schedulix Server

Command Reference
Release 2.11

Dieter Stubler    Ronald Jeninga

January 2, 2024
## Contents

<table>
<thead>
<tr>
<th>Contents</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table of Contents</td>
<td>3</td>
</tr>
<tr>
<td>List of Tables</td>
<td>10</td>
</tr>
</tbody>
</table>

### I. General

1. **Introduction**
   - Introduction ........................................ 17

2. **Utilities**
   - Starting and stopping the server ........................ 25
     - server-start ......................................... 25
     - server-stop .......................................... 26
   - sdmsh .................................................. 27
   - sdms-auto_restart ..................................... 36
   - sdms-get_variable ................................. 38
   - sdms-rerun ........................................... 40
   - sdms-set_state ....................................... 42
   - sdms-set_variable ................................. 44
   - sdms-set_warning .................................... 46
   - sdms-submit ........................................ 48

### II. User Commands

3. **alter commands**
   - alter comment ....................................... 54
   - alter environment .................................... 56
   - alter event .......................................... 57
   - alter exit state mapping .......................... 58
   - alter exit state profile ........................... 59
   - alter exit state translation ..................... 61
   - alter folder ......................................... 62
   - alter footprint ...................................... 64
   - alter group .......................................... 66
   - alter interval ....................................... 67
<table>
<thead>
<tr>
<th>Command Type</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alter job</td>
<td>69</td>
</tr>
<tr>
<td>Alter job definition</td>
<td>74</td>
</tr>
<tr>
<td>Alter named resource</td>
<td>80</td>
</tr>
<tr>
<td>Alter resource</td>
<td>82</td>
</tr>
<tr>
<td>Alter resource state mapping</td>
<td>83</td>
</tr>
<tr>
<td>Alter resource state profile</td>
<td>84</td>
</tr>
<tr>
<td>Alter schedule</td>
<td>85</td>
</tr>
<tr>
<td>Alter scheduled event</td>
<td>87</td>
</tr>
<tr>
<td>Alter scope</td>
<td>88</td>
</tr>
<tr>
<td>Alter session</td>
<td>91</td>
</tr>
<tr>
<td>Alter trigger</td>
<td>93</td>
</tr>
<tr>
<td>Alter user</td>
<td>95</td>
</tr>
<tr>
<td><strong>4. Connect commands</strong></td>
<td><strong>99</strong></td>
</tr>
<tr>
<td>Connect</td>
<td>100</td>
</tr>
<tr>
<td><strong>5. Copy commands</strong></td>
<td><strong>103</strong></td>
</tr>
<tr>
<td>Copy folder</td>
<td>104</td>
</tr>
<tr>
<td>Copy named resource</td>
<td>105</td>
</tr>
<tr>
<td>Copy scope</td>
<td>106</td>
</tr>
<tr>
<td><strong>6. Create commands</strong></td>
<td><strong>107</strong></td>
</tr>
<tr>
<td>Create comment</td>
<td>108</td>
</tr>
<tr>
<td>Create environment</td>
<td>110</td>
</tr>
<tr>
<td>Create event</td>
<td>111</td>
</tr>
<tr>
<td>Create exit state definition</td>
<td>112</td>
</tr>
<tr>
<td>Create exit state mapping</td>
<td>113</td>
</tr>
<tr>
<td>Create exit state profile</td>
<td>114</td>
</tr>
<tr>
<td>Create exit state translation</td>
<td>117</td>
</tr>
<tr>
<td>Create folder</td>
<td>118</td>
</tr>
<tr>
<td>Create footprint</td>
<td>120</td>
</tr>
<tr>
<td>Create group</td>
<td>122</td>
</tr>
<tr>
<td>Create interval</td>
<td>123</td>
</tr>
<tr>
<td>Create job definition</td>
<td>130</td>
</tr>
<tr>
<td>Create named resource</td>
<td>150</td>
</tr>
<tr>
<td>Create resource</td>
<td>154</td>
</tr>
<tr>
<td>Create resource state definition</td>
<td>157</td>
</tr>
<tr>
<td>Create resource state mapping</td>
<td>158</td>
</tr>
<tr>
<td>Create resource state profile</td>
<td>159</td>
</tr>
<tr>
<td>Create schedule</td>
<td>160</td>
</tr>
<tr>
<td>Create scheduled event</td>
<td>162</td>
</tr>
<tr>
<td>Create scope</td>
<td>164</td>
</tr>
<tr>
<td>Create trigger</td>
<td>167</td>
</tr>
</tbody>
</table>
create user ................................................................. 177

7. deregister commands ................................. 179
deregister ............................................................. 180

8. disconnect commands .............................. 181
disconnect ........................................................... 182

9. drop commands ........................................ 183
drop comment ......................................................... 184
drop environment .................................................. 186
drop event .............................................................. 187
drop exit state definition ......................................... 188
drop exit state mapping ............................................ 189
drop exit state profile ............................................. 190
drop exit state translation ....................................... 191
drop folder ............................................................. 192
drop footprint ........................................................ 193
drop group ............................................................... 194
drop interval ............................................................ 195
drop job definition .................................................. 196
drop named resource ............................................... 197
drop resource .......................................................... 198
drop resource state definition ............................... 199
drop resource state mapping .................................... 200
drop resource state profile ..................................... 201
drop schedule ......................................................... 202
drop scheduled event .............................................. 203
drop scope ............................................................... 204
drop trigger ............................................................. 205
drop user ................................................................. 206

10. finish commands ...................................... 207
finish job ............................................................... 208

11. get commands ........................................ 209
get parameter ......................................................... 210
get submittag ........................................................ 211

12. kill commands ......................................... 213
kill session ............................................................. 214

13. link commands ........................................ 215
link resource .......................................................... 216
14. list commands 217
  list calendar ............................................. 218
  list dependency definition ........................................ 220
  list dependency hierarchy ........................................ 222
  list environment ........................................... 227
  list event ................................................ 228
  list exit state definition ................................... 229
  list exit state mapping ....................................... 230
  list exit state profile ...................................... 231
  list exit state translation .................................. 232
  list folder ............................................... 233
  list footprint .......................................... 237
  list group ........................................... 238
  list interval ........................................ 239
  list job .................................................. 241
  list job definition hierarchy ................................. 248
  list named resource ....................................... 252
  list resource state definition ................................ 254
  list resource state mapping ................................ 255
  list resource state profile ................................ 256
  list schedule ........................................ 257
  list scheduled .......................................... 259
  list scheduled event ...................................... 261
  list scope ............................................ 263
  list session ............................................. 265
  list trigger ............................................. 267
  list user .............................................. 271

15. move commands 273
  move folder .......................................... 274
  move job definition ...................................... 275
  move named resource ...................................... 276
  move schedule .......................................... 277
  move scope ........................................... 278

16. multicommand commands 279
  multicommand .......................................... 280

17. register commands 281
  register .................................................. 282

18. rename commands 283
  rename environment ...................................... 284
  rename event ........................................... 285
<table>
<thead>
<tr>
<th>Command</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>rename exit state definition</td>
<td>286</td>
</tr>
<tr>
<td>rename exit state mapping</td>
<td>287</td>
</tr>
<tr>
<td>rename exit state profile</td>
<td>288</td>
</tr>
<tr>
<td>rename exit state translation</td>
<td>289</td>
</tr>
<tr>
<td>rename folder</td>
<td>290</td>
</tr>
<tr>
<td>rename footprint</td>
<td>291</td>
</tr>
<tr>
<td>rename group</td>
<td>292</td>
</tr>
<tr>
<td>rename interval</td>
<td>293</td>
</tr>
<tr>
<td>rename job definition</td>
<td>294</td>
</tr>
<tr>
<td>rename named resource</td>
<td>295</td>
</tr>
<tr>
<td>rename resource state definition</td>
<td>296</td>
</tr>
<tr>
<td>rename resource state mapping</td>
<td>297</td>
</tr>
<tr>
<td>rename resource state profile</td>
<td>298</td>
</tr>
<tr>
<td>rename schedule</td>
<td>299</td>
</tr>
<tr>
<td>rename scope</td>
<td>300</td>
</tr>
<tr>
<td>rename trigger</td>
<td>301</td>
</tr>
<tr>
<td>rename user</td>
<td>302</td>
</tr>
</tbody>
</table>

19. **resume commands** 303
   resume                                      304

20. **select commands** 305
    select                                      306

21. **set commands** 307
    set parameter                               308

22. **show commands** 309
    show comment                                 310
    show environment                             313
    show event                                   316
    show exit state definition                    318
    show exit state mapping                       319
    show exit state profile                       321
    show exit state translation                   323
    show folder                                  325
    show footprint                               327
    show group                                   330
    show interval                                332
    show job                                     338
    show job definition                           357
    show named resource                           370
    show resource                                374
    show resource state definition                380
show resource state mapping ............................. 381
show resource state profile ............................. 383
show schedule ........................................... 385
show scheduled event .................................... 387
show scope .............................................. 389
show session ............................................ 396
show system ............................................. 398
show trigger ............................................. 401
show user ................................................ 405

23. shutdown commands ................................. 409
shutdown ................................................ 410

24. stop commands ........................................ 411
stop server ............................................... 412

25. submit commands ..................................... 413
submit ..................................................... 414

26. suspend commands ................................... 417
suspend .................................................. 418

III. Jobserver Commands ......................... 419

27. Jobserver Commands .............................. 421
alter job .................................................. 422
alter jobserver .......................................... 427
connect .................................................... 428
deregister .................................................. 431
disconnect ............................................... 432
get next job ............................................... 433
multicommand ........................................... 435
reassure .................................................... 436
register .................................................... 437

IV. Job Commands ........................................ 439

28. Job Commands ........................................ 441
alter job ................................................... 442
connect .................................................... 447
disconnect ................................................. 450
get parameter .............................................. 451
get submittag .............................................. 452
V. Programming Examples

Programming Examples

29. Programming examples
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1.</td>
<td>Valid date formats</td>
<td>20</td>
</tr>
<tr>
<td>1.2.</td>
<td>Keywords that can be used with quotes as identifiers</td>
<td>21</td>
</tr>
<tr>
<td>1.3.</td>
<td>Keywords und synonyms</td>
<td>22</td>
</tr>
<tr>
<td>1.4.</td>
<td>Reserved words</td>
<td>23</td>
</tr>
<tr>
<td>6.1.</td>
<td>job definition parameters</td>
<td>141</td>
</tr>
<tr>
<td>6.2.</td>
<td>Named Resource parameter types</td>
<td>152</td>
</tr>
<tr>
<td>6.3.</td>
<td>Named Resource usage</td>
<td>152</td>
</tr>
<tr>
<td>6.4.</td>
<td>job definition parameters</td>
<td>156</td>
</tr>
<tr>
<td>6.5.</td>
<td>List of trigger types</td>
<td>176</td>
</tr>
<tr>
<td>11.1.</td>
<td>get parameter output</td>
<td>210</td>
</tr>
<tr>
<td>11.2.</td>
<td>get submittag output</td>
<td>211</td>
</tr>
<tr>
<td>14.1.</td>
<td>list calendar output</td>
<td>219</td>
</tr>
<tr>
<td>14.2.</td>
<td>list dependency definition output</td>
<td>221</td>
</tr>
<tr>
<td>14.3.</td>
<td>list dependency hierarchy output</td>
<td>226</td>
</tr>
<tr>
<td>14.4.</td>
<td>list environment output</td>
<td>227</td>
</tr>
<tr>
<td>14.5.</td>
<td>list event output</td>
<td>228</td>
</tr>
<tr>
<td>14.6.</td>
<td>list exit state definition output</td>
<td>229</td>
</tr>
<tr>
<td>14.7.</td>
<td>list exit state mapping output</td>
<td>230</td>
</tr>
<tr>
<td>14.8.</td>
<td>list exit state profile output</td>
<td>231</td>
</tr>
<tr>
<td>14.9.</td>
<td>list exit state translation output</td>
<td>232</td>
</tr>
<tr>
<td>14.10.</td>
<td>list folder output</td>
<td>236</td>
</tr>
<tr>
<td>14.11.</td>
<td>list footprint output</td>
<td>237</td>
</tr>
<tr>
<td>14.12.</td>
<td>list group output</td>
<td>238</td>
</tr>
<tr>
<td>14.13.</td>
<td>list interval output</td>
<td>240</td>
</tr>
<tr>
<td>14.14.</td>
<td>list job output</td>
<td>247</td>
</tr>
<tr>
<td>14.15.</td>
<td>list job definition hierarchy output</td>
<td>251</td>
</tr>
<tr>
<td>14.16.</td>
<td>list named resource output</td>
<td>253</td>
</tr>
<tr>
<td>14.17.</td>
<td>list resource state definition output</td>
<td>254</td>
</tr>
<tr>
<td>14.18.</td>
<td>list resource state mapping output</td>
<td>255</td>
</tr>
<tr>
<td>14.19.</td>
<td>list resource state profile output</td>
<td>256</td>
</tr>
<tr>
<td>14.20.</td>
<td>list schedule output</td>
<td>258</td>
</tr>
<tr>
<td>14.21.</td>
<td>list scheduled output</td>
<td>260</td>
</tr>
<tr>
<td>14.22.</td>
<td>list scheduled event output</td>
<td>262</td>
</tr>
<tr>
<td>14.23.</td>
<td>list scope output</td>
<td>264</td>
</tr>
</tbody>
</table>
14.24. list session output ........................................... 266
14.25. list trigger output ........................................... 270
14.26. list user output ........................................... 272
22.1. show comment output ........................................ 312
22.2. show environment output ................................... 314
22.3. show environment RESOURCES subtable structure ....... 314
22.4. show environment JOB_DEFINITIONS subtable structure .. 315
22.5. show event output ........................................... 317
22.6. show event PARAMETERS subtable structure ............... 317
22.7. show exit state definition output ......................... 318
22.8. show exit state mapping output ............................ 319
22.9. show exit state mapping RANGES subtable structure ...... 320
22.10. show exit state profile output ............................ 322
22.11. show exit state profile STATES subtable structure ...... 322
22.12. show exit state translation output ....................... 323
22.13. show exit state translation TRANSLATION subtable structure . 324
22.14. show folder output ........................................ 326
22.15. show footprint output .................................... 328
22.16. show footprint RESOURCES subtable structure ........... 328
22.17. show footprint JOB_DEFINITIONS subtable structure ...... 329
22.18. show group output ........................................ 330
22.19. show group MANAGE_PRIVS subtable structure ........... 331
22.20. show group USERS subtable structure .................... 331
22.21. show interval output ...................................... 334
22.22. show interval SELECTION subtable structure .......... 334
22.23. show interval FILTER subtable structure ................. 334
22.24. show interval DISPATCHER subtable structure .......... 335
22.25. show interval HIERARCHY subtable structure ............ 336
22.26. show interval REFERENCES subtable structure ........... 337
22.27. show interval EDGES subtable structure ................. 337
22.28. show job output ........................................... 346
22.29. show job CHILDREN subtable structure .................. 346
22.30. show job PARENTS subtable structure .................... 347
22.31. show job PARAMETER subtable structure .................. 347
22.32. show job REQUIRED_JOBS subtable structure ............ 350
22.33. show job DEPENDENT_JOBS subtable structure .......... 352
22.34. show job REQUIRED_RESOURCES subtable structure ...... 354
22.35. show job AUDIT_TRAIL subtable structure ............... 354
22.36. show job DEFINED_RESOURCES subtable structure ...... 355
22.37. show job RUNS subtable structure ....................... 356
22.38. show job definition output ............................... 362
22.39. show job definition CHILDREN subtable structure ...... 363
22.40. show job definition PARENTS subtable structure . . . . . . . . . . 365
22.41. show job definition REQUIRED_JOBS subtable structure . . . . . 366
22.42. show job definition DEPENDENT_JOBS subtable structure . . . 368
22.43. show job definition REQUIRED_RESOURCES subtable structure . 369
22.44. show named resource output ................................................... 371
22.45. show named resource RESOURCES subtable structure .......... 372
22.46. show named resource PARAMETERS subtable structure ......... 372
22.47. show named resource JOB_DEFINITIONS subtable structure .. 373
22.48. show resource output ............................................................ 376
22.49. show resource ALLOCATIONS subtable structure .............. 377
22.50. show resource PARAMETERS subtable structure ............... 379
22.51. show resource state definition output ................................. 380
22.52. show resource state mapping output ...................................... 381
22.53. show resource state mapping MAPPINGS subtable structure .. 382
22.54. show resource state profile output ....................................... 384
22.55. show resource state profile STATES subtable structure ....... 384
22.56. show schedule output ............................................................. 386
22.57. show scheduled event output ................................................ 388
22.58. show scope output ................................................................. 391
22.59. show scope RESOURCES subtable structure ......................... 392
22.60. show scope CONFIG subtable structure ................................. 393
22.61. show scope CONFIG_ENVMAPPING subtable structure .......... 393
22.62. show scope PARAMETERS subtable structure ....................... 395
22.63. show session output ............................................................... 397
22.64. show system output ............................................................... 399
22.65. show system WORKER subtable structure ............................. 400
22.66. show trigger output ............................................................... 403
22.67. show trigger STATES subtable structure ............................... 404
22.68. show trigger PARAMETERS subtable structure ..................... 404
22.69. show user output ................................................................. 406
22.70. show user MANAGE_PRIVS subtable structure ..................... 406
22.71. show user GROUPS subtable structure ................................. 407
22.72. show user EQUIVALENT_USERS subtable structure ............. 407
22.73. show user COMMENT subtable structure ............................... 407

25.1. submit output ................................................................. 416

27.1. get next job output ............................................................ 434

28.1. get parameter output .......................................................... 451
28.2. get submittag output ........................................................... 452
28.3. submit output ................................................................. 458
Part I.

General
1. Introduction

Introduction

Essentially, this document is divided into three parts. In the BICsuite Scheduling System, there are three types of users (in the broadest sense of the word):

- Users
- Jobservers
- Jobs

Each of these users has his own command set at his disposal. These command sets only overlap to a certain extent. For example, for jobservers there is the statement `get next job`, which is not valid for either jobs or users. On the other hand, there are forms of the `submit` statement which will only make sense in a job context and which can therefore only be implemented by jobs. Obviously only users are allowed to create objects such as Exit State definitions or job definitions. In contrast, there are also statements such as the `connect` statement which is valid for all types of users.

The structure of this document is oriented to the three types of users. The largest part of this document deals with the user commands, while the two other parts handle job servers and job commands.

For the sake of completeness, the next chapter briefly explains the utility `sdmsh`. This utility is easy to use and is an excellent choice for processing scripts using BICsuite commands.

Since the syntax described here is the only interface to the BICsuite Scheduling Server, all the utilities (and in particular BICsuite!Web) use this web interface.

To simplify the development of proprietary utilities, the server is capable of returning its reactions to statements in various formats. The utility `sdmsh`, for example, uses the `serial` protocol, with which serialised Java objects are transferred. In contrast BICsuite!Web uses the `python` protocol, with which textual representations of Python structures are transferred that can be easily read in using the `eval()` function.
**Syntax diagrams**

The syntax diagrams are comprised of different symbols and metasymbols. The symbols and metasymbols are listed and explained in the table below.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>keyword</td>
<td>A keyword in the language. These have to be entered as shown. One example is the keyword <code>create</code>.</td>
</tr>
<tr>
<td>name</td>
<td>A parameter. In many cases, the user can choose a name or a number to be entered here.</td>
</tr>
<tr>
<td>NONTERM</td>
<td>A non-terminal symbol is represented by <code>SMALL CAPS</code>. A syntax element that is explained further on in the diagram has to be inserted here.</td>
</tr>
<tr>
<td>`&lt; all</td>
<td>any &gt;`</td>
</tr>
<tr>
<td>`&lt; all</td>
<td>any &gt;`</td>
</tr>
<tr>
<td><code>[ or alter ]</code></td>
<td>Optional syntax elements are placed in square brackets.</td>
</tr>
<tr>
<td><code>{ statename }</code></td>
<td>Syntax elements that are placed in braces are repeated 0 to n times.</td>
</tr>
<tr>
<td>JOB_PARAMETER</td>
<td>Cases where elements occur at least once are far more common and are shown as represented here.</td>
</tr>
<tr>
<td><code>, JOB_PARAMETER</code></td>
<td>In lists of possible syntax elements, the single possibilities are separated by a <code>,</code>. Such a list is another way of displaying optional choices. These two different forms of presentation are used for purposes of clarity.</td>
</tr>
</tbody>
</table>
Literals

Literals are only required in the language definition for strings, numbers, and dates/times.

Strings are delimited by single quotes, as in

\[
\text{node} = '\text{puma.independit.de}'
\]

Integers are shown as either unsigned integer or signed signed_integer in the syntax diagrams. A signed_integer can be prefixed with a + or - sign. Valid unsigned integers lie in the range of numbers between \(0\) and \(2^{31} - 1\). Signed integers are therefore within the range between \(-2^{31} + 1\) and \(2^{31} - 1\). If the syntax diagram contains \(id\), an unsigned integer between \(0\) and \(2^{63} - 1\) is expected here.

Much more complicated are dates/times, particularly in statements concerning the time scheduling. These literals are principally shown as strings with a special format.

The following syntax is used to comply with the notations based on ISO8601 as given in Table 1.1:

<table>
<thead>
<tr>
<th>String</th>
<th>Meaning</th>
<th>Range</th>
<th>String</th>
<th>Meaning</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>YYYY</td>
<td>year</td>
<td>1970 .. 9999</td>
<td>hh</td>
<td>hour</td>
<td>00 .. 23</td>
</tr>
<tr>
<td>MM</td>
<td>month</td>
<td>01 .. 12</td>
<td>mm</td>
<td>minute</td>
<td>00 .. 59</td>
</tr>
<tr>
<td>DD</td>
<td>day (of the month)</td>
<td>01 .. 31</td>
<td>ss</td>
<td>second</td>
<td>00 .. 59</td>
</tr>
<tr>
<td>ww</td>
<td>week (of the year)</td>
<td>01 .. 53</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- All other strings stand by themselves.
- No differentiation is made between uppercase and lowercase.
- The earliest permissible point in time is 1970–01-01T00:00:00 GMT.
Identifier

In the BICsuite Scheduling System, objects are identified by their names. (Strictly speaking, objects can also be identified from their internal Id, which is a number, but this practice is not recommended). Valid names comprise a letter, underscore (_), at sign (@) or hash sign (#) followed by numbers, letters or special characters. Language-specific special characters such as the German umlaut are invalid.

Identifiers are treated as being case-insensitive if they are not enclosed in simple quotes. Identifiers enclosed in quotes are case-sensitive. It is therefore not generally recommended to use quotes unless there is a valid reason for doing so.

Identifiers that are allowed to be enclosed in single quotes can also contain spaces and several special characters. Again, this practice is not recommended as spaces are normally interpreted as delimiters and therefore errors can easily occur. Spaces aren’t allowed at the beginning or end of an identifier.

There are a number of keywords in the syntax that cannot be readily used as identifiers. Here it may be practicable to use quotes so that the identifiers are not recognised as keywords. Table 1.2 contains a list of such keywords.

<table>
<thead>
<tr>
<th>Format</th>
<th>Example</th>
<th>Simplified Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>−MM−DDThh:mm:ss</td>
<td>−05−02T07:55:12</td>
<td>−MMDDThhmmss</td>
</tr>
<tr>
<td>−−DD</td>
<td>−−02</td>
<td>−−DDThhmm</td>
</tr>
<tr>
<td>−−DDThh</td>
<td>−−02T07</td>
<td>−−DDThhmmss</td>
</tr>
<tr>
<td>−−DDThh:mm</td>
<td>−−02T07:55</td>
<td></td>
</tr>
<tr>
<td>−−DDThh:mm:ss</td>
<td>−−02T07:55:12</td>
<td>−−DDThhmmss</td>
</tr>
<tr>
<td>Thh</td>
<td>T07</td>
<td></td>
</tr>
<tr>
<td>Thh:mm</td>
<td>T07:55</td>
<td>Thhmm</td>
</tr>
<tr>
<td>Thh:mm:ss</td>
<td>T07:55:12</td>
<td>Thhmmss</td>
</tr>
<tr>
<td>T−mm</td>
<td>T−55</td>
<td></td>
</tr>
<tr>
<td>T−mm:ss</td>
<td>T−55:12</td>
<td>T−mmss</td>
</tr>
<tr>
<td>T−−ss</td>
<td>T−−12</td>
<td></td>
</tr>
<tr>
<td>YYYYWww</td>
<td>1990W18</td>
<td></td>
</tr>
<tr>
<td>Wwww</td>
<td>W18</td>
<td></td>
</tr>
</tbody>
</table>

Table 1.1.: Valid date formats
<table>
<thead>
<tr>
<th>activate</th>
<th>delay</th>
<th>group</th>
<th>milestone</th>
<th>rawpassword</th>
<th>submitcount</th>
</tr>
</thead>
<tbody>
<tr>
<td>active</td>
<td>delete</td>
<td>header</td>
<td>minute</td>
<td>read</td>
<td>sum</td>
</tr>
<tr>
<td>action</td>
<td>dependency</td>
<td>history</td>
<td>mode</td>
<td>reassure</td>
<td>suspend</td>
</tr>
<tr>
<td>add</td>
<td>deregister</td>
<td>hour</td>
<td>move</td>
<td>recursive</td>
<td>sx</td>
</tr>
<tr>
<td>after</td>
<td>dir</td>
<td>identified</td>
<td>multiplier</td>
<td>required</td>
<td>synchronizing</td>
</tr>
<tr>
<td>alter</td>
<td>disable</td>
<td>ignore</td>
<td>name</td>
<td>requestable</td>
<td>syntime</td>
</tr>
<tr>
<td>amount</td>
<td>disconnect</td>
<td>import</td>
<td>nicevalue</td>
<td>rerun</td>
<td>tag</td>
</tr>
<tr>
<td>and</td>
<td>distribution</td>
<td>in</td>
<td>node</td>
<td>restartable</td>
<td>test</td>
</tr>
<tr>
<td>avg</td>
<td>drop</td>
<td>inactive</td>
<td>noinverse</td>
<td>resume</td>
<td>timeout</td>
</tr>
<tr>
<td>base</td>
<td>dump</td>
<td>infinite</td>
<td>nomaster</td>
<td>revoke</td>
<td>timestamp</td>
</tr>
<tr>
<td>batch</td>
<td>duration</td>
<td>interval</td>
<td>noremerge</td>
<td>rollback</td>
<td>to</td>
</tr>
<tr>
<td>before</td>
<td>dynamic</td>
<td>inverse</td>
<td>nonfatal</td>
<td>run</td>
<td>touch</td>
</tr>
<tr>
<td>broken</td>
<td>edit</td>
<td>is</td>
<td>nosuspend</td>
<td>runnable</td>
<td>trigger</td>
</tr>
<tr>
<td>by</td>
<td>embedded</td>
<td>ix</td>
<td>notrace</td>
<td>running</td>
<td>trunc</td>
</tr>
<tr>
<td>cancel</td>
<td>enable</td>
<td>job</td>
<td>notrun</td>
<td>runtime</td>
<td>type</td>
</tr>
<tr>
<td>cancelled</td>
<td>endtime</td>
<td>kill</td>
<td>nowarn</td>
<td>s</td>
<td>update</td>
</tr>
<tr>
<td>cascade</td>
<td>environment</td>
<td>killed</td>
<td>of</td>
<td>scope</td>
<td>unreachable</td>
</tr>
<tr>
<td>change</td>
<td>erlog</td>
<td>level</td>
<td>offline</td>
<td>selection</td>
<td>unresolved</td>
</tr>
<tr>
<td>check</td>
<td>error</td>
<td>liberal</td>
<td>on</td>
<td>serial</td>
<td>year</td>
</tr>
<tr>
<td>child</td>
<td>event</td>
<td>like</td>
<td>online</td>
<td>server</td>
<td></td>
</tr>
<tr>
<td>children</td>
<td>execute</td>
<td>limits</td>
<td>only</td>
<td>session</td>
<td></td>
</tr>
<tr>
<td>childsuspend</td>
<td>expand</td>
<td>line</td>
<td>or</td>
<td>set</td>
<td></td>
</tr>
<tr>
<td>childtag</td>
<td>expired</td>
<td>local</td>
<td>owner</td>
<td>shutdown</td>
<td></td>
</tr>
<tr>
<td>clear</td>
<td>factor</td>
<td>lockmode</td>
<td>parameters</td>
<td>show</td>
<td></td>
</tr>
<tr>
<td>command</td>
<td>failure</td>
<td>logfile</td>
<td>password</td>
<td>sort</td>
<td></td>
</tr>
<tr>
<td>comment</td>
<td>fatal</td>
<td>map</td>
<td>path</td>
<td>started</td>
<td></td>
</tr>
<tr>
<td>condition</td>
<td>filter</td>
<td>maps</td>
<td>pending</td>
<td>starting</td>
<td></td>
</tr>
<tr>
<td>connect</td>
<td>final</td>
<td>mapping</td>
<td>performance</td>
<td>starttime</td>
<td></td>
</tr>
<tr>
<td>constant</td>
<td>finish</td>
<td>master</td>
<td>perl</td>
<td>static</td>
<td></td>
</tr>
<tr>
<td>content</td>
<td>finished</td>
<td>max</td>
<td>pid</td>
<td>status</td>
<td></td>
</tr>
<tr>
<td>copy</td>
<td>folder</td>
<td>mapping</td>
<td>pool</td>
<td>stop</td>
<td></td>
</tr>
<tr>
<td>count</td>
<td>footprint</td>
<td>master_id</td>
<td>priority</td>
<td>strict</td>
<td></td>
</tr>
<tr>
<td>create</td>
<td>for</td>
<td>master</td>
<td>profile</td>
<td>submit</td>
<td></td>
</tr>
<tr>
<td>cycle</td>
<td>force</td>
<td>master_id</td>
<td>protocol</td>
<td></td>
<td></td>
</tr>
<tr>
<td>day</td>
<td>free_amount</td>
<td>max</td>
<td>public</td>
<td></td>
<td></td>
</tr>
<tr>
<td>default</td>
<td>from</td>
<td>min</td>
<td>merged</td>
<td></td>
<td></td>
</tr>
<tr>
<td>definition</td>
<td>get</td>
<td>merge</td>
<td>python</td>
<td></td>
<td></td>
</tr>
<tr>
<td>defer</td>
<td>grant</td>
<td>merged</td>
<td>xml</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1.2.: Keywords that can be used with quotes as identifiers
There are also a number of synonyms. These are essentially keywords that can be written in more than one way. Only one spelling variation is shown in Table 1.2. The synonyms can be used together arbitrarily. Table 1.3 gives a list of such synonyms.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Synonym</th>
<th>Keyword</th>
<th>Synonym</th>
</tr>
</thead>
<tbody>
<tr>
<td>definition</td>
<td>definitions</td>
<td>minute</td>
<td>minutes</td>
</tr>
<tr>
<td>dependency</td>
<td>dependencies</td>
<td>month</td>
<td>months</td>
</tr>
<tr>
<td>environment</td>
<td>environments</td>
<td>node</td>
<td>nodes</td>
</tr>
<tr>
<td>errlog</td>
<td>errlogfile</td>
<td>parameter</td>
<td>parameters</td>
</tr>
<tr>
<td>event</td>
<td>events</td>
<td>profile</td>
<td>profiles</td>
</tr>
<tr>
<td>folder</td>
<td>folders</td>
<td>resource</td>
<td>resources</td>
</tr>
<tr>
<td>footprint</td>
<td>footprints</td>
<td>schedule</td>
<td>schedules</td>
</tr>
<tr>
<td>grant</td>
<td>grants</td>
<td>scope</td>
<td>scopes</td>
</tr>
<tr>
<td>group</td>
<td>groups</td>
<td>server</td>
<td>servers</td>
</tr>
<tr>
<td>hour</td>
<td>hours</td>
<td>session</td>
<td>sessions</td>
</tr>
<tr>
<td>infinit</td>
<td>infinite</td>
<td>state</td>
<td>states, status</td>
</tr>
<tr>
<td>interval</td>
<td>intervals</td>
<td>translation</td>
<td>translations</td>
</tr>
<tr>
<td>job</td>
<td>jobs</td>
<td>user</td>
<td>users</td>
</tr>
<tr>
<td>mapping</td>
<td>mappings</td>
<td>week</td>
<td>weeks</td>
</tr>
<tr>
<td>milestone</td>
<td>milestones</td>
<td>year</td>
<td>years</td>
</tr>
</tbody>
</table>

Table 1.3.: Keywords und synonyms

As in any language, there are also some reserved words and word combinations. An overview is shown in Table 1.4. A special characteristic of word pairs is that replacing a space with an underscore likewise produces a reserved word. The word named_resource is therefore reserved (but “named#resource” isn’t).

