Daily Progress (Siemens R&D Internship)

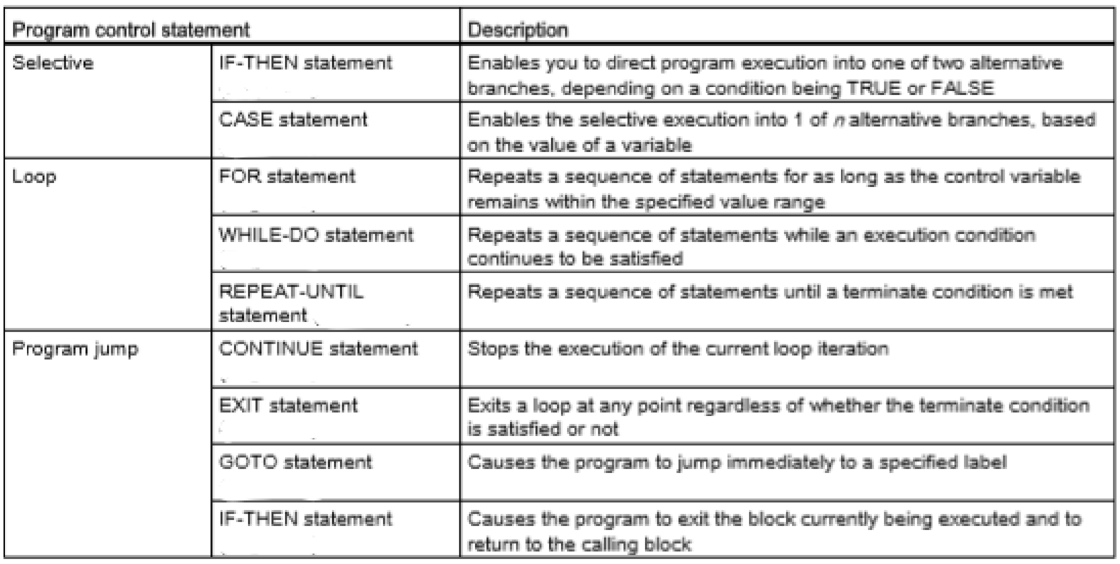
Note:

i) This file contains progress of Week 4(04/06-08/06) and Week 5(11/06-15/06).

ii) This internship has two integral parts: Theory Learning and Practical Application (TIA Portal) This document doesn't segregate the two parts but the final report of two months will track progress in terms of those parts.

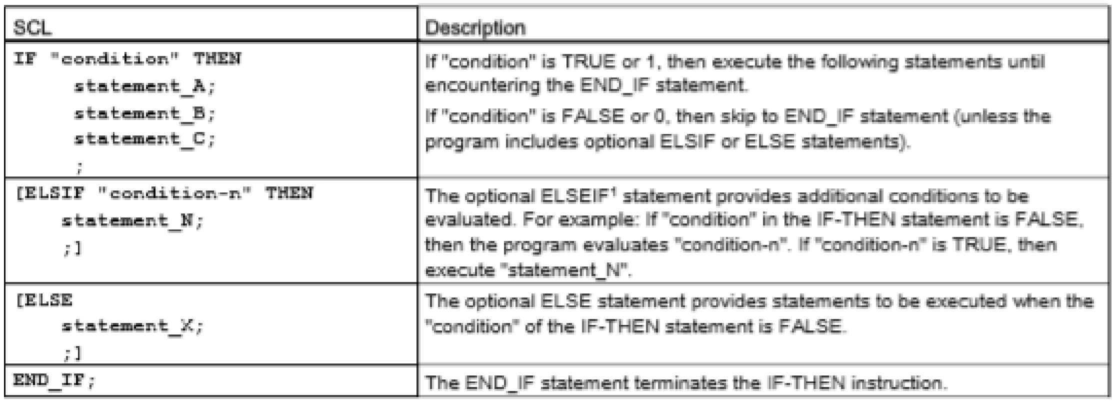
iii) This document doesn't separate day-wise learning but instead follows a concept wise approach. For example, Drives' working and PLC Theory was done over several days so it can't be segregated thus there's no division as such.

- Let's start with where we left. After covering up the basic concepts of PLC, TIA portal followed by some questions to brain storm, this week we'll get back to continue studying more advanced instructions that TIA has to offer.
- First of all, Instructions are of 3 types:-
  - Basic
  - Extended
  - Technology
- So far we have covered most of the basic instructions with application of some looping statements in some questions last week. Now let's see Program Control instructions in detail:



Program control statement		Description
Selective	IF-THEN statement	Enables you to direct program execution into one of two alternative branches, depending on a condition being TRUE or FALSE
	CASE statement	Enables the selective execution into 1 of *n* alternative branches, based on the value of a variable
Loop	FOR statement	Repeats a sequence of statements for as long as the control variable remains within the specified value range
	WHILE-DO statement	Repeats a sequence of statements while an execution condition continues to be satisfied
	REPEAT-UNTIL statement	Repeats a sequence of statements until a terminate condition is met
Program jump	CONTINUE statement	Stops the execution of the current loop iteration
	EXIT statement	Exits a loop at any point regardless of whether the terminate condition is satisfied or not
	GOTO statement	Causes the program to jump immediately to a specified label
	IF-THEN statement	Causes the program to exit the block currently being executed and to return to the calling block

IF:



SCL	Description
`IF "condition" THEN statement_A; statement_B; statement_C; ;`	If "condition" is TRUE or 1, then execute the following statements until encountering the END_IF statement. If "condition" is FALSE or 0, then skip to END_IF statement (unless the program includes optional ELSIF or ELSE statements).
`[ELSIF "condition-n" THEN statement_N; ;]`	The optional ELSEIF[1] statement provides additional conditions to be evaluated. For example: If "condition" in the IF-THEN statement is FALSE, then the program evaluates "condition-n". If "condition-n" is TRUE, then execute "statement_N".
`[ELSE statement_X; ;]`	The optional ELSE statement provides statements to be executed when the "condition" of the IF-THEN statement is FALSE.
`END_IF;`	The END_IF statement terminates the IF-THEN instruction.

CASE:

`CASE "Test_Value" OF`
`      "ValueList": Statement[; Statement, ...]`
`      "ValueList": Statement[; Statement, ...]`
`[ELSE`
`Else-statement[; Else-statement, ...]]`
`END_CASE;`
	The CASE statement executes one of several groups of statements, depending on the value of an expression.

FOR:

`FOR "control_variable" := "begin" TO "end"`
`[BY "increment"] DO`
`      statement;`
`      ;`
`END_FOR;`
	A FOR statement is used to repeat a sequence of statements as long as a control variable is within the specified range of values. The definition of a loop with FOR includes the specification of an initial and an end value. Both values must be the same type as the control variable.

WHILE:

`WHILE "condition" DO`
`      Statement;`
`      Statement;`
`      ...;`
`END_WHILE;`
	The WHILE statement performs a series of statements until a given condition is TRUE.
	You can nest WHILE loops. The END_WHILE statement refers to the last executed WHILE instruction.

REPEAT:

`REPEAT`
`      Statement;`
`      ;`
`UNTIL "condition"`
`END_REPEAT;`
	The REPEAT statement executes a group of statements until a given condition is TRUE.
	You can nest REPEAT loops. The END_REPEAT statement always refers to the last executed Repeat instruction.

CONTINUE statement:

`CONTINUE`
`      Statement;`
`      ;`
	The CONTINUE statement skips the subsequent statements of a program loop (FOR, WHILE, REPEAT) and continues the loop with the examination of whether the condition is met for termination. If this is not the case, the loop continues.

EXIT Statement:

`EXIT;`	An EXIT statement is used to exit a loop (FOR, WHILE or REPEAT) at any point, regardless of whether the terminate condition is satisfied.

GOTO label:

`GOTO JumpLabel;`
`Statement;`
` ... ;`
`JumpLabel: Statement;`
	The GOTO statement skips over statements by jumping to a label in the same block.
	The jump label ("JumpLabel") and the GOTO statement must be in the same block. The name of a jump label can only be assigned once within a block. Each jump label can be the target of several GOTO statements.

