

```

1  # stdglue - canned prolog and epilog functions for the remappable builtin codes
   (T,M6,M61,S,F)
2  #
3  # we dont use argspec to avoid the generic error message of the argspec prolog and give
   more
4  # concise ones here
5
6  # cycle_prolog,cycle_epilog: generic code-independent support glue for oword sub cycles
7  #
8  # these are provided as starting point - for more concise error message you would better
9  # write a prolog specific for the code
10 #
11 # Usage:
12 #REMAP=G84.3 modalgroup=1 argspec=xyzqp prolog=cycle_prolog ngc=g843 epilog=cycle_epilog
13
14 import emccanon
15 from interpreter import *
16 throw_exceptions = 1
17
18 # REMAP=S prolog=setspeed_prolog ngc=setspeed epilog=setspeed_epilog
19 # exposed parameter: #<speed>
20
21 def setspeed_prolog(self,**words):
22     try:
23         c = self.blocks[self.remap_level]
24         if not c.s_flag:
25             self.set_errormsg("S requires a value")
26             return INTERP_ERROR
27         self.params["speed"] = c.s_number
28     except Exception,e:
29         self.set_errormsg("S/setspeed_prolog: %s)" % (e))
30         return INTERP_ERROR
31     return INTERP_OK
32
33 def setspeed_epilog(self,**words):
34     try:
35         if not self.value_returned:
36             r = self.blocks[self.remap_level].executing_remap
37             self.set_errormsg("the %s remap procedure %s did not return a value"
38                               % (r.name,r.remap_ngc if r.remap_ngc else r.remap_py))
39             return INTERP_ERROR
40         if self.return_value < -TOLERANCE_EQUAL: # 'less than 0 within interp's
           precision'
41             self.set_errormsg("S: remap procedure returned %f" % (self.return_value))
42             return INTERP_ERROR
43         if self.blocks[self.remap_level].builtin_used:
44             pass
45             #print "----- S builtin recursion, nothing to do"
46         else:
47             self.speed = self.params["speed"]
48             emccanon.enqueue_SET_SPINDLE_SPEED(self.speed)
49             return INTERP_OK
50     except Exception,e:
51         self.set_errormsg("S/setspeed_epilog: %s)" % (e))
52         return INTERP_ERROR
53     return INTERP_OK
54
55 # REMAP=F prolog=setfeed_prolog ngc=setfeed epilog=setfeed_epilog
56 # exposed parameter: #<feed>
57
58 def setfeed_prolog(self,**words):
59     try:
60         c = self.blocks[self.remap_level]
61         if not c.f_flag:
62             self.set_errormsg("F requires a value")
63             return INTERP_ERROR
64         self.params["feed"] = c.f_number

```

```

65     except Exception,e:
66         self.set_errormsg("F/setfeed_prolog: %s)" % (e))
67         return INTERP_ERROR
68     return INTERP_OK
69
70 def setfeed_epilog(self,**words):
71     try:
72         if not self.value_returned:
73             r = self.blocks[self.remap_level].executing_remap
74             self.set_errormsg("the %s remap procedure %s did not return a value"
75                               % (r.name,r.remap_ngc if r.remap_ngc else r.remap_py))
76             return INTERP_ERROR
77         if self.blocks[self.remap_level].builtin_used:
78             pass
79             #print "----- F builtin recursion, nothing to do"
80         else:
81             self.feed_rate = self.params["feed"]
82             emccanon.enqueue_SET_FEED_RATE(self.feed_rate)
83             return INTERP_OK
84     except Exception,e:
85         self.set_errormsg("F/setfeed_epilog: %s)" % (e))
86         return INTERP_ERROR
87     return INTERP_OK
88
89 # REMAP=T   prolog=prepare_prolog ngc=prepare epilog=prepare_epilog
90 # exposed parameters: #<tool> #<pocket>
91
92 def prepare_prolog(self,**words):
93     try:
94         print "Entering Prolog....."
