```python
# stdglue -  canned prolog and epilog functions for the remappable builtin codes
(T,M6,M61,S,F)
#
# we dont use argspec to avoid the generic error message of the argspec prolog and give
more
# concise ones here

# cycle_prolog,cycle_epilog: generic code-independent support glue for oword sub cycles
#
# these are provided as starting point - for more concise error message you would better
# write a prolog specific for the code
#
# Usage:
#REMAP=G84.3  modalgroup=1 argspec=xyzqp prolog=cycle_prolog ngc=g843 epilog=cycle_epilog

import emccanon
from interpreter import *
throw_exceptions = 1

# REMAP=S   prolog=setspeed_prolog  ngc=setspeed epilog=setspeed_epilog
# exposed parameter: #<speed>

def setspeed_prolog(self,**words):
    try:
        c = self.blocks[self.remap_level]
        if not c.s_flag:
            self.set_errormsg("S requires a value")
            return INTERP_ERROR
        self.params["speed"] = c.s_number
    except Exception,e:
        self.set_errormsg("S/setspeed_prolog: %s)" % (e))
        return INTERP_ERROR
    return INTERP_OK

def setspeed_epilog(self,**words):
    try:
        if not self.value_returned:
            r = self.blocks[self.remap_level].executing_remap
            self.set_errormsg("the %s remap procedure %s did not return a value"
                             % (r.name,r.remap_ngc if r.remap_ngc else r.remap_py))
            return INTERP_ERROR
        if self.return_value < -TOLERANCE_EQUAL: # 'less than 0 within interp's
        precision'
            self.set_errormsg("S: remap procedure returned %f" % (self.return_value))
            return INTERP_ERROR
        if self.blocks[self.remap_level].builtin_used:
            pass
            #print "---------- S builtin recursion, nothing to do"
        else:
            self.speed = self.params["speed"]
            emccanon.enqueue_SET_SPINDLE_SPEED(self.speed)
        return INTERP_OK
    except Exception,e:
        self.set_errormsg("S/setspeed_epilog: %s)" % (e))
        return INTERP_ERROR
    return INTERP_OK

# REMAP=F   prolog=setfeed_prolog  ngc=setfeed epilog=setfeed_epilog
# exposed parameter: #<feed>

def setfeed_prolog(self,**words):
    try:
        c = self.blocks[self.remap_level]
        if not c.f_flag:
            self.set_errormsg("F requires a value")
            return INTERP_ERROR
        self.params["feed"] = c.f_number
    except Exception,e:
        self.set_errormsg("F/setfeed_prolog: %s)" % (e))
        return INTERP_ERROR
    return INTERP_OK

def setfeed_epilog(self,**words):
    try:
        if not self.value_returned:
            r = self.blocks[self.remap_level].executing_remap
            self.set_errormsg("the %s remap procedure %s did not return a value"
                             % (r.name,r.remap_ngc if r.remap_ngc else r.remap_py))
            return INTERP_ERROR
        if self.blocks[self.remap_level].builtin_used:
            pass
            #print "---------- F builtin recursion, nothing to do"
        else:
            self.feed_rate = self.params["feed"]
            emccanon.enqueue_SET_FEED_RATE(self.feed_rate)
        return INTERP_OK
    except Exception,e:
        self.set_errormsg("F/setfeed_epilog: %s)" % (e))
        return INTERP_ERROR
    return INTERP_OK

# REMAP=T   prolog=prepare_prolog ngc=prepare epilog=prepare_epilog
# exposed parameters: #<tool> #<pocket>

def prepare_prolog(self,**words):
    try:
        print "Entering Prolog........"
        cblock = self.blocks[self.remap_level]
        if not cblock.t_flag:
            self.set_errormsg("T requires a tool number")
            return INTERP_ERROR
        tool  = cblock.t_number
        if tool:
            (status, pocket) = self.find_tool_pocket(tool)
            if status != INTERP_OK:
                self.set_errormsg("T%d: pocket not found" % (tool))
                return status
        else:
            pocket = -1 # this is a T0 - tool unload
        self.params["tool"] = tool
        self.params["pocket"] = pocket
        return INTERP_OK
    except Exception, e:
        self.set_errormsg("T%d/prepare_prolog: %s" % (int(words['t']), e))
        return INTERP_ERROR

def prepare_epilog(self, **words):
    try:
        print "Entering Epilog...  Are we fucking here yet?"
