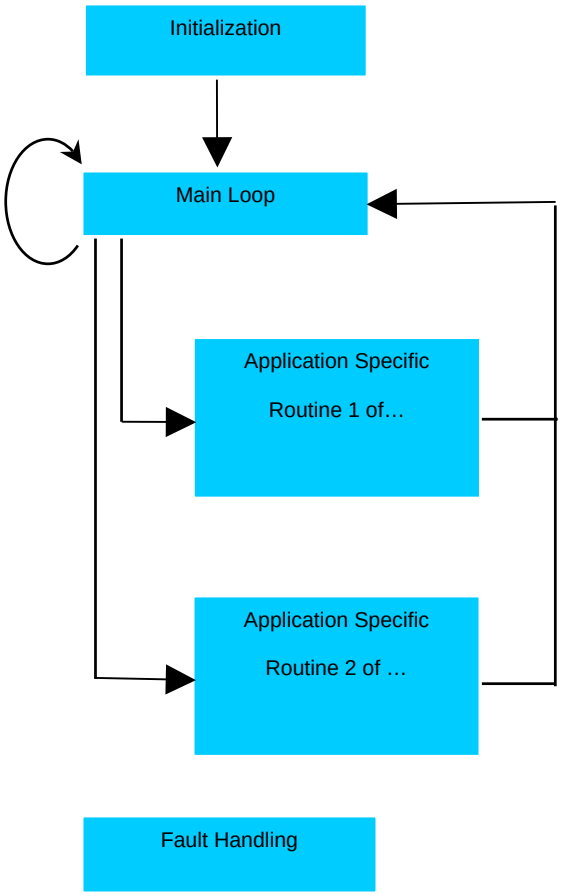
Subject: Recommended SMC Programming Style

Product: All SMC Controllers

Summary: This document discusses the best implementation for an application program in an SMC controller. It highlights and explains some subtle details that mean everything to the robust operation of the machine.

The SMC controller runs an interpreted, structured text program. The structure is not strictly enforced; the program is a simple text file with a list of instructions. Yaskawa encourages providing structure to the programs to increase readability and organization. Programs that are not well structured are more difficult to debug and maintain. Yaskawa recommends the following program template layout because it is suitable for any application.



Although most of the examples in the manual are quite short and simple, many applications require much more comprehensive level of programming. Feel free to include blank lines between routines, and use the TAB key to indent, making the program more readable. YTerm compresses the blank lines and tab characters out of the program because the controller will not accept them. Remember this, because if a program is uploaded from the controller, it will have lost its formatting. YTerm v7.0 and up have an auto format function.

Initialization

Programs should start by initializing any special parameters and user variables that will be used later in the program. Remember that variables must be initialized before use, or a command error will result later in the program if the variable appears on the right side of an equal sign, or as a command's parameter. The initialization section contains code that never needs to run again after the initial power up. Programmers may choose to incorporate a homing routine near the end of the initialization, just before entering the main loop of the program.

Sometimes programmers will not want certain variables to be initialized at power up, because they contain job-specific values. Variables can be saved to flash memory by using the BV command within the program.

Did You Know?

A handy trick for variables that must retain their values after power cycle is to include their initialization ABOVE the #AUTO label. These variables will only get initialized if the "XQ" command is sent over the serial port or any Ethernet handle. This is because the XQ command with no argument causes the program to execute at the first line. In contrast, when the #AUTO label is included in the program, execution starts at the label and continues from there, causing the controller to skip commands above it.