95         cblock = self.blocks[self.remap_level]
96         if not cblock.t_flag:
97             self.set_errormsg("T requires a tool number")
98             return INTERP_ERROR
99         tool = cblock.t_number
100        if tool:
101            (status, pocket) = self.find_tool_pocket(tool)
102            if status != INTERP_OK:
103                self.set_errormsg("T%d: pocket not found" % (tool))
104                return status
105        else:
106            pocket = -1 # this is a T0 - tool unload
107            self.params["tool"] = tool
108            self.params["pocket"] = pocket
109            return INTERP_OK
110    except Exception, e:
111        self.set_errormsg("T%d/prepare_prolog: %s" % (int(words['t']), e))
112        return INTERP_ERROR
113
114 def prepare_epilog(self, **words):
115     try:
116         print "Entering Epilog... Are we fucking here yet?"
117         if not self.value_returned:
118             r = self.blocks[self.remap_level].executing_remap
119             self.set_errormsg("the %s remap procedure %s did not return a value"
120                               % (r.name,r.remap_ngc if r.remap_ngc else r.remap_py))
121             return INTERP_ERROR
122         if self.blocks[self.remap_level].builtin_used:
123             #print "----- T builtin recursion, nothing to do"
124             return INTERP_OK
125         else:
126             if self.return_value > 0:
127                 self.selected_tool = int(self.params["tool"])
128                 self.selected_pocket = int(self.params["pocket"])
129                 emccanon.SELECT_POCKET(self.selected_pocket, self.selected_tool)
130                 return INTERP_OK
131             else:

```

```

132         self.set_errormsg("T%d: aborted (return code %.1f)" %
133             (int(self.params["tool"]),self.return_value))
134         return INTERP_ERROR
135     except Exception, e:
136         self.set_errormsg("T%d/prepare_epilog: %s" % (tool,e))
137         return INTERP_ERROR
138
139 # REMAP=M6 modalgroup=6 prolog=change_prolog ngc=change_epilog=change_epilog
140 # exposed parameters:
141 #   #<tool_in_spindle>
142 #   #<selected_tool>
143 #   #<current_pocket>
144 #   #<selected_pocket>
145
146 def change_prolog(self, **words):
147     try:
148         print "Entering Prolog....."
149         if self.params[5600] > 0.0:
150             if self.params[5601] < 0.0:
151                 self.set_errormsg("Toolchanger hard fault %d" % (int(self.params[5601])))
152                 return INTERP_ERROR
153             print "change_prolog: Toolchanger soft fault %d" % int(self.params[5601])
154
155         if self.selected_pocket < 0:
156             self.set_errormsg("M6: no tool prepared")
157             return INTERP_ERROR
158         if self.cutter_comp_side:
159             self.set_errormsg("Cannot change tools with cutter radius compensation on")
160             return INTERP_ERROR
161         self.params["tool_in_spindle"] = self.current_tool
162         self.params["selected_tool"] = self.selected_tool
163         self.params["current_pocket"] = self.current_pocket # this is probably nonsense
164         self.params["selected_pocket"] = self.selected_pocket
165         return INTERP_OK
166     except Exception, e:
167         self.set_errormsg("M6/change_prolog: %s" % (e))
168         return INTERP_ERROR
169
170 def change_epilog(self, **words):
171     try:
172         print "Entering Epilog..... Are we fucking here yet?"
173         if not self.value_returned:
174             r = self.blocks[self.remap_level].executing_remap
175             self.set_errormsg("the %s remap procedure %s did not return a value"
176                 % (r.name,r.remap_ngc if r.remap_ngc else r.remap_py))
177             yield INTERP_ERROR
178             # this is relevant only when using iocontrol-v2.