<table>
<thead>
<tr>
<th>after final</th>
<th>exit state translation</th>
<th>non fatal</th>
</tr>
</thead>
<tbody>
<tr>
<td>all final</td>
<td>ext pid</td>
<td>requestable amount</td>
</tr>
<tr>
<td>backlog handling</td>
<td>finish child</td>
<td>resource state</td>
</tr>
<tr>
<td>before final</td>
<td>free amount</td>
<td>resource state definition</td>
</tr>
<tr>
<td>begin multicommand</td>
<td>get next job</td>
<td>resource state mapping</td>
</tr>
<tr>
<td>broken active</td>
<td>ignore dependency</td>
<td>resource state profile</td>
</tr>
<tr>
<td>broken finished</td>
<td>immediate local</td>
<td>resource template</td>
</tr>
<tr>
<td>change state</td>
<td>immediate merge</td>
<td>resource wait</td>
</tr>
<tr>
<td>default mapping</td>
<td>initial state</td>
<td>run program</td>
</tr>
<tr>
<td>dependency definition</td>
<td>job definition</td>
<td>rerun program</td>
</tr>
<tr>
<td>dependency hierarchy</td>
<td>job definition hierarchy</td>
<td>scheduled event</td>
</tr>
<tr>
<td>dependency mode</td>
<td>job final</td>
<td>state profile</td>
</tr>
<tr>
<td>dependency wait</td>
<td>job server</td>
<td>status mapping</td>
</tr>
<tr>
<td>end multicommand</td>
<td>job state</td>
<td>suspend limit</td>
</tr>
</tbody>
</table>

Continued on next page
Editions

There are three editions of the BICsuite Scheduling System. Since features from later editions are not always present in the earlier editions, the relevant statements or options within the statements are designated accordingly. A letter in the top corner of the page indicates for which edition of the system this statement is available. Deviations from the general statement are shown in the syntax diagram. The symbols have the following meanings:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>This symbol indicates a feature in the Basic version and all later versions.</td>
</tr>
<tr>
<td>P</td>
<td>This symbol indicates a feature in the Professional and Enterprise versions and all later versions.</td>
</tr>
<tr>
<td>E</td>
<td>This symbol indicates a feature in the Enterprise version.</td>
</tr>
</tbody>
</table>

Table 1.4.: Reserved words
## 2. Utilities

### Starting and stopping the server

#### server-start

**Introduction**

The utility `server-start` is used to start the scheduling server.

**Call**

The following commands are used to call `server-start`:

```
server-start [ OPTIONS ] config-file
```

<table>
<thead>
<tr>
<th>OPTIONS:</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>-admin</td>
<td>The server starts in &quot;admin&quot; mode. This means that user logins are disabled apart from the user SYSTEM.</td>
</tr>
<tr>
<td>-protected</td>
<td>&quot;protected&quot; mode is similar to Admin mode. The difference here is that the internal threads (TimerThread and SchedulingThread) are not started. This allows administrative tasks to be carried out without concurrent transactions being performed.</td>
</tr>
</tbody>
</table>

If the server has already been started, the second server either (depending on the configuration) takes over the operation or repeatedly makes an unsuccessful attempt to start.

The `server-start` utility can be only be used by the user whose Id was used to install the system.
server-stop

Introduction

Introduction  The server-stop utility is used to stop the scheduling server.

Call

Call  The following command is used to call server-stop:

server-stop

Initially, an attempt is made to stop the server ‘gracefully’. First, all the user connections are terminated to stop all the internal threads.
If this approach fails or it takes too long, the server is stopped using the operating system’s mechanisms.
If the server has not been started, the server-stop command has no effect.
The server-stop utility can be only be used by the user whose Id was used to install the system.
sdmsh

Introduction

The sdms sister utility is a small program that enables the user to interactively work with the scheduling server. In contrast to the BICsuite!Web front end, for instance, this working method is text-oriented. This means it is possible to write scripts and execute them using sdms.

The sdms executable is a small script (or batch file) that encapsulates the call of the required Java program. Of course, there is no reason why this Java program should not be called manually. It is only there for convenience’s sake.

Call

The following commands are used to call sdms:

```
sdms [ OPTIONS ] [ username [ password [ host [ port ] ] ] ]
```

OPTIONS:

- `--host` or `-h` *hostname*
- `--port` or `-p` *portnumber*
- `--user` or `-u` *username*
- `--pass` or `-w` *password*
- `--jid` or `-j` *jobid*
- `--key` or `-k` *jobkey*
- `--no |silent | - | no |s >
- `--no |verbose | - | no |v >
- `--ini` or `-ini` *inifile*
- `--no |tls | - | no |tls >
- `--no |help`
- `--info sessioninfo`
- `- | no |S`
- `--timeout timeout`

The individual options have the following meanings:
<table>
<thead>
<tr>
<th>Option</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>`&lt; --host</td>
<td>-h &gt; hostname`</td>
</tr>
<tr>
<td>`&lt; --port</td>
<td>-p &gt; portnumber`</td>
</tr>
<tr>
<td>`&lt; --user</td>
<td>-u &gt; username`</td>
</tr>
<tr>
<td>`&lt; --pass</td>
<td>-w &gt; password`</td>
</tr>
<tr>
<td>`&lt; --jid</td>
<td>-j &gt; jobid`</td>
</tr>
<tr>
<td>`&lt; --key</td>
<td>-k &gt; jobkey`</td>
</tr>
<tr>
<td>`&lt;--[ no ]silent</td>
<td>-[ no ]s &gt;`</td>
</tr>
<tr>
<td>`&lt;--[ no ]verbose</td>
<td>-[ no ]v &gt;`</td>
</tr>
<tr>
<td>`&lt;--ini</td>
<td>-ini &gt; inifile`</td>
</tr>
<tr>
<td>`&lt;--[ no ]tls</td>
<td>-[ no ]tls &gt;`</td>
</tr>
<tr>
<td><code>--help</code></td>
<td>Return a help text</td>
</tr>
<tr>
<td><code>--info sessioninfo</code></td>
<td>Set the accompanying information as descriptive information about the session</td>
</tr>
<tr>
<td><code>-[ no ]S</code></td>
<td>Silent option. This option is obsolete and exists for reasons of backward compatibility</td>
</tr>
<tr>
<td><code>--timeout timeout</code></td>
<td>The number of seconds after which the server terminates an idle session. The value 0 means no timeout</td>
</tr>
</tbody>
</table>

`sdmsh` obviously requires information to connect to the correct BICsuite Scheduling System. The necessary data can be specified in the command line or by using an options file. Missing values for the user name and password are queried by `sdmsh`. If values for the host and port are not given, the defaults values “localhost” and 2506 are used. It is not advisable to specify the password in the command line because this information can frequently be easily read out by other users.

**Options file**

The *options file* has the same format as a Java property file. (Please refer to the official Java documentation for details of the precise syntax specification.)

The following option files play a role:

- `$SDMSCONFIG/sdmshrc`
- `$HOME/.sdmshrc`
- Optionally, a file specified in the command line
The files are valuated in the given order. If options are present in several files, the value in the last valuated file “wins”. Options that are specified in the command line take precedence over all the other specifications.

The following keywords are recognised:

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>User</td>
<td>The user’s name</td>
</tr>
<tr>
<td>Password</td>
<td>The user’s password</td>
</tr>
<tr>
<td>Host</td>
<td>Name or IP address of the host</td>
</tr>
<tr>
<td>Info</td>
<td>Additional information for identifying a connection is set</td>
</tr>
<tr>
<td>Port</td>
<td>Port number of the scheduling server (default: 2506)</td>
</tr>
<tr>
<td>Silent</td>
<td>(Error) messages are not returned</td>
</tr>
<tr>
<td>Timeout</td>
<td>Timeout value for the session (0 means no timeout)</td>
</tr>
<tr>
<td>TLS</td>
<td>Use an SSL/TLS connection</td>
</tr>
<tr>
<td>Verbose</td>
<td>Commands, feedback and additional messages are returned</td>
</tr>
</tbody>
</table>

Since the user’s password is shown in plain text in this file, careful consideration needs to be taken when assigning the access privileges for this file. It is, of course, possible to not specify the password and to enter it every time sdms/h is started.

Only the following keywords can be used in configuration files:

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>KeyStore</td>
<td>Keystore for TLS/SSL communication</td>
</tr>
<tr>
<td>TrustStore</td>
<td>Truststore for TLS/SSL communication</td>
</tr>
<tr>
<td>KeyStorePassword</td>
<td>Keystore password</td>
</tr>
<tr>
<td>TrustStorePassword</td>
<td>Truststore password</td>
</tr>
</tbody>
</table>

**Internal commands**

Apart from the BICsuite commands described in the following chapters, sdms/h also knows a few simple commands of its own. These are briefly described below. Internal commands do not have to be closed with a semicolon.

**disconnect** The disconnect command is used to exit sdms/h. Because different commands are commonly used to exit a tool in different work environments, an attempt was made here to incorporate many varying formulations. The syntax for the disconnect command is:
< disconnect | bye | exit | quit >

**Example** Here is an example of the `disconnect` command.

```
ronald@jaguarundi:~$ sdmsh
Connect

CONNECT_TIME : 23 Aug 2007 07:13:30 GMT

Connected

[system@localhost:2506] SDMS> disconnect
ronald@jaguarundi:~$
```

**echo** If `sdmsh` is being used interactively, it is visually evident which command has just been entered. This is not the case in batch mode, i.e. when processing a script. The `echo` command can be used to enable and disable the rendering of the entered statement. This is enabled by default. The syntax for the `echo` command is:

```
echo < on | off >
```

**Example** The effect of these two options is shown below. Following the command `echo on`

```
[system@localhost:2506] SDMS> echo on
End of Output

[system@localhost:2506] SDMS> show session;
show session;

Session

    THIS : *
SESSIONID : 1001
START : Tue Aug 23 11:47:34 GMT+01:00 2007
USER : SYSTEM
UID : 0
IP : 127.0.0.1
TXID : 136448
IDLE : 0
TIMEOUT : 0
STATEMENT : show session

Session shown

[system@localhost:2506] SDMS> echo off
```

30
[system@localhost:2506] SDMS> show session;

Session

    THIS : *
SESSIONID : 1001
START : Tue Aug 23 11:47:34 GMT+01:00 2007
USER : SYSTEM
UID : 0
IP : 127.0.0.1
TXID : 136457
IDLE : 0
TIMEOUT : 0
STATEMENT : show session

Session shown

[system@localhost:2506] SDMS>

**help**  The *help* command opens a condensed help text about the internal *sdmsh* commands. The syntax for the *help* command is:

```
help
```

**EXAMPLE**  The *help* command only returns a condensed help text about the syntax for the internal *sdmsh* commands. This is shown in the example below. (The lines have been wrapped for this document and so the actual output may differ to what is written here).

[system@localhost:2506] SDMS> help
Condensed Help Feature
----------------------
Internal sdms(h) Commands:
disconnect|bye|exit|quit -- leaves the tool
echo on|off -- controls whether the statement text is printed or not
help -- gives this output
include '<filespec>' -- reads sdms(h) commands from the given file
prompt '<somestring>' -- sets to prompt to the specified value %H = hostname, %P = port, %U = user, %% = %
timing on|off -- controls whether the actual time is printed or not
whenever error
continue|disconnect <integer> -- specifies the behaviour of the program
in case of an error

<!shellcommand> -- executes the specified command. sdmsh has no intelligence at all regarding terminal I/O

End of Output

[system@localhost:2506] SDMS>

**include** Files can be integrated into BICsuite statements using the *include* command.
The syntax for the *include* command is:

```
include 'filespec'
```

**EXAMPLE** In the following example, a file only containing the command "show session;" is inserted.

```
[system@localhost:2506] SDMS> include '/tmp/show.sdms'
Session

  THIS : *
SESSIONID : 1001
START : Tue Aug 23 11:47:34 GMT+01:00 2007
USER : SYSTEM
UID : 0
IP : 127.0.0.1
TXID : 136493
IDLE : 0
TIMEOUT : 0
STATEMENT : show session

Session shown
```

**prompt** The *prompt* command can be used to specify an arbitrary prompt. There are a number of variable values that can be inserted automatically by the program. The codes for the individual variables are shown in the table below:

<table>
<thead>
<tr>
<th>Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>%H</td>
<td>Hostname des Scheduling Servers</td>
</tr>
<tr>
<td>%P</td>
<td>TCP/IP Port</td>
</tr>
<tr>
<td>%U</td>
<td>Username</td>
</tr>
<tr>
<td>%%</td>
<td>Percent character (％)</td>
</tr>
</tbody>
</table>
The default prompt has the following definition: [%U@%H:%P] SDMS>.  
The syntax for the prompt command is:

`prompt 'somestring'`

**Example** In the following example, an empty prompt is defined first. A BICsuite statement is then executed to make the effect more clearly visible. A simple string is then selected as a prompt, and finally the variables are used.

```
[system@localhost:2506] SDMS> prompt ''
End of Output
show session;
show session;
```

```
Session

    THIS : *
SESSIONID : 1001
START : Tue Aug 23 11:47:34 GMT+01:00 2007
USER : SYSTEM
UID : 0
IP : 127.0.0.1
TXID : 136532
IDLE : 0
TIMEOUT : 0
STATEMENT : show session

Session shown
prompt 'hello world '
End of Output
```

```
hello world prompt '[%U@%H:%P] please enter your wish! > '

End of Output

[system@localhost:2506] please enter your wish! >
```

**Timing** The timing command provides information about the execution time for a statement. Normally, this option is disabled and so no information about the execution time is given. The time is stated in milliseconds. The syntax for the timing command is:

```
timing < off | on >
```
**Example** The following example shows the timing information for a simple BIC-suite statement. The execution time for the statements and the time that was required to output the result is shown.

```
[system@localhost:2506] SDMS> timing on
End of Output

[system@localhost:2506] SDMS> show session;
Execution Time: 63
show session;

Session

    THIS : *
SESSIONID : 1002
START : Tue Aug 23 11:53:15 GMT+01:00 2007
USER : SYSTEM
UID : 0
IP : 127.0.0.1
TXID : 136559
IDLE : 0
TIMEOUT : 0
STATEMENT : show session

Session shown

[system@localhost:2506] SDMS>
Render Time : 143
```

**whenever** An error handling routine is absolutely essential particularly when *sdmsh* is being used to execute scripts. The *whenever* statement tells *sdmsh* how to deal with errors. By default errors are ignored, which also corresponds to the desired behaviour for interactive working. The syntax for the *whenever* command is:

```
whenever error < continue | disconnect integer >
```

**Example** The example below shows the behaviour of both the *continue* option and the *disconnect* option. The Exit Code for a process that was started by the Bourne shell (and other Unix shells as well) can be shown by outputting the variable `$?`.

```
[system@localhost:2506] SDMS> whenever error continue
End of Output

[system@localhost:2506] SDMS> show exit state definition does_not_exist;
show exit state definition does_not_exist;
ERROR:03201292040, DOES_NOT_EXIST not found
```
Shell call  It frequently happens that a shell command has to be quickly executed, for instance to see what the file that is to be run (using include) is called. If no special capabilities are required of the terminal, such as is the case when calling an editor, a shell command can be executed by prefixing it with an exclamation mark.

The syntax for a shell call is:

`!shellcommand`

**EXAMPLE** In the following example, a short list of all the *sdms* scripts in the /tmp directory is outputted.

```
[system@localhost:2506] SDMS> !ls -l /tmp/*.sdms
-rw-r--r-- 1 ronald ronald 15 2007-08-23 09:30 /tmp/ls.sdms
End of Output
```

[system@localhost:2506] SDMS>
**sdms-auto_restart**

*Introduction*

The utility *sdms-auto_restart* is used to automatically restart jobs that have failed. A number of simple conditions have to be met to do this. Probably the most important condition is that the job defines a parameter AUTORESTART with the value TRUE. This parameter can naturally also be set at a higher level.

The following parameters influence the behaviour of the AUTORESTART utilities:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>AUTORESTART</td>
<td>The autorestart only functions if this parameter is set to “TRUE”</td>
</tr>
<tr>
<td>AUTORESTART_MAX</td>
<td>Defines the maximum number of automatic restarts if set</td>
</tr>
<tr>
<td>AUTORESTART_COUNT</td>
<td>Is set by the autorestart utility to count the number of restarts</td>
</tr>
<tr>
<td>AUTORESTART_DELAY</td>
<td>The time in minutes before a job is restarted</td>
</tr>
</tbody>
</table>

The AUTORESTART utility can be defined as a trigger. The trigger types IMMEDIATE_LOCAL and FINISH_CHILD can be used. The logic of the option files that applies for the *sdms* utility is also used for *sdms-auto_restart*.

*Call*

The following commands are used to call *sdms-auto_restart*:

```
sdms-auto_restart [OPTIONS] < --host | -h > hostname
< --port | -p > portnumber < --user | -u > username
< --pass | -w > password < --failed | -f > jobid
```

**OPTIONS:**

- `<--silent | -s>`
- `<--verbose | -v>`
- `<--timeout | -t > minutes`
- `<--cycle | -c > minutes`
- `<--help | -h>`
- `<--delay | -d > seconds`
- `<--max | -m > number`
- `<--warn | -W>`
The individual options have the following meanings:

<table>
<thead>
<tr>
<th>Option</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; --host</td>
<td>h &gt; hostname</td>
</tr>
<tr>
<td>&lt; --port</td>
<td>p &gt; portnumber</td>
</tr>
<tr>
<td>&lt; --user</td>
<td>u &gt; username</td>
</tr>
<tr>
<td>&lt; --pass</td>
<td>w &gt; password</td>
</tr>
<tr>
<td>&lt; --failed</td>
<td>-f &gt; jobid</td>
</tr>
<tr>
<td>&lt; --silent</td>
<td>-s &gt;</td>
</tr>
<tr>
<td>&lt; --verbose</td>
<td>-v &gt;</td>
</tr>
<tr>
<td>&lt; --timeout</td>
<td>-t &gt; minutes</td>
</tr>
<tr>
<td>&lt; --cycle</td>
<td>-c &gt; minutes</td>
</tr>
<tr>
<td>&lt; --help</td>
<td>-h &gt;</td>
</tr>
<tr>
<td>&lt; --delay</td>
<td>-d &gt; minutes</td>
</tr>
<tr>
<td>&lt; --max</td>
<td>-m &gt; number</td>
</tr>
<tr>
<td>&lt; --warn</td>
<td>-W &gt;</td>
</tr>
</tbody>
</table>
sdms-get_variable

Introduction

The utility sdms-get_variable offers a simple way of reading out job parameters from the scheduling system. The logic of the option files that applies for the sdmsh utility is also used for sdms-get_variable.

Call

The following commands are used to call sdms-get_variable:

```
sdms-get_variable [ OPTIONS ] < --host | -h > hostname
< --port | -p > portnumber < --jid | -j > jobid
< --name | -n > parametername

OPTIONS:
< --user | -u > username
< --pass | -w > password
< --key | -k > jobkey
< --silent | -s >
< --verbose | -v >
< --timeout | -t > minutes
< --cycle | -c > minutes
< --help | -h >
< --mode | -m > mode
```

The individual options have the following meanings:

<table>
<thead>
<tr>
<th>Option</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; --host</td>
<td>-h &gt; hostname</td>
</tr>
<tr>
<td>&lt; --port</td>
<td>-p &gt; portnumber</td>
</tr>
<tr>
<td>&lt; --user</td>
<td>-u &gt; username</td>
</tr>
<tr>
<td>&lt; --pass</td>
<td>-w &gt; password</td>
</tr>
<tr>
<td>&lt; --key</td>
<td>-k &gt; jobkey</td>
</tr>
<tr>
<td>&lt; --silent</td>
<td>-s &gt;</td>
</tr>
</tbody>
</table>

Continued on the next page
<table>
<thead>
<tr>
<th>Option</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; --verbose</td>
<td>-v &gt;</td>
</tr>
<tr>
<td>&lt; --timeout</td>
<td>-t &gt; minutes</td>
</tr>
<tr>
<td>&lt; --cycle</td>
<td>-c &gt; minutes</td>
</tr>
<tr>
<td>&lt; --help</td>
<td>-h &gt;</td>
</tr>
<tr>
<td>&lt; --mode</td>
<td>-m &gt; mode</td>
</tr>
</tbody>
</table>

**Example**

The example below shows how to get the variable value of the variable RESPONSE of job 5175119.

```
ronald@cheetah:~$ sdms-get_variable -h localhost -p 2506 \ 
    -j 5175119 -u donald -w duck -n RESPONSE
```

Example
sdms-rerun

Introduction

The utility sdms-rerun is used to rerun a job in a restartable state from a script or program. The logic of the option files that applies for the sdmsh utility is also used for sdms-rerun.

Call

The following commands are used to call sdms-rerun:

```
sdms-rerun [ OPTIONS ] < --host | -h > hostname
< --port | -p > portnumber < --jid | -j > jobid
```

OPTIONS:

- `< --user | -u > username`
- `< --pass | -w > password`
- `< --key | -k > jobkey`
- `< --silent | -s >`
- `< --verbose | -v >`
- `< --timeout | -t > minutes`
- `< --cycle | -c > minutes`
- `< --help | -h >`
- `< --suspend | -S >`
- `< --delay | -D > delay`
- `< --unit | -U > unit`
- `< --at | -A > at`

The individual options have the following meanings:

<table>
<thead>
<tr>
<th>Option</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>`&lt; --host</td>
<td>-h &gt; hostname`</td>
</tr>
<tr>
<td>`&lt; --port</td>
<td>-p &gt; portnumber`</td>
</tr>
<tr>
<td>`&lt; --user</td>
<td>-u &gt; username`</td>
</tr>
<tr>
<td>`&lt; --pass</td>
<td>-w &gt; password`</td>
</tr>
<tr>
<td>`&lt; --silent</td>
<td>-s &gt;`</td>
</tr>
</tbody>
</table>

Continued on the next page
<table>
<thead>
<tr>
<th>Option</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; --verbose</td>
<td>-v &gt;</td>
</tr>
<tr>
<td>&lt; --timeout</td>
<td>-t &gt; minutes</td>
</tr>
<tr>
<td>&lt; --cycle</td>
<td>-c &gt; minutes</td>
</tr>
<tr>
<td>&lt; --help</td>
<td>-h &gt;</td>
</tr>
<tr>
<td>&lt; --suspend</td>
<td>-S &gt;</td>
</tr>
<tr>
<td>&lt; --delay</td>
<td>-D &gt; delay</td>
</tr>
<tr>
<td>&lt; --unit</td>
<td>-U &gt; unit</td>
</tr>
<tr>
<td>&lt; --at</td>
<td>-A &gt; at</td>
</tr>
</tbody>
</table>
**sdms-set_state**

**Introduction**

The utility *sdms-set_state* offers a simple way of setting the state of a job in the scheduling system.

The logic of the option files that applies for the *sdmsh* utility is also used for *sdms-set_state*.

**Call**

The following commands are used to call *sdmsh-set_state*:

```
< < OPTIONS | < -- host | -h > hostname
< -- port | -p > portnumber < -- jid | -j > jobid
< -- state | -s > statename
```

**OPTIONS:**

```
< -- user | -u > username
< -- pass | -w > password
< -- key | -k > jobkey
< -- silent | -s >
< -- verbose | -v >
< -- timeout | -t > minutes
< -- cycle | -c > minutes
< -- help | -h >
< -- case | -C >
< -- no | force | -| no | f >
```

The individual options have the following meanings:

<table>
<thead>
<tr>
<th>Option</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; -- host</td>
<td>-h &gt; hostname</td>
</tr>
<tr>
<td>&lt; -- port</td>
<td>-p &gt; portnumber</td>
</tr>
<tr>
<td>&lt; -- user</td>
<td>-u &gt; username</td>
</tr>
<tr>
<td>&lt; -- pass</td>
<td>-w &gt; password</td>
</tr>
<tr>
<td>&lt; -- key</td>
<td>-k &gt; jobkey</td>
</tr>
</tbody>
</table>

*Continued on the next page*
<table>
<thead>
<tr>
<th>Option</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>`&lt; --silent</td>
<td>-s &gt;`</td>
</tr>
<tr>
<td>`&lt; --verbose</td>
<td>-v &gt;`</td>
</tr>
<tr>
<td>`&lt; --timeout</td>
<td>-t &gt; minutes`</td>
</tr>
<tr>
<td>`&lt; --cycle</td>
<td>-c &gt; minutes`</td>
</tr>
<tr>
<td>`&lt; --help</td>
<td>-h &gt;`</td>
</tr>
<tr>
<td>`&lt; --case</td>
<td>-C &gt;`</td>
</tr>
<tr>
<td>`&lt; --state</td>
<td>/-S &gt; state`</td>
</tr>
<tr>
<td>`&lt; --force</td>
<td>-f &gt;`</td>
</tr>
</tbody>
</table>
sdms-set_variable

Introduction

The utility sdms-set_variable offers a simple way of setting job parameters in the scheduling system. The logic of the option files that applies for the sdmsh utility is also used for sdms-set_variable.

