Engineering Solutions

Chuck Raskin P.E. E.E. MSCS
Blaine Minnesota 55434
Cell: 612-207-5285   <ChuckR@EngSolu.com>

Sept. 2, 2005

True Low Cost Stepper Stall Detection

If you have ever used, or designed a system that uses stepper motors, you know that the hardest thing to do is to design it in a way to guarantee that the motor will never stall.

Steppers seem to have a mind of their own. About the time you have the system stable enough to operate properly, someone changes the requirements, the system goes outside the steppers speed torque curve, or, the system develops a vibration that feeds back to the stepper armature and it stalls!

Motor Characteristics

One characteristic of a servo motor is that any sudden change in motor loading will not cause adverse consequences in its operation. True, there will be a definite change in motor current, and a possible change in its following error, and also true is the fact that the motor might speed up or slow down, but, it will keep on running, assuming the load change is still within the torque range of the servo.

A stepper motor, on the other hand, operates best with smooth load transitions. That is, there should be no drastic discontinuities in command frequency or loading. For those of you who don’t know how a stepper motor works, the stepper motor is a zero following error device. That simply means the acceleration, deceleration, slew velocity, loading, and other operating conditions, must all remain within the stepper’s immediate speed torque loading capability, or it will stall. Sudden changes are not allowed!

Although a stepper motor can offer attractive benefits such as hardware simplicity, low cost, ease of operation, open loop capability, it comes with a price. That price is a lengthy list of considerations that can each affect its accuracy, stability, and overall operation of the unit. When you are ready to incorporate a stepper system, you must not to take any of its operating requirements and attributes for granted. It cannot be emphasized enough . . . The one thing a stepper does best is **stall**!

Stall Issues

Mechanical:

If you’ve read any articles on stepper motor operation, or, the paper I wrote titled; “ Using Stepper Motors” (posted on my website under ‘Stepper Design Considerations’) , you’ll know that steppers are true positional devices. That is to say, stepper armatures do not operate by gliding from one angular position to another like a servo, but are forced from one magnetic pole ‘lock’ position to another by injecting command currents into their windings to counteract the holding force between the armature and stator magnets.

Knowing that a stepper motor has both armature and field magnetic poles, it become apparent

that its motion is inherently unstable. To show this instability, mount a stepper motor to a surface.
Next, attach a DC Analog tachometer to the motors shaft and secure the tach to the same surface as the stepper. Attach an Oscilloscope to the tachometer wire to view the voltage produced when the stepper motor shaft is rotated slowly by hand. At the completion of any move, there will be some noticeable 'ringing' (oscillation) riding on the DC level. This is the armature mechanically oscillating. The position that the motor wants to stop at is called the Detent position. In a typical stepper motor, there are 200 detent positions in one revolution of the armature. These are referred to as Full-Step positions.

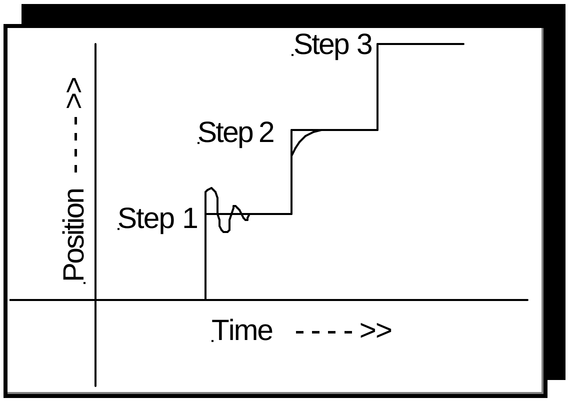


Figure 1

When a stepper motor position step is made, the steppers total inertia (armature + load) will force the armature to overshoot the magnetic detent position, and oscillate (or ring) about the new position point it is trying to stop at.. Figure-1 Step-1, shows this action. This can be attributed to the fact that the motor's magnetic field(s) do not allow the two mechanical masses (stator and armature) to come to rest in a perfect trajectory when moving laterally to each other. Since there is no position feedback with Open Loop control, there will always be some lateral positional overshoot with stepper motion. Full-Step's require more torque to break away from the motors internal magnet attraction and rotate to the next detent position. The more torque required to make a step, the larger the mechanical oscillation will be.