The following is an example of an initialization section:

```
#AUTO; InitPass=0    //   Auto execute at power up, and init success flag set to false.
//   ------------------   Wizard Shell Code Initialization Section   -------------------
    MOX;  OP0    //  Make sure motor is off and outputs are reset.
    Error=0           //  Initialize Application Wizard error code system
    LimIgnor=0    //  Initialize flag for limit switches during homing.
    TRUE=1;  FALSE=0
    NT=_TM/1024   //   Normalized Time, only required if servo update changed from 1000
uSec default.
// -------------- Homing Initialization  -----------------
    homing=0         //   1 When homing, 0 when not
    homed=0          //   1 When homed,0 When not homed
    H_SW=-1;  h_sw=-1    //  Active state of the home input  All caps is original config,
small is program switchable.
//   ------------------   Rotary Table Initialization    ----------------------------
    DM Accel[9]
    DM ApprDist[9]
    DM ApprOP[9]
```

```
    DM Dwell1[9]
    DM Position[9]
    DM PostDWOp[9]
    DM Scurve[9]
    DM Speed[9]
    DM Station[9]
    DM StatOP[9]
    DM Modulus[3], OPosO[3], DL72[3];  i=0        //  For the rotary calculations
    #InitMod; Modulus[i]=0; OPosO[i]=0; DL72[i]=0;  i=i+1; JP #InitMod,i<3
    OldIndex=0; MachCyc=40960.0
    Bit=5; JS #XtoY; BitMaskO=XtoY
    MinBit=5; MaxBit=8; JS #MakeMsk; BitMask=MakeMsk
    WT 500/NT
    SHX; WT500/NT
    InitPass=1      //  Initialization success flag, for special label usage
```

The highlighted commands are of additional importance. The User variable "InitPass" serves as a flag to indicate that the program has successfully made it through all of the initialization code. This is critical for the proper operation of the fault handling routines that may run whenever their conditions are met. The behaviors of the special labels are described later.

Main Loop

The main loop serves as the control center for all general machine logic. This is typically a small part of the overall program. It is more efficient to include mode-specific logic in lower level routines. Mode-specific logic includes operations that are only active during a specific operation mode, such as manual or auto. This allows for faster execution of the main loop logic, and easier code debugging. The SMC runs an interpreter, which means that the text code of the program is always being translated to machine language command by command. Minimizing the amount of structured text code generally increases efficiency.

The following is an example of a small main loop:

```
#MAIN
    JS #HOME,((@IN[4]=1)&(homed=0))
    Index=(_TI&BitMask)/BitMaskO           //  Determine (by input) what Station to locate
    JS #ROTARY,(Index<=9) & (Index<>OldIndex) & (homed=TRUE)
JP #MAIN
```

Notice that the labels listed here show the modes of operation, and includes the logic that makes then execute.

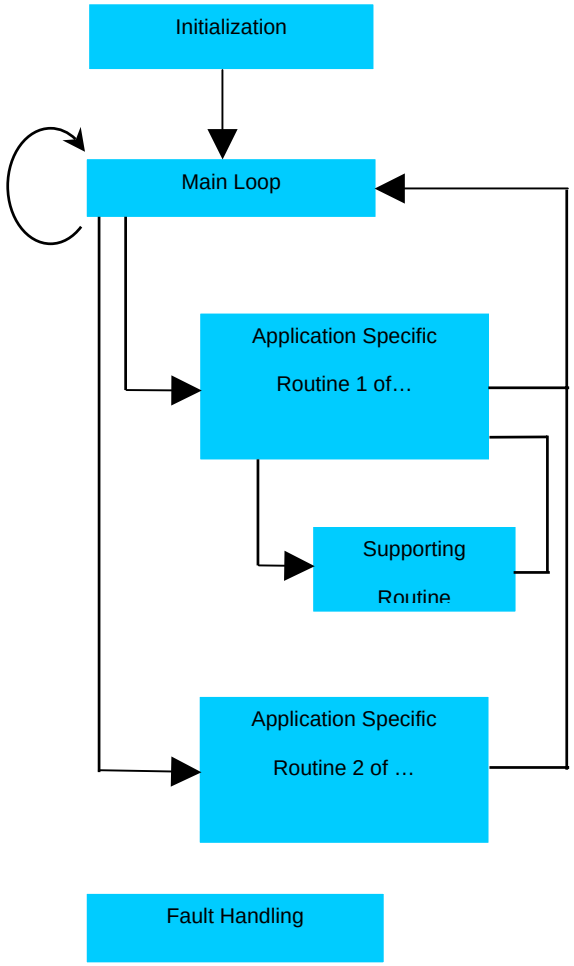
Application Specific Routines

Application specific routines contain the core logic of the machine's modes of operation. The routine only runs when the logic in the main loop has determined that all conditions have been met.

