Val3 addons Manual

Download form Mystaubli/Robot/technical data ***Edit by Wang Peng***

List

VAL3 expansion addon……6
bool $getLink(linkedVariable, &link)……6
string $getName(variable)
void $userScreens(num nScreens)
void $title(numScreen, string sText)
void $userPage(numScreen)
void $gotoxy(numScreen, num nX, num nY)
void $cls(numScreen)
void $put(numScreen, string sText)
void $put(numScreen, num nValue)
void $putln(numScreen, string sText)
void $putln(numScreen, num nValue)
num $get(numScreen)
num $get(numScreen, string& sText)
num $get(numScreen, num& nValue)
num $getKey(numScreen)
bool $isKeyPressed(numScreen, num nCode)
num $setPathMode(num nCommand)
num $setPathMode(num nCommand)
num $getPathContext(string& sTarget, string& sTool, string& sCode)……8
num $taskCallStack(string sTaskName, string& sCallStack)……10
num $getState()
bool $delete(string sPath)……11
bool $copy(string sFrom, string sTo)
bool $rename(string sFrom, string sTo)……12
bool $fileExists(string sPath)
string $exec(string sCodeLine)
string $compile(string sCodeLine)
num $fileOpen(string sPath, string sMode)
void $fileClose(num nFileHandler)
num $fileSet(num nFileHandler, string sLines[], num nNbLines)……13
num $fileSet(num nFileHandler, num nBytes[], num nNbBytes)
num $fileGet(num nFileHandler, string& sLines[], num nNbLines)
num $fileGet(num nFileHandler, num& nBytes[], num nNbBytes)
joint $getPosErr(void)
joint $setMaxPosErr(joint jMaxPosErr)……14
void $getJntForce(num& nForce[])
void $setMaxJntForce(num& jMaxJntForce[])……15
num $getSpeedCmd(tool tTool) ……16
num $getSpeedFbk(tool tTool)
point $getPosFbk(tool tTool, frame fFrame)
bool $envelopeError(void)

num $getPowerCount(void)
void $dins(void& variable, num nIndex)
void $ddel(void& variable, num nIndex) ……17
void $getVersions(string& sVersions)
string $getVersion(string sItem)
void $setTextMode(num nMode)
num $getMoveId(void), void $setMoveId(num nId)
void $stopMove(num nStopTime)
void $jogPage(bool bEnable)
void $jogContext(tool tJogTool, frame fJogFrame, num nStep)……18
void $jogj(joint jTarget, tool tTool)
void $jogj(point pTarget, tool tTool)
void $jogl(point pTarget, tool tTool, num nStep)
point $align(point pPoint, frame fFrame)
bool [lib:]$getJoint(string sName, num nIndex, joint& jJoint)
bool [lib:]$getPoint(string sName, num nIndex, point& pPoint)
bool [lib:]$getTool(string sName, num nIndex, tool& tTool)
bool $setDateTime(num nYear, num nMonth, num nDay, num nHour, num nMin, num nSec)12
num $sioCtrl(sio siPort, string sAttribute, void attributeValue) ……19

VAL3 Iterative Learning Control addon……20
void $trajMove(num nTrajId, tool tTool)……21
num $trajLoadRec(string sRecordFile,string sJntPosVar)……22
bool $trajStore(num nTrajId, string sTrajFile)
num $trajLoad(string sTrajFile)
void $trajClose(num nTrajId)
joint $trajStartJoint(num nTrajId)
num $trajConstSpeed(num nTrajId, tool tTool, num nTransVel, num nAccelTime)..23
num $trajAdjust(num nTrajId, string sRecordFileName)

VAL3 Velocity addon……25

void $velFrame(frame fReference, tool tTool, mdesc mDesc)
void $velJoint(tool tTool, mdesc mDesc)
void $velTool(tool, mdesc)
void $setVelCmd(num cmd[6])
void $velCmd(num cmd[6])

VAL3 motion addon……27
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)

num $rotAngle(trsf tTransformation)
num $getRot(trsf tTransformation, num& nXaxis, num& nYaxis, num& nZaxis)
trsf $setRot(num nAngle, num& nXaxis, num& nYaxis, num& nZaxis)
void $setCartJogAccel(num nTranslationAccel, num nTranslationDecel, num nRotationAccel, num nRotationDecel)
bool $setLength(num nLengths[])……28
bool $setDH(trsf trDHs[])
bool $setOffset(joint jPosition)……29
void $velFrame(frame fReference, tool tTool, mdesc mDesc)
void $velJoint(tool tTool, mdesc mDesc)
void $velTool(tool, mdesc)
void $setVelCmd(num cmd[6])
num $drvSelect(string sSerialName, num nAxis)
num $drvSendCommand(string sSerialName, string sCommand)
num $drvGetAnswer(string sSerialName, string sAnswer)
void $alterAtc(num nAtc[])
void $initFirFilter(num nDeltaT, num nPeriod, num& nPositionCoefs)
void $initFirFilter2(num nDeltaT, num nPeriod, num& nPositionCoefs[], num& nVelocityCoefs[])
num $firFilter(num& nCoefs[], num& nInputs[])
num $getBoxcarDelay()……30
point $getPosFbk(tool tTool, f fReference)
joint $getJntFbk()
joint $getJntPosRef()
void $getJntSpeedCmd(num& nSpeed[])
void $getJntSpeedRef(num& nSpeed[])
void $getJntSpeedFbk(num& nSpeed[])
void $getJntForce(num& nForce[])
bool $addInertia(num nAxis, num nMass, trsf tGravityCentre, num& nInertia[])
bool $setMaxJntVel(num& nMaxJointVel[])……31
num $setMaxTvel(num nMaxTvel)
num $getMoveId(void)
void $setMoveId(num id)
bool $setFriction(num nAxis, num nFriction, num nV0, num nV1)
void $stopMove(num nStopTime)
void $setBoxcarFreq(num nJointFreq)……32
void $setBoxcarFreq(num nCartFreq, num nJointFreq)
bool $coggingParams(num nAxis, num nNbHarms, num nParams[])
bool $coggingEnable(num nAxis, num nNbHarms)
void $getGravity(joint jPosition, num& nVelocity[], num& nAcceleration[], num& nForce[])
void $externalForce(joint jPosition, tool tTool, num& nJointOverForce[], num& nCartForce[])

VAL3 Tracking addon……33
void $trackOn(point pPoint, tool tTool, mdec mDesc)
void $trackOn(point pPoint, tool tTool, mdec mDesc)
void $moveIt(point pPoint, tool tTool, mdesc mDesc)
void $movect(point pIntermediate, point pTarget, tool tTool, mdesc mDesc)
void $trackOff(point pPoint, tool tTool, mdec mDesc)……34
bool $setMobile(frame& fFrame, bool bMobile)
bool $updateFrame(frame& fFrame, trsf tPosition, trsf tSpeed)
num $trackingState()
void $setTrackingAcc(num nMaxTransAcc, num nMaxTransDec, num nMaxRotAcc, num nMaxRotDec)……35

VAL3 Bitlib addon……36
num $bitInt8ToNum(num nSignedByte)
num $bitInt16ToNum(num nSignedWord)
num $bitInt32ToNum(num nSignedLong)
num $bitUint8ToNum(num nUnsignedByte)
num $bitUint16ToNum(num nUnsignedWord)……37
num $bitUint32ToNum(num nUnsignedLong)
bool $bitFlt32ToNum(num nSingleFloat, num& nNumber)
bool $bitFlt64ToNum(num nDoubleFloatLowerBytes, num nDoubleFloatHigherBytes, num& nNumber)

num $numToBitInt8(num nNumber)
num $numToBitInt16(num nNumber)
num $numToBitInt32(num nNumber)
num $numToBitFlt32(num nNumber)
void $numToBitFlt64(num nNumber, num& nDoubleFloatLowerBytes, num& nDoubleFloatHigherBytes)……38

num $bytesToBit32(num nBytes[4])
void $bit32ToBytes(num n32bitMask, num& nBytes[4])

num $and(num 32bitMask1, num 32bitMask1)
num $or(num, num)
num $xor(num, num)

VAL3 move id……39
VAL3 Synchronous movej ("procedural motion")……40
num $movejSyncBegin(void)
bool $movejSync(joint jCommand)
num $movejSyncState(void)
bool $movejSyncEnd(void)

VAL3 Cartesian blending……41
VAL3 compliance runtime license……42
VAL3 PLC runtime licence……44
VAL3 LLI option and runtime license……47
VAL3 Remote MCP runtime license……49
VAL3 Remote Maintenance runtime license……51
VAL3 oemLicence runtime license……52
VAL3 alter runtime license……54

How to change the direction of rotation in a movel ?.....55
How use Starc Encoder in Emulator (Cell SRS)……56
How to optimize friction compensation ?.....57

(s6.5.5) bool $getLink(linkedVariable, &link)

This instruction copies the link value of the linked variable (point , frame or tool). It returns true if the link is defined, false if null.

(s6.5.5) string $getName(variable)

This instruction returns the name of the specified variable; if the variable is a program parameter passed by reference, the returned name is the name of the variable passed by reference, not the name of the parameter itself. You get the name of the parameter if it was passed by value.

(s6.5.5) void $userScreens(num nScreens)

This instruction specifies the number of additionnal user pages (maximum: 20, default: 0) that can be accessed by pressing User PageUp or User PageDown keys.

Each screen can be programmed independantely of the others by adding the screen number as first parameter of the screen-handling instructions:

void $title(numScreen, string sText)

void $userPage(numScreen)

void $gotoxy(numScreen, num nX, num nY)

void $cls(numScreen)

void $put(numScreen, string sText)

void $put(numScreen, num nValue)

void $putln(numScreen, string sText)

void $putln(numScreen, num nValue)

num $get(numScreen)

num $get(numScreen, string& sText)

num $get(numScreen, num& nValue)

num $getKey(numScreen)

bool $isKeyPressed(numScreen, num nCode)

The screen 0 is the default screen that can be also accessed by specifying no screen number.

The screen number is likely to be replaced with a screen type variable in the official product; you should therefore better use variables to specify the screen number for better maintainability.

(s6.5) num $setPathMode(num nCommand) :

(s6.5) num $setPathMode(num nCommand) :

This instruction is in beta test, its interface may change at any time !

This instruction sends a command to the path debugger and returns the status of the command:

0 command taken into account

-1 not ready for this command

-2 'here' command cannot be done (move target is not a data)

The possible commands are:

0 "Move" command: exit path debugging mode. This command can be sent at any time.