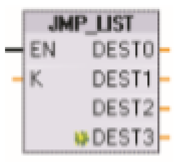
Return statement:

`RETURN;`	The Return instruction exits the code block being executed without conditions. Program execution returns to the calling block or to the operating system (when exiting an OB).

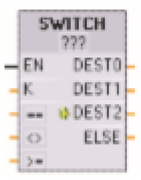
Jump and label instructions:

LAD	FBD	SCL	Description
Label_name —(JMP)—|	Label_name JMP	See the GOT [illegible] 3) statement.	If there is power flow to a JMP coil (LAD), or if the JMP box input is true (FBD), then program execution continues with the first instruction following the specified label.
Label_name —(JMPN)—|	Label_name JMPN		If there is no power flow to a JMPN coil (LAD), or if the JMPN box input is false (FBD), then program execution continues with the first instruction following the specified label.
Label_name	Label_name		Destination label for a JMP or JMPN jump instruction.

JMP_LIST instruction

LAD / FBD	SCL	Description
	CASE k OF 0: GOTO dest0; 1: GOTO dest1; 2: GOTO dest2; [n: GOTO destn;] END_CASE;	The JMP_LIST instruction acts as a program jump distributor to control the execution of program sections. Depending on the value of the K input, a jump occurs to the corresponding program label. Program execution continues with the program instructions that follow the destination jump label. If the value of the K input exceeds the number of labels - 1, then no jump occurs and processing continues with the next program network.

SWITCH instruction:

LAD / FBD	SCL	Description
	Not available	The SWITCH instruction acts as a program jump distributor to control the execution of program sections. Depending on the result of comparisons between the value of the K input and the values assigned to the specified comparison inputs, a jump occurs to the program label that corresponds to the first comparison test that is true. If none of the comparisons is true, then a jump to the label assigned to ELSE occurs. Program execution continues with the program instructions that follow the destination jump label.

Retrigger Instruction:

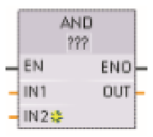
RE_TRIGR EN ENO	RE_TRIGR();	RE_TRIGR (Re-trigger scan time watchdog) is used to extend the maximum time allowed before the scan cycle watchdog timer generates an error.

STOP:

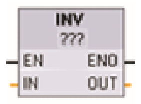
	STP();	STP (Stop scan cycle) puts the CPU in STOP mode. When the CPU is in STOP mode, the execution of your program and physical updates from the process image are stopped.

- There are some error instructions which will be covered later on. Let's move to Word Logic Instructions:

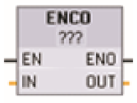
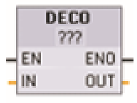
AND, OR, and XOR instruction

LAD / FBD	SCL	Description
	out := in1 AND in2;	AND: Logical AND
	out := in1 OR in2;	OR: Logical OR
	out := in1 XOR in2;	XOR: Logical exclusive OR

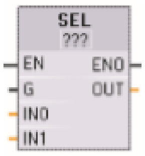
Inverter:

	Not available	Calculates the binary one's complement of the parameter IN. The one's complement is formed by inverting each bit value of the IN parameter (changing each 0 to 1 and each 1 to 0). ENO is always TRUE following the execution of this instruction.

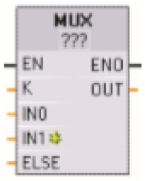
Encoder and Decoder Instructions:

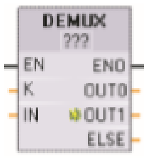
	`out := ENCO(_in_);`	Encodes a bit pattern to a binary number The ENCO instruction converts parameter IN to the binary number corresponding to the bit position of the least-significant set bit of parameter IN and returns the result to parameter OUT. If parameter IN is either 0000 0001 or 0000 0000, then a value of 0 is returned to parameter OUT. If the parameter IN value is 0000 0000, then ENO is set to FALSE.
	`out := DECO(_in_);`	Decodes a binary number to a bit pattern The DECO instruction decodes a binary number from parameter IN, by setting the corresponding bit position in parameter OUT to a 1 (all other bits are set to 0). ENO is always TRUE following execution of the DECO instruction. Note: The default data type for the DECO instruction is DWORD. In SCL, change the instruction name to DECO_BYTE or DECO_WORD to decode a byte or word value, and assign to a byte or word tag or address.

Selection:

	`out := SEL(`
	`g:=_bool_in,`
	`in0:-_variant_in,`
	`in1:=_variant_in);`
		SEL assigns one of two input values to parameter OUT, depending on the parameter G value.

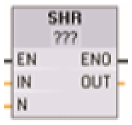
Multiplex and Demultiplex:

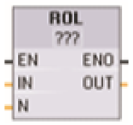
	`out := MUX(`
	`k:=_unit_in,`
	`in1:=variant_in,`
	`in2:=variant_in,`
	
	`[...in32:=variant_in,]`
	`inelse:=variant_in);`
		MUX copies one of many input values to parameter OUT, depending on the parameter K value. If the parameter K value exceeds (IN*n* - 1), then the parameter ELSE value is copied to parameter OUT.

 DEMUX
???
EN
ENO
K
OUT0
IN
OUT1
ELSE
	`DEMUX(`
	`k:=_unit_in,`
	`in:=variant_in,`
	`out1:=variant_in,`
	`out2:=variant_in,`
	
	`[...out32:=variant_in,]`
	
	`outelse:=variant_in);`
		DEMUX copies the value of the location assigned to parameter IN to one of many outputs. The value of the K parameter selects which output selected as the destination of the IN value. If the value of K is greater than the number (OUT*n* - 1) then the IN value is copied to location assigned to the ELSE parameter.

- Shift and Rotate Instructions: Difference is that there's no loss of original bits in ROL unlike Shift.

LAD / FBD	SCL	Description

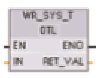
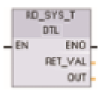
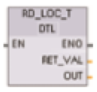
	`out := SHR(`
	`in:=_variant_in_,`
	`n:=_uint_in);`
	`out := SHL(`
	`in:=_variant_in_,`
	`n:=_uint_in);`
		Use the shift instructions (SHL and SHR) to shift the bit pattern of parameter IN. The result is assigned to parameter OUT. Parameter N specifies the number of bit positions shifted:
		• SHR: Shift bit pattern right
		• SHL: Shift bit pattern left

	`out := ROL(`
	`in:=_variant_in_,`
	`n:=_uint_in);`
	`out := ROR(`
	`in:=_variant_in_,`
	`n:=_uint_in);`
		Use the rotate instructions (ROR and ROL) to rotate the bit pattern of parameter IN. The result is assigned to parameter OUT. Parameter N defines the number of bit positions rotated.
		• ROR: Rotate bit pattern right
		• ROL: Rotate bit pattern left

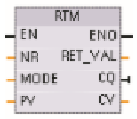
- Moving on to Extended Instructions:
    - Date and Time Instructions: The instructions to program calendar and time calculations are:
        - ➢ T_CONV converts the data type of a time value: (Time to DInt) or (DInt to Time)
        - ➢ T_ADD adds Time and DTL values: (Time + Time = Time) or (DTL + Time = DTL)

- T_SUB subtracts Time and DTL values: (Time - Time = Time) or (DTL - Time = DTL)
- T_DIFF provides the difference between two DTL values as a Time value: DTL - DTL = Time
- T_COMBINE combines a Date value and a Time_and_Date value to create a DTL value

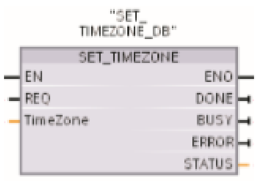
- Set and read system clock

LAD / FBD	SCL	Description
	`ret_val := WR_SYS_T( in:=_DTL_in_);`	WR_SYS_T (Write System Time) sets the CPU time of day clock with a DTL value at parameter IN. This time value does not include local time zone or daylight saving time offsets.
	`ret_val := RD_SYS_T( out=>_DTL_out);`	RD_SYS_T (Read System Time) reads the current system time from the CPU. This time value does not include local time zone or daylight saving time offsets.
	`ret_val := RD_LOC_T( out=>_DTL_out);`	RD_LOC_T (Read Local Time) provides the current local time of the CPU as a DTL data type. This time value reflects the local time zone adjusted appropriately for daylight saving time (if configured).

