RAPL-3
Language Reference Guide
UMI-R3-210
RAPL-3 Language Reference Guide

<table>
<thead>
<tr>
<th>Revision</th>
<th>Revision History</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td>Initial release as UMI-R3-210 for C500C.</td>
<td>99-05</td>
</tr>
<tr>
<td></td>
<td>CROS 1.16.</td>
<td></td>
</tr>
<tr>
<td>001a</td>
<td>CROS 2.0.1080. Recent revision information is in the release notes on the diskettes.</td>
<td>99-09</td>
</tr>
<tr>
<td>001b</td>
<td>Replaced references to the Application Development Guide with references to the Robot System Software Documentation Guide.</td>
<td>00-11</td>
</tr>
</tbody>
</table>

Copyright © 2000 CRS Robotics Corporation

RAPL-3 and RAPL are trademarks of CRS Robotics Corporation and may be used to describe only CRS Robotics products.

All brand names and product names used in this guide are trademarks, registered trademarks, or trade names of their respective holders.

The information in this document is subject to change without notice.

CRS Robotics Corporation makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. CRS Robotics Corporation assumes no responsibility for any errors that may appear in this document. CRS Robotics Corporation makes no commitment to update nor to keep current the information contained in this document.

CRS Robotics Corporation software products shall remain the property of CRS Robotics Corporation.

Questions about the RAPL-3 programming language can be directed to the Customer Support Department.

RAPL-3 and robot training courses are offered at CRS Robotics in Burlington, Ontario, Canada, or can be conducted at your facility. For additional information contact the Customer Support Department.

Additional copies of this guide, or other CRS Robotics literature, may be obtained from the Sales Department or from your distributor.

CRS Robotics Corporation

Mail/Shipment:
5344 John Lucas Drive, Burlington, Ontario L7L 6A6, Canada

Telephone:
1-905-332-2000

Telephone (toll free in Canada and United States):
1-800-365-7587

Facsimile:
1-905-332-1114

E-Mail (General):
info@crsrobotics.com

E-Mail (Customer Support):
support@crsrobotics.com

Web:
www.crsrobotics.com
CRS Licence Agreement

IMPORTANT! – READ CAREFULLY BEFORE OPENING SOFTWARE PACKET(S)

By opening the sealed packet(s) containing the software, you indicate your acceptance of the following CRS Licence Agreement.

A. CRS LICENCE AGREEMENT

This is a legal binding agreement between you, the end user, (either an individual or an entity) and CRS Robotics Corporation (“CRS”). By opening the sealed software packages and/or by using the SOFTWARE program you agree to be bound by the terms of this Agreement. If you do not agree to the terms of this Agreement, promptly return the unopened software and the accompanying items (including printed materials and binders or other containers) to CRS for a full refund.

If you are acting on behalf of a corporation, you represent to CRS that you are authorized to act on behalf of such organization and that your assent to the terms of this Agreement creates a legally enforceable obligation on your organization. As used herein, “you” and “your” refers to you and any organization on behalf of which you are acting.

This Agreement together with any applicable CRS Agreement embodies the entire understanding between the parties with respect to, and supersedes any prior understanding or agreement, oral or written, relating to the SOFTWARE.

B. CRS ROBOTICS CORPORATION (“CRS”) SOFTWARE LICENCE

1. GRANT OF LICENCE: This Licence Agreement permits you to use one copy of the enclosed CRS “POLARA” software program (“SOFTWARE”) on a single computer. The SOFTWARE is in “use” on a computer when it is loaded into temporary memory (ie. RAM) or installed into permanent memory (e.g. hard disk, CD-ROM, or other storage device) of that computer. However, installation on a network for the sole purpose of internal distribution shall not constitute “use” for which a separate licence is required, provided you have a separate licence for each computer which the SOFTWARE is distributed. In no event may the total number of users on a network exceed the number of licences acquired for a network.

If you make additional copies of the SOFTWARE or its accompanying documentation contrary to this Agreement, or if the number of users is greater than that for which you have paid a licence fee, CRS may require that you immediately make payment to CRS for such copies and/or such use at the current list price. This remedy is in addition to any other remedies that CRS may have against you.

2. UPGRADES: Upgrades to SOFTWARE may be provided by CRS at a 20% annual cost of the original price of the software. If the SOFTWARE is an upgrade from another software product licensed to you, whether a CRS product or a third-party product, the SOFTWARE must be used and transferred in conjunction with the upgraded product, unless you destroy the upgraded product. You are authorized to use the SOFTWARE only if you are an authorized user of a qualifying product as determined by CRS.

3. COPYRIGHT: The SOFTWARE (including any images, photographs, animations, video, audio, music and text incorporated into the SOFTWARE is owned by CRS and is protected by copyright laws and international treaty provisions. Therefore, you must treat the SOFTWARE like any other copyrighted
material except that you may make up to two archival copies of the SOFTWARE for the sole purpose of protecting your investment from loss. You may not copy the documentation accompanying the SOFTWARE.

4. OTHER RESTRICTIONS: You may not rent or lease the SOFTWARE. You may not reverse engineer, decompile, or disassemble the SOFTWARE. This licence does not grant you any right to any enhancement or update to the SOFTWARE. Any enhancements and updates, if available, may be obtained from CRS at current pricing, terms and conditions.

5. TERMINATION: The licence will terminate automatically if you fail to comply with the limitations described herein.

6. GOVERNING LAWS: If you acquired this product in Canada, this Agreement is governed by the laws of the Province of Ontario. Each of the parties hereto irrevocably attorns to the jurisdiction of the courts of Ontario and further agrees to commence any litigation which may arise hereunder in the courts located in the Judicial District of York, Province of Ontario.

If you acquired this product in the United States, this Agreement is governed by the laws of the state of Washington.

U.S. Government Restricted Rights. Use, duplication or disclosure by the Government is subject to restrictions set forth in subparagraphs (a) through (d) of the Commercial Computer-Restricted Rights clause at FAR 52.227-10 when applicable, or in paragraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013, and in similar clauses in the NASA FAR Supplement. Contract/manufacturer is CRS Robotics Corporation, 5344 John Lucas Dr. Burlington, ON Canada L7L 6A6.

C. LIMITED WARRANTY

LIMITED WARRANTY: CRS warrants that the SOFTWARE will perform substantially in accordance with the accompanying materials for a period of 60 days. The liability of CRS and your exclusive remedy shall be limited to the amount paid by you for the SOFTWARE and its accompanying documentation.

NO OTHER WARRANTIES: The SOFTWARE is provided on an “as is” basis. To the maximum extent permitted by applicable law. CRS disclaims all other warranties, express or implied, including, but not limited to, implied warranties of merchantability and fitness for a particular purpose, with regard to the SOFTWARE and the accompanying printed materials.

The entire risk as to the results and performance of the SOFTWARE is assumed by you. In particular, CRS does not accept any responsibility for any portions of the SOFTWARE which have been modified by you or on your behalf.

NO LIABILITY FOR CONSEQUENTIAL DAMAGES: In no event shall CRS or its distributors be liable for any damages whatsoever (including but not limited to damages for loss of business profits, business interruption, loss of business information or other pecuniary loss) arising out of the use or inability to use the SOFTWARE or its accompanying documentation, even if CRS has been advised of the possibility of such damages. The entire liability of CRS under this Agreement with respect to the SOFTWARE shall be limited to the amount paid by you for the SOFTWARE and its accompanying documentation.
External Copyright Notices

CROS and RAPL-3 contain portions of code that are copyrighted by other organizations. CRS Robotics Corp. acknowledges the following copyrights.

The Regents of the University of California
Copyright (c) 1993 The Regents of the University of California.
All rights reserved.
Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:
1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. All advertising materials mentioning features or use of this software must display the following acknowledgement:
   This product includes software developed by the University of California, Berkeley and its contributors.
4. Neither the name of the University nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Sun Microsystems, Inc.
Copyright (C) 1993 by Sun Microsystems, Inc. All rights reserved.
Developed at SunPro, a Sun Microsystems, Inc. business.
Permission to use, copy, modify, and distribute this software is freely granted, provided that this notice is preserved.

Carnegie Mellon University
Mach Operating System
Copyright (c) 1992, 1991 Carnegie Mellon University
All Rights Reserved.
Permission to use, copy, modify and distribute this software and its documentation is hereby granted, provided that both the copyright notice and this permission notice appear in all copies of the software, derivative works or modified versions, and any portions thereof, and that both notices appear in supporting documentation.
CARNEGIE MELLON ALLOWS FREE USE OF THIS SOFTWARE IN ITS "AS IS" CONDITION. CARNEGIE MELLON DISCLAIMS ANY LIABILITY OF ANY KIND FOR ANY DAMAGES WHATSOEVER RESULTING FROM THE USE OF THIS SOFTWARE.

Winning Strategies, Inc.

Copyright (c) 1993
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3. All advertising materials mentioning features or use of this software must display the following acknowledgement:
   This product includes software developed by Winning Strategies, Inc.

4. The name of the author may not be used to endorse or promote products derived from this software without specific prior written permission

THIS SOFTWARE IS PROVIDED BY THE AUTHOR "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
Contents at a Glance

Preface ..................................................................................... 1

Chapter 1
General Program Format ....................................................... 5

Chapter 2
Data Types and Variables ....................................................... 13

Chapter 3
Expressions, Assignment, and Operators ................................. 35

Chapter 4
Constants .............................................................................. 41

Chapter 5
Control Flow ........................................................................... 47

Chapter 6
Subroutines, Functions and Commands .................................. 57

Chapter 7
Preprocessor Directives .......................................................... 69

Chapter 8
Structured Exception Handling ............................................. 77

Chapter 9
Library Subprograms .............................................................. 81
Subprograms: Category Listing .............................................. 91
Subprograms: Alphabetical Listing ..................................... 139

APPENDICES
Signals ..................................................................................... 341
Contents

Preface ..................................................................................... 1
 Documentation Conventions .................................................. 2
 Text and Programming Code ............................................. 2
 Commands and Keywords .................................................... 3
 name_of_command/keyword .................................................. 3
 Related Resources ............................................................... 4

Chapter 1
 General Program Format ........................................................... 5
 Example 1: Basic Program in RAPL-II Style .......................... 6
 Example 2: Basic Program in Preferred RAPL-3 Style ...... 7
 The Main Program .................................................................. 8
 main .................................................................................. 8
 Lines of a Program ................................................................ 9
 Comments .......................................................................... 10
 Labels ............................................................................. 11
 Keywords ........................................................................... 12

Chapter 2
 Data Types and Variables ....................................................... 13
 Basic Data Types ................................................................ 14
 int .................................................................................. 14
 float ............................................................................... 14
 string ............................................................................ 14
 cloc ............................................................................. 14
 ploc ............................................................................. 14
 void ................................................................................ 15
 Identifiers .......................................................................... 16
 Character Set ...................................................................... 16
 Case ............................................................................. 16
 Length ........................................................................... 16
 Examples ....................................................................... 16
 Declarations ..................................................................... 17
 Explicit Declarations ........................................................ 17
 Implicit Declarations ....................................................... 17
 Implicit with Explicit ..................................................... 18
 Identifiers ..................................................................... 18
 Strings ............................................................................. 19
 String[number] ................................................................ 19
 String[] ........................................................................ 19
 Notes: ........................................................................... 19
 Termination .................................................................. 20
 Concatenation .................................................................. 20
 Arrays .............................................................................. 21
 Teachables ....................................................................... 23
 Use ................................................................................ 23
 Variable (v3) File ............................................................. 23
 Declarations ................................................................ 23
 Limitations ...................................................................... 24
 Defaults and Un teachables ............................................. 25
 User-Defined Types ........................................................... 26
 Pointers .......................................................................... 27
 Dereferencing ................................................................. 27
 Address-of Operator .......................................................... 27
Chapter 7
Preprocessor Directives ........................................................... 69
  File Inclusion ..................................................................... 70
  Macro Substitution ...................................................... 71
  Conditional Compilation .............................................. 72
Preprocessor Directives in General .................................... 73
The Preprocessor Directives ............................................... 74
  .define......................................................................... 74
  .error .......................................................................... 74
  .ifdef ........................................................................... 74
  .ifndef ......................................................................... 74
  .include....................................................................... 75
  .number “filename” ...................................................... 75
  .undef ......................................................................... 75
Using the Compiler from the Command Line ...................... 76

Chapter 8
Structured Exception Handling ............................................... 77
  try-except Construct ......................................................... 78
    Syntax ........................................................................ 78
Related Keywords and Subprograms ................................... 80

Chapter 9
Library Subprograms.............................................................. 81
  General ........................................................................... 82
    Libraries ..................................................................... 82
    Return Values and Errors ............................................ 82
    Subprogram Names ..................................................... 82
  RAPL-II to RAPL-3 ........................................................... 84
Subprograms: Category Listing ............................................... 91
  Analog Input ..................................................................... 94
  Calibration ...................................................................... 95
  Configuration File Handling .............................................. 96
  Date and Time .............................................................. 97
  Device Input and Output ................................................... 98
  Digital Input and Output ................................................. 99
  Environment Variables .................................................. 100
  Error Message Handling ................................................ 101
  File Input and Output ..................................................... 105
    Flags ......................................................................... 105
    Field width ................................................................ 105
    .precision .................................................................. 105
    x or X ........................................................................ 105
    Unformatted Input .................................................... 105
    Formatted Input ........................................................ 106
    Unformatted Output .................................................. 106
    Formatted Output ..................................................... 106
  File and Device System Management ................................... 107
  Front Panel ..................................................................... 109
    Status Window .......................................................... 109
    Panel Button Subprograms ........................................ 110
  Gripper ........................................................................... 111
  Home .............................................................................. 112
  Location ......................................................................... 113
  Kinematic Conversion ................................................... 113
  Data Manipulation ..................................................... 113
Flags ................................................................. 113
Math ........................................................................ 114
Memory ............................................................... 115
Motion ................................................................. 116
Pendant ................................................................. 118
  Pendant Library Commands ................................. 118
Pointer Conversion and Function pointers ............. 120
Robot Configuration ............................................. 121
Signals .................................................................. 124
Stance .................................................................. 125
Status .................................................................. 126
String Manipulation .............................................. 127
System Process Control ......................................... 129
  Single and Multiple Processes .............................. 129
  Operating System Management .......................... 130
  Point of Control and Observation ....................... 130
Tool Transform and Base Offset ............................ 131
v3 Files ................................................................. 132
  Background ........................................................ 132
  Architecture for v3 Subprograms ....................... 133
  Parameters ......................................................... 133
  Subs, Funcs, and Commands ............................... 134
Win 32 ................................................................. 136
  Win 32 Commands ................................................ 136
  Types Used With Win 32 Commands .................... 136
Subprograms: Alphabetical Listing ....................... 139
  Reading Subprogram Entries ............................... 140
  Using Subprograms ............................................ 141
  abort .................................................................. 142
  accel_get .......................................................... 142
  accel_set .......................................................... 142
  accels_get ........................................................ 143
  accels_set ......................................................... 143
  access .............................................................. 144
  acos ................................................................. 145
  addr_decode ...................................................... 145
  addr_to_file ...................................................... 146
  addr_to_line ...................................................... 146
  align ............................................................... 146
  analogs_get ...................................................... 147
  app_close ........................................................ 148
  app_open ........................................................ 148
  appro .............................................................. 149
  appros ............................................................ 149
  argc ................................................................. 150
  argv ................................................................. 150
  armpower ........................................................ 151
  asin ................................................................. 151
  atan2 ............................................................... 152
  axes_get .......................................................... 152
  axes_set .......................................................... 153
  axis_status ....................................................... 153
  base_get .......................................................... 153
  base_set .......................................................... 154
  boardtemp_get .................................................. 155
  build_cloc ........................................................ 155
  build_ploc ........................................................ 156
  calibrate .......................................................... 156
malarm ................................................................. 222
maxvel_get .......................................................... 223
maxvel_set .......................................................... 223
maxvels_get ......................................................... 224
maxvels_set ......................................................... 224
mem_alloc .......................................................... 225
mem_free ............................................................. 225
memcpy ................................................................. 226
memset ................................................................. 226
memstat ............................................................... 227
MINOR ............................................................... 227
mkdir ................................................................. 227
mknod ................................................................. 228
module_name_get ................................................ 228
motor ................................................................. 229
motor_to_joint .................................................... 229
motor_to_world .................................................... 230
mount ................................................................. 230
move ................................................................. 231
moves ................................................................. 232
msleep ............................................................... 232
mtime ................................................................. 233
net_in_get ............................................................ 233
net_ins_get .......................................................... 234
net_out_set ........................................................ 234
net_outs_get ....................................................... 234
net_outs_set ....................................................... 235
nolimp ............................................................... 235
obs_get .............................................................. 235
obs_rel .............................................................. 236
onbutton ............................................................. 236
online ............................................................... 237
open ................................................................. 238
opennp ............................................................... 240
output ............................................................... 240
output_get .......................................................... 241
output_pulse ....................................................... 242
output_set .......................................................... 242
outputs .............................................................. 242
outputs_get ......................................................... 243
outputs_set ......................................................... 244
panel_button ....................................................... 244
panel_button_wait ................................................. 244
panel_buttons ..................................................... 245
panel_light_get ................................................... 246
panel_light_set .................................................... 246
panel_lights_get .................................................. 247
panel_lights_set .................................................. 247
panel_status ....................................................... 247
pdp_get ............................................................. 248
pdp_set ............................................................. 248
pendant_bell ....................................................... 248
pendant_chr_get .................................................. 249
pendant_close ..................................................... 249
pendant_cursor_pos_set ......................................... 250
pendant_cursor_set .............................................. 250
pendant_flush ..................................................... 250
pendant_home ..................................................... 251
pendant_home_clear ............................................. 251
pendant_open ............................................................251
pendant_write............................................................252
pipe ...........................................................................252
pitch ..........................................................................253
pitchs ........................................................................254
pos_axis_set...............................................................255
pos_get ......................................................................255
pos_set ......................................................................256
pow ...........................................................................256
pow ...........................................................................256
print ..........................................................................257
printf .........................................................................257
rad ............................................................................258
rand ............................................................................259
rand_in ......................................................................259
rcv.............................................................................260
read ...........................................................................261
readdir.......................................................................262
readline .....................................................................263
reads .........................................................................263
readsa ......................................................................264
ready .........................................................................265
rmdir.........................................................................265
robot_abort ................................................................265
robot_cfg_save............................................................266
robot_error_get...........................................................266
robot_flag_enable .......................................................267
robot_info ..................................................................267
robot_mode_get..........................................................267
robot_move ................................................................268
robot_odo...................................................................268
robot_servo_stat.........................................................269
robot_type_get............................................................269
robotisdone................................................................269
robotisfinished...........................................................270
robotishomed.............................................................270
robotislistening ..........................................................271
robotispowered...........................................................271
roll ............................................................................271
rolls...........................................................................272
rotacc_get ..................................................................273
rotacc_set ..................................................................273
rotspd_get..................................................................274
rotspd_set..................................................................274
seek...........................................................................274
select_menu...............................................................275
sem_acquire...............................................................276
sem_release ...............................................................276
sem_test ....................................................................277
send ..........................................................................277
server_get ..................................................................278
server_info .................................................................279
server_protocol...........................................................279
server_set ..................................................................280
server_version............................................................280
setenv ........................................................................280
setprio .......................................................................281
shift_t ........................................................................282
shift_w .......................................................................283
shutdown .....................................................................284
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>sig_arm_set</td>
<td>284</td>
</tr>
<tr>
<td>sig_mask_set</td>
<td>285</td>
</tr>
<tr>
<td>sigfio</td>
<td>285</td>
</tr>
<tr>
<td>signmask</td>
<td>286</td>
</tr>
<tr>
<td>signal</td>
<td>286</td>
</tr>
<tr>
<td>sigsend</td>
<td>287</td>
</tr>
<tr>
<td>sin</td>
<td>287</td>
</tr>
<tr>
<td>size_to_bytes</td>
<td>287</td>
</tr>
<tr>
<td>sizeof</td>
<td>288</td>
</tr>
<tr>
<td>snprintf</td>
<td>288</td>
</tr>
<tr>
<td>socketpair</td>
<td>289</td>
</tr>
<tr>
<td>speed</td>
<td>290</td>
</tr>
<tr>
<td>speed_get</td>
<td>290</td>
</tr>
<tr>
<td>speed_set</td>
<td>291</td>
</tr>
<tr>
<td>split</td>
<td>291</td>
</tr>
<tr>
<td>sqrt</td>
<td>292</td>
</tr>
<tr>
<td>srand</td>
<td>292</td>
</tr>
<tr>
<td>stance_get</td>
<td>293</td>
</tr>
<tr>
<td>stance_set</td>
<td>294</td>
</tr>
<tr>
<td>startup</td>
<td>295</td>
</tr>
<tr>
<td>stat</td>
<td>295</td>
</tr>
<tr>
<td>statfs</td>
<td>296</td>
</tr>
<tr>
<td>statusnp</td>
<td>297</td>
</tr>
<tr>
<td>str_append</td>
<td>298</td>
</tr>
<tr>
<td>str_chr_find</td>
<td>298</td>
</tr>
<tr>
<td>str_chr_get</td>
<td>299</td>
</tr>
<tr>
<td>str_chr_rfind</td>
<td>299</td>
</tr>
<tr>
<td>str_chr_set</td>
<td>300</td>
</tr>
<tr>
<td>str_cksnum</td>
<td>300</td>
</tr>
<tr>
<td>str_dup</td>
<td>300</td>
</tr>
<tr>
<td>str_edit</td>
<td>301</td>
</tr>
<tr>
<td>str_error</td>
<td>301</td>
</tr>
<tr>
<td>str_len</td>
<td>302</td>
</tr>
<tr>
<td>str_len_set</td>
<td>302</td>
</tr>
<tr>
<td>str_limit</td>
<td>303</td>
</tr>
<tr>
<td>str_limit_set</td>
<td>303</td>
</tr>
<tr>
<td>str_scanf</td>
<td>304</td>
</tr>
<tr>
<td>str_signal</td>
<td>305</td>
</tr>
<tr>
<td>str_sizeof</td>
<td>305</td>
</tr>
<tr>
<td>str_substr</td>
<td>305</td>
</tr>
<tr>
<td>str_subsys</td>
<td>306</td>
</tr>
<tr>
<td>str_to_float</td>
<td>306</td>
</tr>
<tr>
<td>str_to_int</td>
<td>306</td>
</tr>
<tr>
<td>str_to_lower</td>
<td>307</td>
</tr>
<tr>
<td>str_to_upper</td>
<td>308</td>
</tr>
<tr>
<td>sync</td>
<td>308</td>
</tr>
<tr>
<td>sysconf</td>
<td>308</td>
</tr>
<tr>
<td>sysid_string</td>
<td>309</td>
</tr>
<tr>
<td>tan</td>
<td>309</td>
</tr>
<tr>
<td>teach_menu</td>
<td>310</td>
</tr>
<tr>
<td>time</td>
<td>310</td>
</tr>
<tr>
<td>time_set</td>
<td>310</td>
</tr>
<tr>
<td>time_to_str</td>
<td>311</td>
</tr>
<tr>
<td>tool_get</td>
<td>311</td>
</tr>
<tr>
<td>tool_set</td>
<td>312</td>
</tr>
<tr>
<td>tx</td>
<td>313</td>
</tr>
<tr>
<td>txs</td>
<td>313</td>
</tr>
<tr>
<td>ty</td>
<td>314</td>
</tr>
<tr>
<td>Function</td>
<td>Page</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>tys</td>
<td>315</td>
</tr>
<tr>
<td>tz</td>
<td>316</td>
</tr>
<tr>
<td>tzs</td>
<td>316</td>
</tr>
<tr>
<td>units_get</td>
<td>317</td>
</tr>
<tr>
<td>units_set</td>
<td>318</td>
</tr>
<tr>
<td>unlink</td>
<td>318</td>
</tr>
<tr>
<td>unlock</td>
<td>319</td>
</tr>
<tr>
<td>unmount</td>
<td>319</td>
</tr>
<tr>
<td>unsetenv</td>
<td>319</td>
</tr>
<tr>
<td>utime</td>
<td>320</td>
</tr>
<tr>
<td>v3_save_on_exit</td>
<td>320</td>
</tr>
<tr>
<td>v3-vars_save</td>
<td>321</td>
</tr>
<tr>
<td>va_arg_get</td>
<td>321</td>
</tr>
<tr>
<td>va_arg_type</td>
<td>322</td>
</tr>
<tr>
<td>var_teach</td>
<td>323</td>
</tr>
<tr>
<td>vars_save</td>
<td>323</td>
</tr>
<tr>
<td>verstring_get</td>
<td>323</td>
</tr>
<tr>
<td>waitpid</td>
<td>324</td>
</tr>
<tr>
<td>WEXITSTATUS</td>
<td>325</td>
</tr>
<tr>
<td>WIFEXITED</td>
<td>325</td>
</tr>
<tr>
<td>WIFSIGNALED</td>
<td>325</td>
</tr>
<tr>
<td>world_to_joint</td>
<td>326</td>
</tr>
<tr>
<td>world_to_motor</td>
<td>326</td>
</tr>
<tr>
<td>write</td>
<td>327</td>
</tr>
<tr>
<td>writeread</td>
<td>327</td>
</tr>
<tr>
<td>writes</td>
<td>328</td>
</tr>
<tr>
<td>WTERMSIG</td>
<td>328</td>
</tr>
<tr>
<td>wx</td>
<td>329</td>
</tr>
<tr>
<td>wxs</td>
<td>329</td>
</tr>
<tr>
<td>wy</td>
<td>330</td>
</tr>
<tr>
<td>wys</td>
<td>330</td>
</tr>
<tr>
<td>wz哈尔</td>
<td>331</td>
</tr>
<tr>
<td>wz哈尔</td>
<td>331</td>
</tr>
<tr>
<td>xpulses_get</td>
<td>332</td>
</tr>
<tr>
<td>xpulses_set</td>
<td>332</td>
</tr>
<tr>
<td>xratio_get</td>
<td>333</td>
</tr>
<tr>
<td>xratio_set</td>
<td>333</td>
</tr>
<tr>
<td>xrot</td>
<td>333</td>
</tr>
<tr>
<td>xrots</td>
<td>333</td>
</tr>
<tr>
<td>yaw</td>
<td>334</td>
</tr>
<tr>
<td>yaws</td>
<td>335</td>
</tr>
<tr>
<td>yrot</td>
<td>336</td>
</tr>
<tr>
<td>yrots</td>
<td>337</td>
</tr>
<tr>
<td>zero</td>
<td>338</td>
</tr>
<tr>
<td>zrot</td>
<td>338</td>
</tr>
<tr>
<td>zrots</td>
<td>339</td>
</tr>
</tbody>
</table>

APPENDICES

Signals...........................................................................................................341
This guide is a reference manual to the RAPL-3 programming language. It contains a comprehensive description of the language including subroutines, functions, and commands in the standard libraries.

This guide is for users who have a basic understanding of RAPL-3 or a good understanding of programming concepts.
### Documentation Conventions

This guide uses the following documentation conventions.

#### Text and Programming Code

<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ready()</td>
<td>evenly spaced computer font</td>
<td>Programming code. In syntax sections, required characters that must be included.</td>
</tr>
<tr>
<td>grip_close()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>finish()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>gripdist_set(distance)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>motor(axis,pulses,c)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>if expression</td>
<td></td>
<td>User supplied item. Can be simple (integer, variable) or complex (expression, statements)</td>
</tr>
<tr>
<td>align_X</td>
<td>align_Y</td>
<td></td>
</tr>
<tr>
<td>M_READ</td>
<td>M_WRITE</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>Y</td>
<td>Z</td>
</tr>
<tr>
<td>message[2,2]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>data[10,4,7]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>grip_close([force])</td>
<td></td>
<td>Optional items in code. Can be included or omitted depending on the needs of the program.</td>
</tr>
<tr>
<td>home([axis][,axis])</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...[flags] [x</td>
<td>X]...</td>
<td></td>
</tr>
<tr>
<td>lock(7)</td>
<td>three dots on one line or on three lines</td>
<td>Omitted code of the example. A place for additional material which is not specified.</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>unlock(7)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\ (backslash)</td>
<td>character(s) with description(s) in parentheses.</td>
<td>Characters referred to in the text which need to be clearly identified.</td>
</tr>
<tr>
<td>_ (underscore)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot; (double quote)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>use with to end when here</td>
<td></td>
<td>Names of commands, functions, keywords, etc. used in the text which could be confused.</td>
</tr>
</tbody>
</table>
**Commands and Keywords**

The following documentation conventions are used for

- all subroutines, functions, and commands in libraries
- all flow control statements
- other keywords (main, return, comment, sizeof)

---

`name_of_command/keyword`

<table>
<thead>
<tr>
<th>Description</th>
<th>A description of the functionality of this subroutine, function, command, control statement, or keyword.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Caution</td>
<td>Any characteristics that could create a problem.</td>
</tr>
<tr>
<td>Syntax</td>
<td>Required characters are in non-italic monospace font. <em>Programmer-supplied identifiers and constructs are in italics.</em> Optional items are in [square brackets]. Long lines may carry over onto a second line on the printed page, but in a program must be written either on one line or with a \ (backslash) line continuation character. Subroutines, functions, and commands are given in declaration form.</td>
</tr>
<tr>
<td>Parameters</td>
<td>A list with explanations and types.</td>
</tr>
<tr>
<td>Arguments</td>
<td>Where a parameter is a standard-library defined enum or struct, the members are listed.</td>
</tr>
<tr>
<td>Returns</td>
<td>The return value of the function or command which also indicates success or error.</td>
</tr>
<tr>
<td>Example</td>
<td>An example of use in a program.</td>
</tr>
<tr>
<td>Result</td>
<td>The example's result, if applicable.</td>
</tr>
<tr>
<td>See Also</td>
<td>Any related RAPL-3 commands, functions, subroutines, statements, keywords, or topics, described in this <em>Reference Guide</em>.</td>
</tr>
<tr>
<td>System Shell</td>
<td>An equivalent command in the CROS/RAPL-3 system shell or application shell, described in the <em>Robot Systems Software Documentation Guide</em>.</td>
</tr>
<tr>
<td>Application Shell</td>
<td>Any similar RAPL-II commands.</td>
</tr>
<tr>
<td>Category</td>
<td>The category of this and related commands which are listed in the category section.</td>
</tr>
</tbody>
</table>
Related Resources

Related material can be found in these documents.

- Release notes on the diskettes.
- Robot Systems Software Documentation Guide
  A guide for developing your robotic application using all components of your
  robot system: arm, controller, teach pendant, personal computer, Robcomm3,
  RAPL-3 programs, application shell, and system shell.
- F3 Robot System Installation Guide
- A465 Arm and C500 Controller User Guides
- A255 Arm and C500 Controller User Guides
CHAPTER 1

General Program Format

All RAPL-3 programs follow the same general format. Some elements are required. Other elements are optional depending on the complexity of the program.
Example 1: Basic Program in RAPL-II Style

A basic program can contain

- only a main function

and follow a style similar to RAPL-II

- implicit declarations of variables

- familiar RAPL-II command names

```plaintext
main
    ;; begin program
    fast = 50 ;; implicitly declare and initialize integers
    slow = 25
    z = 1
    speed(fast) ;; set speed
    move(_safe) ;; move and implicitly declare cartesian location
    do
        ;; begin do loop
        appro(_a,5) ;; pick from location a, implicitly declare location
        grip_open(100)
        grip_finish()
        move(_a)
        finish()
        grip_close(100)
        grip_finish()
        depart(5)
        move(_safe) ;; move to safe location between pick and place
        appro(_b,5) ;; place at location b, implicitly declare location
        move(_b)
        finish()
        grip_open(100)
        grip_finish()
        depart(5)
        move(_safe) ;; move to safe location between place and pick
        z = z + 1 ;; increment counter in loop
        until z == 10 ;; condition to end do loop
    end main ;; end program
```
Example 2: Basic Program in Preferred RAPL-3 Style

A basic program can contain

- a main function
- a subroutine

and follow the preferred style of RAPL-3

- explicit declarations of variables, including teachables

```plaintext
sub io(int out_channel, int out_state, int in_channel)
  int in_state
  output(out_channel, out_state)
  do
    delay(250)
    input(in_channel, in_state)
  until (in_state) == 1
end sub

main

int i ;; explicitly declare variables
teachable int fast, slow, cycles ;; explicitly declare teachable variables
teachable cloc safe, a, b ;; explicitly declare teachable locations

move(safe)
speed(fast)

for i = 1 to cycles ;; use a for loop
  ;; cycles is teachable, set outside
  appro(a,5)
grip_open(100)
io(1,1,2)
speed(slow)
move(a)
grip_close(100)
depart(5)

  speed(fast)
move(safe)

appro(b,5)
io(3,1,4)
speed(slow)
move(b)
grip_open(100)
depart(5)

  speed(fast)
move(safe)
end for
end main
```
### The Main Program

Every RAPL-3 program contains a main function.

<table>
<thead>
<tr>
<th>main</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
</tr>
<tr>
<td><code>main</code> is the place in the program where execution begins.</td>
</tr>
<tr>
<td><strong>Syntax</strong></td>
</tr>
<tr>
<td><strong>Returns</strong></td>
</tr>
</tbody>
</table>
| **Example** | ```
main
teachable cloc pick, place
move(pick)
grip_close()
move(place)
grip_open()
end main
``` |
| **RAPL-II** | RAPL-II did not have a function or structure similar to `main`. RAPL-II's STOP command had a purpose similar to `end main`. |
Lines of a Program

A RAPL-3 program consists of a number of lines of ASCII text. Statements and declarations are terminated by the line end.

Line Continuation
To continue on the next line, end a line with the \ (backslash) character. For example

\[a = \ b + c + d \ \backslash
+ e + f\]

is read as one statement.

Without the continuation character

\[a = \ b + c + d
+ e + f\]

the first part of the statement ends at the end of the first line and is read as a statement. The second part is a fragment which causes a syntax error when compiling.

Lines that end with , (a comma) are automatically considered to be continued. For example,

\[\text{printf("The coordinates are {}, {}, {}
}, x, y, z)\]
Comments

A comment starts with ;; (two semicolons) and extends to the end of the line. A comment can start at the beginning of a line or after some program code. For example:

```plaintext
;; calculate the position error:
x_error = x_pos - desired_x_pos ;; for the x-axis
y_error = y_pos - desired_y_pos ;; for the y-axis
z_error = z_pos - desired_z_pos ;; for the z axis
```
Labels

A statement can be marked with a special identifier called a label. The label has :: (two colons) after the identifier. A labels is used as the target of a goto statement.

Syntax

\[
\text{label_identifier:: statement}
\]

where

\[
\text{label_identifier} \quad \text{is the name of the label and follows the rules for identifiers,}
\]

\[
\text{and}
\]

\[
\text{statement} \quad \text{is the statement line being labelled.}
\]

The statement can be an empty line.

Examples

\[
\text{my_label:: current_location = num}
\]

\[
\text{start_again::}
\]
Keywords

The following identifiers are keywords of RAPL-3. They are reserved for the RAPL-3 language and cannot be redefined. In particular, the following keywords cannot be used as the name of any variable, subroutine, function, or command:

and       glocl       sizeof
break     goto       static
_builtin   if         step
case       ignore     string
cloc       import     struct
command    int        sub
cmt       libversion  teachable
const      loop       then
continue   main       to
do         mod        try
else        not       typedef
elseif      of        union
end         or        unteachable
enum       ploc       until
except      private    var
export      proto      void
float       raise      volatile
for         resume     while
func        return     with
global     retry
CHAPTER 2

Data Types and Variables

RAPL-3 programs can work with many different types of data and also permits user-defined data types. This chapter presents the basic data types supported by RAPL-3, and goes on to look at the kinds of user-defined types that can be constructed.
**Basic Data Types**

RAPL-3 supports the following basic data types.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>32-bit signed integer (Range: -2147483648 to +2147483647)</td>
<td>4</td>
</tr>
<tr>
<td>float</td>
<td>IEEE single precision floating point (Range: ±1.7 x 10±38)</td>
<td>4</td>
</tr>
<tr>
<td>string</td>
<td>variable length string (Range: 0 to 65535 8-bit characters)</td>
<td>4 + number of characters</td>
</tr>
<tr>
<td>cloc</td>
<td>cartesian location</td>
<td>36</td>
</tr>
<tr>
<td>ploc</td>
<td>precision location</td>
<td>36</td>
</tr>
<tr>
<td>void</td>
<td>used for forming generic pointers</td>
<td>—</td>
</tr>
</tbody>
</table>

**int**

An int, or integer, is a signed number without any decimal or fractional part. Examples: 0, 1, 23, 456, -7, -89

**float**

A float, or floating point number, is a number with a decimal or fractional part and an optional exponent. A float has up to seven significant digits. Examples: 4.75, -99.99, 1.0, 3.141593, 1.0e10

**string**

A string is a set of characters: uppercase or lowercase letters, digits, punctuation and other graphic characters, and the blank space. In a string, a digit is a character and does not have numeric value as it does in a number (int or float). RAPL-3 does not have a character data type.

**cloc**

A cloc, or cartesian location, represents a point in the robot arm workspace defined by cartesian co-ordinates. Coordinates have three translational elements (along axes) x, y, and z, and three rotational elements (around axes) z, y, and x. The values of a cloc are independent of arm position and arm type.

**ploc**

A ploc, or precision location, represents a point in the robot arm workspace defined by increments of rotational movement, specifically encoder counts, of each joint of the arm and any additional axes (j1, j2, j3, j4, j5, j6, j7, j8). The values of a ploc are dependent on the robot.
The **void** type is used to form void pointers (pointers that can point to any type).

```
void* x
```

Void pointers are assignment compatible with all other types of pointers.
Identifiers

An identifier is used for the name of a variable, type, subroutine, function, or command.

Character Set

An identifier begins with a letter. This may be followed by zero or more letters, digits, or _ (underscore) characters.

```
abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789
_  
```

Case

Letters may be either uppercase (ABCDE), lowercase (abcde), or mixed (AbCdE). RAPL-3 is case-sensitive with identifiers. For example, the following are all different identifiers.

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>lowercase</td>
</tr>
<tr>
<td>X</td>
<td>uppercase</td>
</tr>
<tr>
<td>symbol</td>
<td>lowercase</td>
</tr>
<tr>
<td>SYMBOL</td>
<td>lowercase</td>
</tr>
<tr>
<td>sYmBoL</td>
<td>mixed</td>
</tr>
<tr>
<td>SyMbOl</td>
<td>mixed</td>
</tr>
</tbody>
</table>

Length

An identifier may be any length, but only the first 32 characters are significant. For example, the following are not different identifiers.

- `location_sensor_data_collection_1`
- `location_sensor_data_collection_2`

Examples

There are many possibilities of valid identifiers.

**Valid**

- `a` ;; a single letter
- `num` ;; several letters
- `my_symbol` ;; letters with underscore
- `MySymbol` ;; letters of different cases
- `x3` ;; letter with digit
- `rack_loc_12` ;; letters, underscores, digits

**Invalid**

- `3a` ;; begins with a digit, not a letter
- `my$symbol` ;; uses a character not in the valid character set
- `&num` ;; uses a character not in the valid character set
  ;; and does not begin with a letter
Declarations

This section details the declaration of: int, float, string, cloc, and ploc. For the declaration of arrays of these types, see the Arrays section. For const, see the Initializers section.

Each variable must be declared as one specific type of variable (int, float, string, cloc, ploc, const). A declaration states the type of variable and the name of the variable.

You can declare a variable explicitly or implicitly. It is good programming practice to explicitly declare all variables.

Explicit Declarations

When you declare a variable explicitly, you list it in a declaration statement before you use it in the program.

Variables being declared as the same type can be listed in the same declaration, separated by commas.

Syntax

\[
\text{type} \quad \text{identifier} \\
\text{type} \quad \text{identifier}, \text{identifier}, \text{identifier} \ldots
\]

where

- \text{type} is the data type, and
- \text{identifier} is the name of the variable and follows the rules for identifiers.

Examples

<table>
<thead>
<tr>
<th>Type</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>int i</td>
<td>i is an integer</td>
</tr>
<tr>
<td>float</td>
<td>float a,b</td>
<td>a and b are floats</td>
</tr>
<tr>
<td>string</td>
<td>string[10] message</td>
<td>message is a string that can hold 10 or fewer characters</td>
</tr>
<tr>
<td>cloc</td>
<td>cloc pick_1, place_1</td>
<td>pick_1 and place_1 are cartesian locations</td>
</tr>
<tr>
<td>ploc</td>
<td>ploc pick_2, place_2</td>
<td>pick_2 and place_2 are precision locations</td>
</tr>
</tbody>
</table>

Implicit Declarations

When you declare a variable implicitly, you indicate the variable’s type with a prefix before its name when you use it in the program for the first time.

If a variable is used without having been explicitly declared, the compiler looks for an implicit declaration prefix character on the variable name to determine the type of variable. If there is no prefix character, the compiler defines the variable as the default type, an int, and issues a warning.

In general, implicit declarations should be avoided. You should always explicitly declare variables.
Syntax

\[\text{[prefix\_character]}\text{identifier}\]

where

- \text{prefix\_character} is the character indicating the data type, and
- \text{identifier} is the name of the variable and follows the rules for identifiers.

**Implicit Declaration Prefix Characters**

<table>
<thead>
<tr>
<th>Prefix Character</th>
<th>Type</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>int</td>
<td>a = 2</td>
</tr>
<tr>
<td>%</td>
<td>float</td>
<td>%b = 10.25</td>
</tr>
<tr>
<td>$</td>
<td>string[64]</td>
<td>$m = &quot;Robot working.\n&quot;</td>
</tr>
<tr>
<td>_</td>
<td>cloc</td>
<td>here _z</td>
</tr>
<tr>
<td>#</td>
<td>ploc</td>
<td>here #y</td>
</tr>
</tbody>
</table>

**Examples**

<table>
<thead>
<tr>
<th>Type</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>e = c + d</td>
<td>e is defined as an int, if it has not been seen before.</td>
</tr>
<tr>
<td>float</td>
<td>%h = f * g</td>
<td>h is defined as a float.</td>
</tr>
<tr>
<td>string</td>
<td>$\text{notice9} = &quot;\text{stop}&quot;</td>
<td>notice9 is defined as a string[64].</td>
</tr>
<tr>
<td>cloc</td>
<td>here(_place22)</td>
<td>place22 is defined as a cloc.</td>
</tr>
<tr>
<td>ploc</td>
<td>here(#material33)</td>
<td>material33 is defined as a ploc.</td>
</tr>
</tbody>
</table>

**Implicit with Explicit**

If an implicit declaration prefix is used in an explicit declaration statement, the implicit prefix is ignored by the compiler. For example,

```
float \%b ;; the variable b is declared as a float
float $c ;; the variable c is declared as a float
float #d ;; the variable d is declared as a float
```

**Identifiers**

The prefix character indicates the type of declaration. It is not part of the identifier, the variable's name. For example, if _m was used in a statement, a cloc with the name m was defined. A later statement with #m causes an error, the same way that cloc m followed by ploc m causes an error.

**Scope**

Two variables with the same scope cannot have the same name. For definitions of scope, see the Scope section of the Subprogram chapter.

**Teachables**

Teachable variables that are declared inside a sub, func, or command must not have the same name as any teachable outer-frame variable.
Strings

The string type is essentially a character array with a fixed size.
The string type must always have a subscript, indicated by [ ] (square brackets).

String[number]

Usually, the subscript contains a number to specify the maximum length of
string that can be stored in it, such as string[10] or string[64].

Syntax

\[ \text{string[number] identifier} \]

where

- string and the square brackets are required,
- number is the character size of the string, and
- identifier is the name of the variable and follows the rules for identifiers.

String[]

In some circumstances, the subscript can be empty.

\[ \text{string[]} \]

This undimensioned string declaration can be used only in the following circumstances.

- A simple single string being initialized. When string[] is used, the compiler
determines the size of the string. In this example, the compiler makes notice9
a string[18].

\[ \text{string[]} \text{ notice9 = “End of work cycle.”} \]

- A function formal parameter or var parameter.

\[ \text{func int strlen(string[])} \]
\[ \text{sub str_append(var string[] dst, string[] src)} \]

- The target of a pointer.

\[ \text{string[]} @ \text{ sptr} \]

For a table of pointers to strings of unknown length, use

\[ \text{string[]} @ [5] \text{ greek = {“alpha”, “beta”, “gamma”, “delta”, “epsilon”}} \]

Notes:

A RAPL-3 string is actually stored as a length, a limit, and an array of
characters. The length value indicates how many characters are actually valid.
Strings can be created with at most space for 65,532 characters. The limit value
indicates how many characters there is actually room for. For example, if we
have a variable:

\[ \text{string[10]} \text{ s} \]
then s is initially created with its length set to 0 (no characters; the empty string)
and its limit set to 12. The limit is 12 because RAPL-3 always allocates storage in
units of 1 word (or 4 characters); string[10] actually needs 1 word for the length
and limit, and an additional 3 words for the characters (which actually is 3 * 4 or
12 characters in size.) After this statement:

\[ \text{s = “hello!”} \]
the length of s is set to 6, and the characters 'h', 'e', 'l', 'l', 'o' and '!' have been stored in the character part of the string.

**Termination**

RAPL-3 does not use any string termination character. The variable is declared and the string of characters is packed into the variable.

**Concatenation**

To concatenate (link together to form a longer string), use the str_append subroutine with string variables. The + (plus) operator can be used to concatenate string constants.
Arrays

An array is a collection of data objects where all are the same data type and all use the same identifier but each has a unique subscript.

Syntax

\[ base\_type[\ subscript\_list\ ]\ identifier \]

where

- \( base\_type \) is the data type of each element in the array,
- \( subscript\_list \) is a comma-separated list of one or more constant expressions defining each dimension, and
- \( identifier \) is the name of the variable and follows the rules for identifiers.

A subscript must be a constant expression, such as a simple integer constant. The compiler must be able to compute the value of each constant expression at compile time.

Types

You can have an array of any type or an arrays of arrays.

Dimensions

There is no limit on the number of dimensions allowed, except for teachable arrays. See Teachables.

Numbering

In RAPL-3, numbering begins with 0.

<table>
<thead>
<tr>
<th>Declaration</th>
<th>Number of Elements</th>
<th>Numbering</th>
</tr>
</thead>
<tbody>
<tr>
<td>int[4] a</td>
<td>4</td>
<td>a[0], a[1], a[2], a[3]</td>
</tr>
<tr>
<td>int[10] a</td>
<td>10</td>
<td>a[0], a[1], a[2], a[3], a[4], a[5], a[6], a[7], a[8], a[9]</td>
</tr>
<tr>
<td>int[20] a</td>
<td>20</td>
<td>a[0], a[1], a[2], a[3], a[4], a[5], a[6], a[7], a[8], a[9], a[10], a[11], a[12], a[13], a[14], a[15], a[16], a[17], a[18], a[19]</td>
</tr>
<tr>
<td>int[100] a</td>
<td>100</td>
<td>a[0], a[1], a[2], a[3], a[4], a[5], a[6], a[7], a[8], a[9], through to a[90], a[91], a[92], a[93], a[94], a[95], a[96], a[97], a[98], a[99]</td>
</tr>
</tbody>
</table>

Review of Strings

<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string[30] z</td>
<td>a string that can hold 30 or fewer characters</td>
</tr>
</tbody>
</table>

One Dimensional Arrays

<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int[5] a</td>
<td>an array of 5 integers a[0], a[1], a[2], a[3], a[4]</td>
</tr>
<tr>
<td>float[10] b</td>
<td>an array of 10 floats b[0], b[1], b[2], ... b[9]</td>
</tr>
<tr>
<td>ploc[20] c</td>
<td>an array of 20 precision locations c[0], c[1], c[2], ... c[19]</td>
</tr>
<tr>
<td>string[30] [10] d</td>
<td>an array of 10 strings d[0], d[1], d[2], ... d[9] each can hold 30 or fewer characters</td>
</tr>
</tbody>
</table>
## Two Dimensional Arrays

<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
</table>
| int[5,10] e | a 2-dimensional array of 50 integers  
|            | e[0,0] ... e[0,9]  
|            | ...  
|            | e[4,0] ... e[4,9]                                                    |
| float[10,20] f | a 2-dimensional array of 200 floats  
|            | f[0,0] ... f[0,19]  
|            | ...  
|            | f[9,0] ... f[9,19]                                                    |
| ploc[5,10] g | a 2-dimensional array of 50 precision locations  
|            | g[0,0] ... g[0,9]  
|            | ...  
|            | g[4,0] ... g[4,9]                                                    |
| string[20][5,10] h | a 2-dimensional array of 50 strings  
|            | h[0,0] ... h[0,9]  
|            | ...  
|            | h[4,0] ... h[4,9]  
|            | each can hold 20 or fewer characters                                  |
| int[10] [5] i | a 2-dimensional array of 50 integers  
|            | same as int[5,10] e  
|            | brackets are applied from left to right                               |
| float[20][10] j | a 2-dimensional array of 200 floats  
|            | same as float[10,20] f  
|            | brackets are applied from left to right                               |
| string[20] [10] [5] k | a 2-dimensional array of 50 strings  
|            | same as string[20][5,10] h                                          |
| string[50][23 + 7] m | an array of 30 strings,  
|            | each can hold 50 or fewer characters                                  |

## Multi Dimensional Arrays

<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
</table>
| int[2,2,2] n | a 3-dimensional array of integers  
|            | n[0,0,0], n[0,0,1],  
|            | n[0,1,0], n[0,1,1],  
|            | n[1,0,0], n[1,0,1],  
|            | n[1,1,0], n[1,1,1]                                                    |
| float[5,5,5,5] p | a 4-dimensional array of integers  
|            | p[0,0,0,0] to p[4,4,4,4]                                               |

## Declarations

You cannot implicitly declare an array.

However, if you use the implicit declaration syntax in a statement with an array, you will not cause a problem, if the array is previously declared and the implicit declaration character matches the base type of the array. For example:

```plaintext
ploc[16,16] a  
...  
here(#a[1,1])
```
Teachables

A variable that is teachable is accessible from outside the program.

Use

Teachables provide an easy way, outside the program, to modify a value for a variable, store that value, and use the value in a program. Using this feature avoids writing (hard-coding) values in the program and having to re-write the program to change the values. It also avoids storing the values in a custom user-designed file and having to carefully edit the file to change values and include a routine in the program to read that custom data file.

Data about teachable variables and their values are stored in the variable file. When you run a program, the operating system takes the program's variable file and uses its values to initialize the variables in the program just before running.

Variable (v3) File

Data about teachable variables are stored in the variable file (also known as a v3 file). You modify data, or “teach” locations and other variables, using the teach pendant or the application shell.

You can create a variable file in a number of ways.

- Refreshing from the Program. When your program file is in a CROS directory (in CROS-500 or CROSnt), ash’s refresh command reviews the program and adds any teachable variables of the program to ash’s database. After assigning values (including teaching locations) to the teachables in the database, this new data is saved to the variable file. This method is used if you write your program before teaching your locations.

- Building Independently. You can build a variable file completely in a CROS directory (in CROS-500 or CROSnt) using ash or the teach pendant. With ash’s or the teach pendant’s database, you create variables and assign values to them. When you are finished this data is saved to in the variable file. This method is used if you teach your locations before writing your program.

See the Robot System Software Documentation Guide chapters on the application shell.

Declarations

You make a variable teachable by adding the keyword “teachable” before the data type at declaration. Teachables are not initialized.

Syntax

```plaintext
teachable  type  identifier
 teachable  type  identifier, identifier, identifier  . . .
```

where

- `teachable` is a necessary keyword
- `type` is the data type, and
- `identifier` is the name of the variable and follows the rules for identifiers.
Examples

<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>teachable int cycles</td>
<td>cycles is a teachable integer</td>
</tr>
<tr>
<td>teachable float a, b, c</td>
<td>a, b, and c are teachable floats</td>
</tr>
<tr>
<td>teachable string[10] note</td>
<td>note is a teachable string that can hold 10 or fewer characters</td>
</tr>
<tr>
<td>teachable cloc pick_1, place_1</td>
<td>pick_1 and place_1 are teachable cartesian locations</td>
</tr>
<tr>
<td>teachable ploc pick_2, place_2</td>
<td>pick_2 and place_2 are teachable precision locations</td>
</tr>
<tr>
<td>teachable int[3] step</td>
<td>step is a teachable array of 3 integers: step[0], step[1], step[2]</td>
</tr>
<tr>
<td>teachable float[5,5] delta</td>
<td>delta is a teachable two-dimensional array of floats: delta[0,0] ... delta[4,4]</td>
</tr>
<tr>
<td>teachable ploc[2,10] spot</td>
<td>spot is a teachable two-dimensional array of precision locations: spot[0,0] ... spot[1,9]</td>
</tr>
</tbody>
</table>

Limitations

Data Types
There are limits on which data types are teachable. Simple, scalar variables can be teachable. One-dimensional arrays of variables can be teachable. Two-dimensional arrays, except string[n], can be teachable. Three-dimensional and higher dimensional arrays cannot be teachable. The void type cannot be teachable.

✓ = can be teachable
✗ = cannot be teachable

<table>
<thead>
<tr>
<th></th>
<th>int</th>
<th>float</th>
<th>string[n]</th>
<th>cloc</th>
<th>ploc</th>
<th>gloc</th>
<th>void</th>
</tr>
</thead>
<tbody>
<tr>
<td>simple</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>one-dimensional array</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>two-dimensional array</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>three-dimensional or higher array</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>

Not Initialized
A variable cannot be both teachable and initialized. You cannot write

teachable int a = 5
teachable string[64] message_12 = "Error recovery underway.".
Storage Class: Static
Variables which are declared as teachable are static. They should not be used in recursive routines except as read only.

Defaults and Unteachables

Scope and Declaration Defaults
The following variables are teachable by default.

Local (within a subprogram or main) and Implicitly Declared
- clocs, and plocs
  sub
  ...
  here(_point)
  end sub

main
...
  here(_place)
  end main

Outer-Frame (outside all subprograms and main) and Explicitly Declared
- clocs, and plocs
- 1-dimensional and 2-dimensional arrays of clocs, and plocs
  ploc start_point
cloc[10] point

sub
...
end sub

main
...
end main

All other variable types are unteachable by default.

Unteachable Declaration
A variable can be declared as unteachable with the unteachable keyword. This can be used to make an outer frame location that is not teachable, for example

unteachable cloc[10] point

sub
...
end sub

main
...
end main
User-Defined Types

A type can be called by a user-specified name. Typedefs can only be global, imported, or outer-frame. There are no local typedefs. Typedefs within a subprogram are available to sections outside of that subprogram.

Syntax

```plaintext
typedef  identifier  type
```

where

- `typedef` is required,
- `identifier` is the name of the type and follows the rules for identifiers, and
- `type` is the keyword indicating the data type.

Examples

<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
</table>
| typedef alpha int[10] ... alpha a,b,c .. alpha[3] x | alpha is an array of 10 ints  
  a, b and c are all int arrays  
  a[0], a[1], a[2],...a[9]  
  b[0], b[1], b[2],...b[9]  
  c[0], c[1], c[2],...c[9]  
  x is an array of 3 alphas  
  x[0,0], x[0,1], x[0,2],...x[0,9]  
  x[1,0], x[1,1], x[1,2],...x[1,9]  
  x[2,0], x[2,1], x[2,2],...x[1,9] |
Pointers

A pointer is a variable that holds the address of another variable. A pointer is declared to point to a specific data type.

Syntax

\[ \text{basetype}@ \text{ identifier} \]

where

- \textit{basetype} is the keyword indicating the data type.
- @ is required and immediately follows the basetype, and
- \textit{identifier} is the name of the type and follows the rules for identifiers.

Examples

<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int@ a</td>
<td>a is a pointer to an int</td>
</tr>
<tr>
<td>float@ b, c</td>
<td>b and c are pointers to floats</td>
</tr>
<tr>
<td>string[20]@ d</td>
<td>d is a pointer to a 20 character string</td>
</tr>
<tr>
<td>cloc@ e</td>
<td>e is a pointer to a cloc</td>
</tr>
<tr>
<td>int[10]@ f</td>
<td>f is a pointer to an array of 10 ints</td>
</tr>
<tr>
<td>int[3,2]@[4] g</td>
<td>g is an array of 4 pointers, each of which points to a two-dimensional array of ints</td>
</tr>
</tbody>
</table>

Note that in all cases, complex declarations are applied from left to right.

Dereferencing

Pointers can be dereferenced with the @ operator. For example, if the variable xp is of type int@, then xp@ refers to the value that the pointer xp points to.

Address-of Operator

A pointer to a data object can be constructed using the ‘&’ (address-of) operator. For example, if x is an integer, then &x is an int@ which points to the value of x.
Enumerated Types

It is often convenient to refer to the values of a variable by name, rather than by number. For example, when referring to the colour of a test-tube, we can define:

```plaintext
class colour_type
enum
    red,
    orange,
    yellow,
    green,
    blue
end enum
```

This defines type `colour_type` as type `int`, and creates the special constants `red`, `orange`, `yellow`, `green` and `blue`, which will have values 0, 1, 2, 3, and 4, respectively. These constants can be used anywhere a numerical constant would be appropriate.

This allows a particular value to be associated with an identifier in the list.

**Syntax**

```plaintext
enum
    item_list
end enum enum_identifier
```

where
- `enum` and `end enum` are required,
- `enum_identifier` is the name of the enum,
- and `item_list` is a comma-separated list of items, where each item can be a simple identifier `identifier` or a statement `identifier = constant_expression`

**Examples**

<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
</table>
| ```
enum
    num_a,
    num_b,
    num_c
end enum x
``` | x is an int num_a is the constant 0 num_b is the constant 1 num_c is the constant 2 |
| ```
enum
    bit_0 = 1,
    bit_1 = 2,
    bit_3 = 4
end enum y
``` | y is an int bit_0 is the constant 1 bit_1 is the constant 2 bit_3 is the constant 4 |
| ```
enum
    x,
    y,
    z
end enum letters
``` | This is illegal after the previous two declarations. The constant identifiers must be unique within the same scope. |
Record Structures

Records structures (like structs in C) are declared as:

\[
\text{struct} \\
\quad \text{field_list} \\
\text{end struct}
\]

Where \text{field_list} is a list of 1 or more entries of the following form. Struct fieldnames can be anything except a type name.

\[
\text{type identifier_list}
\]

For example:

\[
\text{typedef Colour struct} \\
\quad \text{float red, green, blue} \\
\text{end struct} ;; \text{declares a type called} \\
;; \text{Colour with fields called} \\
;; \text{red, green and blue}
\]

\[
\text{typedef my_record struct} \\
\quad \text{int i} ;; \text{values in a linked list} \\
\quad \text{my_record@ next} ;; \text{a pointer to this structure} \\
;; \text{itself, for creating a} \\
;; \text{linked list} \\
\text{end struct}
\]
Unions

Unions (like unions in C) are possible.

    union
        field_list
    end union

Where field_list is a list of declarations which can include int, float, string[], cloc, ploc, or a complex type like struct or union.

    union
        int a
        float b
    end union xxx

    typedef omega union
        int a
        float b
    end union

Unions are referenced like structures, but have one important difference. All of the fields of a structure are located in distinct locations in memory, allowing all fields of a structure to hold values at the same time. However, in unions, all fields are located at the same memory location. Hence in variable xxx above, writing into field a of the union also alters the value of field b. Unions are typically used where a block of information may hold more than one kind of data.
Initializers

You can declare RAPL-3 variables and initialize their values at the same time. Initialization is useful for building tables of data needed by a program during its execution.

The general format of a declaration with an initializer is:

\[
\text{type identifier} = \text{initializer_expression}
\]

For simple variables, \text{initializer_expression} is a simple constant expression.

More complex variables can also be initialized, as shown in the examples below. Array and structure initializers are delimited by \{ \} (braces). Note the use of \{ \} (braces) for constructing each dimension of an array and the contents of each structure. Initializers must exactly match the size of the variable being initialized.

\begin{verbatim}
int a = 3 ; ; a is an int
       ;; with initial value 3
int a = 3, b = 4, c = 5 ; ; a, b, and c are ints
       ;; with initial values 3, 4, and 5
       ;; respectively
float d = 2.0 ; ; d is a float
       ;; with initial value 2.0
int[2] e = { 0, 1 } ; ; e is an array of ints
       ;; e[0] = 0 and e[1] = 1
string[16][3] f_string = { "No error(s)",
                        "Warning error(s)",
                        "Fatal errors(s)"
                   } ; ; f_string is an array of 3 strings
                        ;; f_string[0] contains No error(s)
                        ;; f_string[1] contains Warning error(s)
                        ;; f_string[2] contains Fatal error(s)

struct
   int a
   float b
end struct stv = { 1, 2.7182 }
       ;; stv is an initialized struct

float[2,3] fa = {
   { 1.0, 2.0, 3.0 },
   { 2.0, 3.0, 4.0 } \n
} ; ; two dimensional array initialization
\end{verbatim}

The compiler accepts initializers like:

\begin{verbatim}
string[]@[2] list = { "yes", "no" }
\end{verbatim}

and correctly generates the required data structures, but does not accept:

\begin{verbatim}
int@[2] list2 = { 1, 2 }
\end{verbatim}

For initializing clocs and plocs with cloc[] and ploc[], see the Location Constant section of the Constants chapter.

An initialized entity cannot be teachable.
Named Constants

It is frequently useful to be able to define a named constant in a program. RAPL-3 provides a `const` keyword for this purpose. The format of a constant definition is:

```
const identifier = value
```

Note that it is not necessary to specify a type for a `const` definition; the compiler is able to deduce what type you are referring to by looking at the specified value. Examples of `const` definitions are:

```
const x = 123 ;; an integer constant
const y = 10.3 ;; a floating point constant
const z = "hello" ;; a string constant
```

Only integer, floating point and string constants may be defined in this way. You may use a named constant anywhere it would be legal to use the actual constant itself. For example, if the following definitions are in your program, then this section of code:

```
print("hello", 123, 10.3)
```

is exactly the same as this section of code:

```
print(z, x, y)
```

Typically, named constants are used for setting configurable values in a program. For example, if a robot program rinses a dispense head some number of times in between operations, one might have a const definition like this at the top of the program:

```
const NUMBER_OF_RINSES = 3
```

This way the behaviour of the program can be changed by just changing the constant, and code that refers to this number can use `NUMBER_OF_RINSES`, which is much more obvious than just ‘3’.
Sizeof() Function

The sizeof() function determines the size of a type or a variable. The size of any type (even complex types) can be determined. As a built-in, sizeof is a keyword.

sizeof()

Returns the number of words that the type or variable occupies. (Note that 1 word = 4 bytes = 32 bits.) Used to determine the size of a type or variable.

Syntax

sizeof( type )

sizeof( variable )

Arguments

type a data type

variable any variable

Returns

Returns an integer of the number of words occupied.

Example

int ia = 1, ib = 9999
string[] sa = "a", sb = "Characters in this string are 32"
struct
    float red, orange, yellow
    int green, blue, violet
string[50] brown, black
end struct color
print("int size is ", sizeof(int), "\n")
print("ia size is ", sizeof(ia), "\n")
print("ib size is ", sizeof(ib), "\n")
print("string[] size is ", sizeof(string[]), "\n")
print("sa size is ", sizeof(sa), "\n")
print("sb size is ", sizeof(sb), "\n")
print("color size is ", sizeof(color), "\n")

Result

int size is 1
ia size is 1
ib size is 1
string[] size is 1
sa size is 2
sb size is 9
color size is 34

See Also

sizeof_str number of words to store a string
str_len number of characters in a string
Dimof() Function

**dimof()**

**Description**
Returns the dimensionality of an array.

**Syntax**
dimof(array)

**Parameters**
array   name of array

**Example**

```plaintext
int [20] x
int [5,10] z
print ("dimensionality of x is ", dimof(x), "\n")
print ("dimensionality of z is ", dimof(z), "\n")
print ("dimensionality of z[3] is ", dimof(z[3]), "\n")
```

**Result**
dimensionality of x is 20
dimensionality of z is 5
dimensionality of z[3] is 10
Consider the following short RAPL-3 program:

```plaintext
[1] main
[2]   int x
[3]   x = 1
[4]   while (x <= 10)
[5]     printf("x = {}\n", x)
[6]     x = x + 1
[7]   end while
[8] end main
```

This program counts from 1 to 10, printing out the value of x each time through the `while` loop (see chapter 5 for more information about `while` loops.)

This short example has 4 expressions, 5 variable references and 2 assignment statements.

An expression is a part of a program statement that calculates a value. The following are the expressions in the above example:

1

x <= 10

x

x + 1

A variable reference is just a point in the program that refers to the value of a variable or stores a value in a variable. In the above program, there are 2 places where the value of x is modified or assigned (lines 3 and 6) and 3 places where the value of x is used (lines 4, 5 and 6).