Call

The following commands are used to call sdms-set_variable:

```
sdms-set_variable [ OPTIONS ] < --host | -h > hostname
< --port | -p > portnumber < --jid | -j > jobid
parametername value { parametername value}
```

OPTIONS:
```< --user | -u > username
< --pass | -w > password
< --key | -k > jobkey
< --silent | -s >
< --verbose | -v >
< --timeout | -t > minutes
< --cycle | -c > minutes
< --help | -h >
< --case | -C >```

The individual options have the following meanings:

<table>
<thead>
<tr>
<th>Option</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; --host</td>
<td>-h &gt; hostname</td>
</tr>
<tr>
<td>&lt; --port</td>
<td>-p &gt; portnumber</td>
</tr>
<tr>
<td>&lt; --user</td>
<td>-u &gt; username</td>
</tr>
<tr>
<td>&lt; --pass</td>
<td>-w &gt; password</td>
</tr>
<tr>
<td>&lt; --key</td>
<td>-k &gt; jobkey</td>
</tr>
<tr>
<td>&lt; --silent</td>
<td>-s &gt;</td>
</tr>
</tbody>
</table>

Continued on the next page
<table>
<thead>
<tr>
<th>Option</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; --verbose</td>
<td>Increases the number of messages that are returned</td>
</tr>
<tr>
<td>&lt; --timeout</td>
<td>Number of minutes for attempting to get a server connection</td>
</tr>
<tr>
<td>&lt; --cycle</td>
<td>Number of minutes for the delay between two attempts to set up a server connection</td>
</tr>
<tr>
<td>&lt; --help</td>
<td>Returns a condensed help text about calling the utility</td>
</tr>
<tr>
<td>&lt; --case</td>
<td>Names are case-sensitive</td>
</tr>
</tbody>
</table>
sdms-set_warning

Introduction

The utility sdms-set_warning is used to set the warning flag for a job. A text can be optionally specified. The warning flag can be set for a job by users who have been granted the Operate privilege. A job can set the warning flag for itself. The logic of the option files that applies for the sdmsh utility is also used for sdms-set_warning.

Call

The following commands are used to call sdms-set_warning:

```
sdms-set_warning [OPTIONS] < --host | -h > hostname
< --port | -p > portnumber < --jid | -j > jobid
```

OPTIONS:

- `< --user | -u > username`
- `< --pass | -w > password`
- `< --key | -k > jobkey`
- `< --silent | -s >`
- `< --verbose | -v >`
- `< --timeout | -t > minutes`
- `< --cycle | -c > minutes`
- `< --help | -h >`
- `< --warning | -m > warning`

The individual options have the following meanings:

<table>
<thead>
<tr>
<th>Option</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>`&lt; --host</td>
<td>-h &gt; hostname`</td>
</tr>
<tr>
<td>`&lt; --port</td>
<td>-p &gt; portnumber`</td>
</tr>
<tr>
<td>`&lt; --user</td>
<td>-u &gt; username`</td>
</tr>
<tr>
<td>`&lt; --pass</td>
<td>-w &gt; password`</td>
</tr>
<tr>
<td>`&lt; --key</td>
<td>-k &gt; jobkey`</td>
</tr>
<tr>
<td>`&lt; --silent</td>
<td>-s &gt;`</td>
</tr>
</tbody>
</table>

Continued on the next page
<table>
<thead>
<tr>
<th>Option</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>`&lt;--verbose</td>
<td>-v &gt;`</td>
</tr>
<tr>
<td>`&lt;--timeout</td>
<td>-t &gt; minutes`</td>
</tr>
<tr>
<td>`&lt;--cycle</td>
<td>-c &gt; minutes`</td>
</tr>
<tr>
<td>`&lt;--help</td>
<td>-h &gt;`</td>
</tr>
<tr>
<td>`&lt;--warning</td>
<td>-m &gt; warning`</td>
</tr>
</tbody>
</table>
sdms-submit

Introduction

Introduction  The utility sdms-submit is used to start jobs or batches. These can be started as a standalone workflow or also as a child of an existing job. In the latter case, if it is defined in the parent-child hierarchy an alias can be specified to identify the job or batch that is to be submitted.

The logic of the option files that applies for the sdmsh utility is also used for sdms-submit.

Call

Call  The following commands are used to call sdms-submit:

```
sdms-submit [ OPTIONS ] < --host | -h > hostname
< --port | -p > portnumber < --job | -J > jobname
```

OPTIONS:

```
< --user | -u > username
< --pass | -w > password
< --jid | -j > jobid
< --key | -k > jobkey
< --silent | -s >
< --verbose | -v >
< --timeout | -t > minutes
< --cycle | -c > minutes
< --help | -h >
< --tag | -T > tag
< --master | -M >
< --suspend | -S >
< --delay | -D > delay
< --unit | -U > unit
< --at | -A > at
```

The individual options have the following meanings:

<table>
<thead>
<tr>
<th>Option</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; --host</td>
<td>-h &gt; hostname</td>
</tr>
</tbody>
</table>

Continued on the next page
<table>
<thead>
<tr>
<th>Option</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>`&lt; --port</td>
<td>-p &gt; portnumber`</td>
</tr>
<tr>
<td>`&lt; --user</td>
<td>-u &gt; username`</td>
</tr>
<tr>
<td>`&lt; --pass</td>
<td>-w &gt; password`</td>
</tr>
<tr>
<td>`&lt; --key</td>
<td>-k &gt; jobkey`</td>
</tr>
<tr>
<td>`&lt; --silent</td>
<td>-s &gt;`</td>
</tr>
<tr>
<td>`&lt; --verbose</td>
<td>-v &gt;`</td>
</tr>
<tr>
<td>`&lt; --timeout</td>
<td>-t &gt; minutes`</td>
</tr>
<tr>
<td>`&lt; --cycle</td>
<td>-c &gt; minutes`</td>
</tr>
<tr>
<td>`&lt; --help</td>
<td>-h &gt;`</td>
</tr>
<tr>
<td>`&lt; --tag</td>
<td>-T &gt; tag`</td>
</tr>
<tr>
<td>`&lt; --master</td>
<td>-M &gt;`</td>
</tr>
<tr>
<td>`&lt; --suspnd</td>
<td>-S &gt;`</td>
</tr>
<tr>
<td>`&lt; --delay</td>
<td>-D &gt; delay`</td>
</tr>
<tr>
<td>`&lt; --unit</td>
<td>-U &gt; unit`</td>
</tr>
<tr>
<td>`&lt; --at</td>
<td>-A &gt; at`</td>
</tr>
</tbody>
</table>
Part II.

User Commands
3. alter commands
alter comment

Purpose

The purpose of the alter comment statement is to change the comment for the specified object.

Syntax

The syntax for the alter comment statement is:

```
alter [ existing ] comment on OBJECTURL
with CC_WITHITEM
```

OBJECTURL:

- distribution distributionname for pool identifier { . identifier } in serverpath
- environment environmentname
- exit state definition statename
- exit state mapping mappingname
- exit state profile profilename
- exit state translation transname
- event eventname
- resource identifier { . identifier } in folderpath
- folder folderpath
- footprint footprintname
- group groupname
- interval intervalname
- job definition folderpath
- job jobid
- named resource identifier { . identifier }
- parameter parametername of PARAM_LOC
- resource state definition statename
- resource state mapping mappingname
- resource state profile profilename
- scheduled event schedulepath . eventname
- schedule schedulepath
- resource identifier { . identifier } in serverpath
- < scope serverpath | observer serverpath >
- trigger triggername on TRIGGEROBJECT [ < noinverse | inverse > ]
- user username
Description

The `alter comment` command is used to change the condensed description or URL of the description of the object in question. Of course, the type of information can be changed as well. The comment is versioned. This means that comments are not overwritten. When the commented object is displayed, the displayed comment is the one that matches the version of the displayed object.

The optional `existing` keyword is used to prevent error messages from being displayed and the current operation from being terminated. This is particularly useful in conjunction with `multicommands`.

Output

This statement returns a confirmation of a successful operation.
alter environment

Purpose

The purpose of the alter environment statement is to alter the properties of the specified environment.

Syntax

The syntax for the alter environment statement is

\[
\begin{align*}
\text{alter} \ [\text{existing}] \ &\text{environment} \ \text{environmentname} \\
\text{with} \ &\text{ENV}_\text{WITH}_{-} \text{ITEM} \\
\text{alter} \ [\text{existing}] \ &\text{environment} \ \text{environmentname} \\
\text{add} \ &\text{( ENV}_\text{RESOURCE} \{, \text{ENV}_\text{RESOURCE}\} \} \\
\text{alter} \ [\text{existing}] \ &\text{environment} \ \text{environmentname} \\
\text{delete} \ &\text{( RESOURCEPATH} \{, \text{RESOURCEPATH}\} \}) \\
\end{align*}
\]

ENV\_WITH\_ITEM:

\[
\begin{align*}
\text{resource} \ &= \ \text{none} \\
| \text{resource} \ &= \ \text{( ENV}_\text{RESOURCE} \{, \text{ENV}_\text{RESOURCE}\} \}) \\
\end{align*}
\]

ENV\_RESOURCE:

\[
\begin{align*}
\text{identifier} \{, \text{identifier}\} \ [\ < \text{condition} = \text{string} \ | \text{condition} = \text{none} > ] \\
\end{align*}
\]

RESOURCEPATH:

\[
\begin{align*}
\text{identifier} \{, \text{identifier}\} \\
\end{align*}
\]

Description

The alter environment statement is used to change the resource requests that are defined in this environment. Running jobs are not affected.

The “\textbf{with resource =}” form of the statement replaces the existing group of resource requests. The other types either add the specified requests or deletes them. It is considered an error to delete a request that is not part of the environment or to add a request for an already required resource.

Only administrators are authorised to perform this action.

Output

This statement returns a confirmation of a successful operation.
alter event

Purpose

The purpose of the alter event statement is to change properties of the specified event.

Syntax

The syntax for the alter event statement is

```
alter [ existing ] event eventname
with EVENT_WITHITEM {, EVENT_WITHITEM}

EVENT_WITHITEM:
  action =
    submit folderpath [ with parameter = ( PARAM {, PARAM} ) ]
  | group = groupname

PARAM:
  parametername = < string | number >
```

Description

The alter event statement is used to change the properties of an event. A parameter for a job submit can be specified using the with parameter clause. For a detailed description of these options, refer to the create event statement on page 111.

Output

This statement returns a confirmation of a successful operation.
**alter exit state mapping**

**Purpose**

The purpose of the *alter exist state mapping* statement is to change properties of the specified mapping.

**Syntax**

The syntax for the *alter exit state mapping* statement is

```plaintext
alter [ existing ] exit state mapping mappingname
with map = ( statename { , signed_integer , statename } )
```

**Description**

The *alter exit state mapping* statement defines the mapping of the Exit Codes for logical Exit States. The simplest form of this statement only specifies one Exit State. This means that the job acquires this Exit State when it finishes regardless of its Exit Code. More complex definitions specify more than one Exit State and at least one delimitation.

A statement like

```plaintext
alter exit state mapping example1
with map = ( failure, 0, success, 1, warning, 4, failure);
```

defines the following mapping:

<table>
<thead>
<tr>
<th>Exit code range from</th>
<th>Exit code range until</th>
<th>Resulting exit state</th>
</tr>
</thead>
<tbody>
<tr>
<td>−∞</td>
<td>-1</td>
<td>failure</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>success</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>warning</td>
</tr>
<tr>
<td>4</td>
<td>∞</td>
<td>failure</td>
</tr>
</tbody>
</table>

**Output**

This statement returns a confirmation of a successful operation.
alter exit state profile

Purpose

The purpose of the alter exit state profile statement is to change properties of the specified profile.

Syntax

The syntax for the alter exit state profile statement is

```
alter [ existing ] exit state profile  profilename
with WITHITEM {, WITHITEM}
```

WITHITEM:

```
default mapping = < none | mappingname >
| force
| state = ( ESP_STATE {, ESP_STATE} )
```

ESP_STATE:

```
statename < final | restartable | pending > [ OPTION { OPTION} ]
```

OPTION:

```
batch default
| broken
| dependency default
| disable
| unreachable
```

Description

The alter exit state profile statement is used to add Exit States to the profile or delete them, as well as to define the default Exit State Mapping. For a detailed description of these options, refer to the create exit state profile statement on page 114.

force  The force option labels the Exit State Profiles as being invalid, which only means that the integrity still has to be verified. The label is removed after a successful verification. The verification is carried out by submitting a job definition that uses the Exit State Profiles. The purpose of the force flag is to be capable of changing several Exit State Profiles (and perhaps some other objects) without the need for a consistent state after each change.
Output

Output This statement returns a confirmation of a successful operation.
**alter exit state translation**

**Purpose**

The purpose of the *alter exit state translation* statement is to change properties of the specified exit state translation.

**Syntax**

The syntax for the *alter exit state translation* statement is

```
alter [existing] exit state translation transname
with translation = ( statename to statename {, statename to statename} )
```

**Description**

The *alter exit state translation* statement changes a previously defined Exit State Translation. Running jobs are not affected. If the optional *existing* keyword has been specified, no error is created if the specified Exit State Translation could not be found.

**Output**

This statement returns a confirmation of a successful operation.
**alter folder**

**Purpose**

*Purpose*  The purpose of the *alter folder* statement is to alter the properties of a folder.

**Syntax**

*Syntax*  The syntax for the *alter folder* statement is

```plaintext
alter [ existing ] folder folderpath
with WITHITEM {, WITHITEM}
```

**WITHITEM:**

- `environment = < none | environmentname >`
- `group = groupname [ cascade ]`
- `inherit grant = none`
- `inherit grant = ( PRIVILEGE {, PRIVILEGE} )`
- `parameter = none`
- `parameter = ( parametername = string {, parametername = string} )`

**PRIVILEGE:**

- `approve`
- `cancel`
- `clear warning`
- `clone`
- `create content`
- `drop`
- `edit [ parameter ]`
- `enable`
- `execute`
- `ignore resource`
- `ignore dependency`
- `kill`
- `monitor`
- `operate`
- `priority`
- `rerun`
- `resource`
- `set job status`
- `set state`
- `submit`
- `suspend`
Description

The *alter folder* statement changes the properties of a folder. For a detailed description of these options, refer to the create folder statement on page 118.

If the optional *existing* keyword has been specified, no error is created if the specified folder does not exist.

Although the folder SYSTEM cannot be created, dropped or renamed, it can be altered to some extend. It is not possible to change the owning group, but it is possible to specify an environment or to create parameters.

Output

This statement returns a confirmation of a successful operation.
alter footprint

Purpose

The purpose of the alter footprint statement is to change the properties of the specified footprint.

Syntax

The syntax for the alter footprint statement is

\[
\text{alter [existing] footprint footprintname with resource = ( REQUIREMENT \{, REQUIREMENT\})}
\]

\[
\text{alter [existing] footprint footprintname add resource = ( REQUIREMENT \{, REQUIREMENT\})}
\]

\[
\text{alter [existing] footprint footprintname delete resource = ( RESOURCEPATH \{, RESOURCEPATH\})}
\]

REQUIREMENT:
ITEM { ITEM}

RESOURCEPATH:
identifier {, identifier}

ITEM:
\[
\text{amount = integer}
\]
| < nokeep | keep | keep final |
| identifier {, identifier}

Description

The alter footprint command changes the list of resource requests. There are three kinds of this statement.

- The first one determines all the resource requests.
- The second one adds resource requests to the request list.
- The third kind removes requests from the list.

For a detailed description of these options, refer to the create footprint statement on page 120.
Output

This statement returns a confirmation of a successful operation.
alter group

Purpose

Purpose The purpose of the alter group statement is to alter the user to group assignments.

Syntax

Syntax The syntax for the alter group statement is

```
alter [ existing ] group groupname
with WITHITEM

alter [ existing ] group groupname
ADD_DELITEM {, ADD_DELITEM}

WITHITEM:
  user = none
  | user = ( username {, username} )

ADD_DELITEM:
  < add | delete > user = ( username {, username} )
```

Description

Description The alter group command is used to define which users belong to the group. There are two kinds of this statement:

- The first one defines the list of users who belong to the group.
- The second one adds users to the group or deletes them.

In all cases, deleting users from their default group is considered to be an error. It is not possible to delete users from the PUBLIC group. If a user does not belong to a group, any attempt made to delete the user from this group is ignored. If the existing keyword has been specified, it is not considered to be an error if the group does not exist.

Output

Output This statement returns a confirmation of a successful operation.
alter interval

Purpose

The purpose of the alter interval statement is to change properties of the specified interval.

Syntax

The syntax for the alter interval statement is

\[
\text{alter} \ [\text{existing}] \ \text{interval} \ \text{intervalname} \\
\text{with} \ \text{WITHITEM} \{, \ \text{WITHITEM}\}
\]

WITHITEM:

\[
\begin{align*}
\text{base} &= \text{< none} | \text{period}> \\
\text{dispatch} &= \text{none} \\
\text{dispatch} &= (\text{IVAL_DISPATCHITEM} \{, \text{IVAL_DISPATCHITEM}\}) \\
\text{duration} &= \text{< none} | \text{period}> \\
\text{embedded} &= \text{< none} | \text{CINTERVALNAME}> \\
\text{endtime} &= \text{< none} | \text{datetime}> \\
\text{filter} &= \text{none} \\
\text{filter} &= (\text{CINTERVALNAME} \{, \text{CINTERVALNAME}\}) \\
\text{< noinverse} | \text{inverse}> \\
\text{selection} &= \text{none} \\
\text{selection} &= (\text{IVAL_SELITEM} \{, \text{IVAL_SELITEM}\}) \\
\text{starttime} &= \text{< none} | \text{datetime}> \\
\text{synctime} &= \text{datetime} \\
\text{group} &= \text{groupname}
\end{align*}
\]

IVAL_DISPATCHITEM:

dispatchname < \text{active} | \text{inactive} > \text{IVAL_DISPATCHDEF}

CINTERVALNAME:

(\text{intervalname}

\text{with} \ \text{WITHITEM} \{, \ \text{WITHITEM}\})

| \text{intervalname}

IVAL_SELITEM:

\text{< signed_integer} | \text{datetime} | \text{datetime - datetime} >
IVAL_DISPATCHDEF:
   none CINTERVALNAME < enable | disable >
   | CINTERVALNAME CINTERVALNAME < enable | disable >
   | CINTERVALNAME < enable | disable >

Description

Description  The alter interval command is used to change an interval definition. For a detailed description of these options, refer to the create interval statement on page 123. If the existing keyword has been specified, it is not considered to be an error if the interval does not exist.

Output

Output  This statement returns a confirmation of a successful operation.
**alter job**

**Purpose**

The purpose of the *alter job* statement is to change properties of the specified job. This statement is used by job administrators, job servers, and by the job itself.

**Syntax**

The syntax for the *alter job* statement is

```
alter job jobid
    with WITHITEM {, WITHITEM}
```

```
alter job
    with WITHITEM {, WITHITEM}
```

**WITHITEM:**

```
< disable | enable >
| < suspend | suspend restrict | suspend local | suspend local restrict >
cancel
clear warning
clone [ < resume | suspend > ]
comment = string
error text = string
exec pid = pid
exit code = signed_integer
exit state = statename [ force ]
ext pid = pid
ignore resource = ( id [, id] )
ignore dependency = ( jobid [ recursive ] {, jobid [ recursive ]} )
kill [ recursive ]
nicevalue = signed_integer
priority = integer
renice = signed_integer
rerun | recursive |
resume
| < noresume | resume in period | resume at datetime >
run = integer
state = JOBSTATE
timestamp = string
warning = string
```
Description

The *alter job* command is used for several purposes. Firstly, jobservers use this command to document the progress of a job. All the state transitions a job undergoes during the time when the job is the responsibility of a jobserver are performed using the *alter job* command.

Secondly, some changes such as ignoring dependencies or resources, as well as changing the priority of a job, are carried out manually by an administrator.

The Exit State of a job in a Pending State can be set by the job itself or by a process that knows the job ID and key of the job that is to be changed.

**cancel**  The cancel option is used to cancel the addressed job and all non-Final Children. A job can only be cancelled if neither the job itself nor one of its children is active. Cancelling a running job will set the job in a cancelling state. The effective cancel is postponed until the job is finished.

If a Scheduling Entity is dependent upon the cancelled job, it can become unreachable. In this case the dependent job does not acquire the Unreachable Exit State defined in the Exit State Profiles, but is set as having the Job State “Unreachable”. It is the operator’s task to restore this job back to the job state “Dependency Wait” by ignoring dependencies or even to cancel it.

Cancelled jobs are considered to be just like Final Jobs without a Final Exit. This means that the parents of a cancelled job become final without taking into consideration the Exit State of the cancelled job. In this case the dependent jobs of the parents continue running normally.

The cancel option can only be used by users.

**comment**  The comment option is used to document an action or to add a comment to the job. Comments can have a maximum length of 1024 characters. Any number of comments can be saved for a job.

Some comments are saved automatically. For example, if a job attains a Restartable State, a log is written to document this fact.
**error text**  The error text option is used to write error information about a job. This can be done by the responsible jobserver or a user. The server can write this text itself as well. This option is normally used if the jobserver cannot start the corresponding process. Possible cases are where it is not possible to switch to the defined working directory, if the executable program cannot be found, or when opening the error log file triggers an error.

**exec pid**  The exec pid option is used exclusively by the jobserver to set the process ID of the control process within the server.

**exit code**  The exit code option is used by the jobserver to tell the repository server with which Exit Code the process has finished. The repository server now calculates the matching Exit State from the Exit State Mapping that was used.

**exit state**  The exit state option is used by jobs in a pending state to set their state to another value. This is usually a Restartable or Final State. Alternatively, this option can be used by administrators to set the state of a non-final job. If the Force Flag is not being used, the only states that can be set are those which are theoretically attainable by applying the Exit State Mapping to any Exit Code. The set state must exist in the Exit State Profile.

**ext pid**  The ext pid option is used exclusively by the jobserver to set the process ID of the started user process.

**ignore resource**  The ignore resource option is used to revoke individual Resource Requests. The ignored resource is then no longer requested. If the parameters of a resource are being referenced, that resource cannot be ignored. If invalid IDs have been specified, it is skipped. All other specified resources are ignored. Invalid IDs in this context are the IDs of resources that are not requested by the job. The ignoring of resources is logged.

**ignore dependency**  The ignore dependency option is used to ignore defined dependencies. If the `recursive` flag is used, not only do the job or batch ignore the dependencies, but its children do so as well.
**kill**  The kill option is used to submit the defined Kill Job. If no Kill Job has been defined, it is not possible to forcibly terminate the job from within BICsuite. The job obviously has to be active, that means it must be **running**, **killed** or **broken_active**. The last two states are not regular cases. When a Kill Job has been submitted, the Job State is **to_kill**. After the Kill Job has terminated, the Job State of the killed job is set to **killed** unless it has been completed, in which case it is **finished** or **final**. This means that the job with the Job State **killed** is always still running and that at least one attempt has been made to terminate it.

**nicevalue**  The nicevalue option is used to change the priority or the nicevalue of a job or batch and all of its children. If a child has several parents, any changes you make can, but do not necessarily have to, affect the priority of the child in the nicevalue of one of the parents. Where there are several parents, the maximum nicevalue is searched for.
This means that if Job C has three Parents P1, P2 and P3, whereby P1 sets a nice value of 0, P2 sets a nicevalue of 10 and P3 a nicevalue of -10, the effective nicevalue is -10. (The lower the nicevalue the better). If the nicevalue for P2 is changed to -5, nothing happens because the -10 of P3 is better than -5. If the nicevalue of P3 falls to 0, the new effective nicevalue for Job C is -5.
The nicevalues can have values between -100 and 100. Values that exceed this range are tacitly adjusted.

**priority**  The priority option is used to change the (static) priority of a job. Because batches and milestones are not executed, priorities are irrelevant to them. Changing the priority only affects the changed job. Valid values lie between 0 and 100. In this case, 100 corresponds to the lowest priority and 0 is the highest priority. When calculating the dynamic priority of a job, the scheduler begins with the static priority and adjusts it according to how long the job has already been waiting. If more than one job has the same dynamic priority, the job with the lowest job ID is scheduled first.

**renice**  The renice option is similar to the nicevalue option with the difference that the renice option functions relatively while the nicevalue option functions absolutely. If some batches have a nicevalue of 10, a renice of -5 causes the nicevalue to rise to 5. (It rises because the lower the number, the higher the priority).

**rerun**  The rerun option is used to restart a job in a Restartable State. If you attempt to restart a job that is not restartable, an error message is displayed. A job is restartable if it is in a Restartable State or it has the Job State **error** or **broken_finished**.
If the **recursive** flag has been specified, the job itself and all its direct and indirect children that are in a Restartable State are restarted. If the job itself is final, this is
not considered to be an error. It is therefore possible to recursively restart batches.

**resume**  The resume option is used to reactivate a suspended job or batch. There are two ways to do this. The suspended job or batch can either be reacti-vated immediately or a delay can be set. A delay can be achieved by specifying either the number of time units for the delay the time when the job or batch is to be activated. For details about specifying a time, refer to the overview on page 20. The resume option can be used together with the suspend option. Here, the job is suspended and then resumed again after (or at) a specified time.

**run**  The run option is used by the jobserver to ensure that the modified job matches the current version. Theoretically, the computer could crash after a job has been started by a jobserver. To complete the work, the job is manually restarted from another jobserver. After the first system has been booted, the jobserver can attempt to change the job state to broken\_finished without knowing anything about what happened after the crash. Using the run option then prevents the wrong state from being set.

**state**  The state option is mainly used by jobservers, but it can also be used by administrators. It is not recommended to do so unless you know exactly what you are doing. The usual procedure is that the jobserver sets the state of a job from starting to started, from started to running, and from running to finished. In the event of a crash or any other problems, it is possible for the jobserver to set the job state to broken\_active or broken\_finished. This means that the Exit Code of the process is not available and the Exit State has to be set manually.

**suspend**  The suspend option is used to suspend a batch or job. It always functions recursively. If a parent is suspended, its children are all suspended as well. The resume option is used to reverse the situation. The effect of the restrict option is that cwa resume can be done by members of the group ADMIN only.

**timestamp**  The timestamp option is used by the jobserver to set the times-tamps of the state transition in keeping with the local time from the perspective of jobserver.

**Output**  This statement returns a confirmation of a successful operation.
**alter job definition**

**Purpose**

The purpose of the *alter job definition* statement is to change properties of the specified job definition.

**Syntax**

The syntax for the *alter job definition* statement is

```plaintext
alter [ existing | job definition ] folderpath
    with WITHITEM {, WITHITEM}

alter [ existing | job definition ] folderpath
    AJD_ADD_DEL_ITEM {, AJD_ADD_DEL_ITEM}
```

**WITHITEM:**

- approval = none
- approval = ( OPERATE_APPROVAL {, OPERATE_APPROVAL} )
- children = none
- children = ( JOB_CHILDDEF {, JOB_CHILDDEF} )
- dependency mode = < all | any >
- environment = environmentname
- errlog = < none | filespec [ < notrunc | trunc > ] >
- footprint = < none | footprintname >
- inherit grant = none
- inherit grant = ( PRIVILEGE {, PRIVILEGE} )
- kill program = < none | string >
- logfile = < none | filespec [ < notrunc | trunc > ] >
- mapping = < none | mappingname >
- < nomaster | master >
- nicevalue = < none | signed_integer >
- parameter = none
- parameter = ( JOB_PARAMETER {, JOB_PARAMETER} )
- priority = < none | signed_integer >
- profile = profilename
- required = none
- required = ( JOB_REQUIRED {, JOB_REQUIRED} )
- rerun program = < none | string >
- resource = none
- resource = ( REQUIREMENT {, REQUIREMENT} )
- < noresume | resume in period | resume at datetime >
runtime = integer
runtime final = integer
run program = < none | string >
< nosuspend | suspend >
timeout = none
timeout = period state statename
type = < job | milestone | batch >
group = groupname
workdir = < none | string >

AJD_ADD_DEL_ITEM:
  add [ or alter ] children = ( JOB_CHILDDEF {, JOB_CHILDDEF} )
  add [ or alter ] parameter = ( JOB_PARAMETER {, JOB_PARAMETER} )
  add [ or alter ] required = ( JOB_REQUIRED {, JOB_REQUIRED} )
  add [ or alter ] resource = ( REQUIREMENT {, REQUIREMENT} )
  alter [ existing ] children = ( JOB_CHILDDEF {, JOB_CHILDDEF} )
  alter [ existing ] parameter = ( JOB_PARAMETER {, JOB_PARAMETER} )
  alter [ existing ] required = ( JOB_REQUIRED {, JOB_REQUIRED} )
  alter [ existing ] resource = ( REQUIREMENT {, REQUIREMENT} )
  delete [ existing ] children = ( folderpath {, folderpath} )
  delete [ existing ] parameter = ( parmlist )
  delete [ existing ] required = ( folderpath {, folderpath} )
  delete [ existing ] resource = ( RESOURCEPATH {, RESOURCEPATH} )

OPERATE_APPROVAL:
OPERATE_PRIV APPROVAL_MODE [ leading ]

JOB_CHILDDEF:
JCD_ITEM { JCD_ITEM }

PRIVILEGE:
  approve
  cancel
  clear warning
  clone
  create content
  drop
  edit [ parameter ]
  enable
  execute
ignore resource
general dependency
kill
monitor
operate
priority
rerun
resource
set job status
set state
submit
suspend
use
view

JOB_PARAMETER:

\[ \text{parametername} \ (id) < [\text{JP}_\text{WITHITEM}] \ [\text{default} = \text{string}] \ | \ [\text{JP}_\text{NONDEFWITH}] > ] \\
 [\text{local}] [ < \text{export} = \text{parametername} \ | \text{export} = \text{none} > ] \\

JOB_REQUIRED:

\[ \text{JRQ}\_\text{ITEM} \ \{ \text{JRQ}\_\text{ITEM} \} \]

REQUIREMENT:

\[ \text{JRD}\_\text{ITEM} \ \{ \text{JRD}\_\text{ITEM} \} \]

RESOURCEPATH:

\[ \text{identifier} \ (. \ \text{identifier}) \]

OPERATE_PRIV:

\[ \text{cancel} \\\n\text{clear warning} \\\n\text{clone} \\\n\text{edit parameter} \\\n\text{enable} \\\n\text{ignore resource} \\\n\text{ignore dependency} \\\n\text{kill} \\\n\text{priority} \\\n\text{rerun} \\\n\text{set job status} \]

76
| set state  
suspend |

APPROVAL_MODE:
| approve   
default   
master    
no        
parent    
review    |

JCD_ITEM:
| alias = < none | aliasname > |
| condition = < none | string > |
| < enable | disable > |
| folderpath . jobname |
| ignore dependency = none |
| ignore dependency = ( dependencyname {, dependencyname} ) |
| interval = < none | intervalname > |
| < childsuspend | suspend | nosuspend > |
| merge mode = < nomerge | merge local | merge global | failure > |
| mode = < and | or > |
| nicevalue = < none | signed_integer > |
| priority = < none | signed_integer > |
| < noresume | resume in period | resume at datetime > |
| < static | dynamic > |
| translation = < none | transname > |

JP_WITHITEM:
| import | unresolved |
| parameter |
| reference child folderpath ( parametername ) |
| reference folderpath ( parametername ) |
| reference resource identifier [. identifier] ( parametername ) |
| result |

JP_NONDEFWITH:
| constant = string |
| JP_AGGFUNCTION ( parametername ) |
JRQ_ITEM:

- condition = < none | string >
- dependency dependencyname
- expired = < none | signed_period_rj >
- folderpath, jobname
- mode = < all final | job final >
- resolve = < internal | external | both >
- select condition = < none | string >
- state = none
- state = ( JRQ_REQ_STATE {}, JRQ_REQ_STATE {} )
- state = all reachable
- state = default
- state = unreachable
- unresolved = JRQ_UNRESOLVED

JRD_ITEM:

- amount = integer
- expired = < none | signed_period >
- < nokeep | keep | keep final >
- condition = < string | none >
- lockmode = LOCKMODE
- nosticky
- identifier {}, identifier
- state = none
- state = ( statename {}, statename {} )
- state mapping = < none | rsmname >
- sticky
  [ ( < identifier | folderpath | identifier, folderpath | folderpath, identifier > ) ]

JP_AGGFUNCTION:

- avg
- count
- max
- min
- sum

JRQ_REQ_STATE:

- statename | < condition = string | condition = none > |

JRQ_UNRESOLVED:
Description

The alter job definition command has two different variants.

- The first is similar to the create job definition statement and is used to redefine the job definition. All the affected options are overwritten. All the unaddressed options remain as they are.

- The second variant is used to add, edit or delete entries from the lists of children, resource requests, dependencies or parameters.

The options are described in detail in the create job definition command on page 130. This also applies for the options in the child, resource request, dependency and parameter definitions.

If the existing keyword is being used, an error is not triggered if the addressed job definition does not exist. The same applies if the existing keyword is being used while the list entries are being deleted or edited.

Output

This statement returns a confirmation of a successful operation.
alter named resource

Purpose

The purpose of the alter named resource statement is to change its properties.

Syntax

The syntax for the alter named resource statement is