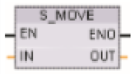
- Runtime meter Instruction

LAD / FBD	SCL	Description
	`RTM(NR:=_uint_in_, MODE:=_byte_in_, PV:=_dint_in_, CQ=>_bool_out_, CV=>_dint_out_);`	The RTM (Run Time Meter) instruction can set, start, stop, and read the run-time hour meters in the CPU.

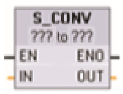
SET_TIMEZONE instruction

LAD / FBD	SCL	Description
 "SET_TIMEZONE_DB" SET_TIMEZONE EN ENO REQ DONE TimeZone BUSY ERROR STATUS	`"SET_TIMEZONE_DB"( REQ:=_bool_in, Timezone:=_struct_in, DONE=>_bool_out_, BUSY=>_bool_out_, ERROR=>_bool_out_, STATUS=>_word_out_);`	Sets the local time zone and daylight saving parameters that are used to transform the CPU system time to local time.

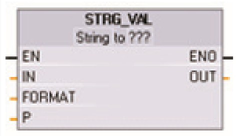
- String Instructions:
  - String Move

LAD / FBD	SCL	Description
	`out := in;`	Copy the source IN string to the OUT location. S_MOVE execution does not affect the contents of the source string.

- String Convert

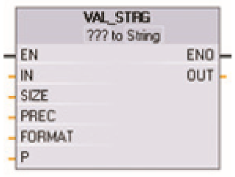
	`out := <Type>_TO_<Type>(in);`	Converts a character string to the corresponding value, or a value to the corresponding character string. The S_CONV instruction has no output formatting options. This makes the S_CONV instruction simpler, but less flexible than the STRG_VAL and VAL_STRG instructions.

- String-to-value

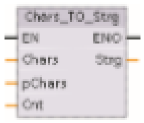
	`"STRG_VAL"( in:=_string_in, format:=_word_in, p:=uint_in, out=>_variant_out);`	Converts a number character string to the corresponding integer or floating point representation.

-

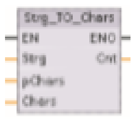
- Value-to-string

 VAL_STRG
??? to String
EN
ENO
IN
OUT
SIZE
PREC
FORMAT
P
	`"VAL_STRG"(`
	`in:=_variant_in,`
	`size:=_usint_in,`
	`prec:=_usint_in,`
	`format:=_word_in,`
	`p:=uint_in,`
	`out=>_string_out);`
		Converts an integer, unsigned integer, or floating point value to the corresponding character string representation.

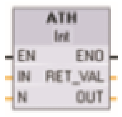
- Chars-string

 Chars_TO_Strg
EN
ENO
Chars
Strg
pChars
Cnt
	`Chars_TO_Strg(`
	`Chars:=_variant_in_,`
	`pChars:=_dint_in_,`
	`Cnt:=_uint_in_,`
	`Strg=>_string_out_);`
		All or part of an array of characters is copied to a string.
		The output string must be declared before Chars_TO_Strg is executed. The string is then overwritten by the Chars_TO_Strg operation.
		Strings of all supported maximum lengths (1..254) may be used.
		The string maximum length value is not changed by Chars_TO_Strg operation. Copying from array to string stops when the maximum string length is reached.
		A nul character '$00' or 16#00 value in the character array works as a delimiter and ends copying of characters into the string.

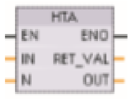
- String-chars

 Strg_TO_Chars
EN
ENO
Strg
Cnt
pChars
Chars
	`Strg_TO_Chars(`
	`Strg:=_string_in_,`
	`pChars:=_dint_in_,`
	`Cnt=>_uint_out_,`
	`Chars:=_variant_inout_);`
		The complete input string Strg is copied to an array of characters at IN_OUT parameter Chars.
		The operation overwrites bytes starting at array element number specified by the pChars parameter.
		Strings of all supported max lengths (1..254) may be used.
		An end delimiter is not written; this is your responsibility. To set an end delimiter just after the last written array character, use the next array element number [pChars+Cnt].

- ASCII-Hex

	`ret_val := ATH(`
	`in:=_variant_in_,`
	`n:=_int_in_,`
	`out=>_variant_out_);`
		Converts ASCII characters into packed hexadecimal digits.

- Hex-ASCII

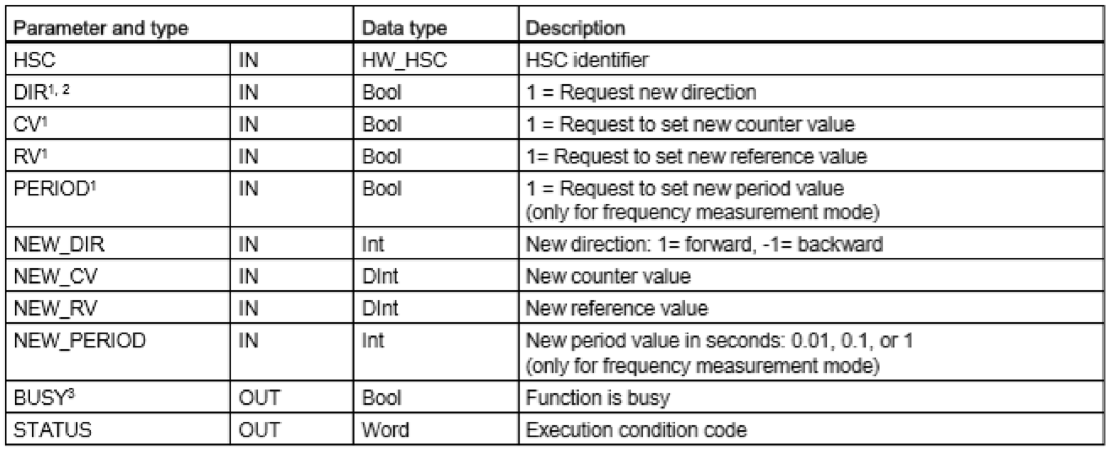
	`ret_val := HTA(`
	`in:=_variant_in_,`
	`n:=_uint_in_,`
	`out=>_variant_out_);`
		Converts packed hexadecimal digits to their corresponding ASCII character bytes.

- String Operations
    1. LEN: LEN (length) provides the current length of the string IN at output OUT. An empty string has a length of zero.
    2. CONCAT (concatenate strings): It joins string parameters IN1 and IN2 to form one string provided at OUT. After concatenation, String IN1 is the left part and String IN2 is the right part of the combined string.
    3. LEFT (Left substring): It provides a substring made of the first L characters of string parameter IN. If L is greater than the current length of the IN string, then the entire IN string is returned in OUT. If an empty string is the input, then an empty string is returned in OUT.
    4. MID (Middle substring): It provides the middle part of a string. The middle substring is L characters long and starts at character position P (inclusive). If the sum of L and P exceeds the current length of the string parameter IN, then a substring is returned that starts at character position P and continues to the end of the IN string.
    5. RIGHT (Right substring): It provides the last L characters of a string. If L is greater than the current length of the IN string, then the entire IN string is returned in

parameter OUT. If an empty string is the input, then an empty string is returned in OUT.

6. DELETE: Deletes L characters from string IN. Character deletion starts at character position P (inclusive), and the remaining substring is provided at parameter OUT. If L is equal to zero, then the input string is returned in OUT. If the sum of L and P is greater than the length of the input string, then the string is deleted to the end.
7. INSERT: Inserts string IN2 into string IN1. Insertion begins after the character at position P.
8. REPLACE: Replaces L characters in the string parameter IN1. Replacement starts at string IN1 character position P (inclusive), with replacement characters coming from the string parameter IN2.
9. FIND: Provides the character position of the substring specified by IN2 within the string IN1. The search starts on the left. The character position of the first occurrence of IN2 string is returned at OUT. If the string IN2 is not found in the string IN1, then zero is returned.