An assignment statement is one that changes the value of a variable. Once again, this happens at lines 3 (where the value of x is set to 1) and 6 (where the value of x is incremented.)

This chapter presents the basic form of a variable reference, looks at how assignment statements are constructed and discusses the operators (like +, -, etc.) that are available for constructing expressions.
Variable References

Variable references have the form:

```
variable_name [ modifiers ]
```

Valid modifiers are:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>[</td>
<td>index-list</td>
</tr>
<tr>
<td>.</td>
<td>fieldname</td>
</tr>
<tr>
<td>@</td>
<td>pointer de-referencing</td>
</tr>
</tbody>
</table>

Variable references are read strictly from left-to-right, and modifiers are applied in that order.

```;
;; declarations for these examples
int i, j ;; an integer
float[10,10] a ;; 2-dimensional array of floats
int@[100] api ;; a 100-element array of pointers to ints
int[100]@ bpi ;; a pointer to a 100-element array of ints
struct ;; st is a simple struct
  int a
  string[] s
end struct st

;; variable references
```

```;
... j ... ;; the variable j
... a[i,j] ... ;; element [i,j] of array a
... api[j]@ ... ;; what is pointed to by the jth pointer in the array api
... bpi@[i] ;; the ith integer in the array that is pointed to by bpi
... st.s ... ;; the string part of struct st
```

Note that because variable modifiers are applied strictly from right to left, the use of a variable resembles the reverse of its declaration; for example, bpi is declared as “int[100]@ bpi” and is used as “bpi@[whatever]”.

Assignment statements

An assignment statement allows the value of a variable to be modified and has the form:

\[ \text{variable} = \text{expression} \]

or

\[ \text{variable} \ \text{simple-op} = \text{expression} \]

Where \text{simple-op} is a simple binary operator like +, -, *, etc. This second form of an assignment statement is interpreted to mean:

\[ \text{variable} = \text{variable} \ \text{simple-op} \ \text{expression} \]

This allows statements like “a = a + 5” to be written more compactly, as “a += 5”.

In addition, the special operators

\[ ++ \quad -- \]

can be used as assignment operators to increment and decrement the value of a variable. For example,

\[ x++ \]

is a shorthand way of saying

\[ x = x + 1 \]

The ++ and -- operators may not be used inside an expression. Constructs like

\[ a = b++ \]

are not allowed.

You can assign an integer variable a floating-point value. For example

```c
int i
i = 1.6
```

In this case, the value is truncated back to an integer, and \( i \) is assigned the value 1. The compiler warns of float to int truncation (unless warnings are disabled).

Void pointers are assignment compatible with all other kinds of pointers.

All other types (string, ploc, clon, arrays and structs) must match exactly for an assignment statement to be legal. For example:

```c
int i, j ;; some variable definitions
int @ip
float a,b
float@ fp
int[100] x,y
string name1,name2
void @vp
...
i = j ;; these are all legal
a = b
a = i
i = a
x = y
name1 = name2
vp = ip
fp = vp
x = name1 ;; these are not legal
y = i
fp = ip
```
Operators

The following operators are supported, and are listed in order of increasing precedence. Within one level of precedence, operators are left-associative.

In the table, the Form column indicates whether the operator is a binary operator ("a op b") or a unary operator ("op a"). The Accepts column lists the type of arguments the operator accepts (I = integer, F = float, S = string, P = ploc, C = cloc, @ = pointer), and the Yields column lists the type of result the operator produces. Note that the special character T denotes a value that is either integer 0 or 1, and L denotes anything which can reasonably appear on the left-hand-side of an assignment statement.

In cases where a binary operator has operands of different types, RAPL-3 will at most promote an int operand to float. If the types still do not match, the compiler will signal a type mismatch error. The one exception to this rule is that pointers may be compared for equality with zero.

Care must be taken in the use of mixed types. For example:

```
int  i ;; variable declarations
float f
... i/2 ... ;; gives an integer result
... f/2 ... ;; gives a floating point result
... i/f ... ;; gives a floating point result
... f/i ... ;; gives a floating point result
```

Sub, func, and command parameters are also checked for type match. As for expressions, arguments can be automatically converted from int to float. Also, cloc and ploc parameters can be automatically converted to glocs.

It is legal to compare pointers to 0 (NULL). It is also legal to compare pointers of the same type, and pointers of any type to void pointers.
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Form</th>
<th>Accepts</th>
<th>Yields</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>, or</td>
<td>binary</td>
<td>IF@</td>
</tr>
<tr>
<td>&amp;&amp;, and</td>
<td>binary</td>
<td>IF@</td>
<td>T</td>
<td>logical AND</td>
</tr>
<tr>
<td></td>
<td></td>
<td>binary</td>
<td>I</td>
<td>I</td>
</tr>
<tr>
<td>^</td>
<td>binary</td>
<td>I</td>
<td>I</td>
<td>bitwise boolean exclusive-OR</td>
</tr>
<tr>
<td>&amp;</td>
<td>binary</td>
<td>I</td>
<td>I</td>
<td>bitwise boolean AND</td>
</tr>
<tr>
<td>==</td>
<td>binary</td>
<td>IFS@</td>
<td>T</td>
<td>is equal to</td>
</tr>
<tr>
<td>!=</td>
<td>binary</td>
<td>IFS@</td>
<td>T</td>
<td>is not equal to</td>
</tr>
<tr>
<td>&gt;</td>
<td>binary</td>
<td>IFS</td>
<td>T</td>
<td>greater than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>binary</td>
<td>IFS</td>
<td>T</td>
<td>greater than or equal to</td>
</tr>
<tr>
<td>&lt;</td>
<td>binary</td>
<td>IFS</td>
<td>T</td>
<td>less than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>binary</td>
<td>IFS</td>
<td>T</td>
<td>less than or equal to</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>binary</td>
<td>I</td>
<td>I</td>
<td>logical shift left</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>binary</td>
<td>I</td>
<td>I</td>
<td>logical shift right</td>
</tr>
<tr>
<td>+</td>
<td>binary</td>
<td>IFS</td>
<td>same</td>
<td>addition, string concatenation of constant strings</td>
</tr>
<tr>
<td>-</td>
<td>binary</td>
<td>IFS</td>
<td>same</td>
<td>subtraction</td>
</tr>
<tr>
<td>*</td>
<td>binary</td>
<td>IF</td>
<td>same</td>
<td>multiplication</td>
</tr>
<tr>
<td>/</td>
<td>binary</td>
<td>IF</td>
<td>same</td>
<td>division</td>
</tr>
<tr>
<td>mod</td>
<td>binary</td>
<td>I</td>
<td>I</td>
<td>remainder</td>
</tr>
<tr>
<td>~</td>
<td>unary</td>
<td>I</td>
<td>I</td>
<td>bitwise boolean NOT</td>
</tr>
<tr>
<td>!, not</td>
<td>unary</td>
<td>IFS@</td>
<td>T</td>
<td>logical NOT</td>
</tr>
<tr>
<td>-</td>
<td>unary</td>
<td>IF</td>
<td>same</td>
<td>negation</td>
</tr>
<tr>
<td>&amp;</td>
<td>unary</td>
<td>L</td>
<td>@</td>
<td>address of</td>
</tr>
<tr>
<td>(expr)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>parenthesized expression</td>
</tr>
<tr>
<td>func_id(args)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>function call</td>
</tr>
</tbody>
</table>
Type Casts

Type casts explicitly force the compiler to convert an expression of one type into another type, and take the form

\[
\text{< type > expression}
\]

For example, if we have

```c
int a
float b
```

then

```c
a = <int> b
```

does not give a truncation warning, since we have told the compiler explicitly to convert \(b\) to an integer.

Note that not all type casts are possible. For example, the compiler cannot be forced to convert a cartesian location into an integer. In general, you can cast:

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
</tr>
</thead>
<tbody>
<tr>
<td>an int or a float</td>
<td>an int or a float</td>
</tr>
<tr>
<td>any pointer type</td>
<td>any other pointer type</td>
</tr>
<tr>
<td>any location type</td>
<td>a generic location (gloc)</td>
</tr>
<tr>
<td>a generic location (gloc)</td>
<td>any location type</td>
</tr>
</tbody>
</table>
For the most part, constants in RAPL-3 expressions are represented very straightforwardly. For example, the number 123 can be written exactly as it looks in the code of a RAPL-3 program. However, RAPL-3 also allows hexadecimal integer constants, exponential notation for floating point constants, string constants and location constants. This chapter presents the way in which these various kinds of constants are constructed.
Numeric Constants

Integer Constants
Any number that has neither a decimal point nor an exponent is an integer constant by default. Integer constants must lie in the range -2147483648 to +2147483647. Examples:

0
1000001
32768

Hexadecimal notation is also permitted. This consists of 0x followed by a sequence of digits (0 through 9, or a through f). Examples:

0x7fffffff ;; +2147483647
0x1000 ;; 4096
0xffffffff ;; -1

Binary Notation is also permitted. This consists of 0b followed by a sequence of binary digits (0 or 1).

Alphanumeric Constants
Alphanumeric constants are really just another form of integer constant. They permit the value of an ASCII code to be used in an expression by enclosing the character with the ’ (single quote) characters. For example, in

x = 'Z'

x is assigned the ASCII value for uppercase Z which is 90 (or 0x5a).

Floating Point Constants
A floating point numeric constant takes the form:

\[ \text{mantissa [ E|e [ +| - ] exponent ]} \]

The mantissa is a set of digits which may contain a decimal point. The base and exponent are optional. The base may be uppercase or lowercase (E or e). If not defined, the exponent is zero by default. The exponent is 1 or 2 digits. The sign, + or -, is optional. If not defined, the sign is + (positive) by default.

Examples:

0.0
1.
.2
1231.232
1e10
1E-5
.2e+6
1.5e+38
String Constants

String constants begin and end with the " (double quote) character and can be any length.

Within the string, the \ (backslash) character is used to form a sequence to represent the " character and other special ASCII codes. The following \ escape sequences are defined:

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Represents</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;</td>
<td>&quot;</td>
</tr>
<tr>
<td>\</td>
<td>\</td>
</tr>
<tr>
<td>\a</td>
<td>BELL</td>
</tr>
<tr>
<td>\b</td>
<td>BS</td>
</tr>
<tr>
<td>\e</td>
<td>ESC</td>
</tr>
<tr>
<td>\f</td>
<td>FF</td>
</tr>
<tr>
<td>\n</td>
<td>LF</td>
</tr>
<tr>
<td>\r</td>
<td>CR</td>
</tr>
<tr>
<td>\t</td>
<td>TAB</td>
</tr>
<tr>
<td>\v</td>
<td>VT</td>
</tr>
<tr>
<td>\ddd</td>
<td>the ASCII code represented by the three decimal digits ddd</td>
</tr>
</tbody>
</table>

Examples:

"This is a test. \n"
A string with a LF at the end, which causes the cursor to move to the next line at the beginning.

"This is \007 a test."
A string with a BELL character (ASCII code 7) in the middle which causes the terminal emulator to beep.

"\\He said, "The robot moves!""
A string with the backslash sequence and two double quote sequences which prints as: \He said "The robot moves!"

String constants can be concatenated (linked together to form a longer string) with the + (plus) operator. Note that the + operator only works on string constants and cannot be used to concatenate string variables.

"Data" + "Test"
is the same as

"DataTest"
String constants can also be used as actual parameters of subprograms. If an attempt is made to use a string constant as a \texttt{var} parameter to a subprogram, the compiler will generate a warning (since it is surely wrong to allow writing to a string constant.)
Location Constants

You can initialize \texttt{cloc} and \texttt{ploc} variables. The RAPL-3 compiler has built-in functions: \texttt{cloc{}} for generating cloc constants, and \texttt{ploc{}} for generating ploc constants. All of the arguments to \texttt{cloc{}} or \texttt{ploc{}} must be constant expressions and the result is a constant expression that can be used in a variable initialization.

The format of \texttt{cloc{}} is:

\begin{verbatim}
cloc( \_flags, x, y, z, zrot, yrot, xrot, e1, e2 )
\end{verbatim}

Where \texttt{flags} specifies extra information about the location, \texttt{x}, \texttt{y}, and \texttt{z} are the translational coordinates along the world axes, \texttt{zrot}, \texttt{yrot}, and \texttt{xrot} are orientational coordinates around world axes, and \texttt{e1} and \texttt{e2} are the coordinates for extra axes such as track. The argument \texttt{flags} must be an int constant expression and all other arguments are float constants.

An example of \texttt{cloc{}} is the following definition:

\begin{verbatim}
cloc my_tool = cloc(0, 0.0, 0.0, 10.0, 0.0, 0.0, 0.0, 0.0, 0.0)
;; tool transform for use with the tool_set() command.
\end{verbatim}

The format of \texttt{ploc{}} is:

\begin{verbatim}
ploc( \_machtype, flags, a1, a2, a3, a4, a5, a6, a7, a8 )
\end{verbatim}

Where \texttt{machtype} is the machine type (each type of machine, F3, A465, A255, .... has a different geometry and configuration resulting in different encoder counts for a given location), \texttt{flags} specifies extra information about the location, and \texttt{a1} to \texttt{a8} specify the number of encoder pulses from zero of the desired position for each axis. The arguments \texttt{machtype}, \texttt{flags}, and \texttt{a1} to \texttt{a8} are integer constant expressions.

An example of \texttt{ploc{}} is:

\begin{verbatim}
ploc start_point = ploc( mc_a465, 0, 3500, 2800, 1000, 4000, 2500, 1500)
;; initialized precision location
\end{verbatim}

A word of warning: initialized \texttt{cloc}s are useful for specifying tool transforms and related information. It is, however, very dangerous to hand-construct \texttt{ploc}s and command to robot to move to them. This is because the robot cannot physically move to any arbitrary joint configuration, and may collide with itself or objects in the workspace. If you must hand-construct locations, use extreme care.
CHAPTER 5

Control Flow

When a program executes, generally the computer executes one line, then the next, then the next. In order to make a program do useful things — for example, to repeat a particular task 10 times — we must be able to alter the way in which control passes from line to line of the program.

This section deals with statements that alter the sequence in which the statements in a program execute, allowing loops and conditional statements.
**break**

**Description**
Exit from a looping construct to the statement immediately following the looping construct (the statement immediately following `until`, `end while`, `end for`, or `end loop`).

Can be used to exit from the following looping constructs: `do`, `while`, `for`, or `loop`.

Often used with a condition such as an `if` or `if-else` statement.

If loops are nested, break exits from only the innermost `do`, `while`, `for`, or `loop` statement that encloses the `break`.

**Syntax**

```
break
```

**Context**

```
while ( expression_1 )
  ...
  if ( expression_2 )
    break
  end if
  ...
end while
```

**Example**

A loop that counts to 10.

```
int i
i = 0
loop
  if i == 10
    break
  end if
  i++
end loop
```

Break exits from the loop when `i` equals 10.

**See Also**

`continue`, `do`, `for`, `loop`, `while`

---

**case**

**Description**

Executes one of several statements, depending on the value of an integer expression. Note that you can implement any case statement with a series of `if` statements; the case statement just provides a compact way to select between several statements based on a value.

**Syntax**

```
case  expression
  [ of  constant_1  : ]
    [ statement(s) ]
  [ of  constant_2 to  constant_3  : ]
    [ statement(s) ]
  [ of  constant_4, constant_5  : ]
    [ statement(s) ]
  ...
  [else
    [ statement(s) ]]
end case
```

**Example 1**

An example with a single value, a list of values, a range of values, a mixed list, and an else value.

```
int tracking
string[64] message_1
... 
case tracking
  of 1:
  ...
```
message_1 = "success"
of 2, 3, 5:
  message_1 = "at maximum limits"
of 6 to 10:
  message_1 = "beyond maximum limits"
of 10 to 15, 20 to 23, 99:
  message_1 = "failure"
else
  message_1 = "unknown"
end case

Example 2

When this code is executed, if \( z = 1, 2, 3, \) or 6, then \( y \) is set to "hello". If \( z \) is 4 or 5, then \( y \) is set to "goodbye". If \( z \) is 7, then \( y \) is set to "right". If \( z \) is not equal to any of these values, then \( y \) is set to "unknown".

case \( z \)
of 1 to 3, 6:
  \( y[] = "hello" \)
of 4, 5:
  \( y[] = "goodbye" \)
of 7:
  \( y[] = "right" \)
else
  \( y[] = "unknown" \)
end case

RAPL-II

No equivalent in RAPL-II.

See Also

if

---

**continue**

**Description**

By-passes the remainder of the body of a loop and goes to the next iteration of the loop: the condition in `do` or `while`, the step increment in `for`, or the beginning of the next iteration in `loop`.

Can be used to by-pass the body of the following looping constructs: `do`, `while`, `for`, or `loop`.

Often used with a condition such as an `if` or `if-else` statement.

If loops are nested, `continue` by-passes the body of the innermost `do`, `while`, `for`, or `loop` statement that encloses `continue`.

**Syntax**

```
continue
```

**Context**

```
while ( expression_1 )
  ...
  if ( expression_2 )
    continue
  end if
  ...
end while
```

**Example**

Print only odd numbers.

```
for i = 1 to 10
  if (i/2)*2==i ;; integer division
    continue ;; it is even
  end if
  print i, "\n"
end for
```

**Result**

```
1
3
5
7
9
```
Control Flow

See Also  
break, do, for, loop, while

**do**

Description  
A looping construct that tests a condition at the end of the loop.

Flow enters the loop and the statements are executed down to the just before the until. The control expression following the until (a condition) is tested. If the expression is true (non-zero), flow goes back to the first statement after do. If the expression is false (zero), flow proceeds to the statement following the until.

Since the controlling expression is executed after the body of the loop, the loop body is always executed at least once, even if the first test of the control expression is false (zero).

A break can be used to exit a do loop and proceed to the line following the until. A continue can be used to by-pass the remainder of the body of a do loop. A goto can be used to jump to another position in the subprogram.

do statements can be nested.

Syntax  
do  
statement(s)  
until expression

Example  
A simple do loop.

```plaintext
i = 0  
do  
  move #safe_path[i]  
  i = i + 1  
until i > 4
```

The loop body executes 5 times, with i having the values 0, 1, 2, 3, and 4. On exit from the loop, i has the value 5.

See Also  
while, for, loop, break, continue, goto

**for**

Description  
A looping construct that executes a loop for a defined number of times.

The for construct controls the number of times the loop is executed by using an integer variable (a counter) with an initial value, a final value, and the size of step (increment) from initial to final.

Defining the step is optional. If step is not specified, it is assumed to be +1.

Step can be negative for a decrementing counter. In any event, the specified step must be a constant expression.

For executes in the following way. The counter variable is initialized to the value of expression_1. The counter is then tested to see if it is greater than (if step expression_3 is positive) or less than (if step expression_3 is negative)
expression_2. If so, execution proceeds at the first statement after the end of the loop (after **end for**). The statements in the body of the loop are executed. At the end of these statements the step (expression_3) is added into the counter. Control then loops back to the condition test and we repeat.

One implication of the way in which the **for** loop is implemented is that it is possible that the body of the loop might never be executed. Consider the following **for** loop:

```c
for x = 1 to 0
    printf("This is never printed\n")
end for
```

The loop does nothing, since the test (is \(x > 0\)) is true initially, causing the body of the loop to be skipped.

**Syntax**

```
for variable = expression_1 to expression_2 [step expression_3 ]
    statement(s)
end for
```

**Example**

With an increment of 1.

```c
for x = 1 to 10
    move #safe[x]
end for
```

**Step** is not specified and is assumed to be + 1. The function **move** is executed 10 times, with \(x = 1, 2, 3, \ldots 10\). The arm moves from safe location 1 to 2 to 3 ... to 10.

With a decrement of 1.

```c
for x = 10 to 1 step -1
    move #safe[x]
end for
```

**Step** is defined as – 1. The function **move** is executed 10 times, with \(x = 10, 9, 8, \ldots 1\). The arm moves from safe location 10 to 9 to 8 ... to 1.

With an increment of 3.

```c
for x = 1 to 11 step 3
    move #safe[x]
end for
```

**Step** is defined as + 3. The function **move** is executed 4 times, with \(x = 1, 4, 7,\) and 10. The arm moves from safe location 1 to 4 to 7 to 10. Note that even though the limit expression_2 is 11, this value is never seen by the body of the loop, since the next value after 10 (13) is in fact beyond the limit.

**See Also**

**do, while, loop**

---

**goto**

**Description**

Jumps to a statement marked with a label.

A label is named with an identifier and follows the rules for identifiers. The label can be before or after the **goto**.
A `goto` can jump only to statements within the main program or within the current subprogram (sub, func, or command). A `goto` can neither jump between the main program and a subprogram, nor between subprograms.

**Caution**

Gotos should be used with caution. Overuse of the `goto` statement can make code extremely difficult to read and debug. Good use of conditionals, loops, break, or continue can almost always eliminate the need for a `goto`.

**Syntax**

The label identifier is followed by two colons. The immediately following statement may be on the same line or the next line.

```
identifier:: statement
    ...
    goto identifier
```

```
 identifier::
    statement
    ...
    goto identifier
```

**Example**

A simple goto.

```
...
label_1::
    ...
    if(query_another_loc()=='Y')
        goto label_1
    end if
    ...
```

The earlier statement declares the label `label_1`. If the condition in the `if` statement is true, the `goto` directs control to the statement following `label_1`.

**See Also**

identifiers, break, continue

---

**if**

**Description**

A conditional construct which causes a statement to be executed only if a specific condition is true (non-zero). Optional else and elseif clauses allow 2-way or multi-way branching.

Begins with `if` and ends with `end if`. The use of `then` is optional. Can be used with `else` and with `elseif`.

You can use `if` with `else`, to execute one set of statements if the condition is true, and execute a different set of statements if the condition is false. This construction is a two-way branching (see syntax (b)). The elseif keyword allows an `if` statement to evaluate several possible conditions in turn creating a multi-way branch like a case statement (see syntax (c)).

`if` statements can be arbitrarily nested.

**Syntax**

(a) a simple `if` statement:

```
    if expression [then]
        statement(s)
    end if
```
(b) \textit{if} with an \textbf{else} clause

\begin{verbatim}
if expression [then]
  statement(s)
else
  statement(s)
end if
\end{verbatim}

(c) \textit{if-elseif} construction

\begin{verbatim}
if expression [then]
  statement(s)
elseif expression
  statement(s)
elseif expression
  statement(s)
else
  statement(s)
end if
\end{verbatim}

Example (a) This is a simple \textit{if} statement.

\begin{verbatim}
if (curr_locnum <= num_safe_path_locs) then
  move #safe_path[curr_path_locnum]
end if
\end{verbatim}

If the condition is true (curr_locnum is less than or equal to num_locs), the \texttt{move} statement executes. If the condition is false, the program flow proceeds to the line following \texttt{end if}.

(b) This is an \textit{if} and \textbf{else} construction.

\begin{verbatim}
if (curr_locnum <= num_locs)
  move #safe_path[curr_locnum]
else
  curr_locnum = curr_locnum - 1
end if
\end{verbatim}

If the condition is true (curr_locnum is less than or equal to num_locs), the \texttt{move} statement executes. If the condition is false, the statements following \texttt{else} execute (curr_locnum is decremented by 1).

(c) This is one example of nested statements. Inner statements must end before outer statements.

\begin{verbatim}
if (num==num_locs+1)
  print_msg_screen("Teach new power loc.")
  teach(#power_loc[num])
  num_locs++
  if(num_locs<10)
    if(query_another_power_loc()=='Y')
      goto labl
    else
      num_locs=0
    end if
  end if
end if
\end{verbatim}
(d) An **elseif** construction.

```plaintext
if(t==123)
    elseif(t<10)
    elseif(t>200)
    else
    end if
```

See Also

case

**loop**

Description

A looping construct with no condition. Begins with **loop** and ends with **end loop**. Since there is no control expression, the loop continues forever until a **break** or if necessary, a **goto**, causes flow to proceed out of the loop. **loop** statements can be nested.

Syntax

```plaintext
loop
    statement(s)
end loop
```

Example

In this example, the program prompts and gets a number to identify a location. The prompting and getting continues indefinitely until the user enters a valid number.

```plaintext
[1] loop
[2]     printf("Enter location number >")
[3]     readline($str, 10)
[4]     if str_to_int(num, $str) < 0
[5]         print("Invalid number\n")
[6]         continue
[7]     end if
[8]     if ((num<0) or (num>20))
[9]         printf("Number is out of range\n")
[10]        continue
[11]     end if
[12]     break ;; if we get here, we are DONE
[13] end loop
```

Line 2 displays a prompt asking the user to enter the number of the desired location. Lines 3 to 7 read in a string typed by the user and try to convert the string to an integer. If this fails, an error message is printed and a **continue** sends control back to the start of the loop. Lines 8 to 11 verify that the number is in the expected range, displaying an error message and sending control back to the start of the loop if it is not. Lastly, line 12, which is reached only if the number is valid and in range, exits the loop.

See Also

do, while, for, break, continue, goto

**while**

Description

A looping construct that tests a condition at the beginning of the loop. Begins with **while** and ends with **end while**. The control expression (a condition) is tested. If the control expression is true (non-zero), then flow enters the loop and the statements are executed. At the end, flow goes back to the control expression for the next test. If the expression is
false (equals zero), flow proceeds to the statement following **end while**.

If the initial test is false (zero), flow never enters the body of the loop and the statements are never executed.

If the control expression never evaluates to zero, or is a non-zero constant, for example `while(1)` , the loop continues indefinitely.

A **break** can be used to exit a **while** loop and proceed to the line following the **end while**. A **continue** can be used to by-pass the remainder of the body of a **while** loop. A **goto** can be used to jump to another position in the program.

**While** statements may be arbitrarily nested.

**Syntax**

```
while expression
    statement(s)
end while
```

**Example**

A simple while statement.

```
i = 0
while i < 5
    move #safe_path[i]
i = i + 1
end while
```

The loop body executes 5 times, with `i` having the values 0, 1, 2, 3, and 4. On exit from the loop, `i` has the value 5.

**See Also**
do, for, loop, break, continue, goto
RAPL-3 has three distinct kinds of executable objects: subroutines (subs), functions (funcs), and commands (commands). Collectively, subs, funcs, and commands are referred to as subprograms. main itself is a special case of a command subprogram.
Subprograms

One way to understand the concept of subprograms is to look at a brief example:

```plaintext
sub sayhello()
    int x
    x = 0
    printf("Hello!\n")
end sub

sub say_n_plus_1(int n)
    printf("n + 1 = {}\n", n + 1)
end sub

func int a_plus_b(int a, int b)
    return a + b
end func

main
    int x, y
    x = 10
    sayhello()
    say_n_plus_1(x)
    y = a_plus_b(1, x)
    printf("x + 1 = {}\n", y)
end main
```

This example defines two subroutines (sayhello() and say_n_plus_1()) and one function (a_plus_b()).

Program execution starts in main. Line 16 declares two variables that belong only to main (local variables) called x and y; in line 17, x is set to have the value 10.

When line 18 is reached, the subroutine sayhello() is executed. sayhello() has its own local variable x, which it sets to have a value of 0 in line 3. sayhello() then executes line 4 which prints a message out on the console. When the end of sayhello() is reached, control returns to main to line 19.

The fact that sayhello() has set its variable x to be 0 does not change the value of main's variable x at all. Any variable declared inside a subprogram is local to that subprogram and cannot be changed by any outside means. Variables that are declared outside of any subprogram are accessible to all subprogram and are called program scope or simply program variables. This concept of local and program variables is part of variable scope.

After sayhello() is executed (called) by main, main calls the subroutine say_n_plus_1(). One difference between the call to sayhello() and the call to say_n_plus_1() is that the latter has an expression (x) inside the brackets next to the subroutine name. This is an argument (or actual parameter) to say_n_plus_1(). The value of x is given (or passed) to the subroutine.

Subprogram say_n_plus_1() then executes with its variable n initially set to 10, since that was the value passed to it by main. n is a special local variable of say_n_plus_one() called a formal parameter. formal parameters get initial values that are given by the caller of the subprogram, in this case, main.

At line 8, say_n_plus_one now prints out the value of n + 1, which is 11 in this case. Control returns to main at line 20.

In line 20, main sets y equal to a_plus_b(1, x). This is an example of a function call; the function a_plus_b() is called with the two arguments (1 and 10 (x)) just like a subroutine is called. Line 12 is the only line in a_plus_b(), and is a return statement. For a function, the return statement indicates that a value (in this case a + b or 11) is to be returned to the calling subprogram. The effect in this example is that y gets set to the value that a_plus_b() returns, or 11.

This result is printed out at line 21, and the program ends. The rest of this chapter explains in detail the elements of RAPL-3 that deal with subprograms.
Kinds of Subprograms

**subs**

A sub (subroutine) is the simplest kind of RAPL-3 subprogram. A sub can take any number of arguments (including none), but does not return any value to the calling subprogram. As a result, a sub cannot appear inside an expression.

**Declaration Syntax**

```plaintext
sub sub_identifier ( parameter_list )
    [ declarations and statements... ]
end sub
```

**Calling Syntax**

```plaintext
sub_identifier(actual_parameter_list)
```

Note that the actual_parameter_list must match the parameter list in the sub declaration. That is, there must be the same number of parameters as those declared, and the types of the expressions must be compatible.

**funcs**

A func is similar to a sub in that it can accept any number of arguments. However, a func returns a value to the calling subprogram. In RAPL-3, func can return any int, float, loc, ploc, gloc or pointer type of value (a func cannot return a string or structure, but can return a pointer to a string or structure.)

For example, `a = sin(x) + cos(y)` calls the `sin()` function to compute the value of the sines of variable x, calls the `cos()` function to compute the cosine of variable y, adds the two and then stores the result in variable a.

**Declaration Syntax**

```plaintext
func type func_identifier ( parameter_list )
    [ declarations and statements... ]
    return value
end func
```

Note that there must be at least one return statement that returns the value of the correct type somewhere in the body of the function. Functions can return only int, float, location, or pointer types.

**Calling Syntax**

There are two ways to call a function. As part of an expression:

```plaintext
... func_identifier(actual_parameter_list)...
```

or by itself as a statement:

```plaintext
func_identifier(actual_parameter_list)
```

In the latter form, the compiler will warn that the return value of the function is being ignored (unless warnings are disabled.)

Once again, the actual_parameter_list must match the parameter list in the func declaration.
commands

A **command** is in many respects identical to a **func int**. Commands must return an integer value, and can appear in expressions just like a **func**. The difference lies in the way that a **command** behaves when it is called as a statement by itself. In this case, the compiler generates code that checks the return value of the command, and if that value is less than zero (negative) it causes an **exception** to be **raised** with the error code equal to the returned value. This provides a default way of handling errors: **commands** that fail should return a negative number describing the error (and **error descriptor**). The system can then handle the error, even if only by aborting the program and issuing an error message.

The section on **structured exception handling** deals with **exceptions**, and with how to handle them, in more detail.

Note that this automatic error check is not performed when the command is used as a function in an expression. This allows the code to look for and handle errors explicitly.

**Declaration Syntax**

```plaintext
command cmd_identifier ( parameter_list )
    [ declarations and statements... ]
    return value
end command
```

Note that there must be at least one **return** statement that returns an integer in the body of the command.

**Calling Syntax**

There are two ways to call a command. As part of an expression:

```plaintext
... cmd_identifier(actual_parameter_list)... 
```

or by itself as a statement:

```plaintext
cmd_identifier(actual_parameter_list)
```

The latter form is the more usual. Unlike **functions**, the compiler does not warn about the return value being ignored, since code is automatically generated to check the return value and act upon it if it is negative.

Once again, the actual_parameter_list must match the parameter list in the **command** declaration.

**Example**

Most of the robot and CROS operations are, in fact, commands. A program can move the robot to a given location using the move() command like this:

```plaintext
move(#this_loc)
```

In this case the system handles any errors that move() reports (by means of its return value.) In the following example, we examine and act on the error explicitly:

```plaintext
r = move(#this_loc)
if (r < 0)
    ;; take action...
    ...
end if
```
Where main fits in

The main part of a RAPL-3 program is actually a special type of command. It differs from a normal command in three respects:

1. It is declared with main and end main
2. It need not contain a return statement; the compiler automatically inserts a “return 0” at the end of main. The user is free, however, to return some other value instead.
3. When the program is run, the main section is called by the startup code.
Parameters

In func, sub and command declarations, the parameter list part is a comma separated list of individual parameter declarations, possibly empty. Each parameter declaration takes the form:

```
[var] [type_declaration] identifier
```

If type_declaration is omitted then int is the default.

To the subprogram, the parameter looks like an ordinary local variable. However, its value is set to the actual parameter value provided by the caller.

The special optional keyword var indicates whether or not changes to the parameter value inside the subprogram change the value of the parameter in the calling subprogram. The default (var keyword omitted) does not change the variable outside the subprogram. For example:

```
sub this_routine(float x)
  x = 2.71828 ;; will have no effect on the
  ;; calling subprogram
end sub

sub that_routine(var float y)
  y = 1.0
end sub

... ;; in the calling subprogram
this_routine(t) ;; t is unchanged after this call
that_routine(t) ;; t is 1.0 after this call
```

Restrictions on Parameters

Function formal parameters (appearing in declarations) that are complex entities like strings, arrays, or structs are treated by the compiler exactly as if they had been declared var. (Internally, this is done by passing where the object is instead of the passing the value of the object itself.)

If this kind of complex parameter is not actually declared var, then the compiler will generate warnings about any code in the subprogram that modifies the variable. This protects the programmer from inadvertently changing the variable’s value in the calling routine.

The compiler also generates a warning if a string constant is used as the actual parameter of a formal "var string[]" parameter.

Var parameters can be of any type, but non-var parameters may be only int, float, cloc, ploc, gloc, or any pointer type. Furthermore, when calling a subprogram, var actual parameters must be expressions that might reasonably occur on the left-hand-side of an assignment. For example:

```
sub alpha(var float x) ;; note the var parameter
...
end sub

... ;; in another subprogram
alpha(a[j*i+1]) ;; this is OK
alpha(q) ;; this is OK
alpha(q+1) ;; but this is not OK
...
sub beta(int[10] a) ;; this is taken to be
  ...
end sub          ;; var int[10] a

sub gamma(int[10]@ a) ;; this is OK
  ...
end sub

sub delta(var int[10] a) ;; this is OK
  ...
end sub
Func, Sub, and Command Prototypes

Funcs, subs and commands must always be defined before they are used in a program. Since it is not always convenient to rearrange a program so that definitions precede uses, a mechanism for prototyping subprograms has been provided. A prototype takes the form:

```
proto func_sub_or_command_header
```

For example:

```
proto func int myfunc(int x, float y) ;; prototypes
proto command qq(int a)

x = myfunc(t,1.5) ;; use of myfunc
qq x ;; and qq

...

func int myfunc(int a, float b);; actual definition
  ...
end func
command qq(int i) ;; actual definition
  ...
end command
```

Note that the names of the arguments of `myfunc` and `qq` need not match the names in their prototypes, but the number of arguments and their types must match exactly.
Libraries

When a RAPL-3 source file (or set of source files) is compiled, the result is a RAPL-3 module. If a module has a main section then it can be run as a program. However, some modules do not have main sections, and instead serve as libraries.

A library is a compiled RAPL-3 module that contains subprograms and variables that can be accessed by other modules. Many of the subprograms commonly used in writing RAPL-3 programs are in fact contained in one of several libraries. For example, the move() command is actually contained in the robot library (robotlib.r), and the printf() command is actually defined in the system library (syslib.r). Libraries are used whenever it is likely that a subprogram or variable will be needed by many different programs. The calling programs need only know the names and types of each element in the library in order to use it. This allows details of how the library works to be hidden – which is actually good, since this means that subroutines in the library can be revised and improved without affecting the programs that use it.

The only differences between a library and a normal program are:

1. the library usually has no main section, and is generally never run by itself.

2. the library makes some of its variables and/or subprograms visible to other modules by declaring them as global or export. This will be discussed in more detail in the next section.

To use a library with your program, there are three requirements:

1. At compile time the compiler must be told which libraries you want to use and must have access to the compiled libraries. See the –L option in the compiler documentation. We say that your program was compiled with reference to the library.

2. the library must be installed where the runtime system can find it. It must either be in the same directory as your program or must be in the /lib directory.
Variable and Subprogram Scope

A Scope Example

Suppose we have the following declarations in two RAPL-3 programs.

In program1.r3:

```
int test_value
...

func int factorial(int n)
  if n == 0 then
    return 1
  else
    return factorial(n-1)*n
  end if
end func
```

In program2.r3:

```
int test_value
global int intglob
export int intexp

export func plusone(x)
  ;; default types are float
  return x+1
end func

global sub do_something()
  ...
end sub
```

Any subprogram in program1 can use and modify the program variable `test_value` in program1. Furthermore, any subprogram in program2 can use and modify the program variable `test_value` in program2. These are, however, two separate variables and the value of the one in program1 has no connection to the value of the other in program2.

Any subprogram within program1 can call the factorial function. For example, a subprogram of program1 might have:

```
a = factorial(10) ;; compute the factorial of 10
;; and store it in a
```

The factorial function is not visible to program2, and cannot be called from program2.

Program2's variable `intglob` and sub `do_something` can be used by any other program in the system, providing they are compiled with reference to program2. For example, any subprogram in program1 can modify `intglob` and call `do_something`, since these objects are both global.

Program1 can also access `intexp` and `plusone()`, provided that it specifies where these functions are to be found. For example, in program1, one could execute the following code:

```
a = program2:plusone(b)
program2:intexp = program2:intexp + 1
```

Alternatively, one can use the `with` statement to avoid having to specify which program to find `plusone` and `intexp` in:

```
with program2
  a = plusone(b)
  intexp = intexp + 1
end with
Relevant Statements

**with**

**Description** The `with` construction allows the search path of the scanner to be changed to search an imported module first, before normal processing.

`with` statements may not be nested.

**Syntax**

```plaintext
with modulename
  ...statements...
end with
```

**Example** See the scope example.

---

**return**

**Description** The return statement causes control to return to the func, sub, or command that called the current subprogram. Inside a sub, the return statement takes the form:

```plaintext
return
```

Funcs and commands each return a value, which must be specified in the return statement:

```plaintext
return value_expression
```

`main` can return an integer value. If it does not, a zero value is returned automatically.

**Syntax**

```plaintext
return ;; in a sub

return value ;; in a func or command
```

**Example**
When a RAPL-3 program is compiled, it actually goes through two distinct stages:

(1) Preprocessing
    The source code is interpreted by the preprocessor, which produces a temporary file for stage (2). This temporary file has had all comments removed, all `#include` directives replaced by the included files, all macros (defined by `#define`) replaced and all conditional compilation directives (`#ifdef` and `#ifndef`) carried out.

(2) Translation
    The actual compiler takes the temporary file prepared by stage (1) and converts it into RAPL-3 object code.

Breaking the compilation into two stages allows a great deal of flexibility. These are the kinds of operations that can be performed by taking advantage of the preprocessing stage:
It is often inconvenient for a program to be located entirely in one source file. For example, it might make sense to break the program up into a section dealing with moving the robot, a section dealing with the user interface and a section dealing with communication to another machine. The `include` directive makes this kind of split very simple. For example consider the following 4 source files:

**In file robot.r3:**

```
;; These routines deal with moving the robot
...
;; end of robot.r3
```

**In file user.r3:**

```
;; These routines deal with the user interface
...
;; end of user.r3
```

**In file comm.r3:**

```
;; These routines deal with communications
...
;; end of comm.r3
```

**In file main.r3:**

```
;; Main program
.include "robot.r3"
.include "user.r3"
.include "comm.r3"
;; Main’s stuff goes here
...
;; end of main.r3
```

What the actual compiler sees, after the preprocessing step has been run, is this: (we have left comments in for the purposes of this example; in reality, the preprocessing step also deletes all comments.)

```
.1 "main.r3"
;; Main program
.1 "robot.r3"
;; These routines deal with moving the robot
...
;; end of robot.r3
.3 "main.r3"
.1 "user.r3"
;; These routines deal with the user interface
...
;; end of user.r3
.4 "main.r3"
.1 "comm.r3"
;; These routines deal with communications
...
;; end of comm.r3
.5 "main.r3"
;; Main’s stuff goes here
...
;; end of main.r3
```
What has happened is that every time a `.include` directive was encountered, the `.include` was replaced by the *entire file* that was named in the `.include` preprocessor directive. As far as the compiler is concerned, it sees only one input file.

You will note the rather odd constructions on the 1st, 3rd, 7th, 8th, etc. lines which are of the form:

```
.number "filename"
```

These are understood by the compiler to mean that the next line of text actually comes from the given line of the given file. This allows error messages during compilation to match up with the actual lines in your source files. Note that the preprocessor generates these automatically for us.

**Macro Substitution**

The preprocessor provides a *macro substitution* facility that has a similar effect to the named constant *(const)* capabilities of the language. However, preprocessor macros work by direct string replacement, allowing a symbol to be replaced with any arbitrary string. (RAPL-3 does not presently support macros with parameters.) Consider this example:

```
.define NAME "Joe"
.define NUMBER 1234
.define WHICH func1

...

printf("The name is {}, and the number is {}\n", NAME, NUMBER)
WHICH(NUMBER)
...
```

After being run through the preprocessor, this sample looks like this to the compiler:

```
...

printf("The name is {}, and the number is {}\n", "Joe", 1234)
func1(1234)
...
```

The `.define` lines are replaced by blanks; the preprocessor strips them out of the file. Since the symbol NAME has been defined to be the characters "Joe" (including the quotes), everywhere NAME appears it gets replaced by this string.

Note that while something similar to the printf() in the 7th line could have been done using name constants (via *const*), the call to func1() in the 8th line could not.

Note also the symbols that were `.defined` are never seen by the translation part of the compilation. As far as the RAPL-3 language is concerned, these symbols do not exist; they are relevant only to the preprocessor.
Conditional Compilation

The preprocessor can be used to effect *conditional compilation*, allowing one set of source code to produce several different versions of program. This is often useful, particularly for debugging purposes. Consider this example:

```c
;; Define this to enable debugging code:
.def DEBUG
...
main
.ifdef DEBUG
  printf("Debugging version\n")
.else
  printf("Normal version\n")
.endif
... lots of code here ...
.ifdef DEBUG
  printf("debug: result was {}\n", n)
.endif
... more code here ...
```

After the preprocessing stage, this looks like this:

```c
...
main

  printf("Debugging version\n")
...

  printf("debug: result was {}\n", n)
... more code here ...
```

The `.ifdef` directive allows code to be selectively included in the output of the preprocessor if a symbol is *defined* – that is, if there has been a `.define` for that symbol before the `.ifdef` in the source code. Note that the first `printf()` was included in the output because the symbol DEBUG had been defined in the 2nd line. The second `printf()` is *not* included because it is in the `.else` clause of the `.ifdef DEBUG`.

Using this technique it is possible to simply leave debugging code in your program and turn it off (by commenting out the `.define DEBUG`, for example) once the program has been debugged. If problems occur later with the program, the debugging code is still there and can be easily turned back on.
Preprocessor Directives in General

Placement
Preprocessor directives can be interspersed with other parts of the program.

Syntax
.preprocessor_directive  [arguments]

On a line, a preprocessor directive cannot be preceeded by anything except blank spaces. Each preprocessor directive begins with a dot. The entire line is processed by the preprocessor. Definitions may not extend over more than one line.

Comments
Comments are stripped from the input file.

Strings
The preprocessor recognizes that “ and ” (double quotes) delimit strings. No macro expansions will be performed on text within “ and ”.

Special Symbols
The following two macros are always defined by the preprocessor, and will be replaced by their appropriate values:

__LINE__ the current line # in the current source file
__FILE__ the current source file as a quoted string

For example, if you place this in your program:

printf("I am at line {} of file {}\n", __LINE__, __FILE___)

the effect will be to have the program print out a message giving what source line and source file the printf() was located on.
Preprocessor Directives

.define

Description: Creates a preprocessor symbol. If no value is specified for the symbol, the preprocessor will set the value of the new symbol to be “1” (without the quotes.)

Syntax: `.define [symbol]`
`.define [symbol] [value]`

Examples:
`.define TRUE 1`
`.define DEBUG`

.error

Description: Forces the preprocessor to issue an error message

Syntax: `.error [message]`

Example: `.ifndef IMPORTANT`
`.error The symbol IMPORTANT must be defined!`
`.endif`

This can be used to make sure that a particular preprocessor symbol (like IMPORTANT in the above example) is actually defined.

.ifdef

Description: Conditionally includes source if symbol is defined.
Can be used with an.else clause.

Syntax: `.ifdef [symbol]`
`lines of source code to be included if symbol is defined`
`.endif`
`.ifdef [symbol]`
`lines of source code to be included if symbol is defined`
`.else`
`lines of source code to be include if symbol is not defined`
`.endif`

Example: See the introduction.

.ifndef

Description: Conditionally includes source if [symbol] is not defined.
Can be used with .else clause.

Syntax: `.ifndef [symbol]`
`lines of source code to be included if symbol is not defined`
`.endif`
`.ifndef [symbol]`
`lines of source code to be included if symbol is not defined`
`.else`
`lines of source code to be include if symbol is defined`
`.endif`
.include
Description: The .include directive inserts text contained in one source file into the current source file at compile time.

Around the filename " " (double quotes) are required. The filename is identified by the programmer. When the program is compiled, the contents of the file filename replace the .include line.

This form searches the current dir first.

Syntax: .include " filename "
Example: see the introduction

.number "filename"
Description: Forces a line to be recognized as line number of file filename.

Syntax: .number "filename"
Example: see the introduction

.undef
Description: Deletes a preprocessor symbol definition.

Syntax: .undef [ symbol ]
Using the Compiler from the Command Line

It is often useful to be able to run the RAPL-3 compiler from a command line instead of from ROBCOMM3. This is particularly useful for large projects with many source files, where tools like `make` are used to build the project.

The compiler is typically located, for example, in “C:\Program Files\CRS Robotics\RAPL-3\bin”, and is called `r3c`. (RAPL-3 Compiler.)

Command line syntax

```bash
r3c [-options] input_file_name
```

Options

- `-o output_file_name`
  send output to a particular file; the default is `r.out`
- `-e error_file_name`
  send all error messages to the specified file
- `-?`
  print a help message
- `-h`
  same as `-?`
- `-fstack=number`
  set the running stack size of the program to `number` words
- `-Wall`
  enable all reasonable warnings
- `-Wmax`
  enable even possibly unreasonable warnings
- `-Wnone`
  disable all warnings
- `-v`
  be verbose; print lots of information about what is happening
- `-Dsymbol`
  make the preprocessor act as if `symbol` had been `.defined`
- `-Dsymbol=value`
  make the preprocessor act as if `symbol` had been `.defined`
- `-O0`
  don’t perform any code optimization
- `-O1`
  perform basic optimizations (default)
- `-s`
  reduce compiled code size by stripping out any symbols
- `-x`
  exclude all symbols except `global` and `export` symbols
RAPL-3 commands provide a means of automatically handling errors. If a command is called like this:

```
theCommand(x, y, z)
```

then the RAPL-3 compiler generates code that automatically checks the command's return value. If the value is negative (less than zero) an exception has occurred.

When an exception occurs, the default way of handling it is for the program to stop and an error message to be printed out. This message typically looks like:

```
Exception raised at line 123 of myprog.r3: file not found
```

Note that the system typically can report the source line and file where the exception occurred. It also attempts to interpret the return code as an error descriptor, and reports the error as the equivalent descriptive string.

One way of explicitly dealing with exceptions in a program is to simply check the return value of all commands. For example:

```
t = theCommand(x, y, z)
if (t < 0)
    ...error recovery...
end if
```

This can be very tedious and can make the code quite difficult to read, as every command will tend to have at least 3 extra lines of code after it to handle possible errors.
try-except Construct

Structured Exception Handling provides a much neater and simpler way of handling exception in program execution. Consider this short example:

```python
try
    ...
    thecommand(x, y, z)
    thatcommand(z, y)
    thiscommand()
    ...
except
    ...error recovery code...
end try
```

The try-except construct allows the way the system reacts to exceptions to be changed in the region between the try and the except. If one of the commands in this section fails (returning a -ve number) then control is immediately transferred into the except part of the construct. The program can then find out what the error code was and even where it happened, and can take corrective action. (Note that the except part is only executed if an exception happens. If the program reaches the end of the try section successfully, then execution continues after the end try.)

There are, in fact, four things the except part of the try-except construct can do:

1. Simply do nothing, and allow control to pass to the statement following the end try.
2. Force the program to go back and execute the entire try section from the start, using the special retry keyword.
3. Force the program to execute the failing statement over again from its start using the resume keyword. For example, if thatcommand() had failed, then resume would go back and continue execution at thatcommand() again.
4. Force the program to continue execution at the statement following the one that failed using the ignore keyword. For example, if thatcommand() had failed, then ignore would force execution to continue from the next line, at thiscommand().

Syntax

The syntax of a structured exception handling section is:

```python
try
    statements
except
    exception_handling_statements
end try
```

On entry to the block, statements are executed in the usual way. If an exception occurs (a command fails) then execution is transferred to the except section.
A subprogram can have at most one active `try` block at a time. That is, `try` blocks cannot be nested within a subprogram, although from within a `try` block, one subprogram can call another one which also uses `try` blocks.

`Gotos` are not allowed inside `try-except` blocks. You can, however, `break`, `continue`, `return` or `raise` to get out of the block.

You cannot define a label inside a `try-except` block, consequently cannot `goto` into the middle of the block.

If an exception occurs inside the `except` part of the `try-except` block, then the exception is handled by the next level up of `try-except` block, or by the system (aborting with an error message) if there is no next level up.

Within the `except` section, the following special keywords are valid:

**retry**

- go back to the start of the `try` block and do the entire block over again.

**resume**

- go back to the statement that caused the exception and continue execution. This allows the offending statement to be re-executed.

**ignore**

- go back to the statement *following* the one that caused the exception and continue execution
Related Keywords and Subprograms

The following keywords and subprograms are related to exception handling:

Keywords:
  raise

Functions:
  error_code(), error_addr(), error_line(), error_file()
  addr_to_line(), addr_to_file()

Commands:
  abort()
CHAPTER 9

Library Subprograms

The libraries contain predefined subroutines, functions, and commands used to perform common programming tasks.

This chapter contains

- General
  general information about libraries, return values, and naming conventions

- RAPL-II to RAPL-3
  a mapping of functionality from RAPL-II to RAPL-3 for users who are familiar with RAPL-II

- Subprograms: Categories
  a description of each category, material common to subprograms in that category, and a list of each subprogram in that category

- Subprograms: Alphabetical
  a detailed description of each subprogram, listed alphabetically
General

Libraries
The subprograms are contained in several CRS-supplied libraries. Since these subprograms have global scope, you do not have to explicitly include a CRS-supplied library to use one of these subprograms, except for the teach pendant library.

Teach Pendant Library
Subprograms in the teach pendant library have export scope. You must explicitly name the teach pendant library when using a teach pendant subprogram. Details are with those subprograms.

Return Values and Errors
Return values less than 0 indicate an error condition. Error codes are listed in the Error Handling section.

Subprogram Names
Names of subroutines, functions, and commands follow these conventions.

Naming Conventions
The first component is the general family of item, such as string or location. The second component is the specific sub-family, often the object being dealt with, such as character, length, limit, cartesian data, or precision data. The last component is the operation, such as get, set, find, or reverse find. The _ (underscore) character is used as a separator.

    str_chr_get()
    str_chr_set()
    str_chr_find()
    str_chr_rfind()
    str_len()
    str_len_set()
    str_limit()
    str_limit_set()

    loc_cdata_get()
    loc_cdata_set()
    loc_pdata_get()
    loc_pdata_set()

Exceptions
Where there is only one operation of interest, such as a query, there is no operation named.

    str_len()
    str_limit()
Where a family, sub-family, or operation is obvious, it is not included. Instances include all arm motion commands and all math functions.

\begin{verbatim}
 depart
 move
 jog
 yaw
 ln
 sin
 sqrt
 mem_alloc
 mem_free
 time_set
\end{verbatim}

Where there is only one sub-family, the underscore may be omitted.

\begin{verbatim}
 griptype_set
 gripdist_get
\end{verbatim}

Where the name is an alias for another subprogram, components may be changed or omitted.

\begin{verbatim}
 jog_w(JOG_X,D) xw(D)
\end{verbatim}
### RAPL-II to RAPL-3

The following are the equivalent RAPL-II and RAPL-3 commands.

In some cases functionality is identical. In other cases functionality is different.

Some RAPL-II commands have been split into two or more RAPL-3 commands.

<table>
<thead>
<tr>
<th>RAPL-II</th>
<th>RAPL-3</th>
<th>ash</th>
<th>system shell</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABORT</td>
<td>abort()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ABS</td>
<td>fabs(), iabs()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ACOS</td>
<td>acos()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ACTUAL</td>
<td>pos_get(POSITION_ACTUAL)</td>
<td></td>
<td>actual</td>
</tr>
<tr>
<td></td>
<td>(precision)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALIGN</td>
<td>align()</td>
<td></td>
<td>align()</td>
</tr>
<tr>
<td>ALLOC</td>
<td>mem_alloc()</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>[only allocates, clears memory]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ANALOG</td>
<td>analog()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AOUT</td>
<td>aout()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>APPRO</td>
<td>appro()</td>
<td></td>
<td>appro</td>
</tr>
<tr>
<td></td>
<td>appros()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ARM</td>
<td>robot_flag_enable()</td>
<td></td>
<td>enable</td>
</tr>
<tr>
<td></td>
<td>open( “dev’estop”...)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>abort()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ASIN</td>
<td>asin()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ATAN2</td>
<td>atan2()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CIRCLE</td>
<td>circle()</td>
<td></td>
<td>circle</td>
</tr>
<tr>
<td>CLOSE</td>
<td>grip_close()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>COMP</td>
<td>loc_cdata_get()</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>loc_pdata_get()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XCOMP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>YCOMP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ZCOMP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OCOMP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ACOMP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TCOMP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CONFIG</td>
<td>ioctl() [put options]</td>
<td></td>
<td>siocfg</td>
</tr>
<tr>
<td>COPY</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>COS</td>
<td>cos()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPATH</td>
<td>cpath()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CTPATH</td>
<td>cpath()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CUT</td>
<td>str_edit() [deletes or inserts characters]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DECODE</td>
<td>str_to_int()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEG</td>
<td>deg()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Command</td>
<td>Description</td>
<td></td>
<td></td>
</tr>
<tr>
<td>------------</td>
<td>-----------------------------------------------------------------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DELAY</td>
<td>delay(), msleep()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DELETE, DEPROG</td>
<td>unlink(), rm, del</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEPART</td>
<td>depart(), depart()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DIR</td>
<td>ls, dir</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DISABLE</td>
<td>robot_flag_enable(), disable</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DLOCN</td>
<td>erase, eraseall</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DO</td>
<td>do [flow control]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DVAR</td>
<td>erase, eraseall</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EDIT</td>
<td>editor of Robcomm3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ELBOW [A255]</td>
<td>stance_set(... elbow ...), testing</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>pose/setstance, pose/setstance</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ENABLE</td>
<td>robot_flag_enable(), enable</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ENCODE</td>
<td>[int to string for printing] snprint(), sprintf()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>END</td>
<td>[flow control]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXECUTE</td>
<td>exec(), execv()</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>run, filename, filename</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FINISH</td>
<td>finish()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FREE</td>
<td>heap_space(), mem [in memory], df [on file system]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GETCH</td>
<td>read(), getch()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GOPATH</td>
<td>ctpath_go(), gopath</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GOSUB</td>
<td>[call to sub, func, or command]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GOTO</td>
<td>goto [flow control]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GRIP</td>
<td>gripdist_set(), grip(), grip</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HALT</td>
<td>halt()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HERE</td>
<td>here()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HOME</td>
<td>home()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HOMEGRIP</td>
<td>homegrip()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HOMESEQ</td>
<td>seekswitch()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HOMEZC</td>
<td>homezc()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IF</td>
<td>if [flow control]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IFPOWER</td>
<td>if robotispowered() ...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IFSIG</td>
<td>if input() ...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IFSTART</td>
<td>fpstart(), front panel library</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IFSSTRING</td>
<td>if [comparing string variables or constants]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INBOUNDS</td>
<td>[is location in bounds]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INPUT</td>
<td>input()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INT</td>
<td>[returns closest integer value]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INVERT</td>
<td>[invert Z, leave X and Y, of coord system]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JOG</td>
<td>jog_w(), wx(), wy(), wz(), xrot(), yrot(), zrot()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JOINT</td>
<td>joint()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>KBHIT</td>
<td>[character exists, to be read from serial input]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LIMP</td>
<td>limp()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LISTL, LLOC</td>
<td>list [in .v3]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LISTP, LPROG</td>
<td>ls, dir</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LISTV, LVAR</td>
<td>list [in .v3]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LN</td>
<td>ln()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOCK</td>
<td>lock()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOG</td>
<td>log()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MA</td>
<td>[move to absolute angles]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MAGRIP</td>
<td>[force applied by magnetic gripper]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MI</td>
<td>[move by increments of angles]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOD</td>
<td>[function] mod [operator]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOTOR</td>
<td>motor()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOVE</td>
<td>move()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NOLIMP</td>
<td>nolimp()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NOTRACE</td>
<td>abort()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OFFSET</td>
<td>base_set()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ONLINE</td>
<td>online()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ONPOWER</td>
<td>loop ... if ... robotispowered() ... delay()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------------</td>
<td>----------------------------------------------------------------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ONSIG</td>
<td>loop ... if input() ... delay()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ONSTART</td>
<td>read( “dev/buttons”... front panel library</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OPEN</td>
<td>grip_open()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUTPUT</td>
<td>output()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PASTE</td>
<td>str_edit, str_deit, str_insert (only inserts characters)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PAUSE</td>
<td>signal() ... input() ... if ...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PENDANT</td>
<td>pendant</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PITCH</td>
<td>pitch(), jog_t(TOOL_PITCH, ...)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>POINT</td>
<td>loc_cdata_set(), loc_pdata_set(), point()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>POSE</td>
<td>stance_set(), pose setstance</td>
<td></td>
<td></td>
</tr>
<tr>
<td>POW</td>
<td>pow()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PRINTF</td>
<td>printf()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RAD</td>
<td>rad()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RANDOM</td>
<td>random()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>REACH</td>
<td>stance_set()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>READY</td>
<td>ready()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RENAME</td>
<td>mv, move</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RETURN</td>
<td>return [from sub, func, or command]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ROLL</td>
<td>roll(), jog_t(TOOL_ROLL, ...)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RUN</td>
<td>run, filename</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SERIAL</td>
<td>ioctl() [get options]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SET</td>
<td>= [assignment] operators</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SHIFT</td>
<td>get/change/move, shift translations only</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SHIFTA</td>
<td>shift_w()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SIN</td>
<td>sin()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPEED</td>
<td>speed_set(), speed(), speed_get(), speed(-1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Library Subprograms</td>
<td>Description</td>
<td>Parameters</td>
<td>Return Type</td>
</tr>
<tr>
<td>---------------------</td>
<td>-------------</td>
<td>------------</td>
<td>-------------</td>
</tr>
<tr>
<td>SQRT</td>
<td>sqrt()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SRANDOM</td>
<td>seed()</td>
<td>(reseeds)</td>
<td></td>
</tr>
<tr>
<td>STATUS</td>
<td></td>
<td></td>
<td>status</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>servostat</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>sysstat</td>
</tr>
<tr>
<td>STRPOS</td>
<td>str_chr_find</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SYSTEM</td>
<td></td>
<td></td>
<td>sysstat</td>
</tr>
<tr>
<td>TAN</td>
<td>tan()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TEACH</td>
<td></td>
<td></td>
<td>pendant</td>
</tr>
<tr>
<td>TIME</td>
<td>mtime()</td>
<td></td>
<td>date</td>
</tr>
<tr>
<td></td>
<td>time()</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>delay()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TOOL</td>
<td>tool_set()</td>
<td></td>
<td>tool</td>
</tr>
<tr>
<td></td>
<td>tool_get()</td>
<td></td>
<td>tool</td>
</tr>
<tr>
<td>TRIGGER</td>
<td>settrigger()</td>
<td>??</td>
<td>trigger</td>
</tr>
<tr>
<td>TRUNC</td>
<td>(int)</td>
<td></td>
<td>typecast</td>
</tr>
<tr>
<td>UNLOCK</td>
<td>unlock()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UNTIL</td>
<td>do ... until</td>
<td>[flow control]</td>
<td></td>
</tr>
<tr>
<td>W0</td>
<td>pos_get(), xforms</td>
<td></td>
<td>w0</td>
</tr>
<tr>
<td>W1</td>
<td>pos_get(), xforms</td>
<td></td>
<td>w1</td>
</tr>
<tr>
<td>W2</td>
<td>pos_get(), xforms</td>
<td></td>
<td>w2</td>
</tr>
<tr>
<td>W3</td>
<td>pos_get(), xforms</td>
<td></td>
<td>w3</td>
</tr>
<tr>
<td>W4</td>
<td>pos_get(), xforms</td>
<td></td>
<td>w4</td>
</tr>
<tr>
<td>W5</td>
<td>pos_get(), xforms</td>
<td></td>
<td>w5</td>
</tr>
<tr>
<td>WAIT</td>
<td>while input()...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WE1</td>
<td></td>
<td>w1</td>
<td></td>
</tr>
<tr>
<td>WE3</td>
<td></td>
<td>w3</td>
<td></td>
</tr>
<tr>
<td>WGRIP</td>
<td>gripdist_get()</td>
<td></td>
<td>wgrisp</td>
</tr>
<tr>
<td>WHILE</td>
<td>while [flow control]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WITH</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>movex()</td>
<td></td>
<td>movex</td>
</tr>
<tr>
<td>XREADY</td>
<td>ready()</td>
<td></td>
<td>ready</td>
</tr>
<tr>
<td>XZERO</td>
<td>zero()</td>
<td></td>
<td>zero</td>
</tr>
<tr>
<td>Y</td>
<td>movey()</td>
<td></td>
<td>movey</td>
</tr>
<tr>
<td>YAW</td>
<td>yaw()</td>
<td></td>
<td>yaws</td>
</tr>
<tr>
<td></td>
<td>jog_t(TOOL_YAW, ...)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>jog_ts(TOOL_YAW, ...)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Z</td>
<td>movez()</td>
<td></td>
<td>movez</td>
</tr>
</tbody>
</table>
| Function   | Description                          | Alias
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>@ACCEL</td>
<td>accel_get(), accels_get()</td>
<td>accel</td>
</tr>
<tr>
<td></td>
<td>accel_set(), accels_set()</td>
<td></td>
</tr>
<tr>
<td>@@CAL</td>
<td>calibrate()</td>
<td>cal</td>
</tr>
<tr>
<td>@@CALGR</td>
<td>grip_cal()</td>
<td>calgrip</td>
</tr>
<tr>
<td>@@CALSEQ</td>
<td>homeseq()</td>
<td>homeseq</td>
</tr>
<tr>
<td>@@CALZC</td>
<td>calzc()</td>
<td>calzc</td>
</tr>
<tr>
<td>@CALRDY</td>
<td>calrdy()</td>
<td>calready</td>
</tr>
<tr>
<td>@CLINACC</td>
<td>linacc_get()</td>
<td>linacc</td>
</tr>
<tr>
<td></td>
<td>linacc_set()</td>
<td></td>
</tr>
<tr>
<td>@CLINSPD</td>
<td>linspd_get()</td>
<td>linspd</td>
</tr>
<tr>
<td></td>
<td>linspd_set()</td>
<td></td>
</tr>
<tr>
<td>@CROTACC</td>
<td>linacc_get()</td>
<td>linacc</td>
</tr>
<tr>
<td></td>
<td>linacc_set()</td>
<td></td>
</tr>
<tr>
<td>@CROTSPD</td>
<td>linspd_get()</td>
<td>linspd</td>
</tr>
<tr>
<td></td>
<td>linspd_set()</td>
<td></td>
</tr>
<tr>
<td>@@DIAG</td>
<td>diagnostics</td>
<td></td>
</tr>
<tr>
<td>@GAIN</td>
<td>gains_set()</td>
<td>gain</td>
</tr>
<tr>
<td></td>
<td>gains_get()</td>
<td></td>
</tr>
<tr>
<td>@LOCATE</td>
<td>pos_set()</td>
<td>locate</td>
</tr>
<tr>
<td>@MAXSPD</td>
<td>maxvel_set(), maxvels_set()</td>
<td>maxvel</td>
</tr>
<tr>
<td></td>
<td>maxvel_get(), maxvels_get()</td>
<td></td>
</tr>
<tr>
<td>@SEEK</td>
<td>seek()</td>
<td></td>
</tr>
<tr>
<td>@SERVERR</td>
<td>get_servoerr_params()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>set_servoerr_params()</td>
<td></td>
</tr>
<tr>
<td>@@SETUP</td>
<td>split into relevant sections</td>
<td></td>
</tr>
<tr>
<td>@TRACK</td>
<td>track_spec_set()</td>
<td>setnoa</td>
</tr>
<tr>
<td>@XLIMITS</td>
<td>jointlim_get()</td>
<td>limits</td>
</tr>
<tr>
<td></td>
<td>jointlim_set()</td>
<td></td>
</tr>
<tr>
<td>@XLINKS</td>
<td>linklen_get(), linklen_set()</td>
<td>linklen</td>
</tr>
<tr>
<td>@XMAXVEL</td>
<td>maxvel_set()</td>
<td>maxvel</td>
</tr>
<tr>
<td></td>
<td>maxvel_get()</td>
<td></td>
</tr>
<tr>
<td>@@XNET</td>
<td>transputernet()</td>
<td></td>
</tr>
<tr>
<td>@XPULSES</td>
<td>xpulses_get(), xpulses_set()</td>
<td></td>
</tr>
<tr>
<td>@XRATIO</td>
<td>xratio_get()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>xratio_set()</td>
<td></td>
</tr>
<tr>
<td>@ZERO</td>
<td>zero()</td>
<td>zero</td>
</tr>
<tr>
<td></td>
<td>pos_set()</td>
<td></td>
</tr>
</tbody>
</table>
Subprograms: Category Listing

These lists give an overview of subprograms by category and can be helpful for comparing related subprograms. Since a category is focused on one set of tasks, some subprograms are listed under more than one category.