Even when the stepper motor armature is rotating, the steppers velocity profile can show this oscillation. To minimize the oscillation reduce the step size to Half-Step or Micro-Step. You can also reduce the current required to make the step move, or dampen the system mechanically or electrically. Figure-1 Step-2, shows this action.

It is important to note, that the motion of the stepper armature physically reverses as it oscillates into position. The degree of reversal may not be large, but if an electrical step pulse, or mechanical impulse is applied at the point when the motor is 'moving' in the reversed direction, it can actually begin rotating, without stalling, in the opposite direction to what the control is commanding it to do! The hardest part to understand about this phenomenon is that you'll see it, but you won't know what just happened!

Electrical Ringing:
The phase coils that create the motion also form a tuned circuit with the motors internal capacitance and any other stray capacitance in the circuitry. This will create electrical ringing on the step waveforms. Damping circuitry is required to either reduce or eliminate this condition. Severe electrical ringing can cause interference with other electronic equipment including the controlling computer, which can have serious side effects on other parts of the system.

Mechanical Resonance in certain Frequency Ranges:
There can be one or more natural resonant frequencies or harmonics associated with any

stepper motor. When operating at these frequencies, a heavy oscillation will be heard and/or felt from the stepper motor. This can have detrimental consequences on the motors ability to operate, and the overall systems ability to function. To eliminate this problem, either refrain from running at these frequencies, use micro-stepping techniques, or do low current stepping in these regions.

Possible Step Misses (Slip) Without Stalling:
Since all motors and load systems exhibit inertia, it is possible for a stepper motor to fall out of, and then back into, synchronization with its controller (translator). If this happens, the position of the system will be corrupted. To prevent this, reduce the accel/decel rates, velocity, or increase the motor to system torque ratio. In other words, get a bigger motor!

Another possible solution is to use a sine wave amplifier in place of PWM amplifier. But regardless . . .

Never allow a stepper motor to miss steps since stalling is just around the corner!

Monitoring for Stall

With a reasonable understanding of how stepper motors work, we can assume that if you are not monitoring the motors mechanical motion, you really don't know if the stepper has problems even though you are monitoring its winding currents (remembering that mechanical impulses at the right moment can make the motor go in the opposite direction to what is being commanded . . . not good!).

Since stepper motors can be running in any step per revolution mode, from very slow to very high speed conditions, and either clockwise or counter clockwise rotation, is there really no way to determine if a mechanical or electrical 'shock wave' has caused the motor to stall, slip, or reverse direction?

Lets analyze this . . .steppers have several motion conditions:

1. Stopped
2. Rotating . . . CW or CCW, at any speed, with any step/rev assignment as commanded
3. Stalled . . . . Moving in one position only (oscillating)
4. Stalled . . . . Moving but with phase or frequency slip
5. Stalled . . . . Moving in an opposite to commanded direction.

Let's analyze these one at a time.

Stopped:
Stopped is reasonably easy in Full or Half step modes because armature phase currents are generally ON or OFF. But more than not, the motor is being operated by a PWM stepper drive meaning the armature currents are most likely in motion (being switched). Unless your hardware and software are in direct control of the armature current, it would be almost impossible for an external current monitor to 'know' if the motor has stalled by waveform analysis. In addition, waveform analysis takes CPU time, which might not be able to keep up with current alterations.

Rotating:

As stated above Current monitoring might not be a good possibility. What about an encoder? Encoders will work, but some problems with attaching an encoder might be:

1. If the encoder is attached to the motor, stall vibrations might wreck the encoder
2. The size of the encoder might not fit into the area where the motor is being mounted
3. The computer running the system might not have encoder reading capability (PLC)
4. CPU time might not be able to respond to motor slip
5. Adding position analysis software might not be practical
6. Cost might be prohibitive
7. The idea in the first place was to be open loop!

Stalled Conditions (Items 3, 4, & 5):

If currents are low, or the load is heavy, the motor might audibly ‘ring’. Monitoring motor BackEMF might not be possible in a PWM system if the system was not being controlled directly from the users hardware and software. And even if it were, there are so many possible BackEMF waveform conditions that could be exhibited, it might not be practical to deal with all of them.