```plaintext
alter [ existing ] named resource identifier { identifier } with WITHITEM {, WITHITEM}
```

WITHITEM:

```plaintext
group = groupname [ cascade ]
| inherit grant = none
| inherit grant = ( PRIVILEGE {, PRIVILEGE} )
| parameter = none
| parameter = ( PARAMETER {, PARAMETER} )
| state profile = <> none | rspname >
```

PRIVILEGE:

```plaintext
approve
| cancel
| clear warning
| clone
| create content
| drop
| edit [ parameter ]
| enable
| execute
| ignore resource
| ignore dependency
| kill
| monitor
| operate
| priority
| rerun
| resource
| set job status
| set state
| submit
```
The `alter named resource` statement is used to change the properties of the Named Resource. For a detailed description of the options, refer to the description of the `create named resource` statement on page 150. If the `existing` keyword has been specified, attempting to modify a non-existent Named Resource will not trigger an error.

**Output**

This statement returns a confirmation of a successful operation.
**alter resource**

**Purpose**

*Purpose*  The purpose of the *alter resource* statement is to change properties of resources.

**Syntax**

*Syntax*  The syntax for the *alter resource* statement is

```
alter [ existing ] RESOURCE_URL [ with WITHITEM { , WITHITEM } ]
```

**RESOURCE URL:**

- resource identifier (. identifier) in folderpath
- resource identifier (. identifier) in serverpath

**WITHITEM:**

- amount = < infinite | integer >
- < online | offline >
- parameter = none
- parameter = ( PARAMETER { , PARAMETER } )
- requestable amount = < infinite | integer >
- state = statename
- touch [ = datetime ]
- group = groupname

**PARAMETER:**

- parametername = < string | default >

**Description**

*Description*  The *alter resource* statement is used to change the properties of resources. For a detailed description of the options, refer to the description of the *create resource* statement on page 154.

If the *existing* keyword has been specified, attempting to modify a non-existent resource will *not* trigger an error.

**Output**

*Output*  This statement returns a confirmation of a successful operation.
alter resource state mapping

Purpose
The purpose of the alter resource state mapping statement is to change properties of the mapping.

Syntax
The syntax for the alter resource state mapping statement is

```
alter [existing] resource state mapping mappingname
with map = ( WITHITEM {, WITHITEM } )

WITHITEM:
    statename maps < statename | any > to statename
```

Description
The alter resource state mapping statement is used to change the properties of the Resource State Mapping. For a detailed description of the options, refer to the description of the create resource state mapping statement on page 158.

If the existing keyword has been specified, attempting to modify a non-existent Resource State Mapping will not trigger an error.

Output
This statement returns a confirmation of a successful operation.
**alter resource state profile**

**Purpose**

*Purpose* The purpose of the *alter resource state profile* statement is to change properties of the specified resource state profile.

**Syntax**

*Syntax* The syntax for the *alter resource state profile* statement is

```
alter [ existing ] resource state profile profilename
with WITHITEM {, WITHITEM}
```

WITHITEM:

- `initial state = statename`
- `state = ( statename {, statename} )`

**Description**

*Description* The *alter resource state profile* statement is used to change the properties of the Resource State Profile. For a detailed description of the options, refer to the description of the *resource state profile* statement on page 159.

If the *existing* keyword has been specified, attempting to modify a non-existent Resource State Profile does not return an error.

**Output**

*Output* This statement returns a confirmation of a successful operation.
**alter schedule**

**Purpose**

The purpose of the `alter schedule` statement is to change properties of the specified schedule.

**Syntax**

The syntax for the `alter schedule` statement is

```sql
alter [existing] schedule schedulepath
with WITHITEM {, WITHITEM}
```

WITHITEM:

- `< active | inactive >`
- `inherit grant = none`
- `inherit grant = ( PRIVILEGE {, PRIVILEGE} )`
- `interval = < none | intervalname >`
- `time zone = string`
- `group = groupname`

PRIVILEGE:

- `approve`
- `cancel`
- `clear warning`
- `clone`
- `create content`
- `drop`
- `edit [ parameter ]`
- `enable`
- `execute`
- `ignore resource`
- `ignore dependency`
- `kill`
- `monitor`
- `operate`
- `priority`
- `rerun`
- `resource`
- `set job status`
- `set state`
- `submit`

85
Description

The *alter schedule* statement is used to change the properties of a schedule. For a detailed description of the options for the *create schedule* statement, refer to page 160.

If the *existing* keyword has been specified, attempting to modify a non-existent schedule will *not* trigger an error.

Output

This statement returns a confirmation of a successful operation.
alter scheduled event

Purpose

The purpose of the *alter scheduled event* statement is to change properties of the specified scheduled event.

Syntax

The syntax for the *alter scheduled event* statement is

```
alter [ existing ] scheduled event schedulepath . eventname
with WITHITEM {, WITHITEM}
```

WITHITEM:

```
< active | inactive >
| backlog handling = < last | all | none >
| calendar = < active | inactive >
| horizon = < none | integer >
| suspend limit = < default | period >
| group = groupname
```

Description

The *alter scheduled event* statement is used to change the properties of a specified Scheduled Event. For a detailed description of the options for the *create scheduled event* statement, refer to page 162. If the *existing* keyword has been specified, attempting to modify a non-existent Scheduled Event does *not* return an error.

Output

This statement returns a confirmation of a successful operation.
alter scope

Purpose

Purpose  The purpose of the alter scope statement is to change the properties of the specified scope.

Syntax

Syntax  The syntax for the alter scope statement is

alter [ existing ] < scope serverpath | jobserver serverpath >
with JS_WITHITEM {, JS_WITHITEM}

alter [ existing ] jobserver
with < fatal | nonfatal > error text = string

alter [ existing ] jobserver
with dynamic PARAMETERS

JS_WITHITEM:

    config = none
| config = ( CONFIGITEM {, CONFIGITEM} )
| < enable | disable >
| error text = < none | string >
| group = groupname [ cascade ]
| inherit grant = none
| inherit grant = ( PRIVILEGE {, PRIVILEGE} )
| node = nodename
| parameter = none
| parameter = ( PARAMETERITEM {, PARAMETERITEM} )
| password = string
| rawpassword = string [ salt = string ]

PARAMETERS:

    parameter = none
| parameter = ( PARAMETERSPEC {, PARAMETERSPEC} )
CONFIGITEM:

\[
\text{parametername} = \text{none} \\
\text{parametername} = ( \text{PARAMETERSPEC}, \text{PARAMETERSPEC} ) \\
\text{parametername} = < \text{string} \mid \text{number} >
\]

PRIVILEGE:

\[
\text{approve} \\
\text{cancel} \\
\text{clear warning} \\
\text{clone} \\
\text{create content} \\
\text{drop} \\
\text{edit [parameter]} \\
\text{enable} \\
\text{execute} \\
\text{ignore resource} \\
\text{ignore dependency} \\
\text{kill} \\
\text{monitor} \\
\text{operate} \\
\text{priority} \\
\text{rerun} \\
\text{resource} \\
\text{set job status} \\
\text{set state} \\
\text{submit} \\
\text{suspend} \\
\text{use} \\
\text{view}
\]

PARAMETERITEM:

\[
\text{parametername} = \text{dynamic} \\
\text{parametername} = < \text{string} \mid \text{number} >
\]

PARAMETERSPEC:

\[
\text{parametername} = < \text{string} \mid \text{number} >
\]

Description

The \textit{alter scope} command is a user command. This command is used to modify the configuration or other properties of a scope.
Output

This statement returns a confirmation of a successful operation.
**alter server**

**Purpose**

The purpose of the *alter server* statement is to enable or disable user connections, or to define the trace level.

**Syntax**

The syntax for the *alter server* statement is

\[
\text{alter server with } \langle \text{enable} \mid \text{disable} \rangle \text{ connect}
\]

\[
\text{alter server with schedule}
\]

\[
\text{alter server with trace level } = \text{integer}
\]

\[
\text{alter server with } \langle \text{suspend} \mid \text{resume} \rangle \text{ integer}
\]

**Description**

The *alter server* command can be used to activate and deactivate the ability to connect to the server. If this possibility has been deactivated, only the "System" user can connect to the server.

The *alter server* command is also used to define the logged server message types. The following information types are defined:

<table>
<thead>
<tr>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fatal</td>
<td>A fatal error has occurred. The server is being run down.</td>
</tr>
<tr>
<td>Error</td>
<td>An error has occurred.</td>
</tr>
<tr>
<td>Info</td>
<td>An important informational message that was not written due to an error.</td>
</tr>
<tr>
<td>Warning</td>
<td>A warning.</td>
</tr>
<tr>
<td>Message</td>
<td>An informative message.</td>
</tr>
<tr>
<td>Debug</td>
<td>Messages that can be used for troubleshooting.</td>
</tr>
</tbody>
</table>

Fatal messages, error messages and info messages are always written to the server log file. Warnings are written at Trace Level 1 or higher. Normal messages are written at Trace Level 2 or higher. Debug messages provide a large volume of output data and are returned at Trace Level 3.

The schedule option is used to make a scheduling thread execute a full reschedule. The suspend/resume option can be used to suspend or resume internal threads.
Output

Output This statement returns a confirmation of a successful operation.
**alter session**

**Purpose**

The purpose of the *alter session* statement is to specify the used protocol, the session timeout value or the trace level for the specified session.

**Syntax**

The syntax for the *alter session* statement is

```
alter session [ sid ]
with WITHITEM {, WITHITEM}
```

```
alter session set user = username [ with WITHITEM {, WITHITEM} ]
```

```
alter session set user = username for username [ with WITHITEM {, WITHITEM} ]
```

```
alter session set user is default
```

WITHITEM:

```
command = ( sdms-command { ; sdms-command } )
| method = string
| protocol = PROTOCOL
| session = string
| timeout = integer
| token = string
| < trace | notrace >
| trace level = integer
```

PROTOCOL:

```
json [ ZERO TERMINATED ]
| line
| perl [ ZERO TERMINATED ]
| python [ ZERO TERMINATED ]
| serial
| xml
```

**Description**

The *alter session* command can be used to enable and disable the trace. If the trace is enabled, all the issued commands are logged in the log file. A communication
protocol can also be selected. An overview of the currently defined protocols is shown in the table below.

<table>
<thead>
<tr>
<th>Protokoll</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Line</td>
<td>Plain ASCII output</td>
</tr>
<tr>
<td>Perl</td>
<td>The output is offered as a Perl structure that can be easily evaluated by the Perl script using eval.</td>
</tr>
<tr>
<td>Python</td>
<td>Like Perl, but this is a Python structure.</td>
</tr>
<tr>
<td>Serial</td>
<td>Serialized Java objects.</td>
</tr>
<tr>
<td>Xml</td>
<td>Outputs an xml structure.</td>
</tr>
</tbody>
</table>

The timeout parameter for the session can be set as a last resort. A timeout of 0 means that no timeout is active. Any number greater than 0 indicates the number of seconds after which a session is automatically disconnected.

The second form of the `alter session` statement can be used by members of the group ADMIN only. It is used to temporarily change the user and the corresponding privileges of the session. The third form of the statements resets the user and the privileges to their original values.

**Output**

*Output* This statement returns a confirmation of a successful operation.
**alter trigger**

**Purpose**

The purpose of the *alter trigger* statement is to change properties of the specified trigger.

**Syntax**

The syntax for the *alter trigger* statement is

```
alter [ existing | trigger triggername on TRIGGEROBJECT | < noinverse | inverse > ]
with WITHITEM {, WITHITEM}
```

**TRIGGEROBJECT:**

```
resource identifier {, identifier} in folderpath
| job definition folderpath
| named resource identifier {, identifier}
| object monitor objecttypename
| resource identifier {, identifier} in serverpath
```

**WITHITEM:**

```
< active | inactive >
| check = period
| condition = < none | string >
| < nowarn | warn >
| event = ( CT_EVENT {, CT_EVENT} )
| group event
| limit state = < none | statename >
| main none
| main folderpath
| < nomaster | master >
| parameter = none
| parameter = ( identifier = expression {, identifier = expression} )
| parent none
| parent folderpath
| rerun
| < noresume | resume in period | resume at datetime >
| single event
| state = none
| state = ( < statename {, statename} |
CT_RSCSTATUSITEM {, CT_RSCSTATUSITEM} > )
```
submit after folderpath
submit folderpath
submitcount = integer
< nosuspend | suspend >
[ type = ] CT_TRiggertype
[ group = groupname ]

CT_EVENT:
< create | change | delete >

CT_RSCSTATUSITEM:
< statename any | statename statename | any statename >

CT_TRIGGERTYPE:
  after final
  before final
  finish child
  immediate local
  immediate merge
  until final
  until finished
  warning

Description

The alter trigger statement is used to change the properties of a defined trigger. If the existing keyword has been specified, changing an existing trigger will not return an error. For a detailed description of these options, refer to the create trigger statement on page 167.

Output

This statement returns a confirmation of a successful operation.
**alter user**

**Purpose**

The purpose of the `alter user` statement is to change properties of the specified user.

**Syntax**

The syntax for the `alter user` statement is

```
alter [ existing ] user username
with WITHITEM { , WITHITEM }

alter [ existing ] user username
ADD_DELITEM [ , ADD_DELITEM ]
```

**WITHITEM:**

- `connect type = < plain | ssl | ssl authenticated >`
- `default group = groupname`
- `< enable | disable >`
- `equivalent = none`
- `equivalent = ( < username | serverpath > { , < username | serverpath > } )`
- `group = ( groupname { , groupname } )`
- `parameter = none`
- `parameter = ( PARAMETERSPEC { , PARAMETERSPEC } )`
- `password = string`
- `rawpassword = string [ salt = string ]`

**ADD_DELITEM:**

- `add | or alter | parameter = ( PARAMETERSPEC { , PARAMETERSPEC } )`
- `< add | delete > group = ( groupname { , groupname } )`
- `alter [ existing ] parameter = ( PARAMETERSPEC { , PARAMETERSPEC } )`
- `delete [ existing ] parameter = ( parmlist )`

**PARAMETERSPEC:**

- `parametername = < string | number >`
Description

The *alter user* statement is used to change the properties of a defined user. If the *existing* keyword has been specified, attempting to modify a non-existent user will *not* trigger an error.

For a detailed description of these options, refer to the *create user* statement on page 177.

The second variant of the statement is used to delete or add the user from or to the specified groups.

Output

This statement returns a confirmation of a successful operation.
4. connect commands
**connect**

**Purpose**

*Purpose*  
The purpose of the `connect` statement is to authenticate a user to the server.

**Syntax**

*Syntax*  
The syntax for the `connect` statement is

```plaintext
connect username identified by string [ with WITHITEM {, WITHITEM} ]
```

WITHITEM:

- `command = ( sdms-command {, sdms-command} )`
- `method = string`
- `protocol = PROTOCOL`
- `session = string`
- `timeout = integer`
- `token = string`
- `< trace | notrace >`
- `trace level = integer`

PROTOCOL:

- `json [ ZERO TERMINATED ]`
- `line`
- `perl [ ZERO TERMINATED ]`
- `python [ ZERO TERMINATED ]`
- `serial`
- `xml`

**Description**

*Description*  
The `connect` command is used to authenticate the connected process on the server to. A communication protocol can be optionally specified. The default protocol is `line`. The selected protocol defines the output format. All protocols except for `serial` return ASCII output. The protocol `serial` returns a serialized Java object. An executable command can also be returned when the connection is established. In this case, the output of the accompanying command is used as the output for the `connect` command. If the command fails, but the `connect` was successful, the connection remains active. An example for all protocols except the `serial` protocol is given below.
**line protocol**  The line protocol only returns an ASCII text as the result from a command.

```java
connect donald identified by 'duck' with protocol = line;
```

Connect

CONNECT_TIME : 19 Jan 2005 11:12:43 GMT

Connected

SDMS>

**XML protocol**  The XML protocol returns an XML structure as the result from a command.

```java
connect donald identified by 'duck' with protocol = xml;
<OUTPUT>
<Data>
<TITLE>Connect</TITLE>
<RECORD>
<CONNECT_TIME>19 Jan 2005 11:15:16 GMT</CONNECT_TIME></RECORD>
</DATA>
<FEEDBACK>Connected</FEEDBACK>
</OUTPUT>
```

**python protocol**  The python protocol returns a Python structure that can be evaluated using the Python eval function.

```java
connect donald identified by 'duck' with protocol = python;
{ 'DATA' :
{ 'TITLE' : 'Connect',
'DESC' : [
'CONNECT_TIME'
],
'RECORD' : {
'CONNECT_TIME' : '19 Jan 2005 11:16:08 GMT'
}
,'FEEDBACK' : 'Connected'
}
```

**perl protocol**  The perl protocol returns a Perl structure that can be evaluated using the Perl eval function.
connect donald identified by 'duck' with protocol = perl;
{
'DATA' =>
{
'TITLE' => 'Connect',
'DESC' => [
'CONNECT_TIME'
],
'RECORD' => {
'CONNECT_TIME' => '19 Jan 2005 11:19:19 GMT'}
},
'FEEDBACK' => 'Connected'
}

**Output**

This statement returns a confirmation of a successful operation.
5. copy commands
**copy folder**

**Purpose**

*Purpose*  
The purpose of the *copy folder* statement is to copy a folder including all contents to some other place in the folder hierarchy.

**Syntax**

*Syntax*  
The syntax for the *copy folder* statement is

\[
\text{copy FOLDER OR JOB \{, FOLDER OR JOB\} to folderpath}
\]

\[
\text{copy FOLDER OR JOB \{, FOLDER OR JOB\} to foldername}
\]

* FOLDER OR JOB:
  * [ < folder folderpath | job definition folderpath > ]

**Description**

*Description*  
If a folder has been copied, every object in the folder is copied as well. If there are any relationships between objects that were copied as the result of a *copy folder* operation (e.g. dependencies, children, triggers, etc.), these are changed accordingly and mapped to the resulting objects from the copy.

For example, if a folder SYSTEM.X.F containing two jobs A and B, and with SYSTEM.X.F.B dependent upon SYSTEM.X.F.A, is copied to the folder SYSTEM.Y, the newly created job SYSTEM.Y.F.B will be dependent upon the newly created job SYSTEM.Y.F.A.

Note that if the jobs were copied using a *copy job definition* command, the new job SYSTEM.Y.F.B would still be dependent upon SYSTEM.X.F.A. This may *not* correspond to the user’s view.

**Output**

*Output*  
This statement returns a confirmation of a successful operation.
copy named resource

Purpose

The purpose of the *copy named resource* statement is to copy a named resource into another category.

Syntax

The syntax for the *copy named resource* statement is

```
copy named resource identifier [. identifier] to identifier [. identifier]
```

```
copy named resource identifier [. identifier] to resourcename
```

Description

The *copy named resource* command is used to save a copy of a Named Resource or an entire category.

If the specified “target resourcepath” already exists as a category, a Named Resource or category with the same name as the source object is created within this category.

If the specified “target resourcepath” already exists as a Named Resource, this is regarded as an error.

Output

This statement returns a confirmation of a successful operation.
**copy scope**

**Purpose**

The purpose of the *copy scope statement* is to copy a scope including all contents to some other place within the scope hierarchy.

**Syntax**

The syntax for the *copy scope* statement is

```
  copy < scope serverpath | jobserver serverpath > to serverpath

  copy < scope serverpath | jobserver serverpath > to scopename
```

**Description**

The *copy scope* command is used to save a copy of entire scopes. This copy also includes the resource and parameter definitions.

If the specified "target servicepath" already exists as a scope, a scope with the same name as the source object is created within this category.

If the specified "target serverpath" already exists as a jobserver, this is regarded as an error.

Since a jobserver is only regarded as a special type of scope, it is possible to copy jobservers using this command. In this case, this command is identical to the *copy jobserver* command.

**Output**

This statement returns a confirmation of a successful operation.
6. create commands
create comment

Purpose

The purpose of the create comment statement is to store a comment for the specified object.

Syntax

The syntax for the create comment statement is

```sql
create [ or alter ] comment on OBJECTURL
with CC_WITHITEM
```

OBJECTURL:

- distribution distributionname for pool identifier [ . identifier ] in serverpath
- environment environmentname
- exit state definition statename
- exit state mapping mappingname
- exit state profile profilename
- exit state translation transname
- event eventname
- resource identifier [ . identifier ] in folderpath
- folder folderpath
- footprint footprintname
- group groupname
- interval intervalname
- job definition folderpath
- job jobid
- named resource identifier [ . identifier ]
- parameter parametername of PARAM_LOC
- resource state definition statename
- resource state mapping mappingname
- resource state profile profilename
- scheduled event schedulepath . eventname
- schedule schedulepath
- resource identifier [ . identifier ] in serverpath
- < scope serverpath | jobserver serverpath >
- trigger triggername on TRIGGEROBJECT [ < noinverse | inverse > ]
- user username
Description
The `create comment` statement is used to create the condensed description or the URL of the description for the object to be commented on. The optional keyword `or alter` is used to update the comment (if one exists). If it is not specified, the presence of a comment will trigger an error.

Output
This statement returns a confirmation of a successful operation.
create environment

Purpose

Purpose  The purpose of the create environment statement is to define a set of static named resources which are needed in the scope a job wants to run.

Syntax

Syntax  The syntax for the create environment statement is

\[
\text{create [ or alter ] environment environmentname [ with ENV\_WITH\_ITEM ]}
\]

\[
\begin{align*}
\text{ENV\_WITH\_ITEM:} & \\
\text{resource} & = \text{none} \\
| & \text{resource} = ( \text{ENV\_RESOURCE} , \text{ENV\_RESOURCE} )
\end{align*}
\]

\[
\begin{align*}
\text{ENV\_RESOURCE:} & \\
\text{identifier} & = \text{identifier} \ [ < \text{condition} = \text{string} \ | \ \text{condition} = \text{none} > ]
\end{align*}
\]

Description

Description  The create environment statement is used to define a series of Static Resource Requests which describe the requisite environment that a job needs. Since the environments cannot be created by ordinary users, and jobs have to describe the environment that they require to run, environments can be used to force jobs to use a specific jobserver.

Resources  The Resources clause is used to specify the Required (Static) Resources. Specified resources that are not static will trigger an error. Since only static resources are specified, no further information is required. It is permissible to specify an empty environment (an environment without resource requests). This is not advisable, though, because it means a loss of control.

Output

Output  This statement returns a confirmation of a successful operation.
**create event**

**Purpose**

The purpose of the *create event* statement is to define an action which can be executed by the time scheduling engine.

**Syntax**

The syntax for the *create event* statement is

```
create [ or alter ] event eventname
with EVENT_WITHITEM {, EVENT_WITHITEM}
```

```
EVENT_WITHITEM:
    action =
      submit folderpath [ with parameter = ( PARAM {, PARAM} ) ]
    | group = groupname
```

```
PARAM:
    parametername = < string | number >
```

**Description**

The *create event* statement is used to define an action that can be scheduled by the Time Scheduling module. The defined action is the submission of a master submittable job or batch.

**action** The submit part of the statement is a restricted variant of the submit command (see page 414).

**group** The group option is used to set the owner group to the specified value. The user must belong to this group unless he belongs to the ADMIN privileged group. In this case, any group can be specified.

**Output**

This statement returns a confirmation of a successful operation.
**create exit state definition**

**Purpose**

*Purpose* The purpose of the *create exit state definition* statement is to create a symbolic name for the state of a job.

**Syntax**

*Syntax* The syntax for the *create exit state definition* statement is

```
create [ or alter ] exit state definition statename
```

**Description**

*Description* The *create exit state definition* statement is used to create a symbolic name for the Exit State of a job, milestone or batch. The optional keyword *or alter* is used to prevent error messages from being triggered and the current transaction from being aborted if an Exit State Definition already exists. This is particularly useful in conjunction with *multicommands*. If it is not specified, the existence of an Exit State Definition with the specified name will trigger an error.

**Output**

*Output* This statement returns a confirmation of a successful operation.

**Example**

*Example* In the following examples, symbolic names have been created for Job States.

```plaintext
create exit state definition success;
create exit state definition error;
create exit state definition reached;
create exit state definition warning;
create exit state definition wait;
create exit state definition skip;
create exit state definition unreachable;
```
create exit state mapping

Purpose

The purpose of the create exit state mapping statement is to create a mapping between the numerical exit code of a process and a symbolic exit state.

Syntax

The syntax for the create exit state mapping statement is

```
create [ or alter ] exit state mapping mappingname
with map = ( statename { , signed_integer , statename } )
```

Description

The create exit state mapping statement defines the mapping of Exit Codes to logical Exit States. The simplest form of this statement only specifies one Exit State. This means that the job automatically reaches this Exit State after it has finished regardless of its Exit Code. More complex definitions specify more than one Exit State and at least one delimitation.

Output

This statement returns a confirmation of a successful operation.

Example

The example below shows a relatively simple, yet realistic mapping of Exit Codes to logical Exit States.

The statement

```
create exit state mapping example1
with map = ( error, 0, success, 1, warning, 4, error);
```

defines the following mapping:

<table>
<thead>
<tr>
<th>Exit Code Range from</th>
<th>Exit Code Range to</th>
<th>Resultant Exit State</th>
</tr>
</thead>
<tbody>
<tr>
<td>−∞</td>
<td>−1</td>
<td>error</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>success</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>warning</td>
</tr>
<tr>
<td>4</td>
<td>∞</td>
<td>error</td>
</tr>
</tbody>
</table>
create exit state profile

Purpose

The purpose of the `create exit state profile` statement is to define a set of valid exit states.

Syntax

The syntax for the `create exit state profile` statement is:

```
create [ or alter ] exit state profile  profilename
with WITHITEM {, WITHITEM}

WITHITEM:
  default mapping = < none | mappingname >
  |  force
  |  state = ( ESP_STATE {, ESP_STATE} )

ESP_STATE:
  statename < final | restartable | pending > [ OPTION { OPTION } ]

OPTION:
  batch default
  |  broken
  |  dependency default
  |  disable
  |  unreachable
```

Description

The `create exit state profile` statement is used to define a quantity of valid Exit States for a job, milestone or batch.

**default mapping**  With the default mapping clause it is possible to define which Exit State Mapping is to be used if no other mapping has been specified. This makes it considerably easier to create jobs.

**force**  While an Exit State Profile is being created, the force option has no effect and is ignored. If `or alter` is specified and the Exit State Profile that you want to create already exists, the force option delays the integrity check until later.
**state**  The state clause defines which Exit State Profiles are valid within this definition. Each Exit State Definition must be classified as being **final**, **restartable** or **pending**. If a job has reached the **final** state it can no longer be started, which means that the state can no longer change. If a job has reached the **restartable** state, it can be started again. This means that the state of such a job can change as well. **pending** means that a job cannot be restarted, but it is not final either. The state must be set externally.

The order in which the Exit States are defined is relevant. The first specified Exit State has the highest preference, while the most recently specified Exit State has the lowest preference. Normally, **final** states are specified later than **restartable** states. A state’s preference is used to decide which state is visible when several different Exit States of children are merged.

Just one Exit State can be declared as being an **unreachable** state. This means that a job, batch or milestone with this profile is mapped to the specified state as soon as it has become unreachable. This Exit State must be **final**.

A maximum of one Exit State within a profile can be designated as being a **broken** state. This means that a job will reach this state as soon as it has switched to the **error** or **broken_finished** state. This can be handled using a trigger. The Exit State that is defined as being a **broken** state must be **restartable**.

A maximum of one state can be declared as being a **batch default** state. An empty batch assumes this status. This allows for an explicit deviation from the standard behaviour. If no status is designated as being **batch default**, an empty batch will automatically assume the final status with the lowest preference that is not designated as being **unreachable**. If such a status does not exist, the **unreachable** state is also considered a candidate.

Any number of Final States can be designated as **dependency default** states. Dependencies that define a default dependency are fulfilled if the required job assumes one of the states designated as **dependency default**.

**Output**

This statement returns a confirmation of a successful operation.

**Example**

These examples show how the Exit State Profiles **example_1** and **example_2** are created.

In the first, very simple example, the Exit State of **success** is to be a Final State.