In these category listings, the descriptions of the subprograms are very brief. For a complete description, see the subprogram listing under the alphabetical listing.

On the following pages, subprograms are grouped under the following categories.

**Analog Input**

**Calibration**
Calibrating arm and gripper.

**Configuration File Handling**

**Date and Time**
Current time and date. Elapsed time in milliseconds.

**Device Input and Output**

**Digital Input and Output**

**Environment Variables**

**Error Message Handling**
Subprograms for handling error descriptors returned from subprogram calls.

**File and Device System Management**
Creating and deleting directories and objects in the file system. Mounting another file system on a directory.

**File Input and Output**
Input and output for files and devices: opening, closing, reading, writing, both unformatted and formatted with format specifiers listed. Input and output for other objects is under Device Input and Output. Input and Output for sockets is under Multi-tasking.

Subcategories include:
- Formatted Input
- Unformatted Input
- Formatted Output
- Unformatted Output

**Front Panel**
Configuring the front panel for custom operation.

**Gripper**
Operating the gripper.
Home
Homing the robot (for A465 and A255).

Location
Packing data from a location to an array and from an array to a location. Converting one type of location to another. Shifting locations in world or tool frame.
Subcategories include:
   Kinematic Conversion
   Data Manipulation
   Flags

Math
Trigonometric, logarithmic, and other math functions. Converting radians to degrees and degrees to radians.

Memory
Allocating and freeing memory. Determining and setting heap.

Motion
Subprograms designed to initiate robot motion.

Pendant
Reading characters and writing strings at the pendant. Manipulating the cursor and screen. Manipulating variables from the teach pendant.

Pointer Conversion and Function Pointers
Special subprograms to convert pointers to variables and to call functions using a pointer.

Robot Configuration
Configuring the arm: number of axes, velocities, accelerations, gains, travel limits, link lengths, etc.

Signals
Sending signals. Setting actions dependant on signals. Determining and setting signal masks.

Stance
Subprograms to adjust the robot stance. RAPL-3 uses the term “stance” for a specific set of joint angles used when reaching a location.

Status

String Manipulation
Editing, appending, copying, etc. of strings. Determining and converting case of characters and strings. Converting strings to other data types and other types to strings.

System Process Control
Subcategories include:
   Single and Multiple Processes
   Operating System Management
   Point of Control and Observation

ToolTransform and Base Offset
Base offsets and tool transform.

V3 Files
The v3 subprograms allow a program to modify a v3 file.
Win 32
These Win 32 commands allow a CROSnt process to communicate with a process in the Windows NT environment.
Analog Input

- **analog_get**: Retrieves the values of the eight analog inputs on the C500C controller.
- **boardtemp_get**: Retrieves the C500C main board temperature, in degrees Celsius.
## Calibration

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>calibrate</td>
<td>Calibrates axes.</td>
</tr>
<tr>
<td>calrdy</td>
<td>Moves the arm to the calibrate position.</td>
</tr>
<tr>
<td>calzc</td>
<td>Calibrates at next zero cross.</td>
</tr>
<tr>
<td>grip_cal</td>
<td>Calibrates the gripper.</td>
</tr>
<tr>
<td>hsw_offset_get</td>
<td>Returns the offset between homing switch and calibration position.</td>
</tr>
<tr>
<td>motor</td>
<td>Rotates a motor by a specified number of encoder pulses.</td>
</tr>
<tr>
<td>pos_get</td>
<td>Gets the position of the arm</td>
</tr>
<tr>
<td>pos_set</td>
<td>Sets the position of the arm</td>
</tr>
<tr>
<td>ready</td>
<td>Moves the arm to the READY position.</td>
</tr>
<tr>
<td>zero</td>
<td>Sets current motor position registers to 0.</td>
</tr>
</tbody>
</table>
Configuration File Handling

- **cfg_load**: Loads a text configuration file for the current application.
- **cfg_load_fd**: Loads a configuration information from a file that is already open.
- **cfg_save**: Re-writes a configuration file for the current application.
- **cfg_save_fd**: Re-writes a configuration file for the current application.
Date and Time

mtime

Obtains the time since system start-up.

time

Returns the current time.

time_set

Sets the current time.

time_to_str

Converts a system time code to an ASCII string.
Device Input and Output

<table>
<thead>
<tr>
<th>Subprogram</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>chmod</td>
<td>Changes access mode information about a file or device.</td>
</tr>
<tr>
<td>fprintf</td>
<td>Writes the specified data to the file associated with file descriptor fd.</td>
</tr>
<tr>
<td>fprintff</td>
<td>Converts and writes output to a device or file.</td>
</tr>
<tr>
<td>freadline</td>
<td>Reads (interactively) a line of characters from a file and echoes to a file.</td>
</tr>
<tr>
<td>ioctl</td>
<td>I/O control operation. Used to configure and control a device.</td>
</tr>
<tr>
<td>mknod</td>
<td>Makes a special node.</td>
</tr>
<tr>
<td>open</td>
<td>Opens a file or device and returns a file descriptor.</td>
</tr>
<tr>
<td>recv</td>
<td>Receives words from a socket.</td>
</tr>
<tr>
<td>send</td>
<td>Sends specified number of words into the socket.</td>
</tr>
<tr>
<td>sigfifo</td>
<td>Sends a signal to all of the readers at the other end of a fifo.</td>
</tr>
<tr>
<td>socketpair</td>
<td>Gets a pair of file descriptors for a private client and server socket.</td>
</tr>
</tbody>
</table>
Digital Input and Output

input
Returns the state of an input.

inputs
Returns an int that represents the bitmapped state of the digital inputs.

net_in_get
Reads input data from the F3 end of arm I/O boards.

net_ins_get
Reads all input data from the F3 end of arm I/O boards.

net_out_set
Sets a specified F3 end of arm output to a specified value.

net_outs_get
Gets the current state of a set of F3 end of arm outputs.

net_outs_set
Allows several F3 end of arm outputs to be set to a specified state at the same time.

output_get
Queries an output channel for its state. Returns the state.

output_pulse
Sets an output channel to one state, waits, and then sets the channel to the opposite state.

output_set
Sets an output channel to a state.

outputs
Sets the entire bank of output channels to states of a bitmapped value.

outputs_set
Queries the bank of output channels. Returns an int that represents the bitmapped state of the outputs.
# Environment Variables

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>environ</code></td>
<td>Allows to retrieve each individual string from its environment.</td>
</tr>
<tr>
<td><code>getenv</code></td>
<td>Allows to retrieve the value of a specified environment string.</td>
</tr>
<tr>
<td><code>setenv</code></td>
<td>Creates/redefines an environment variable's value.</td>
</tr>
<tr>
<td><code>time_to_str</code></td>
<td>Converts a system time code to an ASCII string.</td>
</tr>
<tr>
<td><code>unsetenv</code></td>
<td>Deletes the selected environment string.</td>
</tr>
</tbody>
</table>
Error Message Handling

Rapl-3 commands always return a value. A positive return value indicates that the command completed successfully. A negative return value indicates an error. Errors are designated by _error_descriptors_. Commands upon failure return the negative value of the specific error descriptor.

For example:

```c
int t
  t = open(....) ;; t is assigned the return value from the open command
if (t < 0)
  ;; it FAILED
    printf("The error descriptor is {}
        printf("And it means '{}'
end if
```

The error descriptor (-t) is a 32 bit value, divided into 4 fields, with the following bit description.

```
<table>
<thead>
<tr>
<th>msb</th>
<th>lsb</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ subsystem:7</td>
<td>[ b2:8</td>
</tr>
</tbody>
</table>
```

The Subsystem field defines the part of the system where the error originated. For example, the kernel is subsystem 0, the robot library is subsystem 1 and the robot server is subsystem 2.

Code identifies the specific error code for the given subsystem. Each subsystem has associated with it a specific list of error codes. For example, code 1 is "general error" for the kernel subsystem, and is "illegal straight line move" for the robot library subsystem.

The error codes (and their translations) are located in a set of files in the /lib/errors directory. The file names are of a standard form, "sysNNN.err", where NNN is a 3-digit 0-padded decimal number defining the subsystem. For example, kernel errors are contained in the sys000.err file, robot library errors in sys001.err, robot server in sys002.err.

The format of these files are standard. As a result given the error descriptor the error code can be determined. The first line of the subsystem sysNNN.err file contains the subsystem name. The subsequent lines contain, in sequence, the error code number EEE and an error translation.

```
Line 1: Subsystem name
Following lines: EEE error translation string
```

Where EEE is a 3-digit zero-padded decimal number corresponding to the specific code of the error descriptor. Within the error translation string, the system recognizes two special sequences: "$1" and "$2". On printing errors containing these strings, the system will replace the $1 and $2 with the decimal values of b1 and b2, respectively. For example, consider the following hypothetical error translation file, say, sys064.err:

```
This Demo System
001 Idiotic error
002 Not-so idiotic error
003 Error on robot axis $1 (I think)
004 Error on axis $2 from module $1
005 Oops!
```
When an error descriptor corresponding to the This_Demo System error 004 [0x04060504] is translated using the function str_error(), the error result is "Error on axis 6 from module 5".

Given the error descriptor returned from a failed function call the specific error code can be determined using the error handling functions. As a consequence a listing of the subsystems and their error codes are not explicitly listed. The list of errors can be obtained from sysNNN.err files in the /lib/errors directory.

The Kernel subsystem (subsystem 0) error code are specifically returned in some subprograms to denote errors. An enum type error_code_t defines the kernel subsystem errors as follows:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EOK</td>
<td>0</td>
<td>no error</td>
</tr>
<tr>
<td>ENOENT</td>
<td>2</td>
<td>no such file or directory</td>
</tr>
<tr>
<td>ESRCH</td>
<td>3</td>
<td>no process with that pid number</td>
</tr>
<tr>
<td>EINTR</td>
<td>4</td>
<td>interrupted system call</td>
</tr>
<tr>
<td>EIO</td>
<td>5</td>
<td>input/output error</td>
</tr>
<tr>
<td>ENXIO</td>
<td>6</td>
<td>no device</td>
</tr>
<tr>
<td>E2BIG</td>
<td>7</td>
<td>too many arguments or too long an argument area</td>
</tr>
<tr>
<td>ENOEXEC</td>
<td>8</td>
<td>file is not an executable</td>
</tr>
<tr>
<td>EBADF</td>
<td>9</td>
<td>bad file descriptor</td>
</tr>
<tr>
<td>ECHILD</td>
<td>10</td>
<td>no child process</td>
</tr>
<tr>
<td>EPERM</td>
<td>11</td>
<td>permission denied</td>
</tr>
<tr>
<td>ENOMEM</td>
<td>12</td>
<td>not enough memory</td>
</tr>
<tr>
<td>EACCESS</td>
<td>13</td>
<td>access denied</td>
</tr>
<tr>
<td>EBUSY</td>
<td>16</td>
<td>resource busy</td>
</tr>
<tr>
<td>EEXIST</td>
<td>17</td>
<td>file exists</td>
</tr>
<tr>
<td>EXDEV</td>
<td>18</td>
<td>link across devices attempted</td>
</tr>
<tr>
<td>ENODEV</td>
<td>19</td>
<td>operation not supported by device</td>
</tr>
<tr>
<td>ENOTDIR</td>
<td>20</td>
<td>tried to search a non-directory</td>
</tr>
<tr>
<td>EISDIR</td>
<td>21</td>
<td>tried to open a directory for writing</td>
</tr>
<tr>
<td>EINVAL</td>
<td>22</td>
<td>invalid argument</td>
</tr>
<tr>
<td>ENFILE</td>
<td>23</td>
<td>too many open files on the system</td>
</tr>
<tr>
<td>EMFILE</td>
<td>24</td>
<td>too many open files for this process</td>
</tr>
<tr>
<td>ENOTTY</td>
<td>25</td>
<td>inappropriate ioctl()</td>
</tr>
<tr>
<td>ETXTBSY</td>
<td>26</td>
<td>executable text file busy</td>
</tr>
<tr>
<td>ENOSPC</td>
<td>28</td>
<td>device out of space</td>
</tr>
<tr>
<td>ESPPIPE</td>
<td>29</td>
<td>illegal operation on fifo or socket</td>
</tr>
<tr>
<td>ERANGE</td>
<td>34</td>
<td>result out of range</td>
</tr>
<tr>
<td>EAGAIN</td>
<td>35</td>
<td>resource temporarily unavailable</td>
</tr>
<tr>
<td>ETIMEOUT</td>
<td>37</td>
<td>timed out</td>
</tr>
<tr>
<td>ENOTSOCK</td>
<td>39</td>
<td>tried to send/rcv on a non-socket</td>
</tr>
<tr>
<td>ENOSERV</td>
<td>40</td>
<td>tried to access a socket with no server</td>
</tr>
</tbody>
</table>
ENOC\textit{CLIENT} = 41 server tried to talk to a client that no longer exists or has closed the socket.

ERE\textit{SET} = 42 device is being reset

ENOT\textit{EMPTY} = 43 attempted to delete a non-empty directory

E\textit{OPNOTSUPPORT} = 45 operation not supported

The fields b2, b1 define extra data required to report specific errors. The fields b1 and b2 are not used for all (or even many) error descriptors. If not used each of the bits is set to 0. As an example, when an "axis N out" error is reported, b1 carries the number of the axis that is out.

**Error Descriptors Command Summaries**

The following subprograms exist for handling error descriptors:

- **addr\_decode**
  Looks up the address specified in the line number tables and decodes it into a line and file.

- **addr\_to\_file**
  Converts an address to a file name string.

- **addr\_to\_line**
  Converts an address to a line number.

- **err\_compare**
  Compares two error descriptors for matching subsystem and error code fields.

- **err\_compose**
  The function reconstructs and returns the original error descriptor.

- **err\_get\_b1**
  Given a +ve error descriptor, returns the value of b1.

- **err\_get\_b2**
  Given a +ve error descriptor, returns the value of b2.

- **err\_get\_code**
  Given a +ve error descriptor, returns the value of the errorcode.

- **err\_get\_subs\_sys**
  Given a +ve error descriptor, returns the number of the subsystem originating it.

- **error\_addr**
  Returns the address where the current exception occurred.

- **error\_code**
  Get the current exceptions error code

- **error\_file**
  Returns the name of the file where the current error resides.

- **error\_line**
  Gets the line number of the current error.

- **str\_error**
  Returns a pointer to a string that describes an error code.

- **str\_subs\_sys**
  Returns a string giving the name of the subsystem originating a given error code.

**Warning:** The **str\_error()** and **str\_subs\_sys()** routines share a static string variable for storing their return values. They cannot be called in the same print() or printf() method. For example:

```c
printf(".....", str\_subs\_sys(...), str\_error(...))
```
will NOT work as expected; always break these function calls into separate printf() statements.
File Input and Output

Input and output for files: opening, closing, reading, writing, both unformatted and formatted with format specifiers listed. Input and output for devices such as sockets, pipes and fifos is found in the Device Input and Output category.

Format Specifiers

The format string may consist of two different objects, normal characters which are directly copied to the file descriptor, and conversion braces which print the arguments to the descriptor. The conversion braces take the format:

\[
\{ [ \text{flags} ] [ \text{field width} ] [ .\text{precision} ] [ x | X ] \}
\]

Flags

Flags that are given in the conversion can be the following (in any order):

- – (minus sign) specifies left justification of the converted argument in its field.
- + (plus sign) specifies that the number will always have a sign.
- 0 (zero) in numeric conversions causes the field width to be padded with leading zeros.

Field width

The field width is the minimum field that the argument is to be printed in. If the converted argument has fewer characters than the field, then the argument is padded with spaces (unless the 0 (zero) flag was specified) on the left (or on the right if the – (minus sign) was specified). If the item takes more space than the specified field width, then the field width is exceeded.

.precison

The precision number specifies the number of characters in a string, the number of significant digits in a float, or the maximum number of digits in an integer to be printed.

x or X

This is the hexadecimal flag which specifies whether or not an integer argument should be printed in hexadecimal (base 16) or not. The lowercase x specifies lowercase letters (abcde) are to be used in the hexadecimal display and the uppercase X specifies uppercase letters (ABCDE).

A character sequence of {{ means to print the single { (opening brace) character.

Unformatted Input

**Readonly**

- **freadline** Reads (interactively) a line of characters from a file and echoes to a file.
- **read** Reads a number of words (4 byte entities) from a file descriptor.
- **readline** Reads (interactively) a line of characters from the standard input device, normally the terminal keyboard. Echoes to the standard output device, the terminal screen.
- **reads** Reads a string from a file.
- **readsa** Reads a string from a file and appends it to the end of another string.
- **seek** Provides a method to move through a file arbitrarily rather than sequentially.
Formatted Input

str_scanf Separates the contents of a string according to a specified format and places them into a list of pointers.

Unformatted Output

fprint Writes data to a file, exactly as given.

print Writes data to the standard output device, normally the terminal screen, exactly as given.

snprint Writes data to a string, exactly as given.

write Writes words (4 byte entities) to a file descriptor.

writeread Atomically writes words to a file descriptor and reads words from a file descriptor.

writes Writes a string to a file.

Formatted Output

fprintf Writes data to a file under a specified format.

printf Writes data to the standard output device, normally the terminal screen, under a specified format.

snprintf Writes data to a string, under a specified format.
## File and Device System Management

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>access</strong></td>
<td>Checks whether a file can be accessed in the mode specified.</td>
</tr>
<tr>
<td><strong>chdir</strong></td>
<td>Changes the current working directory to <code>path</code>.</td>
</tr>
<tr>
<td><strong>chmod</strong></td>
<td>Changes access mode of an file or device.</td>
</tr>
<tr>
<td><strong>close</strong></td>
<td>Closes file. Breaks the connection between a file descriptor and an open file.</td>
</tr>
<tr>
<td><strong>dup</strong></td>
<td>Duplicates an existing file descriptor.</td>
</tr>
<tr>
<td><strong>dup2</strong></td>
<td>Duplicates an existing file descriptor.</td>
</tr>
<tr>
<td><strong>flock</strong></td>
<td>Sets and releases advisory locks on a file.</td>
</tr>
<tr>
<td><strong>fstat</strong></td>
<td>Obtains information about a particular open object in the file system.</td>
</tr>
<tr>
<td><strong>ftime</strong></td>
<td>Changes the modification time of an open filesystem object.</td>
</tr>
<tr>
<td><strong>ioctl</strong></td>
<td>I/O control operation. Used to configure and control a device.</td>
</tr>
<tr>
<td><strong>killfifo</strong></td>
<td>Sends a signal to all readers at the other end of the fifo.</td>
</tr>
<tr>
<td><strong>link</strong></td>
<td>Makes a hard link to an existing file or directory. Useful for renaming files, moving files, or sharing data.</td>
</tr>
<tr>
<td><strong>MAJOR</strong></td>
<td>Extracts the major number from a device.</td>
</tr>
<tr>
<td><strong>MINOR</strong></td>
<td>Extracts the minor number from a device.</td>
</tr>
<tr>
<td><strong>mkdir</strong></td>
<td>Creates a new empty directory.</td>
</tr>
<tr>
<td><strong>mknod</strong></td>
<td>Makes a special node (device, fifo, socket).</td>
</tr>
<tr>
<td><strong>mount</strong></td>
<td>Mounts a file system.</td>
</tr>
<tr>
<td><strong>open</strong></td>
<td>Opens a file and returns a file descriptor.</td>
</tr>
<tr>
<td><strong>pipe</strong></td>
<td>Creates a single stream pipe.</td>
</tr>
<tr>
<td><strong>rcv</strong></td>
<td>Receives (reads) words from a socket.</td>
</tr>
<tr>
<td><strong>readdir</strong></td>
<td>Reads a directory entry and stores the structure in <code>buf</code>.</td>
</tr>
<tr>
<td><strong>rmdir</strong></td>
<td>Deletes an empty directory.</td>
</tr>
<tr>
<td><strong>seek</strong></td>
<td>Moves the starting position in a file to read or write.</td>
</tr>
<tr>
<td><strong>server_get</strong></td>
<td>For use with multiple robot systems - Gets the name of the current server name.</td>
</tr>
<tr>
<td><strong>server_info</strong></td>
<td>For use with multiple robot systems - Gets information about the current server.</td>
</tr>
<tr>
<td><strong>server_protocol</strong></td>
<td>Returns the protocol designator from the robot server.</td>
</tr>
<tr>
<td><strong>server_set</strong></td>
<td>For use with multiple robot systems - Sets the current server.</td>
</tr>
<tr>
<td><strong>server_version</strong></td>
<td>Specifies the robot server version.</td>
</tr>
<tr>
<td><strong>sigfifo</strong></td>
<td>Sends a signal to readers of a fifo.</td>
</tr>
<tr>
<td><strong>socketpair</strong></td>
<td>Gets a pair of file descriptors for a client and server socket.</td>
</tr>
<tr>
<td><strong>stat</strong></td>
<td>Obtains information about a particular object in the file system.</td>
</tr>
<tr>
<td>Subprogram</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>statfs</td>
<td>Gets information about a mounted filesystem.</td>
</tr>
<tr>
<td>send</td>
<td>Sends (writes) words to a socket.</td>
</tr>
<tr>
<td>sync</td>
<td>Flushes all the file system buffers of their contents.</td>
</tr>
<tr>
<td>unlink</td>
<td>Removes a link to a file.</td>
</tr>
<tr>
<td>unmount</td>
<td>Unmounts a file system</td>
</tr>
<tr>
<td>utime</td>
<td>Changes the modification time of a filesystem object.</td>
</tr>
</tbody>
</table>
Front Panel

There are five front panel buttons on the controller, two of which can be programmed using RAPL 3 subprograms designed for reading or setting the button status. The ARM POWER button cannot be controlled using the RAPL-3 subprograms. However, the robot_is_powered function can be used to determine, but not set, the status of the arm power.

The other buttons do not have switch position settings on or off, instead they are momentarily set buttons that only register ON (high) when they are pressed. The status of a button is high (ON) only while it is actually pressed. After it is released the status returns to 0 (OFF). The buttons are labeled with one of the following set of labels:

<table>
<thead>
<tr>
<th>Button Label</th>
<th>Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>CYCLE START</td>
<td>F1</td>
</tr>
<tr>
<td>PROGRAM RESET</td>
<td>F2</td>
</tr>
<tr>
<td>PAUSE CONTINUE</td>
<td></td>
</tr>
<tr>
<td>HOME</td>
<td></td>
</tr>
<tr>
<td>ARM POWER</td>
<td></td>
</tr>
</tbody>
</table>

The function of the buttons are identical, only the labels on the buttons are changed. The F1, F2, (CYCLE START PROGRAM RESET) buttons are user programmable. They can be programmed to have specific meanings for different applications. For instance an application can be programmed to require that one or both buttons must be pressed in order to initiate a robot movement.

The PAUSE CONTINUE button if pressed while the robot is in motion causes the robot motion to pause. For example if robot motion is initiated from the command line and then terminated from the keyboard (ALT-A or ALT_E) the operating system takes control, stops the robot, and flashes the PAUSE CONTINUE button. To initiate robot movement again the PAUSE CONTINUE button must be pressed. A message appears on the terminal requesting that the button be pressed.

Each of the buttons has an indicator light. In the case of the ARM POWER button, the light indicates the ARM POWER status. If the light is illuminated, the ARM POWER is ON. Correspondingly if the light is not illuminated, the ARM POWER is OFF. The HOME light is used to indicate that the A series robot is homed or, that the F3 robot is calibrated. The HOME button however does not cause the either robot to be homed or calibrated.

The remaining lights are programmable and have no relationship to the button status. Like the buttons the light function can be programmed using the RAPL 3 subprograms. They can be programmed to indicate certain conditions, or to illuminate when the robot is in a certain position.

Status Window

The status window on the controller, can display two hexadecimal digits. The subprogram panel_status can be used to set and test the status window. The function changes the window display but does not change the system status.
Panel Button Subprograms

The following subprograms can be used to control the front panel:

- **onbutton**
  Waits for one of the buttons to be pressed. The light can be made to blink while waiting for the light to be pressed. The light is left in the same state as when we found it.

- **panel_button**
  Returns True if the button is pressed.

- **panel_button_wait**
  Waits for a particular button to be pushed.

- **panel_buttons**
  Returns the setting of the panel buttons as a bit vector.

- **panel_light_get**
  Gets the status of a particular light.

- **panel_light_set**
  Sets the status of one particular light.

- **panel_lights_get**
  Gets the status of the controller front panel buttons.

- **panel_lights_set**
  Sets the status of the controller front panel buttons.

- **panel_status**
  Sets the front panel status display to show a specified value

**Button_enum type**

A global enumerated type variable button_enum is defined for the buttons as follows:

```plaintext
global typedef button_enum enum
    BF_1    =1,
    BF_2    =2,
    B_PAUSE_CONT =4,
    b_HOME  =8
end enum
```
Gripper

- **grip**
  - Moves servo-gripper fingers to a specified distance apart.

- **gripdist_set**
  - Moves servo-gripper fingers to a specified distance apart.

- **grip_cal**
  - Calibrates the gripper.

- **grip_close**
  - Closes the gripper.

- **grip_finish**
  - Holds program execution until gripper motion is finished.

- **grip_open**
  - Opens the gripper.

- **gripdist_get**
  - Gets the current distance between servo-gripper fingers.

- **gripisfinished**
  - Determines if the gripper is finished moving.

- **gripper_stop**
  - Stops the gripper motion

- **griptype_get**
  - Gets what the robot gripper type is currently set to.

- **griptype_set**
  - Sets the gripper type to correspond to the gripper in use: air or servo-motor.
## Home

<table>
<thead>
<tr>
<th>Subprogram</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>home</td>
<td>Homes specified axes.</td>
</tr>
<tr>
<td>homezc</td>
<td>Homes.</td>
</tr>
<tr>
<td>hsw_offset_get</td>
<td>Returns the offset between homing switch and calibration position.</td>
</tr>
<tr>
<td>robotishomed</td>
<td>Returns current home state.</td>
</tr>
<tr>
<td>zero</td>
<td>Sets all the current motor position registers to 0.</td>
</tr>
</tbody>
</table>
Location

**Kinematic Conversion**

- `joint_to_motor` Converts a location from joint angles to motor pulses.
- `joint_to_world` Converts a location from joint angles to world coordinates.
- `motor_to_joint` Converts a location from motor pulses to joint angles.
- `motor_to_world` Converts a location from motor pulses to world coordinates.
- `world_to_joint` Converts a location from world coordinates to joint angles.
- `world_to_motor` Converts a location from world coordinates to motor pulses.

**Data Manipulation**

- `here` Stores the current commanded location in a location variable.
- `loc_cdata_get` Packs cartesian data from a location into a float array.
- `loc_cdata_set` Packs cartesian data from a float array into a location.
- `loc_check` Tests the checksum of a location.
- `loc_class_get` Returns the class of a location.
- `loc_class_set` Sets the class of a location.
- `loc_pdata_get` Packs precision data from a location into an integer array.
- `loc_pdata_set` Packs precision data from an integer array into a location.
- `loc_re_check` Recalculates and resets the checksum of a location.
- `pos_axis_set` Sets the specified axis to a position.
- `pos_get` Gets the position of the robot.
- `pos_set` Sets all axes to a specified position.
- `shift_t` Alters cartesian location in tool frame of reference.
- `shift_w` Alters cartesian location in world frame of reference.

**Flags**

- `loc_flags_get` Returns the flags of a location.
- `loc_flags_set` Sets the flags of a location.
- `loc_machtype_get` Returns the machine type code of a location.
- `loc_machtype_set` Sets the machine type code of a location.
Math

These functions perform common mathematical calculations. All math functions take floating point arguments.

- **acos**: Calculates the arc cosine.
- **asin**: Calculates the arc sine.
- **atan2**: Calculates the arc tangent.
- **cos**: Calculates the cosine.
- **deg**: Converts radians to degrees.
- **fabs**: Finds the absolute value of a float.
- **iabs**: Finds the absolute value of an int.
- **ln**: Calculates the natural logarithm.
- **log**: Calculates the common logarithm.
- **pow**: Calculates a value raised to a power.
- **rad**: Converts degrees to radians.
- **rand**: A function for generating random numbers (integers).
- **rand_in**: A function for generating random numbers (integers) which fall in the range specified.
- **sin**: Calculates the sine.
- **sqrt**: Calculates the square root.
- **str_to_float**: Converts a string to a float.
- **str_to_int**: Converts a string to an integer.
- **tan**: Calculates the tangent.
Memory

heap_set
Sets the heap size of the current process.

heap_size
Returns the number of words in the heap.

heap_space
Returns the length of the longest contiguous free area in the heap.

mem_alloc
Allocates an area of memory and clears it by initializing it to zeros.

mem_free
Frees an allocated area by returning it to the pool of free space.

memcpy
Copies a block of words (4 byte entities).

memset
Sets a block of words to contain a value.

memstat
Gets information about current memory status.

pdp_get
The function gets the private data area pointer for the current thread.

pdp_set
A subroutine to set the private area memory for the current thread.

str_sizeof
Returns the number of words of memory to store a string.

sync
Flushes file system buffers.
Motion

- **align**
  Aligns “approach/depart” axis to a world axis.

- **appro**
  Moves the tool centre-point to an approach position, not in straight-line mode.

- **appro**
  Moves the tool centre-point to an approach position in straight-line mode.

- **calrdy**
  Moves the arm to the calibrate position.

- **cpath**
  Calculates and immediately executes a path.

- **ctpath**
  Creates and stores a continuous path through an array of locations with triggers for gpio (general purpose input/output).

- **ctpath_go**
  Runs a path previously stored by ctpath.

- **depart**
  Moves the tool centre-point to a depart position in joint interpolated mode.

- **departs**
  Moves the tool centre-point to a depart position in straight-line mode.

- **finish**
  Forces a command to finish before the next command is initiated.

- **grip**
  **gripdist_set**
  Moves the fingers of the servo-gripper to a specified distance apart from each other.

- **grip_close**
  Closes the gripper.

- **grip_finish**
  Holds program execution until gripper motion is finished.

- **grip_open**
  Opens the gripper.

- **gripper_stop**
  Stops the gripper motion.

- **halt**
  Stops the robot motion.

- **jog_t**
  **tx, ty, tz, yaw, pitch, roll**
  Moves the tool centre-point in the tool frame of reference, not in straight-line mode.

- **jog_ts**
  **txs, tys, tzs, yaws, pitchers, rolls**
  Moves the tool centre-point in the tool frame of reference, in straight-line mode.

- **jog_w**
  **wx, wy, wz, zrot, yrot, xrot**
  Moves the tool centre-point in the world frame of reference, not in straight-line mode.
jog_ws, wxs, wys, wzs, zrots, yrots, xrots

Moves the tool centre-point in the world frame of reference, in straight-line mode.

joint

Rotates a rotational joint a specified number of degrees, or moves a linear joint a specified number of current units.

limp

Disengages the servo control of a motor which limps that joint.

lock

Locks an axis.

motor

Rotates a motor by a specified number of encoder pulses.

move

Moves the tool centre-point to a specified location, not in straight-line mode.

moves

Moves the tool centre-point to a specified location, in straight-line mode.

nolimp

Re-engages the servo motor of a joint previously set limp.

online

Sets the online mode

pitch

In the tool frame of reference rotates (joint interpolated motion) around the orientation axis.

pitchs

In the tool frame of reference, rotates (straight line motion) around the orientation axis.

ready

Moves the arm to the READY position.

robot_abort

Stops motion and discards contents of motion queue.

robot_cfg_save

Re-writes the “/conf/robot.cfg” file with the current robot configuration information.

robot_info

Returns whether robot is done moving.

robotisdone

Returns the current robot done state

speed

Sets or gets the speed of arm motions

speed_set

Sets or gets the speed of arm motions

speed_get

Unlocks an axis.
Pendant

The pendant subprograms allow a program to use the teach pendant.

**Pendant Library Commands**

The following commands are exported from the pendant library and need the library name (stp) to be specified in the subprogram call.

- **app_close**
  Closes a pendant application so that a new one can be opened.

- **app_open**
  Selects the application specified by the argument name.

- **clear_error**
  Clears persistent error bits on the DSP

- **confirm_menu**
  Forcs the user to confirm an action before it is carried out.

- **pendant_bell**
  Sounds the pendant bell.

- **pendant_chr_get**
  Reads a character from the pendant

- **pendant_close**
  Closes the pendant in preparation for shutting down a program or the controller.

- **pendant_cursor_pos_get**
  Returns the current position of the pendant cursor.

- **pendant_cursor_pos_set**
  Move the cursor to the position specified

- **pendant_cursor_set**
  Enables or disables the pendant cursor.

- **pendant_flush**
  Flushes any 'junk' characters in the incoming buffer.

- **pendant_home**
  Moves the pendant cursor to the top left side of the pendant screen (home).

- **pendant_home_clear**
  Moves the pendant cursor to the home position and clears the screen.

- **pendant_open**
  Prepares the pendant for access and initializes it to defaults.

- **pendant_write**
  Writes a string to the pendant.

- **robot_move**
  Prepars to move the robot using the pendant

- **select_menu**
  Displays the three lines s1, s2 and s3 on the pendant screen.

- **shutdown**
  Shuts down the pendant subsystem.

- **startup**
  Initializes the pendant i/o in preparation for invoking menus.
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>teach_menu</td>
<td>Selects and teaches variables for an application.</td>
</tr>
<tr>
<td>teach_var_v</td>
<td>Similar to teach_var with the added feature that the variable is written in the location pointed to by a pointer.</td>
</tr>
<tr>
<td>var_create</td>
<td>Creates a variable</td>
</tr>
<tr>
<td>var_teach</td>
<td>Teaches a location variable.</td>
</tr>
<tr>
<td>vars_save</td>
<td>Invokes the v3_vars_save() operation on the currently open application v3 file.</td>
</tr>
</tbody>
</table>
Pointer Conversion and Function pointers

`call_ifunc` Calls an integer function through a pointer.
## Robot Configuration

Configuring the robot arm: number of axes, velocities, accelerations, gains, travel limits, link lengths coordinate systems etc.

Refer also to the Calibrate and Home Categories for specific subprograms for calibration and homing programs.

The following is a listing of the robot configuration commands. For more detail about a command refer to the alphabetical command summary listing.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>accel_get</td>
<td>Gets the acceleration for one axis.</td>
</tr>
<tr>
<td>accel_set</td>
<td>Sets the acceleration for one axis.</td>
</tr>
<tr>
<td>accels_get</td>
<td>Gets the accelerations for all axes.</td>
</tr>
<tr>
<td>accels_set</td>
<td>Sets the accelerations for all axes.</td>
</tr>
<tr>
<td>armpower</td>
<td>Enables and disables the armpower switch.</td>
</tr>
<tr>
<td>axes_get</td>
<td>Gets the number of axes.</td>
</tr>
<tr>
<td>axes_set</td>
<td>Sets the number of axes.</td>
</tr>
<tr>
<td>axis_status</td>
<td>Obtains data on all axes.</td>
</tr>
<tr>
<td>conf_get</td>
<td>Gets a list of robot configuration parameters.</td>
</tr>
<tr>
<td>gains_get</td>
<td>Gets the gains for an axis.</td>
</tr>
<tr>
<td>gains_set</td>
<td>Sets the gains for an axis.</td>
</tr>
<tr>
<td>gripisfinished</td>
<td>Determines if the gripper is finished moving.</td>
</tr>
<tr>
<td>griptype_set</td>
<td>Sets the gripper type to correspond to the gripper in use: air or servo-motor.</td>
</tr>
<tr>
<td>jointlim_get</td>
<td>Gets limits of travel of axes.</td>
</tr>
<tr>
<td>jointlim_set</td>
<td>Sets limits of travel of axes.</td>
</tr>
<tr>
<td>linacc_get</td>
<td>Returns the current value of the robot’s linear acceleration in metric or English engineering units.</td>
</tr>
<tr>
<td>linacc_set</td>
<td>Sets the current value of the robot’s linear acceleration in metric or English engineering units to the value specified by the parameter linacc.</td>
</tr>
<tr>
<td>linklen_get</td>
<td>Gets the link length for an axis.</td>
</tr>
<tr>
<td>linklen_set</td>
<td>Sets the link length for an axis.</td>
</tr>
<tr>
<td>linspd_get</td>
<td>Returns the maximum linear speed for the robot in units of mm or in. per second depending on the configuration.</td>
</tr>
<tr>
<td>linspd_set</td>
<td>Sets the linear speed for the robot in units of mm or in. per second depending on the configuration.</td>
</tr>
<tr>
<td>Subprogram</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>---------------------------------------------------------------</td>
</tr>
<tr>
<td>maxvel_get</td>
<td>Gets the maximum angular velocity for one motor.</td>
</tr>
<tr>
<td>maxvel_set</td>
<td>Sets the maximum angular velocity for one motor.</td>
</tr>
<tr>
<td>maxvels_get</td>
<td>Gets the maximum angular velocities for all motors.</td>
</tr>
<tr>
<td>maxvels_set</td>
<td>Sets the maximum angular velocities for all motors.</td>
</tr>
<tr>
<td>online</td>
<td>Sets the online mode.</td>
</tr>
<tr>
<td>robot_error_get</td>
<td>Returns the latest error state of the robot.</td>
</tr>
<tr>
<td>robot_flag_enable</td>
<td>Enables flags.</td>
</tr>
<tr>
<td>robot_info</td>
<td>Returns whether robot is done moving.</td>
</tr>
<tr>
<td>robot_mode_get</td>
<td>Gets the current mode of motion.</td>
</tr>
<tr>
<td>robot_odo</td>
<td>Gets the current value of the robot arm power odometer.</td>
</tr>
<tr>
<td>robot_servo_stat</td>
<td>Returns status of F3 servo controllers.</td>
</tr>
<tr>
<td>robot_type_get</td>
<td>Gets the current robot code for the installed kinematics.</td>
</tr>
<tr>
<td>robot_type_set</td>
<td>Sets the current robot code for the installed kinematics.</td>
</tr>
<tr>
<td>robotislistening</td>
<td>Determines if the robot server is responding to queries.</td>
</tr>
<tr>
<td>rotacc_get</td>
<td>Returns the value of the maximum rotational acceleration parameter.</td>
</tr>
<tr>
<td>rotacc_set</td>
<td>Sets the value of the maximum rotational acceleration parameter.</td>
</tr>
<tr>
<td>rotspd_get</td>
<td>Retrieves the current value of the maximum rotational speed parameter.</td>
</tr>
<tr>
<td>rotspd_set</td>
<td>Sets the value of the maximum rotational speed parameter.</td>
</tr>
<tr>
<td>server_get</td>
<td>For use with multiple robot systems - Gets the name of the current server name.</td>
</tr>
<tr>
<td>server_info</td>
<td>For use with multiple robot systems - Gets information about the current server.</td>
</tr>
<tr>
<td>server_protocol</td>
<td>Returns the protocol designator from the robot server.</td>
</tr>
<tr>
<td>server_set</td>
<td>For use with multiple robot systems - Sets the current server.</td>
</tr>
<tr>
<td>server_version</td>
<td>Specifies the robot server version.</td>
</tr>
<tr>
<td>units_get</td>
<td>Gets current setting of units: metric or English.</td>
</tr>
<tr>
<td>units_set</td>
<td>Sets current units: metric or English.</td>
</tr>
<tr>
<td>verstring_get</td>
<td>Gets the current kinematics version string.</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>xpulses_get</td>
<td>Gets the number of encoder pulses per revolution of a motor.</td>
</tr>
<tr>
<td>xpulses_set</td>
<td>Sets the number of encoder pulses per revolution of a motor.</td>
</tr>
<tr>
<td>xratio_get</td>
<td>Gets the ratio of conversion from pulses to motion of an axis.</td>
</tr>
<tr>
<td>xratio_set</td>
<td>Sets the ratio of conversion from pulses to motion of an axis.</td>
</tr>
</tbody>
</table>
## Signals

The 16 signals are listed in the Appendix.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>malarm</td>
<td>Requests that the system send the current process a specified signal after a specified delay.</td>
</tr>
<tr>
<td>sig_arm_set</td>
<td>Sets the signal to use to notify in case of an arm state change.</td>
</tr>
<tr>
<td>sig_mask_set</td>
<td>Sets a signal mask and returns the old signal mask.</td>
</tr>
<tr>
<td>sigfifo</td>
<td>Sends a signal to all of the readers at the other end of a fifo.</td>
</tr>
<tr>
<td>sigmask</td>
<td>Returns the correct mask for a signal.</td>
</tr>
<tr>
<td>signal</td>
<td>Sets an action to be performed when a signal is received.</td>
</tr>
<tr>
<td>sigsend</td>
<td>Sends a signal to a process.</td>
</tr>
<tr>
<td>str_signal</td>
<td>Returns a pointer to a string that describes a signal.</td>
</tr>
<tr>
<td>WIFSIGNALED</td>
<td>Determines if the child process was signal-terminated.</td>
</tr>
<tr>
<td>WTERMSIG</td>
<td>Returns the actual signal number that signal-terminated a child process.</td>
</tr>
</tbody>
</table>
Stance

Use of the Term “Stance”
RAPL-3 uses the term "stance" for a specific set of joint angles used when reaching a location. This is a change from RAPL-II that used "pose". ISO standard 8373, Manipulating Industrial Robots – Vocabulary, reserves “pose” for a different meaning.

stance_get

Returns the current stance of the robot.

stance_set

Sets the arm to a specified stance.
Status

robot_error_get
Returns the current (latest) error state of the robot.

robot_odo
Gets the current value of the robot arm power odometer.

robotisdone
Returns the current robot done state.

robotisfinished
Returns the current finished state of the robot.

robotishomed
Returns current home state.

robotislistening
Determines if the robot server is responding to queries.

robotispowered
Returns the current state of the robot arm power.

verstring_get
Gets the current kinematics version string.
String Manipulation

- `chr_is_lower`: Determines whether letter character is lower case.
- `chr_is_upper`: Determines whether letter character is upper case.
- `chr_to_lower`: Converts letter character to lower case.
- `chr_to_upper`: Converts letter character to upper case.
- `sizeof`: Returns the size, in RAPL-3 words, of its argument.
- `str_append`: Appends one string to another string.
- `str_chr_find`: Finds the first occurrence of a character in a string.
- `str_chr_get`: Returns the ASCII value of a specified character in a string.
- `str_chr_rfind`: Finds the last occurrence of a character in a string.
- `str_chr_set`: Sets the value of a specified character in a string.
- `str_cksum`: Computes a 32-bit bytewise checksum of the characters of a string.
- `str_dup`: Allocates space for a string, copies it into the allocated space and returns a pointer to the new string.
- `str_edit`: Replaces a specified part of a string with another string.
- `str_error`: Returns a pointer to a string that describes an error code.
- `str_len`: Returns the length of a string.
- `str_len_set`: Sets the length of a string.
- `str_limit`: Returns the limit on the length of a string.
- `str_limit_set`: Sets the limit on the length of a string.
- `str_scanf`: Separates a string according to a format and places into variables.
- `str_signal`: Returns a pointer to a string that describes a signal.
- `str_sizeof`: Returns the number of words of memory to store a string.
- `str_substr`: Copies a substring (a specified part of a string).
- `str_subsys`: Given a specific error descriptor, the function returns a string giving the name of the subsystem origination the error.
- `str_to_float`: Converts a string to a float.
- `str_to_int`: Converts a string to an integer.
str_to_lower          Converts string to lower case.
str_to_upper          Converts string upper case.
time_to_str           Converts a system time code to an ASCII string
System Process Control

Single and Multiple Processes

Splitting a program.

abort
Returns its argument value.

argc
Returns the number of command-line arguments to the program.

argv
Returns a pointer to the nth command-line argument to the program.

delay
Sleeps for at least the number time specified (millisecond)s.

exec
Loads and executes another program that is given in path.
Use this command when all the command-line arguments are known.

exec
Loads and executes another program that is given in path.
Use this command when all the command-line arguments are not known.

exit
Causes normal program termination.

get_ps
Gets the process status information from a process table.

getopt
Provides a mechanism for handling command line arguments and options.

getpid
Gets the process identification number of the calling program.

getppid
Gets the process identification number of the parent of the calling program.

memstat
Gets information about the current system memory status.
Returns the number of 64 byte units.

module_name_get
Gets the name of the module performing the subroutine call.

msleep
Sleeps for the time specified and then returns to the main program.

robot_error_get
Returns the current (latest) error state of the robot.

sem_acquire
Attempts to acquire a semaphore.

sem_release
Releases a semaphore.

sem_test
Tests a semaphore.

setprio
Sets the priority of a process.

split
Creates a duplicate child process of the current process.
waitpid
Waits for a child process to complete.

WEXITSTATUS
Returns the actual exit code of the child process that exited.

WIFEXITED
Determines if the child process has been exited.

WIFSIGNALED
Determines if the child process was signal-terminated.

Operating System Management
Getting and setting process identification and priority.

setprio
Sets the priority of a process

sigsend
Sends a signal to a process.

socketpair
Gets a pair of file descriptors for a private client and server socket

sysconf
Obtains system configuration information.

sysid_string
Returns a string describing a specified system id.

va_arg_get
Gets the next varargs argument.

va_arg_type
Returns a type descriptor for the next varargs argument.

Point of Control and Observation
These routines get or release point of control or point of observation. Any command which “writes” to the robot (moves, re-sets parameters, etc.) requires point of control. Only one process can have point of control at one time. If one process has point of control, another process requesting point of control will be denied point of control (ctl_get() will fail with an EBUSY error condition).

All library functions which require point of control explicitly ask for it, so there is typically no need for the user to perform this task.

ctl_get
Gets point of control.

ctl_give
Gives control explicitly to the process specified by the pid parameter.

ctl_rel
Releases point of control.

obs_get
Gets point of observation.

obs_rel
Releases point of observation.
Tool Transform and Base Offset

- **base_get**: Gets the current base offset.
- **base_set**: Sets the base offset.
- **tool_get**: Gets the current tool transform, the redefinition of the origin point and the orientation of the tool coordinate system.
- **tool_set**: Sets a tool transform, a redefinition of the origin point and the orientation of the tool coordinate system.
v3 Files

The v3 subprograms allow a program to modify a v3 file.

These v3 subprograms are the same subprograms that are used by the teach pendant and the application shell when you use those tools to modify the teachable variables in a v3 file.

Before modifying a v3 file from a program, ensure that this is necessary.

Background

v3 files have a very specific use.

The v3 File

A v3 file contains the values for the teachable variables of a program. Teachable variables can include: cartesian locations, precision locations, integers, floats, and strings, both scalar and array.

Variables are declared teachable so that their values can be stored outside the program, modified (normally by the teach pendant or the application shell), and used for initializing.

Teaching Variables

The advantage of having variables in a v3 file is being able to modify values outside the program. The primary advantage is being able to teach locations. Using the teach pendant or the application shell, you can move the arm and, with the teach pendant’s teach selection or ash’s here command, have the data of the current position packed into the location variable.

Initializing Variables with the v3 File

In the CROS/RAPL-3 environment, a v3 file is used to initialize teachable variables of a program, at the moment when the program is readied to run. After that, the v3 file is not used. Any changes made to a v3 file have no effect on a program unless the program is run again. When it is run again, the v3 file is used to initialize the teachable variables of the program, again, at the moment when the program is readied to run.

Modifying and Using Variables

Any variable, whether cloc, ploc, int, float, or string, whether declared as teachable or unteachable, can be modified and used within a program independent of any v3 file.

Locations do not all have to be taught. For example, for a pallet (rows x columns of locations) you could teach three corner locations, or for a microplate carousel you could teach the top and bottom locations, and calculate the intermediate locations. These calculated locations can be used in motion commands like any other location variable.

To avoid calculating during each run of the program, you can store the variables.

Storing Variables in Any File

To store variables between runs of a program, or between the running of a set-up program and the application program, the variables must be stored in a file. You do not need to store them in a v3 file. Variables can be written out to a data file and read in from that file with the regular file i/o subprograms.

Even though you can modify a data file from another RAPL-3 program or from another kind of file editing program, you cannot load this file into an application
shell database or teach pendant database for the variables to be modified by the application shell or the teach pendant.

**Storing Variables in a v3 File**
You must use the v3 file when you want to store variables outside the program and also have them accessible using the teach pendant or the application shell.

**Modifying a v3 File from a Program**
There are instances where a v3 file must be modified from a program. One is a situation where locations are determined by the program and need to be available later for use by the teach pendant or the application shell. Another is a situation where, as the program is running, the locations need to be monitored and corrected and these corrected locations need to be used at the next running of the program.

**Using These v3 Subprograms**
To properly modify a v3 file, several of these v3 subprograms must be used in a certain order.

*From a program, modify a v3 file carefully.* An incorrect routine can result in a corrupted v3 file and lost data. You have to construct routines similar to the teach pendant and application shell routines that ensure that the v3 file is properly modified.

**Architecture for v3 Subprograms**
The following files and structures are part of the v3 architecture.

**Program File**
The program file is the executable file containing sub, func, and command calls and other parts of the program. If the program file has any teachable variables, data structures can be created for a corresponding v3 file.

**v3 File**
The v3 file is the file that stores the data structures of teachable variables. The v3 file is used to initialize teachables in a program, as the program is readied to run.

**Backing Store File**
“Backing store file” is another term for the v3 file, highlighting its role as a backup, stored in the file system while the data structures are in memory and being manipulated by v3 commands.

**Incore File**
The incore file is the set of data structures loaded in memory. This “file” is the in-core-memory equivalent to the v3 file stored in the file system, but also has a control block. The file is a linked list of records.

**Control Block**
A structure that contains data about the file, the records, and modifications. There is one control block.

**Record**
A structure that contains data about a variable: its basetype, its identifier, its value, etc. There are as many records as there are teachable variables.

**Parameters**
Commands, functions, and subroutines that manipulate v3 files use the following structs as parameters.
v3_cb

The v3_cb struct is the control block.

```c
v3_cb struct
  v3_incore@ head Head of the linked list
  int entries How many entries in the list (not counting the list head)
  int locks How many v3_lock() calls have been done. The file is unlocked until this count reaches 0 again
  int fd fd of the open file descriptor. -1 is none.
  int dirty In-core data cleanliness flag. 0 is clean, 1 is data only, 2 is structure change.
  v3_header h Header, read from the file. Note: the size of this section is variable depending on the size of the header (sourcename)
end struct
```

v3_incore

The v3_incore struct is the record when loaded in core.

```c
v3_incore struct
  v3_incore@ next For linking.
  v3_incore@ prev For linking.
  int offset Offset in the file where the record is located. 0 is not yet in the file
  void@ valptr The value part of this record.
  v3_record v The v3_record itself. Note that sizeof(this field) gives misleading results since the full name and the data block are stored contiguously here to cut overhead.
end struct
```

Subs, Funcs, and Commands

Opening and Closing Files

These subprograms manage the storage file and the in-core file.

```c
v3_extract Builds data structures from the program file.
v3_f_close Closes the storage file.
v3_f_disconnect Disconnects the storage file from the in-core file.
v3_f_free Frees memory by deleting the in-core file.
v3_f_modified Checks the file for modifications.
v3_f_open Loads a storage file into core memory.
v3_f_save Saves an in-core file to a storage file.
v3_lock Locks the file.
```
v3_new          Creates a new set of core block structures.
v3_save_on_exit Sets the RAPL-3 interpreter so that when the program exits, all of its final v3 variable values will be saved to the specified v3 file.
v3_unlock       Unlocks the file.

Modifying Variables
These subprograms modify variables in the in-core file.

v3_append_lists Appends a second list onto a first list.
v3_create_variable Creates a new variable.
v3_delete_variable Deletes a variable and its value from the list.
v3_find_variable Finds a specified variable.
v3_get_first     Gets the first node on the list.
v3_get_info      Gets information about the in-core structures.
v3_get_next      Gets the next node on the list.
v3_get_prev      Gets the previous node on the list.
v3_get_value_p   Gets the pointer to the value element of an in-core node.
v3_mark_taught   Marks an in-core node as taught.
Win 32

These Win 32 commands allow a CROSnt process to communicate with a process in the Windows NT environment.

The named pipe driver DLL allows servers to be written in RAPL-3 and have non-RAPL-3 based clients. A named pipe is a Win32 inter-process communication object that allows two processes (which do not have to be running on the same machine) to transfer information between each other. The client-server mechanism is used in this form of communication.

Named pipes provide two mechanisms for data transfer: byte-by-byte and message based. Byte-by-byte sends data through the pipe on a byte-by-byte basis. Message based transfers the entire data in one operation. Message based reads can only be used if messaged based writes on the other end of the pipe are enabled.

All transfers are done in overlapped i/o mode. This means that unless the operation can be completed immediately, it is placed in the background. When the operation is complete, a signal is sent to the process that started the operation.

Normal read(), write(), reada(), readsa(), and other i/o operations can be used with named pipes. The read and write calls can return an error, 0 if the I/O operation is placed in the background, or the number of words actually read.

Further Windows NT Information

On the subject of named pipes in Windows NT, refer to Windows NT (Win 32) documentation.

File System Mounting

For commands on mounting a CROSnt file system on a Windows NT file system, see File and Device System Management.

Win 32 Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>connectnp</td>
<td>Checks or waits for a client to connect with the named pipe.</td>
</tr>
<tr>
<td>closenp</td>
<td>Closes a named pipe</td>
</tr>
<tr>
<td>disconnectnp</td>
<td>Breaks a pipe connection with a client.</td>
</tr>
<tr>
<td>opennp</td>
<td>Opens a named pipe in the Windows NT domain.</td>
</tr>
<tr>
<td>statusnp</td>
<td>Returns the current status of a named pipe</td>
</tr>
</tbody>
</table>

See also Device Input and Output for read(), write(), reada(), readsa(), and other i/o operations.

Types Used With Win 32 Commands

The following types are used with the Win 32 commands.

NPIPE_MODES

```c
global typedef NPIPE_MODES enum
  M_READ_MESSAGE = 1
  M_WRITE_MESSAGE = 2
end enum
```
NPIPE_STATUS

global typedef NPIPE_STATUS enum
    NPIPE_OPENED = 0x0001,
    NPIPE_CONNECTED = 0x0002,
    NPIPE_CONNECT_PENDING = 0x0100,
    NPIPE_READ_PENDING = 0x0200,
    NPIPE_WRITE_PENDING = 0x0400,
    NPIPE_TRANSACT_PENDING = 0x0800,
    NPIPE_OPERATION_PENDING = 0x0F00
end enum
Subprograms of the CRS-supplied libraries are listed in alphabetical order on the following pages.
Reading Subprogram Entries
Each subprogram is described in the following format.

name_of_subprogram
Another name for the same subprogram. With some alias entries, there is a
cross-reference from the alias entry to the original entry which contains the full
description of the subroutine, function, or command.

Description
A description of the functionality of this subroutine, function, or command.

Caution
A characteristic that could create a problem.

Library
The library if the subprogram has export scope.

Syntax
The subprogram’s declaration in the library. The declaration follows the rules for
subprogram declarations.

The declaration declares the scope of the subprogram. A few subprograms have
export scope. They are explicitly listed as such and must be called by naming the
library with the subprogram. All other subprograms have global scope. Since
they are visible to all programs, they are called by naming the subprogram only.

The declaration declares whether the subprogram is a subroutine, function, or
command. This determines whether it does not return a value, returns a value,
or returns a success/error integer under the system’s error checking.

If the subprogram is a func, it declares the type of return value: int, float,
location, or pointer.

Next, the declaration names the subprogram with a unique identifier.

Within parentheses the declaration lists parameter(s), giving the type of
parameter and an identifier. The commas separating parameters are required
syntax. Three dots ( . . . ) indicate a variable number of parameters which are
described in the following parameter list.

Parameters
A list with explanations and types.

Arguments
Distinctions are made between parameters passed by value and parameters
passed by reference (var parameters). If a parameter passed by reference is
packed, expected values of the parameter are listed.

With subprograms that are able to take a variable number of parameters
(varargs), distinctions are made between required parameters and optional
parameters.

Parameters are also called arguments.

Returns
The return value of the function or command which indicates success (zero or
positive) or failure (negative).

If a zero or positive value carries specific meaning, it is described.

If a negative value is returned for a specific reason, it is described.

Example
An example of use in a program.

Result
The example’s result.

System Shell
If applicable, an equivalent command in the CROS/RAPL-3 system shell or
application shell, described in the Robot System Software Documentation Guide.
RAPL-II
Any similar RAPL-II commands.

See Also
Any related RAPL-3 subroutines, functions, commands, statements, keywords, or topics, described in this Reference Guide.

Category
The category of this subprogram. All subprograms are briefly listed with related subprograms in the category section.

Using Subprograms
To use the subprogram in your program, call the subprogram by name with parameter(s)/argument(s) of the type indicated. To use an export subprogram, precede the subprogram call with the library name.

Follow the syntax and parameter descriptions, or modify an example.

Required characters are in non-italic monospace font. Programmer-supplied identifiers and constructs are in italics. Optional items are in [square brackets], except for arrays. The continuation character can be used.
aborted

Description
This is a utility command that simply returns its argument value. Since abort() is a RAPL-3 command, a negative argument to abort() will cause a command failure exception at the line where abort was called. If abort() is passed a positive or zero argument, then it does nothing.

Syntax
command abort( int err )

Parameters
err the monitored return value: an int

Returns
The value of the parameter.

Example
if (check_status() > 0)
  n = 1
else
  n = -1
end if
abort(n) ;; will cause an exception if n is -1

RAPL-II
ABORT terminates a program, but not under any system error checking.

See Also
exit terminates program normally

Category
System Process Control: Single and Multiple Process

accel_get

Description
Gets the acceleration for one axis. The units are in deg/sec^2.

Syntax
command accel_get( int axis, var float dst )

Parameters
axis the axis being inquired: an integer
dst a float -packed with the acceleration in

Returns
Success >= 0. The parameter is packed.
Failure < 0

Example
float curr_accel
accel_get(5,curr_accel)

Application Shell
Same as accel.

See Also
accels_get gets the accelerations for all axes
accel_set sets the acceleration for one axis
accels_set sets the accelerations for all axes

Category
Robot Configuration

accel_set

Description
Sets the acceleration for one axis.

<table>
<thead>
<tr>
<th>Joint</th>
<th>F3</th>
<th>A465</th>
<th>A255</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Default</td>
<td>Minimum</td>
<td>Default</td>
</tr>
<tr>
<td>1</td>
<td>879</td>
<td>1758</td>
<td>720</td>
</tr>
<tr>
<td>2</td>
<td>879</td>
<td>1758</td>
<td>720</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>500</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>500</td>
</tr>
</tbody>
</table>
### accel_set

**Syntax**

```
command accel_set(int axis, float accel_in)
```

**Parameters**

- `axis`  
  the axis being set: an int
- `accel_in`  
  the acceleration for that axis in deg/sec²: a float

**Note:** If `accel_in` is less than 10% of the default acceleration value, the value will be set to 10% of the default instead.

**Returns**

- Success `>= 0`
- Failure `< 0`

**Example**

```
accel_set(1, 879)
```

**RAPL-II**

Similar to `@ACCEL`.

**See Also**

- `accel_get` gets the acceleration for one axis
- `accels_get` gets the accelerations for all axes
- `accels_set` sets the accelerations for all axes

**Category**

Robot Configuration

### accels_get

**Description**

Gets the accelerations for all axes. The units are in deg./sec².

**Syntax**

```
command accels_get(var float[8] accels)
```

**Parameters**

- `accels`  
  the accelerations of the axes in deg/sec²: an array of floats

**Returns**

- Success `>= 0`. The parameter is packed.
- Failure `< 0`

**Example**

```
float[8] curr_accels
accels_get(curr_accels)
```

**Application Shell**

Same as `accel`

**See Also**

- `accel_set` sets the acceleration for one axis
- `accels_set` sets the accelerations for all axes

**Category**

Robot Configuration

### accels_set

**Description**

Sets the accelerations for all axes. The units are in deg./sec²:

<table>
<thead>
<tr>
<th></th>
<th>F3</th>
<th>A465</th>
<th>A255</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Default</td>
<td>Maximum</td>
<td>Default</td>
</tr>
<tr>
<td>1</td>
<td>879</td>
<td>1758</td>
<td>720</td>
</tr>
<tr>
<td>2</td>
<td>879</td>
<td>1758</td>
<td>720</td>
</tr>
<tr>
<td>3</td>
<td>879</td>
<td>2637</td>
<td>720</td>
</tr>
</tbody>
</table>
Syntax
command accels_set( var float[8] accels )

Parameters
accels the accelerations for the axes in deg./sec.²; an array of floats
Note: If any element of accels is less than 10% of the default acceleration value for that axis, the value will be set to 10% of the default instead.

Returns
Success >= 0
Failure < 0

Example
float[8] new_accels = {500, 500, 500, 4500, 9000, 0, 0, 0}
accels_set(new_accels)

RAPL-II
Similar to @ACCEL.

See Also
accel_get gets the acceleration for one axis
accels_get gets the accelerations for all axes
accel_set sets the acceleration for one axis

Category
Robot Configuration

access

Description
Checks to see if the file specified in path can be accessed in the way specified by mode.

Syntax
func int access( var string[] path, a_modes mode )

Parameters
path the filename: a variable length string
mode the access mode, of type a_modes:
- F_OK file exists
- X_OK file is executable
- W_OK file is writeable
- R_OK file is readable

Returns
0 Success. The file exists and can be accessed in mode.
-EINVAL Some of the arguments are illegal (bad mode or file path.)
-ENOTDIR One of the components in path was not a directory.
-ENOENT The file denoted by path did not exist.
-EIO An I/O error occurred.
-EACCESS The access specified by mode is not allowed

Example
string[] path = "filename"
... if access( path, F_OK ) == 0
  ;; File Exists
  if access( path, X_OK ) == 0
    ;; File is executable
  end if
  if access( path, W_OK ) == 0
    ;; File is writeable
  end if
  if access( path, R_OK ) == 0
    ;; File is readable
  end if
acos

Description
Calculates the arc cosine of a float.

Argument Range: \( +1.0 \geq \text{argument} \geq -1.0 \)

Syntax
```
func float acos( float x )
```

Returns
Success \( \geq 0 \). The arc cosine of the argument, an angle in degrees.
Failure \( < 0 \)

Example
```
float x = 0.965926
printf("acos of 0.965926 = {}\n",acos( x ))
```

Result
15.000

See Also
asin calculates the arc sine
atan2 calculates the arc tan
cos calculates the cosine

Category
Math

addr_decode

Description
A subroutine for troubleshooting errors. Looks up the address specified in the line number tables and decodes it, if possible, into a line and file. Note that if the string sp is NULL, no file name is copied.

Syntax
```
sub addr_decode(int address, var int line, string[]@ sp)
```

Parameter
- address int defining the address to look up in the line tales
- line int gets packed with the line number
- sp string pointer specifying the file to write the decoded line to.

Returns
nothing. “line” is set to 0 on failure; sp@ (if sp is not NULL) is set to “” on failure.

Example
```
int lnum
string[64] fname

try
    ;;
    ;; some code here...
    ;;

except
    printf("Error {} ({}) happened\n", -error_code(),
            str_error(-error_code()))
    addr_decode(error_addr(), lnum, fname)
    printf(" at line {} of file {}
", lnum, fname)
end try
```

Result
If an error occurs in the try block, the error and its name and the line and file where it occurred will be printed.

See Also
error_code() find the error descriptor of an exception that has occurred
error_addr() find the address where an exception occurred
str_error() convert an error descriptor into a string
addr_to_file

Description
Calls the addr_decode subroutine to convert the given address to a file name string. This provides a simpler interface to addr_decode() for getting at the name of a file where an exception has occurred.