****

Thus the dilemma . . . How do we monitor a Steppers motion without the expense or size of an Encoder or Resolver, and do it in Real Time for just a few dollars?

Defining the Problem

As with any good solution, we must first define the problem.

Problem:

Detect that a stepper motor is vibrating in position, slipping, not moving, or moving in the wrong direction when being commanded to move at a certain velocity, and in a given direction.

This is actually a bit wordy. Perhaps it can be simplified . . . .

Determine if a motor is not doing what it is being commanded to do.

Simplify once more with even fewer words . . . .

Fault if motor and desired motion differ.

Notice that the whole problem comes down to knowing what the control is commanding the motor to do, and what the motor is actually doing. Thus, both the command signal(s) and the motors motion must be monitored. This would be true with an encoder system as well.

The fault is called . . . . **Excessive Following Error**

****

The question is, what motion command signals are available and what motor motion information is available to determine the following error? The problem gets even harder when the motor is micro-stepping. So lets analyze the problem by first looking at the available signals from both the control and the motor.

Stepper Control Signals (Desired):

There are many different types of stepper commands, but for the purpose of this paper, I only want to cover three. The remaining types are left to the readers imagination.

1. Move Pulse with Direction line
2. Move CW & CCW Pulses
3. Micro-Step Selection (Jumper)

Actual Motor motion Feedback:

There are only two different kinds of motor feedback indicators that we care about . . .

1. Motor armature Direction (CW/CCW rotation)
2. Motor armature Rotary or Linear Position in Steps

The challenge now is to monitor all of the above conditions, and develop a 'Watchdog' indicator that will tell us if a problem has occurred?

****

Developing the Solution

Since we want this is to be a Watchdog device, and since it has to make determinations as to command and actual relationships, I will set it up using a microprocessor.

The next step is to define the the I/O assignments . . .
Desired Inputs:

1. Move Command-1
2. Move Command-2
3. Direction
4. P&D or UpDnCntr Jumper (Pulses on In1 or In1 & In2)
5. Microstep selection-1 \
6. Microstep selection-2 \ 0000 = Full Step
7. Microstep selection-3 / 1000 = 51200 Steps / Rev
8. Microstep selection-4 /
9. Measure selection-1 \ 000 = 360 Degrees per measurement
10. Measure selection-2 | 111 = 45 Degrees per measurement
11. Measure selection-3 /

Actual Inputs:

1. Motor Shaft physical position digital indicator-1
2. Motor Shaft physical position digital indicator-2

Output:

1. Ok or Fault

This is about the minimum I/O requirement I could come up with. But remember, I'm developing a simple monitor. Your system could be simpler or more complex based on specific fixed information you know.

Let's analyze what we now have.

I have made a one or two wire Command input structure to insure I can accommodate Pulse and Direction, or CW / CCW pulse input commands.

I have allowed the microprocessor to know the microsteps per revolution in order to determine if stall or slip is occurring.

I have also allowed for rotational motor shaft movement to be captured by having dual directional inputs.

Finally, I have allowed for a test measurement to be made every 45 degrees of rotation if desired.

It will be the requirement of the Microprocessor watchdog software to determine if the actual motion is meeting the required motion conditions.

Hardware Design

Example 1:

Assume a Pulse command input. Let the steps per revolution be 51200, and the velocity be 1000RPM. Then the desired pulse time would be 19 microseconds per pulse. This is a bit fast for real time analysis, so let's look at it another way. Let's setup a pulse counter to count up when it receives desired move count pulses. Let's also set another counter to count up when an actual winding pulse takes place.

Then, once per revolution or up to eight times per revolution we can check to see if the two counter readings are the same, or close to it (allowable following error can be another jumper or switch setting). In addition, for any given speed, a timer can be set, via another jumper or potentiometer setting, to alarm if a count trigger does not trigger in the allotted time. On a direction change, the counter could be reset to zero, thus we would have immediate knowledge of slip. A pure stall would be indicated by an excessive (following error) indication between the two counters.

So what microprocessor should be used, and what should the hardware look like for all of this? For my application, I chose the SILABS C8051F121 100mips chip. The chip pinout is shown in Appendix A. Note, that this microprocessor supports 32 digital I/O, 8 analog inputs, two analog outputs, and two comparator inputs. This chip is a bit strong for the job, but it's fast, and it's cheap! It will do the job nicely. My setup would be as follows . . .

