FAST MOVING TECHNOLOGY

VAL3 motion addon

Click here for motion addon download.

(s6.6.1) **Carried part move instructions**

These instructions can be activated with the valTraj runtime license in demo mode. They may be protected in the future: only the beta version is free !

num **$movelp**(tool tTarget, point pFixedTool, mdesc mSpeed)
num **$movecp**(tool tIntermediate, tool tTarget, point pFixed, mdesc mSpeed)
num **$movejp**(joint,tool,mdesc) (almost no difference with usual joint move)
num **$movejp**(tool tTarget, point pFixedTool, mdesc mSpeed)
(s6.7)
num **$alterMovelp**(tool tTarget, point pFixedTool, mdesc mSpeed)
num **$alterMovecp**(tool tIntermediate, tool tTarget, point pFixed, mdesc mSpeed)
num **$alterMovejp**(joint,tool,mdesc)
num **$alterMovejp**(tool tTarget, point pFixedTool, mdesc mSpeed)

These instructions program a move in carried part mode: see What is the difference between a carried part and a normal move ?.

A resetMotion() is needed to switch from carried part mode to standard mode (and back). Example:

```
...
resetMotion()
$movejp(...) // Start moving part mode
$movelp(...)
movel(...) // Runtime error
...
resetMotion()
movej(...) // Start moving tool mode
movel(...)
$movejp(...) // Runtime error
...
```

(s6.4.1) num **$rotAngle**(trsf tTransformation)
This instruction returns the angle (deg) of the rotation part of the trsf.

(s6.4.1) num **$getRot**(trsf tTransformation, num& nXaxis, num& nYaxis, num& nZaxis)
This instruction returns the angle (deg) of the rotation part of the trsf, and update (nXaxis, nYaxis, nZaxis) with the coordinates of the unitary vector of the rotation axis: nXaxis^2+nYaxis^2+nZaxis^2 = 1.

(s6.4.1) trsf **$setRot**(num nAngle, num& nXaxis, num& nYaxis, num& nZaxis)
This instruction returns a trsf where the orientation is defined with an axis (nXaxis, nYaxis, nZaxis) and an angle (deg). The coordinates of the axis are not necessarily unitary, but nXaxis^2+nYaxis^2+nZaxis^2 should be greater than 0...

(s6.4.1) void **$setCartJogAccel**(num nTranslationAccel, num nTranslationDecel, num nRotationAccel, num nRotationDecel)
This instruction modifies the Cartesian acceleration for the Cartesian jog moves (modes Frame and Tool) and for the velocity instructions ($felFrame, $velTool).
Translation accel / decel are in mm/s^2, rotation accel are in deg/s^2. The default value is typically between 1000 and 2000m/s^2 and 300deg/s^2 for a TX60.
The acceleration is only taken into account when the velocity move is started (with $velFrame, or $velTool).

(s6.4.1) bool **$setLength**(num nLengths[]) (to be replaced with $getDH)

This instruction may be protected with a runtime license in the future. Only the beta version is free !

This instruction modifies the lengths that define the geometry of the arm. The change is immediate, and is also applied to arm.cfx to recover it after reboot. It returns true if the change was applied, false if some lengths offsets are larger than 10mm.

If the change is applied during the preprocessing of a VAL3 movel/movec instruction, an internal motion error may occur (path discontinuity). The change is safe when the motion stack is empty (after waitEndMove() or resetMotion()).
$setLength should also not be used with pending alter or mobile frame instruction.
For 6-axes arms, nLenghts must have 6 parameters, in the order: lengths axis 2 to 3, length axis 4 to 5, length axis 6 to flange, X offset axis 1 to 2, Y offset axis 1 to 2, X offset axis 3 to 4
For RS arms, nLenghts must have 2 parameters, in the order: lengths axis 1 to 2, length axis 2 to 3

(s6.4.1) bool **$setDH**(trsf trDHs[]) (to be suppressed)

(s6.7) This instruction is also activated with the 6axisAbsoluteRobot runtime license; in the future, this license will be REQUIRED to enable this instruction.