The following is an example of an application specific routine that indexes the servo:

```
NOTE:  "Automatic Index Subroutine"
#AUTOCYC
     Status=sAuto
     IF (_MOX=1); SHX; WT 1000; ENDIF
#INDEX
     JP #INDEX, ((MODE & IndexGo)=0) & ((MODE & AutoMode)=AutoMode)
     JP #AUTOERR,_RPX<>XHOME
     ACX=IdxDist/2/(IdxTime/3)/(IdxTime/3)+3300; DCX=_ACX
     SPX=IdxDist/2/(IdxTime/3)
     PRX=IdxDist; BGX; Parts=Parts+1; AMX
     SB INDEX1; WT Dwell; CB INDEX1
     #AUTO_DN
EN

#AUTOERR
     Status=sNotHome
     ATHOME=FALSE
     JP #AUTO_DN
EN
```

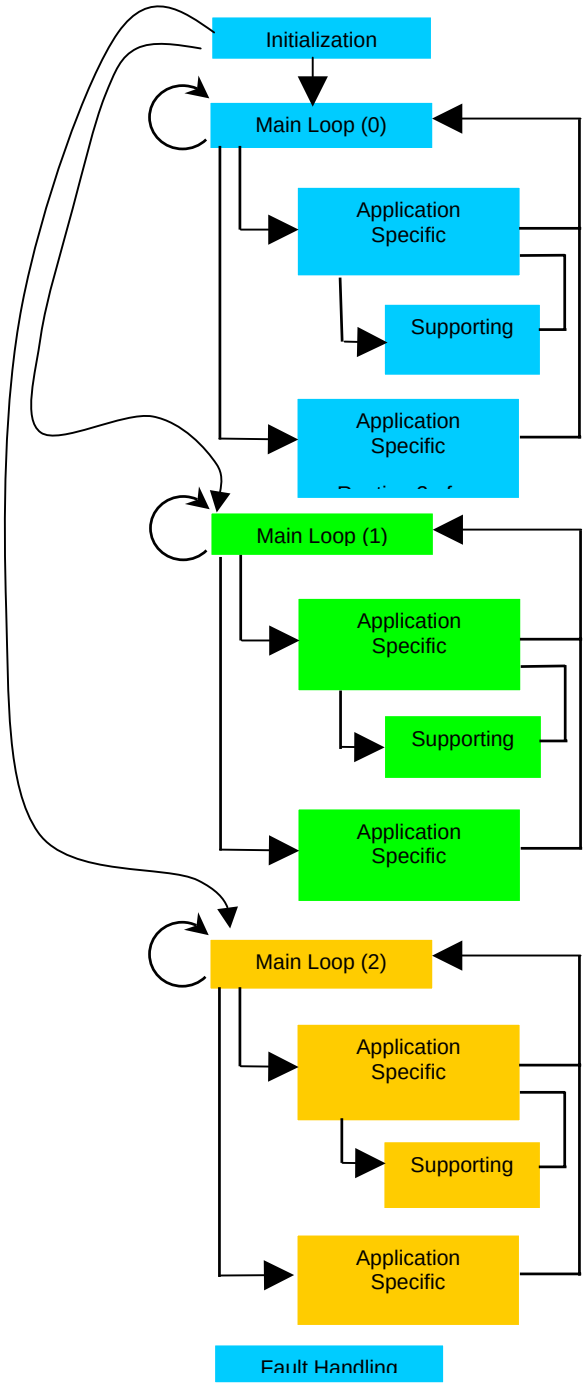


As shown above, supporting routines or functions of an application specific routine can be included below the routine itself, such as #AUTOERR.

Notice the way in which this supporting routine is called. It allows the original Application Specific subroutine to "EN" normally even if a diversion in the code path occurred. The #AUTO_DN label aides this technique.

Multitasking

Below is a block diagram demonstrating the concept of multitasking. Threads can be started and stopped by each other at will. The only limitation is that the thread being launched with the XQ command cannot be already executing. The recommended method is to start threads in the initialization section of the program. Once started, they will continue to operate until halted. The controller will time share between threads on a line-by-line basis. If one program has more commands per line, it will receive more of the total execution time available.



Multitasking is a very useful method of accomplishing several tasks that have very little or nothing do with each other. Focus remains specifically on the given task, whereas extra "juggling" code would be necessary to adequately handle several unrelated events in a single program loop. Good examples of multitasking include:

- Software limit updates for two axes that operate in each other's zone where a collision could occur
- An analog output that must be updated continuously
- Short-term event that must be monitored at the same time as

another process (this is a “spur” thread that runs and dies after a certain event is attained, such as a timer)
- I/O logic that must be solved at a regular interval
- Events that are related, but handled at different times, such as a registration buffer that must store product positions, but the correction for a particular product will occur slightly later

Watch Out!