1 "Step" command: complete current move and stop at end of next move (at the start of the blending)
This command can be sent at any time, even if the previous step is not yet completed. If the motion is not yet in path debugging mode, it will enter the mode with the next
executed VAL3 move instruction (caution: what is already in the motion stack is NOT under control of the path debugger).
2 "Break" command: cancel blending for the current move to reach the target point
This command can be sent at any time in path debugging mode ("Processing" /"Moving" / "Waiting break", ignored in "Breaking" / "Waiting" states)
3 "Here" command: modify the current move target (point or joint, movel, movej or movec, alter or not alter, but not (yet) compliance moves)
This command can be sent only in "Waiting" state (see hereafter).

(s6.5) num $getPathContext(string& sTarget, string& sTool, string& sCode)

 This instruction is in beta test, its interface may change at any time !
This instruction gets the current context of the path debugger (current move target position, tool and VAL3 code line), and returns the current state of the path debugging mode.
The target position and tool have the format: "lib:name[index]", with index given as a numerical value (no index if null). The string is empty if the point / tool is the result of a computation (appro).
The sCode string has the format: "program name, line #: code".

The possible states of the path debugging mode are:
-1 "Off": path debugging mode not activated
0 "Reset": path debugging mode active, waiting next VAL3 move instruction
1 "Processing" VAL3 move instruction (preprocessor working)
2 "Moving" to start of blending (preprocessing completed)
3 "Waiting break" at start of blending (possibly at target position if blending is off)
4 "Breaking" to move to the target position
5 "Waiting" at the target position. Ready for the 'Here' command.

- When a resetMotion() is done in path debugging mode, the state of the path debugging mode goes to 5 ("Waiting"): the "Here" command is then possible even if the target was not reached.
- When there is no blending, you still have to issue a 'Break' command to go from state 3 ("Waiting break") to 5 ("Waiting").

The path debugging mode is always Off when a VAL3 program is started.

A very simple VAL3 code for the path debugger interface:

```
cls()
while true
gotoxy(0,13)
put("Move Step Rst Brk Here")
nKey=getKey()
```

```
gotoxy(0,0)
mMoveState=$getPathContext(sTarget,sTool,sCode)
put("debug move state=")
switch (mMoveState)
case -1
put("OFF ")
break
case 0
put("RESET ")
break
case 1
put("PROCESSING ")
break
case 2
put("MOVING ")
break
case 3
put("WAITING BREAK")
break
case 4
put("BREAKING ")
break
case 5
put("WAITING ")
break
default
put("? ")
break
endSwitch
putln("")
putln(sTarget+" ")
putln(sTool+" ")
putln(sCode+" ")
switch (nKey)
case 271
// Got to move mode
$setPathMode(0)
break
case 272
// Got to step mode
$setPathMode(1)
break
case 273
resetMotion()
```

```
break
case 274
// Break: go to point
$setPathMode(2)
break
case 275
// Here: modify point
$setPathMode(3)
break
default
break
endSwitch
delay(0)
endWhile
```

num $taskCallStack(string sTaskName, string& sCallStack)

(s5.3.2, s5.5, s6.3.2+)

This instruction copies the call stack of the specified task in a string array and returns the number of elements written in the array.

The first element of the string array is the root of the call stack, the last is the current instruction of the task.

Each level in the call stack is described with the format: "*projectName*:*programName*(), line *lineNumber*: *codeLine*".

If the specified task is not stopped, only the current code line is copied and the instruction returns 1. If the specified task does not exist, the instruction returns 0.

num $getState()

(s5.3.2, s5.5, s6.3.2+)

This instruction returns the system state as displayed on the RSI (CS8C) or USBI board, with additionnal details listed hereafter.

- If positive, the returned value is the ASCII code of the character displayed on RSI / USBI board.
- Negative values (CS8C only)

-0.1: ARPS error (internal error)

-0.2: ARPS error (power input failure)

-1.x: RSI board error (the digit after comma gives detailed information for R&D only)

-2.1: Starc board stopped (toggle bit failure)

-2.2: Bus between Starc board and arm encoders stopped

-2.3: Bus between Starc board and drives stopped

-2.4: DSI board boot failure (resident software used)

-2.5: DSI board stopped (toggle bit failure)

-3.xy: Arm encoder error, on axis x, error y. -3.47 means "communication protocol error with encoder on axis 4".

y=1: encoder not connected (possibly encoder power supply error)

y=2: communication error between encoder and DSI board (checksum error)

y=3: encoder error ('Alarm': faulty axial adjustment)

y=4: communication error between encoder and DSI board (protocol error, usually harness issue)

y=5: DSI board error: encoder information could not be processed in time (possibly software issue in DSI board)

y=6: encoder error (no motor phase defined)

y=7: encoder error ('Disabled': encoder was not enabled by controller, usually consequence of a boot failure in the controller)

y=8: encoder error ('Overspeed': maximum joint velocity reached)

-4.xy: Drive error (drive fault), on axis x, error y:

y=01: Drive has not been configured (usually consequence of a boot failure in the controller)

y=02: BusOverVoltage

y=03: BusUnderVoltage

y=04: CurrentSensorNoise

y=05: CurrentSensorOffset

y=06: DriveFoldback

y=07: DspNotReady

y=08: HighPowerStillOn

y=09: IpmFault

y=10: FaultLogIsFull

y=11: MotorFoldback

y=12: NoValidCompensation

y=13: OverSpeed

y=14: DriveOverTemperature

y=15: CommunicationErrorPllNotLocked

y=16: PositionError

y=17: PositionIncoherence

y=18: PowerBoardNotDefined

y=19: SsiIsNotInSynchronousState

y=20: WatchdogOrOverrunOrVoltageSupervisor

-5: No MCP or dummy plug connected.

bool $delete(string sPath)

(s5.3.2, s5.5, s6.1+)

This instruction deletes the specified file or directory, and returns TRUE if the file or directory does not exist after execution.

The path must obey the standard VAL3 path specification “[Drive://]path” where the default drive is ‘Disk’ (/usr/usrapp).

bool $copy(string sFrom, string sTo)

(s5.3.2, s5.5, s6.1+)

This instruction copies the specified file or directory recursively to the specified path, and returns TRUE

if the copy could be completed successfully.

The path must obey the standard VAL3 path specification “[Drive://]path” where the default drive is ‘Disk’ (/usr/usrapp).

(s6.6.1) bool $rename(string sFrom, string sTo)

This instruction moves/renames the specified file or directory to the specified path, and returns TRUE if the move/rename could be completed successfully.

The path must obey the standard VAL3 path specification “[Drive://]path” where the default drive is ‘Disk’ (/usr/usrapp). The drives must be the same in the source and destination paths.

bool $fileExists(string sPath)

(s5.3.2, s5.5, s6.1+)

This instruction returns TRUE if the path points to a file, else FALSE (undefined path, or points to a directory).

The path must obey the standard VAL3 path specification “[Drive://]path” where the default drive is ‘Disk’ (/usr/usrapp).

string $exec(string sCodeLine)

(s5.3.2, s5.5, s6.1+, s5.3.2)

This instruction compiles the specified code line, then executes it. A runtime error "invalid parameter value" is generated if the code line cannot be compiled; a more detailed diagnostic is then given in error logger.

s6.5.5 (but not yet s6.6): This instruction compiles the specified code line, then executes it. It returns the compiler error, if any. No runtime error is generated if the code line cannot be compiled.

CAUTION! The code line should not include instructions that access files (libOpen, libDelete, libList, setProfile) or communicate with some devices (sioGet, sioSet, = operator of sio). The VAL3 interpreter will crash if such instructions are used in the specified code line.

(s6.5.5) string $compile(string sCodeLine)

(s5.3.2, s5.5, s6.1+, s5.3.2)

This instruction compiles the specified code line. It returns the compiler error, if any. No runtime error is generated if the code line cannot be compiled.

num $fileOpen(string sPath, string sMode)

(s5.3.2, s5.5, s6.1+, s5.3.2)

This instruction opens a file for read or write access. The path must obey the standard VAL3 path specification “[Drive://]path” where the default drive is ‘Disk’ (/usr/usrapp). The mode must be either "r" (read), "w" (write, delete if exists), or "a" (append). The returned value is an integer in [0-9] that must be used for further file access. If negative, the file could not be open. A maximum of 10 files can be open simultaneously (s6.3+).


Open files MUST be closed with $fileClose.

With s6.1 and s6.2, $fileOpen crashes when trying to open a 2nd file when the 1st one is not closed yet.

void $fileClose(num nFileHandler)

(s5.3.2, s5.5, s6.1+, s5.3.2)

This instruction closes the specified file handler.


Files are not closed automatically when a VAL3 program ends! A file already open cannot be open again !

num $fileSet(num nFileHandler, string sLines[], num nNbLines)

(s5.3.2, s5.5, s6.1+, s5.3.2)

This instruction writes nNbLines found in variable array sLines, and returns the number of effectively written lines. A line end '\n' (0xa in binary) is written after each string of sLines[]. The nFileHandler is the file identifier returned by the $fileOpen instruction.

Exemple: $fileSet(nFh, sLines[10], 7) : writes sLines[10], sLines[11], ... sLines[16] in 7 different lines

num $fileSet(num nFileHandler, num nBytes[], num nNbBytes)

(s5.3.2, s5.5, s6.1+, s5.3.2)

This instruction writes nNbBytes characters (bytes) found in variable array nBytes, and returns the number of effectively written data bytes. Each entry of nBytes is converted into a byte by computing the modulo 256 rest, then rounding it to the nearest integer. The nFileHandler is the file identifier returned by the $fileOpen instruction.

Exemple: $fileSet(nFh, nBytes[10], 7) : writes nBytes[10], nBytes[11], ... nBytes[16]

num $fileGet(num nFileHandler, string& sLines[], num nNbLines)

(s5.3.2, s5.5, s6.1+, s5.3.2)

This instruction reads nNbLines lines, stores each line in an entry of the variable nLines, and returns the number of effectively read lines. A line end is '\n' (0xa in binary); trailing \n and \r characters are removed in the VAL3 string. The nFileHandler is the file identifier returned by the $fileOpen instruction.

Exemple: $fileGet(nFh, sLines[10], 7) : reads 7 lines and store them in sLines[10], sLines[11], ... sLines[16]

num $fileGet(num nFileHandler, num& nBytes[], num nNbBytes)

(s5.3.2, s5.5, s6.1+, s5.3.2)

This instruction reads nNbBytes bytes, store each byte in an entry of the variable nBytes. Return number of effectively read data.

Exemple: $fileGet(nFh, nBytes[10], 7) : reads 7 bytes and store them in nBytes[10], nBytes[11], ... nBytes[16].

joint $getPosErr(void) (use getPositionErr() in VAL3 s6.4+)

This instruction returns the current position error as a joint value.