Syntax
func string[] @ addr_to_file(int addr)

Parameter
addr an int which specifies the address which is to be converted to a file name

Returns
A pointer to a string containing the file name, or a pointer to an empty string if it fails.

Example
;; in the except block of a try-except construct:
    printf("The exception happened at line {} of file {}
    addr_to_line(error_addr()), addr_to_file(error_addr()))

Result
The line and file where the exception occurred are printed.

See Also
addr_decode()
error_addr()

Category
Error Message Handling

addr_to_line

Description
A function that calls the addr_decode function to convert an address to a line number.

Syntax
func int addr_to_line(int addr)

Parameter
addr an int specifying the address to be converted to a line number.

Returns
The correct line number, or 0 if it fails.

Example
see addr_to_file()

See Also
addr_decode()
addr_to_file()
error_addr()

Category
Error Message handling

align

Description
Aligns the “approach/depart” tool axis parallel to an axis of the world coordinate system.

The “approach/depart” tool axis is a specific axis of the tool coordinate system. With no tool transform set (the tool coordinate system is at its default, identical to the mechanical interface coordinate system), the “approach/depart” tool axis is the axis arising off of, and perpendicular to, the tool flange (mechanical interface). The F3 tool coordinate system (which is similar to a recent international standard) and the A465/A255 tool coordinate system (which is an earlier pre-standard system) are different.

- F3: the “approach/depart” tool axis is the Z axis of the F3 tool coordinate system. The axes of the tool coordinate system are parallel to the
corresponding axes of the world coordinate system when the arm is in the
calrdy position (straight up).

- A465 or A255: the “approach/depart” tool axis is the X axis of the A465/A255
tool coordinate system. The axes of the tool coordinate system are parallel to
the corresponding axes of the world coordinate system when the arm is in the
ready position.

With no tool transform set the “approach/depart” tool axis is the axis
perpendicular to, the tool flange (A-series tool X axis, F-series tool Z axis). The
align() command aligns the approach/depart axis with the world axis specified.

If a tool transform has been set, the tool coordinate system is transformed from
the default setting and the align() command aligns the transformed
“approach/depart” tool axis parallel to an axis of the world coordinate system.

The world axis for alignment is specified with a parameter.

The align() command moves the arm in joint-interpolated motion. The tool centre
point’s start and end point are the same, but the tool centre point travels as a
result of various joint motions, not in straight line mode.

**Syntax**

```
command align ( int speed, align_axis_t axis [, coord_t] )
```

**Parameters**

- `speed` the speed during align, percentage of full speed
- `axis` the axis to align to, one of:
  - `ALIGN_NEAR` aligns to the closest axis of the world coordinate system
  - `ALIGN_X` aligns to the + X axis of world coordinate system
  - `-ALIGN_X` aligns to the – X axis of world coordinate system
  - `ALIGN_Y` aligns to the + Y axis of world coordinate system
  - `-ALIGN_Y` aligns to the – Y axis of world coordinate system
  - `ALIGN_Z` aligns to the + Z axis of world coordinate system
  - `-ALIGN_Z` aligns to the – Z axis of world coordinate system

**Optional Parameter**

- `coord_t`

**Returns**

- Success >= 0
- Failure < 0

**Example**

```
align(_Z) ;; aligns to the Z axis
align(ALIGN_NEAR) ;; aligns to the closest axis
```

**RAPL-II**

Similar to ALIGN.

**See Also**

- `tool_set` re-defines the tool coordinate system

**Category**

- Motion

---

**analogs_get**

**Description**

Retrieves the values of the eight analog inputs (2 of which are available to the
user) on the C500C controller.

**Syntax**

```
command analogs_get( var float[8] values )
```

**Related Definitions**

The following defined symbols give which channel is which:

- `ANA_USER1` -- user analog input 1
- `ANA_USER2` -- user analog input 2
- `ANA_SGAFEEDBACK` -- servo gripper feedback input
- `ANA_BATTERYVOLT` -- lithium backup battery (volts)
- `ANA_V24SUPPLY` -- 24 volt supply (volts)
- `ANA_V12SUPPLY` -- 12 volt supply (volts)
ANA_V5SUPPLY -- 5 volt supply (volts)
ANA_BOARDTEMP -- main board temperature (Celsius)

Returns
Success >= 0; the values[] array filled in with the input readings.
Failure < 0 (-ve error code)

Example
float[8] vals
...
analogs_get(vals)
printf("The board temperature is {} Celsius\n", vals[ANA_BOARDTEMP])

See Also
boardtemp_get()

Category
Analog Input

app_close

Description
Closes a pendant application so that a new one can be opened. Only one application can be open at any given time.

Library
stp

Syntax
export command app_close()

Parameters
None

Returns
Success >= 0
Failure < 0

Example
string[10] name = "my_app_23"
stp:startup
stp:app_open(name, 0)
...
stp:app_close()
...

Result
The current application being accessed from the pendant is closed.

See Also
pendant_close
start_up
app_open

Category
Pendant

app_open

Description
Selects the application specified by the argument name. If the application does not exist and the create parameter is true then create the application. An error code is returned if the application is not found.

Library
stp

Syntax
export command app_open(var string[] name, int create)

Parameter
create_flag
1 create is true
0 create is false

Returns
Success >= 0
Failure < 0

Example
...
stp: app_open("New_Path", 0)
...
Result
If an application New_Path exists, it is selected, if it does not exist, the return is an error descriptor.

See Also
app_close()

Category
Pendant

appro
Description
Moves the tool centre-point to an approach position. The approach position is defined by a location, and a distance from that location along the “approach/depart” tool axis.

Moves in joint-interpolated mode (tool centre-point curves through space as necessary as a result of joint changes). The motion is not cartesian-interpolated (straight-line).

Used to move the arm, usually quickly, to a position near a location before moving the tool, usually slowly, to the location.

Syntax
command appro( gloc location, float distance )

Parameter
location the target location: a cloc or ploc
distance the distance from the location to the approach position: a float

Returns
Success >= 0
Failure < 0

Example
appro(rack_5, 100.0) ;; millimetres
appro(tray_1, 4.0) ;; inches

RAPL-II
Similar to APPRO.

See Also
appros like appro(), but in straight line motion
depart moves to depart position; opposite of appro
departs moves to depart position; opposite of appros
tool_set re-defines the tool coordinate system

Category
Motion

appros
Description
Moves the tool centre-point to an approach position. The approach position is defined by a location, and a distance from that location along the “approach/depart” tool axis.

Moves in cartesian-interpolated mode (straight line motion). The motion is not joint-interpolated (tool centre-point curves through space as necessary as a result of joint changes).

Used to move the arm, usually quickly, to a position near a location before moving the tool, usually slowly, to the location.

Syntax
command appros( gloc location, float distance )

Parameter
location the target location: a cloc or ploc
distance the distance from the location to the approach position: a float

Returns
Success >= 0
Failure < 0

Example
appros(rack_5, 100.0)
appros(tray_1, 4.0)

RAPL-II
Similar to APPRO.

See Also
move like moves(), but not in a straight line
depart moves to depart position; opposite of appro
departs moves to depart position; opposite of appros
tool_set re-defines the tool coordinate system

Category
Motion

argc

Description
Returns the number of command-line arguments to the program. The program name is included as an argument.

Reminder: Arrays are indexed by zero; The following code segment will produce an error:
```
num_args = argc()
args = argv( num_args )
```

Syntax
```
func int argc()
```

Returns
Always succeeds. Returns the number of command line arguments.

Example
```
;; program name: ex_argcv
;; the following example prints out the command line arguments
;; including the name of the process.
main
  const MAX_COUNT = 10
  int num_args, count = 0
  string[0][10] arg_ptr ;; maximum of 9 arguments
                       ;; in addition to the name
  num_args = argc() ;; get num. of line args.
  printf ("number of arguments {}

  while (count<num_args) && (count<MAX_COUNT)
    arg_ptr[count] = argv(count) ;; initialize ptr to string
                                   ;; increment index count
    printf ("arg {8}: {8}\n",count,arg_ptr[count])
    count ++
  end while
end main
```

Result
A command line of “ex_argcv 11 22 33” will produce the following output:
```
arg 0:   ex_argcv
arg 1:    11
arg 2:    22
arg 3:    33
```

See Also
argv returns a pointer to a command-line argument

Category
System Process Control: Single and Multiple Processes

argv

Description
Returns a pointer to the nth command-line argument to the program. By convention, argv(0) is the name of the program itself.

Syntax
```
func string[]@ argv( int n )
```

Returns
Returns a NULL pointer on failure, or a pointer to the string on success.

Example
```
;; program name: ex_argcv
;; the following example prints out the command line arguments
```

main
    const MAX_COUNT = 10
    int num_args, count = 0
    string[10] arg_ptr

    num_args = argc();

    printf("number of arguments \n",num_args)
    while (count < num_args) && (count < MAX_COUNT)
        arg_ptr[count] = argv(count);
        printf("arg {0}: {1}\n",count,arg_ptr[count]);
        count ++;
    end while
end main

Result
    a command line of "ex_argcv 11 22 33" will produce the following output:
    arg 0: ex_argcv
    arg 1: 11
    arg 2: 22
    arg 3: 33

See Also
    argc: returns the number of command-line arguments

Category
    System Process Control: Single and Multiple Processes

armpower

Description
    Enables and disables the armpower switch. As long as one process has the arm power OFF, arm power cannot be turned on.

Syntax
    command armpower( Boolean switch )

Parameter
    switch: Boolean, one of:
    OFF: disables the arm power (turns it off and keeps it off)
    ON: enables arm power (allows arm power to be turned on)

Returns
    Success = 0
    Failure < 0

Example
    armpower(OFF)
    ...
    armpower(ON)

RAPL-II
    Same as ENABLE/DISABLE ARM and ARM ON/OFF.

Category
    Robot Configuration

asin

Description
    Calculates the arc sine of a float.
    Argument Range: +1.0 \geq argument \geq -1.0

Syntax
    func float asin( float x )

Returns
    Success \geq 0  The arc sine of the argument, an angle in degrees.
    Failure < 0
Subprograms: Alphabetical Listing

Example

```c
float x = 0.422618
float y
printf("asin of 0.422618 = {}\n",asin( x ))
```

Result

25.0000

RAPL-II

ASIN

See Also

acos  calculates the arc cosine
atan2  calculates the arc tan
sin    calculates the sine

Category

Math

```c
p      an int.
pstr   the : a pointer to a string.
f      the : a pointer to a string.
l      the : an int.
```

atan2

**Description**
Calculates the arc tangent of a float, an angle in radians whose tangent is \( \frac{a}{b} \), using the signs of \( a \) and \( b \) to determine the quadrant.

**Syntax**

```c
func float atan2( float a, float b )
```

**Returns**

Success >= 0. Returns the angle.
Failure < 0

Example

```c
printf("Q1 2, 2: {}\n",atan2 (2,2))
printf("Q2 2,-2: {}\n",atan2 (2,-2))
printf("Q3 -2,-2: {}\n",atan2 (-2,-2))
printf("Q4 -2, 2: {}\n",atan2 (-2,2))
```

Result

```
Q1 2, 2: 45.00
Q2 2,-2: 135.00
Q3 -2,-2:-135.00
Q4 -2, 2: -45.00
```

RAPL-II

ATAN2

See Also

acos  calculates the arc cosine
asin  calculates the arc sine
tan   calculates the tangent

Category

Math

axes_get

**Description**
Returns the number of machine axes, transform axes, and actual axes installed on the robot. Machine axes are the axes of the robot arm, e.g. 6 for F3. Transform axes are the axes that participate in the kinematics transform, e.g. 7 for F3T (robot arm and track). Actual axes are the total number of axes in the controller, e.g. 8 for T475 with C500-controlled carousel.

**Syntax**

```c
command axes_get( var int machine, var int transform, var int actual )
```

**Parameters**

- `machine`  the machine axes: an int.
- `transform` the transform axes: an int.
- `actual`  the actual axes: an int.

**Returns**

Success = 0. Parameters are packed accordingly.
Failure < 0
Example:  
```
int mach, trans, act
axes_get( mach, trans, act )
```

See Also:  
```
axes_set      sets the number of machine, transform, and actual axes
```

Category:  
```
Robot Configuration
```

---

**axes_set**

**Description**  
The `axes_set` command sets the number of axes in the robot system. An axis is a joint that has its position (motion) controlled by the controller. A track or a carousel can be an axis if connected as part of the robot system. For example, an F3, with 6 axes, can have a track as axis 7.

**Syntax**  
```
command axes_set( int numaxes )
```

**Parameters**  
- `numaxes` the number of axes; an int

**Returns**  
- Success ≥ 0
- Failure < 0

**Example**  
```
axes_set(7) ;; set the system axes to 7.
```

See Also:  
```
axes_get     gets the number of machine, transform, and actual axes
```

Category:  
```
Robot Configuration
```

---

**axis_status**

**Description**  
Obtains data on the status of all axes.

**Syntax**  
```
command axis_status( var int[8] status )
```

**Parameter**  
An array of up to 8 integers into which the status for each axis is stored.

**Returns**  
- Success ≥ 0

The axis status is a bit mask. The bits represent the following:

<table>
<thead>
<tr>
<th>Bit Number</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>home switch state</td>
</tr>
<tr>
<td>1</td>
<td>positive (+) direction limit switch state</td>
</tr>
<tr>
<td>2</td>
<td>negative (-) direction limit switch state</td>
</tr>
<tr>
<td>3</td>
<td>limp command state</td>
</tr>
<tr>
<td>4</td>
<td>axis limp due to collision state</td>
</tr>
<tr>
<td>5</td>
<td>arm for receipt of next zero-cross event</td>
</tr>
<tr>
<td>6</td>
<td>zero-cross event has happened</td>
</tr>
<tr>
<td>7</td>
<td>lock axis from any motion commands</td>
</tr>
<tr>
<td>8</td>
<td>any error condition</td>
</tr>
<tr>
<td>9</td>
<td>servo fault bit</td>
</tr>
<tr>
<td>10</td>
<td>motor fault bit</td>
</tr>
<tr>
<td>11</td>
<td>joint homed</td>
</tr>
<tr>
<td>12</td>
<td>joint calibrated</td>
</tr>
<tr>
<td>13</td>
<td>begin motion</td>
</tr>
<tr>
<td>14</td>
<td>loss of feedback check bit</td>
</tr>
<tr>
<td>15</td>
<td>axis done state</td>
</tr>
</tbody>
</table>

**Example**  
```
int[8] curr_status
...
axis_status(curr_status)
```

**RAPL-II** Similar to STATUS which obtained status data but displayed them at the default device.
### base_get

**Description**

Gets the current base offset, the redefinition of the origin point and the orientation of the world coordinate system.

The default origin is the centre of the base mounting surface of the robot arm.

The offset has translational coordinates, x, y, and z, rotational coordinates, zrot, yrot, and xrot, and extra axes (if any). The data type used is a cloc which also has an integer flag.

**Syntax**

```plaintext
command base_get( var cloc baseloc )
```

**Parameter**

- `baseloc`  the variable to hold offset data: a cloc of variable size

**Returns**

- **Success >= 0**
  - `baseloc`  the offset with flag, x, y, z, zrot, yrot, xrot, e1, e2 data: a cloc
  - `flag`  the : an int
  - `x`  the distance along the X axis, in current units: a float
  - `y`  the distance along the Y axis, in current units: a float
  - `z`  the distance along the Z axis, in current units: a float
  - `zrot`  the rotation around the Z axis, in degrees: a float
  - `yrot`  the rotation around the Y axis, in degrees: a float
  - `xrot`  the rotation around the X axis, in degrees: a float
  - `e1`  the distance or rotation of the first extra axis: a float
  - `e2`  the distance or rotation of the second extra axis: a float

- **Failure < 0**

**Example**

```plaintext
cloc curr_offset
base_get(curr_offset)
print(curr_offset, "\n") ;; no offset applied
```

**Result**

`cloc[9,64(0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)]`

**RAPL-II**

Similar to OFFSET.

**See Also**

- `base_set`  sets a base offset, a re-definition of world coordinates
- `shift_w`  alters coordinate(s)/orientation(s) in world frame of reference
- `tool_get`  gets the current tool transform, the redefinition of tool coordinates

**Category**

Tool Transform and Base Offset

---

### base_set

**Description**

Sets a base offset, a redefinition of the origin point and the orientation of the world coordinate system.

The default origin is the centre of the base mounting surface of the robot arm.

The base_set() command has the capacity for a transformation of a five or six degree-of-freedom arm and one or two extra axes. A cloc data type is used which requires an integer constant flag followed by float constant coordinates. The coordinate system can be redefined by translational coordinates, x, y, and z, and rotational coordinates: zrot, yrot, and xrot. The origin can be further redefined by an extra axis, for example for a track.

A common use of the base_set() command is to transform the coordinate system for an inverted-mounted arm.

**Syntax**

```plaintext
command base_set( var cloc baseloc )
```
Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>baseloc</td>
<td>offset with flag, x, y, z, zrot, yrot, xrot, e1, e2 data: a cloc</td>
</tr>
<tr>
<td>flag</td>
<td>the *: an int</td>
</tr>
<tr>
<td>x</td>
<td>the distance along the X axis, in current units: a float</td>
</tr>
<tr>
<td>y</td>
<td>the distance along the Y axis, in current units: a float</td>
</tr>
<tr>
<td>z</td>
<td>the distance along the Z axis, in current units: a float</td>
</tr>
<tr>
<td>zrot</td>
<td>the rotation around the Z axis, in degrees: a float</td>
</tr>
<tr>
<td>yrot</td>
<td>the rotation around the Y axis, in degrees: a float</td>
</tr>
<tr>
<td>xrot</td>
<td>the rotation around the X axis, in degrees: a float</td>
</tr>
<tr>
<td>e1</td>
<td>the distance or rotation of the first extra axis: a float</td>
</tr>
<tr>
<td>e2</td>
<td>the distance or rotation of the second extra axis: a float</td>
</tr>
</tbody>
</table>

Returns

Success >= 0
Failure < 0

Example

cloc invert
invert = cloc(0, 0, 0, 30, 0, 180, 0, 0, 0)
base_set (invert)
;;;; add 30 units offset to Z
;;;; reverse direction of Z and X
;;;; appropriate for an inverted arm

RAPL-II

Similar to OFFSET.

See Also

base_get gets the current base offset
shift_w alters coordinate(s)/orientation(s) in world frame of reference
tool_set sets a tool transform, a re-definition of the tool coordinate system

Category

Tool Transform and Base Offset

---

boardtemp_get

Description

The boardtemp_get() function retrieves the C500C main board temperature, in degrees Celsius.

Syntax

func float boardtemp_get()

Returns

Success: returns the temperature.

Example

printf(\"The board temperature is {} Celsius\n", boardtemp_get())

See Also

analogs_get()

Category

Analog Input

---

build_cloc

Description

Allows building a cartesian location from a set of constants and variables. It is equivalent to using loc_flags_set() to set the cloc's flags, loc_cdata_set() to set the 8 cartesian axis values and loc_re_check() to recompute the checksum of the resulting location.

Syntax

func cloc build_cloc( int flags, float x, float y, float z, float roll, float pitch, float yaw, float e1, float e2 )

Returns

A cloc constructed from the provided data.

See Also

build_ploc(), loc_flags_set(), loc_cdata_set(), loc_re_check()

Category

Location: Data Manipulation
build_ploc

Description
Allows building a precision location from a set of constants and variables. It is equivalent to using loc_machtype_set() to set the ploc’s machine type, loc_flags_set() to set the ploc’s flags, loc_pdata_set() to set the 8 precision motor pulse values and loc_re_check() to recompute the checksum of the resulting location.

Syntax
func ploc build_ploc( int machtype, int flags, float x, float y, float z, float roll, float pitch, float yaw, float e1, float e2 )

Returns
A ploc constructed from the provided data.

See Also
build_cloc(), loc_machtype_set(), loc_flags_set(), loc_pdata_set(), loc_re_check()

Category
Location: Data Manipulation

calibrate

Description
Finds the proximity sensor, backs up to the last zero cross, and calibrates axes. Data is written to a calibration file named “robot.cal” stored in the conf/ directory. If no arguments are specified, all axes are calibrated.

Syntax
command calibrate( [axis] [,axis] [,axis] ... )

Parameter
axis an axis to calibrate: an int

Returns
Success >= 0
Failure < 0

Example
calibrate()
calibrate(1,3)

RAPL-II
@@CAL

See Also
home homes the axes
calzc calibrates at the next zero cross
zero sets motor position registers to zero

Category
Calibration

call_ifunc

Description
Calls an integer function through a function pointer.

Note:
The function in question cannot be a VARARGS function.
The compiler cannot perform any argument checking, etc. for the call. Use carefully.
What is passed to the function is quite literally what is listed. For example, if <int>x is passed, but the function was expecting a var int parameter, it will fail. Var parameters must be passed as explicit pointers, for example: if the function is expecting “var int x”, then pass variable “int z” as &z.

Syntax
func int call_ifunc(void @funcp, ...)

Returns
Success >= 0
Failure < 0

Example
func int f1(int a, int b)
return a + b
```c
end func
main
    int a, b
    void* vp
    vp = f1  ;; vp points to the function
    a = 2
    b = 3
    printf("f1(a,b) = \n", call_ifunc(vp, a, b))
end main
```

**Result**
The program prints out “f1(a,b) = 5”

**Category**
Pointer Conversion and Function Pointers

### calrdy

**Description**
Moves the arm to the calibrate position. For an F3 or A465, moves the arm straight up. For an A255, moves the arm horizontally outward.

**Syntax**
```c
command calrdy()
```

**Parameter**
```
axis  the axis to calibrate: an int
offset the offset: an int
```

**Returns**
Success >= 0
Failure < 0

**Example**
```
calrdy()
```

**Application Shell**
Same as calrdy.

**RAPL-II**
Same as @CALRDY.

**See Also**
zero sets motor position registers to zero

**Category**
Calibration, Motion

### calzc

**Description**
Calibrates at the next zero pulse of the encoder.

**Syntax**
```c
command calzc( int axis, var int offset )
```

**Parameter**
```
axis  the axis to calibrate: an int
offset the offset: an int
```

**Returns**
Success >= 0
Failure < 0

**Example**
```
int offset = 0
calzc (1, offset) ;; calibrate axis one with no offset
```

**RAPL-II**
```
@@CALZC
```

**See Also**
`homezc`  calibrate calibrates axes
`home` homes the axes
`zero` sets motor position registers to zero

**Category**
Calibration
cfg_load

Description
Loads a text configuration file for the current application. For a concrete example of a configuration file, examine the /conf/robot.cfg robot server configuration file on a typical C500/B/C controller. Text configuration files are useful for holding strings, integers, constant clocs (for tool transforms, etc.) and floating point constants that do not typically change from run to run and do not need to be taught, but nevertheless need to be easily configurable. Note that plocs are not supported.

Syntax
command cfg_load(string[] myname, cfg_record@ crp, int n_records)

Parameters
myname -- used for constructing the config file name.
n_records -- the number of cfg_records pointed to by crp
crp -- points to the cfg_records describing the variables to load

Returns
Success >= 0
Failure   < 0 (-ve error code)

Details
The cfg_load() mechanism works like this:
1. The “myname” argument is used to find the correct configuration file to load. The cfg_load() routine tries “myname.cfg” (ie., in the current directory) first, then “/conf/myname.cfg”. If neither of these files exist, then configure_load() returns the appropriate error code.
2. The config file is read, one line at a time. Anything following a ‘;’ is ignored as a comment (unless the ‘;’ is inside a quoted string.) It is expected that lines will be of the form:
   symbol  value
3. For each “symbol  value” line found, the records pointed to by crp are searched. If a match is found, then the value part of the line is converted and stored in the variable indicated by the cfg_record.

Data structures
The cfg_record structure is a global type definition in the system library, as is defined as:

typedef cfg_record struct
  string[]@ ident ;; field name
  va_types type ;; the type (va_t_int, va_t_float,
  ;; va_t_cloc, va_t_string)
  int  limit ;; length limit, if va_t_string
  void@ where ;; where to put the value
end struct

Example
;; A small example that uses the configuration file routines:

;; These are the variables whose values we wish to configure:
  int reps = 10 ;; note the initialization to a default value
  float height
  cloc ttransform
  string[20] title

;; The cfg_record table:
.define N_CONFIG 4
cfg_record[N_CONFIG] cfg_table = {   
  { "reps", va_t_int, 0, &reps },,
  { "height", va_t_float, 0, &height },,
  { "tool", va_t_cloc, 0, &ttransform },,
  { "title", va_t_string, 20, &title } 
}

;; How we load the config in the main program...
main
cfg_load("test", &(cfg_table[0]), N_CONFIG)
;; At this point, all of the config variables have been
;; read in. If they were absent from the config file,
;; then they still have their default values.

end main

Example .cfg file

; sample .cfg file for the above example:
height 4.2 ; you can have a comment here, too.
reps 20
title "This is a test"
; note the format of the value for a cloc. The first number
; is the flags field, the others are x, y, z ...
tool { 0, 0.0, 0.0, 1.2, 0.0, 0.0, 0.0, 0.0, 0.0 }
; end of the .cfg file

See Also

cfg_load_fd(), cfg_save(), cfg_save_fd(), cfg_token_get()

Category

Configuration File Handling

cfg_load_fd

Description

Loads a configuration information from a file that is already open. Please see
cfg_load() for details.

Syntax

command cfg_load_fd(int fd, string[] myname,
cfg_record@ crp, int n_records)

Parameters

fd -- the open (for reading) config file descriptor
myname -- used for constructing the config file name.
n_records -- the number of cfg_records pointed to by crp
crp -- points to the of cfg_records describing the
variables to load

Returns

Success >= 0
Failure < 0 (-ve error code)

Example

;; See the cfg_load() example above for details.

;; A small example that uses the configuration file routines:

;; These are the variables whose values we wish to configure:
int reps = 10 ;; note the initialization to a default value
float height
cloc ttransform
string[20] title

;; The cfg_record table:
.define N_CONFIG 4
cfg_record[N_CONFIG] cfg_table = { 
{ "reps", va_t_int, 0, &reps },
{ "height", va_t_float, 0, &height },
{ "tool", va_t_cloc, 0, &ttransform },
{ "title", va_t_string, 20, &title } 
}

;; How we load the config in the main program...
main
int fd
...
open(fd, "myconfig.cfg", O_RDONLY, 0) ;; open the file
cfg_load_fd(fd, "whatever", &(cfg_table[0]), N_CONFIG)
...
end main
### cfg_save

**Description**
Re-writes a configuration file for the current application. Please see cfg_load() for many related details. This allows a program to change its own configuration and then re-write its configuration file. Note that the original configuration file is completely overwritten; all comments in it are lost. Also note that cfg_save() will not create a missing config file; the file must already exist (but may be empty).

**Syntax**
```c
command cfg_save(string[] myname, cfg_record@ crp, int n_records)
```

**Parameters**
- `myname` -- used for constructing the config file name.
- `n_records` -- the number of cfg_records pointed to by crp
- `crp` -- points to the cfg_records describing the variables to save

**Returns**
- Success $\geq 0$
- Failure $< 0$ (-ve error code)

**Example**
```c
;; To the example from cfg_load(), add the following code
;; to re-write the configuration file:
...
  cfg_save("test", &cfg_table[0], N_CONFIG)
...
```

**See Also**
cfg_load(), cfg_load_fd(), cfg_save(), cfg_token_get()

**Category**
Configuration File Handling

### cfg_save_fd

**Description**
Re-writes a configuration file for the current application. Please see cfg_load() for many related details. This allows a program to change its own configuration and then re-write its configuration file. Note that the original configuration file is completely overwritten; all comments in it are lost.

**Syntax**
```c
command cfg_save_fd(int fd, string[] myname, 
cfg_record@ crp, int n_records)
```

**Parameters**
- `fd` -- the open (for writing) config file descriptor
- `myname` -- used for constructing the config file name.
- `n_records` -- the number of cfg_records pointed to by crp
- `crp` -- points to the cfg_records describing the variables to save

**Returns**
- Success $\geq 0$
- Failure $< 0$ (-ve error code)

**Example**
```c
;; To the example from cfg_load(), add the following code
;; to re-write the configuration file using cfg_save_fd():
...
  int fd
  open(fd, "myconfig.cfg", O_WRONLY | O_TRUNC, 0); ;; open the file
  cfg_save_fd(fd, "test", &cfg_table[0], N_CONFIG)
...
```

**See Also**
cfg_load(), cfg_load_fd(), cfg_save(), cfg_token_get()

**Category**
Configuration File Handling
**chdir**

**Description**
Changes the current working directory to *path*. The search for all relative pathnames (all pathnames that do not begin with a slash) starts at the current working directory.

**Syntax**
```
command chdir( var string[] path )
```

**Returns**
- 0 (-EOK)  Success
- EINVAL    If *path* was invalid
- ENOTDIR   If *path* is not a directory
- ENOENT    If *path* was not found
- EIO        If an I/O error occurred

**Example**
```c
int fd
chdir("/app/test/test2") ;; set working directory
open(fd, "myfile", O_RDWR|O_CREAT, M_READ|M_WRITE )
fprintf(fd, "file header: 04/23/98")
close(fd)
```

**System Shell**
`cd`

**RAPII**
No equivalent.

**Category**
File and Device System Management

**chmod**

**Description**
Changes access mode information of an object (file or device) in the file system.

**Syntax**
```
command chmod( var string[] path, int mode )
```

**Parameter**
- `path` string defining the path to the file
- `mode` the modes of access, of type mode_flags, any combination of:
  - M_READ   read allowed
  - M_WRITE  write allowed
  - M_EXEC   executable

**Returns**
- 0 (-EOK)  Success
- EINVAL    If the arguments were invalid
- ENOTDIR   If any of the directory components of *path* was not a directory
- ENOENT    If *path* was not found
- EIO       If an I/O error occurred
- EAGAIN    If we are temporarily out of the system resources needed to perform this operation.

**Example**
```c
chdir("/app/test/test2") ;; set working directory
open(fd, "myfile", O_RDWR|O_CREAT, M_READ|M_WRITE )
fprintf(fd, "file header: 04/23/98") ;; write data to file
chmod("/app/test/test2/myfile",M_WRITE) ;; prevent file from being read
close(fd)
```

**System Shell**
`chmod`
chr_is_lower

Description
Determines whether a character is lower case. Returns 1 if true, 0 if false.

Syntax
func Boolean chr_is_lower( int char )

Parameter
char the character: handled as an int

Returns
True = 1
False = 0

Example
int len, i, inval_char=0
string[25] user_input
...
printf ("enter selection (lower case only) : ")
readline (user_input,25)
...
for i = 0 to (str_len (user_input)-1)
  if chr_is_lower(str_chr_get(user_input,i))== 0
    inval_char = 1 ;; set invalid char. flag
  end if
end for

See Also
chr_is_upper checks if a character is upper case

Category
String Manipulation

chr_is_upper

Description
Determines whether a character is upper case. Returns 1 if true, 0 if false.

Syntax
func Boolean chr_is_upper( int char )

Parameter
char the character: handled as an int

Returns
True = 1
False = 0

Example
int len, i, inval_char=0
string[25] user_input
printf ("ENTER SELECTION (UPPER CASE ONLY): ")
readline (user_input,25)
...
for i = 0 to (str_len (user_input)-1)
  if chr_is_upper(str_chr_get(user_input,i))== 0
    inval_char = 1 ;; set invalid char. flag
  end if
end for

See Also
chr_is_lower checks if a character is lower case

Category
String Manipulation

chr_to_lower

Description
Converts a letter from upper case to lower case. If the letter is already lower case, it is not changed.

Syntax
func int chr_to_lower( int char )
### chr_to_upper

**Description**

Converts a letter from lower case to upper case. If the letter is already upper case, it is not changed.

**Syntax**

```plaintext
func int chr_to_upper( int char )
```

**Parameter**

- `char` the character: handled as an int

**Returns**

- Success >= 0
- Failure < 0

**Example**

```plaintext
int char, len, i, flag=0
string[25] user_input
printf("ENTER SELECTION (UPPER CASE ONLY): ")
readline (user_input,25)
...
for i = 0 to (str_len (user_input)-1)
  if chr_is_lower(str_chr_get(user_input,i))== 0
    char = str_chr_get(user_input,i) ;; read lower case char
    char = chr_to_upper(char) ;; convert case of char
    ;; to upper
    str_chr_set (user_input,i,char) ;; write char back to
    ;; string
    flag = 1 ;; set char conversion
  end if
end for
```

**See Also**

- `chr_to_lower` converts a character to lower case
- `str_to_upper` converts a string to upper case

**Category**

String Manipulation

---

### clear_error

**Description**

Clears persistent error bits on the digital signal processor (DSP). This includes runaways, collisions, overspeeds, and encoder faults. After an error of this type,
the clear_error() command must be invoked before the arm power can be re-engaged.

NOTE: This command only works with the F-series arms.

Syntax
command clear_error()

Returns
Success >= 0
Failure < 0 Returns -ve error descriptor if command fails.

Example
clear_error()

Category
Pendant

close

Description
Closes a file or device. The connection between a file descriptor and the open file associated with it is broken. This frees the file descriptor for use with other files.

Syntax
command close( int fd )

Returns
0 (-EOK) Success
-EINVAL The argument was invalid (ie., -ve)
-EBADF fd doesn’t correspond to an open file.
-EIO An I/O error occurred

Example
int fd
... open ( fd, "filename", O_RDONLY, 0 ) ;; open existing file for reading
... close (fd)

RAPL-II
No equivalent

See Also
open opens a file

closenp

close named pipe

Description
Closes a named pipe.

Syntax
closenp( int fd )

Parameter
fd the file descriptor: an int

Returns
Success >= 0
Failure < 0

Example
closenp(pd)
closenp(NT_app_pipe)

RAPL-II
No equivalent.

See Also
opennp opens a named pipe
disconnectnp disconnects a client from a named pipe
connectnp connects to a named pipe
statusnp checks the status of a named pipe
conf_get

**Description**
Gets a list of robot configuration parameters.

**Syntax**
```c
command conf_get( var int[5] config )
```

**Parameter**
- `config` the configuration: an array of ints to hold:
  - [0] product code
  - [1] robot code
  - [2] number of axes
  - [3] config
  - [4] arm power status

**Returns**
- Success $\geq 0$
- Failure $< 0$

**Example**
```c
int[5] config

conf_get(config); ; configuration is copied into the array

printf("Robot configuration data is: ")
for i = 0 to 4
  printf("{}", config[i])
end for
```

**Result**
Robot configuration data is: 7, 9, 6, 79, 0

**confirm_menu**

**Description**
Using the confirm_menu command forces the user to confirm an action before it is carried out. The command allows for up to 3 strings to be sent to the pendant screen. Each string will be placed on a different row of the screen starting with the top row. Each string can have a maximum of 20 characters. Any character beyond this is truncated.

**Library**
```
stp
```

**Syntax**
```c
export func int confirm_menu( var string[] str_1, var string[] str_2, var string[] str_3)
```

**Parameter**
- `str_1` text string displayed on the top row of the pendant screen
- `str_2` text string displayed on the second row of the pendant screen
- `str_3` text string displayed on the third row of the pendant screen

**Returns**
- Success $\geq 0$
- Failure $< 0$

**Example**
```c
int ctrl
string[10] name = "my_app_23"
stp:startup()
stp:app_open(name, 0)
...
  ctrl = stp:confirm_menu("Do You wish to","Continue? ","***")
...
stp:app_close()
...
connectnp

**Description**
Checks or waits for a client to connect with the named pipe.

If the wait parameter is set to TM_NOWAIT, the command returns immediately. If the wait parameter is set to TM_FOREVER (or anything else), it will block (not interruptible) until a client connects.

**Syntax**
```c
command connectnp( int fd, int wait )
```

**Parameters**
- `fd` the file descriptor: an int
- `wait`

**Returns**
- Success >= 0, client has connected.
- Failure < 0

**Example**
```c
connectnp(pd, TM_NOWAIT)
connectnp(NT_app_pipe, TM_FOREVER)
```

**See Also**
- disconnectnp connects a client from a named pipe
- closenp closes a named pipe
- opennp opens a named pipe
- statusnp checks the status of a named pipe

**Category**
Win 32

---

**COS**

**Description**
Calculates the cosine of an angle. Takes an argument in degrees.

**Syntax**
```c
func float cos( float x )
```

**Returns**
- Success >= 0. The cosine of the argument in degrees.
- Failure < 0

**Example**
```c
float x = 45.00
float y
y = cos( x )
```

**Result**
0.7071

**RAPL-II**
COS

**See Also**
- sin calculates the sine
- tan calculates the tangent
- acos calculates the arc cosine

**Category**
Math

---

cpath

**Description**
Calculates and executes a path immediately.

The path is stored as path 0 and can be repeated with cpath_go(0).

**Syntax**
```c
command cpath( gloc@ locname, int start, int finish, \
                var trigger_type triggers )
```
### Parameter

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>locname</td>
<td>the locations: a pointer to an array of locations</td>
</tr>
<tr>
<td>start</td>
<td>the index of the location array to start: an int</td>
</tr>
<tr>
<td>finish</td>
<td>the index of the location array to finish: an int</td>
</tr>
<tr>
<td>triggers</td>
<td>the information to set gpio outputs: an int[16,2] for any of the rows in the array. Elements in the 0 column are the indexes of the location array, elements in the 1 column are the setting and identifiers of gpio output</td>
</tr>
</tbody>
</table>

#### Returns

- **Success = 0**
- **Failure < 0**

#### Example

```c
   teachable  cloc[10]  b
trigger_type  trig2
...
trig2[0,0]=6   ;; first trigger at location 6
trig2[0,1]=-1 ;; first trigger turns output #1 off
trig2[1,0]=7   ;; second trigger at location 7
trig2[1,1]=1   ;; second trigger turns output #1 on
trig2[2,0]=9   ;; third trigger is location 9
trig2[2,1]=15  ;; third trigger turns output #15 on
...
cpath(&b[0], 5, 9, trig1)
   ;; executes a path, starting at b[5] and going to b[9] using trig2 as a trigger table
```

The location name must be given in this form. It is not sufficient to simply enter `b` in the second argument.

### ctl_get

#### Description

Gets point of control.

#### Syntax

```c
   command  ctl_get()
```

#### Returns

- **Success >= 0**
- **Failure < 0** Will fail only due to communications.
  - 16, EBUSY, indicates another process has control.

#### Example

```c
   ctl_get()
```

### ctl_give

#### Description

Gives control explicitly to the process specified by the `pid` parameter.

#### Syntax

```c
   command  ctl_give(int pid)
```

#### Parameter

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pid</td>
<td>specifies the process to be given control</td>
</tr>
</tbody>
</table>

#### Returns

- **Success >= 0**
- **Failure < 0** Returns negative error code if command fails. Two possibilities are:
ctl_rel

Description
Releases point of control.

Syntax
command ctl_rel()

Returns
Success >= 0
Failure < 0

Example
ctl_rel()

RAPL-II
There is no corresponding construct.

See Also
ctl_get gets point of control

Category
System Process Control: Point of Control and Observation

cpath

Description
Creates and stores a continuous path through an array of locations with triggers for gpio (general purpose input/output).

To execute the path, use the cpath_go() command.

Syntax
command cpath( int pathnum, gloc@ locname, int start, int finish, \
var trigger_type triggers [, int speed] )

Parameters
- pathnum the path's index number: an int from 1 to 8
- locname the locations: a pointer to the first location of an array the locations must all be elements of the same one dimensional array
  Note the form in the example.
- start index of the location array to start: an int
- finish index of the location array to finish: an int
- triggers the triggers: an array [16,2] of ints where the 16 triggers(rows in the array) are indexed 0 to 15, the trigger info (columns in the array) are indexed 0 and 1,and for any row, the elements contain
  in column 0, the location, specified by its index in the location array, locname
  in column 1, the setting of the output, specified by a positive or negative sign, and the output channel, specified by its number
  See the example below.

Parameter (Optional)
- speed the percentage of full speed through the path: an int
  if speed is not specified, the current robot speed is used

Returns
Success = 0
Failure < 0

Example
teachable cloc[20] a
trigger_type trigl
...
trigl[0,0]=0 ;; first trigger at location 0
trig1[0,1]=4 ;; first trigger is turning output #4 on
trig1[1,0]=3 ;; second trigger at location 3
trig1[1,1]=1 ;; second trigger is turning output #1 on
trig1[2,0]=5 ;; third trigger is location 5
trig1[2,1]=-4 ;; third trigger is turning output #4 off

ctpath(1, &a[0], 0, 19, trig1, 65)
++; pre-calculates path 1, starting at a[0] and going to a[19]
++; using trig1 as a trigger table and moving at 65% speed.

The location name must be given in this form. It is not sufficient to simply enter
a in the second argument.

cpath(10, &mypoints[0], 20, 30, mytrig)

RAPL-II
Similar to CTPATH and TRIGGER.

See Also
ctpath_go runs the path
cpath

Category
Motion

copath_go

Description
Runs a path previously stored by cpath(). Moves to the beginning of the specified
path and executes the path at the speed previously specified.

Moves the arm in joint-interpolated mode to the starting knot of the path at the
current speed setting. Moves through the path at the previously specified path
speed.

Since a cpath() is stored as path 0, the command ctpath_go(0) executes the
previous cpath().

Syntax
command ctpath_go( int pathnumber )

Parameter
pathnumber the path number defined in ctpath: an int

Returns
Success = 0
Failure < 0

Example
cpath(1, &a[0], 0, 19, trig1, 65)
...
ctpath_go(1)

Example
cpath(3,12,dispense_adhesive)
...
ctpath_go(3)

RAPL-II
Same as GOPATH.

See Also
cpath creates and stores a continuous path with triggers
cpath calculates and executes a path immediately

Category
Motion

deg

Description
Converts radians to degrees.

Syntax
func float deg( float x )

Returns
Success >= 0
Failure < 0
delay

Description

Sleeps for at least the number of milliseconds specified in milliseconds. Repeated signals can cause this delay to be longer than the milliseconds requested. Differs from msleep(). delay() allows sleeping without getting terminated by an EINTR error.

Syntax

command delay ( int milliseconds )

Returns

Always returns 0 (Success)

Example

loop
  print ("Waiting for GPIO input 1. \n")
  if (input(1,state) == 1 )
    break
  end if
  delay (250)
end loop

depart

Description

Moves the tool centre-point from the current position, along the “approach/depart” tool axis, to a depart position. The depart position is defined by a distance from the current position along the “approach/depart” tool axis. Positive distance is away from the location. Negative is towards the location. The starting position can be any position. It does not have to be a location.

This command is used to move the tool, usually slowly, away from a position a short distance before moving the arm, usually quickly, to a position a larger distance away.

Moves in joint interpolated mode. The result is not a straight line.

Syntax

command depart( float distance )

Parameter

distance the distance from the location to the depart position: a float

Returns

Success >= 0
Failure < 0

Example

depart(2.0)
depart(6.0)
speed_set(100)
appro(pick_1, 2.0)
RAPL-II

Similar to DEPART.

See Also

departs  like depart(), but in straight line motion
appro    moves to an approach position; opposite of depart
appro    moves to an approach position; opposite of departs
tool_set re-defines the tool coordinate system

Category  Motion

 departs

Description
Moves the tool centre-point from the current position, along the
“approach/depart” tool axis, to a depart position. The depart position is defined
by a distance from the current location along the “approach/depart” tool axis.
Positive distance is away from the location. Negative is towards the location.

The starting position can be any position. It does not have to be a location.

Used to move the tool, usually slowly, away from a position a short distance
before moving the arm, usually quickly, to a position a larger distance away.

Moves in cartesian interpolated mode. The result is straight line motion.

Syntax
command  departs( float  distance )

Parameter
distance  the distance from the location to the depart position: a float

Returns
Success  >= 0
Failure  < 0

Example
departs(2.0)
departs(6.0)
speed_set(100)
appro(pick_1,2.0)
speed_set(20)
moves(pick_1)
finish()
grip_close()
grip_finish()
departs(2.0)
speed_set(100)
appro(place_1)
**disconnectnp**

**Description**
Breaks a pipe connection with a client. The server forcibly disconnects the client. Must be done to be able to connect with a new client.

**Syntax**
```
command disconnectnp( int fd )
```

**Parameter**

**fd**
the file descriptor: an int

**Returns**

Success &ge; 0
Failure &lt; 0

**Example**

disconnectnp(pd)
disconnectnp(NT_app_pipe)

**RAPL-II**
No equivalent.

**See Also**
connectnp connects to a named pipe
closenp closes a named pipe
opennp opens a named pipe
statusnp checks the status of a named pipe

**Category**
Win 32

---

**dup**

**Description**
Duplicates an existing file descriptor. The new file descriptor is the lowest available file descriptor. The new file descriptor, stored in new_fd, has the following in common with the original file descriptor, old_fd:

- Same open file or device
- Same file pointer
  (Changing the file pointer of one changes file pointer of the other.)
- Same access mode (read, write, read/write)

**Syntax**
```
command dup( var int new_fd, int old_fd )
```

**Parameter**

**new_fd**
the new file descriptor which is a duplication of old_fd: an int

**old_fd**
the file descriptor being duplicated: an int

**Returns**

&ge; 0 Success.
-EAGAIN There are no free file descriptors.
-EINVAL The old_fd argument was invalid (i.e. negative).
-EBADF old_fd does not correspond to an open file.

**Example**
See example for dup2()

**See Also**
dup2 creates a new file handle

**Category**
File and Device System Management

---

**dup2**

**Description**
Duplicates an existing file descriptor. The original file descriptor, old_fd, is duplicated at a new position in the file descriptor table specified by new_fd. The
new file descriptor, \textit{new\_fd}, has the following in common with the original file descriptor, \textit{old\_fd}:

- Same open file or device
- Same file pointer  
  (Changing the file pointer of one changes file pointer of the other.)
- Same access mode (read, write, read/write)

dup2() creates the new handle with the value of \textit{new\_fd}. If there was a file associated with \textit{new\_fd} already open then dup2() first closes this file.

### Syntax

```command
dup2( int \textit{new\_fd}, int \textit{old\_fd} )
```

### Parameter

- \textit{new\_fd}  
  the position of the new duplicated file descriptor: an int
- \textit{old\_fd}  
  the file descriptor being duplicated: an int

### Returns

\begin{itemize}
  \item \texttt{>= 0}  
    Success.
  \item \texttt{-EINVAL}  
    The arguments were invalid (i.e. negative file descriptors).
  \item \texttt{-EBADF}  
    \textit{old\_fd} does not correspond to an open file.
  \item \texttt{-EINVAL}  
    The argument was invalid (i.e. negative file descriptors).
  \item \texttt{-EBADF}  
    \textit{fd} does not correspond to an open file.
  \item \texttt{-EIO}  
    An i/o error occurred.
\end{itemize}

### Example

```c
int nul, oldstdout, STDOUT = 1
string[] msg = "This is a test"

;; create a file
open ( nul, "DUMMY.FIL", O_CREAT | O_RDWR, S_IREAD | S_IWRITE )

;; create a duplicate handle for standard output
dup ( oldstdout, STDOUT )

;; redirect standard output to DUMMY.FIL
;; by duplicating the file handle onto
;; the file handle for standard output
dup2 ( STDOUT, nul )

;; close the handle for DUMMY.FIL
close ( nul )

;; will be redirected into DUMMY.FIL
fprint ( STDOUT, msg )

;; restore original standard output handle
dup2 ( STDOUT, oldstdout )

;; close duplicate handle for STDOUT
close ( oldstdout )
```

### See Also

dup  
creates a new file handle

### Category

File and Device System Management

---

**environ**

**Description**

Allows a program to retrieve each individual string from its environment. [This command is available on the C500C only.]
command environ(var string[] dst, int n)

Parameters
There are two required parameters:

- **dst** a string variable to write the selected environment string into.
- **n** the index of the selected environment string. Starts at zero.

Returns
1 → the selected string was successfully copied into `dst`
0 → there is no environment string with the specified index; `dst` is set to the empty string
< 0 → a negative error code.

Explanation
The environment strings are a set of strings of the form “label=value” that are accessible to each running program. When one program launches another one via `execl()` or `execv()`, it passes on its set of environment strings. Thus if one program adds a new string to its environment or deletes a string from its environment, all of its children inherit these changes.

Environment variables are convenient for storing information about the entire system. When CROS starts up, it sets up the initial environment strings from the diagnostic configuration strings. These strings are always set up by CROS as part of the environment:

- **HOSTTYPE** What kind of processor the controller has. Typically “i386”.
- **OSTYPE** What operating system is running. Typically “CROS”.
- **SerialNumber** The controller serial number.

Example
```
;; This RAPL-3 program displays all of the environment strings:
;;
main
  int n
  string[256] s
  n = 0
  while (environ(s, n) > 0)
    printf("{}\n", s)
    n++
  end while
end main
```

See Also
`getenv()`, `setenv()`, `unsetenv()`

Category
Environment Variables

---

err_compare

Description
Compares two error descriptors for matching subsystem and error code fields. Can be used, for example, to find out if an error is a runaway error (regardless of the axis involved.)

Library
`syslib`

Syntax
`func int err_compare(int d1, int d2)`

Parameters
d1, d2 error descriptors to compare

Returns
1 (True) if the subsystem and error codes match
0 (False) if they do not.
Example

```
t = move(there)
if (err_compare(REAXIS_RUNAWAY, -t))
  ... runaway error ...
end if
```

See Also
error descriptors

Category
Error Message Handling

---

### err_compose

**Description**
The function is passed four integer values representing the subsystem, b2, b1 and code values of a given error descriptor. The function reconstructs and returns the original error descriptor. Refer to the Error Descriptor section for details on the error descriptor.

**Syntax**

```
func int err_compose(int subsys, int b2, int b1, int code)
```

**Parameter**

- **subsys**
  The integer value of the subsystem originating the error
- **b2**
  The integer value of the b2 field
- **b1**
  The integer value of the b1 field
- **code**
  The integer value of the specific error code

**Returns**
Returns the 32 bit error descriptor reconstructed from the 4 separate 8 bit fields. Refer to the Error Handling section for a details on the file descriptor.

**Failure < 0**

**Example**

A program to confirm that the translation from the error descriptor to the error data is correct.

```
int t, comp, err_des
int subsys, code, b2, b1

t = open(fd, "myfile", O_RDONLY, 0)
if (t < 0) ;; error
  err_des = -t...
  subsys = err_get_subsys(err_des)
  code = err_get_code(err_des)
  b2 = err_get_b2(err_des)
  b1 = err_get_b1(err_des)
  if (comp = err_compose(subsys, b2, b1, code) != err_des)
    ...
    ;; Something went wrong in the error translations
    ...
    exit(1)
  else
    printf("The error {} occurred in the {}subsystem 
", str_subsys(err_des))
    printf("The b2 error field is '{}'
", b2)
    printf("The b1 error field is '{}'
", b1)
    exit(1)
end if
```

end if
The error no device occurred in kernel subsystem
The b2 error field is X
The b1 error field is Y ::X and Y are integers.

See Also
err_get_subsys
err_get_b2
err_get_b1
err_get_code

Category
Error Message Handling

**err_get_b1**

**Description**
The function is passed a +ve error descriptor. It returns the integer value of the b1 field in the error descriptor. The error descriptor is a 32 bit integer, the negative value of which is returned when a function call fails. Refer to the Error Descriptor section for details on the error descriptor.

**Syntax**
func int err_get_b1(int descriptor)

**Parameter**
descriptor the parameter int is the error descriptor

**Returns**
Success >= Returns the integer which corresponds to the 8 bits which correspond to the b1 field in the error descriptor. **Note:** if the b2 field is not defined for the specific error, the function returns 0. Refer to the Error Handling section.

Failure < 0

**Example**
int t, err_des
t = open(fd, “myfile”, O_RDONLY, 0)
if (t < 0) ;; error
   err_des = -t ;; change sign of error for use with error functions

   printf(“The b1 error field is ‘\”n’, err_get_b1(err_des))
exit(1)
end if

**Result**
The b1 error field is X X is the integer value of the b2 field of the error descriptor

**See Also**
error_code
addr_decode

**Category**
Error Message Handling

**err_get_b2**

**Description**
The function is passed a +ve error descriptor. It returns the integer value of the b2 field in the error descriptor. The error descriptor is a 32 bit integer, the negative value of which is returned when a function call fails. Refer to the Error Descriptor section for details on the error descriptor.

**Syntax**
func int err_get_b2(int descriptor)

**Parameter**
descriptor the parameter int is the error descriptor

**Returns**
Success >= Returns the integer which corresponds to the 8 bits which correspond to the b2 field in the error descriptor. Note if the b2 field is not defined for the specific error, the function returns 0. Refer to the Error Handling section.

Failure < 0
Example

```c
int t, err_des  
t = open(fd, "myfile", O_RDONLY, 0)  
if (t < 0)    ;; error  
    err_des = -t ;; change sign of error for use with error functions  
    printf("The b2 error field is '{}', err_get_b2(err_des))
    exit(1)
end if
```

Result
The b2 error field is X  X is the integer value of the b2 field of the error descriptor

See Also
error_code
addr_decode

Category
Error Message Handling

---

### err_get_code

**Description**

The function is passed a +ve error descriptor. It returns the integer value of the code field in the error descriptor. The error descriptor is a 32 bit integer, the negative value of which is returned when a function call fails. Refer to the Error Descriptor section for details on the error descriptor.

Note: Use the str_error function to convert the error descriptor to a string.

**Syntax**

```c
func int err_get_code(int descriptor)
```

**Parameter**

descriptor    the parameter int is the error descriptor

**Returns**

Success >= Returns the integer which corresponds to the 8 bits which correspond to the code field in the error descriptor. Refer to the Error descriptor section for details.

Failure < 0

**Example**

```c
int t, err_des  
t = open(fd, "myfile", O_RDONLY, 0)  
if (t < 0)    ;; error  
    err_des = -t ;; change sign of error for use with error functions  
    printf("The error code number is '{}', err_get_b2(err_des))
    exit(1)
end if
```

Result
The error code number is X  X is the integer value of the error code

See Also
str_error

Category
Error Message Handling

---

### err_get_subsys

**Description**

The function is passed a +ve error descriptor. It returns the integer value of the subsystem where the error originated. The error descriptor is a 32 bit integer, the negative value of which is returned when a function call fails. The subsystem information is carried in the error descriptor. Refer to the Error Descriptor section for details on the error descriptor.

**Syntax**

```c
func int err_get_subsys(int descriptor)
```

**Parameter**

descriptor    the parameter int is the error descriptor
Returns  Success >=  Returns the integer corresponding to the subsystem. For example:
  Subsystem 0  kernel
  Subsystem 1  robot library
  Subsystem 2  robot server
  (List is not complete)
  Refer to the Error descriptor section for details on the subsystem error
  files.
Failure < 0
Example  int t, err_des
  t = open(fd, “myfile”, O_RDONLY, 0)
  if (t < 0)    ;; error
      err_des = -t    ;; change sign of error for use with error functions
      printf(“The error occurred in subsystem ‘[‘\n”, err_get_subsys(err_des))
      exit(1)
end if
Result  The error occurred in subsystem X   X is the decimal number of the
  subsystem
See Also  error_code
          addr_decode
Category  Error Message Handling

**error_addr**

Description  The function returns the address where the current exception occurred.
Syntax  func int error_addr()
Parameter  no parameters required
Returns  Success >= 0
  Failure < 0
Example  see the example for addr_to_file()
See Also  error_code
          addr_decode
Category  Error Message Handling

**error_code**

Description  Get the current exception’s error code.
Syntax  func int error_code()
Parameter  no parameter required
Returns  Success >= 0
  Failure < 0
Example  try
  abort(-1) ;; this should cause an exception
  except
      printf(“Error ‘[‘ happened\n”, str_error(-error_code()))
  end try
Result  The program prints out “Error ‘General Error’ happened”
See Also
- error_addr
- addr_decode

Category
Error Message Handling

---

**error_line**

**Description**
Calls the addr_to_line function to determine the line number of the current error. This is equivalent to calling addr_to_line(error_addr()).

**Syntax**
```
func int error_line()
```

**Parameters**
No parameters required

**Returns**
- Success: The line number
- Failure: 0

**Example**
see addr_to_line() for a related example.

**See Also**
- error_addr
- error_file
- addr_to_line
- addr_decode*

Category
Error Message Handling

---

**error_file**

**Description**
Calls the addr_to_file function to convert the current error to a file name where the current error resides. This is equivalent to calling addr_to_file(error_addr()).

**Syntax**
```
func string[] error_file()
```

**Parameters**
No parameters required

**Returns**
- Success: A pointer to the file name string
- Failure: A pointer to an empty string on failure

**Example**
see addr_to_file() for a related example.

**See Also**
- error_addr
- error_line
- addr_to_line
- addr_decode*

Category
Error Message Handling

---

**execl**

**Description**
Loads and executes another program. The program takes all the command-line arguments as string[] parameters. The program that launches the new program is terminated, and the new program takes on the pid number of its terminated parent. The execl() command is often executed from within a child process. This command is used when all of the command-line arguments are known. If they are not known, use execv().

Certain errors can cause the program running execl() to terminate (with exit code 255). For example, missing libraries can cause this.

**Syntax**
```
command  execl( var string[] file_name, var string[] arg, ... )
```
**execv**

**Description**
Loads and executes another program. The program that launches the new program is terminated, and the new program takes on the pid number of its terminated parent. The “execv” command is often executed from within a child process. The program takes one other argument which is a pointer to variable length array of strings, argv. These are the command-line arguments for the program. This command is used when the command-line arguments are not known. If the command-line arguments are known, use execl().

Certain errors can cause the program running execv() to terminate (with exit code 255). For example, missing libraries can cause this.

**Syntax**
```
command  execv( var string[] file_name, var string[]@@ argv )
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>file_name</td>
<td>the file name, including the path, to be executed</td>
</tr>
<tr>
<td>argv</td>
<td>pointer to an array of string pointers</td>
</tr>
</tbody>
</table>

**Example**
```c
int  split_id
string[] my_prog = “My_Program”
...
split_id
if split_id == 0
   execv (my_prog, “arg0”, “arg1”, “arg2”)
else
   waitpid (split_id,&status,0) ;; wait until child has terminated
end if
```

**RAPL-II**
EXECUTE

**See Also**
execv executes another program with unknown arguments

**Category**
System Process Control: Single and Multiple Processes
Returns

Success  no return- the process ceases to exist and is replaced by the specified new running process

Failure:

- EBADF     fd does not represent an open file
- EINTR     was interrupted by a signal
- EINVAL    path is illegal, or there is not at least one command-line argument
- EACCESS   does not have its execute permission bit set
- ENOEXEC   the file is not a recognized executable
- ENOMEM    not enough free memory
- EIO        An I/O error occurred.
- ENOENT     The file specified by file_name does not exist
- ESPIPE    can't r/w on a socket
- EIO        an I/O error occurred
- ENOTDIR    A component of the path to the file was not a directory.

Example

```c
string[20] user_input
string[]@[10] argv_sp
int i, split_id, status, num_args = 0

loop
    printf("* enter argument: ")
    readline(user_input,20)
    if user_input != "x" ;; "x" terminates input
        mem_alloc(argv_sp[num_args], sizeof(user_input))
        argv_sp[num_args]@ = user_input ;; allocate memory and initialize ptr to memory
        num_args ++ ;; increment string counter
    else
        break
    end if
end loop

split_id = split()
if split_id == 0 ;; * child process
   execv(argv_sp[0]@,&(argv_sp[0])) ;; execute new program
elseif split_id !=0 ;; * parent process
   waitpid(split_id,&status,0) ;; wait for child to complete
end if

for i = 0 to (num_args-1)
    mem_free(argv_sp[i]) ;; free allocate memory
end for
```

RAPL-II

EXECUTE

See Also

execl   executes another program with known arguments
argc    returns the number of command-line arguments
argv    returns a pointer to a command-line argument

Category

System Process Control: Single and Multiple Processes
exit

Description
Causes normal program termination. Open files are flushed and closed. The value
n is returned to the parent process indicating success or failure. Conventionally,
0 is used to indicate successful termination and non-zero values to indicate
abnormal termination. Note that only the lowest 8 bits of the ret_val value are
returned to the parent; the value must be in the range 0 to 255.

Syntax
command exit( int ret_val )

Parameter
ret_val the value returned to the parent process: an int

Returns
Never returns.

Example
int pid
...
pid = split()
if pid == 0
  ;; child process does something
  exit (0)
else
  ;; parent process does something
end if

Example
int result
...
result = func_call() ;; evaluate the function return value
  if result != EOK ;; an error occurred during the function execution
    exit (-1)
  else
    exit (0) ;; no error
end if

RAPL-II
ABORT

See Also
abort terminates a program

Category
System Process Control: Single and Multiple Processes

fabs

Description
Calculates the absolute value of a float.

Syntax
func float fabs( float x )

Argument
x the number: a float

Returns
Success >= 0 The absolute value of the argument x.
Failure < 0

Example
float x = -99.9
float y
y = fabs( x )

Result
y is set to 99.9

RAPL-II
ABS

See Also
iabs calculates the absolute value of an int

Category
Math
### finish

| Description | Forces the program to wait at the finish() command until arm motion has finished. Normally a command is executed as soon as its parameters are determined, which can be before the previous command has finished. finish() is often used to finish the motion of the arm to a location before closing the gripper at the location, instead of having the gripper start to close while the arm is still in motion to the location. finish() is also used to synchronize commands, such as input/output, with robot motion. If online mode is off, finish() is not needed between two arm motion commands. In online off mode, arm motion commands are executed as if there is a finish() after each one. There is one exception, the motor() command for different axes. The later motor() command does not wait for the earlier motor() command to finish. |
| Syntax | command  finish() |
| Parameter | No parameters required |
| Returns | Success >= 0  
Failure < 0  |
| Example | appro(pick_1,2.0)  
movemove(pick_1)  
finish()  
grip_close()  
:: Without finish()  
:: the grip_close() command would begin executing  
:: before the movemove(pick_1) command finished. |
| RAPL-II | Similar to FINISH. |
| See Also | online sets online mode off or on  
grip_finish forces program to wait until gripper motion finished  
robotisfinished gets the robot finished state for non-control processes  
robotisdone gets the robot done state for non-control processes |
| Category | Motion |

### flock

**file lock**

| Description | Sets and releases advisory locks on a file. At any one time, a file can have:  
only one exclusive lock, or  
any number of shared locks. A flock() command can interruptably block. If the non-blocking flag, LOCK_NB, is used the operation does not block. If the non-blocking flag is absent, the operation blocks when locking. |
| Syntax | command  flock( int fd, int operation ) |
| Parameter |  
fd the file descriptor: an int  
operation the locking operation; one of: LOCK_SH  
shared lock; block until the lock is made  
LOCK_EX  
exclusive lock; block until the lock is made |
LOCK_SH | LOCK_NB
shared lock; return -EAGAIN immediately if this would have
blocked

LOCK_EX | LOCK_NB
exclusive lock; return -EAGAIN immediately if this would have
blocked

LOCK_UN
unlock

Returns
0 (-EOK) Success
-EINVAL An argument was invalid
-EBADF $fd$ does not correspond to an open file
-EAGAIN The LOCK_NB flag was set and we did not immediately
succeed.
-EINTR This operation was interrupted by a signal.

Example
open ($fd$,"test.txt",O_RDWR|O_TEXT|O_CREAT|O_TRUNC,M_READ|M_WRITE)
flock($fd$,LOCK_EX) ;; obtain an exclusive lock

Category
File and Device System Management

fprint

Description
Writes the specified data to the file associated with file descriptor $fd$. Two types
of arguments can be given in the variable argument list: constants and variables.
The constants are printed exactly as they are given. The variable's value is what
is copied to the file descriptor. The method used in printing is to print the
arguments in the exact order that they were given.

Syntax
command fprint ( int $fd$, ... )

Parameters
$fd$ file descriptor: an int
string constants or variables

Returns
$\geq 0$ Success
-EINVAL If the arguments (notably $fd$) are invalid.
-EBADF If $fd$ does not correspond to an open file.
-EACCESS If the file open on $fd$ is not open for writing.
-ESPIPE If an attempt is made to write to a socket.
-EIO An I/O error occurred.
-EAGAIN (nonblocking I/O only). Not ready to write any
bytes.
-EINTR This operation was interrupted by a signal.

Example
int $fd$
float cycle_count = 4
...
cycle_count = cycle_count + 1 ;; now at 5
open \
(fd,"test.txt",O_RDWR|O_TEXT|O_CREAT|O_TRUNC,M_READ|M_WRITE)
fprint ( $fd$, "Cycle ",cycle_count," data collection.\n")
close ($fd$)
Result

Cycle 5.00000 data collection.

sent to the file associated with file descriptor fd.