This instruction modifies the geometry of the arm (both lengths and orientations). The change is immediate, and is also applied to arm.cfx to recover it after reboot. It returns true if the change was applied, false if some lengths offsets are larger than 10mm or orientation offsets larger than 5 deg.
If the change is applied during the preprocessing of a VAL3 movel/movec instruction, an internal motion error may occur (path discontinuity). The change is safe when the motion stack is empty (after waitEndMove() or resetMotion()).
$setDH should also not be used with pending alter or mobile frame instruction.
The number of specified trsf must match the number of robot axes (5 or 6 today - RS arms are not supported).
The standard DH parameter are, in order: Rz - z - x - Rx - (Ry), for each axis.
- The Rz DH parameter for axis 1to 2 cannot be specified (not needed also).
- The first tsrf specified must contain the z, x, Rx, Ry DH parameters for axis 1 to 2, and the Rz parameter for axis 2 to 3, and so on for next axes (this way of doing is mathematically strictly equivalent to standard DH computation, and allows the use of the VAL3 trsf structure).
- the last trsf applies to the flange (it can be null for a 6 axes robot, but is required for a 5-axes robot to get a correct flange orientation)

For instance, for a tx90, the array of 6 trsf matching the default parameters is:
dh[0]={ 50, 0, 0, -90, 0, **-90**}
dh[1]={ 425, 0, 0, 0, 0, **90**}
dh[2]={ 0, 0, 50, 90, 0, 0}
dh[3]={ 0, 0, 425, -90, 0, 0}
dh[4]={ 0, 0, 0, 90, 0, 0}
dh[5]={ 0, 0, 100, 0, 0, 0}

(s6.6.3) void **$getDH**(num& theta[], num& d[], num& a[], num& alpha[], num& beta[])
(s6.7) void **$getDefaultDH**(num& theta[], num& d[], num& a[], num& alpha[], num& beta[])
This instruction returns in the specified arrays the DH parameters of the arm (d and a in mm, theta, alpha and beta in deg).
The number of entries in the arrays must match the number of robot axes (5 or 6 today - RS arms are not supported). An additionnal entry in the d array may be required to get the flange dimension: when it is missing, a runtime error is generated and a diagnostic message is sent to the logger.

The DH parameters are defined so that the joint position {j1, j2, j3, j4, j5, j6} matches the Cartesian position pCart at flange center point with :
pCart.trsf = {0,0,0,0,0, j1+theta[0]}
* {a[0], b[0], d[0], alpha[0], beta[0], j2+theta[1]} (b[ ] is an optional parameter, not used by getDH, but that can be specified with setDH)
* {a[1], b[1], d[1], alpha[1], beta[1], j3+theta[2]}
* {a[2], b[2], d[2], alpha[2], beta[2], j4+theta[3]}
* {a[3], b[3], d[3], alpha[3], beta[3], j5+theta[4]}
* {a[4], b[4], d[4], alpha[4], beta[4], j6+theta[5]}
* {a[5], b[5], d[5], alpha[5], beta[5], 0}
* {0, 0, d[6], 0, 0, 0} (d[6] is not needed for most arms)

(s6.6.3-s6.6.5) bool **$setDH**(num theta[], num d[], num a[], num alpha[], num beta[])
(s6.7+) bool **$setDH**(num theta[], num d[], num a[], num b[], num alpha[], num beta[])

(s6.7) This instruction is also activated with the 6axisAbsoluteRobot runtime license; in the future, this license will be REQUIRED to enable this instruction.

This instruction modifies the geometry of the arm (both lengths and orientations) : set getDH() for the description of the DH parameters. The change is immediate, and is also applied to arm.cfx to recover it after reboot. It returns true if the change was applied, false if some lengths offsets are larger than 10mm or orientation offsets larger than 5 deg.
If the change is applied during the preprocessing of a VAL3 movel/movec instruction, an internal motion error may occur (path discontinuity). The change is safe when the motion stack is empty (after waitEndMove() or resetMotion()).
$setDH should also not be used with pending alter or mobile frame instruction.
The number of DH arrays must match the number of robot axes. (s6.6.3b: for RS arms, only changes in a are supported today). An additionnal entry in the d array may be required to modify the flange dimension: when it is missing, $setDH returns false and a diagnostic message is sent to the logger.

bool **$setOffset**(joint jPosition)
This instruction modifies the zero position of each axis so that the specified joint coordinates match the current mechanical arm position. The change is immediate, and is also applied to arm.cfx to recover it after reboot. This instruction disables arm power, if needed, before the new zero offsets are applied.