Position error (or "envelope error") is the difference between the position commanded by the drives and the position reported by the resolvers or encoders. The position error is updated in the controller every 4 ms; because this information takes 4 ms to come from the drives, the position error in the controller was the position error of the arm 4 ms before. The position command takes also 4 ms to be sent to the drives, and then 4 ms again to be applied by the drives. Therefore the position error in the controller was the position error 4 ms ago, for a position command that was sent from the controller to the drives (see **herej()** instruction) 12 ms ago.

The drives and the controller have a safety limit for the maximum position error, that can be lowered (see **$setMaxPosErr**() instruction below), but not increased. When the limit is reached, arm power is automatically removed by the system. The **$getPosErr**() instruction can be used to read the current position error and react in the program to unexpected increases (or decreases) in the position error.

The **$getPosErr**() instruction can also be used to improve accuracy on path with a basic learning control algorithm: the position command is modified to compensate a measured position error. This can be done either by measuring the position error on some VAL3 points and modifying them in consequence; or by measuring the position error every 4 ms (or a multiple) and later modifying in consequence with **alter**() instruction the position command every 4 ms.

Example

In this example we are checking that the position error during a move. “j<jExpected” will be true only when all 6 values in j are less then or equal to the corresponding 6 values in jExpected. If any one of the 6 values is greater, then the comparisons will be false and we will stop the arm with the resetMotion instruction.

```
movel(pPoint[0],flange,mNoBlend)
do
//get current position error
j=$getPosErr()
// are any of the 6 absolute values greater then expected
// actually testing to see is all are less
if (abs(j)<jExpected)==false
//stop the arm
resetMotion()
popUpMsg("Position error was greater then expected")
endIf
//check every 4 milliseconds (the refresh rate of the postion data)
delay(0.004)
// until the move is complete
until isEmpty()
```

joint $setMaxPosErr(joint jMaxPosErr)

This instruction changes the controller safety limit for the maximum position error (see **$getPosErr**() for details). The drives safety limits unchanged. The safety limit can not exceed the initial, system-defined limit: the instruction automatically reduces the specified limit if it is too high. The instruction returns the effective limit after execution.

This instruction is effective immediately (the safety check is done every 4 ms).

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.

The joint force is computed from the measured motor currents ; it is updated every 4 ms. This force is the force at the joint level (after gearbox reduction), not the force at the motor level (motor torque).

The drives and the controller have a safety limit for the maximum position error, that can be lowered (see **$setMaxJntForce**() instruction below), but not increased. When the limit is reached in the drives, the motor current is automatically limited ; this will result in an increased position error. The controller safety limit is by default set to a very high value; when the controller safety limit is reached, arm power is automatically removed by the system. The **$getJntForce**() instruction can be used to monitor for any sudden increase in motor force that might indicate an error such as a part not loading correctly into a fixture (collision).

```
if bTeach==true
// teaching expected joint force
//zero array of maximum joint force values
for i = 0 to size(nCurrent)-1
nMax[i]=0
endFor
do
//get current joint force array
$getJntForce(nCurrent)
//set maximum value
for i = 0 to size(nCurrent)-1
nMax[i]=max(nCurrent[i],nMax[i])
endFor
//repeat every 4 milliseconds, update rate of servo data
delay(0.004)
// until movement is finished
until isEmpty()==true
else
// to test current motion again maximums
bError=false
do
// get current force
$getJntForce(nCurrent)
//does it exceed the maximum value plus 10%
for i = 0 to size(nCurrent)-1
if nCurrent[i]>nMax[i]*1.1
//stop the arm and signal the error
resetMotion()
popUpMsg("Joint force higher then expected")
bError=true
endIf
endFor
delay(0.004)
until isEmpty()==true
endIf
```

void $setMaxJntForce(num& jMaxJntForce[])

This instruction changes the controller safety limit for the maximum joint force (see **$getJntForce**() for details). The drives safety limits unchanged. The safety limit can not exceed the initial, system-defined limit: the instruction automatically reduces the specified limit if it is too high. The instruction modifies the input array with the effective limit after execution.

This instruction is effective immediately (the safety check is done every 4 ms).

num $getSpeedCmd(tool tTool) (use getSpeed() in VAL3 s6.4+)

num **$getSpeedCmd**(tool tTool, frame fFrame, num& nX, num& nY, num& nZ, num& nRx, num& nRy, num& nRz)

These instructions return the current Cartesian speed (mmps) at the specified tool center point. It is updated every 4 ms with each new command of the motion generator. This speed is effective 8ms after (delay to send command to the drive and to apply it).

The second instruction also return the coordinates of the velocity and rotation vector in the specified frame.

See also **$getSpeedFbk**() below.

num $getSpeedFbk(tool tTool)

num **$getSpeedFbk**(tool tTool, frame fFrame, num& nX, num& nY, num& nZ, num& nRx, num& nRy, num& nRz)

Same as **$getSpeedCmd**, but the Cartesian speed is updated every 4 ms with each new feedback from the drives. This speed was measured on the arm 4 ms before (delay to receive feedback from the drives).

point $getPosFbk(tool tTool, frame fFrame)

This instruction returns the current Cartesian position at the specified tool center point. It is updated every 4 ms with each new feedback from the drives. This position was measured on the arm 4 ms before (delay to receive feedback from the drives).

The only difference with **here**() is that **here**() is based on the command of the motion generator instead of the drives feedback.

bool $envelopeError(void)

This instruction returns true if there was an envelope error since the last arm power on. This does not necessarily imply that the arm power shutdown was due to an envelope error: when an hardware error is detected during a move (such as drive fault), this will usually also induce an envelope error that is only the consequence of the initial defect.

This instruction can be used in automatic mode to help decide whether it is safe to enable power automatically or if a human intervention is desirable.

num $getPowerCount(void)

This instruction returns the arm power counter, in hour. This is the total number of hours on the controller with arm power enabled (this information is not linked to the arm).

void $dins(void& variable, num nIndex)

This instruction inserts an element in a global array variable at the specified index (between 0 and array size); any global variable type is supported, but local arrays are not supported.

A runtime error is generated if the first parameter is not a variable, if index is invalid or in case of memory problem.


This instruction is not real--time proof, it may lock the VAL3 interpreter during a significant time

(tenths of seconds) if the variable is a large array and controller memory is low.

void $ddel(void& variable, num nIndex)

void **$ddel**(void& variable, num nIndex)

This instruction destroys an element of a global array variable at the specified index (between 0 and array size - 1); any variable type is supported, but of size greater than 1. Local array variables are not supported.

A runtime error is generated if the first parameter is not a variable, if its size is 1, if index is invalid or in case of memory problem.


This instruction is not real--time proof, it may lock the VAL3 interpreter during a significant time (tenths of seconds) if the variable is a large array and controller memory is low.

void $getVersions(string& sVersions)

This instruction modifies the string parameter so that it contains the version identifier for the different components of the controller. The size of the parameter is automatically updated to be large enough. This instruction should better not be used and be replaced with the **$getVersion**() instruction.

string $getVersion(string sItem) (use getVersion() in VAL3 s6.4+)

This instruction returns the version of the specified item, as displayed in the control panel on MCP, or "" (empty string) if the item is not defined or not installed.

The item name is not case sensitive. Supported items are:

- "VAL3" version of VAL3 software system
- "Arm" arm type and tuning
- "Configuration" version of the controller and arm configuration file
- "SerialNumber" serial number of the controller (defined in /sys/configs/options.cfx)
- "ArmSerialNumber" serial number of the arm attached to the controller (defined in /usr/configs/arm.cfx)
- the names of the runtime licences: ("alter", "compliance", "remoteMcp", "valPaint", "valPlast", "plc", "testMode", "oemLicence")

The returned value is then either "enabled", "demo", or "" (empty string), when the licence is not installed or demo mode has expired.

- the names of the addons ("Expansion", "Motion", "Protect", Velocity", etc...)

void $setTextMode(num nMode)

This instruction sets the text mode for the next put instructions.

0 = normal display

1 = video inverse

2 = blinking

3 = blinking video inverse

num $getMoveId(void), void $setMoveId(num nId)

see Move Id specification

void $stopMove(num nStopTime)

This instruction is equivalent to the stopMove() instruction, but deceleration is modified to stop in the specified time (Caution: enveloppe error will result if stop time is too low !).

void $jogPage(bool bEnable)

This instruction enables or disables the display of the jog page. It can be used to keep the user page

displayed when entering the jog mode. It is effective only for the next start of the jog page (no effect if the jog page is already activated)

(s6.5.5) void $jogContext(tool tJogTool, frame fJogFrame, num nStep)

This instruction modifies the current tool and current frame for the jog page on MCP. The step parameter specifies the jog step size (mm or deg); if null, the step mode is disabled.

(s6.5.5) void $jogj(joint jTarget, tool tTool), void $jogj(point pTarget, tool tTool), void $jogl(point pTarget, tool tTool, num nStep)

These instructions set the mode for the jog page on MCP to User / joint move, and defines the target position for the move under user control. The step parameter specifies the jog step size (mm or deg); if null, the step mode is disabled.

point $align(point pPoint, frame fFrame) (use align() in VAL3 s6.4+)

This instruction returns the input point with a modified orientation so that the Z direction of the returned point is aligned on one axis on the input frame. The choice of the axis to align is done to minimize the change in orientation.

bool [lib:]$getJoint(string sName, num nIndex, joint& jJoint) (use getData() in VAL3 s6.4+)

This instruction retrieves a joint value in a library, knowing the name of the variable and the index. It returns true if the joint value was successfully updated, false if there is no joint variable accessible in the library or if the index is not correct. If the lib parameter is not specified, the joint variable is searched in the current project or library.

bool [lib:]$getPoint(string sName, num nIndex, point& pPoint) (use getData() in VAL3 s6.4+)

This instruction retrieves a point value in a library, knowing the name of the variable and the index. The modified point is always linked to World, but matches the position in space of the specified point variable (different behavior than **$getTool**).

The instruction returns true if the point value was successfully updated, false if there is no point variable accessible in the library or if the index is not correct. If the lib parameter is not specified, the point variable is searched in the current project or library.

bool [lib:]$getTool(string sName, num nIndex, tool& tTool) (use getData() in VAL3 s6.4+)

This instruction retrieves a tool value in a library, knowing the name of the variable and the index. The modified tool is a copy of the specified tool variable, not necessarily linked to the flange tool (different behavior than **$getPoint**).