Category

File Input and Output: Unformatted Output

Device Input and Output

`fprintf`

Description

Converts and writes output to the file associated with file descriptor `fd` under the control of a specified format `fmt`.

Format specifications are detailed in the Formatted Output section of File Input and Output.

Syntax

`command fprintf( int fd, var string[] fmt, ... )`

Parameters

- `fd` - file descriptor
- `fmt` - formatted string

Format Specifiers

The format string may consist of two different objects, normal characters, which are directly copied to the file descriptor, and conversion braces which print the arguments to the descriptor. The conversion braces take the format:

```
{ [ flags ] [ field width ] [ .precision ] [ x | X ] }
```

Flags

Flags that are given in the conversion can be the following (in any order):

- `-` (minus sign) specifies left justification of the converted argument in its field.
- `+` (plus sign) specifies that the number will always have a sign.
- `0` (zero) in numeric conversions causes the field width to be padded with leading zeros.

Field width

The field width is the minimum field that the argument is to be printed in. If the converted argument has fewer characters than the field, then the argument is padded with spaces (unless the `0` (zero) flag was specified) on the left (or on the right if the `-` (minus sign) was specified). If the item takes more space than the specified field width, then the field width is exceeded.

precision

The precision number specifies the number of characters to be printed in a string, the number of significant digits in a float, or the maximum number of digits to be printed in an integer.

`x` or `X`

This is the hexadecimal flag which specifies whether or not an integer argument should be printed in hexadecimal (base 16) or not. The lowercase `x` specifies lowercase letters (`abcde`) are to be used in the hexadecimal display and the uppercase `X` specifies uppercase letters (`ABCDE`).

Returns

- `>= 0` - Success
- `-EINVAL` - If the arguments (notably `fd`) are invalid.
- `-EBADF` - If `fd` does not correspond to an open file.
-EACCESS  If the file open on \texttt{fd} is not open for writing.
-ESPIPE  If an attempt is made to write to a socket.
-EIO  An I/O error occurred.
-EAGAIN  (nonblocking I/O only). Not ready to write any bytes.
-EINTR  This operation was interrupted by a signal.

Example

```c
int fd
cycle_count = 4
...
cycle_count = cycle_count +1  ;; now at 5
open (fd,"test.txt",O_RDWR|O_TEXT|O_CREAT|O_TRUNC,M_READ|M_WRITE)
fprintf ( fd, "Cycle {6.4} data collection.\n",cycle_count)
close (fd)
```

Result  Cycle 5.000 data collection.

Category  File Input and Output: Formatted Output
Device Input and Output

---

**freadline**

**file read line**

Description  Reads (possibly interactively) a line of up to \texttt{maxlen} characters from \texttt{infd} into \texttt{str}. If \texttt{outfd} \(\geq 0\), then echoing is done to \texttt{outfd} and interactivity is assumed. The line terminator can be either a carriage return or a line feed. Returns the number of characters actually read including the terminator. A value of 0 means EOF. The function can return up to \texttt{maxlen} +1 since the end of line is included in the count, but not in the returned string.

Syntax  

```
command  freadline ( int \texttt{infd}, int \texttt{outfd}, var string[] \texttt{str}, int \texttt{maxlen} )
```

Parameters

- \texttt{infd}  file descriptor of data source
- \texttt{outfd}  file descriptor of echoed data or \(-1\) if you are reading from a file (with no echoing needed.)
- \texttt{str}  destination of data read from \texttt{infd}
- \texttt{maxlen}  maximum length of character read

Returns

- >= 0  Success; the number of characters read, including the terminator
- EINVAL  the arguments were invalid
- EBADF  one of the file descriptors do not correspond to an open file
- EACCESS  tried to read/write from a file that was not opened for the required access
- ESPPIPE  can’t r/w on a socket
- EIO  an I/O error occurred
- EAGAIN  (nonblocking I/O) no bytes were ready for reading / the device was not ready for writing
-EINTR this operation was interrupted by a signal

Example

```c
int fd
string[64] user_input
open (fd,"log.txt", O_RDWR|O_TEXT|O_CREAT, M_READ|M_WRITE)
seek (fd,0,SEEK_END) ;; append user
   ;; input to file
freadline (stdin,stdout,user_input,64) ;; input is read
   ;; from "stdin"
       ;; input is read
into string "user_input" and echoed out
to "stdout"
writes (fd,user_input,0) ;; write string to
   ;; file
writs (fd,"
",0) ;; write new line
   ;; char. to file
close (fd)
```

See Also

readline

Category

File Input and Output: Unformatted Input
Device Input and Output

---

**fstat**

Description

Obtains information about a particular open object in the file system.

Syntax

```c
command fstat( int fd, var c_dirent buf )
```

Parameters

- `fd` the file descriptor of the open object
- `buf` a `c_dirent` structure. See the information on `stat()` for further details.

Returns

- `>= 0` Success; buf is filled in with data about the object. Note that the `de_name` field will be a null string, as the system cannot currently find the name of the open object.
- `< 0` Failure

Possible failure codes are:
- `-EINVAL` the arguments were invalid.
- `-EBADF` there is no open object corresonding to `fd`.
- `-EIO` I/O error

Example

```c
int fd
c_dirent info
open(fd, "/conf/rc", O_RDONLY, 0)
...
fstat(fd, info)
printf("The /conf/rc file is {} bytes long.\n", info.de_size)
...
```

Result

The size of the /conf/rc file is displayed.

See Also

`stat()`

Category

File and Device System Management
ftime

Description
Changes the modification time of an open filesystem object.

Library
syslib

Syntax
command ftime(int fd, int modtime)

Parameters
There are two required parameters:

- \textit{fd} the open file descriptor
- \textit{modtime} what time to reset the object’s modification time to.

Returns
\begin{align*}
&\geq 0 & \rightarrow & \text{Success} \\
&< 0 & \rightarrow & \text{Failure}
\end{align*}

Possible failure return codes are:
- \texttt{-EINVAL} Invalid argument
- \texttt{-EBADF} There is no open file corresponding to \texttt{fd}.
- \texttt{-EACCESS} Access denied
- \texttt{-EIO} I/O error

Example
\begin{verbatim}
int fd, t
    t = time();       ;; get the time NOW
    open(fd, "myfile", O_RDWR, 0)
    ... 
    ftime(fd, t - 60); ;; reset the timestamp to one minute ago
    ...
\end{verbatim}

See Also
utime()

Category
File and Device System Management

gains_get

Description
Gets the gains for an axis.

Syntax
command gains_get( int axis, var float kp, var float ki, var float kd )

Parameters
\begin{align*}
\textit{axis} & \text{ the axis being inquired: an int} \\
\textit{kp} & \text{ proportional gain: a float} \\
\textit{ki} & \text{ integral gain: a float} \\
\textit{kd} & \text{ derivative gain: a float}
\end{align*}

Returns
\begin{align*}
&\geq 0 & \rightarrow & \text{Success} \\
&< 0 & \rightarrow & \text{Failure}
\end{align*}

Example
\begin{verbatim}
float p, i, d
    gains_get( 1, p, i, d )
    print ("p = ",p,"\ni = ",i,"\nd = ",d,"\n"
\end{verbatim}

Result
\begin{align*}
p &= 12.0000 \\
i &= 0.0200000 \\
d &= 100.0000
\end{align*}

See Also
gains_set sets the gains for an axis

Category
Robot Configuration
**gains_set**

**Description**
Sets the gains for an axis.

**Syntax**
```
command gains_set( int axis, var float kp, var float ki, var float kd )
```

**Parameters**
- `axis` the axis being set: an int
- `kp` proportional gain: a float
- `ki` integral gain: a float
- `kd` derivative gain: a float

**Returns**
- Success >= 0
- Failure < 0

**Example**
```
::An example to create an array of gains for each axis, and then set the gains to
::values stored
::in the array. The gains are then printed for each axis.
::
int axis_num, count
float[6] P, I, D

... initialize the array of gains
... for count =0 to 5
axis_num = count +1
  gains_set(axis_num, P[count], I[count], D[count])
  printf ("Axis_num, P:{}, I{}, D{} 
", P[count], I[count], D[count])
end for
```

**See Also**
gains_get gets the gains for an axis

**Category**
Robot Configuration

---

**get_ps**

**Description**
Obtains an entry in the system’s process table. Can be used to obtain all entries
one at a time, like the system shell’s ps command.

CROS-500 has room in the process table for 20 entries, numbered from 0 to 19.
CROSnt has room in the process table for 64 entries, numbered from 0 to 63.
Data is stored in the table from the back to the front — the oldest process, init, is
entry 19 or 63, the second oldest is 18 or 62, and so on. As a result, printing the
data by incrementing the slot number up to 19 or 63, places the oldest entry last,
like the system shell’s ps command.

Any empty slot in the process table is zeroed. Since processes have pids
numbered from 1, you can test for an empty slot by testing for a pid of 0 (zero).
This get_ps() command gets the process information for the entry identified by
`slot`. The information is stored in the ps_struct `ps`, which is a globally declared
struct. If `slot` is out of range, -EINVAL is returned.

**Syntax**
```
command get_ps( int slot, var ps_struct ps )
```
Parameters

- \textit{slot} the entry of the process table: an int (CROSnt: 0-63; CROS-500: 0-19)
- \textit{ps} the process information: a ps\_struct struct, with members
  - \textit{pid} an int
  - \textit{ppid} an int
  - \textit{flags} a constant of the enum ps\_flags, one of:
    - PR\_IN\_SYSTEM
    - PR\_NO\_SIGNAL
    - PR\_RAPL3 this is a RAPL-3 process
    - PR\_PRIVILEGED this is a privileged system process
    - PR\_INTERRUPTED
    - PR\_TIMEOUT
  - \textit{status} a constant of the enum ps\_status, one of:
    - PS\_FREE
    - PS\_HOLD
    - PS\_READY
    - PS\_RUN
    - PS\_SLEEP
    - PS\_STOP
    - PS\_ZOMBIE
    - PS\_WAITIO
    - PS\_WAITSEM
    - PS\_WAITSOCK
    - PS\_WAIT
  - \textit{prio} a constant of the enum ps\_priority, one of:
    - PR\_LOW
    - PR\_NORM
    - PR\_HIGH
  - \textit{sigmask} an int
  - \textit{sigpending} an int
  - \textit{sys\_fticks} an int
  - \textit{usr\_fticks} an int
  - \textit{rt\_slippage} an int
  - \textit{clicks} an int
  - \textit{argv0} the name of the process or program, a string[32]

Returns

- 0 (-EOK) Success
- -EINVAL \textit{slot} was out of range (negative or too large)

Example

```c
ps\_struct ps
get\_ps( 63, ps)
```

Example

```c
int slot = 0
ps\_struct ps ...
get\_ps( slot, ps)
```

Example

```c
int slot = 0
ps\_struct ps
int pid, status, ret
loop
  ret = get\_ps(slot, ps)
  if ret == -EINVAL
    break
  end if
  pid = ps.pid
  status = ps.status
  printf("pid \{2\} status \{2\} \n", pid, status)
  slot = slot + 1
end loop
```

Example

```c
int slot = 0
ps\_struct ps
```
string[]@[12] status_string = { 
    "FREE ", "HOLD ", "READY", "RUN ", 
    "SLEEP", "STOP ", "ZOMB ", "WIO ", 
    "WSEM ", "WSOCK", "WAIT ", "IWIO " }

...while((get_ps(slot, ps)) != -EINVAL)
    slot++
    if (ps.pid == 0)
        continue
    end if
    printf("pid {2} status {2} name {} \n" 
        ,ps.pid,status_string[ps.status],ps.argv0)
end while

RAPL-III

See Also
getpid  get the process's id number
getppid  get the parent's id number
module_name_get  get the name of the module

Category
System Process Control: Single and Multiple Processes

getenv

Description
Allows a program to retrieve the value of a specified environment string. [getenv() is available on a C500C only.]

Syntax
command getenv(var string[] dst, string[] key)

Parameters
There are two required parameters:
    dst    A string variable in which the result will be stored.
    key    The key to search for.

Returns
0  the key was not found; dst is set to the null string.
1  the key was found; dst is set to the value part of the string.
-ve  a negative error code.

Example
    ;; One of the environment strings that is always defined is
    ;; the SerialNumber string (which looks like:
    ;;  "SerialNumber=XYZ1234"
    ;; This code displays what the controller serial number is.
    ;; If the serial number environment string were as above, then
    ;; it would print the "XYZ1234" portion:
    string[32] sn
getenv(sn, "SerialNumber")
printf("The controller serial number is '{0}'\n", sn)

See Also
environ(), setenv(), unsetenv()

Category
Environment Variables

g getopt

Description
Provides a mechanism for handling command line arguments and options. It is patterned after the getopt(3) function of ANSI C. The getopt() function is based on the assumption that command lines look like this:
	name [-options] otherargs...

where name is the name of the command being run. [-options] is an optional list of option flags, each starting with a '-' character, and otherargs is a set of other items (not starting with '-') on the command line.
func int getopt(string[] opts)

Related vars

There are several related variables exported from syslib to support getopt():

- **int syslib:opterr**: This variable is a flag that the user can set before calling getopt(). If non-zero (which is the default), it indicates that getopt() should report errors on its own. A typical getopt() error message looks like:
  - `name: illegal option -X`
  - `name: option requires an argument -X`
  where `name` is the name of the program (as returned by argv(0)) and `X` is the option character with the problem.

- **int syslib:optind**: This variable indicates which argv() is the next one for getopt() to process.

- **string[256] syslib:optarg**: For options with arguments, getopt() places the argument string in here.

Parameters

- **opts**: A string with a list of all the valid option flags. For example, if the string is “abc”, then getopt() expects that “-a”, “-b” and “-c” are all valid options for the command. If an option letter in opts is followed by a ‘:’, then the option is supposed to have an argument following it. For example, if opts is “af:h”, then the valid options are “-a”, “-h” and “-f argument” or “-f:argument”.

Returns

- **Success**: the character from the opts string that was matched, or EOARGS (which is -1) if we have run out of option flags to parse.
- **Failure**: ‘?’ if an unrecognized or illegal option was found. If syslib:opterr is not zero, then getopt() reports the error before returning the ‘?’.

Example

The getopt() function is rather complex, and in more need than most of an example. The following short program illustrates how to use getopt():

```c
sub usage();
    fprintf(stderr, “Usage: {} [-options] arg1 [arg2...]
”, argv(0))
    fprintf(stderr, “Options are: \n”)
    fprintf(stderr, “-a do something
”)
    fprintf(stderr, “-b do something else
”)
    fprintf(stderr, “-c target do something to someone
”)
    fprintf(stderr, “-h, -? display this message
”)
    exit(1)
end sub

main
int ch
loop
    ch = getopt(“abc:h?”)
    if (ch < 0)
        break
    end if
    case (ch)
        of ‘a’:
            printf(“got -a\n”)
        of ‘b’:
            printf(“got -b\n”)
        of ‘c’:
```
printf("got -c {}
", syslib:optarg)
else
  if \(?\) and \(h\) fall into here as well
    usage()
  end case
end loop

if (syslib:optind == argc())
  ;; we don’t have an arg1 - we are at the end of the list
  fprintf(stderr, "{}: missing argument
", argv(0))
  usage()
end if

printf("The other arguments are:
")
while (syslib:optind < argc())
  printf(" {}
", argv(syslib:optind))
syslib:optind++
end while

exit(0)
end main

See Also
  argc(), argv()

Category
  System Process Control: Single and Multiple Processes

getpid

Description
  Returns the id number of the process of the calling program.

Syntax
  func int getpid()

Returns
  The process id of the calling program.

Example
  int pid ...
  pid = getpid() ;; get our process id number

See Also
  getps gets entry in process table
  getppid get the parent’s id number
  module_name_get get the name of the module

Category
  System Process Control: Single and Multiple Processes

getppid

Description
  Returns the id number of the parent process of the calling program.

Syntax
  func int getppid()

Returns
  The process id of the parent of the calling process.

Example
  int ppid ...
  ppid = getppid() ;; get our parent process id number

See Also
  getps gets entry in process table
  getppid get the parent’s id number
  module_name_get get the name of the module

Category
  System Process Control: Single and Multiple Processes

grip

Alias of
  gripdist_set
**grip**

**Description**
Moves the fingers of the servo-gripper to a specified distance apart from each other.

**Example**
grip(1.0)

**See**
gripdist_get gets the current servo finger separation distance

**Category**
Gripper

**grip_cal**

**Description**
Calibrates the gripper by setting travel distance.

**Syntax**
command grip_cal( float mindist, float maxdist )

**Parameters**
mindist the minimum distance for finger travel: a float
maxdist the maximum distance for finger travel: a float

**Returns**
Success >= 0
Failure < 0

**Example**
grip_cal( 0.0, 50.80 ) ;; millimetres for standard servogripper

grip_cal( 25.0, 50.0 ) ;; min and max for custom fingers and objects

grip_cal( 0.0, 2.0 ) ;; inches for standard servogripper

**See Also**
calibrate calibrate the arm axes
gripdist_set opens/closes servo fingers to specified separation distance
gripdist_get gets current servo finger separation distance
grip_open opens the gripper
grip_close closes the gripper

**Category**
Gripper
Calibration

**grip_close**

**Description**
Closes the gripper. If configured with a servo gripper the command accepts an optional argument specifying the force used by the gripper. The argument is given as a percentage of full force valid range 0 to 100.

Fingers can be machined to surround an object and grasp it on the outside, or machined to be inserted into a hole and grasp the object by exerting force on the insides of the hole. This configuration determines whether the object is grasped by grip_close() and released by grip_open(), or grasped by grip_open() and released by grip_close().

**Warning**
Gripping at a force above 75% for more than a few seconds may shorten the life of the servo-gripper. To grip an object without overloading the gripper, after initially making contact with the object, reduce the force. The servo-gripper mechanics keep a firm grip on the object.

**Syntax**
command grip_close( [int servo_force] )
**grip_close**

**Description**
Like the finish() command, holds execution of the program at the grip_close() command until gripper motion has finished. Normally a command is executed as soon as its parameters are determined, which can be before the previous command has finished. grip_close() is often used to finish the motion of the gripper at or near a location before moving the arm. Also used to synchronize commands, such as input/output, with gripper motion.

If online mode is off, online(OFF), grip_close() is not needed between two gripper motion commands. Gripper motion commands are executed as if there is a grip_close() after each one.

**Syntax**
```plaintext
command grip_close()
```

**Parameter**

- empty

**Returns**

- Success \(\geq 0\)
- Failure \(< 0\)

**Example**
```plaintext
move(get_part)
finish()
grip_close(100)
grip_finish()
msleep(200)
grip_close(60)
```

**See Also**

- grip_open: opens the gripper; opposite of grip_close
- gripdist_set: sets the servo fingers at a separation distance
- gripdist_get: gets the current servo finger separation distance

**Category**

- Gripper
- Motion

---

**grip_open**

**Description**
Opens the gripper. Takes an optional argument for a servo-gripper, of the percentage of force with a valid range between 0 - 100.

Fingers can be machined to surround an object and grasp it on the outside, or machined to be inserted into a hole and grasp the object by exerting force on the
inside of the hole. This configuration determines whether the object is grasped
by gripclose() and released by gripopen(), or grasped by gripopen() and released by
gripclose().

Warning
Gripping at a force above 75% for more than a few seconds may shorten the life
of the servo-gripper. To grip an object without overloading the gripper, after
initially making contact with the object, reduce the force. The servo-gripper
mechanics keep a firm grip on the object.

Syntax
command grip_open( [int servo_force] )

Argument (Optional)
  servo_force the percentage of force applied: an int

Returns
  Success >= 0
  Failure < 0

Example
move(set_part)
finish()
grip_open()
grip_finish()
depart(2.0)

RAPL-II
Similar to OPEN.

See Also
  grip_close closes the gripper; opposite of grip_open
  gripdist_set sets the servo fingers at a separation distance
  gripdist_get gets the current servo-finger separation distance

Category
Gripper
Motion

---

gripdist_get

Description
  Gets the distance between fingers of the servo-gripper.

Syntax
command gripdist_get( var float distance)

Parameter
  distance float variable to store current gripper distance

Returns
  Success >= 0. The finger distance: a float.
  Failure < 0

Example
float my_gripper_dist
... close (100)
grip_finish()
gripdist_get( my_gripper_dist )
if my_gripper_dist <=30
  return (-1) ;; gripper has no part in fingers
else
  return (0) ;; gripper has part in fingers
end if

RAPL-II
WGRIP()

See Also
  grip sets the finger separation distance
  setgriptypesets the gripper type (air, servo, etc.)

Category
Gripper

---

gripdist_set

Alias
  grip

<table>
<thead>
<tr>
<th>alias</th>
<th>same as</th>
</tr>
</thead>
<tbody>
<tr>
<td>grip(...)</td>
<td>gripdist_set(...)</td>
</tr>
</tbody>
</table>

Title: Reference Guide

**Description**
Moves the fingers of the servo-gripper to a specified distance apart from each other.

To attain the grip distance, fingers open or close depending on the starting position.

**Warning**
Do not use this command to hold an object. This will damage the gripper. The `gripdist_set()` command operates at 100% force. To control gripper force and hold an object, use the `gripclose()` and `gripopen()` commands.

**Syntax**
```command gripdist_set( float distance )```

**Parameter**
`distance` the distance between fingers in current units: a float

**Returns**
Success >= 0
Failure < 0

**Example**
gripdist_set(1.0)

**RAPL-II**
Similar to GRIP.

**See Also**
gripdist_get gets the current servo finger separation distance
grip_close closes the gripper (with force for servo)
grip_open opens the gripper (with force for servo)

**Category**
Gripper
Motion

---

**gripisfinished**

**Description**
Determines if the gripper is finished moving. Returns FALSE (0), TRUE, or error <0.

**Syntax**
```command gripisfinished()```

**Parameters**
empty

**Returns**
Success >= 0
Failure < 0

**Example**
```int depart_dis
teachable ploc place...
move(place)
grip_close(50)
loop
  if gripisfinished()
    depart(depart_dis)
  else
    msleep(250)
  endif
end loop```

**Result**
Depart location place after the gripper is closed.

**See Also**
grip_close
grip_finish

**Category**
Gripper
Robot Configuration

---

**gripper_stop**

**Description**
The command stops any gripper motion.
gripter_stop

Syntax
command gripter_stop()

Returns
Success >= 0
Failure < 0  Returns -ve error descriptor if command fails.

Example
...  
gripter_stop()

Result
Gripper motion stops

See Also
  grip_open
  grip_close
  gripdist_set
  gripdist_get

Category
Gripper
Motion

griptype_get

Description
Gets what the robot gripper type is currently set to.

Syntax
command griptype_get(var grip_type gtype)

Returns
Success >= 0; gtype is filled in with the gripper type code.
Failure < 0 (-ve error code)

Example
This RAPL-3 code segment displays, in words, the setting of the gripper type:

    int gtype
    griptype_get(gtype)
    case (gtype)
    of 0:
        printf("No gripper type selected\n")
    of GTYPE_AIR:
        printf("Air gripper selected\n")
    of GTYPE_SERVO:
        printf("Servo gripper selected\n")
    end case

See Also
  griptype_set() 

Category
Gripper

griptype_set

Description
Sets the gripper type to correspond to the gripper in use. Gripper type must be set to GTYPE_SERVO to use the gripdist_set() or gripdist_get() command.

Syntax
command griptype_set(grip_type gtype)

Parameters
One of:
  GTYPE_AIR       for air grippers (the default)
  GTYPE_SERVO    for servo-motor grippers

Returns
Success >= 0
Failure < 0

Example
  griptype_set( GTYPE_SERVO )

RAPL-II
@@SETUP grip type questions
See Also

- `grip_open` opens the gripper
- `grip_close` closes the gripper
- `gripdist_set` opens/closes servo fingers to specified separation distance
- `gripdist_get` gets current servo finger separation distance
- `grip_finish` finishes current gripper motion
- `gripisfinished` determines if the gripper motion is finished

Category

- Gripper
- Robot Configuration

---

**halt**

**Description**
Stops any current robot motion.

**Syntax**
```
command halt()
```

**Parameter**
- (empty)

**Returns**
- Success >= 0
- Failure < 0

**Example**
```
halt()
```

**RAPL-II**
Similar to HALT.

**See Also**
- `finish` finishes current motion command before next motion

**Category**
Motion

---

**heap_set**

**Description**
Sets the heap size for current application. The heap is a storage space that can be allocated under user control. The default size is 4K bytes which equals 1K words (4 bytes = 1 word). The command `heap_set()` sets the heap size of the current process to at least `size` words. Note that if you run out of heap space, the system will attempt to allocate you more. That being said, it is generally better (and faster) to simply allocate enough for your program at the start.

Note that if `heap_set()` is called after allocations have already been done, resetting the heap size may be time consuming.

**Syntax**
```
command heap_set( int size )
```

**Parameter**
- `size` integer value of the size of memory to be allocated in words (word = 4 bytes)

**Returns**
- Success >= 0
- -ENOMEM There is not enough memory for the requested operation.
- -EINVAL `size` is a nonsensical value (ie., negative)

**Example**
```
int mem = 8192
heap_set(mem)
...;
;; allocate memory needed using mem_alloc() command
```

**Result**
Allocates 8192 bytes of memory

**See Also**
- `heap_space` determines the longest free area in the heap
- `heap_size` returns the number of words in heap segment
mem_alloc allocates memory -(can increase allocated heap if necessary)
mem_free free memory space

**Category**
Memory

---

**heap_size**

**Description**
Returns the number of words in the heap segment of the current process. This total size includes free, allocated, and overhead.

**Syntax**
func int heap_size()

**Parameters**
none

**Returns**
Returns the number of words the entire heap currently occupies.

**Example**
```
int size_heap
size_heap=heap_size()
if (size_heap < 16)
    heap_set(16)
end if
```

**Result**
If the heap is not at least 16 Kbytes then it is set to 16 Kbytes.

**See Also**
heap_space() find the amount of free space in the heap
heap_set() set the total amount of space in the heap

**Category**
Memory

---

**heap_space**

**Description**
Determines the length of the longest contiguous free area available in the program's heap. If an object greater than this size is allocated using mem_alloc() then the system will have to expand the size of the heap.

**Syntax**
func int heap_space()

**Returns**
The length of longest contiguous area, in words.

**Example**
```
int heap_bloc, space = 3
void@ ptr
heap_bloc = heap_space()
if heap_bloc < 5
    printf("heap space is low/n")
    ...
    mem_alloc(ptr, space)
else
    mem_alloc(ptr, space)
end if
```

**Result**
Allocates memory of 3 words (12 bytes) - Notifies user if heap space is less than 5 Kbytes.

**RAPL-II**
Similar to FREE

**See Also**
mem_alloc() allocates an area of memory and initializes it
mem_free() de-allocates an area of memory
heap_set() sets the heap size of the current process
heap_size() determines how big the heap is in total.

**Category**
Memory
here

Description
Stores the current commanded robot location in the specified location variable. A precision or cartesian location is stored, depending on the location type of the input variable. Currently, the location’s type must be explicitly defined prior to use in the here() command.

Syntax
command here( var gloc location )

Returns
Success >= 0
Failure < 0

Example
loc_class_set( #first ,loc_precision)
loc_class_set( _last ,loc_cartesian)
...
here( first ) ;;store precision location
...
here( last ) ;;store cartesian location

home

Description
Homes the specified axes in numerical order: 1 (waist), 2 (shoulder), 3 (elbow), 4, 5, 6. This command assumes the robot has been correctly calibrated.

Syntax
command home( [axis] [,axis] [,axis] ... )

Parameter(s)
axis an axis to home

Returns
Success >= 0
Failure < 0

Example
if home(7) >= 0
  if home(1,2,3,4,5,6) >= 0
    else
      print "Error homing arm.\n"
    end if
  else
    print "Error homing track.\n"
  end if
else
  print "Error homing track.\n"
end if

homezc

Description
Homes the axis specified, and returns the offset in pulses.

Syntax
command homezc( int axis, var int offset )

Parameter(s)
axis an axis to home
offset the offset
Returns

Success >= 0
Failure < 0

Example

int machine, transform, actual, i
int[8] offsets

axes_get(machine, transform, actual)
    for i = 1 to machine
        homezc(i, offsets[i])
        printf("axis {1} offset is {}
", i,offsets[i])
    end for

Result
Homing axis 1... OK
axis 1 offset is 519

RAPL-II
Same as HOMEZC.

See Also
calzc calibrates at the next zero pulse of the encoder
calibrate calibrates axes
home homes the specified axes in numerical order
ready moves the arm to the READY position
robotishomed gets the homed or not-homed state of axes

Category Home

hsw_offset_get

Description
Returns the offset between the homing switch and the calibration position of a given axis, in encoder pulses. Used with an A465.

Syntax
func int hsw_offset-get( int axis )

Parameter
axis the axis to be inquired: an int

Returns

Example

int machine, transform, actual, i, robot
int[8] offsets

robot = robot_type_get()
printf("robot is {}
", robot)

    if robot == 465
        axes_get(machine, transform, actual)
        for i = 1 to machine
            offsets[i] = hsw_offset_get(i)
            printf("axis {1} offset is {}
", i,offsets[i])
        end for
    else
        printf("Robot must be a 465 for this command")
    end if

Result
Prints the offsets for each axis, if the robot is an A465

See Also
homezc homes the axis specified

Category
Calibration
Home

iabs

Description
Calculates the absolute value of an int.
Syntax

`func int iabs(int x)`

Argument

`x`  the number: an int

Returns

The absolute value of the integer `x`. Note that one integer (-2147483648) does not have a positive counterpart because of the limitations of 32-bit 2’s complement binary numbers.

Example

```plaintext
int x = -99
int y
y = abs(x)
```

Result

99

RAPL-II

ABS

See Also

`fabs` calculates the absolute value of a float

Category

Math

---

**input**

Description

Queries the specified input channel for its state. Returns the state.

This subprogram is a function, not a command as it was in the earliest versions of RAPL-3.

Syntax

`func int input(int channel)`

Parameters

`channel`  the input channel: an int

Returns

Success >= 0

the state, an int, one of:

0 = off

1 = on

Failure < 0  Returns error code

Example 1

```plaintext
state = input(4)
```

Example 2

```plaintext
if (input(8)) then ;; check sensor for presence of material
    load_part() ;; material present
else
    continue ;; material not present
end if
```

Application Shell

Similar to `input`.

RAPL-II

Similar to `INPUT`, but `INPUT` packed the state into a variable, and could be used for digital and string input.

See Also

`inputs` queries the entire bank of input channels for their states

Category

Digital Input and Output

---

**inputs**

Description

Queries the entire bank of input channels for their states. Returns an integer that represents the bitmapped states of the inputs.
For the C500 controller, each of the first 16 bits represents an input. The least significant bit is input 1, the sixteenth significant bit is input 16. The integer in hex

```
func int inputs()
```

**Parameters**

none

**Returns**

Success >= 0

the input states: an int representing a bitmask where the lower 16 bits each correspond to one of the inputs:

<table>
<thead>
<tr>
<th>Bit</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>off</td>
</tr>
<tr>
<td>1</td>
<td>on</td>
</tr>
</tbody>
</table>

Failure < 0 Returns error code

**Example**

```c
int dig_inputs

dig_inputs = inputs(); ;; read all inputs
dig_inputs = dig_inputs & 0xf ;; enable lower 4 bits only

case dig_inputs
    of 1: ;; first input is high
        task_1();
    of 2: ;; second input is high
        task_2();
    of 4: ;; third input is high
        task_3();
    of 8: ;; fourth input is high
        task_4();
end case
```

**Application Shell**

No equivalent.

**RAPL-II**

No equivalent.

**See Also**

input queries an input channel for its state

outputs sets the entire bank of output channels to states

outputs_get queries the entire bank of output channels for their states

**Category**

Digital Input and Output

---

**ioctl**

**Description**

I/O control operation. Used to configure and control a device.

If a get parameter is used, the data is stored. If a put parameter is used, the data is written.

To change a serial port configuration, read the current status into one of the data structures, change the data for specific members of the struct, and write the new data for the port.

**Syntax**

```c
command ioctl( int fd, ioctl_op op, void* data )
```

**Parameters**

- **fd**
  - the port

- **op**
  - the operation, of type ioctl_op:
    - IOCTL_NOP no operation
    - IOCTL_GETC get configuration information
    - IOCTL_PUTC put configuration information
    - IOCTL_GETS get status information
    - IOCTL_PUTS put status information
    - IOCTL_GETSIG get special signal information
    - IOCTL_PUTSIG put special signal information
    - IOCTL_RDTIME set read timeout
    - IOCTL_WRTIME set write timeout

- **data**
  - a struct of integers of type sio_ioctl_conf:
    - int baud
    - baud rate
    - int res_01
int res_02
int OutxCtsFlow 1 => enable CTS output flow control
int OutxDsrFlow 1 => enable DSR output flow control
int DtrControl 1 => enable DTR flow control
int DsrSensitivity 1 => enable DSR sensitivity
int TXContinueOnXoff 1 => continue trans after sending XOFF
int OutX 1 => enable output Xoff flow control
int InX 1 => enable input Xoff flow control
int res_10
int res_11
int RtsControl 1 => enable RTS flow control
int res_13
int res_14
int res_15
int lowtrig soft flow low trigger (xon point)
int hightrig soft flow high trigger (xoff point)
int wordlen word length (7 or 8 bits)
int parity 0 => none, 1 => odd, 2 => even
int stopbits 1 => 1 bit, 2 => 2 bits, 15 => 1.5 bits
int xonchar soft flow xon char
int xoffchar soft flow xoff char
int res_23
int res_24
int res_25
int fifotrig 0 => 1 byte, 1 => 4; 2 => 8; 3 => 14 bytes
int lfchar (unimpl) lf char for auto cr
int crchar (unimpl) cr char to emit for auto cr
int autocr (unimpl) enable auto cr
int res_30

Returns

>= 0 Success
-EINVAL one of the arguments is invalid
-EBADF fd does not correspond to an open object
-ENODEV the object open on fd is not a device
-ENOTTY the device does not support ioctl()
-EIO an I/O error has occurred

System Shell
Same as siocfg

RAPL-II
CONFIG, SERIAL

Category
Device Input and Output

jog_t

Aliases

tx, ty, tz, yaw, pitch, roll

<table>
<thead>
<tr>
<th>alias</th>
<th>same as</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Description

In the tool frame of reference, moves the tool centre point in a cartesian-axis direction. TOOL_X, TOOL_Y, and TOOL_Z move the tool centre point along the X, Y, and Z axis by the specified distance in current units (millimetres or inches). TOOL_YAW, TOOL_PITCH, and TOOL_ROLL rotate around an axis by the specified rotation in degrees.

Yaw, pitch, and roll are tool motion based, not tool axis based. The command gives the same motion, although the robots have different coordinate systems.

<table>
<thead>
<tr>
<th>motion</th>
<th>common name</th>
<th>F3 coordinate system</th>
<th>A465/A255 coordinate system</th>
</tr>
</thead>
<tbody>
<tr>
<td>yaw</td>
<td>normal</td>
<td>X</td>
<td>Z</td>
</tr>
<tr>
<td>pitch</td>
<td>orientation</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>roll</td>
<td>approach/depart</td>
<td>Z</td>
<td>X</td>
</tr>
</tbody>
</table>

This command, jog_t(), is joint-interpolated.

For motion along an axis (TOOL_X, TOOL_Y, TOOL_Z), the end-point is along the tool axis, but the tool centre point travels as a result of various joint motions, not in a straight line.

Similarly for rotation around an axis (TOOL_YAW, TOOL_PITCH, TOOL_ROLL), the end-point is determined and the tool travels to it as a result of various joint motions. The start point and end point for the tool centre point are the same (no change in distance along the axis or angle between the axis and the tool), but the start position and end position of the tool are different by the amount of rotation.

For cartesian-interpolated (straight line) motion, see jog_ts().

Syntax

command jog_t( tool_axis_t axis, float distance )

Parameters

axis

- TOOL_X along the X axis
- TOOL_Y along the Y axis
- TOOL_Z along the Z axis
- TOOL_YAW around the normal axis
- TOOL_PITCH around the orientation axis
- TOOL_ROLL around the approach/depart axis

distance

the distance of travel, in current units or degrees: a float

Returns

Success = 0
Failure < 0

Example

jog_t(TOOL_Z,200) ;; millimetres
jog_t(TOOL_Y,-200)

Example

move(centre)
jog_t(TOOL_PITCH,45) ;; rotate around Y
jog_t(TOOL_PITCH,-90)

RAPL-II

No equivalents. DEPART moved along the approach/depart axis.
See Also

- jog_ts: jogs like jog_t, but straight line motion
- jog_w: jogs like jog_t, but in world frame of reference
- joint: moves by joint degrees
- motor: moves by encoder pulses

Category

Motion

jog_ts

Aliases

txs, tys, tzs, yaws, pitchs, rolls

<table>
<thead>
<tr>
<th>alias</th>
<th>same as</th>
</tr>
</thead>
<tbody>
<tr>
<td>txs(...)</td>
<td>jog_ts(TOOL_X, ...)</td>
</tr>
<tr>
<td>tys(...)</td>
<td>jog_ts(TOOL_Y, ...)</td>
</tr>
<tr>
<td>tzs(...)</td>
<td>jog_ts(TOOL_Z, ...)</td>
</tr>
<tr>
<td>yaws(...)</td>
<td>jog_ts(TOOL_YAW, ... )</td>
</tr>
<tr>
<td>pitchs(...)</td>
<td>jog_ts(TOOL_PITCH, ... )</td>
</tr>
<tr>
<td>rolls(...)</td>
<td>jog_ts(TOOL_ROLL, ... )</td>
</tr>
</tbody>
</table>

Description

In the tool frame of reference, moves the tool centre point in a cartesian-axis direction. TOOL_X, TOOL_Y, and TOOL_Z move the tool centre point along the X, Y, and Z axis by the specified distance in current units (millimetres or inches). TOOL_YAW, TOOL_PITCH, and TOOL_ROLL rotate around an axis by the specified rotation in degrees.

Yaw, pitch, and roll are tool motion based, not tool axis based. The command gives the same motion, although the robots have different coordinate systems.

<table>
<thead>
<tr>
<th>motion</th>
<th>axes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>common name</td>
</tr>
<tr>
<td>yaw</td>
<td>normal</td>
</tr>
<tr>
<td>pitch</td>
<td>orientation</td>
</tr>
<tr>
<td>roll</td>
<td>approach/depart</td>
</tr>
</tbody>
</table>

This command, jog_ts(), is cartesian-interpolated (straight line).

For motion along an axis (TOOL_X, TOOL_Y, TOOL_Z), the tool centre point travels in a straight line along the axis to the end point.

For rotation around an axis (TOOL_YAW, TOOL_PITCH, TOOL_ROLL), the tool centre point stays on the axis, while the tool rotates around the axis. The tool centre point stays in the same place.

For joint-interpolated (not straight) motion, see jog_t()

Syntax

command jog_ts( tool_axis_t axis, float distance )

Parameters

- axis: the axis for motion
  - TOOL_X: along the X axis
  - TOOL_Y: along the Y axis
  - TOOL_Z: along the Z axis
  - TOOL_YAW: around the normal axis
  - TOOL_PITCH: around the orientation axis
TOOL_ROLL

distance

around the approach/depart axis

the distance of travel, in current units or degrees: a float

Returns

Success = 0
Failure < 0

Example

jog_ts(TOOL_Z, 200) ;; millimetres
jog_ts(TOOL_Y, -200)

Example

move(centre)
jog_ts(TOOL_PITCH, 45) ;; rotate around Y
jog_ts(TOOL_PITCH, -90)

RAPL-II

No equivalents. DEPART moved along the approach/depart axis.

See Also

jog_t jogs like jog_ts, but joint interpolated
jog_ws jogs like jog_ts, but in world frame of reference
joint moves by joint degrees
motor moves by encoder pulses

Category

Motion

jog_w

wx, wy, wz, zrot, yrot, xrot

Aliases

wx(...) jog_w(WORLD_X, ... )
wy(...) jog_w(WORLD_Y, ... )
wz(...) jog_w(WORLD_Z, ... )
zrot(...) jog_w(WORLD_ZROT, ... )
yrot(...) jog_w(WORLD_YROT, ... )
xrot(...) jog_w(WORLD_XROT, ... )

Description

In the world frame of reference, moves the tool centre point in a cartesian-axis direction. WORLD_X, WORLD_Y, and WORLD_Z move the tool centre point along the X, Y, and Z axis by the specified distance in current units (millimetres or inches). WORLD_ZROT, WORLD_YROT, and WORLD_XROT rotate around the Z, Y, and X axis by the specified rotation in degrees.

This command, jog_w(), is joint-interpolated.

For motion along an axis (WORLD_X, WORLD_Y, WORLD_Z), the end-point is along the world axis, but the tool centre point travels as a result of various joint motions, not in a straight line.

Similarly for rotation around an axis (WORLD_ZROT, WORLD_YROT, WORLD_XROT), the end-point is determined and the tool travels to it as a result of various joint motions. The start point and end point for the tool centre point are the same (no change in distance along the axis or angle between the axis and the tool), but the start position and end position of the tool are different.

For cartesian-interpolated (straight line) motion, see jog_ws().

Syntax

command jog_w( world_axis_t axis, float distance )

Parameters

axis the axis for motion

WORLD_X along the X axis
WORLD_Y along the Y axis
WORLD_Z along the Z axis
WORLD_ZROT around the Z axis
WORLD_YROT around the Y axis
WORLD_XROT around the X axis

distance the distance of travel, in current units or degrees: a float

Returns
Success = 0
Failure < 0

Example
move(base_point)
jog_w(WORLD_X,200) ;; millimetres

Example
appro(centre)
pitch(45) ;; pitch around tool point
jog_w(WORLD_XROT,45) ;; rotate around X

RAPL-II
Similar to JOG, X, Y, Z, without straight line parameter.
Also similar to YAW, PITCH, and ROLL. In RAPL-II these names were used for
rotations in the world frame of reference. In RAPL-3, world rotations are called
zrot, yrot, and xrot, and tool rotations are called yaw, pitch, and roll.

See Also
jog_ws jogs like jog_w, but straight line motion
jog_t jogs like jog_w, but in tool frame of reference
joint moves by joint degrees
motor moves by encoder pulses

Category Motion

jog_ws

Aliases

wxs, wys, wzs, zrots, yrots, xrots

Description
In the world frame of reference, moves the tool centre point in a cartesian-axis
direction. WORLD_X, WORLD_Y, and WORLD_Z move the tool centre point along
the X, Y, and Z axis by the specified distance in current units (millimetres or
inches). WORLD_ZROT, WORLD_YROT, and WORLD_XROT rotate around the Z,
Y, and X axis by the specified rotation in degrees.

This command, jog_ws(), is cartesian-interpolated (straight line).

For motion along an axis (WORLD_X, WORLD_Y, WORLD_Z), the tool centre point
travels in a straight line along the axis to the end point.

For rotation around an axis (WORLD_ZROT, WORLD_YROT, WORLD_XROT), the
tool centre point stays on the axis, while the tool rotates around the axis. The tool
centre point stays in the same place.

For joint-interpolated (not straight) motion, see jog_w()
joint

Description
Rotates a rotational joint (e.g. of an articulated arm) by a specified number of degrees, or moves a linear joint (e.g. of a track or gantry) by a defined number of units (millimetres or inches depending on metric or English mode).

Syntax
command joint( int axis, float distance )

Parameters
axis 
the axis being moved: an int

distance 
the distance of travel, in current units: a float

Returns
Success >= 0
Failure < 0

Example
joint(7,20) ;; moves the track (for F3 or A465) 20 units
joint(1,45) ;; moves the waist joint +45 degrees

RAPL-II
Similar to JOINT

See Also
jog 
moves by cartesian increment

motor 
moves by encoder pulses

Category
Motion

joint_to_motor

Description
Converts a location from joint angles to motor pulses. Used if a location of one type needs to be converted to another type for checking or other use within the program.

Syntax
command joint_to_motor( var float[8] joint, var ploc motor )

Parameters
joint 
the location in joint angles, in degrees

motor 
the location in motor pulses: a ploc
**Returns**

Success >= 0

*motor* is packed

Failure < 0

**Example**

```plaintext
float[8] joints1 = (10, -15, 5, 0, 0, 0, 0, 0)
ploc motor1
...
joint_to_motor(joints1, motor1)
```

**Result**

*motor1* is packed with the appropriate pulse data

**RAPL-II**

Similar to SET with different location types.

**See Also**

- *motor_to_joint* converts motor pulses to joint angles
- *joint_to_world* converts joint angles to world coordinates

**Category**

Location: Kinematic Conversion

---

### joint_to_world

**Description**

Converts a location from joint angles to world coordinates. Used if a location of one type needs to be converted to another type for checking or other use within the program.

**Syntax**

```plaintext
command joint_to_world(var float[8] joint, var cloc world)
```

**Parameters**

- *joint* the location in joint angles
- *world* the location in world coordinates: a cloc

**Returns**

Success >= 0

*world* is packed

Failure < 0

**Example**

```plaintext
float[8] joints1 = (10, -15, 5, 0, 0, 0, 0, 0)
cloc world1
...
joint_to_world(joints1, world1)
```

**Result**

*world1* is packed with the appropriate world coordinate data

**RAPL-II**

Similar to SET with different location types.

**See Also**

- *world_to_joint* converts world coordinates to joint angles
- *joint_to_motor* converts joint angles to motor pulses

**Category**

Location: Kinematic Conversion

---

### jointlim_get

**Description**

Gets the positive and negative limits of travel for a specified axis.

**Syntax**

```plaintext
command jointlim_get( int axis, var float poslim, var float neglim )
```

**Parameter**

- *axis* an int specifying the axis
- *poslim* the positive limit: an array of up to 8 floats
- *neglim* the negative limit: an array of up to 8 floats

**Returns**

Success >= 0

Failure < 0

**Example**

```plaintext
int axes, total, trnsfrm
float[8] pluslim, neglim
```
int count, t
...
t = axes_get(axes, trnsfrm, total)
if t>0
    for count = 1 to axes
        jointlim_get(count, pluslim[count-1], neglim[count-1])
        printf("axis {2} limits are: +{5} - {5}/n", count, \\
               pluslim[count-1], neglim[count-1])
    end for
else
    ... use for error handling
end if

Result
Prints the robot joint limits

See Also
jointlim_set

Category
Robot Configuration

jointlim_set

Description
Sets the positive and negative limits of travel for one axis.

Syntax
command jointlim_set( int axis, float poslim, float neglim )

Parameter
axis the axis to set: an int
poslim the positive limit: a float
neglim the negative limit: a float

Returns
Success >= 0
Failure < 0

Example
int count
int axes, total, trnsfrm
teachable float[8] pluslim, neglim
axes_get(axes, trnsfrm, total)
for count = 1 to axes
    jointlim_set(count, pluslim[count-1], neglim[count-1])
end for

RAPL-II
Similar to @XLIMITS, except @XLIMITS took the limit in radians.
See Also
jointlim_get
Category
Robot Configuration

limp

Description
Disengages the servo control of a motor which limps that joint. A single axis or several axes can be specified. All axes are specified by an empty parameter.

Warning
Provide adequate support for arm links before limping any joint. Without adequate support, they can drop suddenly when the joint is limped, and may cause damage or injury.

Syntax
command limp([ int axis ] [, int axis ] ...)

Parameters (Optional)
(empty) all axes limped
axis axis being limped: an int
returns  Success >= 0  
Failure < 0
Example
limp()    ;; limps all axes
limp(3)    ;; limps axis 3
limp(4, 5, 6)    ;; limps axis 4, 5, and 6
RARP-II
Similar to LIMP.
See Also
nolimp    unlimps axes
Category
Motion

linacc_get
Description
Returns the current value of the robot's linear acceleration in metric or English units.
Syntax
command linacc_get(var float linacc)
Parameter
linacc is packed with the current acceleration setting
Returns
Success >= 0  
Failure < 0    Returns -ve error descriptor if command fails.
Example
float acc
printf("The linear acceleration is {}/n", linacc_get(acc))
Result
The linear acceleration is 1016.
See Also
linacc_set    sets the linear speed
units_set    sets the current units metric or English
linspd_get    returns the maximum linear speed
linspd_set    sets the linear speed depending on the configuration
Category
Robot Configuration

linacc_set
Description
Sets the current value of the robot's linear acceleration in metric or English units to the value specified by the parameter linacc.
Syntax
command linacc_set(var float linacc)
Parameter
linacc specifies the requested setting for the robot acceleration.
Returns
Success >= 0  
Failure < 0    Returns -ve error descriptor if command fails.
Example
;; Decrease the acceleration by 50 percent
;; Current acceleration is 1016 mm/sec²
float old_acc, new_acc
linacc_get(old_acc)
printf("The acceleration was {}/n", old_acc)
new_acc = old_acc*0.5
linacc_set(new_acc)
printf("The acceleration is now {}/n",new_acc)
Result
The acceleration was 1016.
The acceleration is now 508.
See Also
linacc_get    sets the linear speed
units_set    sets the current units metric or English
linspd_get   returns the maximum linear speed
linspd_set   sets the linear speed depending on the configuration

Category       Robot Configuration

link
Description   Makes a hard link to an existing file or directory. Useful for renaming files, moving files, or sharing data.
Syntax        command  link( var string[] name1, var string[] name2 )

Parameters

name1    the name of the object to create a new link to
name2    the name of the new link

Returns
>= 0       Success
-EINVAL   one of the file names was invalid
-ENOTDIR   a component of one of the names was not a directory
-ENOENT    the original object was not found
-EIO       an I/O error occurred
-EAGAIN    the system is temporarily out of the resources required to carry out this operation
-EISDIR    can't create a hard link to a directory
-EEXIST    name2 already exists
-EXDEV     tried to link across filesystems

Category       File and Device System Management

linklen_get
Description   Gets the link length for all axes.
Syntax        command  linklen_get( var float[8] length )

Parameter

length   an array of floats

Returns
Success >= 0
Failure < 0

Example
int machine, transform, actual, I
float[8] links

axes_get(machine, transform, actual)
linklen_get(links)
    for i = 1 to machine
        printf("axis {1} link length is ()\n", i,links[i])
    end for

Result
For a 255 robot:
axis 1 link length is 10.0000
axis 2 link length is 10.0000
axis 3 link length is 2.0000
axis 4 link length is 0.0000
axis 5 link length is 0.0000

See Also
linklen_set   sets the link length for an axis
**linklen_set**

Sets the link length for an axis.

**Description**

Sets the link length for an axis.

**Syntax**

```
command linklen_set( int axis, float length )
```

**Parameter**

- `axis`  
  an int
- `length`  
  a float

**Returns**

- **Success >= 0**
- **Failure < 0**

**See Also**

`linklen_get` gets the link lengths of all axes

**Category**

Robot Configuration

---

**linspd_get**

Returns the maximum linear speed for the robot in units of millimetres per second or inches per second depending on the unit configuration.

**Description**

Returns the maximum linear speed for the robot in units of millimetres per second or inches per second depending on the unit configuration.

Cannot be used in the speed() command which takes an integer parameter of percentage of maximum speed, for example

```
speed(<int>linspd_get(t))
```

**Syntax**

```
command linspd_get(var float linspd)
```

**Parameter**

- `linspd`  
  specifies the maximum speed value.

**Returns**

- **Success >= 0**
- **Failure < 0**

Returns negative error code if command fails.

**Example**

```c
float max_lin_spd
int curr_percent_spd
linspd_get(max_lin_spd)
speed_get(curr_percent_spd)
printf("The maximum linear speed is /n", max_lin_spd)
printf("The current speed setting is /n", curr_percent_spd)
```

**Result**

The maximum linear speed is
The current speed setting is

**See Also**

`linspd_set` sets the linear speed
`units_set` sets the units metric or English

**Category**

Robot Configuration

---

**linspd_set**

Sets the linear speed for the robot in units of millimetres per second or inches per second depending on the configuration.

**Description**

Sets the linear speed for the robot in units of millimetres per second or inches per second depending on the configuration.

**Syntax**

```
command linspd_set(var float linspd)
```

**Parameter**

- `linspd`  
  specifies the new speed setting

**Returns**

- **Success >= 0**
- **Failure < 0**

Returns -EINVAL if (linspd < 0) or other error if the command fails.
Example

```plaintext
;; Set the linear speed to the maximum speed
float spd
linspd_get(spd)
linspd_set(spd)
printf("The speed is {}", spd)
```

Result
Sets the linear robot speed to the maximum speed value.

See Also
- speed_get gets the current speed setting
- speed_set sets the speed of arm motions
- linspd_set sets the linear speed
- units_set sets the current units metric or English

Category
Robot Configuration

---

**ln**

Description
Calculates the natural logarithm of a float. Takes a positive argument.

Syntax
```plaintext
func float ln( float x )
```

Returns
The natural logarithm of the argument.

Example
```plaintext
float x = 7.5
float y
y = ln( x )
```

Result
2.014903

See Also
- log calculates the common (base 10) logarithm
- pow calculates a value raised to a power

Category
Math

---

**loc_cdata_get**

Description
Packs the cloc cl into the float array fa. The float[8] array corresponds to the cartesian coordinates x, y, z, yaw, pitch, roll, extra axis 1, extra axis 2; or x, y, z, pitch, roll, extra axis 1, extra axis 2, extra axis 3.

Syntax
```plaintext
sub loc_cdata_get( var cloc cl, var float[8] fa )
```

Parameters
- `cl` cartesian coordinate location variable
- `fa` an array of floats - packed with the location values of `cl`

Example
```plaintext
... teachable cloc cl
float[8] fa
loc_cdata_get(cl, fa)
...
```

See Also
- loc_cdata-set
- loc_pdata_get
- loc_pdata_set

Category
Location: Data Manipulation
**loc_cdata_set**

Description
Packs the cartesian data in \(fa\) into the cloc \(cl\). The float[8] array corresponds to the cartesian coordinates x, y, z, yaw, pitch, roll, extra axis 1, extra axis 2; or x, y, z, pitch, roll, extra axis 1, extra axis 2, extra axis 3.

Syntax
\[
\text{sub loc_cdata_set( var cloc } cl, \text{ var float[8] } fa )
\]

Parameter
- \(cl\) cartesian coordinate location variable packed with the data in \(fa\)
- \(fa\) an array of floats specifying the data for the cloc

Example
...  
cloc cl  
float[8] fa = {2,3,4,0,0,0,0,0}  
loc_cdata_set(cl, fa)  
...  

**loc_check**

Description
Tests the checksum of the generic location \(gl\). If the checksum is OK, returns 1.

Syntax
\[
\text{func int loc_check( var gloc } gl )
\]

Parameter
- \(gl\) generic location to be checked

Returns
- True (1) Success; the checksum is correct.
- False (0) Failure; the checksum is wrong.

Example
\[
gloc gl  
...  
if loc_check( gl ) == 1  
;; everything OK  
else  
;; everything NOT OK  
end if  
\]

**loc_class_get**

Description
Returns the location class of a generic location variable \(gl\). The different classes are loc_unknown, loc_cartesian, and loc_precision.

Syntax
\[
\text{func loc_class loc_class_get( var gloc } gl )
\]

Parameter
- \(gl\) gloc generic location variable

Returns
- loc_class, one of:
  - loc_unknown
  - loc_cartesian
  - loc_precision
**loc_class_set**

**Description**
Sets the class of a generic location variable *gl* to location class *lc*. The different classes are *loc_unknown*, *loc_cartesian*, and *loc_precision*.

**Syntax**
```plaintext
sub loc_class_set( var gloc gl, loc_class lc )
```

**Parameter**
- *gl*  gloc generic location variable
- *lc*  loc_class type: must be
  - *loc_unknown*
  - *loc_cartesian*
  - *loc_precision*

**Example**
```plaintext
gloc gl1, gl2
loc_class lc
...
lc = loc_class_get( gl1 )
loc_class_set( gl2, lc )
```

**Category**
Location Data: Manipulation

---

**loc_flags_get**

**Description**
Returns the flags that are set for the generic location variable *gl*. Warning: the flags are used to mark if the location has been taught and what units it is in. It is potentially dangerous to tamper with the flags of a location.

**Syntax**
```plaintext
func int loc_flags_get( var gloc gl )
```

**Parameter**
- *gl*  location variable (cloc or ploc)

**Returns**
an integer with the bits set according to the following:
- `LOC_INVALID` 0x00
- `LOC_VALID` 0x01
- `LOC_CALIBRATE` 0x02
- `LOC_MARKER` 0x04
- `LOC_NULL` 0x08
- `LOC_METRIC` 0x10
- `LOC_TOOL` 0x20
- `LOC_BASE` 0x40
- `LOC_OFFSET` 0x80

**Example**
```plaintext
int flags
gloc gl
...
```
flags = loc_flags_get( gl )
loc_flags_set( flags + 1 )

See Also
loc_flag_set

Category
Location: Flags

loc_flags_set

Description
Sets the flags on the generic location variable gl to f. Does not re-calculate the checksum.

Syntax
sub loc_flags_set( var gloc gl, int f )

Parameter
- gl: the location: a cloc or ploc
- f: an integer the flag constructed with the bits set according to the following defined constants

Example
int flags
gloc gl
...
flags = loc_flags_get( gl )
loc_flags_set(gl, flags + 1 )

See Also
loc_flags_get

Category
Location: Flags

loc_machtype_get

Description
Returns the machine type code of a generic location gl.

Syntax
func machine_type loc_machtype_get( var gloc gl )

Parameter
- gl: generic location variable

Returns
Success >= 0: Returns a machine_type enumerated type

Example
gloc gl
int mach_type
...
mach_type = loc_machtype_get( gl )

See Also
loc_machtype_set

Category
Location: Flags
**loc_machtype_set**

**Description**
Sets the machine type code of generic location variable *gl* to machine type *mt*. Does not re-calculate the checksum.

**Syntax**
```
sub loc_machtype_set( var gloc gl, machine_type mt )
```

**Parameter**
- *gl*: generic location variable*
- *mt*: machine_type. enumerated type one of:
  - mc_a255: A255
  - mc_a465: A465
  - mc_f2: F2        * see enum

**Example**
```
gloc gl1, gl2
int mt
...
mt = loc_machtype_get( gl1 )
loc_machtype_set( gl2, mt )
```

**See Also**
loc_machtype_get

**Category**
Location: Flags

---

**loc_pdata_get**

**Description**
Packs a gloc into an integer array. The int[8] array corresponds to the motor pulse values for the 8 motors, in order.

**Syntax**
```
sub loc_pdata_get( var ploc pl, var int[8] ia )
```

**Parameter**
- *pl*: ploc (precision location variable)
- *ia*: integer array packed with the motor pulse counts

**Example**
```
... teachable ploc pl
int[8] ia
loc_data_get(pl, ia)
...
```

**See Also**
loc_pdata_set
loc_cdata_get
loc_cdata_set

**Category**
Location: Data Manipulation

---

**loc_pdata_set**

**Description**
Packs the precision data in *ia* into the (should this be a ploc) gloc *pl*. The int[8] array corresponds to the motor pulse values for the 8 motors, in order.

**Syntax**
```
sub loc_pdata_set( var gloc pl, var int[8] ia )
```

**Parameter**
- *pl*: gloc (should this be a ploc) to be packed with the motor pulse counts in *ia*
- *ia*: integer array packed with the motor pulse counts

**Example**
```
... gloc gl
int[8] ia = {
loc_data_get(gl, ia)
...
```

**RAPL-II**
POINT
loc_re_check

Description
Recalculates and re-sets the checksum of a generic location gl.

Syntax
```
sub loc_re_check( var gloc gl )
```

Parameter
gl the location to be checked

Example
gloc gl
... loc_re_check( gl )

See Also
loc_check

Category
Location: Data Manipulation

lock

Description
Locks a specified axis.
Not to be confused with flock() which locks a file.

Syntax
```
command lock( int axis )
```

Parameter
axis the axis to be locked: an int

Returns
Success >= 0
Failure < 0

Example
int axis
... lock(axis)

RAPL-II
Same as LOCK

Category
Motion

log

Description
Calculates the common (base 10) logarithm of a float. Takes a positive argument.

Syntax
```
func float log( float x )
```

Returns
Success >= 0. The common logarithm of the argument.
Failure < 0

Example
float x = 7.5
float y
y = log( x )

Result
0.875061

RAPL-II
LOG

See Also
In calculates the natural logarithm
pow calculates a value raised to a power

Category
Math
MAJOR

Description
Extracts the major number from device dev.

Syntax
func int MAJOR( int dev )

Parameters
dev specifies the device - an int

Returns
Success >= 0
Failure < 0

Example
int dev, major = 23, minor = 1
... 
dev = BUILD_DEV( major, minor )
major = MAJOR( dev )
minor = MINOR( dev )

See Also
MINOR  extracts the minor number from a device

Category
File and Device System Management

malarm

Description
Requests that the system send the current process a specified signal after a specified delay. This can be used to implement timeouts and periodic events in a fairly simple fashion.

Syntax
command malarm(int delay, int sig)

Parameters
There are two required parameters:

delay How long to wait, in milliseconds, before sending signal sig
to the current process. If delay == 0, then we are canceling
a signal request. Note that each time we call malarm() for a
given sig, we reset the time remaining to delay.

sig The signal to send after delay milliseconds has passed.

Returns
>= 0 Success; returns the number of milliseconds that were left
until sig would have been sent. Returns 0 if no previous
signal was requested.

< 0 Failure.

Example1
;;;; This demonstrates an interrupt that will occur at about
;;;; once per second:
sub alarm_handler(int n)
malarm(1000, SIG20) ;; send a SIG20 after 1 second
printf("Beep\n")
end sub

main
signal(SIG20, alarm_handler, NULL) ;; set the signal handler
malarm(1000, SIG20) ;; start the periodic event going
loop
printf("Hello!\n") ;; loop forever, saying Hello
delay(500)
end loop
end main
Result1

The output will look something like this:

Hello!
Hello!
Beep
Hello!
Hello!
Beep
...

Example2

;; This demonstrates using a signal with malarm() to implement
;; a read with a timeout:
;;
;; sub alarm_handler(int n)
;; does’t actually need to do anything but catch the signal
end sub

main
int fd, t
string[32] s
...
open(fd, "\dev/sio1", O_RDWR, 0) ;; open sio1
...
;; read with timeout:
malarm(SIGALRM, 1000) ;; 1 second timeout
 t = reads(fd, s, 32) ;; read!
malarm(SIGALRM, 0) ;; cancel the signal
;; NOW if t is -EINTR, we timed out with no data read
;; if t > 0, we read that many characters
...
end main

See Also
signal(), kill(), sigsend()

Category
Signals

maxvel_get

Description
For one axis, gets maxvel, the maximum angular velocity of the motor, in revolutions per minute. The maxvel is set to ensure proper output by the encoder.

Syntax
func float maxvel_get ( int axis)

Parameter
axis the axis being inquired: an int

Returns
Success: >= 0 Returns the maximum motor velocity in RPM
Failure: < 0

Example
int ax3vel[8]
ax3vel[3] = getmaxvel(3)

See Also
maxvels_get gets the maximum velocities of all motors
maxvel_set sets the maximum velocity of one motor
maxvels_set sets the maximum velocities of all motors

Category
Robot Configuration

maxvel_set

Description
For one axis, sets maxvel, the maximum angular velocity of the motor in revolutions per minute. The maxvel is set to ensure proper output by the

encoder. If the velocity specified is greater than limits set in the robot kinematics the value is truncated to the set limits.

**Syntax**

command maxvel_set(int axis, float maxvel )

**Parameters**

- **axis** the axis being set: an int
- **maxvel** the maximum velocity: a float

**Returns**

- **Success:** >= 0
- **Failure:** < 0

**Example**

```c
;; Example to set maximum velocity for system axis
;; It would be simpler to use maxvels_set
int axis, count
float[8] vel_max {180, 180, 180, 171.089, 172.800, 172.089, 2368.57, 350.002}
for count = 1 to 8
    maxvel_set(count ,vel_max[count-1])
end for
```

**RAPL-II**

Similar to @XMAXVEL.

**See Also**

- maxvel_get gets the maximum velocity of one motor
- maxvels_set sets the maximum velocities of all motors
- maxvels_get gets the maximum velocities of all motors
- configaxis configures an axis including sets maxvel

**Category**

Robot Configuration

---

### maxvels_get

**Description**

For all axes, gets maxvels, the maximum angular velocities of the motors. Maxvels are set to ensure proper outputs by the encoders.

**Syntax**

command maxvels_get( var float[8] maxvel )

**Parameter**

- **maxvel** the maximum velocities in rpm: an array of floats

**Returns**

- **Success:** parameter is packed
- **Failure:** < 0

**Example**