Command signal I/O setup:

P0.0	Pulse Input 1
P0.1	Pulse Input 2
P0.2	Direction Line
P0.3	P&D or UpDn Selection

P1.0	MicroStep Selection 1
P1.1	MicroStep Selection 2

P1.2	MicroStep Selection 3
P1.3	MicroStep Selection 4
P1.4	Measurement Selection 1
P1.5	Measurement Selection 2
P1.6	Measurement Selection 3

Motor signal I/O setup:

P2.0	Motor Shaft Position indicator-1
P2.1	Motor Shaft Position indicator-2

Response Signals to the Host:

P3.0	Velocity is 0
P3.1	Accelerating
P3.2	Running
P3.3	Decelerating
P3.4	Erratic Velocity
P3.5	Stalled
P3.6	Motor is running in the wrong direction

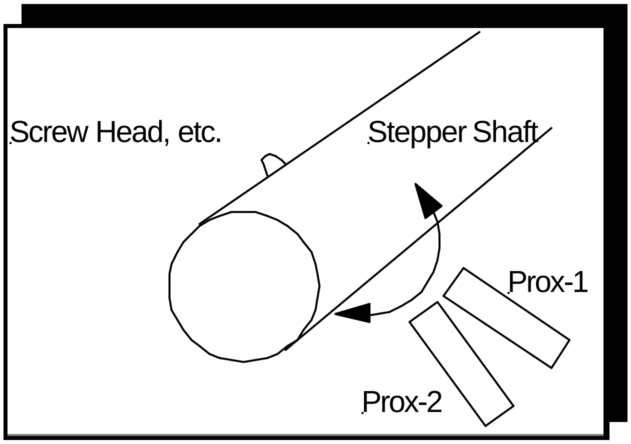


Figure 2

The digital position indicators (P2.0 & P2.1) could be some form of proximity switch, photo-eye, gear tooth / switch, low count encoder, or other device that can turn ON and OFF the inputs. The idea would be to place two digital 'prox' switches a small distance apart so that one will turn on before the other (Figure-2), but forming an uneven duty cycle (80:20 for example). The one that turns on first indicates the direction of rotation. The time between ON/OFF, ON/ON, OFF/ON, and OFF/OFF indications can be analyzed to determine if the motor is accelerating (P3.1), running (P3.2), decelerating (P3.3), stopped (P3.0). The velocity can be calculated based on 1/t. Large jumps in the velocity would indicate slip, potential stall, motor 120Hz vibration, on P3.4. P3.6 would indicate that the motor is running in the wrong direction, and P3.5, would indicate that the motor has fully stalled.

Now that we have our hardware layout ideas in place, lets develop some pseudo-code, and see if we can make this unit come alive.

Software Design

To keep the software to a minimum, the action only has to take place on command pulses from P0.1 or P0.2 and P0.3, and the a rotational indications on (P2.0 & P2.1). However, the unit may not be moving, we really don't want the software to stop running while its waiting for a physical motion indication. So while we monitor the command inputs, we will also run a timer to measure the rotational flags. Then, when the motor stops, the command pulses have also stopped, and we can reset the following Error counters to zero until the motion indicator prox's start once again.

Example Pseudo-Code:

Assumptions:

Pulse and Direction Stepper:
3200 steps per revolution
Proximity indicators for the rotation and direction triggers

Settings:

```
P0.0 =  Digital Command Signal
P0.1 = not Used
P0.2 =  Direction Command Line
P0.3 = P&D Selection

P1.0 =  Microstep selection-1 \
P1.1 =  Microstep selection-2   \   0101 = 3200 Steps/rev
P1.2 =  Microstep selection-3   /
P1.3 =  Microstep selection-4 /
P1.4     Measurement Selection 1 \
P1.5     Measurement Selection 2  |  0110 = One trigger every 60 degrees
P1.6     Measurement Selection 3 /
```

Code Example:

Set P0.0 to trigger a 16-bit Internal Pulse Counter using P0.2 as the count direction.

At 1000 RPM, one step will occur in . . .