It returns true if the tool value was successfully updated, false if there is no tool variable accessible in the library or if the index is not correct. If the lib parameter is not specified, the tool variable is searched in the current project or library.

void $getDateTime(num& nYear, num& nMonth, num& nDay, num& nHour, num& nMin, num& nSec) (use getDate() in VAL3 s6.4+)

This instruction reads the date and time of the controller and store the result in different numerical variables.

bool $setDateTime(num nYear, num nMonth, num nDay, num nHour, num nMin, num nSec)

This modifies the date and time of the controller. Year is in [2000-3000], month in [1-12], day in [1-31],

hour in [0-23], minutes and seconds in [0-59].

num $sioCtrl(sio siPort, string sAttribute, void attributeValue) (use sioCtrl() in VAL3 s6.4+)

This instruction modifies the attributes of a sio data. Supported attributes are:

- "port" num value defining the TCP port for TCP socket servers and clients
- "endOfString" num value defining the ASCII code for end of string delimiter, for TCP socket servers and clients
- "timeout" num value defining the timeout (in seconds) for TCP socket servers and clients
- "clients" num value defining the maximum number of clients for TCP socket servers
- "target" string value in the format "www.xxx.yyy.zzz" (e.g. "10.10.90.1") defining the address of the TCP server, for TCP socket clients.
- "nagle" (s6.5.5) bool value to enable (default) or disable the nagle TCP optimization, for TCP socket servers and clients. See http://en.wikipedia.org/wiki/Nagle%27s_algorithm

The instruction return an error code:

0 no error, attribute is modified with success

-1 unknown attribute

-2 invalid type for the attribute value

-3 invalid value for the attribute value

-4 attribute value could not be modified (not ready)

VAL3 Iterative Learning Control addon

Iterative learning control instructions are available with VAL3 s6.6 by enabling the valTraj runtime license.

1. General description

1.1 Trajectory concept

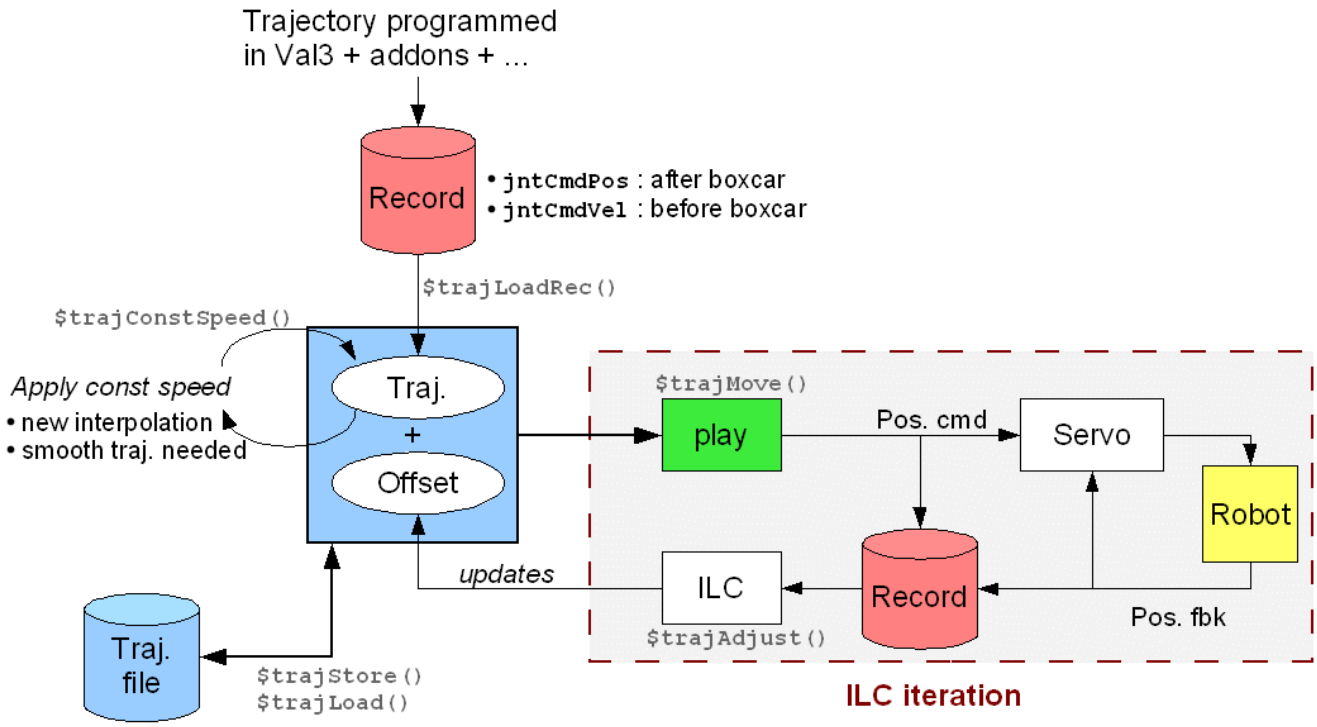
The following instructions bring in the system a new concept of joint trajectories.
These joint trajectories are dedicated to iterative learning control algorithms. They contains:
a nominal sampled joint trajectory. The sampling period may be different of system cycle time (4ms),
a deviation joint trajectory. Deviations are the result of learning control iterations,
the moveId associated to each sample of the trajectory.

1.2 Guidelines to use the add-on

To create a trajectory
- write a normal VAL3 program
- record the trajectory, and store it to a record file (binary format)
- create a new trajectory in memory from the record file (**$trajLoadRec**)
- if necessary, apply constant speed on the trajectory (**$trajConstSpeed**)
- save the trajectory to a trajectory file (**$trajStore**)
- destroy trajectory from memory (**$trajClose**)

To execute the trajectory with the robot
- once, at the beginning of the application, load the trajectories from file to the memory (**$trajLoad**), and get start position (**$trajStartJoint**)
- to execute a trajectory
- move to start point
- execute the trajectory (**$trajMove**)
- at the end of the application, or when a trajectory is not usefull any more, destroy trajectory from memory (**$trajClose**)

To optimize a trajectory using Iterative Learning Control
- load the trajectories from file to the memory (**$trajLoad**), and get start position (**$trajStartJoint**)
- loop
- move to start point
- start recorder
- execute the trajectory (**$trajMove**)
- stop record, and store record file
- adjust trajectory according to recorded movement (**$trajAdjust**)
- until desired precision is obtained
- store optimized trajectory (**$trajStore**)
- destroy trajectory from memory (**$trajClose**)



1.3 Trajectory management in VAL3

- Trajectory IDs

A trajectory loaded in memory is handled in Val3 via an identification number (i.e. trajectory ID). The system manages no more than 30 trajectory IDs (from 0 to 29). So no more than 30 trajectories may be used simultaneously in a VAL3 application. All trajectories are automatically closed when the application is stopped.

- Memory

Each trajectory requires some system memory. The memory allocation is done when the trajectory is loaded (**$trajLoad**, **$trajLoadRec**) or created (**$trajConstSpeed**). The memory is released when the trajectory is closed (**$trajClose**). So the number of trajectories that can be used simultaneously in a VAL3 application also depends on their size, and the size of the running application.

- Concurrent access

A trajectory cannot be modified and executed by the robot at the same time. This is protected by an internal software mechanism. Trying to execute and modify the trajectory will produce errors.

2. Detailled description of each instruction

void $trajMove(num nTrajId, tool tTool)

This instruction records a move along the specified joint trajectory. There is no connection move : the movement starts only if the arm is at the beginning of the trajectory.

No blending is supported with previous and next instruction. In manual mode, maximum joint velocity and cartesian velocity (250mm/s) at the flange and the tool are respected.

A runtime error is generated if the trajectory identity number is invalid.

num $trajLoadRec(string sRecordFile,string sJntPosVar)

This instruction creates a joint trajectory from a record file. It returns a new trajectory identity number. The second argument is the name of the position read in the record : it can be either "jntMotPos" or "jntCmdPos".
If moveId are recorded, the trajectory keeps them. If not, it will range from 0 to 1.

A runtime error is generated if:
the file name is invalid or cannot be read,
the position variable is not "jntMotPos" nor "jntCmdPos",
the position variable is not found in the record,
the trajectory cannot be created.

WARNING: loading a record file that was recorded on another robot type is not detected, and may lead to uncontrolled path geometry (record is joint data).

bool $trajStore(num nTrajId, string sTrajFile)

This instruction stores a joint trajectory in a file. It returns true if the file was correctly written. The file format is specific to joint trajectories.

A runtime error is generated if the trajectory identity number is not valid or the file name is not correct.

num $trajLoad(string sTrajFile)

This instruction load a joint trajectory form a trajectory file. It returns a new trajectory identity number.

A runtime error is generated if:
the file name is invalid or cannot be read,
the trajectory cannot be created.

void $trajClose(num nTrajId)

This instruction erases a trajectory in memory.

A runtime error is generated if the trajectory identity number is not valid or the trajectory cannot be closed.

joint $trajStartJoint(num nTrajId)

This instruction returns the first position of a joint trajectory.

A runtime error is generated if the trajectory identity number is invalid or if the trajectory has not the right number of joints.

num $trajConstSpeed(num nTrajId, tool tTool, num nTransVel, num nAccelTime)

This instruction creates a new joint trajectory with constant speed at tool center point. It returns the new trajectory identity number.

The path is defined by the input joint trajectory. The acceleration time is used at the beginning and at the end of the trajectory (nAccelTime is the time for the velocity to ramp up from zero to the specified constant speed at the beginning of the trajectory, as well as to go down from constant speed to zero at the end of the trajectory).

WARNING : Joint acceleration and velocity are not checked : to be played smoothly by the arm, the input trajectory must be itself smooth (without corners).

Using high velocity values, low acceleration time or if the trajectory has low radius of curvature may lead to possible enveloppe error during trajectory execution, leading to uncontrolled path following.

A runtime error is generated if:

the trajectory identity number is invalid,

the translation velocity or the acceleration time are negative or equal to zero,

the trajectory cannot be created.

num $trajAdjust(num nTrajId, string sRecordFileName)

This instruction reads the desired trajectory in the specified trajectory, and the actual trajectory in the binayr record file. The corrections of the trajectory are updated according to the difference between desired and actual trajectories. The function returns 0 if it worked properly. Otherwise, the return value indicates the error.

The trajectory file must be entirely played at 100% speed (otherwise, the learning algorithm would try to compensated for the trajectory slowing down).

The record must be done at the maximum frequency (normally 250Hz) and it must contain at least the following variables :

jntFbkPos on all joints

moveType

The trajectory must contain the entire trajectory. The position feedback has 3 tics delay with reference to the position command. The record must contain these 3 extra tics.