```c
float[8] vel_max
...
maxvels_get(vel_max)
```

**See Also**

- maxvels_set sets the maximum velocities of all motors
- maxvel_get gets the maximum velocity of one motor
- maxvel_set sets the maximum velocity of one motor

**Category**

Robot Configuration

---

### maxvels_set

**Description**

For all axes, sets maxvels, the maximum angular velocities of the motors. Maxvels are set to ensure proper outputs by the encoders. If the velocity specified is greater than limits set in the robot kinematics the value is truncated to the set limits.

**Syntax**

command maxvels_set( var float[8] maxvel )

**Parameter**

- **maxvel** the maximum velocities in revolutions per minute: an array of floats

**Returns**

- **Success:** >= 0
- **Failure:** < 0
Example

```c
float[8] new_velocities = { 180, 180, 180, 171.089, 172.800, 171.089, 0, 0}
maxvels_set(new_velocities)
```

Result

The maximum velocities are set to the preset limits for the A465 robot arm. The extra axes are set to a zero velocity.

RAPL-II

Similar to @XMAXVEL.

See Also

- `maxvels_get` gets the maximum velocities of all motors
- `maxvel_set` sets the maximum velocity of one motor
- `maxvel_get` gets the maximum velocity of one motor
- `* configaxis` configures an axis including sets maxvel

Category

Robot Configuration

---

### mem_alloc

**Description**

Allocates an area of free memory of length `size`, sets `ptr` to point to the area, and initializes the area to zeros, i.e. 'clears' it. Also tries to allocate more heap space if required.

Along with `mem_free()`, the user can allocate and de-allocate space repeatedly.

**Syntax**

```c
command mem_alloc(var void@ ptr, int size )
```

**Parameters**

- `size` a number of words (4 byte units)

**Returns**

- Success `>= 0`
- Failure `< 0`

**Example**

```c
;; Define a new structure “element” and allocate memory to create a
;; define the new type
;;
typedef element struct
    int val
    element@ previous ;; pointer to struct of type element
    element@ next ;; pointer to struct of type element
end struct

element@ tmp_ptr = NULL ;; pointer used to create new element

;; create new element with pointer ‘tmp_ptr’
mem_alloc(tmp_ptr,sizeof(tmp_ptr@))
```

**RAPL-II**

ALLOC not only allocated memory but performed other tasks with its parameters.

**See Also**

- `mem_free` de-allocates an area of memory
- `heap_space` determines largest area before failure of malloc
- `heap_set`

**Category**

Memory

---

### mem_free

**Description**

Frees memory space. Returns an area of memory, previously allocated by `mem_alloc()`, to the pool of free space. Should never be used with space that has not previously been allocated by `mem_alloc()`, although freeing space with a null pointer is acceptable.
command mem_free( void@ ptr )

Returns
Success >= 0
Failure < 0

Example
;; de-allocate memory for list of elements (structure see mem_alloc)
  printf ("** Deleting list elements\n\n")
  while (head_ptr)
    tmp_ptr = head_ptr@.previous
    printf (" head_ptr addr:{}\n",head_ptr)
    printf (" tmp_ptr addr:{}\n",tmp_ptr)
    mem_free (head_ptr)
    head_ptr = tmp_ptr
  end while

RAPL-II
Different from the RAPL-II command FREE which displayed the status of memory.

See Also
mem_alloc allocates an area of memory and initializes it

Category
Memory

memcopy

Description
Copies a block of words of length len from src to dst.

Syntax
command memcopy( void @ dst, void @ src , int len )

Parameter
dst a pointer to the copy destination
src a pointer to the copy source
len the integer value of the length to be copied

Returns
Success >= 0
Failure < 0

Example
int[100] x
int[8] y
...
;; get elements 20 to 27 from x into y
...
memcopy(&y, &x[20]), sizeof(y) )

See Also
memset

Category
Memory

memset

Description
Sets a block of words of length len at dst to contain value v.

Syntax
command memset( void @ dst, int v, int len )

Parameter
dst pointer to the memory destination to be set
v an int value to be set
len the length of memory to be set to v

Returns
Success >= 0
Failure < 0

Example
int[100] x
teachable int new
...
:: Set elements of x all to value new
memset(&x, new, sizeof(x))

See Also
memcopy

Category
Memory

### memstat

**Description**
Gets information about the current system memory status.

**Syntax**
```
command memstat( int@ run_0, int@ run_1 )
```

**Parameters**
If `run_0` does not equal NULL, then `run_0` is assigned the length of the longest run of unallocated blocks. If `run_1` does not equal NULL, then `run_1` is assigned the length of the longest run of allocated blocks.

**Returns**
Success >= 0  Returns the number of free clicks.
Failure < 0

**Example**
```
int r0, r1, num_blocks
...
num_blocks = memstat( &r0, &r1 )
```

See Also
mem_alloc
heap_set
heap_size
heap_space

Category
Memory

### MINOR

**Description**
Extracts the minor number from device `dev`.

**Syntax**
```
func int MINOR( int dev )
```

**Returns**
Success >= 0
Failure < 0

**Example**
```
int dev, major = 23, minor = 1

dev = BUILD_DEV( major, minor )
major = MAJOR( dev )
minor = MINOR( dev )
```

See Also
MAJOR  extracts the major number from a device

Category
File and Device System Management

### mkdir

**Description**
Creates a new, empty directory specified by `path` with permissions defined by `mode`. The entries for dot and dot-dot are automatically created. A common mistake is to specify the same mode as for a file (read and write only), but for a directory normally one of the execute bits must be enabled to allow access to the filenames within the directory.

**Syntax**
```
command mkdir( var string[] path, int mode )
```

**Returns**
Success >= 0
Failure < 0
-EXIST  if dir already exists
-ENOENT  if the parent dir or a component of it doesn’t exist
-EINVAL  if the file name is invalid
-ENOTDIR  if a component of the path is not a directory
-ENOSPC out of space on the device
-EIO an I/O error occurred

Example

```c
string[] path = "/usr/name/new_dir"
int mode = M_READ|M_EXEC
...
mkdir ( path, mode )
```

System Shell

`mkdir`

See Also

`mknod` Makes special node (device, fifo, socket, directory)

Category

File and Device System Management

---

**mknod**

Description

Makes a special node.

Syntax

```c
command mknod(var string[] path, node_type vt, int mode, int dev)
```

Parameters

- **path**: path to the node location
- **vt**: the node to be made, of type node_type, one of:
  - NT_NON: no entry
  - NT_REG: regular file
  - NT_DIR: directory
  - NT_DEV: device
  - NT_LNK: symbolic link
  - NT_SOCK: inter-process communication socket
  - NT_FIFO: fifo
- **mode**: the modes of access, of type mode_flags, any combination of:
  - M_READ: read allowed
  - M_WRITE: write allowed
  - M_EXEC: executable *
- **dev**: the MAJOR and MINOR device numbers

Returns

- **Success** >= 0
- **Failure** < 0
  - EINVAL: if an invalid argument
  - EXIST: if it already exists
  - ENOENT: if the parent dir or a component of it doesn’t exist
  - ENOTDIR: if a component of the path is not a directory
  - ENOSPC: out of space on the device
  - EIO: an I/O error occurred

System Shell

Same as mkdev, mkfifo, mksock, mkdir.

See Also

`mkdir` makes a new directory

Category

File and Device System Management

---

**module_name_get**

Description

Gets the name of the module performing this subroutine call and places it into `name`, up to `maxlen` characters.

Allows a library to retrieve its own invocation name.

Allows multiple machine instances using only one library.

Syntax

```c
sub module_name_get(var string[] name, int maxlen )
```
motor

Description
Rotates a motor by a defined number of encoder pulses.

There is a third, optional parameter for a specific condition. Under most conditions, no specifier or 0 (zero) is used. If the third parameter is used, the system monitors for the specified state. Motion terminates when the input transitions to (or is in) this state or after the specified number of pulses (second parameter) have been counted, whichever is first. The third parameter is typically used when seeking for homing or limit switches during homing or calibrating operations.

Syntax
\[\text{command motor( int axis, int pulses [, int cond] )}\]

Parameters
\begin{itemize}
  \item \textit{axis} the axis being moved: an int
  \item \textit{pulses} the number of pulses to move: an int
\end{itemize}

Parameter (Optional)
\begin{itemize}
  \item \textit{cond} the condition: one of type \texttt{motor_stop_mode_t} or an int:
    \begin{itemize}
      \item \texttt{MSTOP_NONE} = 0 no specific condition
      \item \texttt{MSTOP_ONHOME} = 32000 stops when homing switch goes on
      \item \texttt{MSTOP_OFFHOME} = -32000 stops when homing switch goes off
      \item +1 stops when GPIO 1 is on
      \item -1 stops when GPIO 1 is off
    \end{itemize}
\end{itemize}

Returns
\begin{itemize}
  \item Success \(\geq 0\)
  \item Failure < 0
\end{itemize}

Example
\texttt{motor(3, 1000, 0)}

RAPL-II
Similar to MOTOR.

See Also
\begin{itemize}
  \item \texttt{joint} moves by joint degrees
  \item \texttt{jog} moves by cartesian increment
\end{itemize}

Category
Motion
Calibration

motor_to_joint

Description
Converts a location from motor pulses to joint angles. Used if a location of one type needs to be converted to another type for checking or other use within the program.
Syntax
command motor_to_joint( ploc motor, var float[8] joint )

Parameters
motor the location in motor pulses: a ploc
joint an array of floats is packed with the location in joint angles

Returns
Success >= 0
joint is packed
Failure < 0

Example
ploc motor1
float[8] joints1
motor_to_joint(motor1, joints1)

Result
joints1 is packed with the appropriate joint positions

RAPL-II
Similar to SET with different location types.

See Also
joint_to_motor converts joint angles to motor pulses
motor_to_world converts motor pulses to world coordinates

Category
Location: Kinematic Conversions

motor_to_world

Description
Converts a location from motor pulses to world coordinates. Used if a location of one type needs to be converted to another type for checking or other use within the program.

Syntax
command motor_to_world( ploc motor, var cloc world )

Parameters
motor the location in motor pulses: a ploc
world the location in world coordinates: a cloc

Returns
Success >= 0
world is packed
Failure < 0

Example
teachable ploc motor1
teachable cloc world1
motor_to_world(motor1, world1)

Result
world1 is packed with the appropriate world coordinate location values

RAPL-II
Similar to SET with different location types.

See Also
world_to_motor converts world coordinates to motor pulses
motor_to_joint converts motor pulses to joint angles

Category
Location: Kinematic Conversions

mount

Description
Mounts a filesystem of type \( t \) on directory \( dir \), with options \( flags \). Special filesystem-specific arguments are passed using the \( data \) pointer.

Syntax
command mount( mount_type \( t \), var string[] dir, \n    mount_flags flags, void* data )

Parameter
\( t \) the type of filesystem, of type mount_type, one of:
MOUNT_MFS Memory File System
MOUNT_CFS CROSnt File System
MOUNT_RFS Remote File System
MOUNT_HOSTFS Host File System

- **dir** the mount point of the CROS directory: a string of var length
- **flags** the option, of type mount_flags:
  - MOUNTF_RDONLY *
- **data** file-system specific arguments
  - (none; data = NULL) for MFS
  - char FAR* points to path of server socket for RFS
  - char FAR* points to host filesystem path for HOSTFS

**Returns**
- **Success >= 0**
- **Failure < 0**
  - -EPERM must be a privileged process to mount()
  - -EINVAL invalid argument
  - -ENOTDIR the mount point is not a directory
  - -ENOENT a component was not found
  - -EIO an I/O error occurred
  - -EAGAIN temporarily out of resources needed to do this
  - -EBUSY the mount point is busy

**Example**
```
#define PATHLEN 32
mount_type type = MOUNT_HOSTFS
string[PATHLEN] dir = "/app/this_app"
mount_flags flags = MOUNTF_RDONLY
c_statfs stat

int check

check = mount(type, dir, flags, NULL)
```

**System Shell**
- Same as mount

**RAPL-II**
- No equivalent.

**See Also**
- unmount unmounts a mounted file system

**Category**
- File and Device System Management

---

**move**

**Description**
Moves the tool centre-point to the specified location in joint-interpolated mode. Individual robot joints start and stop at the same time. The speed of the joint that has to move the farthest is governed by the speed setting, and other joints rotate slower according to joint interpolation. The resulting path is not straight.

The location can be either a cartesian location or a precision location.

**Syntax**
```
command move( gloc location )
```

**Parameter**
- **location** the destination location: a gloc (can be cloc or ploc)

**Returns**
- **Success >= 0**
- **Failure < 0**

**Example**
```
teachable ploc pick_1
teachable cloc place_1
move(pick_1)
...
move(place_1)
```

**RAPL-II**
- Similar to MOVE, without the S parameter.

**See Also**
- moves same as move(), but in straight line
- appro moves to an approach position
moves

Description
Moves the tool centre-point to the specified location in cartesian-interpolated mode. The result is straight-line motion. Individual robot joints start and stop at the same time.

The location can be either a cartesian location or a precision location.

Syntax
command moves( gloc location )

Parameter
location the destination location: a gloc

Returns
Success >= 0
Failure < 0

Example
teachable ploc pick_2
teachable cloc place_2
...
moves(pick_2)
...
moves(place_2)

RAPL-II
Similar to MOVE, with optional S (straight-line) parameter.

See Also
move same as moves(), but joint-interpolated
appro moves to an approach position
depart moves to a depart position
finish finishes current motion before another motion

Category
Motion

msleep

Description
Sleeps for the number of milliseconds specified in milliseconds and then returns to the main program. Can be terminated by an EINTR error. To avoid this, use delay().

Syntax
command msleep( int milliseconds )

Returns
Success >= 0
Failure < 0

EOK no error; timed out normally
EINTR if interrupted by a signal

Example
loop
    print ("Waiting for GPIO input 1. \n")
    if (input(1) == 1 )
        break
    end if
    msleep(250)
end loop

RAPL-II
Similar to DELAY.

See Also
delay sleeps without being terminated by EINTR

Category
System Process Control: Single & Multiple Processes
**mtime**

**Description**
Obtains the number of milliseconds since system start-up.

The data type, `c_mtime_t` is an array of ints, `int[2]`, a 64-bit number, like an unsigned long in C. In the array, `[0]` holds the least significant bit and `[1]` holds the most significant bit. There is space for approximately 584,942,417.4 years, after which the bits "roll over" to zero.

**Syntax**
```
command mtime( c_mtime_t@ ctp )
```

**Parameter**
- `ctp` the number, of type `c_mtime_t`: an `int[2]`

**Returns**
- Success `>= 0`
- Failure `< 0`

**Example**
```
;; print the elapsed time of a delay determined by a random number
;; the time is limited to 65 seconds since only the first element
;; of the mtime array is used

main
  int num_rndm
  int[2] start_tm, end_tm
  srand (10)
  num_rndm = rand_in (1000,65000) ;; limit range of random number
  printf ("random number = {}
",num_rndm)
  mtime(&start_tm) ;; get start time
  delay (num_rndm)
  mtime(&end_tm) ;; get end time
  printf ("time elapsed = {} milliseconds
",end_tm[0]-start_tm[0])
end main
```

**Category**
Date and Time

---

**net_in_get**

**Description**
Reads input data from the F3 end of arm I/O boards.

**Syntax**
```
func int net_in_get(int in)
```

**Parameter**
- `in` the number of the input to be read (1..32)

**Returns**
- Success: 0 -> input off, 1 -> input on
- Failure: `net_in_get()` raises an exception

**Example**
```
;; Read input 3 from the end of arm I/O board:
if (net_in_get(3))
  ;; the output is set...
end if
```

**See Also**
- `net_ins_get()`, `net Outs_get()`, `net out_set()`, `net outs_set()`

**Category**
Digital Input and Output
**net_ins_get**

Description
Reads all input data from the F3 end of arm I/O boards.

Syntax
```c
func int net_ins_get(int mask)
```

Parameter
`mask` bit mask with a "1" for each input whose value is to be read. The least significant bit represents channel 1, the most significant bit represents channel 32.

Returns
Success: an integer with a "1" in each bit corresponding to each input that is on.
Failure: net_ins_get() raises an exception.

Example
```c
int t;
;; Check the status of input 1 through 8:
t = net_ins_get(0x000000ff) ;; bottom 8 bits set
printf("Inputs 1 to 8 are: \x\n", t)
```

See Also
`net_in_get(), net_outs_get(), net_out_set(), net_outs_set()`

Category
Digital Input and Output

**net_out_set**

Description
Sets a specified F3 end of arm output to a specified value.

Syntax
```c
command net_out_set(int outnum, int value)
```

Parameters
`outnum` -- end of arm output to change (1..4)
`value` -- 0 => off, 1 => on

Warning
if the F3 is configured for an air gripper, then end of arm outputs 1 and 2 are reserved, and must not be used.

Returns
Success >= 0
Failure < 0 (-ve error code)

Example
```c
int t;
;; read input 3 and output the opposite of its value to output 3:
t = net_in_get(3)
if (t < 0)
   ;; error...
end if
net_out_set(3, !t)
```

See Also
`net_in_get(), net_ins_get(), net_outs_get(), net_outs_set()`

Category
Digital Input and Output

**net_outs_get**

Description
Gets the current state of a set of F3 end of arm outputs.

Syntax
```c
func int net_outs_get(int mask)
```

Parameters
`mask` indicates which outputs to read; the least significant bit corresponds to output 1, the most significant bit corresponds to output 32. F3 currently only supports 4 outputs

Returns
Success: an integer with a "1" in each bit corresponding to each output that is on.
Failure: net_outs_get() raises an exception
Example

;; Flip the state of outputs 1 through 4:
  t = net_outs_get(0x0000000f)  ;; get the old values
  ;; now set the new values, using “xor” to flip the bits:
  net_outs_set(t xor 0x0000000f, 0x0000000f)

See Also

net_in_get(), net_ins_get(), net_out_set(), net_outs_set()

Category

Digital Input and Output

---

**net_outs_set**

**Description**

Allows several F3 end of arm outputs to be set to a specified state at the same time.

**Syntax**

command net_outs_set(int state, int mask)

**Parameters**

- **state** -- each bit represents what state to set an output to
- **mask** -- each “1” corresponds to each output to change.

Both “state” and “mask” are sets of bits corresponding to outputs. The least significant bits correspond to output 1; the most significant bits correspond to output 32. When the net_outs_set() command is executed, each output with a corresponding 1 in mask will be set to the value of the corresponding bit in state.

**Returns**

- Success >= 0
- Failure < 0 (-ve error code)

**Example**

see the example for net_outs_set(), above.

**See Also**

net_in_get(), net_ins_get(), net_out_set(), net_outs_get()

**Category**

Digital Input and Output

---

**nolimp**

**Description**

Re-engages the servo control of a motor which unlimps that joint. A single axis or several axes can be specified. All axes are specified by an empty parameter.

Used after the command limp().

**Syntax**

command nolimp( [ int axis ] [ , int axis ] ... )

**Parameter (Optional)**

- **axis** axis being unlimped
- (empty) all axes unlimped

**Returns**

- Success >= 0
- Failure < 0

**Example**

limp(4, 5, 6) ;; limps axes 4, 5, and 6
... nolimp(4, 5, 6) ;; unlimps axes 4, 5, and 6

**Application Shell**

nolimp

**RAPL-II**

Similar to NOLIMP.

**See Also**

limp limps axes

**Category**

Motion

---

**obs_get**

**Description**

Gets point of observation.
Syntax       command obs_get()
Returns      Success >= 0
Failure < 0.  Will fail only due to communications.
Example      obs_get()
RAPL-II      There is no corresponding construct.
See Also     obs_rel releases point of observation
Category     System Process Control: Points of Control and Observation

**obs_rel**

Description  Releasing point of observation.
Syntax       command obs_rel()
Returns      Success >= 0
Failure < 0.  Will fail only due to communications.
Example      obs_rel()
RAPL-II      There is no corresponding construct.
See Also     obs_get gets point of observation
Category     System Process Control: Points of Control and Observation

**onbutton**

Description  Waits for a button specified by \( b \) to be pressed. If the argument \( \text{blink} \) is True, the corresponding light blinks until the button is pushed. After execution the light is returned to the state it was in before the command call. The command utilizes the panel_button_wait subprogram.
Syntax       command onbutton(int \( b \), int \( \text{blink} \))
Parameter     \( b \) specifies the button to be pressed button_enum type one of
\[
\begin{align*}
\text{B_F1} & = 1 \\
\text{B_F2} & = 2 \\
\text{B_PAUSE_CONT} & = 4 \\
\text{B_HOME} & = 8
\end{align*}
\]
\( \text{blink} \) TRUE to blink the light while waiting, otherwise FALSE
Returns      Success >= 0
Failure < 0      Returns an error.
Example      ;;Program to demonstrate Panel Button subroutines.
;;Move the robot to a position aa when the F1 button is pressed
;;While the robot is moving turn on the F1 light. Set status
;;window AA after move. Then, after F2 is pressed it moves to
;;second position, turns on the F2 light, sets the status window
;;to BB
main
  teachable clock aa, bb
  panel_lights_set(0xf,0x0) ;; turn off the panel lights
  online(ON)
  ;;Wait for button F1 to be pushed before moving to location AA
  printf("Press F1 to move robot to AA/n")
  loop
    if(onbutton(B_F1, ON))
      panel_light_set(B_F1, ON)
      move(aa)
      break
    else
      delay(250)
      continue
end if
end loop

;; Finish move to location aa, Set AA in status window
finish()
panel_status(OxAA)
panel_light_set(B_F1, OFF) ;; turn off the F1 light
;; Wait for button F2 to be pushed before moving to location bb no time out
printf("Press F2 to move to BB/n")
loop
  if (panel_button_wait(B_F2, -1))
    panel_light_set(B_F2, ON)
    move(bb)
    break
  else
    delay(250)
    continue
  end if
end loop
finish() ;; Set Status to BB when robot is in location BB
panel_status(0xBB)
panel_lights_set(0xff, 0x00) ;; Turn off lights

See Also
panel_button_wait
panel_button_set

Category
Front Panel

---

## online

**Description**

Sets the online mode to one of the values: OFF, ON, WAIT, PROCEED, TRACK, NOTRACK.

With OFF, there is only space in the queue for one motion command. The command is taken from the queue to be processed, and must be taken out for the next command to be put in. In effect, flow proceeds in a manner similar to having a `finish()` command after each motion command.

With ON, there is space in the queue for 8 motion commands.

With WAIT, the queue fills up with motion commands. Commands are calculated while execution of the motion waits. Execution begins when the queue is full or PROCEED is encountered.

With PROCEED, the motions are executed. The robot moves through the locations without stopping at each location.

**Syntax**

```
command online( int online_flag )
```

**Parameters**

- `online_flag`
  - OFF
  - ON
  - WAIT
  - PROCEED
  - ENA_TRACK
  - DIS_TRACK

**Returns**

- Success >= 0
- Failure < 0

**Example**

```c
online(ON) ;; turn mode on
online(WAIT) ;; wait while queue fills
move(a) ;;
```
move(b) ;; fill queue with these motions
move(c)
move(d)
online(PROCEED) ;; flush motion queue

RAPL-II
Similar to ONLINE.

See Also
finish finishes current arm motion command before next arm motion
gripfinish finishes current gripper motion command before next gripper motion
robotisdone gets the robot done state for non-control processes

Category
Motion
Robot Configuration

---

**open**

**Description**
Opens an object in the file system, a file or device specified in *name*, with access mode given in *flags*. At successful completion (a positive value), the command returns the file descriptor *fd*, which is used to access the file throughout the program. If there is a problem, the command returns a negative error code.

O_RDONLY is the default mode. O_TEXT allows CROS to create DOS compatible text file, i.e., with CR-LF line terminations instead of CROS’ LF-only line terminations. O_TEXT does not affect sockets.

An open() command with O_CREAT and O_EXCL on a file that already exist returns an error, -EEXIST. This allows standard file locking to work.

**Syntax**
```
command open( var int fd, var string[] name, o_flags flags, int mode )
```

**Parameters**
- *fd* the file descriptor: an integer
- *name* the file to be opened: a variable length string
- *flags* flags, of type o_flags, one or more of:
  - with files
    - O_RDONLY read only
    - O_WRONLY write only
    - O_RDWR read and write
    - O_NONBLOCK non-blocking mode
    - O_APPEND always append to EOF on writing
    - O_BINARY binary mode; all writes of ‘\n’ get converted to line feed
    - O_TEXT text mode; all writes of ‘\n’ get converted to carriage return and line feed ‘\r\n’
    - O_CREAT create file if it doesn’t exist
    - O_TRUNC truncate file to 0 bytes
    - O_EXCL give error if file already exists
  - with sockets
    - O_SERVER server
    - O_CLIENT client

The two flags, O_CLIENT and O_SERVER, can only be used for sockets and they are mutually exclusive.

The other flags can only be used for files and can all be used together.

Examples:
- O_RDONLY | O_NONBLOCK read only, non-blocking reads
- O_CREAT | O_TRUNC | O_RDWR create a new file (or truncate an old one)
and open for reading and writing

O_APPEND | O_CREAT | O_WRONLY append to an existing file, or create a new file if one doesn’t exist, and write it
O_RDWR is the same as O_RDONLY | O_WRONLY

With any value for flags other than one including O_CREAT, opening a non-existent file is an error.

If flags contains O_CREAT, then the file is created if it doesn’t exist and is given permissions specified in mode.

mode access mode, of type mode_flags, one or more of:

M_READ readable
M_WRITE writeable
M_EXEC executable

The modes limit the ways in which programs opening the file can access it. For example, if mode is only M_READ, a program can read the file, but cannot write to it. Modes may be combined with the bitwise OR operator, represented by | (a single vertical bar/pipe), to form any desired combination.

M_READ
M_READ | M_EXEC
M_READ | M_WRITE
M_READ | M_WRITE | M_EXEC

Returns

>= 0 Success
-EAGAIN The system does not presently have the resources needed to carry out this operation. For example, there may be too many files open.
-EINVAL The flags are inconsistent or the name is invalid.
-EEXIST Tried to open a file with O_EXCL | O_CREAT, and the file already existed.
-ENOENT Some component of the path did not exist, or we are not O_CREATing and the file did not exist.
-EISDIR Tried to open a directory for writing.
-ENXIO Tried to open an unsupported device.
-ETXTBSY Tried to open an executing program for writing.
-ENOTDIR A component of the path to the file was not a directory.
-EIO An I/O error occurred
-EBUSY Tried to open a socket as server, but a server had already opened the socket. There can be at most 1 server.
-ENOSERV Tried to open a socket as client, but no server was present.

Example

int fd
...
open ( fd, "filename.txt", O_RDONLY, 0 )

See Also

close closes the file or device
chmod change the mode
write writes to the file
read reads from the file
send sends to the socket
recv receives from the socket
chmod change the mode

Category

File and Device System Management
Device Input and Output
opennp

open named pipe

Description
Opens a named pipe in the Windows NT domain.
Servers must specify a pipe on the local machine.
The maximum number of named pipes that can be open at one time is 9.

Syntax
command opennp( var int fd, string[] pipename, o_flags flags, int mode, var int signal )

Parameters
- **fd** the file descriptor: an int
- **pipename** the pipe name: a string of maximum length [128]
- **flags** flags, of type o_flags, one or more of:
  - O_RDONLY read only
  - O_WRONLY write only
  - O_RDWR read and write
  - O_SERVER open as server
  - O_CLIENT open as client
- **mode** access modes specific to named pipes, one or more of:
  - M_READ_MESSAGE readable
  - M_WRITE_MESSAGE writable
- **signal** the signal to send when overlapped i/o is complete: an int

Returns
Success >= 0
Failure < 0

Example
opennp( pd, //./pipe/pipe_on_this_machine, O_SERVER|O_RDWR,
      M_READ_MESSAGE|M_WRITE_MESSAGE, 13 )
opennp( NT_app_pipe, //lab/pipe/app2_pipe, O_SERVER|O_RDWR,
      M_READ_MESSAGE|M_WRITE_MESSAGE, 22 )

RAPL-II
No equivalent.

See Also
closenp closes a named pipe
connectnp connects to a named pipe
disconnectnp disconnects a client from a named pipe
statusnp checks the status of a named pipe

Category
Win 32

output

output_set

Description
Sets the single specified output channel to the specified state. The Boolean parameter bypass is optional. If set TRUE the execution of the output command bypasses the online motion queue.

Syntax
command output( int channel, int state [, boolean bypass] )

Parameters
- **channel** the GPIO channel: an int. Channels 1 to 16 correspond to actual GPIO output points; channels 17 to 24 are “virtual outputs” that act exactly like real outputs but do not connect to a physical signal. By watching virtual outputs, a process can synchronize itself to the motion queue.
**output_set**

Description

Sets the entire bank of output channels to states.

Syntax

```
func output_set( int channel )
```

Parameters

There is one parameter:

- `channel` : an int. Channels 1 to 16 correspond to actual GPIO output points; channels 17 to 24 are “virtual outputs” that act exactly like real outputs but do not connect to a physical signal. By watching virtual outputs, a process can synchronize itself to the motion queue.

Returns

Success >= 0

Failure < 0

Example

```
int state
int channel
...
state = output_get(channel)
```

Result

state = 1 if output is on, state = 0 if output is off

RAPL-II

No equivalent.

See Also

- output sets an output channel to a state
- output_pulse sets and reverses an output for its state
output_pulse

Sets the specified output channel to the specified state, waits 50 milliseconds and then sets the channel to the opposite state. The Boolean parameter bypass is optional. If set TRUE the execution of the output command bypasses the online motion queue.

Outputs can be pulsed on or pulsed off.

If the initial state of the output is different from the first state of this command, the output is set to that first state and then set to the opposite (the output’s initial) state. If the initial state of the output is the same as the first state of this command, the setting of the first state makes no change and the output is then set to the opposite state.

**Syntax**

```c
command output_pulse( int channel, int state[, boolean bypass])
```

**Parameters**

- `channel`: the GPIO channel: an int
- `state`: the state: an int, one of:
  - 0   off
  - 1   on
- `bypass`: boolean either
  - TRUE (1) execution bypasses the online queue
  - FALSE (0) default option - output execution is queued

**Returns**

- Success >= 0
- Failure < 0

**Example**

```c
int stateint channel...
...state = output_pulse(channel, state, 1)
```

**Result**

- output defined by int channel is pulsed, the command is not queued

**See Also**

- output_set
- outputs
- output_get
- input

**Category**

Digital Input and Output
**outputs/outputs_set**

### Description
Sets the entire bank of output channels to the specified states with a bitmask. The Boolean parameter bypass is optional. If set TRUE the execution of the output command bypasses the online motion queue.

### Syntax
```plaintext
command outputs(int fieldstate, int mask[, boolean bypass])
```

### Parameters
- **fieldstate**: a bit mapped state of the outputs
- **mask**: the output state of each bit will only be updated by the \textit{“new_val”} if the corresponding mask bit is high.
- **bypass**: True (1) -> execution bypasses the online queue and is not synchronized to robot motion
  False (0) -> output execution is queued in the motion queue. This is the default if this argument is omitted.

### Returns
- Success $\geq 0$
- Failure $< 0$

### Example
```plaintext
int mask = 0xFFFF ;; bit mask all 1’s
int state = 0...
outputs(state, mask, 0)
```

### Result
All outputs are set low, the command is queued in the online motion queue

### RAPL-II
No equivalent.

### See Also
- **output**: sets an output channel to a state
- **outputs_get**: queries the entire bank of output channels for their states
- **inputs**: queries the entire bank of input channels for their states

### Category
Digital Input and Output

---

**outputs_get**

### Description
Gets the current state of all the output channels.

### Syntax
```plaintext
func outputs_get()
```

### Parameters
- None

### Returns
- Success $\geq 0$
  - the state: an int, which is a bit map of the channel output states:
    - 0 = off
    - 1 = on
- Failure $< 0$

### Example
```plaintext
int state ;; present outputs
int state2 ;; desired outputs
int channel = 0xffff ;; selects all outputs (1111111111111111)

state = outputs_get();
if state == state2 ;; what is wanted
  else ;; set outputs to the state specified in state2
    outputs_set(channel, state2)
end if
```

### Result
Set outputs to the state specified in state2

### RAPL-II
No equivalent.
outputs_set

Alias
outputs

Syntax
command outputs_set( int fieldstate, int mask[, boolean bypass] )

Category
Digital Input and Output

panel_button

Description
Determines the status of the button specified by argument b. The return will be 0, unless the button is pressed. While the button is pressed the returned value is TRUE.

Syntax
func int panel_button(button_enum b)

Parameter
b button_enum type - one of:
B_F1 = 1
B_F2 = 2
B_PAUSE_CONT = 4
B_HOME = 8

Returns
Success >= 0 Returns TRUE if the button specified is pressed.
Failure < 0 Error descriptor

Example
printf("Press F1 to move the robot")
loop
    t=panel_button(B_F1)
    if t
        move(position)
        break
    else
        delay(250)
        continue
    end if
end loop

Refer also to the onbutton command description for further example of the panel button subprograms.

See Also
panel_buttons
on_button
panel_button_wait

Category
Front Panel

panel_button_wait

Description
Command waits for a particular button to be pressed. If the time specified by the timeout (seconds) argument is exceed an error descriptor is returned.

Syntax
command panel_button_wait(button_enum b, int timeout)
Parameter  

*b*  button_enum type one of:

- B_F1  = 1
- B_F2  = 2
- B_PAUSE_CONT = 4
- B_HOME = 8

*timeout*  waiting time in seconds, -1 (TM_FOREVER) means no time limit

Returns  

Success  >= 0
Failure  < 0   ETIMEOUT if waiting time is exceed

Example  

```c
;;Wait for button F2 to be pressed then move
loop
if(panel_button_wait(B_F2, -1))
  panel_light_set(B_F2, ON)
  move(bb)
  break
else
  delay(250)
  continue
end if
end loop
```

Refer to the onbutton command description for an example of the panel button subprograms

See Also  

onbutton  
panel_button  
panel_buttons

Category  

Front Panel

---

**Panel_buttons**

Description  

Gets the status of the panel buttons. The status is returned as a bit vector. The bits which are high (1) indicate which buttons are pressed. The value returned is zero if no buttons are pressed. If the value 3 (0...0011) is returned then panel buttons F1 and F2 are pressed.

Syntax  

```c
func int panel_buttons()
```

Returns  

Success  >= 0  Returns an integer high bits indicate which buttons were pressed.
Failure  < 0   Returns an error descriptor

Example  

```c
printf("Press F1 and F2 to move the robot)
loop
  t=panel_buttons()
  if t ==3  ;;F1 and F2 must be pressed together
      move(position)
      break
  else
      delay(250)
      continue
  end if
end loop
```

Also refer to the onbutton command description for further example of the panel button subprograms

Result  

When buttons F1 and F2 are both pressed at the same time the robot will move.

See Also  

panel_buttons  
on_button  
panel_button_wait
**panel_light_get**

**Description**
The function returns the status of the front panel light specified. Returns TRUE if the light is on FALSE if it is off.

**Syntax**
```c
func int panel_light_get(button_enum b)
```

**Parameter**
- `b` Specifies the light to check, `button_enum` type one of:
  - `B_F1` = 1
  - `B_F2` = 2
  - `B_PAUSE_CONT` = 4
  - `B_HOME` = 8

**Returns**
- Success >= 0 Returns ON if the light specified if the light is on.
- Failure < Error descriptor

**Example**
```c
int light_stat
...
;;Get status of the HOME light
light_stat = panel_light_get(B_HOME)
```

Refer to the onbutton command description for an example of the panel button subprograms.

**See Also**
- `panel_lights_get`
- `panel_light_set`
- `panel_lights_set`

**Category**
Front Panel

---

**panel_light_set**

**Description**
The command causes the light specified with the `button_enum` type to be set to the status specified by the int `on`. Use this command to link light status to conditions in robot applications.

**Syntax**
```c
command panel_light_set(button_enum b, int on)
```

**Parameter**
- `button` Refer to the Front Panel section for the `button_enum` definitions
- `on` If ON (ON = 1) turns light on, if OFF (OFF = 0) sets light off

**Returns**
- Success >= 0
- Failure < 0

**Example**
```c
panel_light_set(B_F1,OFF) ;; turn off the F1 light
```

Refer to the onbutton command description for an example of the front panel subprograms.

**See Also**
- `panel_light_get`
- `panel_lights_get`
- `panel_lights_set`

**Category**
Front Panel
panel_lights_get

Description
Returns the status of the four panel lights in bit vector format. If the light is ON the corresponding bit in the return integer is high. For example if the return value is 10 (0..01010), the F2 and HOME lights are ON.

Syntax
func int panel_lights_get()

Returns
Success >= 0    An integer with high bits corresponding to the ON lights.
Failure < 0    error descriptor

Example
  t=panel_lights_get() ;; returns the lights that are on
  if t
    ;; at least on light is ON
    panel_lights_set(0xff, 0x00) ;; turn lights off
  end if

Also refer to the onbutton command description for a further example of the front panel subprograms.

See Also
panel_light_get
panel_light_set

Category
Front Panel

panel_lights_set

Description
Set the panel lights selected by the argument mask to the corresponding values as specified by the argument value.

Syntax
command panel_lights_set(int mask, int value)

Parameter
mask    integer used for selecting the lamps. For each high bit (1) the corresponding light is selected. For example mask = 9 (0...01001) the F1 and Home lights are selected.
value    Specifies the values for the selected lights. For example 0 sets all the selected lights to OFF, 9 sets the F1 and HOME lights to ON.

Returns
Success >= 0
Failure < 0    Returns an error descriptor

Example
panel_status(0xBB)
panel_lights_set(0xff, 0x00) ;; Turn off lights

Refer to the onbutton command description for an example of the front panel subprograms.

See Also
panel_lights_get
panel_light_get
panel_light_set

Category
Front Panel

panel_status

Description
Sets the front panel status window to display the argument value. Note the command is intended to test the function of the window. Changing the display does not change the actual system status.

Syntax
command panel_status(int value)
Parameter

\textit{value} \par
the value to be displayed in the status window. The window can display 2 hexadecimal integers, therefore only the 8 LS bits are meaningful in the argument value.

Returns

\begin{itemize}
  \item \textbf{Success} $\geq 0$
  \item \textbf{Failure} $< 0$
\end{itemize}

Example

\begin{verbatim}
int i
for i=0 to 255
  delay(100) ;;;short delay
  panel_status(i) ;;;display window combinations in sequence
end for
\end{verbatim}

Also refer to the \texttt{onbutton} command description for an example of the front panel subprograms.

Category

Front Panel

---

\textbf{pdp\_get}

Description

The function gets the private data area pointer for the current thread.

Syntax

\texttt{func \textit{void}@ pdp\_get()}

Parameters

\begin{itemize}
  \item \textit{no parameters}
\end{itemize}

Returns

\begin{itemize}
  \item \textbf{Success} $\geq 0$ \quad \text{Returns void pointer to the data area for current thread.}
  \item \textbf{Failure} $< 0$
\end{itemize}

Example

\begin{verbatim}
\texttt{void}@ \textit{ptr}
if !(\textit{ptr}=\textit{pdp\_get()})
  ;;;error in function call
else
  ;;;program commands
end if
\end{verbatim}

Category

Memory

---

\textbf{pdp\_set}

Description

A subroutine to set the private area memory for the current thread

Syntax

\texttt{sub pdp\_set(\textit{void}@ \textit{ptr})}

Parameters

\begin{itemize}
  \item \textit{ptr} is a \textit{void ptr} which points to the private data area for the current thread.
\end{itemize}

Returns

\textit{subroutines do not return a value}

Example

\begin{verbatim}
\texttt{void}@ \textit{ptr}
pdp\_set(\textit{ptr})
\end{verbatim}

Category

Memory

---

\textbf{pendant\_bell}

Description

The serial teach pendant has a small speaker that may be used to signal events. There are three sounds which can be sent to the speaker. The sound is specified by the type \texttt{pendant\_bell\_t} argument passed in the command call with.

Library

\texttt{stp}

Syntax

\texttt{export command pendant\_bell(pendant\_bell\_t \textit{bell\_type})}
Parameter

The pendant_bell_t bell_type has the following definition:

```c
typedef pendant_bell_t enum
    pendant_bell_short = 1,
    pendant_bell_long,
    pendant_bell_alert ;; stuttering beep
end enum
```

Returns

Success >= 0
Failure < 0

Example

```c
stp:pendant_bell(pendant_bell_alert)
```

RAPL-II

Same as PRINTF 0,"\e[0q or \e[1q or \e[2q or \e[3q"

Category

Pendant

---

**pendant_chr_get**

Description

Reads a character from the pendant. This command does not wait until a return is entered and thus yields a null string if data is not ready.

Library

`stp`

Syntax

```c
export command pendant_chr_get(var string[] buffer)
```

Parameter

- `buffer` the character is stored in the buffer string

Returns

- Success >= 0 buffer is packed with character
- Failure < 0

Example

```c
stp:pendant_chr_get(answer)
```

Result

Reads character at teach pendant

RAPL-II

Same as INPUT <string_number(&1-4),<Device_zero(0)>

Category

Pendant

---

**pendant_close**

Description

Close the pendant in preparation for shutting down a program or the controller. The command disables the liveman switch.

Library

`stp`

Syntax

```c
export command pendant_close()
```

Parameter

None

Returns

- Success >= 0
- Failure < 0

Example

```c
stp:pendant_close()
```

RAPL-II

Same as PENDANT OFF

See Also

`shutdown`

Category

Pendant
**pendant_cursor_pos_set**

Description
Move the cursor to the position specified by the row and column arguments. If the position specified is not a valid position an error is returned. The pendant screen has 4 rows and 18 columns.

Library
stp

Syntax
```
export command pendant_cursor_pos_set(int row, int column)
```

Parameter
- row 1-4 are valid rows
- column 1-18 are valid columns

Returns
- Success >= 0
- Failure < 0

Example
```
... stp:pendant_cursor_pos_set(4,1) ;;set the cursor to the
;;;;bottom row first column
... RAPL-II
```

See Also
pendant_home
pendant_home_clear

Category
Pendant

**pendant_cursor_set**

Description
Enables or disables the pendant cursor, depending on the argument passed. A disabled cursor is not visible on the pendant screen. The enabled cursors, default setting, causes the cursor to blink on the screen.

Library
stp

Syntax
```
export command pendant_cursor_set(Boolean new_cursor)
```

Parameter
- new_cursor 1 enabled
- new_cursor 0 disabled

Returns
- Success >= 0
- Failure < 0

Example
```
... pendant_cursor_set( 1 )
... RAPL-II
```

Category
Pendant

**pendant_flush**

Description
Flushes any 'junk' characters in the incoming buffer.

Library
stp

Syntax
```
export command pendant_flush()
```

Parameter
None

Returns
- Success >= 0
- Failure < 0
Example

... 
stp:pendant_flush()
stp:pendant_close()
...

Result
Flushes

See Also
pendant_chr_get
pendant_close

Category
Pendant

---

**pendant_home**

Description
Moves the pendant cursor to the top left side of the pendant screen, row 1, column 1. The home position.

Library
stp

Syntax
`export command pendant_home()`

Parameter
None

Returns
Success >= 0
Failure < 0

Example

... 
stp:pendant_home()
...

Category
Pendant

---

**pendant_home_clear**

Description
Moves the pendant screen cursor to the home position and clears the screen.

Library
stp

Syntax
`command pendant_home_clear()`

Parameter
None

Returns
Success >= 0
Failure < 0

Example

... 
stp:pendant_home_clear()
...

RAPL-II
Same as PRINTF 0,"\e[1;1f\e[1s"

See Also
pendant_home

Category
Pendant

---

**pendant_open**

Description
Prepare the pendant for access and initialize it to defaults.

Library
stp

Syntax
`command pendant_open()`

Parameter
None
### pendant_write

**Description**
Writes a string to the pendant. The string can include standard ANSI escape codes to format the display on the screen. The pendant_write command calls the writes command from the File and Device Input and Output category.

**Library**
stp

**Syntax**
```
stp:export command pendant_write(var string[] buffer)
```

**Parameter**
- **buffer** the text to be displayed on the pendant screen

**Returns**
- Success >= 0
- Failure < 0

**Example**
```
... 
pendant_write("...")
... 
```

**RAPL-II**
Same as PRINTF Device_0," Text"

**See Also**
writes

**Category**
Pendant

---

### pipe

**Description**
Creates a single stream pipe between two file descriptors. In a pipe, data can flow only in one direction. Calling pipe() creates a file descriptor `rd_fd` that is mode RD_ONLY and another file descriptor `wr_fd` that is mode WR_ONLY. Closing the write end is the only way of sending an EOF indication to the read end. Also, writing to the write end of a pipe whose read end is closed results in a SIGPIPE being sent to the writer.

Generally, pipe() is called prior to a split, and then the pipe is used between parent and child communication. The parent then closes either the write or the read descriptor, depending on the direction of flow wanted, and the child closes the remaining descriptor.

**Syntax**
```
command pipe( var int rd_fd, var int wr_fd )
```

**Parameter**
- **rd_fd** an int- file descriptor for the read end of the pipe
- **wr_fd** an int- file descriptor for the write end of the pipe

**Returns**
- >= 0 Success
- EINVAL the arguments were invalid
- EAGAIN The system does not have sufficient resources to carry out this operation at this time.

**Example**
```
main
    int ps_id,i,status
```
```c
int fd_pipe_rd, fd_pipe_wr
pipe (fd_pipe_rd, fd_pipe_wr) ;; pipe file is opened in
     ;; blocking mode for reads
ps_id = split()
if ps_id == 0
    close (fd_pipe_wr) ;; child will read
             ;; data
    for i = 1 to 5
        read (fd_pipe_rd, &i, 1) ;; if data is not available
             ;; the read will be blocked
        printf ("\nchild read - ({}), i)
    end for
    close (fd_pipe_rd)
else
    close (fd_pipe_rd) ;; parent will write
             ;; data
    for i=1 to 5
        write (fd_pipe_wr, &i, 1)
        delay (500)
    end for
    close (fd_pipe_wr)
    waitpid (ps_id, &status, 0) ;; wait for child to
             ;; complete
end if
printf ("\n")
end main
```

**Result**

child read - 1
child read - 2
child read - 3
child read - 4
child read - 5

**Category**

File and Device System Management:

---

**pitch**

**Alias**

`jog_t ...`

<table>
<thead>
<tr>
<th>alias</th>
<th>same as</th>
</tr>
</thead>
<tbody>
<tr>
<td>pitch</td>
<td><code>jog_t(TOOL_PITCH, ... )</code></td>
</tr>
</tbody>
</table>

**Description**

In the tool frame of reference, rotates around the orientation axis, the Y axis, by the specified number of degrees.

<table>
<thead>
<tr>
<th>Motion</th>
<th>axis</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>common name</td>
<td>F3 coordinate system</td>
</tr>
<tr>
<td>pitch</td>
<td>orientation</td>
</tr>
</tbody>
</table>

This command, `pitch()`, is joint-interpolated. The end position is determined and the tool travels to it as a result of various joint motions. The start point and end point for the tool centre point are the same (no change in distance along the axis or angle between the axis and the tool), but the start position and end position of the tool are different by the amount of rotation.

For cartesian-interpolated (straight line) motion, see `pitchs()`.

**Syntax**

```
command pitch( float distance )
```

**Parameter**

`distance` the amount of rotation in degrees: a float
### pitchs

**Alias**

<table>
<thead>
<tr>
<th>alias</th>
<th>same as</th>
</tr>
</thead>
<tbody>
<tr>
<td>pitchs</td>
<td>jog_ts(TOOL_PITCH, ... )</td>
</tr>
</tbody>
</table>

**Description**

In the tool frame of reference, rotates around the orientation axis, the Y axis, by the specified number of degrees.

<table>
<thead>
<tr>
<th>Motion</th>
<th>axis</th>
</tr>
</thead>
<tbody>
<tr>
<td>pitchs</td>
<td>orientation</td>
</tr>
</tbody>
</table>

This command, pitchs(), is cartesian-interpolated (straight-line) motion. The tool centre point stays on the axis, in the same place, while the tool rotates around the axis.

For joint-interpolated motion, see pitch().

**Syntax**

```
command pitchs( float distance )
```

**Parameter**

- **distance**  the amount of rotation in degrees: a float

**Returns**

- Success = 0
- Failure < 0

**Example**

- pitchs(22.5)
- pitchs(-90)

**Application Shell**

- Same as pitchs.

**RAPL-II**

No equivalent. In RAPL-II, PITCH performed a different motion. See yrots.

**See Also**

- pitch moves around the tool orientation axis, but joint-interpolated
- rolls moves around the tool approach/depart axis in straight line motion
- yaws moves around the tool normal axis in straight line motion

**Category**

- Motion
pos_axis_set
Description
Sets a specified axis to a specified position. Similar to zero(), but with a non-zero value.
Syntax
command pos_axis_set( int axis, int pos )
Parameter
axis : the axis ... : an int
pos : motor pulse count ... : an int
Returns
Success >= 0
Failure < 0
Example
int pulses
int axis
...
pos_axis_set(axis, pulses)
Result
Moves the joint “axis” by “pulses” pulse counts in the positive direction
See Also
pos_get
Category
Location: Data Manipulation

pos_get
Description
Gets the location information from the position registers.
Syntax
command pos_get(position_t postype, var ploc position )
Parameter
postype : the type of robot position:
POSITION_ACTUAL : the actual robot position
POSITION_COMMANDED : the commanded robot position
POSITION_ENDPOINT : the end-point robot position
POSITION_HOLD : the hold robot position
position : the position of the robot: a ploc
Returns
Success > 0, position is packed with the precision location
Failure < 0
Example
int test
ploc place
...
test = pos_get(POSITION_ACTUAL, place) ;; use test for error check
RAPL-II
Similar to:
W0, W1 pos_get(POSITION_COMMANDED)
W2, W3 pos_get(POSITION_ACTUAL)
W4 pos_get(POSITION_ENDPOINT)
ACTUAL pos_get(POSITION_ACTUAL)
except that RAPL-II generated output and ACTUAL also gave cartesian.
See Also
here stores the current location in a location variable
pos_set sets the position registers of the robot
Category
Location: Data Manipulation
Calibration
**pos_set**

Description
Load the robot position registers with location or pose information. Similar to `zero()`, but with a non-zero value. Does not move the arm.

Syntax
`command pos_set( ploc pos )`

Parameter
- `pos`: a ploc

Returns
- Success \( \geq 0 \)
- Failure \(< 0\)

Example
```
... teachable ploc there
... pos_set(there)
```

Result
Sets all axes to the position specified by the teachable ploc "there".

**pow**

Description
Calculates a value raised to a power. Takes a non-negative value and a non-negative power.

Syntax
`func float pow( float a, float b )`

Arguments
- `a`: the value
- `b`: the power

Returns
- Success \( \geq 0 \). The value `a` raised to the power `b`.
- Failure \(< 0\)

Example
```
float a = 2.5, b = 3.0
float y
y = pow( a, b )
```

Result
15.625

**print**

Description
Writes the specified data to standard output device, normally the terminal screen. Two types of arguments can be given in the variable argument list: constants and variables. The constants are printed exactly as they are given.
The variable’s value is what is copied to the output device. The method used in printing is to print the arguments in the exact order that they were given.

Syntax

command print ( ... )

Returns

- >= 0 Success.
- -EIO An I/O error occurred.
- -EINTR This operation was interrupted by a signal.

Example

count_cycle = 1048
print ( "Robot has worked ",count_cycle," cycles.\n" )

Result

Robot has worked 1048 cycles.
displayed at the terminal screen and the cursor advanced to a newline.

See Also

printf format print command to the standard output

Category

File Input and Output: Unformatted Output

printf

print formatted

Description

Converts and writes output to the standard output device, normally the terminal screen, under the control of a specified format \texttt{fmt}.

Format specifications are detailed in the Formatted Output section of File Input and Output

Syntax

command printf( var string[] fmt, ... )

Format Specifiers

The format string may consist of two different objects, normal characters, which are directly copied to the file descriptor, and conversion braces which print the arguments to the descriptor. The conversion braces take the format:

{ [ flags ] [ field width ] [ .precision ] [ e|E|f|g|G|x|X ] }

Flags

Flags that are given in the conversion can be the following (in any order):

- - (minus sign) specifies left justification of the converted argument in its field.
- + (plus sign) specifies that the number will always have a sign.
- 0 (zero) in numeric conversions causes the field width to be padded with leading zeros.

Field width

The field width is the minimum field that the argument is to be printed in. If the converted argument has fewer characters than the field, then the argument is padded with spaces (unless the 0 (zero) flag was specified) on the left (or on the right if the – (minus sign) was specified). If the item takes more space than the specified field width, then the field width is exceeded.

.precision

The precision number specifies the number of characters to be printed in a string, the number of significant digits in a float, or the maximum number of digits to be printed in an integer.

e or E
[For floating point numbers only] This flag indicates that a floating point number should be printed in exponential notation, which looks like:

```
[-d.dddde+dd] (e format)
or
[-d.dddE+dd] (E format)
```

The .precision refers to the number of digits after the decimal point, and defaults to 6 if it is omitted.

f

[For floating point numbers only] This flag indicates that a floating point number should be printed in ordinary floating point notation, which looks like:

```
[-d.ddd]
```

The precision refers to the number of digits after the decimal point, and defaults to 6 if it is omitted.

g or G

[For floating point numbers only. This is the default format for floating point.] This flag indicates that a floating point number should be printed either in f or e|E format, whichever is more compact. (e|E type is used if the exponent is less than –4 or the exponent is >= the .precision.) Note that for this mode only, the .precision indicates the number of significant digits to be printed, not the number of digits after the decimal point.

x or X

This is the hexadecimal flag which specifies whether or not an integer argument should be printed in hexadecimal (base 16) or not. The lowercase x specifies lowercase letters (abcdef) are to be used in the hexadecimal display and the uppercase X specifies uppercase letters (ABCDEF).

A character sequence of {{ means to print the single { (opening brace) character.

**Returns**

- `>= 0` Success.
- `-EINVAL` The arguments were invalid.
- `-EIO` An I/O error occurred.
- `-EINTR` This operation was interrupted by a signal.

**Example**

```c
float a = 1.23, b = 12.345, c = 1.234
float d = 98.7, e = -987654.3210, f = 9876.5
printf("a = (%.2f), b = (%+08.3f), c = (%-8.3f) \n", a, b, c)
printf("d = (%.2f), e = (%+08.3f), f = (%-8.3f) \n", d, e, f)
```

**Result**

```
a = 1.2, b = +00012.3 , c = 1.23
```
```
d = 99, e = -9.88e+005 , f = 9.88e+003 *
```

**Category**

File Input and Output: Formatted Output

---

**rad**

**Description**

Converts degrees to radians.

**Syntax**

```c
func float rad( float x )
```

**Returns**

The angle converted to radians.

**Example**

```c
float x = 45.0
float y
y = rad( x )
```

**Result**

```
0.785398
```

**RAPL-II**

RAD
### rand

**Description**
A function for generating random numbers (integers). The function uses a seed value which can be set using the `rand_next` function.

**Syntax**
```plaintext
func int rand()
```

**Returns**
Returns a random number.

**Example**
```plaintext
int r = 5
int seed = 13
int[] random

... srandom(int seed) ;; sets the seed value rand_next = 13
...

;; generate a 5 element array of random numbers
for j = 1 to r
    random[j-1] = rand()
end for
```

**Result**
A 5 element array of random number integers.

**See Also**
- `rand_in` generates random numbers within a specified range
- `srandom` sets the random generator seed value

**Category**
Math

### rand_in

**Description**
A function for generating random numbers (integers) which fall in the range specified. The function uses a seed value which can be set using the `rand_next` function.

**Syntax**
```plaintext
func int rand_in(int min, int max)
```

**Parameters**
`min`, `max` are integer values which define the range of random numbers returned.

**Returns**
Returns a random number in the range `[min..max]`.

**Example**
```plaintext
int r = 5
int seed = 13
int min = {expression}
int max = {expression}
int[] random(min max)
int j

... srandom(int seed) ;; sets the seed value rand_next = 13

;; generate a 5 element array of random numbers
for j = 1 to r
    random[j-1] = rand_in(min, max)
end for
```

**Result**
A 5 element array of random number integers with values between `min` and `max`.

**See Also**
- `rand` generates random numbers
- `srandom` sets the random generator seed value

**Category**
Math
**rcv**

Description

Receives words from a socket. If the rcv() command succeeds, it returns the (positive) number of words (4 byte entities) read. This may be less than *nwords*, the length of the receive buffer. If the rcv() command fails, it returns a negative error code. If the timeout is specified, rcv() will try to read for *timeout* milliseconds before returning. Words that are read are placed into *buf*, which must be at least of size *nwords*. If *ppid* is a NULL pointer, the receive can be from any process. If *ppid* is not a NULL pointer, the value of the variable being pointed to is the pid of the process from which you are trying to receive. If that *ppid* is 0, it receives from any process and returns the pid of that process.

If a server tries to receive from a client with a timeout of TM_NOWAIT and the client is non-existent, the error code -ENOCLIENT is returned.

rcv() is similar to read() which is used for all other (non-socket) entities.

Syntax

```c
command rcv(int fd, void *buf, int nwords, int timeout, int *ppid)
```

Parameters

- **fd**
  The file descriptor referring to the open socket.

- **buf**
  Points to where to store the received data.

- **nwords**
  The number of word to receive, maximum. Note that it is not an error for the sending process to send fewer than *nwords* words.

- **timeout**
  How long to wait for the transaction, in milliseconds. There are two special values, TM_NOWAIT (don't wait at all) and TM_FOREVER (wait forever.)

- **ppid**
  If this is NULL, then we are trying to rcv() from any other process. If non-NULL, then this is a pointer to an integer in which the desired process id (pid) of the sender is stored (with 0 meaning any). On success, rcv() stores the actual sending process id in *ppid*.

Returns

- >= 0
  Success. Returns the number of words received.

- EINVAL
  The arguments were invalid (eg., *fd* was –ve)

- EBADF
  The file descriptor does not correspond to an open object.

- ENOTSOCK
  The object open on *fd* is not a socket.

- EAGAIN
  Too large a receive was attempted; also returned when a TM_NOWAIT rcv() does not immediately succeed.

- ETIMEOUT
  The *timeout* expired.

- EINTR
  The operation was interrupted by a signal.

- ENOSERV (client only)
  There is no server serving this socket.

- ENOCLIENT
  There is no client matching the parameters of the
Reading from a file descriptor:

Example:

```c
int sock_fd
string[30] mbuf
...
;; Open a socket for a client.
open (sock_fd, "mydev", O_CLIENT, 0)
...
;; Receive message from the socket.
rcv (sock_fd, &mbuf, sizeof(mbuf), TM_FOREVER, NULL)
```

See Also:

- `send` sends words to a socket
- `open` opens a socket and other entities

Category:

- Device Input and Output

**read**

Description:

Attempts to read `nwords` from the file descriptor `fd` and store the result in `buf`. If the number of words specified in `nwords` cannot be read the command will perform a blocking read, unless the file descriptor was opened with mode `O_NONBLOCK`. After reading, the file position is moved by the number of words read. This provides a sequential move through the file.

The `read()` command reads 4-byte words (32 bits). The `reads()` command reads characters (8 bits).

Similar to `rcv()` which is used for sockets.

Syntax:

```
command read( int fd, void* buf, int nwords )
```

Parameters:

- `fd` the open file descriptor
- `buf` a pointer to where to store the read data
- `nwords` the number of 4-byte words to be read: an int

Returns:

- `> 0` Success; the number of words actually read.
- `0` The end of file was encountered.
- `-EINVAL` The arguments were invalid.
- `-EBADF` `fd` does not correspond to an open file.
- `-EACCESS` The file is not open for reading.
- `-ESPIPE` Attempted to read a socket.
- `-EIO` An I/O error occurred.
- `-EAGAIN` (nonblocking I/O) No bytes were ready for reading.
- `-EINTR` This operation was interrupted by a signal.

Example:

```c
int fd
int[10] buf
...
open ( fd, "filename.txt", O_RDONLY, 0 )
read ( fd, buf, sizeof(buf) )
```

Example:

```c
int a ;; reads four characters from keyboard
read ( stdin, &a, 1 ) ;; and stores them as an int
print ( a, "\n" ) ;; returns only when four characters are entered
```

RAPL-II

GETCH
See Also
reads reads a string from a file
readsA reads a string from a file and appends it to a string
write writes to a file
writes writes a string to a file
open opens a file to read, write, etc.

Category File Input and Output: Unformatted Input

readdir

Description
Reads a directory entry and stores the structure in \textit{buf}. Reading from the
directory automatically increments the file pointer for \textit{fd}.

Syntax
\begin{verbatim}
command readdir( int \textit{fd}, var c_dirent \textit{buf} )
\end{verbatim}

Parameters
\begin{itemize}
\item \textit{buf} a c_dirent structure with the following fields:
\begin{itemize}
\item string[32] \textit{de_name}
\item int \textit{de_type}
\item int \textit{de_links}
\item mode_flags \textit{de_mode}
\item int \textit{de_size}
\item int \textit{de_mtime}
\item int \textit{de_dev}
\item int \textit{de_ident}
\end{itemize}
\item \textit{fd} The file descriptor to read from.
\end{itemize}

Returns
\begin{itemize}
\item 1 Success.
\item 0 The end of the directory was encountered.
\item -EINVAL The arguments were invalid.
\item -EBADF \textit{fd} does not correspond to an open file.
\item -EACCESS The file is not open for reading.
\item -ENOTDIR \textit{fd} does not correspond to an open directory.
\item -EIO An I/O error occurred.
\item -EINTR This operation was interrupted by a signal.
\end{itemize}

Example
\begin{verbatim}
string[] dir = "/temp"
c_dirent buf
int fd
...
open ( fd, dir, O_RDONLY, 0 )
...
result = readdir( fd, buf )
while result > 0
    print ( buf.de_name,"\n" )
    result = readdir( fd, buf )
end while
\end{verbatim}

Category File and Device System Management
**readline**

Description  
Interactively reads a line of up to `maxlen` characters from stdin to `s` and echos to stdout. The line terminator can be either a carriage return or a line feed. Returns the number of characters actually read including the terminator. A value of 0 means EOF.

Syntax  
`command readline ( var string[] s, int maxlen )`

Parameters  
- `s` Where to store the read data
- `maxlen` The maximum number of characters to read.

Returns  
- `> 0` Success; the number of words actually read.
- `0` The end of file was encountered.
- `-EINVAL` The arguments were invalid.
- `-EIO` An I/O error occurred.
- `-EINTR` This operation was interrupted by a signal.

Example  
```
int maxlen
string[32] safe = myfile.txt
...
readline ( safe, maxlen)
```

Results  
Reads “`maxlen`” characters from the standard input and writes them to “`myfile.txt`, and to stdout.

See Also  
reads  
read

Category  
File Input and Output: Unformatted Input

**reads**

Description  
Reads a string from a file of at most `maxlen` characters. This is different from the read command in that a string is used, and the length of the string is updated. The number of characters read is returned, or a negative error code if the read fails.

The reads() command reads characters (8 bits). The read() command reads 4-byte words (32 bits).

Syntax  
`command reads( int fd, var string[] s, int maxlen )`

Parameters  
- `s` Where to store the read data.
- `maxlen` The maximum number of characters to read.
- `fd` The file descriptor to read from.

Returns  
- `> 0` Success; the number of words actually read.
- `0` The end of file was encountered.
- `-EINVAL` The arguments were invalid.
-EBADF  
  $fd$ does not correspond to an open file.
-EACCESS  
  The file is not open for reading.
-ESPIPE  
  Attempted to read a socket.
-EIO  
  An I/O error occurred.
-EAGAIN  
  (nonblocking I/O) No bytes were ready for reading.
-EINTR  
  This operation was interrupted by a signal.

Example

string[20] buf
int fd
open ( fd, ”/temp/reads_test”, O_RDONLY, 0 )
reads ( fd, buf, 20 )
print ( buf,"\n"
)

Example

string[1] a ;; reads a string of 1 character
reads ( stdin, a, 1 ) ;; when a key is pressed, the command
returns
print ( a,"\n" ) ;; useful for keyboard input

See Also

read  
  read words (4 byte units) from a file
readsa  
  read a string from a file and append it to a string

Category

File Input and Output: Unformatted Input

**readsa**

Description

Reads a string (of at most $maxlen$ characters) from a file, and appends it on the end of string $s$.

Syntax

```
command readsa(int $fd$, var string[] $s$, int $maxlen$)
```

Parameters

$s$  
Where to store the read data.

$maxlen$  
The maximum number of characters to read.

$fd$  
The file descriptor to read from.

Returns

- $> 0$  
  Success; the number of words actually read.
- $0$  
  The end of file was encountered.
-EINVAL  
  The arguments were invalid.
-EBADF  
  $fd$ does not correspond to an open file.
-EACCESS  
  The file is not open for reading.
-ESPIPE  
  Attempted to read a socket.
-EIO  
  An I/O error occurred.
-EAGAIN  
  (nonblocking I/O) No bytes were ready for reading.
-EINTR  
  This operation was interrupted by a signal.

Example

