

External Profile Mode

The operation of external profile mode and the setup procedure is described in this chapter. Code for the generation of profile data and the use of external profile mode is provided with C-Motion, relieving the developer from implementing any of the details and algorithms mentioned in this chapter.

When an axis has its profile mode set to trapezoidal, velocity contouring or s-curve, the motion processor is executing a trajectory generation algorithm. The algorithm generates a motion profile based on a set of constraints programmed by the host. The constraints include: target position (**SetPosition**), maximum velocity (**SetVelocity**), maximum acceleration (**SetAcceleration**), maximum deceleration (**SetDeceleration**) and maximum jerk (**SetJerk**)¹. Based on these constraints, the profile algorithm generates a new commanded position, velocity and acceleration once per chip cycle. These three values are then used by the servo filter or step generator to move and control the motor. This is shown in the figure below.

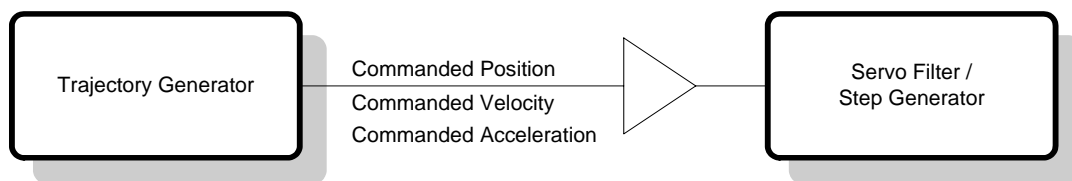


Figure 1 – Internal trajectory generation

In contrast, when an axis is placed in external profile mode, the host is responsible for the calculation of the commanded position, velocity and acceleration. In effect, the host replaces the motion processor trajectory generator. It therefore becomes the responsibility of the host profile algorithm to make sure that the generated profile does not exceed any motion constraints.

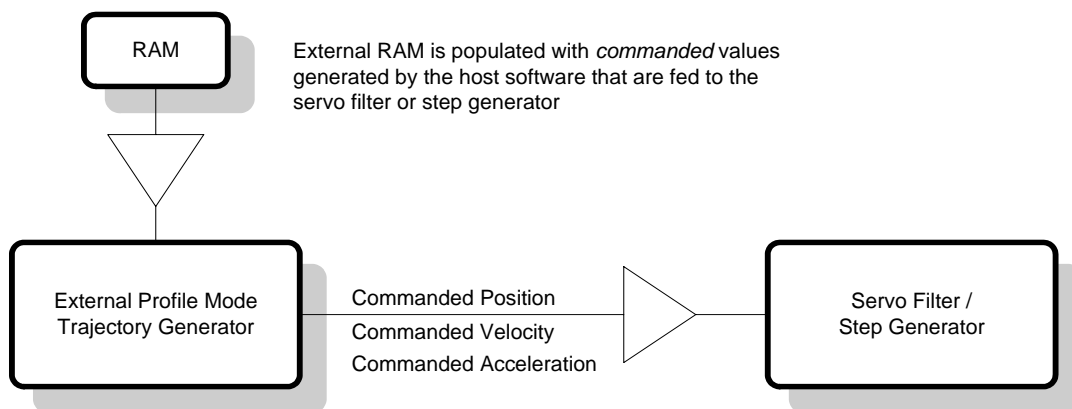


Figure 2 - External trajectory generation

The basic operation of external profile mode is to read *commanded* values that are stored in external RAM and feed these to the servo filter or step generator. In addition, with the use of the optional time buffer (explained below) external profile mode can be used to generate intermediate *commanded* values using an iterative calculation.

¹ Note that not all of the constraints are used by each profile mode

A complete motion profile is made up of multiple trajectory segments, with each segment providing the parameters for a period of motion. A trajectory segment is made up of five trajectory “variables” that correspond to an internal trajectory variable. The variables and their format are shown in the table below.

Variable	Internal trajectory variable	Format	Range
<i>Position</i>	commanded position	32.0*	-2,147,483,648 to 2,147,483,647 counts
<i>Velocity</i>	commanded velocity	16.16	-32,768 to 32,767 + 65,535/65,536 counts/cycle
<i>Acceleration</i>	commanded acceleration	16.16	-32,768 to 32,767 + 65,535/65,536 counts/cycle ²
<i>Jerk</i>	commanded jerk	0.32	-2,147,483,648 to 2,147,483,647/4,294,967,296 counts/cycle ³
<i>Time</i>	segment time	32.0	0 to 2,147,483,647 cycles

*The fixed-point encoding of trajectory parameters is discussed in section 1.7.