```
create exit state profile example_1
with
    state = ( success final );
```
In the second example, the Exit State failure is defined as being restartable. This state has a higher priority than the (final) state success and must therefore be listed as the first state.

```plaintext
create exit state profile example_2 with
  state = ( failure restartable,
            success final
  );
```
create exit state translation

Purpose

The purpose of the create exit state translation statement is to create a translation between child and parent exit states.

Syntax

The syntax for the create exit state translation statement is

\[
\text{create [ or alter ] exit state translation transname}
\]

\[
\text{with translation = ( statename to statename [, statename to statename] )}
\]

Description

The create exit state translation statement is used to define a translation between two Exit State Profiles. Such a translation can be used (but does not have to be) in parent-child relationships if the two involved Exit State Profiles are incompatible. The default translation is the identity. This means that Exit States are translated to Exit States of the same name unless specified otherwise. It is not possible to translate a Final State to a Restartable State. If the Exit State translation already exists and the "or alter" keyword has been specified, the specified Exit State translation is changed. Otherwise, an already existing Exit State translation with the same name will trigger an error.

Output

This statement returns a confirmation of a successful operation.

Example

In the following example, the Exit State of the child warning is translated to the Exit State of the parent skip

\[
\text{create exit state translation example1}
\]

\[
\text{with translation = ( warning to skip );}
\]
create folder

Purpose

The purpose of the `create folder` statement is to create a container for job definitions and/or other folders.

Syntax

The syntax for the `create folder` statement is

```
create [ or alter ] folder folderpath [ with WITHITEM {, WITHITEM} ]
```

WITHITEM:

```
environment = < none | environmentname >
group = groupname [ cascade ]
inherit grant = none
inherit grant = ( PRIVILEGE {, PRIVILEGE} )
parameter = none
parameter = ( parametername = string {, parametername = string} )
```

PRIVILEGE:

```
approve
cancel
clear warning
close
create content
drop
edit [ parameter ]
enable
execute
ignore resource
ignore dependency
kill
monitor
operate
priority
rerun
resource
set job status
set state
submit
suspend
```
Description

This command creates a folder and has the following options:

**environment**  If an environment has been assigned to a folder, every job in the folder and its subfolders will inherit all the Resource Requests from the Environment Definition.

**group**  The group option is used to set the owner group to the specified value. The user must belong to this group unless he belongs to the ADMIN privileged group. In this case, any group can be specified.

**parameter**  The parameter option can be used to define key/value pairs for the folder. The complete list of parameters must be specified within a command.

**inherit grant**  The inherit grants clause allows you to define which privileges are to be inherited through the hierarchy. If this clause is not specified, all privileges are inherited by default.

Output

This statement returns a confirmation of a successful operation.
create footprint

Purpose

The purpose of the *create footprint* statement is to create a set of often used system resource requirements.

Syntax

The syntax for the *create footprint* statement is

```
create [or alter] footprint footprintname
with resource = ( REQUIREMENT {, REQUIREMENT} )
```

```
REQUIREMENT:
ITEM { ITEM}
```

```
ITEM:
amount = integer
| < nokeep | keep | keep final >
| identifier {. identifier}
```

Description

The *create footprint* command creates a set of Resource Requests which can be re-used. The Required Resources are all System Resources. The Required Resources are described by their names, a set with zero by default, and optionally a keep option.

keep

The keep option in a Resource Request defines the time when the resource is released. The keep option is valid for both System and Synchronizing Resources. There are three possible values. Their meanings are explained in the table below:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>nokeep</td>
<td>The resource is released at the end of the job. This is the default behaviour.</td>
</tr>
<tr>
<td>keep</td>
<td>The resource is released as soon as the job has reached the Final State.</td>
</tr>
<tr>
<td>keep final</td>
<td>The resource is released when the job and all its children are final.</td>
</tr>
</tbody>
</table>
amount  The amount option is only valid with requests for Named Resources of the type System or Synchronizing. The amount in a Resource Request expresses how many units of the Required Resource are allocated.

Output

This statement returns a confirmation of a successful operation.
create group

Purpose

Purpose
The purpose of the create group statement is to create an object to which privileges can be granted.

Syntax

Syntax
The syntax for the create group statement is

\[
\text{create [ or alter ] group groupname [ with WITHITEM ]}
\]

WITHITEM:
\[
\begin{align*}
\text{user} &= \text{none} \\
&\quad \mid \quad \text{user} = ( \text{username} \,, \text{username} )
\end{align*}
\]

Description

Description
The create group statement is used to create a group. If the “or alter” keyword has been specified, an already existing group is changed. Otherwise, an already existing group is considered an error.

user
The user clause is used to specify which users are group members.

Output

Output
This statement returns a confirmation of a successful operation.
create interval

Purpose

The purpose of the create interval statement is to define a periodic or aperiodic pattern at which events can, must not, be triggered.

Syntax

The syntax for the create interval statement is

\[
\text{create [ or alter]} \ interval \ intervalname \ [ \text{with} \ \text{WITHITEM} \ {,} \ \text{WITHITEM} \ } \\
\]

WITHITEM:

- base = < none | period >
- dispatch = none
- dispatch = (IVAL_DISPATCHITEM {, IVAL_DISPATCHITEM})
- duration = < none | period >
- embedded = < none | CINTERVALNAME >
- endtime = < none | datetime >
- filter = none
- filter = ( CINTERVALNAME {, CINTERVALNAME} )
- < noinverse | inverse >
- selection = none
- selection = (IVAL_SELITEM {, IVAL_SELITEM})
- starttime = < none | datetime >
- synctime = datetime
- group = groupname

IVAL_DISPATCHITEM:

\[
\text{dispatchname} < \text{active} | \text{inactive} > \text{IVAL_DISPATCHDEF} \\
\]

CINTERVALNAME:

\[
(\ intervalname \text{with} \ \text{WITHITEM} {,} \ \text{WITHITEM} ) \text{ intervalname} \\
\]

IVAL_SELITEM:

\[
< \text{signed_integer} | \text{datetime} | \text{datetime - datetime} > \\
\]

123
The intervals are the core of the Time Scheduling. They can be regarded as block patterns. These patterns can be periodic or non-periodic. Within a **period (Base)** which, in the case of a non-periodic interval, has a length infinity (∞), there are blocks of a predetermined length **Duration**. The last block may be incomplete if the period length is not an integer multiple of the duration is. The duration can also have a length ∞. This means that the blocks have the same length as the periods.

It is not necessary for all of the blocks to be actually present. You can choose which blocks are present. This **choice** can be made by specifying the block number relative to the beginning or end of a period (1, 2, 3 or −1, −2, −3) or by stating "from - to" (all days between 3.4. and 7.6.). This results in complex patterns as shown in Figure 6.2.

The selection is 1-based, i.e. the first block has the number 1. The last block is addressed with the number -1. This means that a block 0 does not exist.

**Infinite intervals** With a non-periodic interval without a duration (infinity), the start time plays a special role: it then defines the only positive edge of this interval. Similarly, an end time defines the only negative edge.
When a selection is made, this respectively results in blocks being created. The selection ",-0315T18:40," creates a block from 18:40 to 18:41 every year on March 15. Selecting blocks using the position (first, second, etc.) is, of course, nonsense. This is also ignored for infinite intervals.

**Inverse** If, for example, the time between Christmas and New Year has been positively defined for a particular purpose, at the moment there is no way to easily define the complementary time. In this example this is not a serious problem, but with more complex patterns this incapability will result in complex and error-prone dual definitions.

For this reason, an Inverse flag has been implemented which causes the specified selection list to be interpreted complementarily, i.e. only those blocks that would not have been chosen without a set invert flag are selected. In the case of the last working day of the month, the inverse flag is set on all working days except for the last working day of that month.

**Filter** The selection of blocks can be restricted even further. For example, if you have defined an interval "day of the month" (i.e. the base is one month, the duration is one day) and then selected the second block, such an interval would have a block on the respective second day of a month. If you want to define this only for the odd months (January, March, May, etc.), that would not be possible without a filter function because of the leap years.

The solution to the problem is to define a further interval (month of the year) with the selection 1, 3, 5, 7, 9, 11. This interval is then specified as a filter for the first interval.

Here, the first interval only shows a block if the second interval also shows a block at that "time".

If several intervals have been specified as a filter, it is sufficient for one of these intervals to have a block at the required time (OR). To map an AND relationship between the filter intervals, the filter intervals are created as a chain (A filters B C filters, etc.). The order of the filters is not important.

**Embedded** Unfortunately, the world is not always so simple. In particular, it is not inconsequential whether you first perform an operation and then make a selection, or if you have to choose first and then perform the operation. In other words, there is a big difference if you talking about the last day of the month - if this is a working day - or about the last working day of the month.

We obviously also want to include this possibility for making a differentiation in our model. An embedding functionality has been implemented for this purpose.

Here, we begin by taking over all the parameters for the embedded interval. This is followed by an evaluation of the selection list. Although it is allowed, selecting a "from - to" period is obviously senseless since this functionality can also be
achieved with simple multiplication. Much more interesting is the possibility of making a relative selection. If the working days in a month are embedded and then the day \(-1\) is selected, for instance, overall we now have an interval that defines the last working day of each month. If, on the other hand, the interval with the working days in a month is multiplied by an interval that returns the last day of a month, we will only get a hit if the last day of the month is a working day. Embedding can therefore also be understood as follows: When selecting the blocks, not all of the embedded blocks are considered (and above all counted), but only the active blocks.

**Synchronisation** What have still not been taken into consideration are those situations involving multiple single periods. A period of 40 days, for example, could have its rising edge at midnight (00:00) on any day. That is why a synchronisation time (synctime) has been implemented which selects the earliest edge that is \(\geq\) this point in time. If no such time has been explicitly specified, the date when the definition was created (create) is used. Fundamentally, the first block of a period initially starts at its beginning. In cases where this is not possible (period = \(\infty\), duration \(\geq\) period, Period XOR Duration have the unit “week”), the beginning of the period is used as the synchronisation time. If this is not possible either (period = \(\infty\)), the normal synchronisation time is used. The result of this approach is that the first block of a period may be incomplete as well (and is then never active).

**Dispatcher** Although the previous syntax components are extremely powerful and can describe practically any rhythm, their usage is not always intuitive. This is not problematic when the interval is created, but it can become a problem during later maintenance. The Dispatcher allows the user to develop interval definitions which are much easier to understand. As an example, let us assume that a job is to be started at 10:00 on Mondays, but at 09:00 on the other days of the week. First of all, we develop an interval that is triggered at 10:00 on Mondays:

```sql
create or alter interval MONDAY10
with
base = none,
duration = none,
selection = ('T10:00'),
filter = (MONDAYS
with
base = 1 week,
duration = 1 day,
selection = (1))
);
```

126
The possibility to define filters and embedded intervals “inline” can result in a streamlined definition here.

The interval that is triggered at 09:00 on the other days of the week looks similar to this:

```sql
create or alter interval WEEKDAY09
with
base = none,
duration = none,
selection = ('T09:00'),
filter = (WEEKDAYS
    with
    base = 1 week,
duration = 1 day,
selection = (2, 3, 4, 5)
);
```

The combined interval without a Dispatcher therefore looks like this:

```sql
create or alter interval MO10_DI_FR09
with
base = none,
duration = none,
selection = ('T09:00', 'T10:00'),
filter = (MONDAY10, WEEKDAY09);
```

The two possible times are selected and both filters are evaluated. On Mondays, only the time 10:00 is let through, on other days only the time 9:00.

The same functionality, but now with a Dispatcher, is easier to understand:

```sql
create or alter interval D_MO10_DI_FR09
with
base = none,
duration = none,
filter = none,
selection = none,
dispatch = (MONDAY_RULE
    active
    (MONDAYS
        with
        base = 1 week,
duration = 1 day,
selection = (1)
    )
    (MONDAY_TIME
        with
        base = none,
duration = none,

```
selection = ('T10:00')
)
enable,
WEEKDAY_RULE
active
(WEEKDAYS
with
base = 1 week,
duration = 1 day,
selection = (2, 3, 4, 5)
)
(WEEKDAY_TIME
with
base = none,
duration = none,
selection = ('T09:00')
)
enable
);

The requirement is clearly presented in this form, easy to understand and just as easy to maintain.

A Dispatcher definition is relatively simple. First of all, it consists of a list of rules. The order of these rules is meaningful. If two or more rules are "responsible", the first rule in the list "wins".

In the example above, the WEEKDAYS interval could be changed so that the Monday is selected:

...  
WEEKDAY_RULE
active
(WEEKDAYS
with
base = 1 week,
duration = 1 day,
selection = (1, 2, 3, 4, 5)
)
...

But since the first rule MONDAY_RULE is already handling the Monday, the change would not have any effect.

A Dispatch rule consists of 5 parts. It begins with a name that must comply with the usual rules for an identifier. The name has no implication, and essentially serves as a way of clarifying the idea behind the rule. The name (as the name of a rule) must be unique within the Dispatcher.

The next part is the active flag. If it is set to inactive, no blocks are generated, respectively all blocks are filtered out. If it is set to active, the Interval filter is valuated.
The third part is the “Select Interval”. This interval defines the times at which the rule is valid. If the rule is valid, the Interval value is valuated provided that the rule is marked as being active.
If the keyword none is entered as the Select Interval, this equates to an infinite interval without any other properties. In turn, this basically means that it is always valid.
The fourth part is the “Filter Interval”. This interval does the actual work. In the example above, it creates a block with a start time of 09:00 (Mondays).
The Filter Interval can be omitted. Here, too, this equates to an infinite interval without any other properties. As a driver there are no blocks; as a filter it lets everything through.
The combination of none as Select Interval and omitting the Filter Interval is not permissible.
The last part is the enable flag. This switch can be used to enable or disable rules. If a rule is disabled, it is ignored.

Output

This statement returns a confirmation of a successful operation.
**create job definition**

**Purpose**

The purpose of the `create job definition` statement is to create a scheduling entity object which can be submitted, standalone or as part of a larger hierarchy.

**Syntax**

The syntax for the `create job definition` statement is