- There are a lot of extended instructions left yet but let's put that on hold and give a look to most used instructions in real industry based solutions i.e. Technology Instructions:
  i. Counting
  ii. PID Control
  iii. Motion Control
- **Counting**: Having already studied normal counters, we move to HSC (High Speed Counters) now which are used in industries for high speed operations. The high-speed counter (HSC) counts events that occur faster than the OB execution rate. If the events to be counted occur within the execution rate of the OB, one can use CTU, CTD, or CTUD counter instructions. If the events occur faster than the OB execution rate, then use the HSC.
- The CTRL_HSC instruction allows the user program to programmatically change some of the HSC parameters.



Parameter and type		Data type	Description
HSC	IN	HW_HSC	HSC identifier
DIR[1, 2]	IN	Bool	1 = Request new direction
CV[1]	IN	Bool	1 = Request to set new counter value
RV[1]	IN	Bool	1= Request to set new reference value
PERIOD[1]	IN	Bool	1 = Request to set new period value (only for frequency measurement mode)
NEW_DIR	IN	Int	New direction: 1= forward, -1= backward
NEW_CV	IN	DInt	New counter value
NEW_RV	IN	DInt	New reference value
NEW_PERIOD	IN	Int	New period value in seconds: 0.01, 0.1, or 1 (only for frequency measurement mode)
BUSY[3]	OUT	Bool	Function is busy
STATUS	OUT	Word	Execution condition code

If the following Boolean flag values are set to 1 when the CTRL_HSC instruction is executed, the corresponding NEW_xxx value is loaded to the counter:

- DIR = 1 is a request to load a NEW_DIR value, 0 = no change
- CV = 1 is a request to load a NEW_CV value, 0 = no change
- RV = 1 is a request to load a NEW_RV value, 0 = no change
- PERIOD = 1 is a request to load a NEW_PERIOD value, 0 = no change

The current count value is not available in the CTRL_HSC parameters. The process image address that stores the current count value is assigned during the hardware configuration of the high-speed counter. One may use program logic to directly read the count value. The value returned to the program will be a correct count for the instant in which the counter was read. The counter

will continue to count high-speed events. Therefore, the actual count value could change before the program completes a process using an old count value.
In the case of an error, ENO is set to 0, and the STATUS output contains a condition code.

STATUS	Description
0	No error
80A1	HSC identifier does not address a HSC
80B1	Illegal value in NEW_DIR
80B2	Illegal value in NEW_CV
80B3	Illegal value in NEW_RV
80B4	Illegal value in NEW_PERIOD
80C0	Multiple access to the high-speed counter
80D0	High-speed counter (HSC) not enabled in CPU hardware configuration.

For example: Let's use the HSC as an input for an incremental shaft encoder. The shaft encoder provides a specified number of counts per revolution and a reset pulse that occurs once per revolution. The clock(s) and the reset pulse from the shaft encoder provide the inputs to the HSC. The HSC is loaded with the first of several presets, and the outputs are activated for the time period where the current count is less than the current preset. The HSC provides an interrupt when the current count is equal to preset, when reset occurs, and also when there is a direction change. As each current-count-value-equals-preset-value interrupt event occurs, a new preset is loaded and the next state for the outputs is set. When the reset interrupt event occurs, the first preset and the first output states are set, and the cycle is repeated. Since the interrupts occur at a much lower rate than the counting rate of the HSC, precise control of high-speed operations can be implemented with relatively minor impact to the scan cycle of the CPU. The method of interrupt attachment allows each load of a new preset to be performed in a separate interrupt routine for easy state control. (Alternatively, all interrupt events can be processed in a single interrupt routine.)

All HSCs function the same way for the same counter mode of operation. There are four basic types of HSC:

- Single-phase counter with internal direction control
- Single-phase counter with external direction control
- Two-phase counter with 2 clock inputs
- A/B phase quadrature counter

One can use each HSC type with or without a reset input. When one activates the reset input (with some restrictions, see the following table), the current value is cleared and held clear until one deactivates the reset input.

- Frequency function: Some HSC modes allow the HSC to be configured (Type of counting) to report the frequency instead of a current count of pulses. Three different frequency measuring periods are available: 0.01, 0.1, or 1.0 seconds. The frequency measuring period determines how often the HSC calculates and reports a new frequency value. The reported frequency is an average value determined by the total number of counts in the last measuring period. If the frequency is rapidly changing, the reported value will be an intermediate between the highest and lowest frequency occurring during the measuring period. The frequency is always reported in Hertz (pulses per second) regardless of the frequency-measuring-period setting.

- Counter modes and inputs: The following table shows the inputs used for the clock, direction control, and reset functions associated with the HSC. The same input cannot be used for two different functions, but any input not being used by the present mode of its HSC can be used for another purpose. For example, if HSC1 is in a mode that uses built-in inputs but does not use the external reset (I0.3), then I0.3 can be used for edge interrupts or for HSC2.

Counting modes summed in table:

Type	Input 1	Input 2	Input 3	Function
Single-phase counter with internal direction control	Clock	(Optional: direction)	-	Count or frequency
			Reset	Count
Single-phase counter with external direction control	Clock	Direction	-	Count or frequency
			Reset	Count
Two-phase counter with 2 clock inputs	Clock up	Clock down	-	Count or frequency
			Reset	Count
A/B-phase quadrature counter	Phase A	Phase B	-	Count or frequency
			Reset[1]	Count

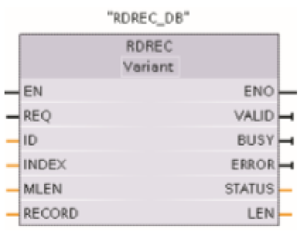
- Configuration of the HSC:
  The CPU allows to configure up to 6 high-speed counters. Edit the "Properties" of the CPU to configure the parameters of each individual HSC. Use the CTRL_HSC instruction in the user program to control the operation of the HSC. Enable the specific HSC by selecting the "Enable" option for that HSC. After enabling the HSC, configure the other parameters, such as counter function, initial values, reset options and interrupt events.

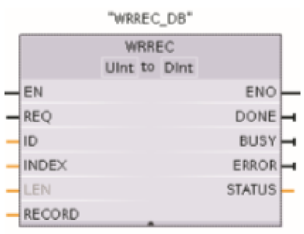
Let's start again the extended instructions where we had left earlier.

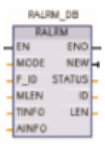
Distributed I/O Instructions: The following Distributed I/O instructions can be used with PROFINET, PROFIBUS, or AS-i:

- RDREC instruction: You can read a data record with the number INDEX from a module or device.
- WRREC instruction: You can transfer a data record with the number INDEX to a module or device defined by ID.
- RALRM instruction: You can receive an interrupt with all corresponding information from a module or device and supply this information to its output parameters.
- DPRD_DAT instruction: You must read consistent data areas greater than 64 bytes from a module or device with the DPRD_DAT instruction.
- DPWR_DAT instruction: You must write consistent data areas greater than 64 bytes to a module or device with the DPWR_DAT instruction.
- The DPNRM_DG instruction can only be used with PROFIBUS. You can read the current diagnostic data of a DP slave in the format specified by EN 50 170 Volume 2, PROFIBUS.

LAD / FBD	SCL	Description

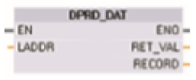
 "RDREC_DB"
RDREC
Variant
EN
ENO
REQ
VALID
ID
BUSY
INDEX
ERROR
MLEN
STATUS
RECORD
LEN
	`"RDREC_DB"(`
	`req:=_bool_in_,`
	`ID:=_word_in_,`
	`index:=_dint_in_,`
	`mlen:=_uint_in_,`
	`valid=>_bool_out_,`
	`busy=>_bool_out_,`
	`error=>_bool_out_,`
	`status=>_dword_out_,`
	`len=>_uint_out_,`
	`record:=_variant_inout_);`
		Use the RDREC instruction to read a data record with the number INDEX from the component addressed by the ID, such as a central rack or a distributed component (PROFIBUS DP or PROFINET IO). Assign the maximum number of bytes to read in MLEN. The selected length of the target area RECORD should have at least the length of MLEN bytes.