The motion processor contains instructions for setting up buffers (circular arrays) in external memory and assigning them to trajectory variables. Once setup these buffers can be used to store the trajectory data used by external profile mode. The required commands and their use are described in section 1.1. A buffer for the position trajectory variable is always required, while a buffer for the velocity, acceleration, jerk and time trajectory variables is optional. The variables that are specified and their values determine the shape of the profile that is executed by the motion processor.

An axis must have one buffer created and assigned for each trajectory variable that will be used for trajectory generation. Each variable entry within the buffer occupies 32 bits. Once instructed to do so, the motion processor gathers a set of trajectory data by reading corresponding position, velocity, acceleration, jerk and time entries from the buffers. Corresponding entries are located at the same offset within each buffer. The figure below shows a typical buffer configuration where all trajectory variables are being used. Note that each buffer must have the same length.

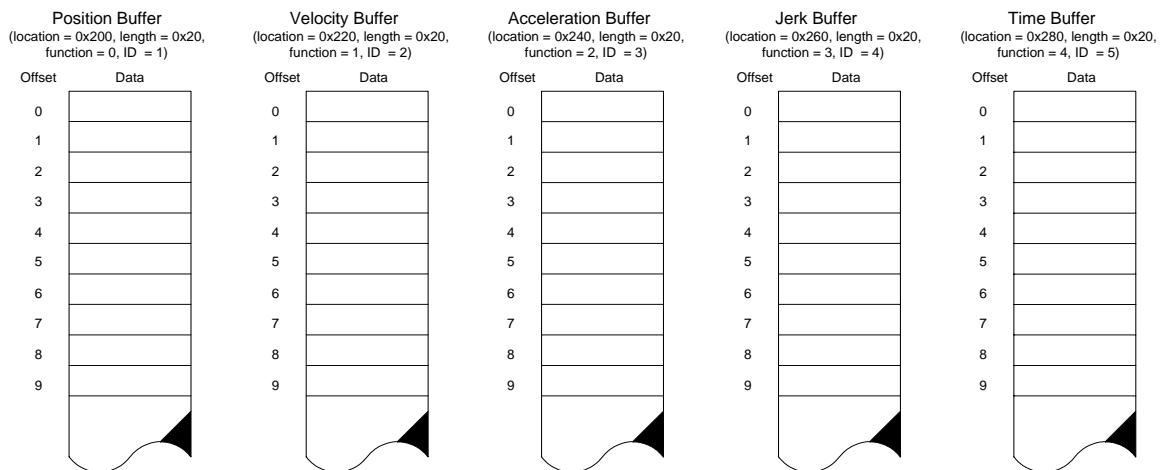


Figure 3 – Example of an external memory buffer configuration

The variable buffers can be considered a table where each row corresponds to a set of values that are used for a segment of motion. When the external profile is first started, the data at offset 0 of the position, velocity, acceleration, jerk and time buffers is read. In the case where any of the velocity,

acceleration or jerk buffers are not defined the corresponding internal variable is assigned a value of zero. If a time buffer is not defined, the segment time is assigned a value of one.

The pseudo code below shows the high level operation of external profile mode and is included here to help explain how entries in the time buffer are utilized. This sequence is executed once every chip cycle for each axis executing an external profile, and stops when either the segment_time read from the time buffer is zero, a limit switch/motion error occurs or when a **SetStopMode AbruptStop** is executed.

```

if (segment_time == 0)
    // get the next set of position,velocity,acceleration,jerk
    // and time values
    CommandedPosition = GetNextPositionBufferEntry()
    CommandedVelocity = GetNextVelocityBufferEntry()
    CommandedAcceleration = GetNextAccelerationBufferEntry()
    CommandedJerk = GetNextJerkBufferEntry()
    segment_time = GetNextTimeBufferEntry()
    if (segment_time == 0)
        StopTrajectory()
else
    CommandedPosition += CommandedVelocity+
        CommandedAcceleration/2+CommandedJerk/6
    CommandedVelocity += CommandedAcceleration+CommandedJerk/2
    CommandedAcceleration += CommandedJerk

segment_time = segment_time-1