```
create [ or alter ] job definition folderpath . jobname
with WITHITEM {, WITHITEM}
```

WITHITEM:

- `approval = none`
- `approval = ( OPERATE_APPROVAL {, OPERATE_APPROVAL} )`
- `children = none`
- `children = ( JOB_CHILDDEF {, JOB_CHILDDEF} )`
- `dependency mode = < all | any >`
- `environment = environmentname`
- `errlog = < none | filespec [ < notrunc | trunc > ] >`
- `footprint = < none | footprintname >`
- `inherit grant = none`
- `inherit grant = ( PRIVILEGE {, PRIVILEGE} )`
- `kill program = < none | string >`
- `logfile = < none | filespec [ < notrunc | trunc > ] >`
- `mapping = < none | mappingname >`
- `< nomaster | master >`
- `nicevalue = < none | signed_integer >`
- `parameter = none`
- `parameter = ( JOB_PARAMETER {, JOB_PARAMETER} )`
- `priority = < none | signed_integer >`
- `profile = profilename`
- `required = none`
- `required = ( JOB_REQUIRED {, JOB_REQUIRED} )`
- `rerun program = < none | string >`
- `resource = none`
- `resource = ( REQUIREMENT {, REQUIREMENT} )`
- `< noresume | resume in period | resume at datetime >`
- `runtime = integer`
- `runtime final = integer`
- `run program = < none | string >`
< nosuspend | suspend >
timeout = none
timeout = period state statename
type = < job | milestone | batch >
group = groupname
workdir = < none | string >

OPERATE_APPROVAL:
OPERATE_PRIV APPROVAL_MODE [ leading ]

JOB_CHILDDEF:
JCD_ITEM { JCD_ITEM }

PRIVILEGE:
approve
cancel
clear warning
clone
create content
drop
edit [ parameter ]
enable
execute
ignore resource
ignore dependency
kill
monitor
operate
priority
rerun
resource
set job status
set state
submit
suspend
use
view

JOB_PARAMETER:

parametername { ( id ) } < [ JP_WITHITEM ] [ default = string ] | JP_NONDEFWITH >
[ local ] [ < export = parametername | export = none > ]
JOB_REQUIRED:
  JRQ_ITEM { JRQ_ITEM }

REQUIREMENT:
  JRD_ITEM { JRD_ITEM }

OPERATE_PRIV:
  cancel
  | clear warning
  | clone
  | edit parameter
  | enable
  | ignore resource
  | ignore dependency
  | kill
  | priority
  | rerun
  | set job status
  | set state
  | suspend

APPROVAL_MODE:
  approve
  | default
  | master
  | no
  | parent
  | review

JCD_ITEM:
  alias = < none | aliasname >
  | condition = < none | string >
  | < enable | disable >
  | folderpath . jobname
  | ignore dependency = none
  | ignore dependency = ( dependencyname { , dependencyname } )
  | interval = < none | intervalname >
  | < childsuspend | suspend | nosuspend >
  | merge mode = < nomerge | merge local | merge global | failure >
  | mode = < and | or >
nicevalue = < none | signed_integer >
priority = < none | signed_integer >
< noresume | resume in period | resume at datetime >
< static | dynamic >
translation = < none | transname >

JP_WITHITEM:
import [ unresolved ]
| parameter
| reference child folderpath ( parametername )
| reference folderpath ( parametername )
| reference resource identifier [ identifier ] ( parametername )
| result

JP_NONDEFWITH:
constant = string
| JP_AGGFUNCTION ( parametername )

JRQ_ITEM:
condition = < none | string >
dependency dependencyname
expired = < none | signed_period_rj >
folderpath . jobname
mode = < all final | job final >
resolve = < internal | external | both >
select condition = < none | string >
state = none
state = { JRQ_REQ_STATE | JRQ_REQ_STATE }
state = all reachable
state = default
state = unreachable
unresolved = JRQ_UNRESOLVED

JRD_ITEM:
amount = integer
expired = < none | signed_period >
< nokeep | keep | keep final >
condition = < none | string >
lockmode = LOCKMODE
nosticky

133
This command creates or (optionally) modifies job, batch or milestone definitions.

As jobs, batches and milestones have a lot in common, we usually use the following general technical term "Scheduling Entity" whenever the behaviour is the same for all three types of Job Definition. The expressions "Job", "Batch" and "Milestone" are used for Scheduling Entities of the corresponding type Job, Batch and Milestone. If the "or alter" modifier is used, the command will alter this according to the specified options if a scheduling entity with the same name already exists.
The aging describes how quickly the priority is upgraded.

The Approval System allows a 4-eyes principle (approval) or at least a subsequent review to be activated for all important operator actions (cancel, re-run, enable/disable, set state, ignore dependency, ignore resource, clone, edit parameter, kill and set job state).

An approval is considered to be more restrictive than a review, as with an approval the operation is only carried out after it has been approved by an authorised person, whereas a review only ever takes place after the operation has been carried out.

This can be activated for the entire workflow or just for parts of it. The behaviour is defined by means of an Approval mode and a Leading flag per operation.

The following Approval modes can be used:

<table>
<thead>
<tr>
<th>Approval mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>APPROVE</td>
<td>The action must be approved by an authorised person.</td>
</tr>
<tr>
<td>PARENT</td>
<td>The setting for the parent is taken over.</td>
</tr>
<tr>
<td>MASTER</td>
<td>The setting for the master is taken over if a more restrictive setting of a leading parent does not exist. This is the default behaviour.</td>
</tr>
<tr>
<td>REVIEW</td>
<td>After the action has been performed, a review is requested unless a more restrictive setting exists.</td>
</tr>
<tr>
<td>NO</td>
<td>No restriction unless a more restrictive setting of a leading parent exists.</td>
</tr>
</tbody>
</table>

If no explicit setting is made at master level, the NO setting applies here. This means that the Approval System is normally not active.

If Approval (or Review) mode is set for a job or batch without a Leading flag, this only has an effect on the job or batch itself as well as on all children that have set PARENT as the mode.

Setting the Leading flag means that the setting of the parent cannot be ignored. It is taken over unless a more restrictive setting applies.

Ultimately, configuring the system is simple and requires little effort. If no approvals or reviews are required, nothing needs to be done. If approvals or reviews are required for an entire workflow, the desired mode is set at master level with a Leading flag. If an approval or review is required for a part of the workflow, this is set at parent level with a Leading flag. Individual objects can be given their own setting. The Leading flag causes all objects further down in the hierarchy to take over at least the setting of the parent. If the Leading flag is missing, the setting is only passed on to direct children that have specified the PARENT mode.

The Children section of a job definition statement defines a list of child objects and is used to build up a hierarchy that enables the modelling of complex...
job structures. Whenever a Scheduling Entity is submitted, all the static children are recursively submitted.

In addition, children that are not static can be submitted during the execution be a Running Job or Trigger.

The children are then specified using a comma-separated list of Scheduling Entity path names and additional properties.

The properties of the Child Definitions are described below:

**ALIAS** This option allows the implementation of the submitted jobs to be kept independent of the folder structure, and it will function regardless of whether objects are moved within the folder structure.

The alias for a Child Definition is only used when jobs submit dynamic children.

**ENABLE** In the parent-child relationship, you can specify whether a child is to be enabled (default) or disabled. This can either be set unconditionally or dependent upon the time of the submit or the result of a condition, or even both.

If an interval is specified, the job will be enabled if the submit time would be let through if the interval were to be used as a filter.

If a job is disabled, it will behave like an empty batch. As soon as all the dependencies have been fulfilled, the job assumes the Exit Status, which is final and the batch default, or the Final Status with the lowest preference and which can be attained via an Exit Status Mapping.

**IGNORE DEPENDENCY** Dependencies of parent jobs are normally inherited by their children. In some rare situations this is undesirable. In this case the ignore dependency option can be used to ignore such dependencies.

**MERGE MODE** A single Scheduling Entity can be used as a child of more than one Parent Scheduling Entity. If two or more such parents are part of a Master Run, the same children are repeatedly instantiated within this Master Run. This is not always a desirable situation. Setting the Merge Mode controls how the system handles this scenario.

The following table gives an overview of the possible Merge Modes and their meanings:

<table>
<thead>
<tr>
<th>merge mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nomerge</td>
<td>A duplicate instance of the Scheduling Entity is created. This is the default behaviour.</td>
</tr>
<tr>
<td>merge global</td>
<td>A duplicate instance is not created. A link is created between the Parent Submitted Entity and the already existing Child Submitted Entity.</td>
</tr>
<tr>
<td>merge local</td>
<td>Like Merge Global, but only Submitted Entities that were created in a single submit are merged.</td>
</tr>
<tr>
<td>failure</td>
<td>The submit attempting to create a duplicate Submitted Entity fails.</td>
</tr>
</tbody>
</table>

136
**NICEVALUE** The nice value defines an offset of the priority used to calculate the priorities of the child and its children. Values between -100 and 100 are permitted.

**PRIORITY** The specified priority in a Child Definition overwrites the priority of the Child Scheduling Entity Definition. Values between 0 (high priority) and 100 (low priority) are permitted.

**TRANSLATION** Setting the Exit State Translation for a child results in the Exit State of the child being translated to an Exit State which is merged in the resultant Exit State of the Parent Submitted Entity.

If no translation is specified, a Child State that is not at the same time a valid Parent State is ignored.

If a translation has been specified, all the Child States have to translated to a valid Parent State.

**SUSPEND CLAUSE** The child suspend clause defines whether a new Submitted Job is suspended in the context of this Child Definition.

The table below shows the possible values and their meaning regarding the suspend clause:

<table>
<thead>
<tr>
<th>suspend clause</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>suspend</td>
<td>The child is suspended regardless of the value of the suspend flag specified in the Child Scheduling Entities.</td>
</tr>
<tr>
<td>nosuspend</td>
<td>The child is not suspended regardless of the value of the suspend flag specified in the Child Scheduling Entities Definition.</td>
</tr>
<tr>
<td>childsuspend</td>
<td>The child is suspended if the suspended flag has been set in the Child Scheduling Entity.</td>
</tr>
</tbody>
</table>

If **suspend** has been specified, a resume clause can optionally be given as well which triggers an automatic resume at the specified time or at the end of the specified interval.

The submit time is taken as the reference for partially qualified points in time. T16:00 means, therefore, that if the submit time 15:00 has been set, the job will start after about an hour. If the submit time is later than 16:00, however, the job will wait until the next day.

**DYNAMIC CLAUSE** The child dynamic clause defines whether the child is always automatically submitted by the system when the parent is submitted as well.

Dynamic children are used by Running Jobs in the context of Trigger Definitions and programmatic submits. To be able to submit a child, this child must be defined as a dynamic child.

The table below shows the possible values in the dynamic clause and their meanings.
dynamic clause | Description
--- | ---
static | The child is automatically submitted with the parent.
dynamic | The child is not automatically submitted with the parent.

Milestones use different semantics for their children. Whenever a Scheduling Entity is dynamically submitted in a Master Run that is also a child of a milestone in the same Master Run, the Submitted Scheduling Entity is bound to this milestone as a child. This means that a milestone can only be final if its dependencies have been fulfilled and all its children are final. In other words, a Milestone collects child instances that are dynamically submitted by other Submitted Entities and waits until these Submitted Entities have finished. For this to function correctly, a dependency of the Submitted Scheduling Entity should be defined.

**dependency mode** The dependency mode defines which Required Submitted Entities have to achieve a Final State before the dependent Submitted Entity can exit the ‘Dependency Wait’ System State.

The table below shows the possible Dependency Modes and their meanings.

<table>
<thead>
<tr>
<th>dependency mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>all</td>
<td>The Submitted Entity exits the Dependency Wait State after all the dependencies have been fulfilled.</td>
</tr>
<tr>
<td>any</td>
<td>The Submitted Entity exits the Dependency Wait State after at least one dependency has been fulfilled.</td>
</tr>
</tbody>
</table>

**environment** Each job has to define which environment is needed to execute the job.
The job can only be executed by jobservers that fulfil all the Static Resource requirements listed in the Environment Definition.
The environment option only applies for jobs.

**errlog** The errlog option defines the file where error outputs (stderr) from the process to be executed are written.
If the file name is relative, the file is created relative to the working directory of the job.
This option is only valid for jobs.

**footprint** Footprints are sets of requirements for System Resources. If several jobs are defined with similar requirements, this is made that much easier by using footprints.
The job can only be executed by jobservers that fulfil all the Static Resource requirements listed in the Footprint Definition.
The footprint option only applies for jobs.
**group**  The group option is used to set the owner group to the specified value. The user must belong to this group unless he belongs to the ADMIN privileged group. In this case, any group can be specified.

**inherit grant**  The inherit grants clause allows you to define which privileges are to be inherited through the hierarchy. If this clause is not specified, all privileges are inherited by default.

**kill program**  This option is used to create the possibility for prematurely terminating running processes from within the Scheduling System. Usually, the kill program contains the PID of the Running Job as a parameter (e.g. `kill -9 ${PID}`). For details about command line parsing, variants and parameter substitutions, refer to the “run program” option on page 148.

**logfile**  The logfile option defines the file where the standard output (STDOUT) from the process to be executed is written. If the file name is relative, the file is created relative to the working directory of the job. This option is only valid for jobs.

**mapping**  The mapping option defines the Exit State Mapping that is used to translate operating system Exit Codes of an executable program to an Exit State. If a job does not have a mapping, the default Exit State Mapping of the job’s Exit State Profile is used. For a detailed description of the Exit State Mapping, refer to the ”create exit state mapping” command on page 113.

**master**  The master option defines whether this Scheduling Entity can be submitted in order to create a Master Run.

**nicevalue**  The nicevalue option defines a correction that is used for the calculation of the priorities for the job and its children. Values between -100 and 100 are permitted.

**parameter**  The parameters section defines which parameters and input values are required by a job and how the job exchanges data with other jobs and the scheduling system. The parameters can be used in the specification of the Run program, Rerun program, Kill program, working directory, log file and error log file, as well as in triggers and Dependency Conditions.
A job can also query or set parameters at runtime. Variables that have been defined at runtime and not by the job definition are only visible to the job itself and cannot be referenced. The same is also true, of course, for all variables that are defined as local as well as for the system variables mentioned below. Occasionally, however, it is necessary to make one or more of the (e.g.) system variables known externally. This can be easily done by means of a small trick. If the value of a parameter contains a character string in the form \$something (i.e. the characters $ followed by a name), this is interpreted as being the name of a variable, and an attempt is made to resolve this variable in the scope of the object that delivered the original value for the parameter.

This is how, for example, a job SYSTEM.A can define a constant called MYJOBNAME with \$JOBNAME as its content. If the constant MYJOBNAME is now addressed from outside the system via a reference, the delivered result is the value SYSTEM.A.

A number of system variables are always defined for each job. These are set by the system and can be read by the job.

These system variables are:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>JOBID</td>
<td>Submitted entity id for the job</td>
</tr>
<tr>
<td>MASTERID</td>
<td>Submitted entity id for the Master Job or Batch</td>
</tr>
<tr>
<td>KEY</td>
<td>&quot;Password&quot; of the job for connecting to the scheduling system as a job with &quot;JOBID&quot;</td>
</tr>
<tr>
<td>PID</td>
<td>The operating system process id of the job. This parameter is only set for Kill programs.</td>
</tr>
<tr>
<td>LOGFILE</td>
<td>Name of the log file (stdout)</td>
</tr>
<tr>
<td>ERRORLOG</td>
<td>Name of the error log file (stderr)</td>
</tr>
<tr>
<td>SDMHOST</td>
<td>Host name of the scheduling server</td>
</tr>
<tr>
<td>SDMSPORT</td>
<td>Listen port of the scheduling server</td>
</tr>
<tr>
<td>JOBNAME</td>
<td>Name of the job</td>
</tr>
<tr>
<td>JOBTAG</td>
<td>Child tag for the job is given if the job is being dynamically submitted</td>
</tr>
<tr>
<td>TRIGGERNAME</td>
<td>Name of the trigger</td>
</tr>
<tr>
<td>TRIGGERTYPE</td>
<td>Type of trigger (JOB_DEFINITION or NAMEDRESOURCE)</td>
</tr>
<tr>
<td>TRIGGERBASE</td>
<td>Name of the triggering object that activates the trigger</td>
</tr>
<tr>
<td>TRIGGERBASEID</td>
<td>ID of the triggering Object Definition that activates the trigger</td>
</tr>
<tr>
<td>TRIGGERBASEJOBID</td>
<td>ID of the triggering object that activates the trigger</td>
</tr>
<tr>
<td>TRIGGERORIGIN</td>
<td>Name of the triggering object that defines the trigger</td>
</tr>
<tr>
<td>TRIGGERORIGINID</td>
<td>ID of the triggering Object Definition that defines the trigger</td>
</tr>
</tbody>
</table>

Continued on next page
The TRIGGER... system variables are only populated if the job was submitted by a trigger. For a more detailed description of the TRIGGER... system variables, refer to the create trigger statement on page 167.

When a job is executed, the parameters used in commands, workdir and file specifications are resolved conform to the sequence given below:

1. System variable
2. The job’s own address space

### Table 6.1. List of System Variables

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRIGGERORIGINJOBID</td>
<td>ID of the triggering object that defines the trigger</td>
</tr>
<tr>
<td>TRIGGERREASON</td>
<td>Name of the triggering object that directly or indirectly activates the trigger</td>
</tr>
<tr>
<td>TRIGGERREASONID</td>
<td>ID of the triggering Object Definition that directly or indirectly activates the trigger</td>
</tr>
<tr>
<td>TRIGGERREASONJOBID</td>
<td>ID of the triggering object that directly or indirectly activates the trigger</td>
</tr>
<tr>
<td>TRIGGERSEQNO</td>
<td>Number of times the trigger was activated</td>
</tr>
<tr>
<td>TRIGGERREASONSTATE</td>
<td>(New) status of the object caused by the trigger for Resource Trigger</td>
</tr>
<tr>
<td>TRIGGERNEWSTATE</td>
<td>(New) status of the object that causes the trigger to be activated</td>
</tr>
<tr>
<td>SUBMITTIME</td>
<td>Submit time</td>
</tr>
<tr>
<td>STARTTIME</td>
<td>Start time</td>
</tr>
<tr>
<td>EXPRUNTIME</td>
<td>Expected runtime</td>
</tr>
<tr>
<td>JOBSITE</td>
<td>Exit State of the job</td>
</tr>
<tr>
<td>MERGEDSTATE</td>
<td>Merged Exit State of the job</td>
</tr>
<tr>
<td>PARENTID</td>
<td>ID of the Parent Job (submission tree)</td>
</tr>
<tr>
<td>STATE</td>
<td>Current state of the job (Running, Finished, etc.)</td>
</tr>
<tr>
<td>ISRESTARTABLE</td>
<td>Is the job restartable? 1 = yes, 0 = no</td>
</tr>
<tr>
<td>SYNCTIME</td>
<td>Time of the transition to Synchronize Wait</td>
</tr>
<tr>
<td>RESOURCETIME</td>
<td>Time of the transition to Resource Wait</td>
</tr>
<tr>
<td>RUNNABLETIME</td>
<td>Time of the transition to Runnable</td>
</tr>
<tr>
<td>FINISHTIME</td>
<td>Finish time</td>
</tr>
<tr>
<td>SYSDATE</td>
<td>Current date</td>
</tr>
<tr>
<td>SEID</td>
<td>ID of the job definition</td>
</tr>
<tr>
<td>TRIGGERWARNING</td>
<td>Text in the warning that activated this trigger</td>
</tr>
<tr>
<td>LAST_WARNING</td>
<td>Text in the last issued warning. If no current warning is present, this parameter is empty.</td>
</tr>
<tr>
<td>RERUNSEQ</td>
<td>The number of reruns until now</td>
</tr>
<tr>
<td>SCOPENAME</td>
<td>Name of the scope (jobserver) in which the job is running or last ran</td>
</tr>
</tbody>
</table>
3. The address space of the job and submitting parents, from bottom to top
4. The address space of the jobserver executing the job
5. The address space of the parent scopes of the jobserver executing the job, from bottom to top
6. The job definition’s parent folders, from bottom to top
7. The parent folders of the parent jobs, from bottom to top

If the configuration parameter ‘ParameterHandling’ for the server has been set to ‘strict’ (default), accessing variables that are not defined in the job definition will trigger an error message unless it is a system variable.

If the contents of a variable includes a reference to another parameter, this parameter is evaluated and replaced in the context of the defining job.

The different parameter types and their semantics are described below:

**IMPORT** Import-type parameters are used to hand over the data for a Job Scheduling Environment to another job. This type is almost like the parameter type, although import type parameters cannot be handed over like parameters when a job is submitted. Import-type parameters can have a default value, which is used if no value can be acquired from the scheduling environment.

**PARAMETER** Parameter-type parameters are used to hand over the data from a Job Scheduling Environment to another job. This type is almost like the import type, but parameter-type parameters can be handed over as parameters when a job is submitted. Parameter-type parameters can have a default value, which is used if no value can be acquired from the scheduling environment.

**REFERENCE** Reference-type parameters are normally used to hand over results from one job to another.

The fully qualified name of the job definition and the name of the referencing parameter are required to create a reference. The Submitted Entity with the closest match to the job definition of the reference is sought to resolve the reference. If this allocation cannot be made clearly enough, this triggers an error message. If a matching Submitted Entity could not be found, the default value (if defined) is returned.

**REFERENCE CHILD** Child Reference parameters are used to refer to the parameters of direct or indirect children. This can be useful for reporting purposes, for example. A Child Reference parameter is defined using a fully qualified job definition name together with the name of the parameter to be qualified. When resolving the parameter, the Submission Hierarchy is searched downwards instead of upwards as is the case with Reference Parameters. The behaviour for the resolution is otherwise identical to the resolution of Reference Parameters.

**REFERENCE RESOURCE** Resource Reference-type parameters are used to refer to parameters of allocated resources.
This parameter type requires the fully qualified name of a Named Resource together with an additional parameter name to specify the default reference. The prerequisite for using a Resource Reference parameter is that the resource is also requested. The value is determined in the context of the allocated resource.

**RESULT** Result-type parameters can acquire a value from the job (using the API). As long as this value has not been set, the optional default value is returned when the value is queried.

**CONSTANT** Constant-type parameters are parameters that have a value specified in the definition. This value can therefore not change during runtime.

**LOCAL** These variables are only visible from the perspective of the defining job.

**priority** The priority of a job determines the order in which jobs are executed. Values between 0 (high priority) and 100 (low priority) are permitted. The priority option only applies for jobs.

**profile** The profile defines the Exit State Profile that describes the valid Exit State of the Scheduling Entity. For a detailed description of the Exit State Profile, refer to the “create exit state profile” command on page 114.

**required** The required section defines the dependencies of other submitted entities in a Master Run which must be fulfilled until the Submitted Entity is capable of carrying on running.

Whether all the dependencies have to be fulfilled or just one of them is defined by the ‘dependency’ mode’.

Dependencies are defined in a comma-separated list of fully qualified names of Scheduling Entities (including folder path names).

Dependencies only apply between the Submitted Entities of the Master Run. Synchronizing Resources have to be used to synchronise the Submitted Entities from different Master Runs.

After the Submitted Entity instances of the Submitted Scheduling Entity hierarchy have been created, the system searches for the dependencies as follows: Beginning with the parent of the dependent Submitted Entity, all the children are searched for an instance of the Required Scheduling Entity whereby the branch with the dependent Submitted Entity is obviously ignored. If no instance is found, the search continues in the Submit Hierarchy Parents until precisely one instance has been found. If an instance can still not be found, the property ‘unresolved’ defines how this situation is handled by the system. If more than one Submitted Entity is found, the submit fails with an ‘ambiguous dependency resolution’ error.

During the execution of a Master Run, a Scheduling Entity can attain an ‘unreachable’ state because the dependencies can no longer be fulfilled. This can happen if a
Required Scheduling Entity reaches a Final State that is not entered in the list of required states for dependencies or by cancelling a Submitted Entity that is required by another Submitted Entity. These two cases are handled differently.

If the unreachable situation is caused by a Submitted Entity that finishes with an unsuitable Exit State, the system determines the Exit State Profile of the dependent Submitted Entity and sets the Exit State to the state that is marked as being ‘unreachable’ in the profile.

If none of the Profile States is marked as an unreachable state or the unreachable state was caused by a Submitted Entity being cancelled, the dependent Submitted Entity is set to the unreachable state, which can only be resolved by an operator ignoring the dependency or cancelling the dependent entity.

All the direct or indirect children of a job or batch inherit all the parent’s dependencies. This means that no child of a job or batch can exit the dependency wait state as long as the parent itself is in this state. Children of milestones do not inherit the dependencies from their parent.

The properties of the dependency definitions are described below:

**CONDITION** It is possible to stipulate a condition for a dependency. The dependency is only fulfilled if the evaluation of the condition returns the truth value “true”. If no condition is specified, the condition is always deemed to have been fulfilled.

**DEPENDENCY NAME** A name can be optionally specified for the dependency when defining a function. Children (both direct and indirect) can refer to the name in order to ignore this dependency.

**MODE** The mode property is only relevant if the required Scheduling Entity is a job with children. In this case, the Dependency Mode defines the time when the dependency is fulfilled.

The table below shows the possible values and their meanings.

<table>
<thead>
<tr>
<th>dependency mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>all_final</td>
<td>The required job and all its children must have reached a Final State.</td>
</tr>
<tr>
<td>job_final</td>
<td>Only the required job itself has to reach a Final State, the state of the children is irrelevant.</td>
</tr>
</tbody>
</table>

**STATE** The state property of a dependency defines a list of Final States that the required Scheduling Entity can achieve to fulfil the dependency.

Without this option, the dependency is fulfilled if the required Scheduling Entity reaches a Final State.

It is also possible to stipulate a condition for a state. If a condition has been specified, the dependency is only deemed to have been fulfilled if the condition is fulfilled as well. The syntactic rules for specifying conditions are the same as those that apply to triggers. For more details, refer to the create trigger statement on page 167. Several implicit definitions are also available as options:
• **default** — The dependency is fulfilled if the predecessor has reached one of the states that are defined in its profile as being a default dependency.

• **all reachable** — The dependency is fulfilled if the predecessor has reached one of the states that are not defined as being unreachable.

• **reachable** — The dependency is fulfilled if the predecessor has reached the state defined as being unreachable.

**UNRESOLVED** The unresolved property specifies how the system should handle a situation where no Submitted Entity instance could be found during a Submit Operation for a required Scheduling Entity.

The possible behavioural patterns are described in the table below:

<table>
<thead>
<tr>
<th>unresolved</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>error</td>
<td>The submit operation fails with an error message.</td>
</tr>
<tr>
<td>ignore</td>
<td>The dependency is tacitly ignored.</td>
</tr>
<tr>
<td>suspend</td>
<td>The dependency is ignored, but the dependent Submitted Entity is placed in a ‘suspended’ state and requires a user action to continue.</td>
</tr>
<tr>
<td>defer</td>
<td>This option promises that the predecessor will be dynamically submitted later.</td>
</tr>
<tr>
<td>defer ignore</td>
<td>This option expects that the predecessor will be dynamically submitted later. If this doesn’t happen, the dependency will be ignored.</td>
</tr>
</tbody>
</table>

**rerun program** If a rerun program command line has been defined for a job, this is executed instead of the run command line when the job is restarted after a failure.

For details about command line parsing, variants and the substitution parameter, refer to the “run program” option on page 148.

**resource** The resource section of a job definition defines resource requirements in addition to those requirements indirectly defined by the environment and footprint options.

If the same Named Resource as in the footprint is required here, the requirement in the Resource Section overwrites the requirement in the footprint.

Since environments only require Named Resources with the usage static and footprints only require Named Resources with the usage system, the Resource Section in a job definition is the only place where resource requirements for Named Resources with the usage synchronizing can be defined.

Resource requirements are defined by the fully qualified path name to a Named Resource defined with the following additional requirement options:
**AMOUNT** The amount option is only valid with requests for Named Resources of the type System or Synchronizing. The amount in a Resource Request expresses how many units of the Required Resource are allocated.

**EXPIRED** The expired option is only valid for Synchronizing Resources with a defined Resource State Profile. If the expired option is specified, the time to which the Resource State of the resource has been set cannot be less recent than the time given by the expire option. A negative Expire value means that a resource must be at least as old as given here. The Resource State can only be set by the old resource command (see page 82) or automatically when defining a Resource State Mapping which converts the Exit State and Resource State into a new Resource State. Even if, in such a case, the new Resource State is the same as the old Resource State, the Resource State is considered to have been set.

**LOCKMODE** The lockmode option in a resource requirement is only valid for Synchronizing Resources. Five possible lockmodes are defined:

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>Exclusive lock</td>
</tr>
<tr>
<td>S</td>
<td>Shared lock</td>
</tr>
<tr>
<td>SX</td>
<td>Shared exclusive lock</td>
</tr>
<tr>
<td>SC</td>
<td>Shared compatible lock</td>
</tr>
<tr>
<td>N</td>
<td>Nolock</td>
</tr>
</tbody>
</table>

The important aspect here is the compatibility matrix:

```
   X S SX SC N
X  N N N N Y
S  N Y N Y Y
SX N N Y Y Y
SC N Y Y Y Y
N  Y Y Y Y Y
```

The purpose of the exclusive lock is to have exclusive access to the resource to be able to set the Resource State and possibly parameter values. A common example of where the exclusive lock is used is when reloading a database table.

The purpose of the shared lock is to allow other users to use the resource in the same way while preventing them from making any changes. The most frequent scenario for using shared locks is for a large-scale ongoing reading of a database table. Other read processes can simply be tolerated, but no write transactions are allowed.

The purpose of the shared exclusive lock is to have a second shared lock which is not compatible with the normal shared lock. If we use the normal use shared lock for large read transactions, then we use the shared exclusive lock for small write
transactions. Small write transactions can easily run in parallel, but if they create a large read transaction when doing so, they will almost certainly cause a "snapshot too old" or other similar problems.

The purpose of the shared compatible lock is to have a shared lock that is compatible with both the shared and exclusive locks. This lock type is intended for short read transactions which do not conflict with small write transactions or large read transactions. Small read transactions obviously don’t conflict with other small read transactions. Running small read and large write transactions in parallel may cause problems.

The purpose of the nolock is to ensure that the resource exists and that all the other properties of the resource cover requirements. The resource is not locked and anything can happen, including state changes.

STATE The state option is only valid for Synchronizing Resources with a Resource State Profile. It is used to specify valid Resource States for this job. A resource can only be allocated if it is in one of the required states.

STATE MAPPING The state mapping option is only valid for Synchronizing Resources that specify a Resource State Profile and are requested with an "exclusive" lock-mode. The mapping defines a function that maps the combinations of Exit States and Resource States in a new Resource State. For more detailed information about resource state mappings, refer to the create resource state mapping statement on page 158.

KEEP The keep option in a Resource Request defines the time when the resource is released. The keep option is valid for both System and Synchronizing Resources. There are three possible values. Their meanings are explained in the table below:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>nokeep</td>
<td>The resource is released at the end of the job. This is the default behaviour.</td>
</tr>
<tr>
<td>keep</td>
<td>The resource is released as soon as the job has reached the Final State.</td>
</tr>
<tr>
<td>keep final</td>
<td>The resource is released when the job and all its children are final.</td>
</tr>
</tbody>
</table>

STICKY The sticky option is only valid for Synchronizing Resources. If sticky is specified, the resource is allocated by the master batch (this is called a MASTER RESERVATION) for as long as other jobs in the batch that require the sticky resource. The amount and lockmode for the Master Reservation are derived from all the sticky requirements of all the children. The amount is the maximum needed by any job.

The lockmode is exclusive as long as at least two jobs exist which request the resource with a lockmode other than nolock. An exception is the combination of Shared and Shared Compatible lock requests. This combination results in lockmode Shared.
An attempt is made to fulfil all the requirements from the Master Reservation.
A name can be optionally assigned for the sticky allocation. As a basic principle,
only those requests with the same name are taken into account for the previously
described method. That’s why a master batch can have several MASTER_RESER-
VATIONS at the same time. Several separate critical regions can be realised within
a sequence with the aid of the names.
A parent job or batch can be specified in addition to, or even instead of, the name.
The corresponding instance of the parent is then determined at runtime from the
submission hierarchy. The sticky request is only valid from the parent downwards.
In principle, this can be interpreted as if the parent’s Id represents a part of the
name of the sticky request. This mechanism allows separate critical regions to be
easily implemented in dynamically submitted sub-workflows.

**runtime**  The runtime option is used to define the estimated runtime of a job.
This time can be valued when activating triggers.

**run program**  The run program command line is mandatory for jobs because it
specifies the command that is to be executed for this job.
The command line is separated by whitespace characters in a command and a list
of arguments. The first element in the command line is regarded as the name of
the executable program that is to be run, and the rest are the parameters for the
program.
Whether the jobservier uses the PATH environment variable when searching for the
executable file is a characteristic of the jobservier.
System and job parameters can be addressed with $ Notation.
Quoting can be used to forward whitespace characters and $ characters as part of
the command line. The quoting complies with Unix Bourne shell rules. This means
that double quotes prevent whitespace characters from being interpreted as sepa-
rators. Single quotes also prevent variables from being resolved. Backticks can be
used for quoting. The parts of the command line that have been quoted in back-
ticks are regarded as having been single quoted, but the backticks remain a part of
the argument. Other quotes are removed.

Example:

The run command line ‘sh -c ’'example.sh ${JOBID} \$HOME'' 'SHELL''
will execute the program ‘sh’ with the parameters’-c’,’example.sh 4711 \$HOME’
and ‘SHELL’ (assuming that the Submitted Entity has the ID 4711).
If the executable program (the first element of the command line) is a valid integer,
the command line is not run by the jobservier. Instead, the job is treated as if it
had completed itself with the integer as the Exit Code. Dummy jobs with ‘true’
or ‘false’ as the program can now be implemented as ‘0’ instead of ‘true’ or ‘1’
instead of ‘false’ and are therefore processed much more efficiently and quickly by the system.
Should it really be necessary to run an executable with a number as the name, this can be achieved by using a path prefix (‘./42’ instead of ‘42’).

**suspend**  The suspend option defines whether a Submitted Entity is suspended at the submit time.
If the suspend option is specified, the resume clause can be optionally used. This can then trigger an automatic resume at or after the specified time.
If the resume time is specified by the incomplete date format (see also page 20), the resume takes place at the first suitable time after the submit time.
If a submit takes place at 16:00, for example, and T17:30 is entered as the resume time, the resume will take place on the same day at 17:30. But if T15:55 is specified as the resume time, the job will have to wait until the next day at 15:55.

**timeout**  The timeout clause of a job definition defines the maximum time for which the job waits until its resource requirements are fulfilled.
When the timeout condition is reached, the job gets the Exit State specified in the timeout clause. This Exit State must be an element of the Exit State Profile.
If no timeout option is given, the job will wait until all the requirements have been fulfilled.

**type**  The type option specifies the Scheduling Entity type that is being created or modified.

**workdir**  The workdir of a Scheduling Entity-type job defines the directory where the run, rerun or kill program is executed.

**Output**

This statement returns a confirmation of a successful operation.
create named resource

Purpose
Purpose
The purpose of the create named resource statement is to define a class of resources.

Syntax
Syntax
The syntax for the create named resource statement is

create [ or alter ] named resource identifier [ . identifier ]
with WITHITEM { , WITHITEM }

WITHITEM:
   group = groupname [ cascade ]
   inherit grant = none
   inherit grant = ( PRIVILEGE { , PRIVILEGE } )
   parameter = none
   parameter = ( PARAMETER { , PARAMETER } )
   state profile = < none | rspname >
   usage = RESOURCE_USAGE

PRIVILEGE:
   approve
   cancel
   clear warning
   clone
   create content
   drop
   edit [ parameter ]
   enable
   execute
   ignore resource
   ignore dependency
   kill
   monitor
   operate
   priority
   rerun
   resource
   set job status
   set state
   submit
| suspend  |
|         |
| use     |
| view    |

**PARAMETER:**

- `parametername constant` = `string`
- `parametername local constant` = `string`
- `parametername parameter` = `string`

**RESOURCE_USAGE:**

- `category`
- `static`
- `synchronizing`
- `system`

**Description**

The *create named resource* statement is used to define classes of resources. These classes define the name, the usage type and optionally the utilised Resource State Profile as well as the parameters.

**group** The group option is used to set the owner group to the specified value. The user must belong to this group unless he belongs to the ADMIN privileged group. In this case, any group can be specified.

**parameter** It may be useful to use its parameters in conjunction with allocating resources. For example, a resource like `RESOURCE_TEMP_SPACE` could have a parameter called `LOCATION`. This would allow a job to use a resource and allocate temporary storage space somewhere dependent upon the current instance of the Named Resource.

There are three types of parameters in a resource context:

<table>
<thead>
<tr>
<th>Typ</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>constant</td>
<td>This parameter type defines the value that is constant for all resources.</td>
</tr>
<tr>
<td>local constant</td>
<td>This parameter type defines a non-variable parameter whose value can deviate between instances of the same Named Resource.</td>
</tr>
</tbody>
</table>

*Continues on next page*
Type | Meaning
--- | ---
parameter | The value of such a parameter can be changed by jobs that have exclusively locked this resource.

A State Resource Profile can be specified in the case of Synchronizing Resources. This allows jobs to request the resource in a particular state. Resource State changes can be used to activate triggers.

The usage of the Named Resource can be one of the following:

Usage | Meaning
--- | ---
category | Categories behave like folders and can be used to arrange the Named Resources in a clearly organised hierarchy.
static | Static resources are resources which, if requested, must be present in the scope in which the job is running but which cannot be used up. Possible examples of Static Resources are a particular operating system, shared libraries for DBMS access operations or the presence of a C compiler.
system | System Resources are resources that can be counted. Possible examples are the number of processes, the capacity of the temporary memory or the availability of (a number of) tape drives.
synchronizing | Synchronizing Resources are the most complex resources and are used to synchronise multiple access operations. One possible example is a database table. Multiple access operations may be tolerated or not depending on the type of access (large read transactions, large write transactions, multiple small write transactions, etc.).
pool | pool-type Named Resources are used to create so-called Resource Pools. These pools allow the distribution of amounts for System Resources to be regulated centrally and flexibly.

When creating a Named Resource, the factor by which the specified amounts in a resource request are multiplied can be specified. The default factor is 1. This factor can be overwritten for each instance of this Named Resource (i.e. for each resource).
**inherit grant**  The inherit grants clause allows you to define which privileges are to be inherited through the hierarchy. If this clause is not specified, all privileges are inherited by default.

**Output**

This statement returns a confirmation of a successful operation.
create resource

Purpose

The purpose of the create resource statement is to create an instance of a named resource within a scope, folder or job definition.

Syntax

The syntax for the create resource statement is

```
create [ or alter | resource identifier [. identifier] in < serverpath | folderpath > [ with WITHITEM {, WITHITEM} ]
```

WITHITEM:

- `amount = < infinite | integer >`
- `< online | offline >`
- `parameter = none`
- `parameter = ( PARAMETER {, PARAMETER} )`
- `requestable amount = < infinite | integer >`
- `state = statename`
- `touch [ = datetime ]`
- `group = groupname`

PARAMETER:

- `parametername = < string | default >`

Description

The create resource statement is used to instantiate Named Resources within scopes, folders or job definitions. In the latter case, only a template is created which is materialised as soon as the job is submitted and automatically destroyed as soon as the Master Run is Final or Cancelled.

If the or alter option is specified, an existing resource is changed; otherwise, it is considered to be an error if the resource already exists.

**amount**  The amount clause defines the Available Amount for this resource. The amount option is not specified in the case of static resources.

**base multiplier**  The base multiplier is only relevant if the Resource Tracing is being used. The base multiplier determines the multiplication factor for trace base. If the trace base is designated as being $B$ and the trace multiplier as being $M$,
the mean allocation is determined for the periods $B \ast M^0$, $B \ast M^1$ and $B \ast M^2$. The default value is 600 (10 minutes) so that the values for $B$, $10B$ and $100B$ (in minutes) are determined.

**factor** A Resource Factor has been implemented to allow resource requirements for jobs to be adjusted externally. This can be set in both the Named Resource and individually in the resource. Whether a job can be allocated a particular resource is determined by comparing the original request with the Requestable Amount. However, the actual allocation is taken from

$$\text{ceil}(\text{Requirement} \ast \text{Factor})$$

**group** The group option is used to set the owner group to the specified value. The user must belong to this group unless he belongs to the ADMIN privileged group. In this case, any group can be specified.

**online** The online clause defines whether the resource is online or offline. A resource is not available if it is offline. This means that a job that requires this resource cannot run within this scope. But since the resource can be set to online, the job will wait and will not be set to an error state. This also applies to static resources.

**parameter** The parameter clause is used to set the values of the parameters that have been defined for the Named Resource. Parameters that are declared as a constant at Named Resource Level are not permitted here. All the other parameters can be specified, although this is not mandatory. If a parameter or a default value for this parameter has not been specified at Named Resource Level, the resolution returns an empty string. If parameter name = default is specified when changing the resource, the parameter takes on the default value analogue to the Named Resource. If the parameter is changed on the Named Resource level, this is visible on the Resource level for all the parameters that have been set to the default value. A number of system variables are always defined for each resource. These are set by the system and are available to jobs which allocate the resource for read access using “RESSOURCEREFERENCES”. These system variables are:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATE</td>
<td>The Resource State of a “synchronizing” resource with a state model</td>
</tr>
</tbody>
</table>

*Continued on next page*
Continued from previous page

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMOUNT</td>
<td>The total amount of available resources</td>
</tr>
<tr>
<td>FREE_AMOUNT</td>
<td>The total amount of available free resources</td>
</tr>
<tr>
<td>REQUESTABLE_AMOUNT</td>
<td>The maximum amount that can be allocated by a job</td>
</tr>
<tr>
<td>REQUESTED_AMOUNT</td>
<td>The amount requested by the job</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>The touch timestamp of a “synchronizing” resource with a state model</td>
</tr>
</tbody>
</table>

Table 6.4.: List of System Variables

requestable amount  The requestable amount clause defines the amount of this resource that can be requested by a single job. This does not have to be the same as the available amount. If the requested amount is smaller than the amount, it is certain that a job cannot allocate all the available resources. If the Requestable Amount is greater than the amount, jobs can request more than the available amount without triggering a “cannot run in any scope” error.

If the Requestable Amount is not specified, it is the same as the amount.
The requestable amount option is not specified in the case of static resources.

state  The state clause defines the resource’s state.
This option is only valid for Synchronizing Resources with a Resource State Profile.

tag  To facilitate evaluating the trace table, resources and pools can now be marked with a tag. This tag should be unique within the resources and pools (i.e. the use of a tag for both a resource and a pool is prohibited as well).

touch  The touch clause defines the last time when the status of the resource (of a job) was changed. This timestamp is not set if a Resource State has been set manually.
This option is only valid for Synchronizing Resources with a Resource State Profile.

trace base  Tracing is deactivated if the trace base is none. Otherwise it is the basis for the valuation period.

trace interval  The trace interval is the minimum time in seconds between when Trace Records are written. Tracing is deactivated if the trace interval is none.

Output

Output  This statement returns a confirmation of a successful operation.
create resource state definition

Purpose

The purpose of the create resource state definition statement is to create a symbolic name for a state of a resource.

Syntax

The syntax for the create resource state definition statement is

create [ or alter ] resource state definition statename

Description

The create resource state definition statement is used to define a symbolic name for a Resource State. The optional keyword or alter is used to prevent error messages from being triggered and the current transaction from being aborted if a Resource State Definition already exists. If it is not specified, the existence of a Resource State Definition with the specified name will trigger an error.

Output

This statement returns a confirmation of a successful operation.

Example

A number of names for Resource States are defined in these examples.

create resource state definition empty;
create resource state definition valid;
create resource state definition invalid;
create resource state definition stage1;
create resource state definition stage2;
create resource state definition stage3;
create resource state mapping

Purpose

The purpose of the *create resource state mapping* statement is to define a mapping between the exit states of a job and the resulting resource state of a resource.

Syntax

The syntax for the *create resource state mapping* statement is

\[
\text{create [ or alter ] resource state mapping mappingname with map = ( WITHITEM {}, WITHITEM )}
\]

\[
\text{WITHITEM: stasename maps < stasename | any > to stasename}
\]

Description

The *create resource state mapping* statement defines the mapping of Exit States in combination with Resource States to create new Resource States. The first state name must be an Exit State. The second and third state have to each be a Resource State. If a job terminates with the given Exit State, the resource state is set to the new state if the current state matches the first named state. If *any* is specified as the initial state, any Resource State is mapped to the new one. If both a specific mapping and a general mapping have been specified, the specific mapping has the highest priority.

Output

This statement returns a confirmation of a successful operation.

Example

The example shows a mapping that propagates the state of the resource to the next "PHASE" each time the mapping is applied. Also \( \text{PHASE1} \rightarrow \text{PHASE2} \rightarrow \text{PHASE3} \rightarrow \text{PHASE1} \rightarrow \ldots \)

create or alter resource state mapping 'PHASE_MODEL'
with map = {
    'SUCCESS' maps 'PHASE1' to 'PHASE2',
    'SUCCESS' maps 'PHASE2' to 'PHASE3',
    'SUCCESS' maps 'PHASE3' to 'PHASE1'
};
create resource state profile

Purpose

The purpose of the create resource state profile statement is to create a set of valid resource states.

Syntax

The syntax for the create resource state profile statement is

```
create [ or alter ] resource state profile profilename
with WITHITEM {, WITHITEM}
```

WITHITEM:
```
  initial state = statename
  | state = ( statename {, statename} )
```

Description

The create resource state profile statement is used to define a set of valid Resource States for a (Named) Resource.

state  The state clause defines which Resource State Definitions are valid within this profile.

initial state  The initial state clause determines the initial state of a resource with this profile. The initial state does not have to be present in the list of states from the state clause. This allows a resource to be created without it immediately playing an active role in the system.

Output

This statement returns a confirmation of a successful operation.

Example

In this example, the Exit State is to become invalid if it is empty.