1 / (1000 x 3200 / 60) = 1 / 53,333plspersec = 18.75usec

Then: 3200 / (360 / 60) = 533.3 steps per actual motor shaft indicator trigger.
If Velocity is required, then every 0.25 seconds put the change in Pulse count into a Velocity register and calculate as follows: Velocity = (V x 4 x 60) / 3200ppr   RPM

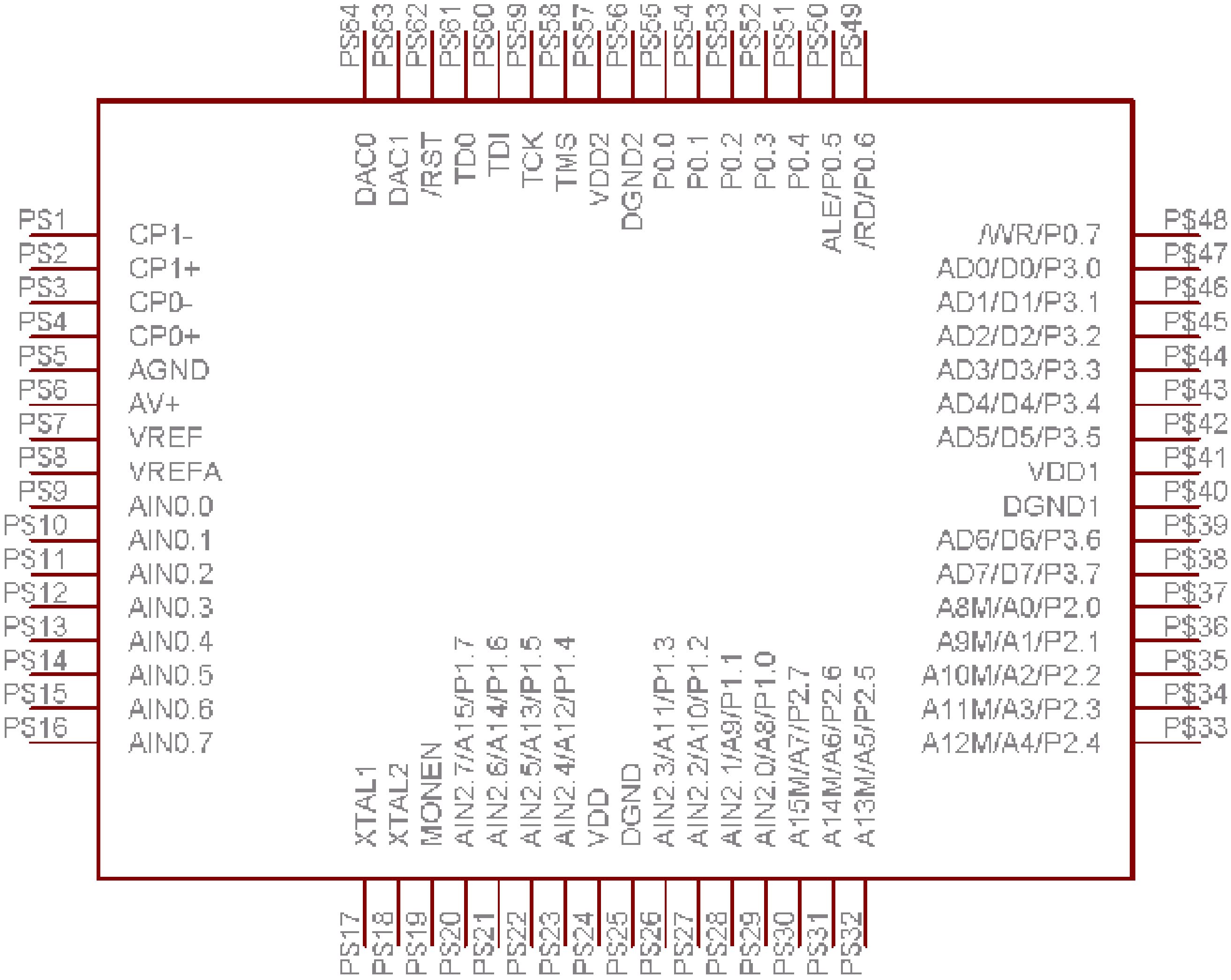
```
// Test to determine if the motor is going the right way
P2.6 = 0;                                              // Reset the direction Fault Indicator
If (P2.0 = = 1 && P2.1 = = 0)                          // Motor is turning CW
   If (DirLineInpt = = 1)          P3.6 = 1;           // Wrong Direction
Else If (P2.0 = = 0 && P2.1 = = 1)                     // Motor is turning CCW
   If   (DirLineInpt = = 0)        P3.6 = 1;           // Wrong Direction

// If Dir is OK & First time through this sequence
If  (WaitForRst = = 0 &&  P3.6 = = 0 && MathDoneFlag = = 0)
{   If (P2.0 = = 1 && P2.1 = = 0)
            MathDoneFlag = 1;                          // Only one time through on CW test
    Else if (P2.0 = = 0 && P2.1 = = 1)
            MathDoneFlag = 2;                          // Only one time through on CCW test

    DeltaCnt =  this_pulse_cnt -  last_pulse_cnt;      // How far have we moved
    // DeltaCnt should never exceed a MaxDeltaCnt
    P3.5 = 0;
    If (DeltaCnt > MaxDeltaCnt)          P3.5 = 1;

    ActPos = ActPos + DeltaCnt                     // Update the absolute position counter

    Fault_Cnt = 0;                                 // Reset the Fault Pulse Counter
}

If (MathDoneFlag != 0)
    (P2.0 = = 0 && P2.1 = = 0)   MthRstFlg = 1     // Get ready to reset the Fault Detector

If (MathDoneFlag != 0 && MthRstFlg != 0)           // Reset Requested?
{   If (MathDoneFlag = = 1 && P2.1 = = 1)
    {   MathDoneFlag = 0;                          // Reset the Flags
        MthRstFlg = 0;
     }
    Else If (MathDoneFlag = = 2) && P2.0 = = 1)
    {   MathDoneFlag = 0;                          // Reset the Flags
        MthRstFlg = 0;
     }
   WaitForRst = 1;
}

If (WaitForRst = = 1)
    If P2.0 = = 0 && P2.1 = = 0)   WaitForRst = 0;  // Reset Complete
```