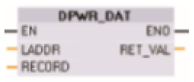
 "WRREC_DB"
WRREC
UInt to DInt
EN
ENO
REQ
DONE
ID
BUSY
INDEX
ERROR
LEN
STATUS
RECORD
	`"WRREC_DB"(`
	`req:=_bool_in_,`
	`ID:=_word_in_,`
	`index:=_dint_in_,`
	`len:=_uint_in_,`
	`done=>_bool_out_,`
	`busy=>_bool_out_,`
	`error=>_bool_out_,`
	`status=>_dword_out_,`
	`record:=_variant_inout_);`
		Use the WRREC instruction to transfer a data RECORD with the record number INDEX to a DP slave/PROFINET IO device component addressed by ID, such as a module in the central rack or a distributed component (PROFIBUS DP or PROFINET IO).
		Assign the byte length of the data record to be transmitted. The selected length of the source area RECORD should, therefore, have at least the length of LEN bytes.

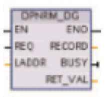
 RALRM_DB
RALRM
EN
ENO
MODE
NEW
F_ID
STATUS
MLEN
ID
TINFO
LEN
AINFO
	`"RALRM_DB"(`
	`mode:=_int_in_,`
	`f_ID:=_word_in_,`
	`mlen:=_uint_in_,`
	`new=>_bool_out_,`
	`status=>_dword_out_,`
	`ID=>_word_out_,`
	`len=>_uint_out_,`
	`tinfo:=_variant_inout_,`
	`ainfo:=_variant_inout_);`
		Use the RALRM (read alarm) instruction to read diagnostic interrupt information from PROFIBUS or PROFINET I/O modules/devices.
		The information in the output parameters contains the start information of the called OB as well as information of the interrupt source.
		Call RALRM in an interrupt OB to return information regarding the event(s) that caused the interrupt. In the S7-1200, only diagnostic interrupts (OB82) are supported.

RALRM instruction operating modes

MODE	Description
0	• ID contains the hardware identifier of the I/O module that triggered the interrupt. • Output parameter NEW is set to TRUE. • LEN produces an output of 0. • AINFO and TINFO are not updated with any information.
1	• ID contains the hardware identifier of the I/O module that triggered the interrupt. • Output parameter NEW is set to TRUE. • LEN produces an output of the amount in bytes of AINFO data that is returned. • AINFO and TINFO are updated with interrupt-related information.
2	If the hardware identifier assigned to input parameter F_ID has triggered the interrupt then: • ID contains the hardware identifier of the I/O module that triggered the interrupt. Should be the same as the value at F_ID. • Output parameter NEW is set to TRUE. • LEN produces an output of the amount in bytes of AINFO data that is returned. • AINFO and TINFO are updated with interrupt-related information.

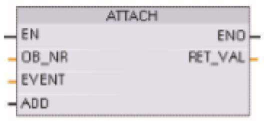
	`ret_val := DPRD_DAT(`
	`laddr:=_word_in_,`
	`record=>_variant_out_);`
		Use the DPRD_DAT instruction to read the consistent data of a DP standard slave/PROFINET IO device. If no errors occur during the data transfer, the data read is entered into the target area set up by the RECORD parameter. The target area must have the same length as you configured with STEP 7 for the selected module. When you call the DPRD_DAT instruction, you can only access the data of one module / DP identification under the configured start address.

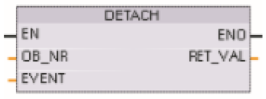
	`ret_val := DPWR_DAT(`
	`laddr:=_word_in_,`
	`record:=_variant_in_);`
		Use the DPWR_DAT instruction to transfer the data in RECORD consistently to the addressed DP standard slave/PROFINET IO device. The source area must have the same length as you configured with STEP 7 for the selected module.

 DPNRM_DG
EN
ENO
REQ
RECORD
LADDR
BUSY
RET_VAL
	`ret_val := DPNRM_DG(`
	`req:=_bool_in_,`
	`laddr:=_word_in_,`
	`record=>_variant_out_,`
	`busy=>_bool_out_);`
		Use the DPNRM_DG instruction to read the current diagnostic data of a DP slave in the format specified by EN 50 170 Volume 2, PROFIBUS. The data that has been read is entered in the destination area indicated by RECORD following error-free data transfer.

Interrupts:

1. ATTACH and DETACH Instructions:

	`ret_val := ATTACH(`
	`ob_nr:=_int_in_,`
	`event:=_event_att_in_,`
	`add:=_bool_in_);`
		ATTACH enables interrupt OB subprogram execution for a hardware interrupt event.

	`ret_val := DETACH(`
	`ob_nr:=_int_in_,`
	`event:=_event_att_ in);`
		DETACH disables interrupt OB subprogram execution for a hardware interrupt event.

Hardware interrupt events:

The following hardware interrupt events are supported by the CPU:

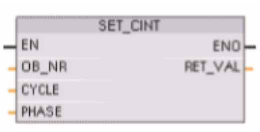
- Rising edge events (all built-in CPU digital inputs and SB digital inputs) – A rising edge occurs when the digital input transitions from OFF to ON as a response to a change in the signal from a field device connected to the input.
- Falling edge events (all built-in CPU digital inputs and SB digital inputs) – A falling edge occurs when the digital input transitions from ON to OFF.
- High-speed counter (HSC) current value = reference value (CV = RV) events (HSC 1 through 6) – A CV = RV interrupt for a HSC is generated when the current count transitions from an adjacent value to the value that exactly matches a reference value that was previously established.
- HSC direction changed events (HSC 1 through 6) – A direction changed event occurs when the HSC is detected to change from increasing to decreasing, or from decreasing to increasing.
- HSC external reset events (HSC 1 through 6) – Certain HSC modes allow the assignment of a digital input as an external reset that is used to reset the HSC count value to zero. An external reset event occurs for such a HSC, when this input transitions from OFF to ON.

❖ Enabling hardware interrupt events in the device configuration:
Hardware interrupts must be enabled during the device configuration. One must check the enable-event box in the device configuration for a digital input channel or a HSC, if one wants to attach this event during configuration or run time.

OB_NR parameter: All existing hardware-interrupt OB names appear in the device configuration "HW interrupt:" drop-down list and in the ATTACH / DETACH parameter OB_NR drop-list.

EVENT parameter: When a hardware interrupt event is enabled, a unique default event name is assigned to this particular event. One can change this event name by editing the "Event name:" edit box, but it must be a unique name. These event names become tag names in the "Constants" tag table, and appear on the EVENT parameter drop-down list for the ATTACH and DETACH instruction boxes. The value of the tag is an internal number used to identify the event.

2. CYCLIC Interrupts:
   - Set Cyclic Interrupt:

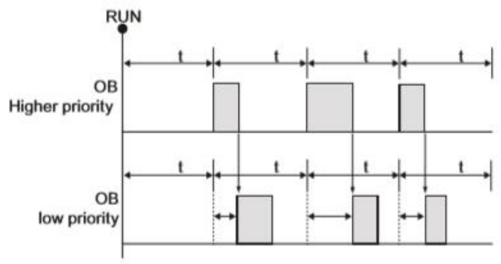
	`ret_val := SET_CINT(`
	`ob_nr:=_int_in_,`
	`cycle:=_udint_in_,`
	`phase:=_udint_in_);`
		Set the specified interrupt OB to begin cyclic execution that interrupts the program scan.

Time parameter examples:

- If the CYCLE time = 100 us, then the interrupt OB referenced by OB_NR interrupts the cyclic program scan every 100 us. The interrupt OB executes and then returns execution control to the program scan, at the point of interruption.
- If the CYCLE time = 0, then the interrupt event is deactivated and the interrupt OB is not executed.
- The PHASE (phase shift) time is a specified delay time that occurs before the CYCLE time interval begins. You can use the phase shift to control the execution timing of lower priority OBs.