Subroutines and other sections of code should not be intermixed between threads to avoid complexity and possible erratic behavior.

Did You Know?

There is no need to multitask when moving two or more servos if these axes always move in a predefined sequence. Some users incorrectly assume that they must use a separate thread per axis. Experience will dictate when it is better to use multitasking, and when one task will suffice.

Did You Know?

All variables & arrays defined in the program are global, meaning that all threads can read and write the data. This is very useful when threads must share information, such as a product registration buffer. This can be troubling if the programmer unknowingly uses the same variable in both threads for something simple, such as the variable “X” as a counter.

Special Label & Fault Handling Routines

Fault Handling routines are the special labels that automatically execute without being referred to in the program with a JP or JS. They are the following:

#AUTO	Auto program execute at power up or reset.
#ININT	Input interrupt for any combination of local inputs
#CMDERR	Bad or illogical command handler
#POSERR	Excessive position error
#LIMSWI	Limit switch (Soft or Hard)
#TCPERR	Ethernet TCP Error
#MCTIME	Motion complete timeout

The #CMDERR, #POSERR, and #LIMSWI routines are highly recommended for all applications to greatly increase their robustness. It is very important to integrate Special Label routines with care by thinking about the program logic required to make the machine run properly. The following are some DOs and DON'Ts:

Recommended	NOT Recommended
Abort motion for safety if a #CMDERR occurs. Always use AB1.	Don't return to the program using RE if a #CMDERR occurs. Notify the operator, and make the code restart by jumping to #AUTO, or simply use the AB, which requires power cycle to restart.
Under #CMDERR, include a method for the error code (_TC) and line number (_ED) to be identified by someone who can help debug the program. Store them in a variable and present them on an operator screen.	Avoid calling subroutines from within the special label routine. You should only use the special label event handlers to redirect the code and get the controller back on its feet. Keep the routines simple and short. If the special label routine causes another fault, it can cause a recursive condition that cannot be recovered from without power cycle.
Check for software and hardware limits for each axis in the #LIMSWI routine.	Don't set the software limits FL and BL to the same value; the servo will not be able to move once it crosses that position. Check the Stop Code (SC)
Keep the fault handlers short and simple. Do not perform motion from within them. Instead, re-direct the code to subroutines that can correct the problem or allow the machine to resume successfully. The system is at risk for compounded problems caused by trying to run elaborate code from within the fault routines. Just use them to trap errors, and signal the main parts of the program to handle the recovery.	
	Don't use the ZS command to correct programming problems with the JS and JP commands. The best use of ZS is to close out the special label event, otherwise, it is bad programming practice.

Other considerations

- There is always a chance that special label routines could execute just after power up, but before the initialization section has completed. If any of the special routines jump to the #MAIN loop after recovery, problems could occur because of commands that were "skipped" in the initialization. To avoid this, use a flag such as the one in the above examples called "InitPass," and check it's value when deciding where to resume after a special label event. It is possible that the servo could have following error when the program starts, and the program will immediately jump to #POSERR. Once the following error is cleared, the #POSERR must check the initialization flag to decide whether to jump back to the top of the program or to the #MAIN loop.

- If a Command Error occurs in the #CMDERR routine, #CMDERR will recursively call itself (to indicate the additional command errors), and cause a stack overflow, stopping execution of the application program. Be extra cautious when programming in the #CMDERR routine to minimize this possibility. If you suspect a #CMDERR in the #CMDERR routine, it can be debugged by placing an MG command as the first command within the routine so you can see the message print out at the terminal screen of Yterm each time an error occurs.

- If a thread other than thread zero causes a command error, this can be determined in code by using the _ED1 parameter. If another thread has an error, it is possible to restart it, although it will eventually require a program fix. Use the HX and XQ commands within the #CMDERR routine to restart the troubled thread at it's main label. Usually it's too complex to resume after a serious fault. Remember that all special label routines run as subroutines of thread zero, so in this case, an RE1 command will be necessary for thread zero to resume.