Return Error code	Description	Input trajectory modified
0	No error	YES, OK
1	Cannot modify trajectory: Trajectory is beeing modified at the same time in another task, or is executed by the robot.	May be only partially modified
2	Cannot open record file: invalid name, file missing, not a proper record binary file	NO
3	Invalid record frequency: record frequency does not match to the controller base cycle time	NO
4	Input record does not contain the variable "moveType"	NO

5	The "jntFbkPos" variable is missing in the record, on one or several joints	NO
6	Cannot find the beginning of the trajectory in the record	NO
7	Record too short : the record does not contain the entire trajectory. Possible causes: - recorder stopped too early because mem full - 3 tics more needed after end of trajectory because of delay between command and feedback	NO
8	Recorded trajectory duration does not match the trajectory duration. Possible causes: - trajectory not played entirely at SP 100% - speed limitation during trajectory execution	NO
9	Corrections computed during trajectory adjustment are too large. Possible causes: - input record does not correspond to the input trajectory - very large tracking errors during trajectory execution	YES, but some trajectory corrections were saturated
10	Not enough memory to perform the operation	NO

A runtime error is generated if the trajectory identity number is invalid,

3. Examples

Computing and saving a constant speed trajectory (200 mm/s and 0.5 seconds of accel/decel) from a record file:

```
nIdRec = $trajLoadRec("Usr://recorder/record.rec","jntCmdPos")
nIdCst = $trajConstSpeed(nIdRec,tTool,200,0.5)
$trajStore(nIdCst,"Usr://trajectories/Traj_v200.trj")
$trajClose(nIdCst)
$trajClose(nIdRec)
```

Loading and playing previous constant speed trajectory:

```
nId = $trajLoad("Usr://trajectories/Traj_v200.trj")
jStart = $trajStartJoint(nId)
movej(jStart,tTool,mDesc)
waitEndMove()
$trajMove(nId,tTool)
waitEndMove()
```

Program example of ILC implementation.

VAL3 Velocity addon

The velocity addon provides motion instructions to control the arm in velocity, rather than in position.

The use of these instructions should be safe (see potential bug with $setVelCmd), safety rules should be applied (monitor speed, move/Hold button, eStop, Cartesian limit in manual mode) but this has not been tested by R&D.

void $velFrame(frame fReference, tool tTool, mdesc mDesc):

Initializes a velocity move in the Cartesian space using a frame as reference.

The motion descriptor specifies the speed and acceleration limits to be applied during the move.

The tool is used for the Cartesian speed computations; the Cartesian speed limits of the motion descriptor are computed for this tool.

void $velJoint(tool tTool, mdesc mDesc):

Initializes a velocity move in the joint space

The motion descriptor specifies the speed and acceleration limits to be applied during the move.

The tool is used for the Cartesian speed computations; the Cartesian speed limits of the motion descriptor are computed for this tool.

void $velTool(tool, mdesc):

(s6.5.3+) The tool is now correctly set, not always flange.

Initializes a velocity move in the Cartesian space using a tool as reference

The motion descriptor specifies the speed and acceleration limits to be applied during the move.

The tool is used for the Cartesian speed computations; the Cartesian speed limits of the motion descriptor are computed for this tool.

$velTool makes the system fail with RS arms in VAL3 -s6.3.

(s6.4+) void $setVelCmd(num cmd[6]):

Specify the speed for the 6 coordinates (mm/s (x, y, z) and deg/s (rx, ry, rz) for frame and tool modes, deg/s (rotoid axis) mm/s (linear axis) for joint mode)

void $velCmd(num cmd[6]) (to be removed with VAL3 s7)

Same as $setVelCmd, but specifies the speed in m/s or rad/s instead mm/s and deg/s.

$velCmd makes the system fail with RS arms in Frame mode in VAL3 -s6.3.

- Once the vel command is initialized, the $setVelCmd instruction can change the speed (and direction with negative speed) in real time. The change is applied by the synchronuous task every 4ms.

The $setVelCmd () is not real-time proof and a partial, possibly corrupted, command can result if the synchronuous task is activated during the execution of the VAL3 velCmd instruction (very rare). This can be avoided by the use of a synchronuous task.

- To stop the velocity command, a resetMotion() is required ($setVelCmd () with cmd = 0 will stop the arm, but the vel instruction is still activated).

A very simple example of use of the vel commands :

```
num nCmd[6]
begin
$velFrame(world,flange,mNominal)
while true
nCmd[0]=100
nCmd[1]=100
$setVelCmd (nCmd)
delay(3)
nCmd[0]=-100
nCmd[1]=-100
$setVelCmd (nCmd)
delay(3)
endWhile
end
```

(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)

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[])

 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[])

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

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}

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 $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, s6.4.1)

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.
s6.6.1+: The instruction now works also at any time, not only for programmed moves.

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.
s6.6.1+: The instruction now works also at any time, not only for programmed moves. 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.

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

VAL3 Tracking addon

Tracking instructions are available with VAL3 s6.4 by enabling the valTrack runtime license.

void $trackOn(point pPoint, tool tTool, mdec mDesc)

This instruction records a joint move to connect to the specified mobile target point (defined in a mobile frame). At the end of the movement, if no other move instruction is recorded, the arm tracks the specified mobile target as long as this remains possible.
The blending type is either off or on (Cartesian and joint blending have same effect)

A runtime error is generated if mDesc has invalid values or if a preceeding recorded move command can not be completed.
Tracking instructions are available with VAL3 s6.4 by enabling the valTrack runtime license.

void $trackOn(point pPoint, tool tTool, mdec mDesc)

This instruction records a joint move to connect to the specified mobile target point (defined in a mobile frame). At the end of the movement, if no other move instruction is recorded, the arm tracks the specified mobile target as long as this remains possible.
The blending type is either off or on (Cartesian and joint blending have same effect)

A runtime error is generated if mDesc has invalid values or if a preceeding recorded move command can not be completed.

void $movelt(point pPoint, tool tTool, mdesc mDesc)

This instruction records a linear move to the specified mobile target point (defined in a mobile frame). At the end of the movement, if no other move instruction is recorded, the arm tracks the specified mobile target as long as this remains possible.
The blending type is either off or Cartesian (joint blending is not supported)

A runtime error is generated if mDesc has invalid values, if a preceeding recorded move command can not be completed or if the preceeding recorded move command is not using the same mobile frame.

void $movect(point pIntermediate, point pTarget, tool tTool, mdesc mDesc)

This instruction records a circular move to the specified mobile target point (defined in a mobile frame), through the specified intermediate point. The tool orientation is interpolated so that it can remain constant in World frame, or relatively to the trajectory. At the end of the movement, if no other move instruction is recorded, the arm tracks the specified mobile target as long as this remains possible.
The blending type is either off or Cartesian (joint blending is not supported)

A runtime error is generated if mDesc has invalid values, if a preceeding recorded move command can not be completed or if the preceeding recorded move command is not using the same mobile frame.

void $trackOff(point pPoint, tool tTool, mdec mDesc)

This instruction records a joint move to disconnect from a mobile point (defined in a mobile frame) and reach the specified target position.

The blending type is either off or on (Cartesian and joint blending have same effect)

A runtime error is generated if mDesc has invalid values or if a preceeding recorded move command can not be completed.

bool $setMobile(frame& fFrame, bool bMobile)

This instruction declares the specified frame as mobile (bMobile true) or fixed (bMobile false). It returns:
true The mobile/fixed state of the frame is set as desired.

false when trying to declare as mobile a frame that is already linked to another mobile frame
or when trying to declare fixed a frame that is being used by a tracking move command.

bool $updateFrame(frame& fFrame, trsf tPosition, trsf tSpeed)

This instruction updates the position and velocity of the specified mobile frame. The position transformation is the transformation from the father frame to the mobile frame; the speed transformation defines the Vx, Vy, Vz, Rx, Ry and Rz coordinates of translation and rotation velocity of the mobile frame defined in the mobile frame (not in the father frame).

The instruction returns:
true if the frame is being used by a tracking move command
false if the frame is not used by a tracking move command

A runtime error is generated if the specified frame was not previoulsy declared as mobile.

num $trackingState()

This instruction returns the state of the tracking move command. It returns:
0 there is no pending tracking move command
1 a trackOn command is being executed
2 a tracking move command (linear or circular) is being executed
3 the arm is tracking a fixed point on a mobile frame
-1 the tracking is in error

For instance, during the execution of the following instructions, the function successively returns :

```
$movej(…) : 0
$trackOn(…) : 1
$movelt(…) : 2
```

```
$waitEndMove(): 3
$movect(…) : 2
$trackOff(…) : 0
```

void $setTrackingAcc(num nMaxTransAcc, num nMaxTransDec, num nMaxRotAcc, num nMaxRotDec)

This instruction sets the cartesian accelerations and decelerations in translation and rotation, for $moveIt() and $movect() moves.

The parameters are in user units (mm/s² or in/s² for translation, deg/s² for rotation).

Default values for most arms are : 2000 mm/s² for translation accel and decel, 300 deg/s² for rotation accel and decel.

Encoder position and velocity filtering :

The positions and velocities given to $updateFrame function must be consistent and not too noisy.

Here is an example showing how to filter an encoder input and how to compute the velocity using **$initFirFilter2** and **$firFilter** functions :

nPositionCoefs, nVelocityCoefs, nInputs and nInputsCopy are four num arrays of the same length N.

To init the FIR filter :

```
nPeriod = 0.004 // Cycle time (seconds)
nTimeDelay = ($getBoxcarDelay()*0.001)/2 + nCommunicationDelay // Total delay (seconds)
$initFirFilter2(nTimeDelay, nPeriod, nPositionCoefs,nVelocityCoefs)
```

Each 4ms, in a synchronous task :

```
nInputs[0] = nCoder*nRatio+nOffset
nInputsCopy[0] = nInputs[0]
l_trPos.x = $firFilter(nPositionCoefs, nInputs)
l_trVel.x = $firFilter(nVelocityCoefs,nInputsCopy)
$updateFrame(fMoving,l_trPos,l_trVel)
```

Use the length N of the arrays to filter more or less the encoder data. Some conveyor tracking application use N=80 for instance.

VAL3 Bitlib addon

The bitlib addon provides tools to handle a num as a 32bits mask. Such bitmasks are integer values between 0 and 2^32=4294967296 (UINT32).

The tools makes it possible:

- to interpret a bitmask a a signed or unsigned byte (8bits), word (16 bits), long (32 bits), single precision float (32 bits) or double precision float (64 bits).
- to convert a numerical value into a bitmask matching a signed or unsigned byte (8bits), word (16 bits), long (32 bits), single precision float (32 bits) or double precision float (64 bits).
- to assemble four 8 bits masks into a 32 bits mask, and vice-versa
- to compute basic bit operation on a bitmask

Bit field operation is supported with the standard VAL3 product since VAL3 s6.4. The bitlib addon will be discontinued with the next major VAL3 release s7.