void **$velFrame**(frame fReference, tool tTool, mdesc mDesc):
void **$velJoint**(tool tTool, mdesc mDesc):
void **$velTool**(tool, mdesc):
void **$setVelCmd**(num cmd[6]):
See velocity addon documentation.

num **$drvSelect**(string sSerialName, num nAxis)
num **$drvSendCommand**(string sSerialName, string sCommand)
num **$drvGetAnswer**(string sSerialName, string sAnswer)
These instructions (CS8, CS8HP only) can be used to communicate with the serial interface of the Servotronix drives. They can be used for instance to have first 5 axes commanded by the CS8 controller and the 6th axis commanded in VAL3 with drive velocity commands.
The $drvSelect() instruction selects the drive to dialog with; the $drvSendCommand() instruction sends a command to the selected axis; the $drvGetAnswer() instruction returns the answer of the previous command.
These instructrions return 0 if successfull, -1 if the serial line could not be configured, -2 or -3 for communication problem, -4 if the serial line name sSerialName is not correct.
The serial name can be either an external cabling between the CPU and the drive holders; or an internal serial line can be used by specifying the name "serialDaps".

void **$alterAtc**(num nAtc[])
This instruction specifies an additionnal force feedforward to be applied to each axis, in N or N.m. This additionnal feedforward is applied until another feedforward is specified. The feedforward is applied with the next drive command (every 4ms).
void **$getFeedForward**(num& nAtc[])
This instruction returns the force feedforward applied to each axis, in N or N.m. The feedforward is updated with each drive command (every 4ms).

void **$initFirFilter**(num nDeltaT, num nPeriod, num& nPositionCoefs)
void **$initFirFilter2**(num nDeltaT, num nPeriod, num& nPositionCoefs[], num& nVelocityCoefs[])
num **$firFilter**(num& nCoefs[], num& nInputs[])
These instructions implements a FIR ('F'inite 'I'mpulse 'R'esponse) filter for a numerical input, typically to reduce noise and compensate delay on an encoder input.
The $initFirFilter instructions computes the filter parameters, for a position or position and velocity filter. The dimension of the filter is given by the size of the 'coefs' data. nDeltaT is the time advance for the filtered value (ms) ; nPeriod is the period of the data to filter.
The $firFilter() instruction returns the filtered value from the n previous unfiltered values nInputs. The last unfiltered value must be placed in nInputs[0] before calling $firFilter. $firFilter will then shift data in the nInputs array so that nInputs[x] becomes nInputs[x+1], leaving nInputs[0] ready to be used for a new filtering.
See tracking documentation for an example of use.

num **$getBoxcarDelay**()
This instruction returns the delay (ms) induced by the internal filtering of alter or $updateFrame commands.

point **$getPosFbk**(tool tTool, f fReference)
This instruction is similar to the standard here() instruction, but uses as input the joint position measured by the encoders instead of the joint position command sent to the drives.

joint **$getJntFbk**()
This instruction is similar to the standard herej() instruction, but returns the joint position measured by the encoders instead of the joint position command sent to the drives.

joint **$getJntPosRef**() (s5.6)
This instruction is similar to the standard herej() instruction, but returns the joint position before internal boxcar filter instead of the joint position command sent to the drives after the boxcar filter is applied.

void **$getJntSpeedCmd**(num& nSpeed[])
This instruction returns in nSpeed the joint velocity command sent to the drives after the boxcar filter is applied.

void **$getJntSpeedRef**(num& nSpeed[])
This instruction returns in nSpeed the joint velocity before internal boxcar filtering. When arm power is disabled (brake release), velocity is always null.

void **$getJntSpeedFbk**(num& nSpeed[])
This instruction returns in nSpeed the joint velocity measured by the encoders.

void **$getJntForce**(num& nForce[]) (use getJointForce() in VAL3 s6.4+)
This instruction return in an array of num the current joint force (torque in N.m) applied by the motors.

(s6.7) void **$getDrvForce**(num& nForce[])
This instruction return in an array of num the current motor torque (in N.m).

(s6.7) bool **$setInertia**(num nAxis, num nMass, trsf tGravityCentre, num& nInertia[])
bool **$addInertia**(num nAxis, num nMass, trsf tGravityCentre, num& nInertia[])
This instruction modifies the inertia parameters of the dynamic model for one axis. The impact on arm behaviour depends on the arm tuning:
- the dynamic model is not configured for most of the CS8 arms ($addInertia will then return false)
- only the gravity model is implemented for 6-axes CS8C arms today (to prevent arm drop when enabling power), but use of complete inertia should be enabled with s6.5 to allow further arm behaviour improvements.