Example Fault handling Routines

```
#CMDERR                                          A
     JP #LIMSWI,_TC=22                           B
     JP #PROBLEM,_TC<>30; RE1                    C
     #PROBLEM                                    D
     AB1;  CB DREMEL                             E
     STATUS=CMDERRSL                             F
// MG "Error "{N};  TC1{N}                       G
// MG " on line",_ED{F3.0}                       H
     #CWAIT                                      I
        WT250/NT                                 J
JP#CWAIT,(vbCMD&vbRESET)=vbRESET                 K
```

```
    vbCMD=vbIDLE;  ZS;  IIESTOP,,,$80   L
JP#AUTO                                 M
```

In the above routine, there is first a check to see if the fault code is a limit switch (line B). The SMC controllers have an odd behavior that treats the limit switch as either a limit fault, or command error based on when the limit was exceeded. If the limit is already exceeded when a BG command executes, then a command fault occurs. Adding a jump to #LIMSWI is a handy way to redirect the issue, making all limit switch faults handled the same way no matter when they occur.

Secondly, there is another nuisance fault code that may come up when doing vector motion. If a sequence is zero length, meaning there would be no motion, the controller issues the command error "Sequence Segment is too short." Since this otherwise causes no problem in the controller, it is safe to ignore the fault, and let the code resume (line C). There is an IF ELSE statement set up (line C & D); the controller skips the RE statement and jumps to #PROBLEM for any other fault. So if 30 is the fault, #CMDERR exits via the RE1 statement, otherwise, there is more to be done.

Next, motion is aborted with the AB1 command. An output named DREMEL is turned off (line E).

Because this example program is controlled by a Visual Basic application running on a PC, the status of the machine is stored in a user-defined variable called STATUS (line F). Because the program is now in the #CMDERR routine, the STATUS variable is changed to reflect this in the PC. The PC application can then notify the user of the problem.

The commented-out code is handy during debug. These lines produce a nicely formatted message on the terminal screen of Y-Term. (Lines G &H)

Next, the program simply waits in a loop for the Visual Basic application to change the "vbCMD" variable to acknowledge the fault (lines I,J,K.) The SMC will then prepare to exit the #CMDERR routine (lines L & M.) The subroutine

stack is cleared, making possible a successful jump back to the start of the program. This is effectively the same as a restart of the controller.

#POSERR	// This is an automatically executing subroutine if position error is exceeded at any time.
AB1; L=0; MODE=0	
#ERRLP2	
OP$FF; WT500/NT; OP0; WT500/NT	// Toggle all outputs on and off every 500mSec.
L=L+1	// Increment counter
JP#ERRLP2,L<15	// Repeat 15 times
AI-3; AI3	// Wait for rising edge of input 3
SH; MO	// SH command clears out following error.
ZS	// Clear subroutine stack, this also resets the fault routine.
JP#MAIN,OK=1	// Jump back to the main loop only if the OK flag is 1, meaning the initialization has completed
JP#AUTO	// Jump back to the start of the program
EN	

If any axis exceeds the following error limit (absolute value of TE > ER) during any servo update cycle AND this label is included in the program, program execution will automatically jump to #POSERR. This includes following error for slaves on SMC3010 distributed control topologies, because the slave status is provided to the master at the slave update interval.

The only thing the controller firmware will do automatically is disable the servo, but ONLY of the OE command is set to 1. It's the programmer's responsibility to make sure the controller can recover from the fault. Yaskawa recommends using the AB1 command to abort the motion profiler. Some machines may not tolerate an abrupt halt of the servo caused by AB1. For those machines that require an abort, but with gentler deceleration, use the ST and DC command. It's true that the deceleration parameter can't be changed while performing a point to point move, but the controller will allow the deceleration to be changed AFTER the ST has been issued. Setting the deceleration to a quick but gentle value is usually best on a large machine, with high inertia or a high mechanical gear ratio.

This example program toggles all outputs every ½ second. Any method that alerts the operator will work. The program will then wait for the operator to reset the fault by toggling input 3.

The next part is very important. Remember that the following error is measured every servo cycle, all the time. If the following error is not reduced / eliminated before exiting the fault routine, #POSERR will immediately execute again. In this example, the SH and MO commands clear the following error. (SH enables and clears following error, the MO command used because we don't want the motor enabled just yet.) The final command, ZS, is the official exit of the fault routine. All code after it is considered normal, or external to the fault routine. This can be verified by checking the TB command.