****

Although the code seems rather simplistic, it will do the job. The key is to monitor both the command and what the motion is actually doing.

Appendix A

SILABS C8051F121 Pinout


PS64
PS63
PS62
PS61
PS60
PS59
PS58
PS57
PS56
PS55
PS54
PS53
PS52
PS51
PS50
PS49
DAC0
DAC1
/RST
TDO
TDI
TCK
TMS
VDD2
DGND2
P0.0
P0.1
P0.2
P0.3
P0.4
ALE/P0.5
/RD/P0.6
PS1
PS2
PS3
PS4
PS5
PS6
PS7
PS8
PS9
PS10
PS11
PS12
PS13
PS14
PS15
PS16
CP1-
CP1+
CP0-
CP0+
AGND
AV+
VREF
VREFA
AIN0.0
AIN0.1
AIN0.2
AIN0.3
AIN0.4
AIN0.5
AIN0.6
AIN0.7
/WR/P0.7
AD0/D0/P3.0
AD1/D1/P3.1
AD2/D2/P3.2
AD3/D3/P3.3
AD4/D4/P3.4
AD5/D5/P3.5
VDD1
DGND1
AD6/D6/P3.6
AD7/D7/P3.7
A8M/A0/P2.0
A9M/A1/P2.1
A10M/A2/P2.2
A11M/A3/P2.3
A12M/A4/P2.4
P$48
P$47
P$46
P$45
P$44
P$43
P$42
P$41
P$40
P$39
P$38
P$37
P$36
P$35
P$34
P$33
XTAL1
XTAL2
MONEN
AIN2.7/A15/P1.7
AIN2.6/A14/P1.6
AIN2.5/A13/P1.5
AIN2.4/A12/P1.4
VDD
DGND
AIN2.3/A11/P1.3
AIN2.2/A10/P1.2
AIN2.1/A9/P1.1
AIN2.0/A8/P1.0
A15M/A7/P2.7
A14M/A6/P2.6
A13M/A5/P2.5
PS17
PS18
PS19
PS20
PS21
PS22
PS23
PS24
PS25
PS26
PS27
PS28
PS29
PS30
PS31
PS32