To interpret a bit mask as a numerical value:

num $bitInt8ToNum(num nSignedByte)

Usage:

num nNumber, num nSignedByte

nNumber = $bitInt8ToNum(nSignedByte)

Compute the numerical value matching the specified 8 bits bitmask.

Interprets sSignedByte as the 8 bits of a signed byte integer and convert it into the internal num format. Result ranges from -128 to +127.

sSignedByte is rounded down, the rest of the integer division by 256 defines the 8 bits of a signed 8bits integer (sign is given by the 8th bit).

Example:

nSignedByte = 1.2; is rounded to 1; corresponding 8bits signed integer mask is 0 0 0 0 0 0 0 1; $bitInt8ToNum returns 1

nSignedByte = 258.5 is rounded to 258 modulo 256 = 2; $bitInt8ToNum returns 2

nSignedByte = 255 is rounded to 255; corresponding 8bits signed integer mask is I I I I I I I I; $bitInt8ToNum returns -1

num $bitInt16ToNum(num nSignedWord)

Same as $bitInt8ToNum but nSignedWordis interpreted as a signed 16 bits integer. Result ranges from -32768 to +32767.

num $bitInt32ToNum(num nSignedLong)

Same as $bitInt8ToNum but nSignedLong is interpreted as a signed 32 bits integer. Result ranges from -2147483648 to +2147483647.

num $bitUint8ToNum(num nUnsignedByte)

Same as $bitInt8ToNum but nUnsignedByte is interpreted as an unsigned 8 bits integer. Result ranges from 0 to +256.

num $bitUint16ToNum(num nUnsignedWord)

Same as $bitInt8ToNum but nUnsignedWord is interpreted as an unsigned 16 bits integer. Result ranges from 0 to +65536.

num $bitUint32ToNum(num nUnsignedLong)

Same as $bitInt8ToNum but nUnsignedLong is interpreted as an unsigned 32 bits integer. Result ranges from 0 to +4294967296.

bool $bitFlt32ToNum(num nSingleFloat, num& nNumber)

Interprets nSingleFloat as the 32 bits of a single precision float and convert it into the num format.
Returns true if the specified 32 bits match a valid single precision float, else nNumber is unchanged and the funcion returns false.

bool $bitFlt64ToNum(num nDoubleFloatLowerBytes, num nDoubleFloatHigherBytes, num& nNumber)

Interprets nDoubleFloatLowerBytes (resp. nDoubleFloatHigherBytes) as the 32 lower (resp. higher) bits of a double precision float and convert it into the num format.
Returns true if the specified 64 bits match a valid double precision float, else nNumber is unchanged and the funcion returns false.

To convert a num into a bit mask:

num $numToBitInt8(num nNumber)

Usage:
num nNumber, num nSignedByte
nSigneByte = $numToBitInt8(nNumber)

Computes the signed 8 bits bitmask matching the specified numerical value.
nNumber is rounded to the nearest integer, converted it to a signed byte, and the corresponding bitmask is returned. Result is a 8bit bitmask (ranging from 0 to 256).

num $numToBitInt16(num nNumber)

Compute the signed 16 bits bitmask matching the specified numerical value.
nNumber is rounded to the nearest integer, converted it to a signed word, and the corresponding bitmask is returned. Result is a 16bit bitmask (ranging from 0 to 65536).

num $numToBitInt32(num nNumber)

Compute the signed 32 bits bitmask matching the specified numerical value.
nNumber is rounded to the nearest integer, converted it to a signed long, and the corresponding bitmask is returned. Result is a 32bit bitmask (ranging from 0 to 4294967296).

num $numToBitFlt32(num nNumber)

Compute the 32 bits bitmask of a single precision float matching the specified numerical value.
nNumber is rounded to the nearest single precision float, and the corresponding bitmask is returned. Result is a 32bit bitmask (ranging from 0 to 4294967296).

void $numToBitFlt64(num nNumber, num& nDoubleFloatLowerBytes, num& nDoubleFloatHigherBytes)

Compute the 64 bits bitmask of a double precision float matching the specified numerical value.
nNumber is converted in a 64 bits bitmask, and returned in two 32bits bitmasks matching the lower and higher parts.

Byte operations:

num $bytesToBit32(num nBytes[4])

Usage:
num n32BitMask, num nBytes[4]
n32BitMask = $bytesToBit32(nBytes)

Compute the 32 bits bitmask consisting in four 8bits bitmask : result is nByte[0] + nByte[1]<<8 + nByte[2]<<16 + nByte[3]<< 24
nByte content is rounded down into unsigned 8bits bytes before operation. Result is a 32bit mask (ranging from 0 to 4294967296).

void $bit32ToBytes(num n32bitMask, num& nBytes[4])

Computes in nByte array the four bytes of the specified 32 bits bitmask : n32bitMask = nByte[0] + nByte[1]<<8 + nByte[2]<<16 + nByte[3]<< 24
n32bitMask is rounded down into unsigned 32bits long before operation. Result is an array of four 8bit masks (ranging from 0 to 256).

Bit operations:

num $and(num 32bitMask1, num 32bitMask1)

Logical bitwise and. Input data are first rounded down into a 32bits integer value. Result is a 32bit mask (ranging from 0 to 4294967296).

num $or(num, num)

Logical bitwise or. Input data are first rounded down into a 32bits integer value. Result is a 32bit mask (ranging from 0 to 4294967296).

num $xor(num, num)

Logical bitwise xor. Input data are first rounded down into a 32bits integer value. Result is a 32bit mask (ranging from 0 to 4294967296).

VAL3 move id

Implemented in s5.3.2 ; change in blending behaviour in s6.0 and s5.4 ; change in resetMotion in s5.4

- each move instruction now returns a num value, that is automatically incremented by the system with each move: this is the move id.
- the num **$getMoveId**(void) (motion / expansion addon) returns the id of the current move. The decimal part is the % of progress on the move: for instance, 17.57 means that the current command position is at 57% of the move with id 17. This % position is in advance of roughly 50ms on the real arm position. This unaccuracy will be improved in the future.
- move id is interpolated in blending moves

nId = movel(pointA) // return 15

nId = movel(pointB) // return 16 (with or without blending)

waitEndMove() // After waitEndMove(), $getMoveId() always return the last move Id +1, here 17

On this example:

$getMoveId() = 15.8 means that the current position is at 80% of the move to point A

$getMoveId() = 16.57 means that the current position is at 57% of the move between point A and point B.

$getMoveId() = 17 means that it is at 100% of the move 16, so it is *at* point B.

If blending leave point is at 80% of move 15 and reach point is at 40 of move 16, then blending will start at 15.8 and stop at 16.4 ; then $getMoveId() = 16 corresponds to 1/3 of the blending move, 16.1 to the middle of the blending move.

$getMoveId() is therefore a continuous function, except for null moves:

nId = movel(pointA) // Return 15

nId = movel(pointA) // Return 16

nId = movel(pointB) // Return 17

$getMoveId() will "jump" from 16 to 17, because move 16 is cancelled.

- the id of the move is reset to 0 after resetMotion.
- the instruction void **$setMoveId**(num id) resets the id for the next move command. It can be used to make sure that one specific move always has the same id.

resetMotion() => reset move Id to 0

nId = $getMoveId() => returns 0

setMoveId(1000)

nId = $getMoveId() => return 0

nId = movel(...) => return 1001

VAL3 Synchronous movej ("procedural motion")

The "procedural motion" makes it possible to specify a joint position to the robot every 4ms, that will be applied immediately (with a delay of 8ms) to the arm. This is done with only basic safety checks (arm joint limits, arm velocity & accel limits, enveloppe error).

The "procedural motion" is started with the **$movejSyncBegin** instruction; then the **$movejSync** instruction should be called every 4ms. The procedural motion mode is exited with either the **$movejSyncEnd** instruction, or **resetMotion** (hopefully when the arm speed is null, but this is NOT checked...).

These instructions are installed on all controllers, without addon, since s5.3 (s5.3.2 for the motion Id return code of **$beginMovejSync**). To be activated, both the alter licence and the valTraj license (valPaint or PaintiXen license for -s5.5) must be effective (possibly in demo mode).
If the alter licence is in demo mode, **$movejSyncBegin** will not work any more after 2 hours.

num $movejSyncBegin(void) : Push a movejSync sequence on the motion stack; will be effective when last move is done.
Returns the move Id of the corresponding move (effective until $endMovejSync is called)
bool $movejSync(joint jCommand) : Send the joint command for the next 4ms. Returns true if command accepted, false if refused
(movejSync instruction not started or already completed)
num $movejSyncState(void) : Get the state of the current movejSync sequence:
0 movejSynch not started
1 movejSynch running, no error
2 movejSynch stopped in error (after invalid command)
3 movejSynch running, last command was not refreshed in time
4 movejSynch running, commands are locked because motion is stopped
(eStop, moveHold, stopMove())
bool $movejSyncEnd(void) : Stops the current movejSynch sequence ; returns false if movejSynch sequence not started or
already stopped

The **alterStopTime**() instruction can be used to check how much time (s) there is before motion is stopped (moveHold, eStop, stopMove()). If 0, motion is already stopped. If negative, motion is not being stopped.

VAL3 Cartesian blending

How to enable Cartesian blending?

Cartesian blending is available as beta version with VAL3 s6.3+: the 'Cartesian' keyword is defined and can be used to configure a motion descriptor:

```
mMyMdesc.blend = Cartesian
```

Because it is a beta version:

- to avoid incompatibilities with SRS VAL3 Studio, the Cartesian blending is NOT supported in the .dtx file, and is replaced with 'joint' blending. Therefore the Cartesian blending must be initialized in the VAL3 code as described above.
- the 'Cartesian' blending type cannot be selected from the motion descriptor editor on MCP, until a motion descriptor already configured with Cartesian blending is edited. To make Cartesian blending directly accessible from the motion descriptor editor on MCP after reboot, use a 'magic flag' option.

Cartesian blending should be supported in the official product in July 2008.

What is the difference between Cartesian and joint blending ?

To make it very simple, a joint blending is like a movej between the leave and reach points, the Cartesian blending is like a movec between these points.

- Joint blending is usually faster than Cartesian blending ; but joint blending may lead to strange path when there is complex orientation change (typically a circle in a plane followed with a circle in an orthogonal plane), or for pure rotation moves.
- The speed & accel control is more accurate with the Cartesian blending.

The shape of the Cartesian blending is a Bezier curve (a bit more complex than a circle or ellipse). The most optimized shape depends on the application, but the VAL3 interpreter has to choose the shape automatically. The choice may lead to a shape that is not convex when the effective leave and reach distances are quite different: this 'S' shape reduces the curvature of the path for speed optimization, but may not be desirable for some process application. When leave and reach distances are equal, the Cartesian blending always result in a convex shape.