```
create resource state profile example1
with
  state = (empty);
```
create schedule

Purpose

The purpose of the create schedule statement is to create an active container for scheduled events.

Syntax

The syntax for the create schedule statement is

```sql
create [ or alter ] schedule schedulepath | with WITHITEM {, WITHITEM}
```

WITHITEM:

- `< active | inactive >`
- `inherit grant = none`
- `inherit grant = ( PRIVILEGE {, PRIVILEGE} )`
- `interval = < none | intervalname >`
- `time zone = string`
- `group = groupname`

PRIVILEGE:

- `approve`
- `cancel`
- `clear warning`
- `clone`
- `create content`
- `drop`
- `edit [ parameter ]`
- `enable`
- `execute`
- `ignore resource`
- `ignore dependency`
- `kill`
- `monitor`
- `operate`
- `priority`
- `rerun`
- `resource`
- `set job status`
- `set state`
- `submit`
- `suspend`
Description

With the `create schedule` statement, complex schedules can be created for jobs and batches using simple definitions.

**active** The active option causes the schedule to always trigger events in step with the specified interval (assuming that any events have been defined). The inactive option, on the other hand, prevents the schedule from triggering events in step with the specified interval. A hierarchical arrangement of schedules thus allows exception periods (such as downtimes) to be defined, for example.

**Group** The group option is used to set the owner group to the specified value. The user must belong to this group unless he belongs to the ADMIN privileged group. In this case, any group can be specified.

**Interval** The given interval acts as a ‘clock’ for the schedule. If an event is linked to the schedule, this event is triggered in rhythm with the interval.

**inherit grant** The inherit grants clause allows you to define which privileges are to be inherited through the hierarchy. If this clause is not specified, all privileges are inherited by default.

Output

This statement returns a confirmation of a successful operation.
create scheduled event

Purpose

The purpose of the create scheduled event is to define a connection between a schedule and an event.

Syntax

The syntax for the create scheduled event statement is

```sql
create [ or alter ] scheduled event schedulepath . eventname [ with
WITHITEM { , WITHITEM} ]
```

WITHITEM:

- `<active | inactive>`
- `backlog handling = <last | all | none>`
- `calendar = <active | inactive>`
- `horizon = <none | integer>`
- `suspend limit = <default | period>`
- `group = groupname`

Description

Scheduled Events represent a link between events (what is to be done) and schedules (when should it be done).

**backlog handling**  The backlog handling function indicates how events that happened during a server downtime are to be handled. The three possible actions are shown in the table below:

<table>
<thead>
<tr>
<th>Action</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>last</td>
<td>Only the last event is triggered</td>
</tr>
<tr>
<td>all</td>
<td>All the events that happened in the meantime are triggered</td>
</tr>
<tr>
<td>none</td>
<td>None of the events that happened in the meantime are triggered</td>
</tr>
</tbody>
</table>

**Group**  The group option is used to set the owner group to the specified value. The user must belong to this group unless he belongs to the ADMIN privileged group. In this case, any group can be specified.

**active**  Scheduled Events can be marked as being active or inactive. If they are marked as being active, events are triggered. Correspondingly, events are not triggered if the Scheduled Event is marked as being inactive. This option can be used to deactivate Scheduled Events without the definition being lost.
**suspend limit**  The suspend limit defines the length of the delay before a job belonging to an event is automatically submitted with the suspend option. A delay can arise if, for whatever reason, the Scheduling Server goes offline. After the server has booted up again, events that have happened during the downtime are triggered dependent upon the **backlog handling** option. This means that the execution time is later than the scheduled execution time.

**Output**

This statement returns a confirmation of a successful operation.
create scope

Purpose

The purpose of the create scope statement is to create a scope within the scope hierarchy.

Syntax

The syntax for the create scope statement is

```
create [ or alter ] < scope serverpath | jobserver serverpath > [ with
JS_WITHITEM {, JS_WITHITEM} ]
```

```
JS_WITHITEM:
  | config = none
  | config = ( CONFIGITEM {, CONFIGITEM} )
  | < enable | disable >
  | error text = < none | string >
  | group = groupname [ cascade ]
  | inherit grant = none
  | inherit grant = ( PRIVILEGE {, PRIVILEGE} )
  | node = nodename
  | parameter = none
  | parameter = ( PARAMETERITEM {, PARAMETERITEM} )
  | password = string
  | rawpassword = string [ salt = string ]
```

```
CONFIGITEM:
  | parametername = none
  | parametername = ( PARAMETERSPEC {, PARAMETERSPEC} )
  | parametername = < string | number >
```

```
PRIVILEGE:
  | approve
  | cancel
  | clear warning
  | clone
  | create content
  | drop
  | edit [ parameter ]
```
enable
execute
ignore resource
ignore dependency
kill
monitor
operate
priority
rerun
resource
set job status
set state
submit
suspend
use
view

PARAMETERITEM:

\[
\text{parametername} = \text{dynamic} \\
\text{parametername} = < \text{string} | \text{number} >
\]

PARAMETERSPEC:

\[
\text{parametername} = < \text{string} | \text{number} >
\]

Description

The create scope command is used to define a scope or jobserver and its properties.

**Config** The config option allows a jobserver to be configured using key/value pairs. The configuration is inherited downwards so that general configuration parameters can be set at scope level. This means that they are valid for all the jobservers created below this level provided that the parameters at the lower level are not overwritten. When the jobserver logs onto the scheduling server, the server is given the list with the configuration parameters.

**Enable** The enable option allows the jobserver to connect to the repository server. This option is not valid for scopes and is tacitly ignored if it is specified.

**Disable** The disable option forbids the jobserver from connecting to the repository server. This option is not valid for scopes and is tacitly ignored if it is specified.
**Group** The group option is used to set the owner group to the specified value. The user must belong to this group unless he belongs to the ADMIN privileged group. In this case, any group can be specified.

**Node** The node specifies the computer on which the jobserver is running. This field has a purely documentary character.

**Parameter** Parameters can be used for communication and data transfer purposes between jobs. They are available for use with the jobs and programs that are executed within the jobs. The parameters of scopes and jobservers can be used to specify information about a job’s runtime environment. A Dynamic Parameter is fulfilled after the jobserver has logged on from within its own process environment. If the process environment of a jobserver is changed, attention has to be paid to this Dynamic Variable because otherwise race conditions can easily arise.

**Inherit grant** The inherit grants clause allows you to define which privileges are to be inherited through the hierarchy. If this clause is not specified, all privileges are inherited by default.

**Password** The password option is used to set the password for the jobserver. This option is not valid for scopes and is tacitly ignored if it is specified.

**Output**

*Output* This statement returns a confirmation of a successful operation.
**create trigger**

**Purpose**

The purpose of the *create trigger* statement is to create an object which submits a job dynamically when a certain condition is met.

**Syntax**

The syntax for the *create trigger* statement is

```sql
create [ or alter ] trigger triggername on CT_OBJECT [ < noinverse | inverse > ] with WITHITEM {, WITHITEM}
```

**CT_OBJECT:**

```sql
job definition folderpath
| named resource identifier [. identifier]
| object monitor objecttypename
| resource identifier [. identifier] in < folderpath | serverpath >
```

**WITHITEM:**

```sql
< active | inactive >
| check = period
| condition = < none | string >
| < nowarn | warn >
| event = ( CT_EVENT {, CT_EVENT} )
| group event
| limit state = < none | statename >
| main none
| main folderpath
| < nomaster | master >
| parameter = none
| parameter = ( identifier = expression {, identifier = expression} )
| parent none
| parent folderpath
| rerun
| < noresume | resume in period | resume at datetime >
| single event
| state = none
| state = ( < statename {, statename} | CT_RSCSTATUSITEM {, CT_RSCSTATUSITEM} > )
```
submit after folderpath
submit folderpath
submitcount = integer
< nosuspend | suspend >
[ type = ] CT_TRIGGERTYPE
[ group = ] groupname

CT_EVENT:
< create | change | delete >

CT_RSCSTATUSITEM:
< statename any | statename statename | any statename >

CT_TRIGGERTYPE:
  after final
  before final
  finish child
  immediate local
  immediate merge
  until final
  until finished
  warning

Description

The create trigger statement is used to create an object that waits for a certain event to happen following which a job or batch is submitted in response to this event.
If the or alter option is specified, an existing trigger is changed; otherwise, it is considered to be an error if the trigger already exists.

Triggers can be defined for Scheduling Entities or Synchronizing (Named) Resources. In the latter case, the trigger is evaluated every time the state of the resource or instance of the Named Resource changes. Resource Triggers are always so-called Master Triggers, i.e. they submit a new Master Batch or Master Job. Although triggers in Scheduling Entities can submit Master Batches, by default they submit new children. These children must be defined as (dynamic) children of the triggering Scheduling Entities.

active The active option enables the trigger to be activated or deactivated. This means that the trigger action can be temporarily suppressed without having to delete the trigger.
**check**  The check option is only valid for **until final** and **until finished** triggers. It defines the time intervals between two evaluations of the conditions. The condition is always evaluated when a job finishes regardless of the defined intervals.

**condition**  The condition option can be specified to define an additional condition which has to be checked before the trigger is activated. This condition is a Boolean expression and the trigger is activated if this condition returns true.

**BOOLEAN OPERATORS**  Since this condition is a Boolean expression, Boolean operators can be used to create multiple complex conditions. This Boolean operators are:

- **not** (unary negation operator)
- **and**
- **or**

The usual priority rules apply. The ‘not’ operator takes priority over the ‘and’ operator, which in turn takes priority over the ‘or’ operator. Parentheses can be used to force a valuation sequence.

It is also permitted to use the Boolean constants **false** and **true**.

**COMPARISON OPERATORS**  Comparisons can be used as part of Boolean expressions. The following comparison operators are defined.

- **==** (equal to)
- **>=** (greater than or equal to)
- **<=** (less than or equal to)
- **! =** (not equal to)
- **>** (greater than)
- **<** (less than)
- **=∼** (pattern matches)
- **! ∼** (pattern does not match)

All comparison operators can work with strings. With character strings, the ‘larger than’ and ‘less than’ operators use the ASCII value of the characters. The matching operators do not work with numbers.

For a full description of the regular expressions that can be used by the match operators, please refer to the original Java documentation for java.util.regexp.

**NUMERIC OPERATORS**  Since it cannot be guaranteed that decisions cannot only be made by comparing two values, the use of (numeric) operators is also permitted. The valid operators are:
• + (unary operator)
• − (unary negation operator)
• * (multiplication operator)
• / (division operator)
• % (modulo Operator)
• + (binary addition operator)
• − (binary subtraction operator)

**Literals and Variables** Literals are numbers (integers and floating point numbers) or character strings. Strings are delimited using double quotes ("."). It is possible to use variables that are resolved within the context of the triggering job or resource. Variables are addressed by prefixing their name with a dollar sign ($). When a variable is resolved, it is initially assumed that it is a trigger variable. If this is not the case, it is interpreted as a job variable. This kind of resolution is often, but unfortunately not always, correct. The prefix job., trigger. or resource., as well as in the context of dependencies, dependent. and required., can be used to explicitly specify which object will initiate a search for the variable. Variables are usually created in uppercase. This can be prevented by quoting the name. However, the name is converted back to uppercase when addressing the variables in conditions. To avoid this, the name and prefix (where applicable) have to be written in braces.

The operands are interpreted as character strings or numbers depending upon the operator and the first operand. Multiplication, division, modulo and subtraction operations, as well as unary processes, are only defined for numeric values. The addition operator in a character string context causes the operands to be strung together.

**Functions** Not everything can be simply expressed using (numeric) expressions, and so some additional functions have been added. The following functions are defined at this time:

• `abs(expression)` – the absolute value of the expression is returned
• `int(expression)` – the integer value of the expression is returned
• `lowercase(expression)` – the result of the expression is converted to lowercase and returned
• `round(expression)` – the expression is rounded and returned
• `str(expression)` – the expression is returned as a character string
• **substr(source, from [, until])** – returns part of the character string source beginning at the position `from` up to the end of the string or, if `until` is specified, up to the position `until`

• **str(expression)** – the expression is returned without a space at the end

• **uppercase(expression)** – the result of the expression is converted to uppercase and returned

Functions can be nested in one another without any restrictions.

**Examples** To clarify this, here are some statements that specify the conditions. Since conditions are not just found in trigger definitions, some other examples are given here as well. However, the syntax is always the same.

The first example shows a trigger that is activated when the job state changes to **WARNING** or **FAILURE** after it has already processed some rows ($\text{NUM\_ROWS} > 0$).

```sql
CREATE OR ALTER TRIGGER ON_FAILURE
ON JOB DEFINITION SYSTEM.EXAMPLES.E0100_TRIGGER.TRIGGER
WITH
  STATES = (FAILURE, WARNING),
  SUBMIT SYSTEM.EXAMPLES.E0100_TRIGGER.ON_FAILURE,
  IMMEDIATE MERGE,
  ACTIVE,
  NOMASTER,
  SUBMITCOUNT = 3,
  NOWARN,
  NOSUSPEND,
  CONDITION = '$\text{NUM\_ROWS} > 0$';
```

The second example shows an environment that requires the value of the resource variable **AVAILABLE** to begin with a **T** (such as TRUE, True, true or Tricky).

```sql
CREATE ENVIRONMENT SERVER@LOCALHOST
WITH RESOURCE = (
  RESOURCE.EXAMPLES.STATIC.NODE.LOCALHOST
  CONDITION = '$\text{RESOURCE\_AVAILABLE} =~ \^[tT].*$',
  RESOURCE.EXAMPLES.STATIC.USER.SERVER
);
```

The third example shows the same as the second one, except that here the parameter name is defined as being mixed case.

```sql
CREATE ENVIRONMENT SERVER@LOCALHOST
WITH RESOURCE = (
  RESOURCE.EXAMPLES.STATIC.NODE.LOCALHOST
  CONDITION = '${\text{RESOURCE\_Available}} =~ \^[tT].*$',
  RESOURCE.EXAMPLES.STATIC.USER.SERVER
);
```
**event**  The event option is only relevant for Object Monitor Triggers. It specifies for which types of events the trigger should be activated.

**group**  The group option is used to set the owner group to the specified value. The user must belong to this group unless he belongs to the ADMIN privileged group. In this case, any group can be specified.

**main**  The main option is only relevant for Object Monitor Triggers. If the main option is specified, the specified job or batch is submitted when the trigger fires. The actual trigger job must be defined as a child of the main job, unless the parent option is specified. For each Object Instance that has been changed according to the trigger specification (newly created, modified or deleted), an instance of the trigger job is added as a child of the main job. If the master option isn’t specified, the main job must be defined as a (dynamic) child of the Watcher job. If the master option is used, the main job must be master submittable.

**master**  The main option is only relevant for Object Monitor Triggers. If the main option is specified, the specified job or batch is submitted when the trigger fires. The actual trigger job must be defined as a child of the main job, unless the parent option is specified. For each Object Instance that has been changed according to the trigger specification (newly created, modified or deleted), an instance of the trigger job is added as a child of the main job. If the master option isn’t specified, the main job must be defined as a (dynamic) child of the Watcher job. If the master option is used, the main job must be master submittable.

**parameter**  The parameter option is used to specify parameters for the job that is to be triggered.

The expressions are valuated in the context of the triggering object. When the triggered job is submitted, the results are then handed over as the value for the specified parameter.

The syntax of the expressions corresponds to that of the conditions. Not only Boolean expressions, but also numeric or string-manipulating expressions are naturally allowed as well.

The operands are interpreted numerically or as strings dependent upon the operator. In case of doubt, the implicit data type of the first operand is definitive.

Some examples of expressions are given below to illustrate this. Here, we assume that the triggering job has defined some parameters:

\[
\begin{align*}
&A = "5" \\
&B = "10" \\
&C = "hello" \\
&D = "world"
\end{align*}
\]

The following equations apply with these parameters (i.e. as a Condition they would be valuated as being True):

\[
\begin{align*}
172
\end{align*}
\]
$A + \$B == 15
"" + $A + \$B == "510"
$A + "0" + \$B == 15
$C + " " + \$D == "hello world"
$A + \$C == "5hello"
\text{int}("" + \$A + \$B) * 2 == 1020
$C + (\$A + \$B) == "hello15"

Errors deliver expressions such as
\begin{align*}
C * \$A \\
C - \$D \\
\$B / (\$A - 5)
\end{align*}

The first two expressions are wrong because $C cannot be interpreted as a numeric value. In the last expression, an attempt is being made to divide by 0.
If the valuation of an expression runs into an error, the triggering also fails.

\begin{itemize}
\item \textbf{parent} \hspace{1em} The parent option is only relevant for Object Monitor Triggers. It can also only be specified in combination with the main option.
If it is specified, a search is run for the corresponding job (or batch) within the tree submitted using the main job is sought and the trigger jobs are appended below the parent.

\item \textbf{rerun} \hspace{1em} The rerun option can only react to restartable states and initiate an automatic rerun. In many cases, it will be practicable to also specify the suspend/resume options to allow a certain period of time between the resumes.
Either the submit option or the rerun option have to be specified.

\item \textbf{resume} \hspace{1em} The resume option can be used together with the suspend option to cause a delayed execution. There are two ways to do this. A delay can be achieved by specifying either the number of time units for the delay the time when the job or batch is to be activated.
If an incomplete time is defined, such as T16:00, the time for the trigger activation is taken as the reference time.

\item \textbf{state} \hspace{1em} The state option is valid for all triggers apart from until final and until finished triggers. A list of Exit States can be specified for triggers that act on jobs. When the job in which the trigger is defined reaches an Exit State that is listed in the Trigger Definition, this activates the trigger (unless a condition has been specified that is valuated as false).
A list of state changes can be specified in the case of a trigger that acts on a (Named) resource. This allows each state change to be explicitly addressed. It is possible to activate a trigger when a state is exited by using the keyword any on the right. It is always possible to activate a trigger on reaching a certain state by specifying any on the left. The state option is omitted to activate a trigger after every state change.
\end{itemize}
submit  The submit option defines which job or batch is submitted when the trigger is activated. Either the submit option or the rerun option have to be specified.

submitcount  The submitcount option is only permitted for triggers that act on jobs. It defines the number of times that a trigger can be activated. If this option is not specified, a submitcount of 1 is used. If a submitcount of 0 is specified, the submitcount is set to the server parameter TriggerSoftLimit (the default value for this is 50). In the case of a rerun trigger, however, a submitcount of 0 means that there is no limit to the number of restart attempts. If a submitcount greater than the TriggerSoftLimit is specified, the submitcount is restricted to the server parameter TriggerHardLimit (the default value for this is 100). This is done to avoid endless loops. The TriggerHardLimit can be set in the server configuration to $2^{31} - 1$ in order to virtually eliminate the restriction above.

suspend  The suspend option is used to submit the job or batch in a suspend state. This option is valid for all trigger types.

type  There are several types of triggers on jobs. The most important difference between them is the time at which they are checked. The table below shows a list of all the types with a brief description of their behaviour. It must be emphasised that the type option is not valid for (named) resource triggers.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Check time</td>
</tr>
<tr>
<td>after final</td>
<td>Only after a final state is reached is a check run to establish whether the defined trigger has to be activated. If the trigger is not a Master Trigger, the newly submitted job will have the same parent as the triggering job. A special situation arises if the triggering job triggers its own submit. In this case, the newly submitted job replaces the triggering job. Since this exchange takes place before the dependency was checked, all the dependent jobs wait until the newly submitted job is final.</td>
</tr>
</tbody>
</table>

Continues on next page
### Field Description

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>before final</strong></td>
<td>Immediately before a final state is reached, a check is run to establish whether the defined trigger is to be activated. This is the last opportunity to submit new children. If this is done, the job or batch will not reach a Final State at this time.</td>
</tr>
<tr>
<td><strong>finish child</strong></td>
<td>A finish child trigger checks whether it is to be activated every time when a direct or indirect child finishes.</td>
</tr>
<tr>
<td><strong>immediate local</strong></td>
<td>The immediate local trigger local checks whether it has to be activated when a job is terminated. Only the Exit State of the job is taken into consideration.</td>
</tr>
<tr>
<td><strong>immediate merge</strong></td>
<td>The immediate merge trigger checks whether it has to be activated as soon as the Merged Exit State changes.</td>
</tr>
<tr>
<td><strong>until final</strong></td>
<td>The until final trigger periodically checks whether it has to be activated. This check starts as soon as a job or batch has been submitted and does not stop until it is final. The until final trigger imperatively requires a condition. This condition is checked at least once. This check takes place when the job or batch switches to the finished state.</td>
</tr>
<tr>
<td><strong>until finished</strong></td>
<td>The until finished trigger is similar to the final trigger. The only difference is that the until finished trigger stops the check as soon as the job is finished. The until finished trigger imperatively requires a condition. This condition is checked at least once. This check takes place when the job or batch switches to the finished state.</td>
</tr>
</tbody>
</table>

*Continued on next page*
Table 6.5.: Description of the different types of triggers

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>This statement returns a confirmation of a successful operation.</td>
</tr>
</tbody>
</table>
create user

Purpose

The purpose of the create user statement is to create a pair of values which can be used to authenticate oneself to the server.

Syntax

The syntax for the create user statement is

```
create [ or alter ] user username
with WITHITEM {, WITHITEM}
```

WITHITEM:

- `connect type = < plain | ssl | ssl authenticated >`
- `default group = groupname`
- `< enable | disable >`
- `equivalent = none`
- `equivalent = ( < username | serverpath > {, < username | serverpath >} )`
- `group = ( groupname {, groupname} )`
- `parameter = none`
- `parameter = ( PARAMETERSPEC {, PARAMETERSPEC} )`
- `password = string`
- `rawpassword = string [ salt = string ]`

PARAMETERSPEC:

- `parametername = < string | number >`

Description

The create user statement is used to create a user. If "or alter" is specified, an already existing user is changed. Otherwise, an existing user will trigger an error. The default group clause is used to specify the Default Group.

**connect type** The connect type clause specifies which kind of connection must be used by the user to connect to the server.

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>plain</td>
<td>Every kind of connection is permitted</td>
</tr>
<tr>
<td>ssl</td>
<td>Only SSL connections are permitted</td>
</tr>
<tr>
<td>ssl authenticated</td>
<td>Only SSL connections with client authentication are permitted</td>
</tr>
</tbody>
</table>
**default group**  The default group clause defines the group that is used as the owner for all its objects created by the user if an explicit group was not specified when the object was created. The default group must be one of the user’s groups.

**enable**  The enable option allows the user to connect to the repository server.

**disable**  The disable option forbids the user from connecting to the repository server.

**group**  The group clause is used to specify the groups to which the user belongs. Every user is a member of the PUBLIC system group.

**password**  The password option is used to set the password for the user.

**rawpassword**  The rawpassword is used to set the user’s password that is required to connect to the repository server. The rawpassword is the already encrypted password. The rawpassword option has been implemented to be able to dump and restore users.

**Output**  

*Output*  This statement returns a confirmation of a successful operation.
7. deregister commands
deregister

Purpose

*Purpose*  The purpose of the *deregister* statement is to notify the server that the jobserver is not to process jobs anymore. See also the *register* statement on page 282.

Syntax

*Syntax*  The syntax for the *deregister* statement is

\[
\text{deregister serverpath . servername}
\]

Description

*Description*  The *deregister* statement is used to notify the server about a more or less permanent failure of a jobserver. This message prompts different server actions. Firstly, all the running jobs on the jobserver (i.e. jobs in the state *started*, *running*, *to_kill* and *killed*) are set to the state *broken_finished*. Jobs in the state *starting* are reset to *runnable*. The jobserver is then removed from the list of jobservers that are able to process jobs so that this jobserver is consequently no longer allocated any more jobs. A side effect of this is that jobs that can only run on this server due to their resource requirements are set to the state *error* with the message “Cannot run in any scope because of resource shortage”. Finally, a complete reschedule is executed so that jobs are redistributed among the jobservers. The jobserver is added to the list of job-processing jobservers again by re-registering it (refer to the *register* statement on page 282).

Output

*Output*  This statement returns a confirmation of a successful operation.
8. disconnect commands
**disconnect**

**Purpose**

*Purpose*  The purpose of the *disconnect* statement is to terminate the server connection.

**Syntax**

*Syntax*  The syntax for the *disconnect* statement is

```
disconnect
```

**Description**

*Description*  The connection to the server can be shut down using the *disconnect* statement.

**Output**

*Output*  This statement returns a confirmation of a successful operation.
9. drop commands
drop comment

Purpose
Purpose  The purpose of the drop comment statement is to remove the comment.

Syntax
Syntax  The syntax for the drop comment statement is

```
drop [ existing ] comment on OBJECTURL
```

OBJECTURL:

- distribution distributionname for pool identifier [. identifier] in serverpath
- environment environmentname
- exit state definition statename
- exit state mapping mappingname
- exit state profile profilename
- exit state translation transname
- event eventname
- resource identifier [. identifier] in folderpath
- folder folderpath
- footprint footprintname
- group groupname
- interval intervalname
- job definition folderpath
- job jobid
- named resource identifier [. identifier]
- parameter parametername of PARAM_LOC
- resource state definition statename
- resource state mapping mappingname
- resource state profile profilename
- scheduled event schedulepath . eventname
- schedule schedulepath
- resource identifier [. identifier] in serverpath
- < scope serverpath | observer serverpath >
- trigger triggername on TRIGGEROBJECT [ < noinverse | inverse > ]
- user username

PARAM_LOC:

- folder folderpath
- job definition folderpath
named resource identifier [. identifier]
< scope serverpath | jobserver serverpath >

TRIGGEROBJECT:
  resource identifier [. identifier] in folderpath
  job definition folderpath
  named resource identifier [. identifier]
  object monitor objecttypename
  resource identifier [. identifier] in serverpath

Description

The drop comment statement deletes the existing comment for the specified object. If the existing keyword is not specified, the absence of a comment is considered to be an error.

Output

This statement returns a confirmation of a successful operation.
drop environment

Purpose

The purpose of the drop environment statement is to remove the specified environment.

Syntax

The syntax for the drop environment statement is

\[ \text{drop [ existing ] environment environmentname} \]

Description

The drop environment statement is used to delete a definition from an environment. An error is triggered if jobs are still using this environment. If the existing keyword is being used, it is not considered to be an error if the specified environment does not exist.

Output

This statement returns a confirmation of a successful operation.
**drop event**

**Purpose**

The purpose of the *drop event* statement is to remove the specified event.

**Syntax**

The syntax for the *drop event* statement is

```
drop [existing] event eventname
```

**Description**

The *drop event* statement is used to delete a definition of an event. If the *existing* keyword is being used, it is not considered to be an error if the specified event does not exist. An event cannot be deleted if Scheduled Events belong to it.

**Output**

This statement returns a confirmation of a successful operation.
**drop exit state definition**

**Purpose**

The purpose of the *drop exit state definition* statement is to remove the specified exit state definition.

**Syntax**

The syntax for the *drop exit state definition* statement is

