: ○
○   tpos = jobs+(bsize*a) ○
○   TABLE(tpos, -1,-1,-1,-1,-1,-1,-1,-1,-1,-1) ○
○ RETURN ○
○ find_blank: ○
○   job_no=-1 ○
○   FOR a = 0 TO 49 ○
○     IF TABLE(jobs+(bsize*a)+5)=-1 THEN job_no=a ○
○   NEXT a ○
○ RETURN ○

```

0	⁰	1 2 3 4
1	¹	
2	²	
3	³	
4	⁴	
5	⁵	
6	⁶	
7	⁷	
8	⁸	
9	⁹	
10	¹⁰	4 4 5 6
11	¹¹	
12	¹²	
13	¹³	
14	¹⁴	
15	¹⁵	
16	¹⁶	
17	¹⁷	
18	¹⁸	
19	¹⁹	
20	²⁰	9 0 8 7 6
21	²¹	
22	²²	
23	²³	
24	²⁴	
25	²⁵	
26	²⁶	
27	²⁷	
28	²⁸	
29	²⁹	
30	³⁰	4 5 4 5 6
31	³¹	
32	³²	
33	³³	
34	³⁴	
35	³⁵	
36	³⁶	
37	³⁷	
38	³⁸	
39	³⁹	

job_code = 90876
GOSUB find_it

(job_no=2)

0	⁰	1 2 3 4
1	¹	
2	²	
3	³	
4	⁴	
5	⁵	
6	⁶	
7	⁷	
8	⁸	
9	⁹	
10	¹⁰	9 0 8 7 6
11	¹¹	
12	¹²	
13	¹³	
14	¹⁴	
15	¹⁵	
16	¹⁶	
17	¹⁷	
18	¹⁸	
19	¹⁹	
20	²⁰	- 1
21	²¹	- 1
22	²²	- 1
23	²³	- 1
24	²⁴	- 1
25	²⁵	- 1
26	²⁶	- 1
27	²⁷	- 1
28	²⁸	- 1
29	²⁹	- 1
30	³⁰	4 5 4 5 6
31	³¹	
32	³²	
33	³³	
34	³⁴	
35	³⁵	
36	³⁶	
37	³⁷	
38	³⁸	
39	³⁹	

GOSUB delete

(job_no=2)

0	⁰	1 2 3 4
1	¹	
2	²	
3	³	
4	⁴	
5	⁵	
6	⁶	
7	⁷	
8	⁸	
9	⁹	
10	¹⁰	9 0 8 7 6
11	¹¹	
12	¹²	
13	¹³	
14	¹⁴	
15	¹⁵	
16	¹⁶	
17	¹⁷	
18	¹⁸	
19	¹⁹	
20	²⁰	1 2 3 4 5
21	²¹	
22	²²	
23	²³	
24	²⁴	
25	²⁵	
26	²⁶	
27	²⁷	
28	²⁸	
29	²⁹	
30	³⁰	4 5 4 5 6
31	³¹	
32	³²	
33	³³	
34	³⁴	
35	³⁵	
36	³⁶	
37	³⁷	
38	³⁸	
39	³⁹	

GOSUB find_blank
GOSUB store

(job_no=2)