The Cartesian blending applies to both position & orientation. A complex orientation change may impact the shape of the blending, leading to unexpected results. There are also some restrictions with orientation change: like on a circle, big changes in orientation result in a motion error when several solutions are possible but the system has no criteria to choose one. When this happens, additionnal intermediate point(s) are needed to help the system find the correct orientation interpolation.

VAL3 compliance runtime license

A compliance software is available from CS8 SW s3.x. "Compliance" is a general term which includes very different technologies, whereas the CS8 controller's compliance software only covers a limited sphere of application, as explained below.

What is compliance?

A robot is called "compliant" as soon as it stands some variations between the controlled and the real arm position. Usually, the robot tries on the contrary to respect the controlled trajectory at best; when variations occur, the motors are prompted by the robot to catch up with the desired position.

How does it work?

The implementation of compliance in the CS8 controller is based on an assessment of forces undergone by the arm, from the motors' current values. The trajectory generator then modifies the arm trajectory in real time to control the external force, for instance by minimizing it ("released" arm) or by keeping it close to a predefined value (force control).

From a programming point of view, a compliant motion can be programmed very easily using a **movelf**() VAL3 instruction similar to the usual movel() instruction. But the transition between a usual motion and a compliant motion must be managed rigorously (you should wait until the arm is effectively compliant before controlling the jack!).

What is supported, what is not?

An arm can be compliant in various ways, each of them corresponding to different applications. For a given application, it is thus necessary to know which type of compliance is required, for instance:

- When **picking** a part with a tool, if the part is not perfectly centered or oriented you may wish the arm to recover its position in order not to force the part. Then, a total arm motion freedom when picking the part is desirable (arm "released", or arm "loose" like a spring).
- When **inserting** a part (or the tool), the arm must remain "master" as regards the direction of insertion, but it must be compliant in the plane orthogonal to the insertion.
- To come into **contact**, the arm must remain "master" in one direction while being able to stop as soon as contact is detected. Then, a force control in this direction is required. Additionally, you may also wish the part to adapt to the contact surface; in this case, compliance in orientation is needed.
- When the arm is subject to an external force (molding machine's ejection jack for instance), you may wish on the contrary that the arm, pushed by an uncontrolled external force, follows a defined trajectory. The arm must then be compliant in the motion direction but "rigid" in the other directions so as to keep on the trajectory.

The current implementation of compliance in the CS8 controller corresponds to this last scenario only: the arm is compliant (with a force control) in the motion direction but remains "rigid" in the other directions and in orientation. Our targeted market is molding machines' ejection. This implementation also allows to come into contact in certain conditions (without compliance in orientation), but few checks

have been done on this use and further testing will be required before we can commit ourselves to a result.

Limitations

The assessment of external forces from currents greatly depends on the conditions of use. It varies with the axes (reversibility, friction), with the arm position or with the speed. It will be even better since external forces exert high torques on reversible axes (main axes, in particular axis 1): compliance on the wrist orientation is very limited.

The accuracy obtained as regards force assessment is at best of the order of the kilogramme; a much better accuracy could be available using a force sensor (not planned for the time being).

Today, the force control parameter of the VAL3 instruction is significantly restricted. If this limitation is a brake upon the development of an application, please contact Stäubli Faverges.

Other limitations concerning the trajectory generator are described in the VAL3 handbook (Chapter "Options").

Availability

To support compliance, each arm must be adjusted so as to identify certain parameters (weight, friction, ...).

Compliance is not yet available on the following arms:

- RX60B EX, RX60B LEX, RX130B XL, RX170B, RX170B HP, RX260, RX260 L, RX270,
- TX 40, TX60, TX60 L,
- RS family arms.

Compliance is supported for ceiling-mounted but not for wall-mounted arms. For the RX60B and RX60B L arms, you should make sure that mounting is correct in the "arm.cfx" configuration file.

The technology implemented for this first compliance solution will be reused and enriched in order to meet the other above mentioned needs. However, this improvement will require significant efforts in Research & Development which will be planned depending on the needs that have been identified in the field and passed on to the R & D Dept.

Installation

The compliance software is installed on all CS8 controllers but is protected by means of a runtime license. To activate this runtime license, a password needs to be added in the options.cfx configuration file.

Note: It is also possible to activate the runtime license without password for a limited access time of two hours to allow people to test it before purchasing.

The purchase of a compliance runtime license to be used on an already delivered controller must be managed like the purchase of a spare part. A user license with the password to be installed will be supplied to the customer.

To allow password calculation, the CS8 controller's « MacAddress » displayed on the control panel must be given. Since « MacAddress » is a CS8 controller PC board's parameter, changing a PC board will make the password's update necessary.

VAL3 PLC runtime licence

What is the PLC option?

The PLC option allows to write and run programs written with IEC 1131 Standard PLC languages (ST, ladder, ...) on a CS8 controller.

Operation

The PLC program must be developed using the optional PLC development environment of SRS. A runtime license is then required for each controller that must run the PLC program.

The PLC program is launched automatically at boot. It is scheduled with a configurable cycle time. The VAL3 program is executed between two cycles of the PLC. The PLC's effective cycle time is displayed on the MCP so that it is possible to evaluate the CPU used by the PLC and the CPU available for VAL3.

The PLC program interacts with the system and the VAL3 program by means of the controller's digital and analog I/Os.

Performance

The cycle time for the software PLC in the CS8 controller is the sum of:
- the PLC context update (impact of the number of inputs and outputs),
- the PLC execution time (impact of the PLC language and the program complexity),
- the CS8 system overhead (impact of the VAL3 version and the CPU clock).

The effective cycle time for the PLC hardware signals also depends (often essentially) on the hardware refresh delay (depending itself on the hardware technology and configuration).

The approximate cycle time for PLC programs evaluating 5 logical operations for each input and output, at each cycle, is given in the following table:

Cycle time table	Fieldbus		Modbus		BIO / MIO	
CPU	teknor (1)	men (2)	teknor (1)	men (2)	teknor (1)	men (2)
32 I / 32 O	0.8 ms	1.1 ms	0.6 ms	0.8 ms	0.6 ms	0.8 ms
128 I / 128 O	1.8 ms	2.4 ms	1.4 ms	1.8 ms	-	-
512 I / 512 O	6.5 ms	8.5 ms	3.5 ms	4.6 ms	-	-
Hardware delay	6 ms or more (3)		16 ms (4)		3 to 18 ms (USBI - CS8) 4 to 7 ms (RSI - CS8C)	

(1) The teknor CPU runs at 650Mhz. It was delivered with CS8 / CS8M controllers

(2) The different men CPUs run at 400Mhz. They are delivered with CS8C / CS8 / CS8M controllers

(3) The hardware delay greatly varies depending on hardware and configuration. 5 or 6 ms is the best case.

(4) The modbus refresh delay is increased when the VAL3 cycle time exceeds 16 ms.

When the PLC development environment is connected to the CS8 controller, a 0.3-ms overhead is added to each cycle time.

The part of the cycle time due to the PLC program execution is about 1 µs per logical expression containing 5 logical operators (1 ms for 512 I / 512 O in the above table).

Limitations

The number of I/Os for the PLC should not exceed 1,000. Indeed, the CS8 controller has not been optimized to support a large number of I/Os.

The CS8 controller does not support serial line or file access in PLC programs.

The integration of the PLC in the system will be improved in the future, so that PLC programs have direct access to the complete robot state (power, working mode, ...).

Installation

The PLC software is installed on all CS8/CS8C controllers but is protected by means of a runtime license. To activate this runtime license, a password needs to be added in the options.cfx configuration file.

Note: It is possible to activate this runtime license without password for a limited access time of 2 hours to allow people to test it before purchase.

The purchase of a PLC runtime license to be used on an already delivered controller must be managed like the purchase of a spare part. A user license with the password to be installed will be supplied to the customer.

To allow password calculation, the CS8 controller's "MacAddress" displayed on the control panel must be notified. Since "MacAddress" is a CS8 controller PC board's parameter, changing a PC board will make the password's update necessary.

VAL3 LLI option and runtime license

What is "LLI"?

The LLI (Low Level Interface) option was designed to allow universities or research centers to use a STÄUBLI robot arm and fully re-program motion algorithms.

The LLI option provides a programming interface in C language which allows in particular:

- to power on/off the arm,
- to control the arm motions by sending instructions every 1, 2 or 4 ms, in position or torque control loop.
- to access to the CS8 controller's I/Os (BIO, MIO, Modbus and field bus),
- to manage the arm calibration.

For a research center, this system is thus completely open and allows the control of a standard industrial arm whose maintenance is done by STÄUBLI, just as the other usual applications.

How does it work?

The LLI option consists of a CD-ROM including programming software tools and a 2-day support by the R & D Dept. (email only) to facilitate commissioning and provide further technical information. It is sold as a spare part. A runtime license is then required for each controller that must run a LLI program.

The CS8 controller is delivered in its usual VAL3 configuration; it can easily be reconfigured to support either the VAL3 language or the LLI programming interface.

A deeper description of this option is done in the article presented at ISR 2004 (March 2004):

ISR2004.pdf

Limitations

The provided programming tools allow the creation of an emulator on PC using Microsoft Developer Studio, and an embedded real time version using Windriver's Tornado development environment. These environments must be purchased by the customer; our R & D support does not include any support on these products.


The R & D agreement is required before selling a LLI option.

Installation

The LLI program must be built and installed by the customer. A readme file is delivered with the LLI CD-ROM.

To activate the LLI runtime license, a password needs to be added in the options.cfx configuration file.

The purchase of a "LLI" runtime license to be used on an already delivered controller must be managed like the purchase of a spare part. A user license with the password to be installed will be supplied to the customer.

 To allow the password calculation, the CS8 controller's « MacAddress » displayed on the control panel must be given. Since « MacAddress » is a CS8 controller PC board's parameter, changing a PC board will make the password's update necessary.

VAL3 Remote MCP runtime license

What is "Remote MCP"?

The "Remote MCP" runtime license was designed to allow the replacement of a Stäubli MCP with a MCP developed by an OEM customer or an integrator. It enables the customer to:

- change working modes,
- control power on, also in manual mode,
- control manual displacement motions (speed command),
- control validation signals in manual mode (deadman and park),
- eliminate speed limitation in test mode.

For instance, it allows an OEM to integrate the complete management of the robot in his own controller, using his own user interface. It also allows an integrator to replace the Stäubli MCP with another MCP better suited to the application.

How does it work?