```

The sequence always starts with the motion processor reading a set of values from external memory and assigning their values to the internal trajectory variables. These values are used for one chip cycle. If the time buffer contains a value that is greater than one, at the next cycle the chip calculates a new set of trajectory parameters and decrements the segment time. The commanded position, velocity and acceleration are updated (as shown above) according to the following equations:

$$P_n = \left\{ p_{n-1} + v_{n-1} + \frac{a_{n-1}}{2} + \frac{j_{n-1}}{6} \mid n = 2 \dots t \right\}$$

$$V_n = \left\{ v_{n-1} + a_{n-1} + \frac{j_{n-1}}{2} \mid n = 2 \dots t \right\}$$

$$A_n = \left\{ a_{n-1} + j_{n-1} \mid n = 2 \dots t \right\}$$

where:

t is the total segment time.

P, p is the commanded position

V, v is the commanded velocity

A, a is the commanded acceleration

j is the commanded jerk

The following simple example demonstrates the result of the calculations performed when the time buffer contains a non-zero value.

	Actual Chip Time	Commanded Position	Commanded Velocity	Segment Time	Resultant Motion
Initial set of values read from external memory	11003	0	10000h	10	constant velocity
Chip performs calculations to generate intermediate values until the segment time equals 0	11004	1	10000h	9	
	11005	2	10000h	8	
	11006	3	10000h	7	
	11007	4	10000h	6	
	11008	5	10000h	5	
	11009	6	10000h	4	
	11010	7	10000h	3	
	11011	8	10000h	2	
	11012	9	10000h	1	
Next set of values read from external memory	11013	10	0	15	stationary

The value 10000h (hex) represents a value of 1.0 in the PMD 16.16 fixed-point format. The motion processor has calculated the values shown in gray according to the equations shown above.

For the case where the time buffer contains a value of one (or if a time buffer is not created) the motion processor will only use the current set of variables for one cycle. It will not perform any calculations and will read a new set of variables during the next cycle. In this way, data generated by the host has complete control over the generated profile because it provides new commanded values for every cycle of the chip. This requires a high bandwidth for populating external memory and therefore is only recommended in designs that utilize dual port RAM.

In external profile mode all buffers are read in a circular fashion such that when the final set of data is read from the buffer (defined by the buffer length), the read index pointer wraps to offset zero.

1.1 Setting up the trajectory data buffers

The variable buffers are setup using the chip commands **SetBufferStart**, **SetBufferLength**, and **SetBufferFunction**. For a complete description of these commands refer to the *Navigator or Pilot Programmer's Reference*.

One buffer must be created for each trajectory variable and all buffers must be of the same length. Generally, it makes sense to select a buffer size that maximizes the use of available memory. For example, if 32Kx16 words of memory are available and 2 axes will be executing an external profile, each variable buffer should be 1536 double words in length.

32Kx16 = 16384 double words (double word = 32 bits)

16384-512 (start of memory reserved by motion processor) = 15872 double words available

$15360/(2*5) = 1536$ double words per axis, per variable

If the chip trace function is used, the total memory available for external profile mode must be reduced by the size of the trace buffer.

The code below shows a sequence for creating the variable buffers for the example given above.

```
enum { PMDBufferFunctionNone=-1, PMDBufferFunctionPosition=0,
       PMDBufferFunctionVelocity, PMDBufferFunctionAcceleration,
       PMDBufferFunctionJerk, PMDBufferFunctionTime };

num_axes = 2;
final_variable = PMDBufferFunctionTime;
memory_start = 0x200;      // first 1Kx16 reserved by motion processor
memory_size = 0x4000;     // 16Kx32
buffer_id = 2;            // ID 0 is reserved for the trace buffer
                          // ID 1 is also used as a read index for
                          // the trace buffer

buffer_start = memory_start;
buffer_length = (memory_size - memory_start)/(num_axes*num_variables);
for (int axis = 0; axis < num_axes; axis++)
    for (int variable = 0; variable <= final_variable; variable++)
    {
        buffer_start = memory_start+(axis*variable*buffer_length);
        PMDSetBufferStart(&hAxis[axis],buffer_id,buffer_start);
        PMDSetBufferLength(&hAxis[axis],buffer_id,buffer_length);
        // assign a trajectory variable to the buffer
        PMDSetBufferFunction(&hAxis[axis],variable,buffer_id);
        buffer_id++;
    }
```