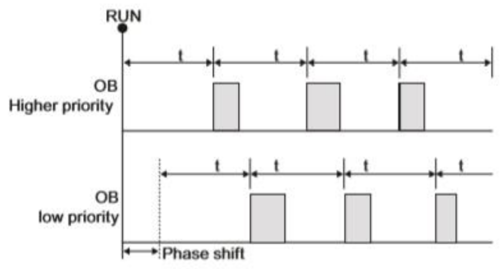
If lower and higher priority OBs are called in the same time interval, the lower priority OB is only called after the higher priority OB has finished processing. The execution start time for the low priority OB can shift depending on the processing time of higher priority OBs.

OB call without phase shift

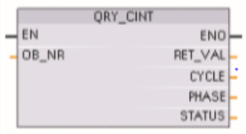


If you want to start the execution of a lower priority OB on a fixed time cycle, then phase shift time should be greater then the processing time of higher priority OBs.

OB-call with phase shift



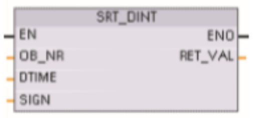
- QRY_CINT (Query cyclic interrupt):

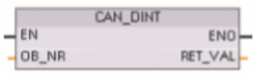
	`ret_val := QRY_CINT(`
	`ob_nr:=_int_in_,`
	`cycle=>_udint_out_,`
	`phase=>_udint_out__,`
	`status=>_word_out_);`
		Get parameter and execution status from a cyclic interrupt OB. The values that are returned existed at the time QRY_CINT was executed.

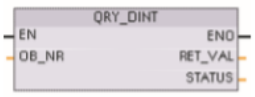
3. Time delay interrupts:

SRT_DINT, CAN_DINT, and QRY_DINT instructions

LAD / FBD	SCL	Description

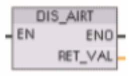
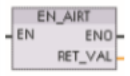
	`ret_val := SRT_DINT(`
	`ob_nr:=_int_in_,`
	`dtime:=_time_in_,`
	`sign:=_word_in_);`
		SRT_DINT starts a time delay interrupt that executes an OB when the delay time specified by parameter DTIME has elapsed.

	`ret_val := CAN_DINT(`
	`ob_nr:=_int_in_);`
		CAN_DINT cancels a time delay interrupt that has already started. The time delay interrupt OB is not executed in this case.

	`ret_val := QRY_DINT(`
	`ob_nr:=_int_in_,`
	`status=>_word_out_);`
		QRY_DINT queries the status of the time delay interrupt specified by the OB_NR parameter.

4. Asynchronous event interrupts:
Use the DIS_AIRT and EN_AIRT instructions to disable and enable alarm interrupt processing.

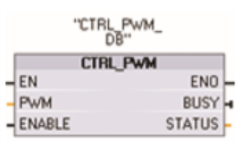
DIS_AIRT and EN_AIRT instructions

LAD / FBD	SCL	Description
	`DIS_AIRT();`	DIS_AIRT delays the processing of new interrupt events. You can execute DIS_AIRT more than once in an OB.
	`EN_AIRT();`	EN_AIRT enables the processing of interrupt events that you previously disabled with the DIS_AIRT instruction. Each DIS_AIRT execution must be cancelled by an EN_AIRT execution. The EN_AIRT executions must occur within the same OB, or any FC or FB called from the same OB, before interrupts are enabled again for this OB.

PULSE:-

1. CTRL_PWM instruction

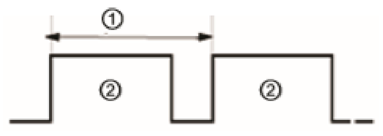
LAD / FBD	SCL	Description

 "CTRL_PWM_DB"
CTRL_PWM
EN
ENO
PWM
BUSY
ENABLE
STATUS
	`"CTRL_PWM_DB"(`
	`PWM:=_word_in_,`
	`enable:=_bool_in_,`
	`busy=>_bool_out_,`
	`status=>_word_out_);`
		Provides a fixed cycle time output with a variable duty cycle. The PWM output runs continuously after being started at the specified frequency (cycle time). The pulse width is varied as required to affect the desired control.

The CTRL_PWM instruction stores the parameter information in the DB. The data block parameters are not separately changed by the user, but are controlled by the CTRL_PWM instruction. Specify the enabled pulse generator to use, by using its tag name for the PWM parameter. When the EN input is TRUE, the PWM_CTRL instruction starts or stops the identified PWM based on the value at the ENABLE input. Pulse width is specified by the value in the associated Q word output address. Because the CPU processes the request when the CTRL_PWM instruction is executed, parameter BUSY will always report FALSE. If an error is detected, then ENO is set to FALSE, and parameter STATUS contains a condition code.

The pulse width will be set to the initial value configured in device configuration when the CPU first enters RUN mode. One can write values to the Q-word location specified in device configuration ("Output addresses" / "Start address :") as needed to change the pulse width. One can use an instruction such as a move, convert, math, or PID box to write the desired pulse width to the appropriate Q word. One must use the valid range for the Q-word value (percent, thousandths, ten-thousandths, or S7 analog format).

2. Operation of the pulse outputs



① Cycle time

② Pulse width

Pulse width can be expressed as hundredths of the cycle time (0 to 100), as thousandths (0 to 1000), as ten thousandths (0 to 10000), or as S7 analog format.

The pulse width can vary from 0 (no pulse, always off) to full scale (no pulse, always on).

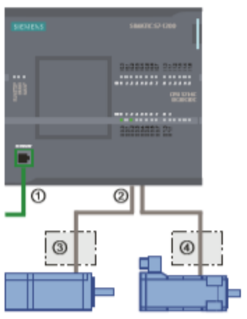
Since the PWM output can be varied from 0 to full scale, it provides a digital output that in many ways is the same as an analog output. For example, the PWM output can be used to control the speed of a motor from stop to full speed, or it can be used to control position of a valve from closed to fully opened. Two pulse generators are available for controlling high-speed pulse output functions: PWM and Pulse train output (PTO). PTO is used by the motion control instructions. One can assign each pulse generator to either PWM or PTO, but not both at the same time. The two pulse generators are mapped to specific digital outputs as shown in the following table. One can use on board CPU outputs, or you can use the optional signal board outputs. The output point numbers are shown in the following table (assuming the default output configuration). If one has changed the output point numbering, then the output point numbers will be those one assigned. Regardless, PTO1/PWM1 uses the first two digital outputs, and PTO2/PWM2 uses the next two digital outputs, either on the CPU or on the attached signal board. Note that PWM requires only one output, while PTO can optionally use two outputs per channel. If an output is not required for a pulse function, it is available for other uses.

3. Configuring a pulse channel for PWM

To prepare for PWM operation, first configure a pulse channel in the device configuration by selecting the CPU, then Pulse Generator (PTO/PWM), and choose either PWM1 or PWM2. Enable the pulse generator (check box). If a pulse generator is enabled, a unique default name is assigned to this particular pulse generator. One can change this name by editing it in the "Name:" edit box, but it must be a unique name. Names of enabled pulse generators will become tags in the "constant" tag table, and will be available for use as the PWM parameter of the CTRL_PWM instruction.

This much extended instructions for now. Let's move to Motion Control now:-

- **Motion Control**: It's an industry term to describe a range of applications that involve movement with varying degrees of precision. There are Motion Control applications which require an object to be just moved with limited concern for acceleration, deceleration or speed of motion. On the other hand there are applications which require precise coordination of all kinds of motion including high coordination of multiple simultaneous motions.
- The CPU provides motion control functionality for the operation of stepper motors and servo motors with pulse interface. The motion control functionality takes over the control and monitoring of the drives.
  - The "Axis" technology object configures the mechanical drive data, drive interface, dynamic parameters, and other drive properties.
  - Configure the pulse and direction outputs of the CPU for controlling the drive.
  - User program uses the motion control instructions to control the axis and to initiate motion tasks.
  - Use the PROFINET interface to establish the online connection between the CPU and the programming device. In addition to the online functions of the CPU, additional commissioning and diagnostic functions are available for motion control.



① PROFINET

② Pulse and direction outputs

③ Power section for stepper motor

④ Power section for servo motor

The DC/DC/DC variants of the CPU S7-1200 have onboard outputs for direct control of drives. The relay variants of the CPU require the signal board with DC outputs for drive control.