```
string[MAXLEN] results
int fd
int length, check

open(fd, ”mydirectory\result.txt”, O_RDONLY, 0)
```
check = readsa(fd, results, length)

Result
"check" is equal to the number characters appended to string "results"

See Also
read             read words (4 byte units) from a file
reads            read a string from a file

Category
File Input and Output: Unformatted Input

---

**ready**

Description
Moves the arm to the READY position.

Syntax
command ready()

Returns
Success >= 0
Failure < 0

Example
if (ready() >= 0)
    move (a)
end if

RAPL-II
Similar to READY.

See Also
home       homes the axes

Category
Calibration
Motion

---

**rmdir**

Description
Deletes an empty directory.

Syntax
command rmdir( var string[] path )

Parameters
path       full path name of the directory to delete

Returns
Success >= 0
Failure < 0
-EINVAL      invalid argument
-ENOTDIR     the path is not a directory
-ENOENT      a component was not found
-EIO         an I/O error occurred
-EAGAIN      temporarily out of resources needed to do this
-EBUSY       the directory is busy
-ENOTEMPTY   the directory is not empty

Example
string[20] path =/mydirectory
...
rmrdir(path)

Result
The directory /mydirectory is deleted

See Also
mknod
mkdir

Category
File and Device System Management

---

**robot_abort**

Description
Stops current motion and discards the contents of the motion queue.
robot_abort() operates by locating the pid of the server (by a zero-length rcv() on the /dev/robot socket) and sending the server a SIGABRT. If the rcv() fails, then robot_abort() opens /dev/estop, which forces arm power off.

| Syntax | command robot_abort() |
| Parameter | empty |
| Returns | Success = 0  
Failure < 0 |
| Example | ...  
robot_abort()  
. . |
| Category | Motion |

### robot_cfg_save

**Description**
Re-writes the “/conf/robot.cfg” file with the current robot configuration information, which includes:
1. whether or not the robot has a track
2. the number of axes on the controller
3. the tool transform
4. the base offset
5. the positive and negative track travel limits
6. the gripper type
7. the robot units (metric or English)

It must be pointed out that changing one of these parameters in your program does not change the default for when the system is rebooted; you must perform a robot_cfg_save() to make the changes permanent.

| Syntax | command robot_cfg_save() |
| Returns | Success >= 0  
Failure < 0 (-ve error code) |
| Example | ;; "permanently” set a tool transform:  
tool_set(cloc{0, 0, 0, 1, 0, 0, 0, 0, 0})  
robot_cfg_save() |
| See Also | tool_set(), base_set(), griptype_set()  
/diag/setup (system shell command) |
| Category | Motion |

### robot_error_get

**Description**
Returns the current (latest) error state of the robot.

| Syntax | command robot_error_get( var int[5] error ) |
| Parameter | error * : an array of up to 5 ints |
| Returns | Success >= 0  
Failure < 0 |
| Category | Robot Configuration  
System Process Control: Single and Multiple Processes |
robot_flag_enable

Description
Enables flags.

Syntax
```
command robot_flag_enable( enable_flag_t flag, int state )
```

Parameter
- `flag` a variable of the enumerated type `enable_flag_t` an
- `state` an int

Returns
Success >= 0
`flag` is packed with one of:
- `EFLAG_INVALID`: 0
- `EFLAG_TRAPEZOID`: 1
- `EFLAG_TRIGGER`: 2
Failure < 0

Category
Robot Configuration

robot_info

Description
Returns robot info in the variables “homed”, and “done” whether the robot is done moving and homed.

Syntax
```
command robot_info(var int homed, var int done)
```

Parameter
- `homed` packed with the homed status
- `done` packed with the robot motion status

Returns
Success = 0
Failure < 0

Example
```
int homed, done
robot_info(homed, done)
if (homed != 0 && done != 0)
    printf("robot is homed and not moving\n")
else
    if (done == 0)
        printf("robot in motion \n")
    end if
    if (homed == 0)
        printf("robot is not homed\n")
    end if
end if
```

Result
Reports if the robot is homed and if it is in motion

See Also
- server_info
- robotisfinished

Category
Robot Configuration
Motion

robot_mode_get

Description
Gets the current mode of motion and packs it into a variable of an enum type.

Syntax
```
command robot_mode_get( var motion_mode_t mmode )
```
### robot_mode

**Description**
Allow the user to move the robot using the pendant

**Library**
stp

**Syntax**
`export command robot_move()`

**Parameter**
None

**Returns**
Success >= 0
Failure < 0

**Example**
```c
string[10] name = “my_app_23”
stp:startup
stp:app_open(name, 0)
...
  stp:robot_move()  stp:app_close()
...
stp:app_close()
...
```

**Category**
Pendant

### robot_odo

**Description**
Gets the current value of the robot arm power odometer, which indicates the number of seconds that arm power has been turned “on” for.

**Syntax**
`command robot_odo(var int seconds)`

**Returns**
Success >= 0; seconds gets the odometer value.
Failure < 0 (-ve error code)

**Example**
```c
int otime
...
robot_odo(otime)
printf(“The robot arm power has been on for {} seconds.\n”, otime)
```
See Also

robot_servo_stat

robot_type_get

robotisdone

RAPL-II FINISH

See Also

robotisfinished

finish allows robot motions to catch up to process

Category

Robot Configuration

System Process Control: Single and Multiple Processes
**robotisfinished**

Description

The robotisfinished function uses the same finish service as the finish() command except now a mode flag is passed into the service. The finish_mode_t is a global enum. The function returns 1, if the robot is finished, 0 if not finished and a error code if error occurs.

Syntax

```
func int robotisfinished()
```

Parameter

no parameter is required

Returns

Success > 0 1 robot is finished move
0 robot is not finished move
Failure < 0 error code

Example

```
;; Use command to synchronize robot motion
.define PALLET_NUM 25
.teachable ploc[10] pallet
.teachable ploc safe_pallet
.int i

if o r i = 0 t o PALLET_NUM
   move(pallet[i])
   loop
      if robotisfinished()
         grip_close(50)
      else
         msleep(250)
      end if
   end loop
   move (safe_pallet)
   ...
end for
```

Result

Program waits until robot is at pallet location before closing gripper

RAPL-II

Similar to FINISH

See Also

robotisdone
finish

Category

Status

**robotishomed**

Description

Returns the current robot home state. This function checks all transform axes for a home state and returns the logical AND of these states. All transform axes must be homed for this routine to return TRUE (>=0)

Syntax

```
func int robotishomed()
```

Returns

Success

> 0  all axes of arm are homed
= 0  at least one axis is not homed
Failure < 0

Example

```
home_state = robotishomed()
if (home_state)
   ;; robot is homed continue
```
else
    ::home the robot
    home(i,2,3,4,5,6)
end if

See Also
    calibrate calibrates the robot
    home homes the robot

Category
    Home

------

robotislistening

Description
    A function to determine if the robot server is responding to queries. The function
    returns TRUE if the robot responds to the arm power query. If no response, it
    returns FALSE.

Syntax
    funct int robotislistening()

Returns
    Success >= 0   TRUE or FALSE
    Failure < 0   Does not return a negative error code.

Example
    if robotislistening()
        printf("Robot is ready begin")
        ;; program here
    else
        printf("Robot is not listening")
    end if

See Also
    robotisfinished
    robotishomed

Category
    Robot Configuration
    Status

------

robotispowered

Description
    Returns the current state of the robot arm power. Useful for checking arm power
    status before proceeding to further program execution.

Syntax
    fun int robotispowered()

Returns
    Success
    > 0   arm power is ON
    = 0   arm power is OFF
    Failure < 0

Example
    if robotispowered() == 0
        print "Waiting for arm power.\nTurn on arm power.\n" 
        do
            msleep 1000
        until robotispowered() > 0
    end if

RAPL-II
    Similar to ONPOWER.

Category
    Status

------

roll

Alias
    jog_t ...

<table>
<thead>
<tr>
<th>alias</th>
<th>same as</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**In the tool frame of reference, rotates around the approach/depart axis, by the specified number of degrees.**

<table>
<thead>
<tr>
<th>motion</th>
<th>axis</th>
<th>common name</th>
<th>F3 coordinate system</th>
<th>A465/A255 coordinate system</th>
</tr>
</thead>
<tbody>
<tr>
<td>roll</td>
<td>approach/depart</td>
<td>Z</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

This command, roll(), is joint-interpolated. The end position is determined and the tool travels to it as a result of various joint motions. The start point and end point for the tool centre point are the same (no change in distance along the axis or angle between the axis and the tool), but the start position and end position of the tool are different by the amount of rotation.

For cartesian-interpolated (straight line) motion, see rolls().

**Syntax**

```
command roll( float distance )
```

**Parameter**

- **distance** the amount of rotation in degrees: a float

**Returns**

- Success = 0
- Failure < 0

**Example**

```
roll(11.25)
roll(-45)
```

**Application Shell**

Same as roll

**RAPL-II**

No equivalent. In RAPL-II, ROLL performed a different motion. See xrot.

**See Also**

- rolls moves around the tool approach/depart axis, but in straight line motion
- pitch moves around the tool orientation axis
- yaw moves around the tool normal axis

**Category**

Motion

---

**rolls**

**Alias**

- **jog_ts ...**

<table>
<thead>
<tr>
<th>alias</th>
<th>same as</th>
</tr>
</thead>
<tbody>
<tr>
<td>rolls</td>
<td>jog_ts(TOOL_ROLL, ... )</td>
</tr>
</tbody>
</table>

**Description**

In the tool frame of reference, rotates around the approach/depart axis, by the specified number of degrees.

<table>
<thead>
<tr>
<th>motion</th>
<th>axis</th>
<th>common name</th>
<th>F3 coordinate system</th>
<th>A465/A255 coordinate system</th>
</tr>
</thead>
<tbody>
<tr>
<td>roll</td>
<td>approach/depart</td>
<td>Z</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

This command, rolls(), is cartesian-interpolated (straight-line) motion. The tool centre point stays on the axis, in the same place, while the tool rotates around the axis.

For joint-interpolated motion, see roll().

**Syntax**

```
command rolls( float distance )
```
**Parameter**
- **distance**  the amount of rotation in degrees: a float

**Returns**
- Success = 0
- Failure < 0

**Example**
- `rolls(45)`
- `rolls(-10.5)`

**Application Shell**
- Same as `rolls`.

**RAPL-II**
- No equivalent. In RAPL-II, ROLL performed a different motion. See `xrots`.

**See Also**
- `roll` moves around the tool approach/depart axis, but joint-interpolated
- `pitchs` moves around the tool orientation axis in straight line motion
- `yaws` moves around the tool normal axis in straight line motion

**Category**
- Motion

---

**rotacc_get**

**Description**
Returns the value of the maximum rotational acceleration parameter. This parameter is used to regulate rotational accelerations when performing straight-line motions in online mode and when using the teach pendant. Units are in degrees/second/second.

**Syntax**
```
command rotacc_get(var float rotaccel)
```

**Parameter**
- `rotaccel` a float into which the current rotational acceleration value is placed

**Returns**
- Success >= 0
- Failure < 0

**Example**
```
float rotaccel
...
rotacc_get(rotaccel)
printf(“Max. rotational accel is set to {} deg/sec/sec”, rotaccel)
```

**See Also**
- `rotacc_set`, `rotspd_set`, `rotspd_get`

**Category**
- Robot Configuration

---

**rotacc_set**

**Description**
Sets the value of the maximum rotational acceleration parameter. This parameter is used to regulate rotational accelerations when performing straight-line motions in online mode and when using the teach pendant. It is not possible to set the value of this parameter higher than the default value, which is robot dependent. Units are in degrees/second / second.

**Syntax**
```
command rotacc_set( var float rotacc )
```

**Parameters**
- `rotacc` a float which carries the new rotational acceleration value

**Returns**
- Success >= 0
- Failure < 0

**Example**
```
float rotacc
if nextpart == KRUMHORN
  rotacc = 20
  rotacc_set(rotspeed)
end if
```

**See Also**
- `rotacc_get`, `rotspd_set`, `rotspd_get`

**Category**
- Robot Configuration
### rotspd_get

**Description**
Retrieves the current value of the maximum rotational speed parameter. This parameter is used to regulate rotational velocities when performing straight-line motions in online mode and when using the teach pendant. Units are in degrees/second.

**Syntax**
```plaintext
command rotspd_get( var float rotspeed )
```

**Parameter**
- `rotspeed` a float into which the rotational speed value is placed

**Returns**
- Success \( \geq 0 \)
- Failure \(< 0\)

**Example**
```plaintext
float rotspeed, dispensing_limit
...
dispensing_limit = 155
rotspd_get(rotspeed)
if rotspeed > dispensing_limit
    rotspd_set(dispensing_limit)
end if
...
```

**See Also**
rotspd_set, rotacc_set, rotacc_get

**Category**
Robot Configuration

### rotspd_set

**Description**
Sets the value of the maximum rotational speed parameter. This parameter is used to regulate rotational velocities when performing straight-line motions in online mode and when using the teach pendant. It is not possible to set the value of this parameter higher than the default value, which is robot dependent. Units are in degrees/second.

**Syntax**
```plaintext
command rotspd_set( var float rotspeed )
```

**Parameters**
- `rotspeed` a float which carries the new rotational speed value

**Returns**
- Success \( \geq 0 \)
- Failure \(< 0\)

**Example**
```plaintext
float rotspeed
if nextpart == DASHBOARD
    rotspeed = 100
    rotspd_set(rotspeed)
end if
...
```

**See Also**
rotspd_get, rotacc_set, rotacc_get

**Category**
Robot Configuration

### seek

**Description**
Provides a method to move through a file arbitrarily rather than sequentially (see read() and write().) The position is moved to a place in the file specified by `offset` from the base given in `whence`. Subsequent reading and writing begin at this new position.

**Syntax**
```plaintext
command seek( int fd, int offset, seek_base whence )
```
Parameters

\( fd \) identifies the file

\( whence \) can be one of

- \( SEEK_SET = 0 \) move from beginning of file
- \( SEEK_CUR = 1 \) move from current position
- \( SEEK_END = 2 \) move from end of file

offset offset position form the base specified by whence

Returns

Success \( \geq 0 \)

Failure \( < 0 \)

- \( EINVAL \) the arguments were invalid (ie., -ve \( fd \)), or this operation is not legal on this device.
- \( EBADF \) the file descriptor isn't open
- \( ESPIPE \) can't seek on a pipe or socket

Example

```
int fd
string[] buffer = "seek test"
...
open ( fd, "filename", O_RDWR, 0 ); Open the file
write ( fd, buffer, 9 ); Write to the file
seek ( fd, 0, SEEK_SET ); Rewind the file
```

See Also

read read from a file
write write to a file

Category

File Input and Output: Unformatted Input

---

**select_menu**

Description

Displays the three lines \( s_1 \), \( s_2 \) and \( s_3 \) on the pendant screen. Show key labels \( k_1 \) to \( k_4 \) and then wait for the user to select a function key. The integer number of the key selected is returned.

Note that if any of the function key labels (\( k_1 \) - \( k_4 \)) are null strings then the corresponding key will NOT be enabled. The \( kn \) strings are printed literally; but they must be limited by the programmer to 4 characters.

Syntax

```
stp:func int select_menu(var string[] s1, var string[] s2, var string[] s3, \nvar string[] k1, var string[] k2, var string[] k3, var string[] k4)
```

Parameters

- \( s_1 \) string displayed in the top line of the pendant
- \( s_2 \) string displayed in the second line of the pendant
- \( s_3 \) string displayed in the third line of the pendant
- \( k_1 \) Function key 1 label (max 4 characters)
- \( k_2 \) Function key 2 label (max 4 characters)
- \( k_3 \) Function key 3 label (max 4 characters)
- \( k_4 \) Function key 4 label (max 4 characters)

Returns

Success \( \geq 0 \) Returns the integer number of the Function key selected, 0 if the user exits the pendant menu

Failure \( < 0 \)

Example

```
int ctrl = 0
...
stp:startup()
...
ctrl=stp:select_menu("Welcome", "Just Call me Teach", "Do you want to", 
"Cont","Exit","","")
if ctrl == 1
    ::continue
    ....
end if
if ctrl == 2
    ::exit
    ...
```
sem_acquire

Description
Attempts to acquire a semaphore specified by key. If the semaphore is granted the command returns successful, otherwise a negative error code is returned. A timeout can be specified which causes the function to wait to acquire the semaphore until timeout has been reached. Timeout is in milliseconds.

Syntax
command sem_acquire( int key, int timeout )

Parameter
key an int
timeout an int time in milliseconds

Returns
Success >= 0
Failure < 0 Returns negative error code
- EOK success
- EAGAIN the system is out of semaphore slots, or TM_NOWAIT was specified and we did not acquire the semaphore right away.
- ETIMEOUT timed out
- EINTR the operation was interrupted by a signal.

Example
int result, key = 1
int timeout = 50
...
result = sem_acquire( key, timeout )
if result == EOK
    ;; enter critical section
    sem_release( key, timeout )
end if

Category
System Process Control: Single and Multiple Processes

sem_release

Description
Releases the semaphore specified by key. If the semaphore can be successfully released, the command returns successful, otherwise the command returns an error code. If the timeout is specified, the command will keep attempting to release the semaphore until timeout value is reached.

Trying to release a semaphore that has not be acquired will result in the command attempting to acquire it first, and then release it.

Syntax
command sem_release( int key, int timeout )

Parameter
key an int
timeout an int time in milliseconds

Returns
Success >= 0
Failure < 0 Returns negative error code.
- EOK success
- EAGAIN the system is out of semaphore slots, or TM_NOWAIT was specified and we did not acquire the semaphore right away.
- ETIMEOUT timed out
- EINTR the operation was interrupted by a signal.

Example
int result, key = 1
int timeout = 50
...
result = sem_acquire( key, timeout )
if result == EOK
    ;; enter critical section
sem_release( key, timeout )
end if

Category
System Process Control: Single and Multiple Processes

---

**sem_test**

**Description**
Tests the semaphore specified by `key`.

**Syntax**
command sem_test( int key )

**Parameter**
`key` an int specifies the semaphore

**Returns**
Success >= 0 Returns 1 if the semaphore is set, 2 if it is set and is owned by the calling process, and 0 if it is clear.
Failure < 0

**Example**
int result, key = 1
int timeout = 50
...
loop
  result = sem_test( key )
  if result == EOK
    break
  end if
end loop
result = sem_acquire( key, timeout )
if result == EOK
  ;; enter critical section
  sem_release( key, timeout )
end if

Category
System Process Control: Single and Multiple Processes

---

**send**

**Description**
Sends `nwords` words into the socket described by `d`. The number of words actually written is returned. If `timeout` is not TM_FOREVER, send will only attempt to write words for `timeout` milliseconds. If `pid` is not 0, the message is sent to a client process specified by `pid`. (This must be the server). Otherwise, the sender is the client.

If a server tries to send to a client with a timeout of TM_NOWAIT and the client is non-existent, the error code -ENOCLIENT is returned.

`send()` is similar to `write()` which is used for all other (non-socket) entities.

**Syntax**
command send(int d, void *buf, int nwords, int timeout, int pid)

**Parameters**
`d` an int -specifies the socket
`nwords` an int - number of words
`pid` an int- specifies the process (must be server or 0)

```
TM_NOWAIT
TM_FOREVER
```

**Returns**
Success >= 0 the number of words written
Failure < 0
-EINVAL the arguments were invalid (ie., -ve fd)
-EBADF the file descriptor isn't open
-ENOTSOCK the file was not a socket
Subprograms: Alphabetical Listing

-EAGAIN  too large a write; also returned on TM_NOWAIT sends that immediately time out.
-ETIMEOUT the timeout expired
-EINTR the operation was interrupted by a signal

Client only:
-ENOSERV there is no server

Server only:
-EBUSY there is already a server waiting to send
-ENOCLIENT there is no client that fits the send()

Example

```c
int sock_fd
string[] mbuf = "1 client"
...
;; Open a socket for a client
open ( sock_fd, "/mydev", O_CLIENT, 0 )

;; Send Message to the socket.
send (sock_fd, &mbuf, sizeof(mbuf), TM_FOREVER, 0)
```

See Also
rcv  receives words from a socket

Category
Device Input and Output

server_get

Description
Used with multi-robot systems.

Gets the name of the current server socket device, the socket/robot server that the library is communicating with.

Syntax
```
cmd server_get ( var string[] currserver )
```

Parameter
currserver  string a variable for the name of the current server: a variable length string

Returns
Success = 0  EOK if successful
name of current server packed in currserver

Failure < 0
-EIO  server is not connected

Example

```c
string[32] cur_serve, serve
int pcode, mach_type, tran_ax, act_ax, mach_ax, power
int t
...
serve = "robot1"
t= server_get(cur_serve)
if (t >= 0 && cur_serve == serve)
    server_info(mach_type, pcode, mach_ax,\
    tran_ax, act_ax, power )
    printf("Robot is {}n Product Code is {}n", mach_type,\
    pcode)
else
    server_set(serve)
    server_info( mach_type, pcode, mach_ax,\
    tran_ax, act_ax, power )
    printf("Robot is {}n Product Code is {}n", mach_type, pcode)
end if
```
See Also  
server_info  
server_protocol  
server_version  

Category  
File and Device System Management  
Robot Configuration  

server_info  

Description  
Similar to robot_info. Obtains: machine type, product code, machine axes,  
transform axes, actual axes, arm power.

Syntax  
global command server_info( var int mtype, var int pcode,  
\           var int axm, var int axt, var int axa, \   
\           var int power )

Parameter  
mtype  
a string for machine type data  
pcode  
a string for product code data  
axm  
an int for machine axis data  
axt  
an int for transform axis data  
axa  
an int for actual axis data  
power  
an int for the arm power status

Returns  
Success >= 0  Variables are packed with the server info  
Failure < 0

Example  
;;; An inefficient example program to show function of  
;;; server_get, server_info, server_set commands.  
;;; In the end prints the Machine type and Product code data  
;;; for the machine talking to the server "serve"...

string[32] cur_serve, serve  
int pcode, mach_type, tran_ax, act_ax, mach_ax,power  
int t  
...

serve = "robot1"  
t= server_get(cur_serve)  
if (t >= 0 && cur_serve == serve)  
    server_info(mach_type,pcode, mach_ax,\  
        tran_ax, act_ax, power )  
    printf("Robot is {}/n Product Code is {}/n", mach_type, pcode)  
else  
    server_set(serve)  
    server_info( mach_type, pcode, mach_ax,\  
        tran_ax, act_ax, power )  
    printf("Robot is {}/n Product Code is {}/n", mach_type, pcode)  
end if

See Also  
server_get  
server_set  

Category  
File and Device System Management  
Robot Configuration  

server_protocol  

Description  
Server_protocol function returns the protocol designator from the robot server.

Syntax  
func int server_protocol()
server_set

Description
Used with multi-robot systems.

Sets the robot server socket connection in the library to the specified new server
value, changing the socket/robot server that the library is communicating with.
Any existing socket connection is closed and the new socket opened.

A parameter of DEFAULT sets the socket connection back to /dev/robot.

If the command fails to open the new socket, any subsequent attempts to access
the robot server fail with an -EIO.

Syntax
command server_set( var string[] newserver )

Parameter
newserver the name of the new server: a variable length string
                  [path]    the path of any valid socket
                  DEFAULT the default socket, /dev/robot

Returns
Success = 0
Failure < 0
-EIO    failed to open new socket

Category
File and Device System Management
Robot Configuration

server_version

Description
The server_version function returns an integer which specifies the robot server
version.

Syntax
func int server_version()

Returns
Success >= 0    Returns integer which specifies the version.
Failure < 0      Returns negative error code if command fails.

See Also
server_protocol Returns the protocol designator from the server.

Category
File and Device System Management
Robot Configuration

setenv

Description
Creates / redefines an environment variable's value. (See the section on environ() for more explanation.) (C500C only)

Syntax
command setenv(string[] key, string[] value, int rewrite)

Parameters
There are three required parameters:

key            The key to define / change. (This is the portion on the
setenv

Description
Sets the priority of a process by adjusting the priority by an increment, \textit{delta}. Also, gets the current priority of a process.

There are three priority levels: high (3), normal (2), and low(1). The normal level is the usual priority level. During processing, the system alternates among processes. A process at a higher level can exclude a process at a lower level. Improper use of setenv() could starve other processes including the robot server.

The setenv() command is useful, for example, to do independent calculations at a low priority without slowing down processing for robot activity, or to respond immediately to a GPIO input by adjusting a process to a higher priority. The system can raise or lower a priority across the entire range. A user can lower a process below normal and raise it back to normal.

To change the priority of the current process, pid is 0 (zero).

To get the current priority level, use 0 (zero) for the increment, \textit{delta}. A child process is created with whatever priority level the parent had.

Returns the new priority as an absolute integer (not an increment).

Syntax
func int setenv( string name, string value, bool rewrite )

Parameter
\textit{name} the name of the environment variable
\textit{value} the value to set the right hand side of the ”=” in the environment string to.
\textit{rewrite} If False (0), do not modify an existing environment string; only create a new one if one does not yet exist. If True (1), rewrite the environment string if it already exists.

Returns
Success: returns 0. Not rewriting an existing string (rewrite == 0) is also considered success.
Failure: returns -1

Example

;; Define a new variable called ”TestMode”, whose value is ”yes”
setenv("TestMode", "yes", True)

See Also
environ(), getenv(), unsetenv()

Category
Environment Variables

setprio

Description
Sets the priority of a process by adjusting the priority by an increment, \textit{delta}. Also, gets the current priority of a process.

There are three priority levels: high (3), normal (2), and low(1). The normal level is the usual priority level. During processing, the system alternates among processes. A process at a higher level can exclude a process at a lower level. Improper use of setprio() could starve other processes including the robot server.

The setprio() command is useful, for example, to do independent calculations at a low priority without slowing down processing for robot activity, or to respond immediately to a GPIO input by adjusting a process to a higher priority. The system can raise or lower a priority across the entire range. A user can lower a process below normal and raise it back to normal.

To change the priority of the current process, pid is 0 (zero).

To get the current priority level, use 0 (zero) for the increment, \textit{delta}. A child process is created with whatever priority level the parent had.

Returns the new priority as an absolute integer (not an increment).

Syntax
func int setprio( int pid, int delta )

Parameter
\textit{pid} the process id number (0 is current process)
\textit{delta} amount of adjustment of priority

Returns
Success > 0
The new priority: an absolute int.
1 is PR_LOW
2 is PR_NORM
3 is PR_HIGH
Failure < 0
-EINVAL the arguments were not valid
-EPERM a non-privileged process can only change its OWN priority

Example

setprio( 26, 0) ;; get process 26 priority
setprio( 26,-1) ;; set process 26 priority down 1 level
setprio( 0,-1) ;; set current process priority down 1 level
setprio( 26,+1) ;; set process 26 priority up 1 level
( 0,+1) ;; set current process up 1 level
See Also

- getpid gets the id number of the process of the calling program
- getppid gets the id number of the parent process of the calling program

Category
System Process Control: Single and Multiple Processes

### shift_t

**Description**

In the tool frame of reference, alters the cartesian coordinates of a location. A precision location cannot be changed with this command. There are two possible formats: using a cloc type or using individual displacements. In both formats, the first argument is the location to be shifted.

If a cloc type is used, the displacement values are earlier stored in a cloc which is used as a parameter in shift_t.

If individual displacements are used, a displacement for each axis is listed. From 1 to 6 displacements can be listed, but only in the order X, Y, Z, roll, pitch, yaw. A displacement of 0.0 value can be used as a placeholder in the list.

#### cloc type

**Syntax**

```
command shift_t( var gloc location, cloc displacement_amount )
```

**Parameter**
- `location` the location to be shifted: a cloc
- `displacement_amount` the amounts of the shift, in current units: a cloc

**Example**

```
teachable cloc place
cloc difference_a = (0.0, 0.0, 10.0, 0.0, 45.0, 0.0)
...
shift_t( place, difference_a)
```

**Example**

```
teachable cloc place
cloc difference_b
float[6] b = {10.0, 0.0, 0.0, 0.0, 45.0, 0.0}
...
difference_b = {b[0], b[1], b[2], b[3], b[4], b[5]}
shift_t( place, difference_b)
```

#### displacements

**Syntax**

```
command shift_t( var gloc location, float x, [float y, [float z, 
```

**Parameter**
- `location` the location to be shifted: a cloc
- `x` the displacement along the X axis, in current units: a float
- `y` the displacement along the Y axis, in current units: a float
- `z` the displacement along the Z axis, in current units: a float
- `yaw` the displacement around the Z axis, in degrees: a float
- `pitch` the displacement around the Y axis, in degrees: a float
- `roll` the displacement around the X axis, in degrees: a float

**Example**

```
teachable cloc place
...
shift_t( place, 0.0, 0.0, 10.0, 0.0, 45.0, 0.0)
...
shift_t( place, 0.0, 0.0, -10.0)
```

**Example**

```
teachable cloc place
float displace = 2.5
...
shift_t(place, 0.0, displace)
...
displace = displace + 2.5
shift_t(place, 0.0, displace)

Returns
Success >= 0
Failure < 0

Application Shell
Same as tshift

See Also
shift_w shifts a location in the world frame of reference
tool_set sets a tool transform
base_set sets a base offset

Category
Location: Data Manipulation

**shift_w**

**Description**
In the world frame of reference, alters the cartesian coordinates of a location. A precision location cannot be changed with this command. There are two possible formats: using a cloc type or using individual displacements. In both formats, the first argument is the location to be shifted.

If a cloc type is used, the displacement values are earlier stored in a cloc which is used as a parameter in shift_w.

If individual displacements are used, a displacement for each axis is listed. From 1 to 6 displacements can be listed, but only in the order X, Y, Z, X-rotation, Y-rotation, Z-rotation. A displacement of 0.0 value can be used as a placeholder in the list.

**cloc type**

**Syntax**
command shift_w( var gloc location, cloc displacement_amount )

**Parameter**
location the location to be shifted: a cloc

displacement_amount the amounts of the shift, in current units: a cloc

**Example**
teachable cloc place
cloc difference_a = (0.0, 0.0, 20.0, 0.0, 45.0, 0.0)
...
shift_w(place, difference_a)

teachable cloc place
cloc difference_b
float[6] b = {0.0, 0.0, 0.0, 0.0, 0.0, 0.0)
...
difference_b = {b[0], b[1], b[2], b[3], b[4], b[5]}  
shift_w(place, difference_b)
...
difference_b = {b[0], b[1], b[2], b[3], b[4], b[5]}
shift_w(place, difference_b)

displacements

**Syntax**
command shift_w( var gloc location, float x, [float y, [float z, 
    [float z-rot, [float y-rot, [float x-rot, 


### shutdown

**Description**

Shuts down the pendant subsystem.

This command differs from pendant_close() which closes the pendant in preparation for shutting down a program or the controller.

**Library**

`stp`

**Syntax**

```plaintext
export command shutdown()
```

**Parameter**

None

**Returns**

- **Success** >= 0
- **Failure** < 0

**Example**

```plaintext
stp:startup()
;...
stp:shutdown()
```

**RAPL-II**

Same as PENDANT OFF

**See Also**

- `pendant_close`

**Category**

Pendant

### sig_arm_set

**Description**

Set the signal which will be issued to the controlling process in the event of an arm state change. Signals are listed in the Appendices

**Syntax**

```plaintext
command sig_arm_set( int signal )
```
### sig_arm_set

**Description**
Sets the current process’s signal mask, and returns the old one. If the bit corresponding to a given signal is 1, then that signal is ignored. All signals except SIGKILL are maskable. Signals are listed in the Appendices.

**Syntax**
```
func int sig_mask_set( int mask )
```

**Parameter**
- `mask` an int defines the signal mask

**Returns**
- Success >= 0
- Failure < 0

**Example**
```
int mask, old_mask
...
old_mask = sig_mask_set(-1)
mask = sigmask(SIGHUP)|old_mask
sig_mask_set(mask)
...
old_mask = sig_mask_set(-1)
mask = old_mask & ~ (sigmask(SIGHUP)|sigmask(SIGINT))
sig_mask_set(mask)
```

**See Also**
- sigarm_set Set the signal for change in arm power status

**Category**
Signals

### sigfifo

**Description**
Sends the signal `sig` to all of the readers at the other end of the fifo `fd`. The different types of signals are found in the Appendix.

**Syntax**
```
command sigfifo( int fd, signal_code sig )
```

**Parameters**
- `fd` an int identifies the fifo
- `sig` an enumerated type specifying the signal. The integer corresponding to the signal is listed in the Appendices.

**Returns**
- Success >= 0
- Failure < 0

**Example**
```
signal_code sig = 13 ;; SIG_13 to notify impending closure
int fd, check
string[32] thisfifo = "this_device.txt"
open(fd, thisfifo, O_RDWR | O_CREAT, M_READ | M_WRITE)

;;Prepare to close fd
check = sigfifo(fd, sig)
```
**sigmask**

Description
Returns the correct mask for the signal `sig`, which is used in conjunction with `sig_mask_set`.

Syntax
```c
func int sigmask( signal_code sig )
```

Parameter
`sig` signal_code enumerated type specifies the signal (see Appendix)

Returns
Success >= 0
Failure < 0

Example
```c
int mask, old_mask
...
old_mask = sigsetmask(-1)
mask = sigmask(SIGHUP)|old_mask
sigsetmask(mask)
...
old_mask = sigsetmask(-1)
mask = old_mask & ~ (sigmask(SIGHUP)|sigmask(SIGINT))
sigsetmask(mask)
```

See Also
signal
sigmask
sigfifo

Category
Signals

**signal**

Description
Sets an action that is to be performed whenever the current process receives signal `sig`. `sigsub` is the address of a subroutine which takes 1 integer parameter, (signal number `sig`). If `oldsigsub` is not NULL, then `oldsigsub` is set to the previous handler’s routine. If `sigsub` is NULL, then the default action is given to the signal.

Syntax
```c
command signal( signal_code sig, void@ sigsub, void@@ oldsigsub )
```

Returns
Success >= _-EOK
Failure < -EINVAL bad signal code

Example
```c
sub on_HUP( int sig )
    print ("Got SIGHUP!\n")
end sub

main
    signal( SIGHUP, on_HUP, NULL )
end main
```

Category
Signals
**sigsend**

**Description**
Sends the signal *sig* to the process specified in *pid*. Valid signals are listed in the Appendix.

**Syntax**
```
command sigsend( int pid, signal_code sig )
```

**Returns**
Success >= 0  
Failure < 0

**Example**
```
int pid ...
  pid = split() ...
  if (... & pid==0)
    sigsend (pid, SIGHUP) ;; Stop the child process
end if
```

**Category**
Signals  
System Process Control: Operating System Management

---

**sin**

**Description**
Calculates the sine of an angle. Takes an argument in degrees.

**Syntax**
```
func float sin( float x )
```

**Parameters**
- *x*  
  a float angle in degrees

**Returns**
Success >= 0. The sine of the argument.  
Failure < 0

**Example**
```
float x = 25.0 ;; value is 25.0 degrees
float y
  y = sin( x )
```

**Result**
```
y is 0.422618
```

**See Also**
- cos  
  calculates the cosine  
- tan  
  calculates the tangent  
- asin  
  calculates the arc sine

**Category**
Math

---

**size_to_bytes**

**Description**
Converts the output of sizeof() (which is the number of RAPL-3 words occupied by a data structure) to the corresponding number of bytes. It is typically used with binary data files and seek() (which expects a byte offset) for seeking to a specified record in the file.

**Syntax**
```
func int size_to_bytes(int words)
```

**Returns**
Success >= 0  
Failure < 0 (-ve error code)
Example

;; if fd is an open data file full if mystruct records,
;; this seeks to the third record in the file:
seek(fd, size_to_bytes(2 * sizeof(mystruct)), SEEK_SET)

See Also

seek(), sizeof()

Categories

File Input and Output.

---

**sizeof**

**Description**

The `sizeof()` operation is built in to the RAPL-3 compiler, and returns the size, in RAPL-3 words, of its argument. It differs from ordinary functions in that it does not require a value as its argument; instead it can accept any variable or any type.

**Syntax**

\[ \text{sizeof(any data object or type)} \]

**Returns**

the number of words occupied by the data object, or the number of words a data object of the specified type would occupy.

**Example**

if we have:

```plaintext
int x
int[10] y
ploc@ pp
string[10] s
string[100]@ sp
```
then:

```plaintext
sizeof(int) returns 1
sizeof(float) returns 1
sizeof(ploc) returns 9
sizeof(int[20]) returns 20
sizeof(float[2,5]) returns 10
sizeof(string[10]) returns 4
sizeof(string[100]) returns 26
sizeof(x) returns 1
sizeof(pp) returns 1
sizeof(pp@) returns 9
sizeof(y) returns 10
sizeof(y[x]) returns 1
sizeof(s) returns 4
sizeof(sp@) returns 26
```

See Also

`sizeof_str()`

Category

File Input and Output

String Manipulation

---

**snprint**

**Description**

Writes the specified data into the string `buf`, up to a maximum of `maxlen` characters. Two types of arguments can be given in the variable argument list: constants and variables. The constants are printed exactly as they are given. The variable’s value is what is copied to the file descriptor. The method used in printing is to print the arguments in the exact order that they were given.

**Syntax**

\[ \text{command snprint ( var string[] buf, int maxlen,... )} \]

**Parameters**

`buf` a string - the write destination
`maxlen` an int - the maximum number of characters written

**Returns**

Success >= 0
Failure < 0
Example

```c
#define MAXLEN 128
int speed, check

string[MAXLEN] store

check = speed_get(speed)
snprintf(store, MAXLEN, "Current speed is: ", speed)
printf("\{128\}\n", store)
```

Result

Current speed is: “speed”

RAPL-II
ENCODE

See Also
snprintf

Category
File Input and Output: Unformatted Output

---

**snprintf**

`string number print formatted`

**Description**

Converts and writes output into the string `buf` to a maximum length of `maxlen` under the control of a specified format `fmt`.

Format specifications are detailed in the Formatted Output section of File and Device Input and Output

**Syntax**

```c
command snprintf( var string[] buf, int maxlen, var string[] fmt, ... )
```

**Parameters**

- `buf` - a string - the write destination
- `maxlen` - an int - the maximum number of characters written

**Returns**

Success >= 0
Failure < 0

**Example**

```c
#define MAXLEN 128
int speed, check

string[MAXLEN] store

check = speed_get(speed)
snprintf(store, MAXLEN, "Current speed is:{4} m/s", speed)
printf("\{128\}\n", store)
```

Result

Current speed is: “speed” m/s

RAPL-II
ENCODE

See Also
snprintf

Category
File Input and Output: Formatted Output

---

**socketpair**

**Description**

Gets a pair of file descriptors for a private client and server socket. `client_fd` is set to the file descriptor opened as O_CLIENT, and `server_fd` is set to the file descriptor opened as O_SERVER.

**Syntax**

```c
command socketpair( var int client_fd, var int server_fd )
```

**Parameters**

- `client_fd` - an int - packed with the client file descriptor
- `server_fd` - an int- packed with the server file descriptor
Returns
Success >= 0 Returns 0.
Failure < 0
-EINVAL the arguments were invalid
-EAGAIN there are no free fd’s or related resources.

Example
int client, server
...
socketpair(client, server)

See Also
open opens a device

Category
Device Input and Output
System Process Control: Operating System Management

---

**speed**

**speed_set**

<table>
<thead>
<tr>
<th>alias</th>
<th>same as</th>
</tr>
</thead>
<tbody>
<tr>
<td>speed(...)</td>
<td>speed_set(...)</td>
</tr>
</tbody>
</table>

Description
Sets or gets the speed of arm motions. Takes an integer value. The value is the percentage (from 1 to 100) of full speed.

A value of –1 returns the current speed without changing it.

Example
speed(25) ;; sets the speed to 25%

Example
speed_now = speed_get() ;; gets the current speed
if (speed_now > 50)
speed(50)
end if

RAPL-II
Similar to SPEED.

See Also
speed_set sets the current speed
speed_get gets the current speed (can pass variable by reference)

Category
Motion

---

**speed_get**

Description
Gets the current speed setting. Can be used in two ways.

First, a parameter can be passed by reference. If a variable is used in the command call, the command packs the value of the current speed in the variable.

Second, the return value can be used. The command returns the value of the current speed. In the command call, use -1 instead of a variable.

Syntax
command speed_get(var int currspeed)

Parameter
currspeed: the variable to store the current speed setting: an int

Returns
Success >= 0
currspeed has the value of the current speed
returns the current speed value

Failure < 0

Example
int cspeed
...
speed_get(cspeed) ;; parameter passed by reference
if (cspeed > 50)
    speed_set(50)
end if

Example:
int cspeed
...
cspeed = speed_get( -1 ) ;; assign the return value

RAPL-II
Similar to SPEED.
See Also
speed_set sets the speed
Category
Motion

speed_set

Alias

<table>
<thead>
<tr>
<th>alias</th>
<th>same as</th>
</tr>
</thead>
<tbody>
<tr>
<td>speed(...)</td>
<td>speed_set(...)</td>
</tr>
</tbody>
</table>

Description
Sets the speed for all subsequent motions. Takes an integer value. The value is the percentage (from 1 to 100) of full speed.

Syntax
command speed_set( int newspeed )

Parameter
newspeed the new speed setting: an int

Returns
Success >= 0 the speed is set to newspeed
Failure < 0

Example
speed_set(10)
...
speed_set(100)

RAPL-II
Similar to SPEED.
See Also
speed_get gets the current speed setting
Category
Motion

split

Description
Creates a duplicate child process of the current process. The parent process (the one that issued the split) receives the child’s process id, and the child process receives 0.

The parent and child share all resources: text, data, and heap (entities such as open files, memory allocated at run time, outer-frame variables) except that the parent and child have separate stacks (local variables are not shared).

Syntax
func int split()

Returns
Success >= 0. The child gets returned value 0. The parent gets the (positive) child process id.
Failure < 0. No child process generated. Split returns:
–EAGAIN if the process table is full or the memory allocation tables are full
–ENOMEM if there is not enough memory for the new process’s stack

Example
int pid
...
pid = split()
if pid == 0 then
Example

```
int enable = 0
main
    string[80] cmd
    int pid
    int counter
    int result
    ...
    pid = split()
    ...
    if pid == 0 then
        ;; Child
        printf("I am the child, and my pid is {}. \n        My parent is {}.\n", getpid(), getppid())
        loop ;; forever
        result = msleep(1000)
        if enable == 1 then
            printf("Count = \n\n", counter)
            counter = counter + 1
        end if
    end loop
    else
        ;; Parent
        printf("I am the parent, and my pid is {}. \n        My child is {}.\n", getpid(), pid)
        msleep(500) ;; Give the child time to speak
        loop ;; forever
            printf("start, stop, terminate, or quit> ")
            readline(cmd,80)
            if cmd == "start" then
                enable = 1
            elseif cmd == "stop" then
                enable = 0
            elseif cmd == "terminate" then
                ;; Terminate child
                sigsend(pid, SIGHUP)
                pid = 0
            elseif cmd == "quit" then
                break
            else
                printf("I don't understand!")
            end if
        end loop
        ;; Terminate child
        if pid != 0 then
            sigsend(pid, SIGHUP)
        end if
    end if
    end main
```

**Category**  System Process Control: Single and Multiple Processes

---

**sqrt**

**Description**  Calculates the square root of a float. Takes a positive argument.

**Syntax**

```
func float sqrt( float x )
```

**Parameter**  

- `x` a float
**Returns**

Success \(\geq 0\). The square root of the argument.

Failure \(< 0\)

**Example**

```c
float x = 50.0
float y
y = sqrt( x )
```

**Result**

7.071068

**RAPL-II**

SQRT

**See Also**

pow calculates a value raised to a power

**Category**

Math

---

**srand**

**Description**

A subroutine for setting the seed value for the random number generating functions rand and rand_in.

**Syntax**

```c
sub srand(int seed)
```

**Parameters**

- `seed` an int - the seed value for random number generation

**Example**

```c
;;;Set the seed value and generate an array of 5 random numbers.

int r =5
int seed = 13
int[10] random
int j
...
srand(int seed) ;; sets the seed value rand_next = 13
;;;generate a 5 element array of random numbers
for j = 1 to r
    random[j-1] = rand()
end for
```

**Result**

A 5 element array of random number integers

**See Also**

rand_in generates random numbers within a specified range

rand generates a random number

**Category**

Math

---

**stance_get**

**Description**

Gets the current requested or physical stance of the arm. A stance is a specific configuration of one or more joints.

**Syntax**

```c
command stance_get( stance_type_t type, var shoulder_t reach, / var elbow_t elbow, var wrist_t wrist )
```

**Parameters**

- `type` enumerated type `stance_type_t`
  - STANCE_REQUESTED requested stance, not necessarily the physical stance
  - STANCE_PHYSICAL current actual stance
- `reach` enumerated type `shoulder_t` stance of shoulder, joint 2
- `elbow` enumerated type `elbow_t` stance of elbow, joint 3
- `wrist` enumerated type `wrist_t` stance of wrist, joints 4, 5, and 6

**Returns**

Success: parameters are packed.

- `reach`, one of:
  - REACH_FREE shoulder, joint 2, free (robot picks best)
  - REACH_FORWARD shoulder, joint 2, forward (toward front of robot)
  - REACH_BACKWARD shoulder, joint 2, backward
- `elbow`, one of:
  - ELBOW_FREE elbow, joint 3, free (robot picks best)
ELBOW_UP elbow, joint 3, up (away from base)
ELBOW_DOWN elbow, joint 3, down

\textit{wrist}, one of:
- WRIST_FREE joint 4 and joint 6, free (robot picks best)
- WRIST_FLIP joint 4 and joint 6 rotated 180 degrees, and joint 5 reversed
- WRIST_NOFLIP no rotation or reversal

Failure < 0

Example

\begin{verbatim}
stance_type_t mode = 0 ;; STANCE_REQUESTED
shoulder_t reach
elbow_t elbow
wrist_t wrist

stance_get( mode, reach, elbow, wrist )
if (reach != REACH_FREE || wrist != WRIST_FREE)
    reach = REACH_FREE
    wrist = WRIST_FREE
    elbow = ELBOW_FREE
    stance_set(reach, elbow, wrist)
else
    ;; Continue
end if
\end{verbatim}

Result

Returns the requested stance in the var variables reach, elbow, wrist.
If the stance is not right sets the stance.

\textbf{RAPL-II} Similar to POSE
- \texttt{REACH FORWARD|BACKWARD|XFREE}
- \texttt{ELBOW UP|DOWN|XFREE}
- \texttt{WRIST NOFLIP|FLIP|XFREE}

\textbf{See Also}
- \texttt{stance_set} sets the stance of the robot

\textbf{Category}
- Stance

\textbf{stance_set}

\textbf{Description}

Specifies a stance of the arm. A stance is a specific configuration of one or more joints.

\textbf{Syntax}

\begin{verbatim}
command stance_set( shoulder_t reach, elbow_t elbow, wrist_t wrist)
\end{verbatim}

\textbf{Parameters}

\begin{itemize}
    \item \textit{reach}
        - \texttt{REACH_FREE} shoulder, joint 2, free (robot picks best)
        - \texttt{REACH_FORWARD} shoulder, joint 2, forward (toward front of robot)
        - \texttt{REACH_BACKWARD} shoulder, joint 2, backward
    \item \textit{elbow}
        - \texttt{ELBOW_FREE} elbow, joint 3, free (robot picks best)
        - \texttt{ELBOW_UP} elbow, joint 3, up (away from base)
        - \texttt{ELBOW_DOWN} elbow, joint 3, down
    \item \textit{wrist}
        - \texttt{WRIST_FREE} joint 4 and joint 6, free (robot picks best)
        - \texttt{WRIST_FLIP} joint 4 and joint 6 rotated 180 degrees, and joint 5 reversed
        - \texttt{WRIST_NOFLIP} no rotation or reversal
\end{itemize}

\textbf{Returns}

- Success \texttt{>= 0}
- Failure \texttt{< 0}

\textbf{Example}

\begin{verbatim}
stance_type_t mode = 0 ;; STANCE_REQUESTED
shoulder_t reach
\end{verbatim}
elbow_t elbow
wrist_t wrist

stance_get( mode, reach, elbow, wrist )
    if (reach != REACH_FREE || wrist != WRIST_FREE)
        reach = REACH_FREE
        wrist = WRIST_FREE
        elbow = ELBOW_FREE

        stance_set(reach, elbow, wrist)
    else
        ;; Continue
    end if

Result
Returns the requested stance in the var variables reach, elbow, wrist.
If the stance is not right sets the stance.

RAPL-II
Similar to POSE
    REACH FORWARD|BACKWARD|XFREE
    ELBOW UP|DOWN|XFREE
    WRIST NOFLIP|FLIP|XFREE

See Also
    stance_get gets the stance of the robot

Category
Stance

startup

Description
Initializes the pendant i/o in preparation for invoking menus. This command MUST be called before other high-level commands are invoked.

This command differs from pendant_open() which prepares the pendant for access and initializes it to defaults.

Library
stp

Syntax
export command startup()

Parameter
None

Returns
Success >= 0
Failure < 0

Example
stp:startup()

RAPL-II
Same as PENDANT ON

See Also
    pendant_open

Category
Pendant

stat

Description
Obtains information about a particular object in the file system.

Syntax
command stat( var string[] path, var c_dirent buf )

Parameter
    path a string - identifies the device
    buf c_dirent structure has the following fields:
        string[32] de_name
        int de_type
        int de_links
        mode_flags de_mode
        int de_size
        int de_mtime
Subprograms: Alphabetical Listing

int de_dev
int de_ident

The options for mode_flags type are:
- M_READ readable
- M_WRITE writable
- M_EXEC executable *

Modes may be combined with the bitwise OR operator, represented by | (a single vertical bar/pipe).
- M_READ
- M_READ | M_EXEC
- M_READ | M_WRITE
- M_READ | M_WRITE | M_EXEC

Returns
- Success >= 0 buf is packed with the data
- Failure < 0
  - EINVAL the arguments were invalid
  - ENOTDIR a component is not a directory
  - ENOENT a component was not found
  - EIO an I/O error occurred
  - EAGAIN temporarily out of resources needed to do this

Example
int fd, check
c_dirent dev_info
string[32] thisfifo = "this_device.txt"
open(fd, thisfifo, O_RDWR | O_CREAT, M_READ | M_WRITE)
... check = stat(thisfifo, dev_info )

Result
Fields of c_dirent type dev_info is packed with data

See Also
- statfs Gets information about mounted file system
- statusnp Gets status of named pipe

Category
- File and Device System Management

---

**statfs**

**Description**
Gets information about a mounted file system.

**Syntax**
```c
command statfs( var string[] path, var c_statfs buf )
```

**Parameter**
- `path` a string specifying the path to the file
- `buf` a variable of type c_statfs - the struct to hold the information:
  - `mount_type` filesystem type code, one of:
    - MOUNT_MFS memory file system
    - MOUNT_CFS CROSnt file system
    - MOUNT_RFS remote file system
    - MOUNT_HOSTFS host file system
  - `int fs_bsize` size of 1 block, in bytes
  - `int fs_free` number of free blocks

**Returns**
- Success >= 0
- Failure < 0
  - EOK success
  - EINVAL invalid argument
  - ENOTDIR a component of the path was not a directory
  - ENOENT the specified file was not found
  - EIO an I/O error occurred
  - EAGAIN temporarily out of resources needed to do this

**Example**
```c
.define PATHLEN 32
mount_type type = MOUNT_HOSTFS
```
string[PATHLEN] dir = "/app/this_app"
mount_flags flags = MOUNTF_RDONLY
c_statfs stat

int check

check = mount(type, dir, flags, NULL)
...
check = statfs(dir, stat)

Result

c_statfs type stat is packed with the data

System Shell Application Shell mount
See Also
mount mount a file system
Category
File and Device System Management

statusnp

status named pipe

Description

Returns the current status of a named pipe.
Also returns how far the pending operation has completed, or the completed transfer length.

Syntax

func int statusnp( int fd, var int nwords )

Parameter

fd the file descriptor: an int
nwords the number of words: an int

Returns

>0
the current status of the named pipe
NPIPE_OPENED 0x0001
NPIPE_CONNECTED 0x0002
NPIPE_CONNECT_PENDING 0x0100
NPIPE_READ_PENDING 0x0200
NPIPE_WRITE_PENDING 0x0400
NPIPE_TRANSACT_PENDING 0x0800
the number of words transferred thus far in the current i/o operation
the number of words in the last i/o operation
=0 no previously pending i/o operation waiting for pick-up
<0 error

Example

statusnp(pd, stat)
statusnp(NT_app_pipe, words)

RAPL-II

No equivalent.

See Also
opennp opens a named pipe
closenp closes a named pipe
connectnp connects to a named pipe
disconnectnp disconnects a client from a named pipe

Category
Win 32
### str_append

**Description**
Takes string src and appends it onto string dst. String length of dst must be of sufficient length to contain the string being appended.

**Syntax**
```
sub str_append( var string[] dst, var string[] src )
```

**Parameter**
- **dst**: a string the destination string
- **src**: string appended to string dst

**Example**
```
string[20] dst = "Name:"
...
print ( dst, "\n" )
str_append( dst, "J. Doe" )
print ( dst, "\n" )
```

**Result**
Name:
Name: J. Doe

**Category**
String Manipulation

### str_chr_find

**Description**
Finds the first occurrence of c in string src. Returns the index of the character. If not found, returns -1.

**Syntax**
```
func int str_chr_find( string[] src, int c )
```

**Parameter**
- **src**: a string
- **c**: an int - the character to be found in string src.

**Returns**
- Success >= 0
- Failure < 0

**Example**
```
.define MAXLEN 128
string[MAXLEN] indata, str, newstr
int cmd, outnum, outval,i
...
cmd=str_chr_get(indata,0) ;; find command type
case cmd
  
  of 'O': ;; O<outnum>,<state><lf> this will set outputs
     i=str_chr_find(indata,',') ;; find position of ','
     if i>=2 then
       ;; make new "str" with data <outnum>
       str_substr(str,indata,1,i-1)
       ;;convert "str" to int outnum
       str_to_int(outnum,str)
       ;; newstr is <state>
       str_substr(newstr,indata,i+1,MAXLEN)
       ;; convert newstr to int
       str_to_int(outval,newstr)
       ;; set output "outnum" to "outval"
       output_set(outnum,outval)
  end if
end case
```

**Result**
Outputs set as defined in the command line input

**RAPL-II**
STRPOS found substring (not character) in a string.

**See Also**
- str_chr_rfind

**Category**
String Manipulation
str_chr_get

Description
Returns the ASCII value of the character indexed by index in string s. Reminder: string indexes begin at 0.

Syntax
func int str_chr_get( var string s, int index )

Parameters
s a string
index an int - specifies the character in the string

Returns
Success >= 0
Failure < 0

Example
string[] s = "str_chr_get example"
...print ("Letter 'e' has ASCII value ")
ch = str_chr_get( s, 9 )
...print (ch,"\n")

Result
Letter 'e' has ASCII value 101

See Also
str_chr_find
str_chr_rfind

Category
String Manipulation

str_chr_rfind

Description
Finds the last occurrence of c in string src. Returns the index of the character. If not found, returns -1.

Syntax
func int str_chr_rfind( string[] src, int c )

Parameter
src a string, searched for the int c

c an int, the character to be located in the string src

Returns
Success >= 0 Returns the index of the last occurrence of the character c.
Failure < 0 -1 if character is not found

Example
;;Does a sentence end with proper punctuation "." or "?"
.define MAXLEN 128
string[MAXLEN] sentence
int i, length, j, count

;; prompt for sentence
printf("Enter a sentence (max 128 characters)\n")

;; Read sentence
count=readline(sentence,MALLEN)
length = str_len(sentence) ;;sentence length starts from 0
i = str_chr_rfind(sentence, ".")
j = str_chr_rfind(sentence, "?")
if i == length-1 || j == length-1 ;; proper punctuation
    printf("Good punctuation\n")
else
    printf("Sentence punctuation incorrect\n")
end if

RAPL-II
STRPOS found substring (not character) in a string

See Also
str_chr-find

Category
String Manipulation
### str_chr_set

**Description**
Sets the value of the character indexed by `index` in string `s` to `ch`. Reminder: string indexes begin with 0.

**Syntax**
```
sub str_chr_set( var string[] s, int index, int ch )
```

**Example**
```
string[] s = "str_chr_set example"
...
print (s, \"\n")
str_chr_set( s, 13, 'e' )
...
print (s, \"\n")
```

**Result**
```
str_chr_set example
str_chr_set eexample
```

**See Also**
- str_edit
- str_chr_find
- str_chr-rfind

**Category**
String Manipulation

---

### str_cksum

**Description**
Computes a 32-bit bytewise checksum of the characters of string, for characters from `start` to `start + len - 1`.

**Syntax**
```
func int str_cksum( var string[] s, int start, int len )
```

**Parameters**
- `s`: string for which the checksum is calculated
- `start`: int the start character for the checksum
- `len`: the string length for the checksum

**Returns**
- Success `>= 0`
- Failure `< 0`

**Example**
```
#define MAXLEN 128
string[MAXLEN] the_string = "What is the checksum of the_string?"
int len, check

len = sizeof(the_string)
check = str_cksum(the_string, 0, len)
printf("{} \nChecksum = {} \n", the_string, check)
```

**Result**
```
What is the checksum of the_string
Checksum = 3145
```

**Category**
String Manipulation

---

### str_dup

**Description**
Allocates space for a string, copies it into the allocated space and returns a pointer to the new string. This is principally useful for constructing dynamic data structures.

**Syntax**
```
func string[]@ str_dup(string[] str)
```

**Parameter**
- `str`: the string to allocate space for and copy.

**Returns**
- a pointer to the new string. Raises an exception if the memory allocation fails.
string[]@sp
...
sp = str_dup("This is a test string...")
printf("The new string is '()\n", sp@)

"The new string is 'This is a test string...’' is printed out.

See Also

mem_alloc()

Category

String Manipulation

---

**str_edit**

**Description**

Replaces the characters in `dst` at position `start` and `len` characters with the string `src`. This subroutine can be used to both delete characters (if `src == ''`) and insert substrings (if `len == 0`, for example.) Note that if `dst` doesn’t have a `start` character, then `src` is simply appended to the end of `dst`.

**Syntax**

```c
sub str_edit( var string[] dst, string[] src, int start, int len )
```

**Parameter**

- `dst` : a string to be edited
- `src` : the string to be used to places in `dst`
- `start` : the start character index of `dst`
- `len` : the length (number) of characters to be replaced

**Returns**

Success >= 0
Failure < 0

**Example**

```c
;; Remove vowels from a string
string[128] sentence
int i = 0
int count = 0
int len

;; prompt for sentence
printf("Enter a sentence (max 128 characters)\n")

;; Read sentence
count=readline(sentence,128)
len = str_len(sentence) ;;sentence length starts from 0
;; find and remove vowels
while (i <= len) && (count != NULL)
    count= str_chr_get(sentence, i)
    if count=='a'||count=='e'||count=='i'||count=='o'||count=='u'
        str_edit(sentence,"",i,1)
    else
        i++
    end if
end while
printf("\n\n", sentence)
```

Prints the string sentence with the vowels removed.

**See Also**

str_chr_find

**Category**

String Manipulation

---

**str_error**

**Description**

Returns a pointer to a string that describes a given error code specified in `n`.  

---
A failed command or function returns a negative integer (error descriptor) which corresponds to a particular error. The message strings, corresponding to the error descriptor, are stored in a string array indexed by positive integers. The negative return value of the failed command or function must be converted to a positive value for str_error() to access the array.

Refer to the section Error Handling for a description of the error descriptor and the error codes.

**Syntax**

```
func string[]@ str_error( int n )
```

**Parameters**

- `n` - an int error descriptor

**Returns**

- Success >= 0
- Failure < 0

**Example**

```java
int t, fd
...
t = open(fd, "myfile", O_RDONLY, 0)
if (t < 0) ;; error
    print("Error is:", str_error(-t), "\n")
... exit(1)
end if
```

**Result**

Error is: not found

**See Also**

- str_signal returns a pointer to a string describing a signal code

**Category**

String Manipulation

**Error Message Handling**

---

**str_len**

**Description**

Returns the length of string `s` or 0 (zero) if no limit. Reminder: the length is different from the initial declared size.

**Syntax**

```
func int str_len( var string[] s)
```

**Parameter**

- `s` - a string

**Returns**

- Success >= 0
  - positive, the size of the string
  - zero, no limit
- Failure < 0

**Example**

```java
string[20] s = "str_len example"
int i
...
i = str_len(s)
print (i, "\n")
```

**Result**

15

**See Also**

- str_limit Returns string limit

**Category**

String Manipulation

---

**str_len_set**

**Description**

Sets the length of string `s` to `len`. This subroutine is equivalent to truncating a string to length `len`, if `s` is longer than `len` and extending a string `s` to length `len`, if `s` is shorter than `len`. 
Length, *len*, of 0 (zero) allows any length. This is useful with dynamic allocation where length is controlled by `mem_alloc()`.

**Syntax**

```c
sub str_len_set( var string[] s, int len )
```

**Example**

```c
string[] s = "str_len_set example"
...  
print (s, "\n")
str_len_set ( s, 11 )
print (s, "\n")
```

Result

```
str_len_set example
```str_len_set

**See Also**

- `str_len`
- `str-limit`

**Category**

String Manipulation

---

**str_limit**

**Description**

Returns the limit on the length of a string.

**Syntax**

```c
func int str_limit( var string[] s)
```

**Parameter**

- `s` A string

**Returns**

- Success >= 0 Returns integer value of the string length limit.
- Failure < 0

**Example**

```c
#define MAXLEN 128

string[MAXLEN] sentence = "This is a string"
int length

length = str_limit(sentence)
printf("str_limit is {}
",length)
```

Result

```
str_limit is 128
```str_limit

**See Also**

- `str_len` actual string length

**Category**

String Manipulation

---

**str_limit_set**

**Description**

Sets the limit on the length of a string.

**Syntax**

```c
sub str_limit_set( var string[] s, int len )
```

**Parameter**

- `s` A string
- `len` an int the limit for the string

**Returns**

- Success >= 0
- Failure < 0

**Example**

```c
#define MAXLEN 128

string[MAXLEN] sentence = "This is a string"
int length =32
int len

str_limit_set(sentence, length)
len = str_limit(sentence)
printf("str_limit is {}
",len)
```

Result

```
str_limit is 32
```str_limit_set
str_scanf

**string scan formatted**

**Description**
Parses (separates) the contents of string s according to fmt into a list of pointers to variables. Returns the number of items matched. Scanning may stop before the end of s if str_scanf() runs out of format specifiers.

**Syntax**
```plaintext
command str_scanf ( var string[] s, var string fmt, ... )
```

**Parameters**
The string fmt can contain:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>{}</td>
<td>(opening brace and closing brace)</td>
</tr>
<tr>
<td>{10F}</td>
<td>fixed field of 10 characters wide (no extra whitespace before or after)</td>
</tr>
<tr>
<td>(10)</td>
<td>an item of given maximum width (not fixed; whitespace ignored)</td>
</tr>
<tr>
<td></td>
<td>(blank space)</td>
</tr>
<tr>
<td>\</td>
<td>space means 0 or more spaces</td>
</tr>
<tr>
<td></td>
<td>(two backslashes)</td>
</tr>
<tr>
<td>,</td>
<td>means exactly 1 comma</td>
</tr>
<tr>
<td>x</td>
<td>means exactly 1 of that character</td>
</tr>
</tbody>
</table>

**Returns**

- Success >= 0
- Failure < 0

**Example 1**
```plaintext
str_scanf ( buf, "{}{} {}", &intvar1, &intvar2, &floatvar )
```
will scan for:
- any whitespace
- an integer (stored in intvar1)
- any whitespace
- an integer (stored in intvar2)
- any whitespace
- a float (stored in floatvar)
- any whitespace

**Example 2**
```plaintext
str_scanf ( buf, "{}{20}, {}", &stringvar, &intvar )
```
will scan for:
- any whitespace
- a non-whitespace string (first 20 chars stored in stringvar)
- any whitespace
- a comma
- any whitespace
- an integer (stored in intvar)
- any whitespace

**Example 3**
```plaintext
str_scanf ( buf, "{10F},{10F},{20F} ", &floatvar, &intvar, &stringvar )
```
will scan for:
- exactly 10 characters to be converted to a float and stored in floatvar
- exactly 1 comma
- exactly 10 characters to be converted to an int and stored in intvar
- exactly 1 comma
- exactly 20 characters to be converted to a string and stored in stringvar
- any amount of whitespace
**str_signal**

**Description**
Returns a pointer to a string that describes a given signal code specified in $n$. Valid signal codes are found in the Appendix.