179         if self.params[5600] > 0.0:
180             if self.params[5601] < 0.0:
181                 self.set_errormsg("Toolchanger hard fault %d" % (int(self.params[5601])))
182                 yield INTERP_ERROR
183             print "change_epilog: Toolchanger soft fault %d" % int(self.params[5601])
184
185         if self.blocks[self.remap_level].builtin_used:
186             #print "----- M6 builtin recursion, nothing to do"
187             yield INTERP_OK
188         else:
189             if self.return_value > 0.0:
190                 # commit change
191                 self.selected_pocket = int(self.params["selected_pocket"])
192                 emccanon.CHANGE_TOOL(self.selected_pocket)
193                 self.current_pocket = self.selected_pocket
194                 self.selected_pocket = -1
195                 self.selected_tool = -1
196                 # cause a sync()
197                 self.set_tool_parameters()
198                 self.toolchange_flag = True

```

```

198         yield INTERP_EXECUTE_FINISH
199     else:
200         self.set_errormsg("M6 aborted (return code %.1f)" % (self.return_value))
201         yield INTERP_ERROR
202 except Exception, e:
203     self.set_errormsg("M6/change_epilog: %s" % (e))
204     yield INTERP_ERROR
205
206 # REMAP=M61 modalgroup=6 prolog=settool_prolog ngc=settool_epilog=settool_epilog
207 # exposed parameters: #<tool> #<pocket>
208
209 def settool_prolog(self,**words):
210     try:
211         c = self.blocks[self.remap_level]
212         if not c.q_flag:
213             self.set_errormsg("M61 requires a Q parameter")
214             return INTERP_ERROR
215         tool = int(c.q_number)
216         if tool < -TOLERANCE_EQUAL: # 'less than 0 within interp's precision'
217             self.set_errormsg("M61: Q value < 0")
218             return INTERP_ERROR
219         (status,pocket) = self.find_tool_pocket(tool)
220         if status != INTERP_OK:
221             self.set_errormsg("M61 failed: requested tool %d not in table" % (tool))
222             return status
223         self.params["tool"] = tool
224         self.params["pocket"] = pocket
225         return INTERP_OK
226 except Exception,e:
227     self.set_errormsg("M61/settool_prolog: %s)" % (e))
228     return INTERP_ERROR
229
230 def settool_epilog(self,**words):
231     try:
232         if not self.value_returned:
233             r = self.blocks[self.remap_level].executing_remap
234             self.set_errormsg("the %s remap procedure %s did not return a value"
235                             % (r.name,r.remap_ngc if r.remap_ngc else r.remap_py))
236             return INTERP_ERROR
237
238         if self.blocks[self.remap_level].builtin_used:
239             #print "----- M61 builtin recursion, nothing to do"
240             return INTERP_OK
241         else:
242             if self.return_value > 0.0:
243                 self.current_tool = int(self.params["tool"])
244                 self.current_pocket = int(self.params["pocket"])
245                 emccanon.CHANGE_TOOL_NUMBER(self.current_pocket)
246                 # cause a sync()
247                 self.tool_change_flag = True
248                 self.set_tool_parameters()
249             else:
250                 self.set_errormsg("M61 aborted (return code %.1f)" % (self.return_value))
251                 return INTERP_ERROR
252         except Exception,e:
253             self.set_errormsg("M61/settool_epilog: %s)" % (e))
254             return INTERP_ERROR
255
256 # educational alternative: M61 remapped to an all-Python handler
257 # demo - this really does the same thing as the builtin (non-remapped) M61
258 #
259 # REMAP=M61 modalgroup=6 python=set_tool_number
260
261 def set_tool_number(self, **words):
262     try:
263         c = self.blocks[self.remap_level]
264         if c.q_flag:

```

```

265         toolno = int(c.q_number)
266     else:
267         self.set_errormsg("M61 requires a Q parameter")
268         return status
269     (status,pocket) = self.find_tool_pocket(toolno)
270     if status != INTERP_OK:
271         self.set_errormsg("M61 failed: requested tool %d not in table" % (toolno))
272         return status
273     if words['q'] > -TOLERANCE_EQUAL: # 'greater equal 0 within interp's precision'
274         self.current_pocket = pocket
275         self.current_tool = toolno
276         emccanon.CHANGE_TOOL_NUMBER(pocket)
277         # cause a sync()