1.2 The format of the trajectory data

As described in a previous section, the trajectory data is a table of values that correspond to trajectory variables stored within the motion processor. Below is shown a recommended structure for storing the data prior to uploading it to the motion processor.

```
typedef long PMDFIXED32;      // -2,147,483,648 to 2,147,483,647
typedef long PMDFIXED1616;   // -16384 to 16383 + 65535/65536
typedef long PMDFIXED032;    // -2,147,483,648 to 2,147,483,647/4,294,967,296

typedef unsigned long PMD_UINT32;    // 0 to 4,294,967,296
typedef unsigned short PMD_UINT16;   // 0 to 32,768
typedef PMDFIXED32 PMD_POSITION;    // count
typedef PMDFIXED1616 PMD_VELOCITY;  // count/cycle
typedef PMDFIXED1616 PMD_ACCELERATION; // count/(cycle*cycle)
typedef PMDFIXED032 PMD_JERK;       // count/(cycle*cycle*cycle)
typedef PMD_UINT32 PMD_TIME;        // cycle
```


The above mechanisms work for uploading data prior to the start of motion. During motion, the *buffer read index* is used to determine when it is safe to overwrite data already in the buffers. An example of this is shown below.

```
PMDUINT32 readindex, writeindex;
// read the indexes for the position buffer (buffer #2) of axis#1
// the read/write index will be the same for all variables
// since they are read as a set
PMDGetBufferReadIndex(&hAxis[PMDAxis1],2,&readindex);
PMDGetBufferWriteIndex(&hAxis[PMDAxis1],2,&writeindex);
if (readindex > writeindex)
    free_buffer_entries = readindex - writeindex;
else
    free_buffer_entries =
        (memory_size - memory_start) - (writeindex - readindex);

for (int variable = 0; variable < num_variables; variable++)
    for (int index = 0; index < free_buffer_entries; index++)
    {
        // the buffer_id starts at 2
        buffer_id = 2+variable;
        PMDWriteBuffer(&hAxis[PMDAxis1],buffer_id,
            data[PMDAxis1][variable][index]);
    }
```

1.4 Starting the profile

Before starting the profile the trajectory buffers must be created and populated and all axes should be stationary. It is crucial that the current commanded position controlling the motion processor corresponds to the first entry within the tables stored within external memory. This ensures a smooth start of motion and transition to the external profile mode. A trapezoidal or s-curve move can be made prior to external profile mode operation to guarantee this is the case or the first entry in the position buffers can be created according to the current commanded position of the active axes. An example of the initial move method is shown in the following code.

```
// move axis#1 to its starting location
PMDSetProfileMode(&hAxis[PMDAxis1],PMDTrapezoidal);
PMDSetVelocity(&hAxis[PMDAxis1],max_vel);
PMDSetAcceleration(&hAxis[PMDAxis1],max_acc);
PMDSetPosition(&hAxis[PMDAxis1],data[PMDAxis1][PMDBufferFunctionPosition][0]);
PMDUpdate(&hAxis[PMDAxis1]);
// move axis#2 to its starting location
PMDSetProfileMode(&hAxis[PMDAxis2],PMDTrapezoidal);
PMDSetVelocity(&hAxis[PMDAxis2],max_vel);
PMDSetAcceleration(&hAxis[PMDAxis2],max_acc);
PMDSetPosition(&hAxis[PMDAxis2],data[PMDAxis2][PMDBufferFunctionPosition][0]);
PMDUpdate(&hAxis[PMDAxis2]);
```

Like all profile modes, external profile mode is started by first selecting it as the desired profile mode and then issuing the **Update** command.

```
PMDSetProfileMode(&hAxis[PMDAxis1], PMDExternalProfile);
PMDUpdate(&hAxis[PMDAxis1]);
```

When more than one axis is being controlled with external profile mode and a synchronized start is required, a **MultiUpdate** command should be used to start motion.

```
PMDSetProfileMode(&hAxis[PMDAxis1], PMDExternalProfile);
PMDSetProfileMode(&hAxis[PMDAxis2], PMDExternalProfile);
PMDSetProfileMode(&hAxis[PMDAxis3], PMDExternalProfile);
// the mask selects which axis is updated
PMDMultiUpdate(&hAxis[PMDAxis1], PMDAxis1Mask+PMDAxis2Mask+PMDAxis3Mask);
```

It is not recommended that an axis be switched into or out of External profile mode while the axis is in motion.