**Syntax**
```c
func string[] @ str_signal( int n )
```

**Parameter**
$n$ an int specifies the signal number

**Returns**
Success $\geq 0$
Failure $< 0$

**Example**
```c
string[] @ sig_msg ...
sig_msg = str_signal( SIGHUP )
print ( sig_msg, "\n" )
```

**Result**
SIGHUP

**RAPL-II**
No equivalent.

**See Also**
str_error returns a pointer to a string describing an error code

---

**str_sizeof**

**Description**
Returns the number of words it takes to store a string of length $n$.

**Syntax**
```c
func int str_sizeof( int n )
```

**Parameters**
$n$ an int the size of the string (# of characters)

**Returns**
Success $\geq 0$. Returns $1 + ( (n + 3) >> 2 )$
Failure $< 0$

**Example**
```c
int size, max_size
int words, max_words
string[128] gnirts = "How much memory to store this string"

size = str_len(gnirts)
max_size = str_limit(gnirts)

words = str_sizeof(size)
max_words = str_sizeof(max_size)

printf( "memory for string is: \n", words )
printf( "max memory for string is: {} \n", max_words )
```

**Result**
memory for string is 10
max memory for string is 33

**See Also**
str_limit
str_limit_set

**Category**
String Manipulation
**str_substr**

Description
Copies the substring of src starting at the start<sup>th</sup> character and len characters long into dst. Only as much of the substring as actually exists is copied. Characters are numbered from 0.

Syntax
```
sub str_substr( var string[] dst, string[] src, int start, int len )
```

Parameter
- **dst** the destination string
- **src** the source string
- **start** an int the start point in the src string
- **len** an int the length to be copied

Returns
- Success >= 0
- Failure < 0

Example
```
str_substr(d,s,0,10)
;; copies the first 10 characters of s into d.
```

See Also
str_edit

Category
String Manipulation

**str_subsys**

Description
The str_subsys function, given a specific error descriptor returns a string giving the name of the subsystem origination the error. For details on the error descriptor refer to the Error Handling section.

Syntax
```
func string[]@ str_subsys(int descriptor)
```

Parameters
- **descriptor** an int - value returned when error occurs in subprogram

Returns
- Success >= 0 Returns a string with specifying the subsystem.
- Failure < 0

Example
```
int t, err_des

err_des = -t...
t = open(fd, "myfile", O_RDONLY, 0)
if (t < 0) ;; error
    printf("The error occurred in the {} subsystem \n", str_subsys(err_des))
    exit(1)
end if
```

Result
The error occurred in the [kernel] subsystem

See Also
err_get_subsys
str_error

Category
Error Message Handling
String Manipulation

**str_to_float**

Description
Converts an ASCII string in src to a floating point number and places the result in dst. If the string is not a proper floating point number, the command fails.

Syntax
```
command str_to_float( var float dst, var string[] src )
```
**str_to_float**

**Description**
Converts string `src` to a float value.

**Syntax**
```
str_to_float( var float dst, var string[] src )
```

**Parameters**
- `dst`: a float - the value of the string `src`
- `src`: a string - string to be converted to a float value

**Returns**
- Success >= 0
- Failure < 0

**Example**

```plaintext
string[] s = "12345.67"
f ...
str_to_float (f, s)
print (f, "\n")
```

**Result**
12345.67

**Category**
String Manipulation
Math

---

**str_to_int**

**Description**
Converts string `src` into a hexadecimal integer if there is a leading 0x or 0X, octal integer if there is a leading 0, or decimal integer otherwise. Stores the result in `dst`. LONG_MAX or LONG_MIN are stored if overflow occurred, depending on the sign of the value.

**Syntax**
```
str_to_int( var int dst, var string[] src )
```

**Parameters**
- `dst`: an int - the value of the string `src`
- `src`: a string - string to be converted to a integer value

**Returns**
- Success >= 0
- Failure < 0
- EINVAL if error occurred during conversion.

**Example**

```plaintext
string[] s = "12345"
i ...
str_to_int (i, s)
print (i, "\n")
```

**Result**
12345

**Category**
String Manipulation
Math

---

**str_to_lower**

**Description**
For a string specified by the variable `str`, converts the letters in the string from upper case to lower case. If a letter is already lower case, does not change it.

**Syntax**
```
str_to_lower( var string[] str )
```

**Parameter**
- `str`: the string to be converted: a variable length string

**Example**

```plaintext
string[128] path = "MY_DIRECTORY\MY_FILE"
...
str_lower(path)
printf("\n", path)
```

**Result**
my_directory\my_file
See Also

- `str_to_upper` : converts a string to upper case
- `chr_to_lower` : converts a character to lower case

Category

String Manipulation

---

**str_to_upper**

**Description**

For a string specified by the variable `str`, converts the letters in the string from lower case to upper case. If a letter is already upper case, does not change it.

**Syntax**

```c
sub str_to_upper( var string[] str )
```

**Parameter**

- `str` : the string to be converted: a variable length string

**Example**

```c
sentence = "emphasis here"
str_to_upper(sentence)
printf("{}\n",sentence)
```

**Result**

EMPHASIS HERE

See Also

- `str_to_lower` : converts a string to lower case
- `chr_to_upper` : converts a character to upper case

Category

String Manipulation

---

**sync**

**Description**

Flushes all the file system buffers of their contents.

**Syntax**

```c
command sync()
```

**Returns**

Commands do not return a value

**Example**

```c
int fd
string[] buffer = "sync test"
...
open( fd, "filename", O_WRONLY, 0 );; Open file
fprint( fd, buffer ) ;; Write value
sync() ;; Force writing
```

Category

File and Device System Management

---

**sysconf**

**Description**

Obtains system configuration information and places it in a struct (c_sysconf). The data is a struct of ints, 32 bit numbers. The `sc_items` parameter must be initialized to indicate how many items to transfer/accept.

The `sysid_string()` command is used to print the system identifier.

**Syntax**

```c
command sysconf( var c_sysconf scp )
```

**Parameter**

- `scp` : the system configuration data: a struct of type `c_sysconf`
  - `int sc_items` : number of entries to transfer/accept
  - `int sc_sysid` : system identifier word
  - `int sc_version` : version code, major.minor where major == upper 16 bits
  - minor == lower 16 bits
  - `int sc_click_size` : bytes per click
int sc_msec_per_tick  milliseconds per scheduled tick
int sc_build

Returns
Success >= -EOK  success
Failure < 0  -EINVAL  the argument was invalid (improperly initialized buffer)

Example
    c_sysconf sysconf_buf
    int[4] datain
    int[8] dataout
    int value
    ...
    sysconf_buf.sc_items = sizeof(sysconf_buf)
    sysconf(sysconf_buf)
    ...
    print("\nSystem type: ", sysid_string(sysconf_buf.sc_sysid), "\n")
    print("Version: ", (sysconf_buf.sc_version >> 16), ".", \
         (sysconf_buf.sc_version & 0xffff), ".", \
         sysconf_buf.sc_build, "\n")
    print("Click size: ", sysconf_buf.sc_click_size, "\n")
    print("msec/tick: ", sysconf_buf.sc_msec_per_tick, "\n")
    ...

Category
System Process Control: Operating System Management

\**sysid_string**

Description
Returns a string describing a specified system id.

Syntax
func string[]@ sysid_string( int sysid )

Parameter
- **sysid**  an int - specifies the system

Returns
Success >= 0.
- Returns 1  CROS on a C500
- Returns 2  CROS on a C500B
- Returns 3  CROS on a C600
- Returns 4  CROS under Windows NT
- Returns 5  CROS under MSDOS

Failure < 0

Example
    c_sysconf sysconf_buf
    int[4] datain
    int[8] dataout
    int value
    ...
    sysconf_buf.sc_items=sizeof(sysconf_buf)
    sysconf(sysconf_buf)
    ...
    print("\nSystem type: ", sysid_string(sysconf_buf.sc_sysid), "\n")
    print("Version: ", (sysconf_buf.sc_version >> 16), ".", \
         (sysconf_buf.sc_version & 0xffff), ".", \
         sysconf_buf.sc_build, "\n")
    print("Click size: ", sysconf_buf.sc_click_size, "\n")
    print("msec/tick: ", sysconf_buf.sc_msec_per_tick, "\n")

Category
System Process Control: Operating System Management

\**tan**

Description
Calculates the tangent of an angle. Takes an argument in degrees.
func float tan(float x)

Parameter
x a float - angle in degrees

Returns
Success >= 0. The tangent of the argument.
Failure < 0

Example
float x = 65.0 ;; value is in degrees
float y
y = tan(x)

Result
2.144507

RAPL-II
TAN

See Also
cos calculates the cosine
sin calculates the sine
atan2 calculates the arc tangent

Category
Math

teach_menu

Description
Use this command to select and teach variables for an application. Note that you
cannot use this command unless there is an open v3 file.

Library
stp

Syntax
export sub teach_menu()

Parameter
None

Returns
Success >= 0
Failure < 0

Example
stp:teach_menu()

Category
Pendant

time

Description
Returns the current calendar time, or -1 if the time is not available. The calendar
time is given as a 32 bit integer and represents the number of elapsed seconds
since the beginning of Jan. 1, 1970.

Syntax
func int time()

Returns
Success >= 0 Returns the time
Failure < 0 -1

Example
int t

t = time()
print (t, "\n")

Result
834539842

See Also
time-set sets the current time
time_to_str converts a system time code to an ASCII string

Category
Date and Time

time_set

Description
Sets the current time to the calendar time contained in now. The calendar time
represents the elapsed number of seconds since the beginning of Jan. 1, 1970.

Syntax
command time_set(int now)
Parameter | now | an int - calendar time
---|---|---
Returns | Success >= 0 | Failure < 0
 | -EOK | success
Example | int t
 | t = time() ;; Get the current system time
 | t = t - 24 * 3600 ;; Set the time back to
 | ;; same time yesterday
 | time_set (t)
See Also | time | returns the current calendar time
 | time_to_str | converts a system time code to an ASCII string
Category | Date and Time

**time_to_str**

Description | Converts a system time code to an ASCII string of the form:
 | Day Mth DD HH:MM:SS YYYY
For example, time = 836211600 returns
 | Mon Jul 1 09:00:00 1996
The result is stored in dst, which must have space for at least 25 characters.
Syntax | command time_to_str(var string[] dst, int time)
Parameter | dst | a string for storing date and time
 | time | an int the system time
Returns | Success >= 0 | Failure < 0
Example | int check
 | int time = 836211600
 | string[128] time_date
 | check = time_to_str(time_date, time)
 | printf("\n",time_date)
Result | Mon Jul 1 09:00:00 1996
See Also | set_time | sets the current time
 | time | returns the current calendar time
Category | Date and Time
 | String Manipulation

**tool_get**

Description | Gets the current tool transform, the redefinition of the origin point and the orientation of the tool coordinate system.
The default origin is the centre of the surface of the mechanical interface (tool flange). The transform has translational coordinates, x, y, and z, and rotational coordinates, yaw, pitch, and roll. The data type used is a cloc which also has an integer flag.
Syntax | command tool_get(var cloc toolloc )
Parameter | toolloc | a cloc packed with the tool transform data
Subprograms: Alphabetical Listing

Returns

Success >= 0  
   toolloc is packed with current transform data
Failure < 0

Example

   teachable cloc tool_trsfrm
   cloc old_tool

   tool_get(old_tool)
   if old_tool != tool_trsfrm
      tool_set(tool_trsfrm)
   end if

Result

Tool transform is set to the teachable cloc “tool_trsfrm”

RAPL-II

   Similar to TOOL

See Also

   tool_set re-defines the current tool offset
   base_get gets the current base offset

Category

   Tool Transform and Base Offset

---

tool_set

Description

Sets a tool transform, a redefinition of the origin point and the orientation of the
tool coordinate system.

The default origin is the centre of the surface of the mechanical interface (tool flange).

The tool_set() command has the capacity for a 6 degree-of-freedom transformation. The origin can be re-defined by translational coordinates: x, y, and z. The orientation can be re-defined by rotational coordinates: yaw, pitch, and roll. A cloc data type is used which requires an integer constant flag followed by float constant coordinates.

Syntax

   command tool_set( var cloc toolloc )

Parameter

   toolloc  
      the transform with flag, x, y, z, yaw, pitch, roll information: a cloc
   flag  
      the *: an int
   x  
      the distance along the X axis, in current units: a float
   y  
      the distance along the Y axis, in current units: a float
   z  
      the distance along the Z axis, in current units: a float
   yaw  
      the rotation around the Z axis, in degrees: a float
   pitch  
      the rotation around the Y axis, in degrees: a float
   roll  
      the rotation around the X axis, in degrees: a float

Returns

   Success >= 0
   Failure < 0

Example

   tool_set( 0, 2.0, 0.0, 3.0, 0.0, 90.0, 0.0 )
   ;; for a tool with a tool centre-point 2.0 units along the X axis
   ;; and 3.0 units along the Z axis from the default origin
   tool( 0, 2.0, 0.0, 3.0, 0.0, 90.0, 0.0 )
   ;; for the same tool as the previous example oriented with
   ;; a 90 degree pitch

RAPL-II

   Similar to TOOL.

See Also

   tool_get gets the current tool offset
   shift_t alters coordinate(s)/orientation(s) in the tool frame of reference
   base_set re-defines the world coordinate system

Category

   Tool Transform and Base Offset
tx

Jog_t ...

<table>
<thead>
<tr>
<th>alias</th>
<th>same as</th>
</tr>
</thead>
<tbody>
<tr>
<td>tx(...)</td>
<td>jog_t(TOOL_X, ...)</td>
</tr>
</tbody>
</table>

Description

In the tool frame of reference, moves the tool centre point to the end point which is a specified distance along the X axis, in current units (millimetres or inches).

The following table describes the positive X axis for each tool coordinate system.

<table>
<thead>
<tr>
<th>arm position</th>
<th>F3 coordinate system</th>
<th>A465/A255 coordinate system</th>
</tr>
</thead>
<tbody>
<tr>
<td>any</td>
<td>(see below)</td>
<td>X is perpendicular to (arises out of) the tool flange.</td>
</tr>
<tr>
<td>ready</td>
<td>X is vertical pointing down parallel to negative world Z.</td>
<td>X is horizontal, pointing ahead, past the front of the arm, parallel to world X.</td>
</tr>
<tr>
<td>straight up</td>
<td>X is horizontal, pointing ahead, past the front of the arm parallel to world X.</td>
<td>X is vertical pointing up parallel to world Z.</td>
</tr>
</tbody>
</table>

This command, tx(), is joint-interpolated. The tool centre point travels as a result of various joint motions, not in a straight line.

For cartesian-interpolated (straight line) motion, see txs().

Syntax

command tx( float distance )

Parameters

distance the distance of travel, in current units: a float

Returns

Success = 0
Failure < 0

Example

move(base_point)
tx(200) ;; millimetres

RAPL-II

No equivalent.

See Also

txs jogs like tx, but in straight line motion
jog_t alias of tx and moves along other axes
ty jogs like tx, but along Y axis
tz jogs like tx, but along Z axis
depart moves along approach/depart axis
jog_w jogs like tx, but in world frame of reference

Category

Motion

txs

Jog_ts ...

<table>
<thead>
<tr>
<th>alias</th>
<th>same as</th>
</tr>
</thead>
<tbody>
<tr>
<td>txs()</td>
<td>jog_ts(TOOL_X, ...)</td>
</tr>
</tbody>
</table>

Description

In the tool frame of reference, moves the tool centre point along the X axis by the specified distance in current units (millimetres or inches).

The following table describes the positive X axis for each tool coordinate system.

<table>
<thead>
<tr>
<th>arm position</th>
<th>F3 coordinate system</th>
<th>A465/A255 coordinate system</th>
</tr>
</thead>
</table>
### Subprograms: Alphabetical Listing

<table>
<thead>
<tr>
<th>Subprogram</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>any</td>
<td>(see below)</td>
</tr>
<tr>
<td>ready</td>
<td>X is vertical pointing down parallel to negative world Z.</td>
</tr>
<tr>
<td>straight up</td>
<td>X is horizontal, pointing ahead, past the front of the arm, parallel to world X.</td>
</tr>
</tbody>
</table>

This command, t(x)s(), is cartesian-interpolated (straight line).

For joint-interpolated (not straight) motion, see tx().

#### Syntax

```
command txs( float distance )
```

#### Parameters

- **distance**: the distance of travel, in current units or degrees: a float

#### Returns

- **Success = 0**
- **Failure < 0**

#### Example

```
move(base_point)
txs(200) ;; millimetres
```

#### RAPL-II

No equivalent.

#### See Also

- **tx**: jogs like txs, but joint interpolated
- **jog_ts**: alias of txs and moves along other axes
- **tys**: jogs like txs, but along Y axis
- **tzs**: jogs like txs, but along Z axis
- **depart**: moves along approach/depart axis
- **jog_ws**: jogs like txs, but in world frame of reference

#### Category

Motion

---

### ty

#### Alias

`jog_t ...`

<table>
<thead>
<tr>
<th>alias</th>
<th>same as</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ty(...)</code></td>
<td><code>jog_t(TOOL_Y, ...)</code></td>
</tr>
</tbody>
</table>

#### Description

In the tool frame of reference, moves the tool centre point to the end point which is a specified distance along the Y axis, in current units [millimetres or inches].

The following table describes the positive Y axis for each tool coordinate system.

<table>
<thead>
<tr>
<th>arm position</th>
<th>F3 coordinate system</th>
<th>A465/A255 coordinate system</th>
</tr>
</thead>
<tbody>
<tr>
<td>any</td>
<td>(see below)</td>
<td>(see below)</td>
</tr>
<tr>
<td>ready</td>
<td>Y is horizontal, pointing out to one side of the arm, parallel to positive world Y.</td>
<td>Y is horizontal, pointing out to one side of the arm, parallel to positive world Y.</td>
</tr>
<tr>
<td>straight up</td>
<td>Y is horizontal, pointing out to one side of the arm, parallel to positive world Y.</td>
<td>Y is horizontal, pointing out to one side of the arm, parallel to positive world Y.</td>
</tr>
</tbody>
</table>

This command, t(y)(), is joint-interpolated. The tool centre point travels as a result of various joint motions, not in a straight line.

For cartesian-interpolated (straight line) motion, see tys().

#### Syntax

```
command ty( float distance )
```

#### Parameters

- **distance**: the distance of travel, in current units: a float
Returns
Success = 0
Failure < 0

Example
move(base_point)
  ty(200) ;; millimetres

RAPL-II
No equivalent.

See Also
tys jogs like ty, but in straight line motion
jog_t alias of tys and moves along other axes
tx jogs like tys, but along X axis
tz jogs like tx, but along Z axis
depart moves along approach/depart axis
jog_w jogs like tys, but in world frame of reference

Category
Motion

tys
Alias
jog_ts ...

<table>
<thead>
<tr>
<th>alias</th>
<th>same as</th>
</tr>
</thead>
<tbody>
<tr>
<td>tys(...)</td>
<td>jog_ts(TOOL_Y, ...)</td>
</tr>
</tbody>
</table>

Description
In the tool frame of reference, moves the tool centre point along the Y axis by the
specified distance in current units (millimetres or inches).

The following table describes the positive Y axis for each tool coordinate system.

<table>
<thead>
<tr>
<th>arm position</th>
<th>F3 coordinate system</th>
<th>A465/A255 coordinate system</th>
</tr>
</thead>
<tbody>
<tr>
<td>any</td>
<td>(see below)</td>
<td>(see below)</td>
</tr>
<tr>
<td>ready</td>
<td>Y is horizontal, pointing out to one side of the arm, parallel to positive world Y.</td>
<td>Y is horizontal, pointing out to one side of the arm, parallel to positive world Y.</td>
</tr>
<tr>
<td>straight up</td>
<td>Y is horizontal, pointing out to one side of the arm, parallel to positive world Y.</td>
<td>Y is horizontal, pointing out to one side of the arm, parallel to positive world Y.</td>
</tr>
</tbody>
</table>

This command, tys(), is cartesian-interpolated (straight line).

For joint-interpolated (not straight) motion, see ty()

Syntax
command tys( float distance )

Parameters
distance the distance of travel, in current units or degrees: a float

Returns
Success = 0
Failure < 0

Example
move(base_point)
tys(200) ;; millimetres

RAPL-II
No equivalent.

See Also
ty jogs like tys, but joint interpolated
jog_ts alias of tys and moves along other axes
txs jogs like tys, but along X axis
tzs jogs like tys, but along Z axis
depart moves along approach/depart axis
jog_ws jogs like tys, but in world frame of reference

Category
Motion
**tz**

**Alias**

<table>
<thead>
<tr>
<th>alias</th>
<th>same as</th>
</tr>
</thead>
<tbody>
<tr>
<td>tz(...)</td>
<td>jog_t(TOOL_Z, ...)</td>
</tr>
</tbody>
</table>

**Description**

In the tool frame of reference, moves the tool centre point to the end point which is a specified distance along the Z axis, in current units (millimetres or inches).

The following table describes the positive Z axis for each tool coordinate system.

<table>
<thead>
<tr>
<th>arm position</th>
<th>F3 coordinate system</th>
<th>A465/A255 coordinate system</th>
</tr>
</thead>
<tbody>
<tr>
<td>any</td>
<td>Z is perpendicular to (arises out of) the tool flange.</td>
<td>(see below)</td>
</tr>
<tr>
<td>ready</td>
<td>Z is horizontal, pointing ahead, past the front of the arm, parallel to world X.</td>
<td>Z is vertical pointing up, parallel to positive world Z.</td>
</tr>
<tr>
<td>straight up</td>
<td>Z is vertical pointing up, parallel to positive world Z.</td>
<td>Z is horizontal, pointing back, parallel to negative world X.</td>
</tr>
</tbody>
</table>

This command, tz(), is joint-interpolated. The tool centre point travels as a result of various joint motions, not in a straight line.

For cartesian-interpolated (straight line) motion, see tzs().

**Syntax**

command tz( float distance )

**Parameters**

* distance the distance of travel, in current units: a float

**Returns**

Success = 0  
Failure < 0

**Example**

move(base_point)  
tz(200) ;; millimetres

**RAPL-II**

No equivalent.

**See Also**

tzs jogs like tz, but in straight line motion
jog_t alias of tz and moves along other axes
tx jogs like ty, but along X axis
ty jogs like ty, but along Y axis
depart moves along approach/depart axis
jog_w jogs like tz, but in world frame of reference

**Category**

Motion

---

**tzs**

**Alias**

<table>
<thead>
<tr>
<th>alias</th>
<th>same as</th>
</tr>
</thead>
<tbody>
<tr>
<td>tzs(...)</td>
<td>jog_tsz(TOOL_Z, ...)</td>
</tr>
</tbody>
</table>

**Description**

In the tool frame of reference, moves the tool centre point along the Z axis by the specified distance in current units (millimetres or inches).

The following table describes the positive Z axis for each tool coordinate system.
<table>
<thead>
<tr>
<th>arm position</th>
<th>F3 coordinate system</th>
<th>A465/A255 coordinate system</th>
</tr>
</thead>
<tbody>
<tr>
<td>any</td>
<td>Z is perpendicular to (arises out of) the tool flange.</td>
<td>(see below)</td>
</tr>
<tr>
<td>ready</td>
<td>Z is horizontal, pointing ahead, past the front of the arm, parallel to world X.</td>
<td>Z is vertical pointing up, parallel to positive world Z.</td>
</tr>
<tr>
<td>straight up</td>
<td>Z is vertical pointing up, parallel to positive world Z.</td>
<td>Z is horizontal, pointing back, parallel to negative world X.</td>
</tr>
</tbody>
</table>

This command, tzs(), is cartesian-interpolated (straight line).

For joint-interpolated (not straight) motion, see tzj

Syntax

```plaintext
command tzs( float distance )
```

Parameters

- **distance** the distance of travel, in current units or degrees: a float

Returns

- **Success** = 0
- **Failure** < 0

Example

```plaintext
move(base_point)
tzs(200) ;; millimetres
```

RAPL-II

No equivalent.

See Also

- tz jogs like tzs, but joint interpolated
- jog_ts alias of tzs and moves along other axes
- txs jogs like tzs, but along X axis
- tys jogs like tzs, but along Y axis
- depart moves along approach/depart axis
- jog_ws jogs like tzs, but in world frame of reference

Category

Motion

units_get

Description

Gets the current setting of units of linear measure, either metric (millimetres) or English (inches).

Syntax

```plaintext
command units_get( var unit_type linear_measure )
```

Parameter

- **linear_measure** the variable

Returns

- **Success >= 0**
  
  the parameter is loaded with one of:

  ```plaintext
  UNITS_METRIC
  UNITS_ENGLISH
  ```

- **Failure < 0**

Example

```plaintext
unit_type units
units_get(units)
if units == UNITS_METRIC
  print("Using metric units")
else
  print("Using English units")
end if
```

Result

prints the current units

See Also

- units_set sets the current units

Category

Robot Configuration
units_set

Description
Sets current units to metric (millimetres) or English (inches).
Sets the system of measurement for linear distances. Does not affect the system of measurement for rotational distances.

The default units are:
- F3 Metric
- A465, A255, earlier models English

If a cartesian location was taught in one system of units, it cannot be used in a program with the other system of units. The units setting does not affect precision locations.

Syntax
command units_set( unit_type linear_measure )

Parameter
linear_measure  the system of units, of type unit_type, one of:
- UNITS_METRIC
- UNITS_ENGLISH

Returns
Success >= 0
Failure < 0

Example
unit_type units = UNITS_METRIC
...
units_set(units)

Result
Configures robot for metric units

See Also
units_get  gets the current units

Category
Robot Configuration

unlink

Description
The unlink command removes a link to the file specified by path. If the link count is zero, the file is deleted.

Syntax
command unlink( var string[] path )

Parameter
path  A string defining the file and the path to the file

Returns
Success >= 0
Failure < 0

Example
string[32] path ="my_directory\myfile"
int check, fd

check = open(fd, path,O_RDWR, M_READ|M_WRITE)
if (check) =
  ;; continue ...
end if
...
unlink(path)

Result
Opens the file “path” - deletes it later
**unlock**

Description: Unlocks a specified axis.

Syntax: `command unlock( int axis )`

Parameter:
- `axis`  the axis to be unlocked: an int

Returns:
- Success >= 0
- Failure < 0

Example:
```
;;Unlock joint 1, move robot, lock joint 1
unlock(1)
  ... robot motion
lock(1)
```

**unmount**

Description: Unmounts a mounted filesystem from directory `dir`.

Syntax: `command unmount( var string[] dir )`

Parameter:
- `dir`  the mount point of the CROS directory: a string of var length

Returns:
- Success >= 0
- Failure < 0
  - EPERM must be a privileged process to unmount()
  - EINVAL invalid argument
  - ENOTDIR the mount point is not a directory
  - ENOENT a component was not found
  - EIO an I/O error occurred
  - EAGAIN temporarily out of resources needed to do this
  - EBUSY the mounted filesystem is busy

Example:
```
string[32] directory = "my_directory"
unmount(directory)
```

**unsetenv**

Description: Deletes the selected environment string. (See the section on environ() for more explanation.) (C500C only)

---

System Shell: Same as: rm, del.

RAPL-II: DELETE, DPROG

See Also: link, open

Category: File and Device System Management
**unsetenv**

**Syntax**

`command unsetenv(string[] key)`

**Parameter**

There is one required parameter:

- `key`  The key (left hand side before the '=' character) of the string to delete.

**Returns**

- **Success**: 0. (even if the key is not found, 0 is returned.)
- **Failure**: < 0 (-ve error code)

**Example**

```
;; Delete "MyString" from the environment
unsetenv("MyString")
```

**See Also**

environ(), getenv(), setenv()

**Category**

Environment Variables

---

**utime**

**Description**

Changes the modification time of a filesystem object.

**Library**

syslib

**Syntax**

`command utime(string [] path, int modtime)`

**Parameters**

There are two required parameters:

- `path`  the path of the object to modify
- `modtime`  what time to reset the object’s modification time to.

**Returns**

- `>= 0`  `→`  Success
- `< 0`  `→`  Failure

Possible failure return codes are:

- EINVAL  Invalid argument
- EBADF  There is no open file corresponding to `fd`.
- EACCESS  Access denied
- EIO  I/O error
- ENOTDIR  a component was not a directory
- ENOENT  the object was not found

**Example**

```
int t

t = time() ;; get the time NOW
...
utime("/tmp/xfile", t - 60) ;; reset the timestamp to one minute ago
...
```

**See Also**

mtime()

**Category**

File and Device System Management

---

**v3_save_on_exit**

**Description**

Sets the RAPL-3 interpreter so that when the program exits, all of its final v3 variable values will be saved to the specified v3 file. Note that the automatic save will fail if the file is not a valid v3 file with entries corresponding to each teachable variable in the current program.

The v3_save_on_exit() mechanism can be used to simulate persistent variables like the RAPL-II language had.
command v3_save_on_exit(int fd)

Parameter:

(fd) -- file descriptor of the open v3 file (must be open for both reading and writing.) If fd == -1, then the call cancels a previously requested save-on-exit.

Returns:

Success >= 0
Failure < 0 (-ve error code)

Example:

int fd
open(fd, "whatever.v3", O_RDWR, 0) ;; open my v3 file
v3_save_on_exit(fd)

Category:

v3 files

v3_vars_save

Description:

Writes the current program’s teachable variables to the file open on fd. The command will fail if the file is not a valid v3 file with entries corresponding to the current programs teachable variables. Note that the file (fd) is always closed after the command call whether the command succeeds or fails.

Syntax:

command v3_vars_save(int fd)

Parameter:

(fd) the file open

Returns:

Success = 0
Failure < -ve error descriptor

Example:

int fd
open(fd, "myname.v3", O_RDWR, 0)
v3_vars_save(fd)

See Also:

vars_save

Category:

v3 Files

va_arg_get

Description:

Gets the next argument into dst (converting to vat if required), advances va_next_ptr, and decrements va_count.

Used for subroutines and functions that have a variable number of arguments.

Syntax:

command va_arg_get(var int va_count, var void@ va_next_ptr, \va_types vat, void@ dst)

Parameters:

va_count an int
va_next_ptr void pointer
vat one of
global typedef va_types enum

va_t_void, ;; void
va_t_int, ;; int
va_t_float, ;; float
va_t_string, ;; string[: (can’t happen)
va_t_ploc, ;; ploc
va_t_cloc, ;; cloc
va_t_gloc, ;; gloc
va_t_unknown, ;; unknown; (can’t happen)

va_t_void_p = 0x10, ;; void@
va_t_int_p, ;; int@
va_t_float_p, ;; float@
va_t_string_p, ;; string[]@
va_t_ploc_p, ;; ploc@
va_t_cloc_p, ;; cloc@
va_t_gloc_p, ;; gloc@
va_t_ptr ;; other pointer type
end enum

dst void pointer

Returns
Success >= 0
Failure < 0
-ERANGE if there are no arguments left to get
-EINVAL if there is a problem getting the type of argument

Category
System Process Control: Operating System

va_arg_type

Description
Returns a type descriptor for the next varargs argument.
Used for subroutines and functions that have a variable number of arguments.

Syntax
func va_types va_arg_type(void@ va_next_ptr)

Parameters
va_next_ptr void pointer

Returns
Success >= 0. An enumeration constant (type va_types)

va_t_void ;; void
va_t_int ;; int
va_t_float ;; float
va_t_string ;; string[] (can’t happen)
va_t_cloc ;; cloc
va_t_ploc ;; ploc
va_t_gloc ;; gloc
va_t_unknown ;; unknown (can’t happen)
va_t_void_p ;; void@
va_t_int_p ;; int@
va_t_float_p ;; float@
va_t_string_p ;; string[]@
va_t_cloc_p ;; cloc@
va_t_ploc_p ;; ploc@
va_t_gloc_p ;; gloc@
va_t_ptr ;; other pointer type

Failure < 0

Example
sub do_something( int a, ...)
  int b
  ...
  case va_count:
  of 0:
    b = 0 ;; default
  else
    if (va_type_arg(va_next_ptr) == va_t_int)
      va_get_arg(va_count, va_next_ptr, va_t_int, &b)
    else ;; wrong type passed
      b = 0 ;; use default
    end if
  end case
  ...
end sub

Category
System Process Control: Operating System
**var_teach**

**Description**
Teach the variable whose name is “name”. Returns True if successful, False if not correctly taught or negative if not found or otherwise in error. Refer also to the var_teach_v command.

**Library**
stp

**Syntax**
```export command var_teach(var string[] name, int index_1, int index_2)```

**Parameter**
- **name**: name of the variable to be taught
- **index_1**: first argument of an array
- **index_2**: second argument in a two dimensional array

**Returns**
- **Success >= 0**: True if taught, False if not taught
- **Failure < 0**: error descriptor

**Example**
```...stp:var_teach("new_array",1,1)...
```

**See Also**
var_teach_v

**Category**
Pendant

---

**vars_save**

**Description**
Invokes the v3_vars_save() operation on the currently open application v3 file. This presupposes that the calling program is open application and that the variables in the open application are actually desired variables. If this assumption is false the command will likely fail or do something unpredictable (and NOT useful.).

**Library**
stp

**Syntax**
```export command var_save()```

**Parameter**
No parameters

**Returns**
- **Success >= 0**: Returns 0 if successful
- **Failure < 0**
  - -1 no application open
  - Returns error descriptor

**Example**
```int fd
open(fd, "myname.v3", O_RDWR, 0)
...
stp:vars_save()
...
```

**Result**
Saves the open application’s variables to file fd.

**See Also**
v3_vars_save

**Category**
Pendant

---

**verstring_get**

**Description**
Gets the current kinematics version string.
command verstring_get( var string[] s )

Parameters
s the string variable for the kinematics version

Returns
Success >= 0
the variable is packed
Failure < 0

Category
Status
Robot Configuration

waitpid

Description
Waits for the child process wpid to complete. If wpid=W_ANY, waits for any child process to complete. If status is not NULL, the child process status is stored in status@.

Syntax
func int waitpid( int wpid, int@ status, int options )

Parameters
wpid an int - the child process
status pointer to an int
options
0
W_ANY waits for any child
W_NOHANG waitpid checks for child completion and returns immediately

Returns
Success >= 0
positive pid the pid of the child, if the requested child terminated
0 (-EOK) if W_NOHANG is in effect and no child has terminated
Failure < 0
-ESRCH no process with that pid exists
-ECHILD no child process exists
-EINTR was interrupted by a signal

Example
int pid
...
pid = split()
if pid == 0
  :: Child process
  exec("/bin/ls")
  exit(0)
else
  :: Parent waits for child
  while waitpid( pid, NULL, 0 ) == 0
  end while
  :: Finish Code
end if

See Also
WEXITSTATUS
WIFEXITED
WIFSIGNALED
WTERMSIG

Category
System Process Control: Single and Multiple Processes
**WEXITSTATUS**

- **Description**: If `status` is the child status returned by waitpid, then WEXITSTATUS returns the actual exit code of the child process that exited. (This is simply the lower byte of `status`.)

- **Syntax**: `func int WEXITSTATUS( int status )`

- **Parameter**: `status` - an int - child status

- **Returns**: Success >= 0  
  Failure < 0

- **Example**:  
  ```c
  int status ...
  status = WEXITSTATUS( status )
  ```

- **Category**: System Process Control: Single and Multiple Processes

---

**WIFEXITED**

- **Description**: WIFEXITED returns 1 if `status` indicates that the child process exited, and returns 0 otherwise.

- **Syntax**: `func int WIFEXITED( int status )`

- **Parameters**: `status` - an int - child process status

- **Returns**: Success >= 0  
  Failure < 0

- **Example**:  
  ```c
  int status ...
  if WIFEXITED( status )
    ;; Process exited
  else
    ;; Process was signaled
  end if
  ```

- **Category**: System Process Control: Single and Multiple Processes

---

**WIFSIGNALED**

- **Description**: WIFSIGNALED returns 1 if the child process was signal-terminated, and returns 0 otherwise.

- **Syntax**: `func int WIFSIGNALED( int status )`

- **Parameters**: `status` - an int - child process status

- **Returns**: Success >= 0  
  Failure < 0

- **Example**:  
  ```c
  int status ...
  if WIFSIGNALED( status )
    ;; Process was signaled
  else
    ;; Process exited
  end if
  ```
world_to_joint

Description
Converts a location from world coordinates to joint angles. Used if a location of one type needs to be converted to another type for checking or other use within the program.

Syntax
```
command world_to_joint( cloc world, var float[8] joint )
```

Parameters
- `world`: the location in world coordinates: a cloc
- `joint`: the location in joint angles (an array of floats)

Returns
- Success >= 0
  - `joint` is packed
- Failure < 0

Example
```
float[8] joints1
teachable cloc world1
...
world_to_joint(world1, joints1)
```

Result
`joint1` is packed with the appropriate joint data

RAPL-II
Similar to SET with different location types.

See Also
- `joint_to_world` converts joint angles to world coordinates
- `world_to_motor` converts world coordinates to motor pulses

Category
Location: Kinematic Conversions

world_to_motor

Description
Converts a location from world coordinates to motor pulses. Used if a location of one type needs to be converted to another type for checking or other use within the program.

Syntax
```
command world_to_motor( cloc world, var ploc motor )
```

Parameters
- `world`: the location in world coordinates: a cloc
- `motor`: the location in motor pulses: a ploc

Returns
- Success >= 0
  - `motor` is packed
- Failure < 0

Example
```
ploc motor1
teachable cloc world1
...
world_to_joint(world1, motor1)
```

Result
`motor1` is packed with the appropriate joint coordinate data

RAPL-II
Similar to SET with different location types.

See Also
- `motor_to_world` converts motor pulses to world coordinates
- `world_to_joint` converts world coordinates to joint angles

Category
Location: Kinematic Conversions
### write

**Description**
Attempts to write `nwords` from `buf` to the file descriptor `fd`. If the number of words specified in `nwords` cannot be written the command performs a blocking write, unless the file descriptor was opened with mode O_NONBLOCK. After writing, the file position is increased by the number of words written. This provides a sequential move through the file.

write() handles 4-byte words. writes() handles characters.

Similar to send() which is used with sockets.

**Syntax**
```
command write( int fd, void@ buf, int nwords )
```

**Returns**
- Success >= 0
- Failure < 0
  - EINVAL the arguments were invalid (i.e., -ve fd)
  - EBADF the file descriptor isn't open
  - EACCESS not open for writing
  - ESPPIPE can't r/w on a socket
  - EIO an I/O error occurred
  - ENOSPC out of space on the device
  - ENOMEM (mfs only) out of memory
  - EAGAIN (nonblocking I/O) not ready to write any bytes
  - EINTR was interrupted by a signal

**Example**
```
int fd
int[10] buf
...
open ( fd, "filename.txt", O_RDONLY, 0 )
write ( fd, buf, sizeof(buf) )
```

**See Also**
- read read words from a file
- writes write a string to a file
- send write to a socket

**Category**
File Input and Output: Unformatted Output

### writeread

**Description**
Writes `wlen` number of words to the file descriptor `fd` and then reads at most `rlen` number of words from the file descriptor `fd`.

This command may or may not block, depending on the flags (O_NONBLOCK) used when opening the file descriptor `fd` and the device driver (which may not support blocking or non-blocking modes). Many devices do not support this call, and with those devices writeread() returns -ENODEV on invocation. For example, all the file systems (MFS, NFS, etc.) do not support writeread().

**Syntax**
```
command writeread( int fd, void@ wbuf, int wlen, void@ rbuf, int rlen )
```

**Returns**
- Success >= 0 Returns the number of words read.
- Failure < 0
  - EINVAL the arguments were invalid (i.e., -ve fd)
  - EBADF the file descriptor isn't open
  - EACCESS not open for reading and writing
  - ESPPIPE can't r/w on a socket
  - ENODEV this is not a device that supports writeread().
  - EIO an I/O error occurred

**See Also**
- write write words from a buffer to the file
- writes write a string to a file
write words to a file

description

writes

writes the string s to the file indicated by fd. this is different from the write command in that a string is used, and the starting location start is the first character of the string to be sent.

syntax

command writes( int fd, var string[] s, int start )

returns

success >= 0 returns the number of characters written to the file

failure < 0 returns a negative error code if the write fails.

example

string[] buf = “only writes_test”
int fd
open ( fd, “/temp/writes_test”, O_RDONLY, 0 )
;; only write “writes_test”
writest ( fd, buf, 5 )
;; start from the character ‘w’

see also

write write words to a file

description

wtermsig

returns the actual signal number that terminated a wifsignaled() process.

syntax

func signal_code wtermsig(int status)

returns

success >= 0, one of:

SIGKILL = 1
SIGSEGV = 2
SIGILL = 3
SIGFPE = 4
SIGSYS = 5
SIGABRT = 6
SIGINT = 7
SIGALRM = 8
SIGHUP = 9
SIGPIPE = 10
SIGCHLD = 11
SIG2 = 12
SIG13 = 13
SIG14 = 14
SIG15 = 15
SIG16 = 16
SIG17 = 17
SIG18 = 18
SIG19 = 19
SIG20 = 20
SIG21 = 21
SIG22 = 22
SIG23 = 23
SIG24 = 24

failure < 0

category

system process control: single and multiple processes

signal handling
**WX**

**Alias**

* jog_w ...

<table>
<thead>
<tr>
<th>alias</th>
<th>same as</th>
</tr>
</thead>
<tbody>
<tr>
<td>wx(...)</td>
<td>jog_w(WORLD_X, ...)</td>
</tr>
</tbody>
</table>

**Description**

In the world frame of reference, moves the tool centre point to the end point which is a specified distance along the X axis, in current units (millimetres or inches). This command, wx() is joint-interpolated. The tool centre point travels as a result of various joint motions, not in a straight line.

For cartesian-interpolated (straight line) motion, see wxs().

**Syntax**

`command wx( float distance )`

**Parameters**

* distance the distance of travel, in current units: a float

**Returns**

Success = 0
Failure < 0

**Example**

`move(base_point)`

wx(200) ;; millimetres

**RAPL-II**

Similar to JOG and X, without straight line parameter.

**See Also**

wx jogs like wxs, but in straight line motion
jog_w alias of wx and moves along other axes
wy jogs like wx, but along Y axis
wz jogs like wx, but along Z axis
jog_t jogs like wx, but in tool frame of reference
joint moves by joint degrees
motor moves by encoder pulses

**Category**

Motion

---

**WXS**

**Alias**

* jog_ws ...

<table>
<thead>
<tr>
<th>alias</th>
<th>same as</th>
</tr>
</thead>
<tbody>
<tr>
<td>wxs(...)</td>
<td>jog_ws(WORLD_X, ...)</td>
</tr>
</tbody>
</table>

**Description**

In the world frame of reference, moves the tool centre point along the X axis by the specified distance in current units (millimetres or inches). This command, wxs(), is cartesian-interpolated (straight line).

For joint-interpolated (not straight) motion, see wx().

**Syntax**

`command wxs( float distance )`

**Parameters**

* distance the distance of travel, in current units or degrees: a float

**Returns**

Success = 0
Failure < 0

**Example**

`move(base_point)`

wxs(200) ;; millimetres

**RAPL-II**

Similar to JOG and X, with straight line parameter.

**See Also**

wx jogs like wxs, but joint interpolated
jog_ws alias of wxs and moves along other axes
wys jogs like wxs, but along Y axis
Subprograms: Alphabetical Listing

wzs  jogs like wxs, but along Z axis
jog_ts  jogs like wxs, but in tool frame of reference
joint  moves by joint degrees
motor  moves by encoder pulses

Category  Motion

**wy**

Alias  
**jog_w** ...

<table>
<thead>
<tr>
<th>alias</th>
<th>same as</th>
</tr>
</thead>
<tbody>
<tr>
<td>wy(...)</td>
<td>jog_w(WORLD_Y, ...)</td>
</tr>
</tbody>
</table>

Description  In the world frame of reference, moves the tool centre point to the end point which is a specified distance along the Y axis, in current units (millimetres or inches). This command, wy() is joint-interpolated. The tool centre point travels as a result of various joint motions, not in a straight line.

For cartesian-interpolated (straight line) motion, see wys().

Syntax  
command  wy( float distance )

Parameters  
*distance*  the distance of travel, in current units: a float

Returns  
Success = 0
Failure < 0

Example  
move(base_point)
wy(200) ;; millimetres

RAPL-II  Similar to JOG and Y, without straight line parameter.

See Also  
wys  jogs like wy, but in straight line motion
jog_w  alias of wy and moves along other axes
wx  jogs like wy, but along X axis
wz  jogs like wy, but along Z axis
jog_t  jogs like wy, but in tool frame of reference
joint  moves by joint degrees
motor  moves by encoder pulses

Category  Motion

**wys**

Alias  
**jog_ws** ...

<table>
<thead>
<tr>
<th>alias</th>
<th>same as</th>
</tr>
</thead>
<tbody>
<tr>
<td>wys(...)</td>
<td>jog_ws(WORLD_Y, ...)</td>
</tr>
</tbody>
</table>

Description  In the world frame of reference, moves the tool centre point along the Y axis by the specified distance in current units (millimetres or inches). This command, wys(), is cartesian-interpolated (straight line).

For joint-interpolated (not straight) motion, see wy()

Syntax  
command  wys( float distance )

Parameters  
*distance*  the distance of travel, in current units or degrees: a float
 Returns  
Success = 0  
Failure < 0  

 Example  
move(base_point)  
  wys(200) ;; millimetres  

 RAPL-II  
Similar to JOG and Y, with straight line parameter.  

 See Also  
  wy  jogs like wys, but joint interpolated  
  jog_ws alias of wys and moves along other axes  
  wxs  jogs like wys, but along X axis  
  wzs  jogs like wys, but along Z axis  
  jog_ts  jogs like wys, but in tool frame of reference  
  joint moves by joint degrees  
  motor moves by encoder pulses  

 Category  
Motion  

 wz  

 Alias  
 jog_w ...  

 Description  
In the world frame of reference, moves the tool centre point to the end point  
which is a specified distance along the Z axis, in current units (millimetres or  
inches). This command, wz() is joint-interpolated. The tool centre point travels as  
a result of various joint motions, not in a straight line.  
For cartesian-interpolated (straight line) motion, see wzs().  

 Syntax  
command wz( float distance )  

 Parameters  
distance the distance of travel, in current units: a float  

 Returns  
Success = 0  
Failure < 0  

 Example  
move(base_point)  
  wz(200) ;; millimetres  

 RAPL-II  
Similar to JOG and Z, without straight line parameter.  

 See Also  
  wzs jogs like wz, but in straight line motion  
  jog_w alias of wz and moves along other axes  
  wx  jogs like wz, but along X axis  
  wy  jogs like wz, but along Y axis  
  jog_t  jogs like wz, but in tool frame of reference  
  joint moves by joint degrees  
  motor moves by encoder pulses  

 Category  
Motion  

 wzs  

 Alias  
 jog_ws ...  

 Description  
In the world frame of reference, moves the tool centre point to the end point  
which is a specified distance along the Z axis, in current units (millimetres or  
inches). This command, wzs() is joint-interpolated. The tool centre point travels as  
a result of various joint motions, not in a straight line.  
For cartesian-interpolated (straight line) motion, see wzs().  

 Syntax  
command wzs( float distance )  

 Parameters  
distance the distance of travel, in current units: a float  

 Returns  
Success = 0  
Failure < 0  

 Example  
move(base_point)  
  wzs(200) ;; millimetres  

 RAPL-II  
Similar to JOG and Z, without straight line parameter.  

 See Also  
  wzs jogs like wz, but in straight line motion  
  jog_ws alias of wzs and moves along other axes  
  wx  jogs like wz, but along X axis  
  wy  jogs like wz, but along Y axis  
  jog_t  jogs like wz, but in tool frame of reference  
  joint moves by joint degrees  
  motor moves by encoder pulses  

 Category  
Motion  


Description
In the world frame of reference, moves the tool centre point along the Z axis by
the specified distance in current units (millimetres or inches). This command,
wzs(), is cartesian-interpolated (straight line).

For joint-interpolated (not straight) motion, see wz()

Syntax
command wzs( float distance )

Parameters
distance the distance of travel, in current units or degrees: a float

Returns
Success = 0
Failure < 0

Example
move(base_point)
wzs(200) ;; millimetres

RAPL-II
Similar to JOG and Z, with straight line parameter.

See Also
wz jogs like wzs, but joint interpolated
jog_ws alias of wzs and moves along other axes
wxs jogs like wzs, but along X axis
wys jogs like wzs, but along Y axis
jog_ts jogs like wzs, but in tool frame of reference
joint moves by joint degrees
motor moves by encoder pulses

Category Motion

xpulses_get

Description
Gets xpulses, the number of encoder pulses per revolution of a motor, for all axes.

Syntax
command xpulses_get( var int[8] pulses )

Parameter
pulses the pulses of all axes: an array of ints

Returns
Success >= 0.
          The array 'pulses' is packed.
Failure < 0

See Also
xpulses_set sets the number of pulses per revolution for an axis

Category Robot Configuration

xpulses_set

Description
For an axis, sets xpulses, the number of encoder pulses per revolution of the motor.

Syntax
command xpulses_set( int axis, int xpulses )

Parameters
axis the axis being set: an int
xpulses the number of pulses per revolution: an int

Returns
Success >= 0
Failure < 0

Example
xpulses_set(8,1000)

RAPL-II
@XPULSES

See Also
configaxis configures an axis including sets pulses
xpulses_get gets the number of pulses per revolution for all axes
xratio_get

Description
Gets xratio, the ratio of the number of motor turns (revolutions) per unit of joint
displacement (degrees for robot joints and carousels, mm or inch for track).

Syntax
command xratio_get( var float[8] ratio )

Parameter
ratio the ratios for all axes: an array of up to 8 floats

Returns
Success >= 0. the parameter is packed
Failure < 0

Example
float[8] ratios
int check
;; get pulse to motion conversions
check = xratio_get(ratios)

See Also
xratio_set sets the ratio of conversion

category Robot Configuration

xratio_set

Description
Sets xratio, the ratio of the number of motor turns (revolutions) per unit of joint
displacement (degrees for robot joints and carousels, mm or inch for track).

Syntax
command xratio_set( int axis, float xratio )

Parameters
axis the axis being set: an int
xratio the ratio of conversion: a float

Returns
Success >= 0
Failure < 0

Example
xratio_set(8,11.5)

RAPL-II @XRATIO

See Also
configaxis configures an axis including sets ratio
xratio_get gets the ratio of conversion

category Robot Configuration

xrot

Alias
jog_w ...

<table>
<thead>
<tr>
<th>alias</th>
<th>same as</th>
</tr>
</thead>
<tbody>
<tr>
<td>xrot(...)</td>
<td>jog_w(WORLD_XROT, ...)</td>
</tr>
</tbody>
</table>

Description
In the world frame of reference, rotates the tool around the X axis by the specified
degrees.

This command, xrot(), is joint-interpolated. The end-point is determined and the
tool travels to it as a result of various joint motions. The start point and end
point for the tool centre point are the same (no change in distance along the axis.
or angle between the axis and the tool), but the start position and end position of the tool are different.

For cartesian-interpolated (straight line) motion, see xrots().

Syntax

```
command xrot( float distance )
```

Parameters

distance

the distance of travel, in current units or degrees: a float

Returns

Success = 0

Failure < 0

Example

```
appro(centre)
pitch(45) ;; pitch around tool point
xrot(45) ;; rotate around world X axis
```

RAPL-II

Similar to JOG, without straight line parameter.

Also similar to ROLL. In RAPL-II this name was used for a rotation in the world frame of reference. In RAPL-3, the world rotation is called xrot and the tool rotation is called roll.

See Also

xrots like xrot, but in straight-line mode

jog_w like xrot and around and along all axes

yrot rotates around world Y axis

zrot rotates around world Z axis

jog_t jogs, but in tool frame of reference

joint moves by joint degrees

motor moves by encoder pulses

Category

Motion

---

**xrots**

Alias

```
alias jog_ws ...
```

<table>
<thead>
<tr>
<th>alias</th>
<th>same as</th>
</tr>
</thead>
<tbody>
<tr>
<td>xrots(...)</td>
<td>jog_ws(WORLD_XROT, ...)</td>
</tr>
</tbody>
</table>

Description

In the world frame of reference, rotates the tool around the X axis by the specified degrees.

This command, xrots(), is cartesian-interpolated (straight-line). The tool centre point travels in a straight line along the axis to the end point.

For joint-interpolated (not straight) motion, see xrot().

Syntax

```
command xrots( float distance )
```

Parameters

distance

the distance of travel, in current units or degrees: a float

Returns

Success = 0

Failure < 0

Example

```
appro(centre)
pitch(45) ;; pitch around tool point
xrots(45) ;; rotate around world X axis
```

RAPL-II

Similar to JOG, with straight line parameter.

Also similar to ROLL. In RAPL-II this name was used for a rotation in the world frame of reference. In RAPL-3, the world rotation is called xrot and the tool rotation is called roll.
**See Also**
- xrot: like xrots, but joint-interpolated
- jog_w: like xrots and around and along all axes
- yrots: rotates around world Y axis
- zrots: rotates around world Z axis
- jog_t: jogs, but in tool frame of reference
- joint: moves by joint degrees
- motor: moves by encoder pulses

**Category**
Motion

---

**yaw**

**Alias**
- jog_t...

<table>
<thead>
<tr>
<th>alias</th>
<th>same as</th>
</tr>
</thead>
<tbody>
<tr>
<td>yaw(...)</td>
<td>jog_t(TOOL_YAW, ...)</td>
</tr>
</tbody>
</table>

**Description**
In the tool frame of reference, rotates around the normal axis, by the specified number of degrees.

<table>
<thead>
<tr>
<th>motion</th>
<th>axis</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>common name</td>
</tr>
<tr>
<td>yaw</td>
<td>normal</td>
</tr>
</tbody>
</table>

This command, yaw(), is joint-interpolated. The end position is determined and the tool travels to it as a result of various joint motions. The start point and end point for the tool centre point are the same (no change in distance along the axis or angle between the axis and the tool), but the start position and end position of the tool are different by the amount of rotation.

For cartesian-interpolated (straight line) motion, see yaws().

**Syntax**
```
command yaw( float distance )
```

**Parameter**
- distance: the amount of rotation in degrees: a float

**Returns**
- Success = 0
- Failure < 0

**Example**
- yaw(45)
- yaw(-8.25)

**Application Shell**
Same as yaw.

**RAPL-II**
No equivalent. In RAPL-II, YAW performed a different motion. See zrot.

**See Also**
- yaws: moves around the tool normal axis, but in straight line motion
- pitch: moves around the tool orientation axis
- roll: moves around the tool approach/depart axis

**Category**
Motion

---

**yaws**

**Alias**
- jog_ts...

<table>
<thead>
<tr>
<th>alias</th>
<th>same as</th>
</tr>
</thead>
</table>

---
### yaws

**Description**
In the tool frame of reference, rotates around the normal axis, by the specified number of degrees.

<table>
<thead>
<tr>
<th>motion</th>
<th>axis</th>
</tr>
</thead>
<tbody>
<tr>
<td>common name</td>
<td>F3 coordinate system</td>
</tr>
<tr>
<td>yaw</td>
<td>normal</td>
</tr>
</tbody>
</table>

This command, `yaws()`, is cartesian-interpolated (straight-line) motion. The tool centre point stays on the axis, in the same place, while the tool rotates around the axis.

For joint-interpolated motion, see `yaw()`.

**Syntax**
```
command yaws( float distance )
```

**Parameter**
- `distance` the amount of rotation in degrees: a float

**Returns**
- Success = 0
- Failure < 0

**Example**
```
yaws(45)
yaws(-57.5)
```

**Application Shell**
Same as `yaws`

**RAPL-II**
No equivalent. In RAPL-II, YAW performed a different motion. See `zrots`.

**See Also**
- `yaw` moves around the tool normal axis, but joint-interpolated
- `pitchs` moves around the tool orientation axis in straight line motion
- `rolls` moves around the tool approach/depart axis in straight line motion

**Category**
Motion

### yrot

**Alias**
```
jog_w ...
```

```
<table>
<thead>
<tr>
<th>alias</th>
<th>same as</th>
</tr>
</thead>
<tbody>
<tr>
<td>yrot(...)</td>
<td>jog_w(WORLD_YROT, ...)</td>
</tr>
</tbody>
</table>
```

**Description**
In the world frame of reference, rotates the tool around the Y axis by the specified degrees.

This command, `yrot()`, is joint-interpolated. The end-point is determined and the tool travels to it as a result of various joint motions. The start point and end point for the tool centre point are the same (no change in distance along the axis or angle between the axis and the tool), but the start position and end position of the tool are different.

For cartesian-interpolated (straight line) motion, see `yrots()`.

**Syntax**
```
command yrot( float distance )
```

**Parameter**
- `distance` the distance of travel, in current units or degrees: a float

**Returns**
- Success = 0
- Failure < 0
**yrots**

**Alias**

**jog_ws ...**

<table>
<thead>
<tr>
<th>alias</th>
<th>same as</th>
</tr>
</thead>
<tbody>
<tr>
<td>yrots(...)</td>
<td>jog_ws(WORLD_YROT, ...)</td>
</tr>
</tbody>
</table>

**Description**

In the world frame of reference, rotates the tool around the Y axis by the specified degrees.

This command, yrots(), is cartesian-interpolated (straight-line). The tool centre point travels in a straight line along the axis to the end point.

For joint-interpolated (not straight) motion, see yrot().

**Syntax**

command yrots( float distance )

**Parameter**

*distance* the distance of travel, in current units or degrees: a float

**Returns**

Success = 0
Failure < 0

**Example**

appro(centre)
pitch(45) ;; pitch around tool point
yrot(45) ;; rotate around world Y axis

**RAPL-II**

Similar to JOG, with straight line parameter.

Also similar to PITCH. In RAPL-II this name was used for a rotation in the world frame of reference. In RAPL-3, the world rotation is called yrot and the tool rotation is called pitch.

**See Also**

yrot like yrots, but joint-interpolated
jog_w like yrots and around and along all axes
xrots rotates around world X axis
zrots rotates around world Z axis
jog_t jogs, but in tool frame of reference
joint moves by joint degrees
motor moves by encoder pulses

**Category**

Motion
zero

Description
Sets all the current motor position registers to 0.

Syntax
command zero()

Returns
Success >= 0
Failure < 0

Example
zero()

RAPL-II
Same as @ZERO.

See Also
here stores a location in a location variable
pos_get gets the position of the robot
pos_set sets the position of the robot to any value

Category
Calibration
Home

zrot

Alias
jog_w ...

<table>
<thead>
<tr>
<th>alias</th>
<th>same as</th>
</tr>
</thead>
<tbody>
<tr>
<td>zrot(...)</td>
<td>jog_w(WORLD_ZROT,...)</td>
</tr>
</tbody>
</table>

Description
In the world frame of reference, rotates the tool around the Z axis by the specified degrees.

This command, zrot(), is joint-interpolated. The end-point is determined and the tool travels to it as a result of various joint motions. The start point and end point for the tool centre point are the same (no change in distance along the axis or angle between the axis and the tool), but the start position and end position of the tool are different.

For cartesian-interpolated (straight line) motion, see zrots().

Syntax
command zrot( float distance )

Parameter
distance the distance of travel, in current units or degrees: a float

Returns
Success = 0
Failure < 0

Example
appro(centre)
pitch(45) ;; pitch around tool point
zrot(45) ;; rotate around world Z axis

RAPL-II
Similar to JOG, without straight line parameter.

Also similar to YAW. In RAPL-II this name was used for a rotation in the world frame of reference. In RAPL-3, the world rotation is called zrot and the tool rotation is called yaw.

See Also
zrots like zrot, but in straight-line mode
jog_w like zrot and around and along all axes
xrot rotates around world X axis
yrot rotates around world Y axis
jog_t jogs, but in tool frame of reference
joint moves by joint degrees
motor moves by encoder pulses

Category
Motion
zrots

Alias

\textit{jog\_ws} ...

<table>
<thead>
<tr>
<th>alias</th>
<th>same as</th>
</tr>
</thead>
<tbody>
<tr>
<td>zrots(...)</td>
<td>\textit{jog_ws} (WORLD_ZROT, ...)</td>
</tr>
</tbody>
</table>

Description

In the world frame of reference, rotates the tool around the Z axis by the specified degrees.

This command, \textit{zrots()}, is cartesian-interpolated (straight-line). The tool centre point travels in a straight line along the axis to the end point.

For joint-interpolated (not straight) motion, see \textit{zrot()}.

Syntax

\texttt{command zrots( float distance )}

Parameter

- \textit{distance} \hfill the distance of travel, in current units or degrees: a float

Returns

- Success = 0
- Failure < 0

Example

\begin{verbatim}
appro(centre)
pitch(45) ;; pitch around tool point
zrots(45) ;; rotate around world Z axis
\end{verbatim}

RAPL-II

Similar to JOG, with straight line parameter.

Also similar to YAW. In RAPL-II this name was used for a rotation in the world frame of reference. In RAPL-3, the world rotation is called zrot and the tool rotation is called yaw.

See Also

- \textit{zrot} \hfill like \textit{zrots}, but joint-interpolated
- \textit{jog\_w} \hfill like \textit{zrots} and around and along all axes
- \textit{xrots} \hfill rotates around world X axis
- \textit{yrots} \hfill rotates around world Y axis
- \textit{jog\_t} \hfill jogs, but in tool frame of reference
- \textit{joint} \hfill moves by joint degrees
- \textit{motor} \hfill moves by encoder pulses

Category

Motion
## Signals

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Number</th>
<th>Description</th>
<th>Default Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGKILL</td>
<td>1</td>
<td>Kill (cannot be masked or modified)</td>
<td>Terminate</td>
</tr>
<tr>
<td>SIGSEGV</td>
<td>2</td>
<td>Segmentation violation</td>
<td>Terminate</td>
</tr>
<tr>
<td>SIGILL</td>
<td>3</td>
<td>Illegal instruction</td>
<td>Terminate</td>
</tr>
<tr>
<td>SIGFPE</td>
<td>4</td>
<td>Floating point exception</td>
<td>Terminate</td>
</tr>
<tr>
<td>SIGSYS</td>
<td>5</td>
<td>Bad argument to system call</td>
<td>Terminate</td>
</tr>
<tr>
<td>SIGABRT</td>
<td>6</td>
<td>Abort</td>
<td>Terminate</td>
</tr>
<tr>
<td>SIGINT</td>
<td>7</td>
<td>Interrupt</td>
<td>Terminate</td>
</tr>
<tr>
<td>SIGALRM</td>
<td>8</td>
<td>Alarm clock</td>
<td>Terminate</td>
</tr>
<tr>
<td>SIGHUP</td>
<td>9</td>
<td>Hang up</td>
<td>Terminate</td>
</tr>
<tr>
<td>SIGPIPE</td>
<td>10</td>
<td>Write to pipe, but no process to read it</td>
<td>Terminate</td>
</tr>
<tr>
<td>SIGSOCK</td>
<td>11</td>
<td>Write to socket, but no process to read it</td>
<td>Terminate</td>
</tr>
<tr>
<td>SIGRPWR</td>
<td>12</td>
<td>Robot power fail</td>
<td>Terminate</td>
</tr>
<tr>
<td>SIG13</td>
<td>13</td>
<td>Undefined</td>
<td>Terminate</td>
</tr>
<tr>
<td>SIG14</td>
<td>14</td>
<td>Undefined</td>
<td>Terminate</td>
</tr>
<tr>
<td>SIG15</td>
<td>15</td>
<td>Undefined</td>
<td>Terminate</td>
</tr>
<tr>
<td>SIG16</td>
<td>16</td>
<td>Undefined</td>
<td>Terminate</td>
</tr>
<tr>
<td>SIGCHLD</td>
<td>17</td>
<td>Child process died</td>
<td>Ignore</td>
</tr>
<tr>
<td>SIG18</td>
<td>18</td>
<td>Undefined</td>
<td>Ignore</td>
</tr>
<tr>
<td>SIG19</td>
<td>19</td>
<td>Undefined</td>
<td>Ignore</td>
</tr>
<tr>
<td>SIG20</td>
<td>20</td>
<td>Undefined</td>
<td>Ignore</td>
</tr>
<tr>
<td>SIG21</td>
<td>21</td>
<td>Undefined</td>
<td>Ignore</td>
</tr>
<tr>
<td>SIG22</td>
<td>22</td>
<td>Undefined</td>
<td>Ignore</td>
</tr>
<tr>
<td>SIG23</td>
<td>23</td>
<td>Reserved for system use</td>
<td>Ignore (non-interruptible)</td>
</tr>
<tr>
<td>SIG24</td>
<td>24</td>
<td>Reserved for system use</td>
<td>Ignore (will interrupt a process blocked on socket i/o)</td>
</tr>
</tbody>
</table>

Any signal interrupts `msleep()` or `waitpid()`. Signal <= 8, SIGKILL to SIGALRM, interrupts WAITIO, WAITSOCK, WAITSEM. Signal 11, SIGSOCK, interrupts WAITSOCK. WAITIO, WAITSOCK, and WAITSEM are states that a process can be in.