The nAxis parameter is between 1 and (nbAxis+1): this is the axis where the mass is attached, (nbAxis+1) standing for the tool payload attached to the arm flange.
The nMass parameter specifies the load (kg) to be added to the spoecified axis (or flange). A negative value can be used to removed a mass previously added (or to reduce the payload which is the nominal payload by default).
The tGravityCentre parameter gives the x,y,z coordinates of the gravity centre of the mass relatively to the axis's base (or to flange when specified axis is nbAxis+1). rx, ry, rz are ignored.
The nInertia parameter is optional:
- the parameter has no effect if it is a numerical constant such as '0'
- If the parameter is an array of 3 elements, they are used as the main inertia parameters Ixx, Iyy and Izz of the additional payload.
- If the parameter is an array of 9 elements, they are used as the complete inertia parameters Ixx, Ixy, Ixz, Iyx, Iyy, Iyz, Izx, Izy and Izz of the additional payload.

bool **$setMaxJntVel**(num& nMaxJointVel[])
This instruction modifies the maximum joint velocity for both automatic and manual modes. The maximum velocity is then verified by different redundant safety mechanisms, in the DSI board (every 0.2ms), in the drives (every 0.2ms), and in the CPU (every 4ms), assuring a very high safety level when the maximum joint speed is low.
This instruction should not be called in loop, it could result in communication failure on the drive bus. Before s6.4, the VAL3 motion may send commands with joint velocity higher than the specified safety maximum, resulting in a sudden safety stop. With s6.4+, the VAL3 motion takes the maximum joint speed into account in the command.

num **$setMaxTvel**(num nMaxTvel)
This instruction modifies the maximum Cartesian velocity for both automatic and manual modes and returns the effective maximum Cartesian velocity after call. The maximum Cartesian velocity in manual mode is limited to 250 mmps.

num **$getMoveId**(void)
void **$setMoveId**(num id)
See move id documentation.

bool **$setFriction**(num nAxis, num nFriction, num nV0, num nV1)
See how to optimize friction compensation documentation.

void **$stopMove**(num nStopTime)
This instruction is similar to the standard stopMove() instruction, but specifies a stop time (in second) instead of using the mdesc deceleration parameter to stop. If the stop time is too short, it may result in an enveloppe error and the arm may then leave its nominal trajectory.

void **$setBoxcarFreq**(num nJointFreq)
void **$setBoxcarFreq**(num nCartFreq, num nJointFreq)
These instructions modify the dimension of the internal filtering of move commands: the higher the frequency (in Hz), the less filtering. The nJointFreq specifies a filtering in the joint space; the nCartFreq specifies a filtering along the path (not used by default).

bool **$coggingParams**(num nAxis, num nNbHarms, num nParams[])
bool **$coggingEnable**(num nAxis, num nNbHarms)
These instructions can be used to optimize motor cogging compensation.

void **$getGravity**(joint jPosition, num& nVelocity[], num& nAcceleration[], num& nForce[])
This instruction computes the theoretical motor forces (using gravity model or Newton model, today gravity only) depending of position, velocity, and acceleration (position only for gravity)

void **$externalForce**(joint jPosition, tool tTool, num& nJointOverForce[], num& nCartForce[])
This instruction computes the theoretical external efforts at the specified tool center point corresponding to measures joint forces, as described in the following example:

pos=$getJntFbk() read actual position (feedback)
$getJntForce(current) read actual currents (feedback)
$getGravity(pos,vel,accel,gravity) compute theoretical currents (gravity model or Newton model, today gravity only) depending of pos, vel, accel (pos only for gravity)
for i=0 to 5
current[i]=current[i]-gravity[i] Compute difference between theoretical and actual currents
endFor
$externalForce(pos,flange,current,force) Compute TCP Cartesian forces and torques corresponding to the current difference

force[0] = along X World in Newton
force[1] = along Y World in Newton
force[2] = along Z World in Newton
force[3] = along RX World in Newton.meter
force[4] = along RY World in Newton.meter
force[5] = along RZ World in Newton.meter