- Before we study about Motion Control there's a need to study about servo motors and stepper motors. So let's dive into detailing of motors.

Servo Motors

A servomotor is a rotary actuator or linear actuator that allows for precise control of angular or linear position, velocity and acceleration. It consists of a suitable motor coupled to a sensor for position feedback. It also requires a relatively sophisticated controller, often a dedicated module designed specifically for use with servomotors.
Servomotors are not a specific class of motor although the term servomotor is often used to refer to a motor suitable for use in a closed-loop control system.

Fig: Steering actuator of a large robot vehicle.

Industrial servomotor: The grey/green cylinder is the brush-type DC motor. The black section at the bottom contains the planetary reduction gear, and the black object on top of the motor is the optical rotary encoder for position feedback.

Servomotors are used in applications such as robotics, CNC machinery or automated manufacturing.
A servomotor is a closed-loop servomechanism that uses position feedback to control its motion and final position. The input to its control is a signal (either analogue or digital) representing the position commanded for the output shaft.

The motor is paired with some type of encoder to provide position and speed feedback. In the simplest case, only the position is measured. The measured position of the output is compared to the command position, the external input to the controller. If the output position differs from that required, an error signal is generated which then causes the motor to rotate in either direction, as needed to bring the output shaft to the appropriate position. As the positions approach, the error signal reduces to zero and the motor stops.

The very simplest servomotors use position-only sensing via a potentiometer and bang-bang control of their motor; the motor always rotates at full speed (or is stopped). This type of servomotor is not widely used in industrial motion control, but it forms the basis of the simple and cheap servos used for radio-controlled models.

More sophisticated servomotors use optical rotary encoders to measure the speed of the output shaft and a variable-speed drive to control the motor speed. Both of these enhancements, usually in combination with a PID control algorithm, allow the servomotor to be brought to its commanded position more quickly and more precisely, with less overshooting.

Stepper Motors

A stepper motor or step motor or stepping motor is a brushless DC electric motor that divides a full rotation into a number of equal steps. The motor's position can then be commanded to move and hold at one of these steps without any position sensor for feedback (an open-loop controller), as long as the motor is carefully sized to the application in respect to torque and speed.
Switched reluctance motors are very large stepping motors with a reduced pole count, and generally are closed-loop commutated.

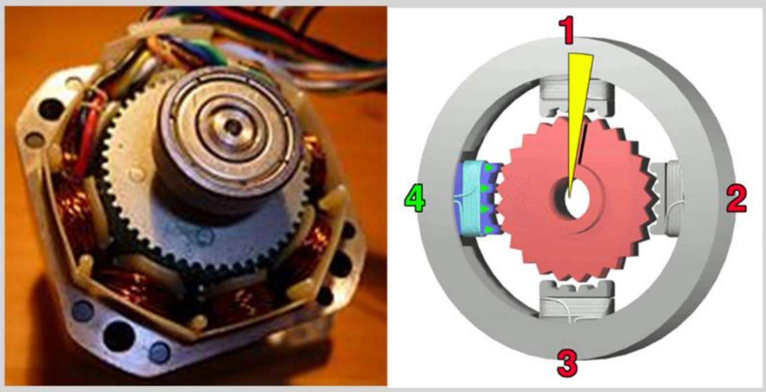


Fig: Stepper Motor

In second figure:-

1) The top electromagnet (1) is turned on, attracting the nearest teeth of the gear-shaped iron rotor. With the teeth aligned to electromagnet 1, they will be slightly offset from right electromagnet (2).

2) The top electromagnet (1) is turned off, and the right electromagnet (2) is energized, pulling the teeth into alignment with it. This results in a rotation of 3.6° in this example.
3) The bottom electromagnet (3) is energized; another 3.6° rotation occurs.
4) The left electromagnet (4) is energized, rotating again by 3.6°. When the top electromagnet (1) is again enabled, the rotor will have rotated by one tooth position; since there are 25 teeth, it will take 100 steps to make a full rotation in this example.

Fundamentals of operation

Brushed DC motors rotate continuously when DC voltage is applied to their terminals. The stepper motor is known by its property to convert a train of input pulses (typically square wave pulses) into a precisely defined increment in the shaft position. Each pulse moves the shaft through a fixed angle.

Stepper motors effectively have multiple "toothed" electromagnets arranged around a central gear-shaped piece of iron. The electromagnets are energized by an external driver circuit or a micro controller. To make the motor shaft turn, first, one electromagnet is given power, which magnetically attracts the gear's teeth. When the gear's teeth are aligned to the first electromagnet, they are slightly offset from the next electromagnet. This means that when the next electromagnet is turned on and the first is turned off, the gear rotates slightly to align with the next one. From there the process is repeated. Each of those rotations is called a "step", with an integer number of steps making a full rotation. In that way, the motor can be turned by a precise angle.

The circular arrangement of electromagnets is divided into groups, each group called a phase, and there is an equal number of electromagnets per group. The number of groups is chosen by the designer of the stepper motor. The electromagnets of each group are interleaved with the electromagnets of other groups to form a uniform pattern of arrangement. For example, if the stepper motor has two groups identified as A or B, and ten electromagnets in total, then the grouping pattern would be ABABABABAB.

Electromagnets within the same group are all energized together. Because of this, stepper motors with more phases typically have more wires (or leads) to control the motor.

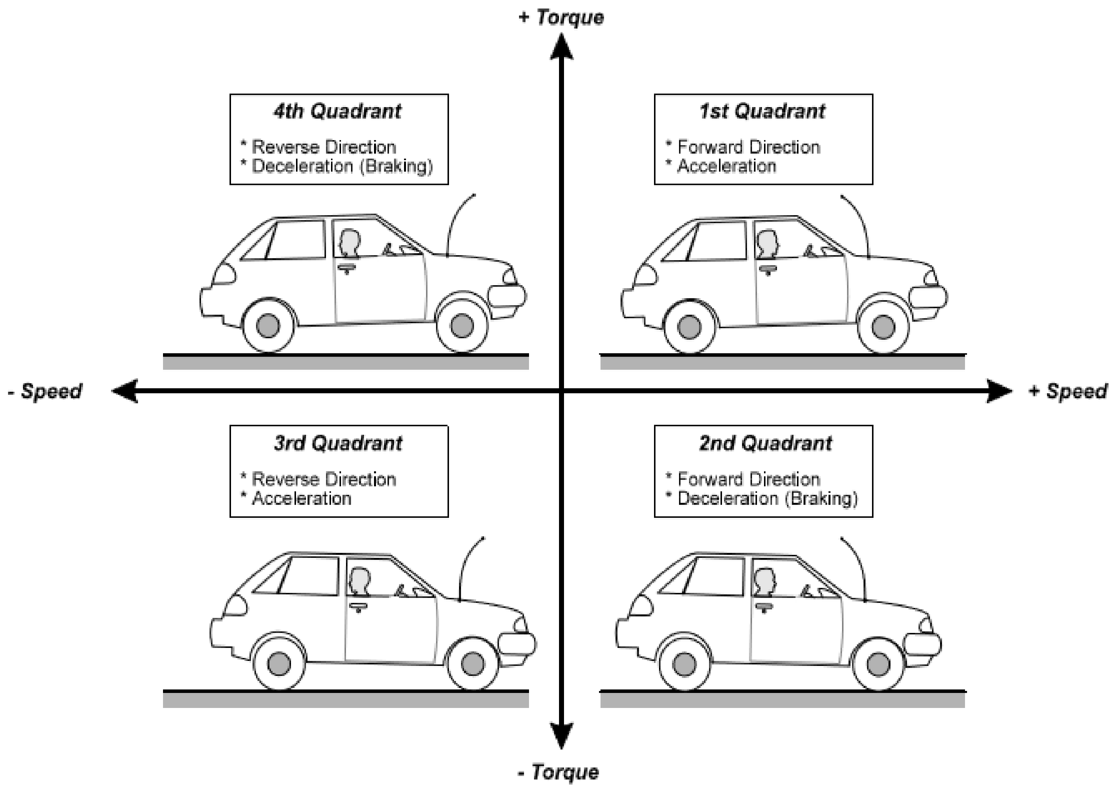
Servomotors vs Stepper motors

Servomotors are generally used as a high-performance alternative to the stepper motor. Stepper motors have some inherent ability to control position, as they have built-in output steps. This often allows them to be used as an open-loop position control, without any feedback encoder, as their drive signal specifies the number of steps of movement to rotate, but for this the controller needs to 'know' the position of the stepper motor on power up. Therefore, on first power up, the controller will have to activate the stepper motor and turn it to a known position, e.g. until it activates an end limit switch. This can be observed when switching on an inkjet printer; the controller will move the ink jet carrier to the extreme left and right to establish the end positions. A servomotor will immediately turn to whatever angle the controller instructs it to, regardless of the initial position at power up.