The principle is very simple: the keys of the Stäubli MCP are simulated by digital or analog inputs. A configuration file allows the assignment to each key of a signal replacing the key when the runtime license is activated. The input signals can be originated from the BIO or MIO boards, a field bus or even a Modbus TCP "software" configuration.

The "Remote MCP" software is activated when the Stäubli MCP is replaced with the plug delivered with the CS8 controller. As soon as the Stäubli MCP is reinstalled, the software is deactivated and the Stäubli MCP takes control of the arm without the need to reboot the controller, thus making maintenance operations easier.

Limitations

To use this runtime license, the customer must conform to the current standards, which must be known and implemented.
The CS8 controller's documentation provides an example of implementation and the list of the main standards to conform to.

To have a complete control of the robot, an OEM needs to recover the CS8 controller's errors. Today, it is only possible to recover messages as displayed in the CS8 controller's pop-ups, via serial line or Ethernet socket.

For maintenance operations, you may need to have a Stäubli MCP at your disposal (to adjust axes for instance).

Installation

To activate the runtime license, a password needs to be added in the options.cfx configuration file.
Note: It is also possible to activate the runtime license without password for a limited access time of two hours to allow people to test it before purchasing.

The purchase of a "remote MCP" runtime license to be used on an already delivered controller must be

managed like the purchase of a spare part. A user license with the password to be installed will be supplied to the customer.

To allow the password calculation, the CS8 controller's « MacAddress » displayed on the control panel must be given. Since « MacAddress » is a CS8 controller PC board's parameter, changing a PC board will make the password's update necessary.

VAL3 Remote Maintenance runtime license

What is "Remote Maintenance"?

"Remote Maintenance" is a PC tool delivered with Stäubli Robotics Studio. It enables the customer to view on PC the same screen as on the real MCP, to navigate in the user interface, and edit data, programs or settings. Some keys are *not* available on the PC interface, for safety reasons:

- Move/Hold button
- Enable power button
- Run / Stop buttons
- Manual movement control keys

The "Remote Maintenance" tool makes remote maintenance and debugging easier.

How does it work?

Stäubli Robotics Studio connects to the controller through the Ethernet network. A popup message is displayed when the connection starts; several simultaneous connections are possible, to the same or different controllers.

The connection between SRS and a controller is only possible if a software license is available, either on PC (dongle) or on controller:
- when a remote maintenance license is installed on a controller, any PC running SRS may connect to it, even without SRS dongle.
- when a SRS Remote Maintenance license dongle is installed on PC, this PC can connect to any controller, even if the controller has no Remote Maintenance license.

Limitations

The network connection between the PC and the controller can only be done if they belong to the same network. Going through network firewalls may be done with a phone connection between the remote PC and a local PC running SRS.

Installation

To activate the runtime license, a password needs to be added in the options.cfx configuration file.

The purchase of a "Remote Maintenance" runtime license to be used on an already delivered controller must be managed like the purchase of a spare part. A user license with the password to be installed will be supplied to the customer.

To allow the password calculation, the controller's « MacAddress » displayed on the control panel must be given. Since « MacAddress » is a controller PC board's parameter, changing a PC board will make the password's update necessary.

VAL3 oemLicence runtime license

What is the oemLicence runtime license?

The oemLicence runtime license allows an OEM or integrator to sell features of its VAL3 software by protecting them with a controller-specific license. A PC tool is delivered with SRS to let the OEM generate its own software licenses ; there is no limitation on the number of OEM licenses that can be installed on a controller.

Demonstration mode for the OEM licenses are also possible; on the emulator, the demonstration mode gives complete access to the protected features (no license is needed).

OEM licenses can also be used to make sure that the OEM VAL3 software is not reused by other parties without its agreement.

Operation

An OEM license is defined with a name (public) and a password (private) ; to enable an OEM license on a controller, the SRS VAL3 license password generator converts this private password into a controller-specific OEM license that must then be installed on the controller. The computation of the controller-specific OEM license is based on the MacAddress of the controller (unique for each CPU).

To limit the use of part of its VAL3 software, an OEM simply has to use the **getLicence**() VAL3 instruction to test if its controller-specific OEM license is installed or not. This instruction is effective only if the oemLicence runtime license is installed on the controller. To make sure that this VAL3 check is not deleted by the end user, the OEM VAL3 application should also be encrypted by using the SRS VAL3 program protection tool.

The SRS VAL3 license password generator and SRS VAL3 program protection tool are protected with the SRS VAL3 program protection license.

Limitations

The oemLicence runtime license is available with VAL3 s6.1. The SRS tools to encrypt a library and generate a VAL3 license password are delivered with SRS 6.0 and on the VAL3 CdRom; there are however not yet fully integrated in the SRS application suite:

- the encryption tool consists in the DOS executable passwordzip.exe that converts the private zip password of the OEM into a public VAL3 password.
- the license generation tool consists in the DOS executable licence.exe that computes a public, controller-specific OEM license from a private OEM password and the controller's MacAddress.
- the installation of the OEM license on the controller must be done by adding in the options.cfx configuration file the license definition:

<Licence name="*licenceName*" value="*oemLicense*"/>

Installation

The support of oem licenses is available on all controllers (s6.1+) but is protected by means of a runtime license. To activate this runtime license, a password needs to be added in the options.cfx configuration file.

The purchase of an oemLicence runtime license to be used on an already delivered controller must be managed like the purchase of a spare part. A user license with the password to be installed will be supplied to the customer.

To allow the password calculation, the controller's « MacAddress » displayed on the control panel must be given. Since « MacAddress » is a controller PC board's parameter, changing a PC board will make the password's update necessary.

VAL3 alter runtime license

What is the alter runtime license?

The alter runtime license allows to modify in real time the nominal path of the arm, to adapt it to a changing context such as conveyor position or tool wear out. It can also be used to replace the usual VAL3 motion control with a customer-specific motion control.

Operation

To be alterable, the path must be programmed with the specific alterMovej, alterMovel or alterMovec instructions. The alteration is specified as a transformation (rotation and/or translation) that can be modified every 4ms.

Two alter modes are supported: one allows to define the alter transformation relatively to the arm base (World or any other frame in World); the other allows to define the alter transformation relatively to the arm extremity (flange or any other tool on flange).

Limitations

The correct use of the alter instruction is difficult when the alter deviation is changing quickly: sensor filtering, synchronisation and desynchronization algorithms are then required to keep a good control on the arm position and velocity, without discontinuity or noise.

Some technical limitations are described in the VAL3 reference manual.

Installation

The alter software is installed on all CS8 controllers but is protected by means of a runtime license. To activate this runtime license, a password needs to be added in the options.cfx configuration file.

Note: It is also possible to activate the runtime license without password for a limited access time of two hours to allow people to test it before purchasing.

The purchase of a compliance runtime license to be used on an already delivered controller must be managed like the purchase of a spare part. A user license with the password to be installed will be supplied to the customer.

To allow the password calculation, the CS8 controller's « MacAddress » displayed on the control panel must be given. Since « MacAddress » is a CS8 controller PC board's parameter, changing a PC board will make the password's update necessary.

How to change the direction of rotation in a movel ?

There are usually several possible positions of the joint 6 (or 4 for a Scara), differing by one turn (or more), matching the same Cartesian position. In a movel, the VAL3 motion generator chooses the joint 6 target position to minimize the path, for cycle time optimisation. It is however sometimes needed to control the final position of joint 6 and therefore force the system to choose an alternate, longer path, with a different direction of rotation.

The attached move library provides a movel() program that selects the alternate direction of rotation compared to the standard VAL3 movel. It requires the motion addon for VAL3 s6.4.1+.
Its syntax is the same as for the standard movel, with as first parameter the start position of the movel, required for internal computation:
call move:movel(pStart, pTarget, tTool, mNominal)

How use Starc Encoder in Emulator (Cell SRS)

The STARC2 board supports Encoder IO board . It's possible to add 3 encoder board.

SRS doesn't managed the configuration on Cell Manager. To enable it you must modify the options.cfx on the sys/configs folder of the cell.

First enable the Start2 board with the following line.

```
<Bool name="buildStarc2" value="true" />
```

Then add the following lines for each encoder board. Today only the flag 'DUAL-ABZ-CODERS-' is possible.

Be Careful of the index attribute that represent the number of the Encoder board.

```
<starc2Slot index="0" board="DUAL-ABZ-CODERS-" />
<starc2Slot index="1" board="DUAL-ABZ-CODERS-" />
<starc2Slot index="2" board="DUAL-ABZ-CODERS-" />
```

VAL3 Compatibilities:

These options works for VAL3 versions below

Version 5: versions up to s5.5

Version6: versions up to s6.3.1

How to optimize friction compensation ?

Friction compensation can be optimized with the $setFriction addon (delivered with 'motion' addon):

bool $setFriction(num nAxis, num nFriction, num nV0, num nV1)

Friction compensation requires VAL3 s5.3.2+, and an adequate arm tuning. **Friction compensation is today NOT compatible with the compliance runtime license**.

Friction compensation works without additionnal restriction for:

- tx60-S1-R1, tx60-S1-R2
- tx90-S1-R4, tx90-S1-R5
- tx90l-S1-R2
- tx90xl-S1-R3

Friction compensation works with 2 attention points for:

- tx60l-S1-R1
- tx90-S1-R1, tx90-S1-R2
- tx90l-S1-R1
- tx90xl-S1-R1, tx90xl-S1-R2
- rx160-S1-R1, rx160-S1-R2, rx160-S1-R4, rx160-S1-R5
- rx160l-S1-R1, rx160l-S1-R3
- rx60b-S2-R5-V1, rx60b-S2-R5-V2, rx60b-S2-R5-V3
- rx60bl-S2-R4-V1
- rx90b-S2-R5-V1
- rx90bl-S2-R3-V1
- rx130b-S1-R4-V1
- rx130bl-S1-R3-V1
- rx170b-S1-R7-V1
- rx170bl-S1-R3-V1


The motion addon for these tunings must be dated july 2007 or later.


With these tunings, there must be at least 4ms between two calls of the $setFriction instruction (if not, only the last call is effective). A delay(0.004) is required after each call:

```
$setFriction(1, ...)
delay(0.004)
$setFriction(2 ...)
```

Friction compensation does NOT work for:

- rx130bxl, rx170bhp, CS8HP arms (a new tuning is required)
- tx40-S1-R1, tx40-S1-R2, tx40-S2-R1 (a new tuning is required)
- tx90xl-S1-R4 (new friction model not supported with $setFriction)
- rx160-S1-R3 (incompatiblity with compliance support), rx160-S1-R6 (new friction model not supported with $setFriction)
- rx160l-S1-R2 (incompatiblity with compliance support)