        if not self.value_returned:
            r = self.blocks[self.remap_level].executing_remap
            self.set_errormsg("the %s remap procedure %s did not return a value"
                             % (r.name,r.remap_ngc if r.remap_ngc else r.remap_py))
            return INTERP_ERROR
        if self.blocks[self.remap_level].builtin_used:
            #print "---------- T builtin recursion, nothing to do"
            return INTERP_OK
        else:
            if self.return_value > 0:
                self.selected_tool = int(self.params["tool"])
                self.selected_pocket = int(self.params["pocket"])
                emccanon.SELECT_POCKET(self.selected_pocket, self.selected_tool)
                return INTERP_OK
            else:
```

```python
                    self.set_errormsg("T%d: aborted (return code %.1f)" %
                    (int(self.params["tool"]),self.return_value))
                    return INTERP_ERROR
        except Exception, e:
            self.set_errormsg("T%d/prepare_epilog: %s" % (tool,e))
            return INTERP_ERROR

    # REMAP=M6  modalgroup=6 prolog=change_prolog ngc=change epilog=change_epilog
    # exposed parameters:
    #    #<tool_in_spindle>
    #    #<selected_tool>
    #    #<current_pocket>
    #    #<selected_pocket>

    def change_prolog(self, **words):
        try:
            print "Entering Prolog........"
            if self.params[5600] > 0.0:
                if self.params[5601] < 0.0:
                    self.set_errormsg("Toolchanger hard fault %d" % (int(self.params[5601])))
                    return INTERP_ERROR
                print "change_prolog: Toolchanger soft fault %d" % int(self.params[5601])

            if self.selected_pocket < 0:
                self.set_errormsg("M6: no tool prepared")
                return INTERP_ERROR
            if self.cutter_comp_side:
                self.set_errormsg("Cannot change tools with cutter radius compensation on")
                return INTERP_ERROR
            self.params["tool_in_spindle"] = self.current_tool
            self.params["selected_tool"] = self.selected_tool
            self.params["current_pocket"] = self.current_pocket # this is probably nonsense
            self.params["selected_pocket"] = self.selected_pocket
            return INTERP_OK
        except Exception, e:
            self.set_errormsg("M6/change_prolog: %s" % (e))
            return INTERP_ERROR

    def change_epilog(self, **words):
        try:
            print "Entering Epilog........ Are we fucking here yet?"
            if not self.value_returned:
                r = self.blocks[self.remap_level].executing_remap
                self.set_errormsg("the %s remap procedure %s did not return a value"
                                 % (r.name,r.remap_ngc if r.remap_ngc else r.remap_py))
                yield INTERP_ERROR
            # this is relevant only when using iocontrol-v2.
            if self.params[5600] > 0.0:
                if self.params[5601] < 0.0:
                    self.set_errormsg("Toolchanger hard fault %d" % (int(self.params[5601])))
                    yield INTERP_ERROR
                print "change_epilog: Toolchanger soft fault %d" % int(self.params[5601])

            if self.blocks[self.remap_level].builtin_used:
                #print "---------- M6 builtin recursion, nothing to do"
                yield INTERP_OK
            else:
                if self.return_value > 0.0:
                    # commit change
                    self.selected_pocket =  int(self.params["selected_pocket"])
                    emccanon.CHANGE_TOOL(self.selected_pocket)
                    self.current_pocket = self.selected_pocket
                    self.selected_pocket = -1
                    self.selected_tool = -1
                    # cause a sync()
                    self.set_tool_parameters()
                    self.toolchange_flag = True
                    yield INTERP_EXECUTE_FINISH
                else:
                    self.set_errormsg("M6 aborted (return code %.1f)" % (self.return_value))
                    yield INTERP_ERROR
        except Exception, e:
            self.set_errormsg("M6/change_epilog: %s" % (e))
            yield INTERP_ERROR

    # REMAP=M61  modalgroup=6 prolog=settool_prolog ngc=settool epilog=settool_epilog
    # exposed parameters: #<tool> #<pocket>

    def settool_prolog(self,**words):
        try:
            c = self.blocks[self.remap_level]
            if not c.q_flag:
                self.set_errormsg("M61 requires a Q parameter")
                return INTERP_ERROR
            tool = int(c.q_number)
            if tool < -TOLERANCE_EQUAL: # 'less than 0 within interp's precision'
                self.set_errormsg("M61: Q value < 0")
                return INTERP_ERROR
            (status,pocket) = self.find_tool_pocket(tool)
            if status != INTERP_OK:
                self.set_errormsg("M61 failed: requested tool %d not in table" % (tool))
                return status
            self.params["tool"] = tool
            self.params["pocket"] = pocket
            return INTERP_OK
        except Exception,e:
            self.set_errormsg("M61/settool_prolog: %s)" % (e))
            return INTERP_ERROR

    def settool_epilog(self,**words):
        try:
            if not self.value_returned:
                r = self.blocks[self.remap_level].executing_remap
                self.set_errormsg("the %s remap procedure %s did not return a value"
                                 % (r.name,r.remap_ngc if r.remap_ngc else r.remap_py))
                return INTERP_ERROR

            if self.blocks[self.remap_level].builtin_used:
                #print "---------- M61 builtin recursion, nothing to do"
                return INTERP_OK
            else:
                if self.return_value > 0.0:
                    self.current_tool = int(self.params["tool"])
                    self.current_pocket = int(self.params["pocket"])
                    emccanon.CHANGE_TOOL_NUMBER(self.current_pocket)
                    # cause a sync()
                    self.tool_change_flag = True
                    self.set_tool_parameters()
                else:
                    self.set_errormsg("M61 aborted (return code %.1f)" % (self.return_value))
                    return INTERP_ERROR
        except Exception,e:
            self.set_errormsg("M61/settool_epilog: %s)" % (e))
            return INTERP_ERROR

    # educational alternative: M61 remapped to an all-Python handler
    # demo - this really does the same thing as the builtin (non-remapped) M61
    #
    # REMAP=M61 modalgroup=6 python=set_tool_number

    def set_tool_number(self, **words):
        try:
            c = self.blocks[self.remap_level]
            if c.q_flag:
                toolno = int(c.q_number)
            else:
                self.set_errormsg("M61 requires a Q parameter")
                return status
            (status,pocket) = self.find_tool_pocket(toolno)
            if status != INTERP_OK:
                self.set_errormsg("M61 failed: requested tool %d not in table" % (toolno))
                return status
            if words['q'] > -TOLERANCE_EQUAL: # 'greater equal 0 within interp's precision'
                self.current_pocket = pocket
                self.current_tool = toolno
                emccanon.CHANGE_TOOL_NUMBER(pocket)
                # cause a sync()
                self.tool_change_flag = True
                self.set_tool_parameters()
                return INTERP_OK
            else:
                self.set_errormsg("M61 failed: Q=%4" % (toolno))
                return INTERP_ERROR
        except Exception, e:
            self.set_errormsg("M61/set_tool_number: %s" % (e))
            return INTERP_ERROR

    _uvw = ("u","v","w","a","b","c")
    _xyz = ("x","y","z","a","b","c")
    # given a plane, return  sticky words, incompatible axis words and plane name
    # sticky[0] is also the movement axis
    _compat = {
        emccanon.CANON_PLANE_XY : (("z","r"),_uvw,"XY"),
        emccanon.CANON_PLANE_YZ : (("x","r"),_uvw,"YZ"),
        emccanon.CANON_PLANE_XZ : (("y","r"),_uvw,"XZ"),
        emccanon.CANON_PLANE_UV : (("w","r"),_xyz,"UV"),
        emccanon.CANON_PLANE_VW : (("u","r"),_xyz,"VW"),
        emccanon.CANON_PLANE_UW : (("v","r"),_xyz,"UW")}

    # extract and pass parameters from current block, merged with extra parameters on a continuation line
    # keep tjose parameters across invocations
    # export the parameters into the oword procedure
    def cycle_prolog(self,**words):
        # self.sticky_params is assumed to have been initialized by the
        # init_stgdlue() method below
        global _compat
        try:
            # determine whether this is the first or a subsequent call
            c = self.blocks[self.remap_level]
            r = c.executing_remap
            if c.g_modes[1] == r.motion_code:
                # first call - clear the sticky dict
                self.sticky_params[r.name] = dict()

            self.params["motion_code"] = c.g_modes[1]

            (sw,incompat,plane_name) =_compat[self.plane]
            for (word,value) in words.items():
                # inject current parameters
                self.params[word] = value
                # record sticky words
                if word in sw:
                    if self.debugmask & 0x00080000: print "%s: record sticky %s = %.4f" % (r.name,word,value)
                    self.sticky_params[r.name][word] = value
                if word in incompat:
                    return "%s: Cannot put a %s in a canned cycle in the %s plane" % (r.name, word.upper(), plane_name)

            # inject sticky parameters which were not in words:
            for (key,value) in self.sticky_params[r.name].items():
                if not key in words:
                    if self.debugmask & 0x00080000: print "%s: inject sticky %s = %.4f" %
                    (r.name,key,value)
                    self.params[key] = value

            if not "r" in self.sticky_params[r.name]:
                return "%s: cycle requires R word" % (r.name)
            else:
                if self.sticky_params[r.name] <= 0.0:
                    return "%s: R word must be > 0 if used (%.4f)" % (r.name, words["r"])

            if "l" in words:
                # checked in interpreter during block parsing
                # if l <= 0 or l not near an int
                self.params["l"] = words["l"]

            if "p" in words:
                p = words["p"]
                if p < 0.0:
                    return "%s: P word must be >= 0 if used (%.4f)" % (r.name, p)
                self.params["p"] = p

            if self.feed_rate == 0.0:
                return "%s: feed rate must be > 0" % (r.name)
            if self.feed_mode == INVERSE_TIME:
                return "%s: Cannot use inverse time feed with canned cycles" % (r.name)
            if self.cutter_comp_side:
                return "%s: Cannot use canned cycles with cutter compensation on" % (r.name)
            return INTERP_OK

        except Exception, e:
            raise
            return "cycle_prolog failed: %s" % (e)

    # make sure the next line has the same motion code, unless overriden by a
    # new G code
    def cycle_epilog(self,**words):
        try:
            c = self.blocks[self.remap_level]
            self.motion_mode = c.executing_remap.motion_code # retain the current motion mode
            return INTERP_OK
        except Exception, e:
            return "cycle_epilog failed: %s" % (e)

    # this should be called from TOPLEVEL __init__()
    def init_stdglue(self):
        self.sticky_params = dict()

```