If the specified values for this profile mode are not set correctly, the axis may move suddenly in one direction or another. It is the host's responsibility to provide position, velocity, acceleration and jerk values that result in safe motion within acceptable position limits.

1.5 Stopping the profile

External profile mode stops automatically when it reads a value of zero from the time buffer. It also stops when a limit event or motion error event occurs. In all of these cases, when the profile stops the axis is held in a stationary position.

The profile can also be stopped using the chip **SetStopMode** command. When using this command, the only stop mode that has any effect in external profile mode is **AbruptStop**. If a **SmoothStop** is issued no change in motion will occur. The following code demonstrates the use of **SetStopMode** to halt axis motion. It must be issued for every active axis.

```
PMDSetStopMode(&hAxis[PMDAxis1], PMDAbruptStop);
PMDUpdate(&hAxis[PMDAxis1]);
```

When more than one axis is being controlled with external profile mode and a synchronized stop is required, a **MultiUpdate** command should be used.

```
PMDSetStopMode(&hAxis[PMDAxis1], PMDAbruptStop);
PMDSetStopMode(&hAxis[PMDAxis2], PMDAbruptStop);
PMDSetStopMode(&hAxis[PMDAxis3], PMDAbruptStop);
// the mask selects which axis is updated
PMDMultiUpdate(&hAxis[PMDAxis1], PMDAxis1Mask+PMDAxis2Mask+PMDAxis3Mask);
```

1.6 Detecting the completion of motion

In the standard (non-external) profile modes, the motion complete bit contained within the event status register is used to detect the completion of motion. This bit is set when the axis comes to rest, which can occur many times during a complicated multi-axis move. As such, it cannot be used for determining the completion of motion when external profile mode is used. However, there is an alternative method of detection.

When external profile mode starts, it sets the *profile segment number* to 1 in the activity status register. When the trajectory is stopped, or when a trajectory segment is read from external RAM with its time buffer value set to zero, the external profile mode is halted and the *profile segment number* is set to 0. Therefore, the profile segment number contained with the activity status register can be used to determine when motion has completed. More information on the activity status register can be found in the *Navigator or Pilot Programmer's Reference*.

This bit can be polled or the breakpoint mechanism can be used to automatically cause some action at the completion of the motion. In its simplest form, a breakpoint based on the trajectory segment number can be used to generate an interrupt. This is shown below.

```
// start motion
PMDSetProfileMode(&hAxis[PMDAxis1],PMDExternalProfile);
PMDUpdate(PMDAxis1);
// program a breakpoint to generate an interrupt upon
// completion of the profile
PMDSetInterruptMask(&hAxis[PMDAxis1],PMDEventBreakpoint1Mask);
// when bit 13 of the activity status register goes low,
// the profile has finished
PMDSetBreakpointValue(&hAxis[PMDAxis1],PMDBreakpoint1, (0x2000<<16) | 0x0000 );
PMDSetBreakpoint(&hAxis[PMDAxis1],PMDBreakpoint1,axis,PMDBreakpointNoAction,
                 PMDBreakpointActivityStatus );
```

1.7 Fixed-point encoding of trajectory parameters

PMD motion processors send and receive trajectory parameters using a fixed-point representation. In other words, a fixed number of bits are used to represent the integer portion of a real number, and a fixed number of bits are used to represent the fractional component of a real number. The chip uses three formats.

Format	Word size	Range	Description
32.0	32 bits	- 2,147,483,648 to +2,147,483,647	Unity scaling. This format uses an integer only representation of the number. The desired value is sent to the chip with no scaling.
16.16	32 bits	-32,768 to 32,767 + 65,535/65,536	Uses $1/2^{16}$ scaling. The chipset expects a 32 bit number which has been scaled by a factor of 65,536. For example to specify a velocity of 2.75, 2.75 is multiplied by 65,536 and the result is sent to the chip as a 32 bit integer (180,224 decimal or 2c000 hex.).
0.32	32 bits	- 2,147,483,648/4,294,967,296 to +2,147,483,647/4,294,967,296	Uses $1/2^{32}$ scaling. The chip expects a 32 bit number which has been scaled by a factor of 4,294,967,296 (2^{32}). For example to specify a value of .0075, .0075 is multiplied by 4,294,967,296 and the result is sent to the chip as a 32 bit integer (32,212,256 decimal or 1eb8520 hex).