The lack of feedback of a stepper motor limits its performance, as the stepper motor can only drive a load that is well within its capacity, otherwise missed steps under load may lead to positioning errors and the system may have to be restarted or recalibrated. The encoder and controller of a servomotor are an additional cost, but they optimise the performance of the overall system (for all of speed, power and accuracy) relative to the capacity of the basic motor. With larger systems, where a powerful motor represents an increasing proportion of the system cost, servomotors have the advantage.

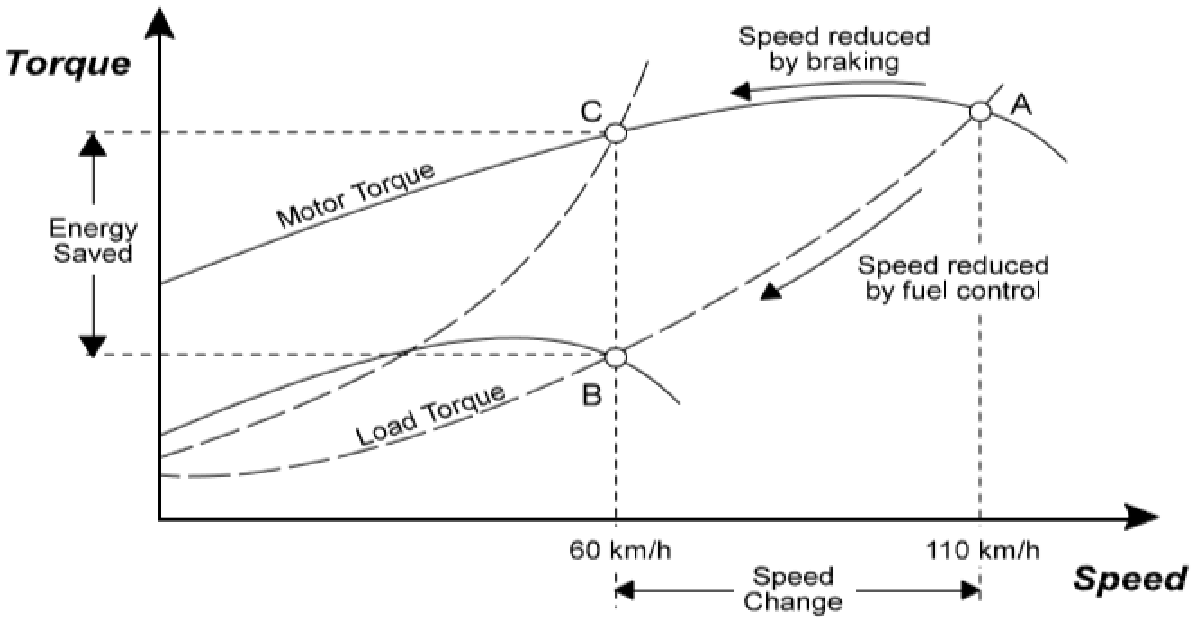
There has been increasing popularity in closed loop stepper motors in recent years. They act like servomotors but have some differences in their software control to get smooth motion. The main benefit of a closed loop stepper motor is it's relatively low cost. There is also no need to tune the PID controller on a closed loop stepper system.
Many applications, such as laser cutting machines, may be offered in two ranges, the low-priced range using stepper motors and the high-performance range using servomotors.

- Let's have a look on Variable Speed Drives now as it forms a basis of good chunk of knowledge we are going to gather. They are used to:
    - ➢ Match the speed of a drive to the process requirements
    - ➢ Match the torque of a drive to the process requirements
    - ➢ Save energy and improve efficiency



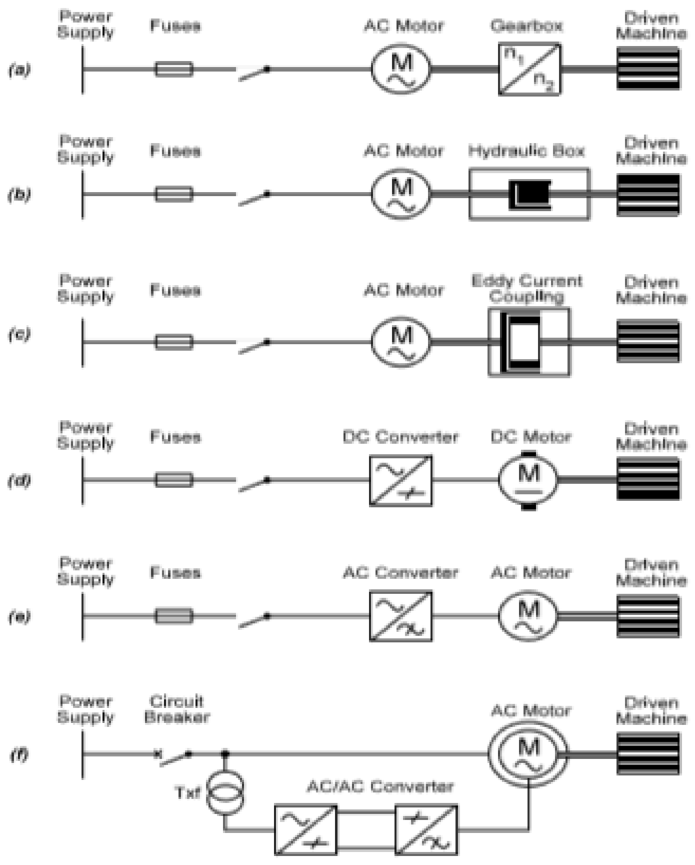
- Torque–speed curves can be used to compare two alternative methods of speed control and to illustrate the differences in energy consumption between the two strategies:
    - ✓ Speed controlled by using drive control: adjusting the torque of the prime mover. In practice, this is done by adjusting the fuel supplied to the engine, using the accelerator for control, without using the brake. This is analogous to using an electric variable speed drive to control the flow of water through a centrifugal pump.
    - ✓ Speed controlled by using load control: adjusting the overall torque of the load. In practice, this could be done by keeping a fixed accelerator setting and using the brakes for speed control. This is analogous to controlling the water flow through a centrifugal pump by throttling the fluid upstream of the pump to increase the head.

- Using the motorcar as an example, the two solid curves in Figure below represent the drive torque output of the engine over the speed range for two fuel control conditions:



- High fuel position – accelerator full down
- Lower fuel position – accelerator partially down

- The two dashed curves in the Figure above represent the load torque changes over the speed range for two mechanical load conditions. The mechanical load is mainly due to the wind resistance and road friction, with the restraining torque of the brakes added.
  - Wind & friction plus brake ON – high load torque
  - Wind & friction plus brake OFF – low load torque
- Types of VSDs:



a) Typical mechanical VSD with an AC motor as the prime mover;
b) Typical hydraulic VSD with an AC motor as the prime mover;
c) Typical electromagnetic coupling or Eddy Current coupling;
d) Typical electrical VSD with a DC motor and DC voltage converter;
e) Typical electrical VSD with an AC motor and AC frequency converter;
f) Typical slip energy recovery system or static Kramer system;

- So we ended with implementing motion control in TIA partially so it won't be covered in this week's document. And we started off Drives Theory in much more detail which would be still continuing over the course of next week.