278         self.tool_change_flag = True
279         self.set_tool_parameters()
280         return INTERP_OK
281     else:
282         self.set_errormsg("M61 failed: Q=%4" % (toolno))
283         return INTERP_ERROR
284     except Exception, e:
285         self.set_errormsg("M61/set_tool_number: %s" % (e))
286         return INTERP_ERROR
287
288     _uvw = ("u","v","w","a","b","c")
289     _xyz = ("x","y","z","a","b","c")
290     # given a plane, return sticky words, incompatible axis words and plane name
291     # sticky[0] is also the movement axis
292     _compat = {
293         emccanon.CANON_PLANE_XY : (("z","r"),_uvw,"XY"),
294         emccanon.CANON_PLANE_YZ : (("x","r"),_uvw,"YZ"),
295         emccanon.CANON_PLANE_XZ : (("y","r"),_uvw,"XZ"),
296         emccanon.CANON_PLANE_UV : (("w","r"),_xyz,"UV"),
297         emccanon.CANON_PLANE_VW : (("u","r"),_xyz,"VW"),
298         emccanon.CANON_PLANE_UW : (("v","r"),_xyz,"UW")}
299
300     # extract and pass parameters from current block, merged with extra parameters on a
301     # continuation line
302     # keep tjose parameters across invocations
303     # export the parameters into the oword procedure
304     def cycle_prolog(self,**words):
305         # self.sticky_params is assumed to have been initialized by the
306         # init_stgdlue() method below
307         global _compat
308         try:
309             # determine whether this is the first or a subsequent call
310             c = self.blocks[self.remap_level]
311             r = c.executing_remap
312             if c.g_modes[1] == r.motion_code:
313                 # first call - clear the sticky dict
314                 self.sticky_params[r.name] = dict()
315
316             self.params["motion_code"] = c.g_modes[1]
317
318             (sw,incompat,plane_name) = _compat[self.plane]
319             for (word,value) in words.items():
320                 # inject current parameters
321                 self.params[word] = value
322                 # record sticky words
323                 if word in sw:
324                     if self.debugmask & 0x00080000: print "%s: record sticky %s = %.4f" %
325                         (r.name,word,value)
326                     self.sticky_params[r.name][word] = value
327                 if word in incompat:
328                     return "%s: Cannot put a %s in a canned cycle in the %s plane" %
329                         (r.name, word.upper(), plane_name)
330
331             # inject sticky parameters which were not in words:

```

```

329     for (key,value) in self.sticky_params[r.name].items():
330         if not key in words:
331             if self.debugmask & 0x00080000: print "%s: inject sticky %s = %.4f" %
                 (r.name,key,value)
332             self.params[key] = value
333
334     if not "r" in self.sticky_params[r.name]:
335         return "%s: cycle requires R word" % (r.name)
336     else:
337         if self.sticky_params[r.name] <= 0.0:
338             return "%s: R word must be > 0 if used (%.4f)" % (r.name, words["r"])
339
340     if "l" in words:
341         # checked in interpreter during block parsing
342         # if l <= 0 or l not near an int
343         self.params["l"] = words["l"]
344
345     if "p" in words:
346         p = words["p"]
347         if p < 0.0:
348             return "%s: P word must be >= 0 if used (%.4f)" % (r.name, p)
349         self.params["p"] = p
350
351     if self.feed_rate == 0.0:
352         return "%s: feed rate must be > 0" % (r.name)
353     if self.feed_mode == INVERSE_TIME:
354         return "%s: Cannot use inverse time feed with canned cycles" % (r.name)
355     if self.cutter_comp_side:
356         return "%s: Cannot use canned cycles with cutter compensation on" % (r.name)
357     return INTERP_OK
358
359     except Exception, e:
360         raise
361         return "cycle_prolog failed: %s" % (e)
362
363     # make sure the next line has the same motion code, unless overridden by a
364     # new G code
365     def cycle_epilog(self,**words):
366         try:
367             c = self.blocks[self.remap_level]
368             self.motion_mode = c.executing_remap.motion_code # retain the current motion mode
369             return INTERP_OK
370         except Exception, e:
371             return "cycle_epilog failed: %s" % (e)
372
373     # this should be called from TOPLEVEL __init__()
374     def init_stdglue(self):
375         self.sticky_params = dict()
376

```