```
drop [ existing ] exit state definition statename
```

**Description**

The *drop exit state definition* statement is used to delete an Exit State Definition. It is considered to be an error if Exit State Profiles are still using this Exit State Definition. If the *existing* keyword is being used, it is *not* considered to be an error if the specified Exit State Definition does not exist.

**Output**

This statement returns a confirmation of a successful operation.
**drop exit state mapping**

**Purpose**

The purpose of the *drop exist state mapping* statement is to remove the specified mapping.

**Syntax**

The syntax for the *drop exit state mapping* statement is

```
drop [ existing ] exit state mapping mappingname
```

**Description**

The *drop exit state mapping* statement is used to delete an Exit State Mapping. It is considered to be an error if jobs or Exit State Profiles are still using this Exit State Mapping. If the *existing* keyword is being used, it is *not* considered to be an error if the specified Exit State Mapping does not exist.

**Output**

This statement returns a confirmation of a successful operation.
**drop exit state profile**

**Purpose**

*Purpose*  The purpose of the *drop exit state profile* statement is to remove the specified profile.

**Syntax**

*Syntax*  The syntax for the *drop exit state profile* statement is

```
drop [ existing ] exit state profile profilename
```

**Description**

*Description*  The *drop exit state profile* statement is used to delete a definition of an Exit State Profile. It is considered to be an error if jobs are still using this Exit State Profile. If the *existing* keyword is being used, it is *not* considered to be an error if the specified Exit State Profile does not exist.

**Output**

*Output*  This statement returns a confirmation of a successful operation.
drop exit state translation

Purpose
The purpose of the drop exit state translation statement is to remove the specified exit state translation.

Syntax
The syntax for the drop exit state translation statement is

\[\text{drop [existing] exit state translation transname}\]

Description
The drop exit state translation statement is used to delete Exit State Translations. It is considered to be an error if the translation is still being used in parent-child relationships. If the existing keyword is being used, it is not considered to be an error if the specified Exit State Translation does not exist.

Output
This statement returns a confirmation of a successful operation.
**drop folder**

**Purpose**

*Purpose*  The purpose of the *drop folder* statement is to remove a folder and its contents from the system.

**Syntax**

*Syntax*  The syntax for the *drop folder* statement is

```
drop [ existing ] FOLDER_OR_JOB {, FOLDER_OR_JOB} [ cascade ] [ force ]
```

FOLDER_OR_JOB:
```
[ < folder folderpath | job definition folderpath > ]
```

**Description**

*Description*  The *drop folder* statement removes folders and their contents from the system. There are two options:

- **Cascade**  The cascade option deletes folders, job definitions and subfolders, but only if they are not referenced to the job definitions, for example as required job.

- **Force**  With the force option, references to job definitions are removed as well. Force implies cascade. Folders cannot be deleted if they are not empty unless cascade or force has been specified.

**Output**

*Output*  This statement returns a confirmation of a successful operation.
**drop footprint**

**Purpose**

The purpose of the *drop footprint* statement is to remove the specified footprint.

**Syntax**

The syntax for the *drop footprint* statement is

```
drop [existing] footprint footprintname
```

**Description**

The *drop footprint* statement is used to delete footprints and resource requirements. If the *existing* keyword is being used, it is *not* considered to be an error if the specified footprint does not exist.

**Output**

This statement returns a confirmation of a successful operation.
**drop group**

*Purpose*

The purpose of the *drop group* statement is to remove a group from the system.

*Syntax*

The syntax for the *drop group* statement is

\[ \text{drop [ existing ] group groupname} \]

*Description*

The *drop group* statement is used to delete a group. If there are still any group members in this group, their membership is automatically terminated. It is considered to be an error if the group is still the owner of an object. It is not possible to delete a group that is defined as the Default Group for a user. If the *existing* keyword is being used, it is not considered to be an error if the specified group does not exist.

*Output*

This statement returns a confirmation of a successful operation.
**drop interval**

**Purpose**

The purpose of the *drop interval* statement is to remove the specified interval.

**Syntax**

The syntax for the *drop interval* statement is

\[
drop \ [ \text{existing} \] \ interval \ intervalname
\]

**Description**

The *drop interval* statement is used to delete intervals. If the *existing* keyword is being used, it is not considered to be an error if the specified interval does not exist.

**Output**

This statement returns a confirmation of a successful operation.
**drop job definition**

**Purpose**

The purpose of the *drop job definition* statement is to remove the specified scheduling entity object.

**Syntax**

The syntax for the *drop job definition* statement is

```
drop [ existing ] job definition folderpath [ force ]
```

**Description**

The *drop job definition* statement deletes the given job definition. If a job definition is referenced (for instance as Required Job), it cannot be deleted unless the force option is specified. If the force option is being used, all references to a job definition are also deleted.

**Output**

This statement returns a confirmation of a successful operation.
**drop named resource**

**Purpose**

The purpose of the *drop named resource* statement is to delete a class of resources.

**Syntax**

The syntax for the *drop named resource* statement is

\[ \text{drop } [\text{existing }] \text{ named resource identifier [. identifier]} [\text{cascade}] \]

**Description**

The *drop named resource* statement is used to delete Named Resources. It is considered to be an error if the Named Resource is still instantiated in scopes, job definitions and/or folders and the *cascade* option is not specified. On the other hand, Scope Resources as well as folders and Job Definition Resources are deleted if the *cascade* option is specified. If any requirements exist for the Named Resources that are to be deleted, the statement will fail. If the *existing* keyword is being used, it is not considered to be an error if the specified Name Resource does not exist.

**Output**

This statement returns a confirmation of a successful operation.
**drop resource**

**Purpose**

The purpose of the `drop resource` statement is to remove an instance of a named resource from a scope, folder or job definition.

**Syntax**

The syntax for the `drop resource` statement is

\[
\text{drop [ existing ] RESOURCE_URL [ force ]}
\]

**Description**

The `drop resource` statement is used to delete a resource. It is considered to be an error if the resource is still being allocated by Running Jobs. If the `existing` keyword is being used, it is *not* considered to be an error if the specified resource does not exist.

**Output**

This statement returns a confirmation of a successful operation.
drop resource state definition

Purpose

The purpose of the *drop resource state definition* statement is to remove the definition.

Syntax

The syntax for the *drop resource state definition* statement is

```
drop [existing] resource state definition statename
```

Description

The *drop resource state definition* statement is used to delete Resource State Definitions. It is considered to be an error if Resource State Profiles are still using this Resource State Definition. If the *existing* keyword is being used, it is *not* considered to be an error if the specified Resource State Definition does not exist.

Output

This statement returns a confirmation of a successful operation.
drop resource state mapping

Purpose
The purpose of the drop resource state mapping statement is to delete the mapping.

Syntax
The syntax for the drop resource state mapping statement is

\[
\text{drop [ existing ] resource state mapping mappingname}
\]

Description
The drop resource state mapping statement is used to delete a Resource State Mapping. It is considered to be an error if job definitions are using this Resource State Mapping. If the existing keyword is being used, it is not considered to be an error if the specified Resource State Mapping does not exist.

Output
This statement returns a confirmation of a successful operation.
drop resource state profile

Purpose

The purpose of the drop resource state profile statement is to remove a resource state profile.

Syntax

The syntax for the drop resource state profile statement is

\[ \text{drop [ existing ] resource state profile profilename} \]

Description

The drop resource state profile statement is used to delete the definition of a Resource State Profile. It is considered to be an error if Named Resources are still using this Resource State Profile. If the existing keyword is being used, it is not considered to be an error if the specified Resource State Profile does not exist.

Output

This statement returns a confirmation of a successful operation.
drop schedule

Purpose

Purpose  The purpose of the drop schedule statement is to remove the specified schedule.

Syntax

Syntax  The syntax for the drop schedule statement is

    drop [ existing ] schedule schedulepath

Description

Description  The drop schedule statement is used to delete schedules. If the existing keyword is being used, it is not considered to be an error if the specified schedule does not exist.

A schedule cannot be deleted if it has a Scheduled Event that belongs to it. It cannot be deleted either if child objects exist.

Output

Output  This statement returns a confirmation of a successful operation.
**drop scheduled event**

**Purpose**

The purpose of the *drop scheduled event* is to remove the specified scheduled event.

**Syntax**

The syntax for the *drop scheduled event* statement is

```
drop [existing] scheduled event schedulepath . eventname
```

**Description**

The *drop interval* statement is used to delete Scheduled Events. If the *existing* keyword is being used, it is *not* considered to be an error if the specified Schedule Event does not exist.

**Output**

This statement returns a confirmation of a successful operation.
**drop scope**

**Purpose**

The purpose of the *drop scope* statement is to remove a scope and its contents from the scope hierarchy.

**Syntax**

The syntax for the *drop scope* statement is

```
drop [ existing ] < scope serverpath | jobserver serverpath > [ cascade ]
```

**Description**

This statement is synonymous to the *drop jobserver* statement. The *cascade* option deletes the scope together with its contents.

**Output**

This statement returns a confirmation of a successful operation.
**drop trigger**

**Purpose**

The purpose of the *drop trigger* statement is to remove the specified trigger.

**Syntax**

The syntax for the *drop trigger* statement is

```sql
drop [ existing ] trigger triggername on TRIGGEROBJECT [ < noinverse | inverse > ]
```

**Description**

The *drop trigger* statement is used to delete a trigger.

If the *existing* keyword is being used, it is *not* considered to be an error if the specified trigger does not exist.

**Output**

This statement returns a confirmation of a successful operation.
**drop user**

**Purpose**

*Purpose* The purpose of the *drop user* statement is to remove the user from the system.

**Syntax**

*Syntax* The syntax for the *drop user* statement is

```
drop [ existing ] user username
```

**Description**

*Description* The *drop user* statement is used to logically delete a user. If the *existing* keyword is being used, it is *not* considered to be an error if the specified user does not exist.

**Output**

*Output* This statement returns a confirmation of a successful operation.
10. finish commands
**finish job**

**Purpose**

*Purpose* The purpose of the *finish job* command is to inform the server about the termination of a job.

**Syntax**

*Syntax* The syntax for the *finish job* statement is

```
finish job jobid
with exit code = signed_integer
```

```
finish job
with exit code = signed_integer
```

**Description**

*Description* The *finish job* command is used by the jobserver to report the Exit Code for a process to the server. During the course of repair work, it may be necessary for an administrator to tell the server in this way that a job has terminated. Jobs can themselves report that they have finished. To do this, they connect to the server and use the second form of the statement.

**Output**

*Output* This statement returns a confirmation of a successful operation.
11. Get commands
get parameter

Purpose

Purpose

The purpose of the get parameter statement is to get the value of the specified parameter within the context of the requesting job, respectively the specified job.

Syntax

Syntax

The syntax for the get parameter statement is

\[
\text{get parameter parametername [ < strict | warn | liberal > ]}
\]

\[
\text{get parameter of jobid parametername [ < strict | warn | liberal > ]}
\]

Description

Description

The get parameter statement is used to get the value of the specified parameter within the context of a job.

The additional option has the following meaning:

<table>
<thead>
<tr>
<th>Option</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>strict</td>
<td>The server returns an error if the requested parameter is not explicitly declared in the job definition.</td>
</tr>
<tr>
<td>warn</td>
<td>A message is written to the server’s log file when an attempt is made to determine the value of an undeclared parameter.</td>
</tr>
<tr>
<td>liberal</td>
<td>An attempt to query an undeclared parameter is tacitly allowed. The default behaviour depends on the configuration of the server.</td>
</tr>
</tbody>
</table>

Output

Output

This statement returns an output structure of type record.

Output Description

The data items of the output are described in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VALUE</td>
<td>Value of the requested parameter</td>
</tr>
</tbody>
</table>

Table 11.1.: Description of the output structure of the get parameter statement
**get submittag**

**Purpose**

The purpose of the *get submittag* statement is to get a (server local) unique identifier from the server. This identifier can be used to avoid *race conditions* between frontend and backend when submitting jobs.

**Syntax**

The syntax for the *get submittag* statement is

```
get submittag
```

**Description**

The *get submittag* statement is used to acquire an identification from the server. This prevents race conditions between the front end and back end when jobs are submitted. Such a situation arises when feedback about the submit does not reach the front end due to an error. By using a submittag, the front end can safely start a second attempt. The server recognises whether the job in question has already been submitted and responds accordingly. This reliably prevents the job from being submitted twice.

**Output**

This statement returns an output structure of type record.

**Output Description** The data items of the output are described in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VALUE</td>
<td>The requested Submit Tag</td>
</tr>
</tbody>
</table>

Table 11.2.: Description of the output structure of the get submittag statement
12. kill commands
**kill session**

**Purpose**

*Purpose* The purpose of the *kill session* is to terminate the specified session.

**Syntax**

*Syntax* The syntax for the *kill session* statement is

```
kill session sid
```

**Description**

*Description* The *list session* command can be used to display a list of active sessions. The displayed session Id can be used to terminate the session in question with the *kill session* command. Only administrators (i.e. members of the ADMIN group) are allowed to use this statement. It is not possible to terminate your own session.

**Output**

*Output* This statement returns a confirmation of a successful operation.
13. link commands
**link resource**

**Purpose**

The purpose of the *link resource* statement is to create a reference to a resource of another scope.

**Syntax**

The syntax for the *link resource* statement is

```
link resource identifier [identifier] in serverpath to < scope serverpath | jobserver serverpath > [ force ]
```

**Description**

With the *link resource* statement it is possible to make the resource of another scope visible and usable in a scope. This is necessary if a logical process requires resources from more than one scope. This is very well the case, for example, with processes that communicate with a database system.

From the system's perspective, it can scarcely differentiate between a Resource Link and the referenced resource. All operations such as allocating, locking, reading or setting variables take place on the base resource. This means that the link behaves as if it were the base resource. The only difference lies in the view of the allocations. With the base resource, all the allocations are shown. With a link, only those allocations that take place via the link are shown.

It is also possible to set links to links.

The *force* option can be used to overwrite an existing link. An already existing resource is deleted and the link is created. These operations are obviously only possible if the resource or link is not being used, i.e. if there are no allocations or reservations present.

**Output**

This statement returns a confirmation of a successful operation.
14. list commands
list calendar

Purpose

Purpose   The purpose of the list calendar statement is to get an overview of scheduled jobs.

Syntax

Syntax   The syntax for the list calendar statement is

```plaintext
list calendar [ with LC_WITHITEM {, LC_WITHITEM} ]

LC_WITHITEM:
   endtime = datetime
   | filter = LC_FILTERTERM [or LC_FILTERTERM]
   | starttime = datetime
   | timezone = string

LC_FILTERTERM:
   LC_FILTERITEM [and LC_FILTERITEM]

LC_FILTERITEM:
   ( LC_FILTERTERM [or LC_FILTERTERM] )
   | job . identifier < cmpop | like | not like > RVALUE
   | name like string
   | not ( LC_FILTERTERM [or LC_FILTERTERM] )
   | owner in ( groupname {, groupname} )

RVALUE:
   expr ( string )
   | number
   | string
```

Description

Description   The list calendar statement gives you a list of all the calendar entries sorted by the start dates of the executable objects. If a period is specified, those objects whose start time plus the Expected Final Time lies in the selected period are also displayed.
Output

This statement returns an output structure of type table.

Output Description The data items of the output are described in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>The repository object Id</td>
</tr>
<tr>
<td>SE_NAME</td>
<td>Name of the Scheduling Entity</td>
</tr>
<tr>
<td>SE_TYPE</td>
<td>Type of the Scheduling Entity (job or batch)</td>
</tr>
<tr>
<td>SE_ID</td>
<td>Id of the Scheduling Entity</td>
</tr>
<tr>
<td>SEOWNER</td>
<td>Owner of the Scheduling Entity</td>
</tr>
<tr>
<td>SE_PRIVS</td>
<td>Privileges for the Scheduling Entity</td>
</tr>
<tr>
<td>SCE_NAME</td>
<td>Name of the schedule</td>
</tr>
<tr>
<td>SCE_ACTIVE</td>
<td>Flag that indicates if the schedule is active</td>
</tr>
<tr>
<td>EVT_NAME</td>
<td>Name of the event</td>
</tr>
<tr>
<td>STARTTIME</td>
<td>Start time</td>
</tr>
<tr>
<td>EXPECTED_FINAL_TIME</td>
<td>Expected date and time the job or batch will reach a final state</td>
</tr>
<tr>
<td>TIME_ZONE</td>
<td>The used time zone for date and time display</td>
</tr>
</tbody>
</table>

Table 14.1.: Description of the output structure of the list calendar statement
**list dependency definition**

**Purpose**

*Purpose* The purpose of the `list dependency definition` statement is to get a list of all dependencies of a job definition.

**Syntax**

*Syntax* The syntax for the `list dependency definition` statement is

```
lst dependency definition folderpath
```

**Description**

*Description* The `list dependency definition` statement gives you a list of all the dependencies of a job definition.

**Output**

*Output* This statement returns an output structure of type table.

**Output Description** The data items of the output are described in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>The repository object Id</td>
</tr>
<tr>
<td>SE_DEPENDENT_PATH</td>
<td>The folder containing the dependent Scheduling Entity</td>
</tr>
<tr>
<td>DEPENDENT_NAME</td>
<td>The name of the dependent Scheduling Entity</td>
</tr>
<tr>
<td>SE_REQUIRED_PATH</td>
<td>The folder containing the required Scheduling Entity</td>
</tr>
<tr>
<td>REQUIRED_NAME</td>
<td>The name of the required Scheduling Entity</td>
</tr>
<tr>
<td>NAME</td>
<td>The object name</td>
</tr>
<tr>
<td>UNRESOLVED_HANDLING</td>
<td>The Unresolved Handling field describes what to do if a dependent object instance is not present in the current Master Batch. The following options are available: Ignore, Error and Suspend.</td>
</tr>
<tr>
<td>MODE</td>
<td>The Dependency Mode states the context in which the list of dependencies has to be viewed. The following options are available: ALL and ANY.</td>
</tr>
</tbody>
</table>

*Continued on next page*

220
<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATE_SELECTION</td>
<td>The State Selection defines how the required Exit States are determined. The options here are FINAL, ALL_REACHABLE, UNREACHABLE and DEFAULT. In the case of FINAL, the required Exit States can be explicitly listed.</td>
</tr>
<tr>
<td>ALL_FINALS</td>
<td>This field defines whether the dependency is already fulfilled when a Final State is reached (True) or if the required states are explicitly listed (False).</td>
</tr>
<tr>
<td>CONDITION</td>
<td>The condition that has to be fulfilled is entered in the Condition field.</td>
</tr>
<tr>
<td>STATES</td>
<td>This is the list of all the valid Exit States which the required object must have for the dependency to be fulfilled and so that the dependent job can start.</td>
</tr>
<tr>
<td>RESOLVE_MODE</td>
<td>The Resolve Mode defines the context in which the dependency is to be resolved. The possible values are:</td>
</tr>
<tr>
<td></td>
<td>Value</td>
</tr>
<tr>
<td></td>
<td>internal</td>
</tr>
<tr>
<td></td>
<td>both</td>
</tr>
<tr>
<td></td>
<td>external</td>
</tr>
<tr>
<td>EXPIRED_AMOUNT</td>
<td>When resolving an external dependency, the time when the required job or batch was active plays a role. The expired amount defines for how many time units this may lie in the past.</td>
</tr>
<tr>
<td>EXPIRED_BASE</td>
<td>The expired base defines the time unit for the expired amount.</td>
</tr>
<tr>
<td>SELECT_CONDITION</td>
<td>The select condition defines a condition that must be fulfilled so that a job or batch can be regarded as being a required job.</td>
</tr>
</tbody>
</table>

Table 14.2.: Description of the output structure of the list dependency definition statement
list dependency hierarchy

Purpose

Purpose The purpose of the list dependency hierarchy statement is to get a list of all dependencies of a submitted entity.

Syntax

Syntax The syntax for the list dependency hierarchy statement is

\[
\text{list [ condensed] \ dependency hierarchy jobid [ with EXPAND ]}
\]

EXPAND:

\[
\begin{align*}
\text{expand} & = \text{none} \\
| \quad \text{expand} & = < ( \text{id}, \text{id}) | \text{all} >
\end{align*}
\]

Description

Description The list dependency hierarchy statement gives you a list of all the dependencies of a Submitted Dependency.

expand The expand option can be used to make the hierarchy visible at children level. This is done by specifying in the list the IDs of the nodes whose children are to be made visible. If none is specified as an expand option, only the level below the requested node is made visible.

Output

Output This statement returns an output structure of type table.

Output Description The data items of the output are described in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>The Id of the Dependency Instance</td>
</tr>
<tr>
<td>DD_ID</td>
<td>The Id of the Dependency Definition</td>
</tr>
<tr>
<td>DEPENDENT_ID</td>
<td>This is the Id of the dependent job.</td>
</tr>
<tr>
<td>DEPENDENT_NAME</td>
<td>This is the fully qualified name of the dependent</td>
</tr>
<tr>
<td></td>
<td>job.</td>
</tr>
<tr>
<td>REQUIRED_ID</td>
<td>This is the Id of the required job.</td>
</tr>
</tbody>
</table>

Continued on next page
### Field Description

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>REQUIRED_NAME</td>
<td>This is the fully qualified name of the required job.</td>
</tr>
<tr>
<td>DEP_STATE</td>
<td>This is the current state of the dependency relationship. The following variants are used: Open, Fulfilled and Filed.</td>
</tr>
<tr>
<td>DEPENDENCY_PATH</td>
<td>This is a ';'-separated list of job hierarchies (parent-child relationships). Each job hierarchy is a list of path names separated by a colon (' ').</td>
</tr>
<tr>
<td>SE_DEPENDENT_ID</td>
<td>The Id of the dependent Scheduling Entity</td>
</tr>
<tr>
<td>SE_DEPENDENT_NAME</td>
<td>The fully qualified name of the dependent Scheduling Entity</td>
</tr>
<tr>
<td>SE_REQUIRED_ID</td>
<td>The Id of the required Scheduling Entity</td>
</tr>
<tr>
<td>SE_REQUIRED_NAME</td>
<td>The fully qualified name of the required Scheduling Entity</td>
</tr>
<tr>
<td>DD_NAME</td>
<td>Name of the Dependency Definition</td>
</tr>
<tr>
<td>UNRESOLVED_HANDLING</td>
<td>The Unresolved Handling field describes what to do if a dependent object instance is not present in the current Master Batch. The following options are available: Ignore, Error and Suspend.</td>
</tr>
<tr>
<td>MODE</td>
<td>States the currently used Dependency Mode (ALL_FINAL or JOB_FINAL).</td>
</tr>
<tr>
<td>STATE_SELECTION</td>
<td>The State Selection defines how the required Exit States are determined. The options here are FINAL, ALL_REACHABLE, UNREACHABLE and DEFAULT. In the case of FINAL, the required Exit States can be explicitly listed.</td>
</tr>
<tr>
<td>MASTER_ID</td>
<td>This is the Id of the Master Job that was submitted in order to create this runtime object.</td>
</tr>
<tr>
<td>SE_TYPE</td>
<td>This is the Scheduling Entity type (job, batch or milestone).</td>
</tr>
<tr>
<td>PARENT_ID</td>
<td>This is the Id of the parent runtime object that submitted the current job. If the job does not have a parent, NONE is displayed here.</td>
</tr>
<tr>
<td>PARENT_NAME</td>
<td>This is the fully qualified name of the parent runtime object that submitted the current job.</td>
</tr>
<tr>
<td>OWNER</td>
<td>The group owning the object</td>
</tr>
</tbody>
</table>

*Continued on next page*
<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCOPE</td>
<td>This is the fully qualified name of the jboss server on which the job was started. If the job has not yet been started, ‘null’ is displayed here.</td>
</tr>
<tr>
<td>EXIT_CODE</td>
<td>The Exit Code is the exit value that the Run program had when the process finished.</td>
</tr>
<tr>
<td>PID</td>
<td>This is the process Id of the Job Executor.</td>
</tr>
<tr>
<td>EXTPID</td>
<td>This is the Id of the process that is being executed.</td>
</tr>
<tr>
<td>JOB_STATE</td>
<td>The current Job State</td>
</tr>
<tr>
<td>JOB_ESD</td>
<td>This is the job’s Exit State. If the job has not yet finished, ‘null’ is displayed here.</td>
</tr>
<tr>
<td>FINAL_ESD</td>
<td>This is the Merged Exit State.</td>
</tr>
<tr>
<td>JOB_IS_FINAL</td>
<td>Specifies whether the job is Final (True) or not (False)</td>
</tr>
<tr>
<td>CNT_REQUIRED</td>
<td>The number of jobs that are dependent on the current job if its status is dependency_wait</td>
</tr>
<tr>
<td>CNT_RESTARTABLE</td>
<td>The number of children in a Restartable state</td>
</tr>
<tr>
<td>CNT_SUBMITTED</td>
<td>The number of children in a Submitted state</td>
</tr>
<tr>
<td>CNT_DEPENDENCY_WAIT</td>
<td>The number of children in a Dependency_Wait state</td>
</tr>
<tr>
<td>CNT_RESOURCE_WAIT</td>
<td>The number of children in a Resource_Wait state</td>
</tr>
<tr>
<td>CNT_RUNNABLE</td>
<td>The number of children in a Runnable state</td>
</tr>
<tr>
<td>CNT_STARTING</td>
<td>The number of children in a Starting state</td>
</tr>
<tr>
<td>CNT_STARTED</td>
<td>The number of children in a Started state</td>
</tr>
<tr>
<td>CNT_RUNNING</td>
<td>The number of children in a Running state</td>
</tr>
<tr>
<td>CNT_TO_KILL</td>
<td>The number of children in a To_Kill state</td>
</tr>
<tr>
<td>CNT_KILLED</td>
<td>The number of children in a Killed state</td>
</tr>
<tr>
<td>CNTCANCELLED</td>
<td>The number of children in a Cancelled state</td>
</tr>
<tr>
<td>CNT_FINAL</td>
<td>The number of children in a Final state</td>
</tr>
<tr>
<td>CNT_BROKEN_ACTIVE</td>
<td>The number of children in a Broken_Active state</td>
</tr>
<tr>
<td>CNT_BROKEN_FINISHED</td>
<td>The number of children in a Broken_Finished state</td>
</tr>
<tr>
<td>CNT_ERROR</td>
<td>The number of children in an Error state</td>
</tr>
<tr>
<td>CNT_SYNCHRONIZE_WAIT</td>
<td>The number of children in a Synchronize_Wait state</td>
</tr>
<tr>
<td>CNT_FINISHED</td>
<td>The number of children in a Finished state</td>
</tr>
<tr>
<td>SUBMIT_TS</td>
<td>The time when the job was submitted</td>
</tr>
</tbody>
</table>
### Field Description

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNC_TS</td>
<td>The time when the job switched to the state synchronize_wait</td>
</tr>
<tr>
<td>RESOURCE_TS</td>
<td>The time when the job switched to the state Resource_wait</td>
</tr>
<tr>
<td>RUNNABLE_TS</td>
<td>The time when the job reached the state Runnable</td>
</tr>
<tr>
<td>START_TS</td>
<td>The time when the job was reported by the jobserver as having been started</td>
</tr>
<tr>
<td>FINISH_TS</td>
<td>The time when the job reached the state Finished</td>
</tr>
<tr>
<td>FINAL_TS</td>
<td>The time when the job reached the state Final</td>
</tr>
<tr>
<td>ERROR_MSG</td>
<td>The error message that was displayed on reaching the state Error</td>
</tr>
<tr>
<td>DEPENDENT_ID_ORIG</td>
<td>The Id of the object that defined the dependency</td>
</tr>
<tr>
<td>DEPENDENCY_OPERATION</td>
<td>The Dependency Operation defines whether all the dependencies (All) or just one single dependency have to be fulfilled.</td>
</tr>
<tr>
<td>CHILD_TAG</td>
<td>Marker for differentiating between multiple dynamically submitted children</td>
</tr>
<tr>
<td>CHILDREN</td>
<td>The number of the children of the job</td>
</tr>
<tr>
<td>REQUIRED</td>
<td>The number of dependent jobs</td>
</tr>
<tr>
<td>DD_STATES</td>
<td>A comma-separated list of the required Exit States</td>
</tr>
<tr>
<td>IS_SUSPENDED</td>
<td>This field defines whether the job is suspended (True) or not (False).</td>
</tr>
<tr>
<td>PARENT_SUSPENDED</td>
<td>This field defines whether the job is suspended (True) or not (False) through one of its parents.</td>
</tr>
<tr>
<td>CNT_UNREACHABLE</td>
<td>The number of children whose dependencies cannot be fulfilled</td>
</tr>
<tr>
<td>DEPENDENT_PATH_ORIG</td>
<td>The fully qualified name of the object that defined the dependency</td>
</tr>
<tr>
<td>IGNORE</td>
<td>Ignore indicates whether this dependency is ignored (True) or not (False)</td>
</tr>
</tbody>
</table>

*Continued on next page*
### Table 14.3.: Description of the output structure of the list dependency hierarchy statement

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RESOLVE_MODE</td>
<td>The Resolve Mode defines the context in which the dependency is to be resolved. The possible values are:</td>
</tr>
<tr>
<td></td>
<td><strong>Value</strong></td>
</tr>
<tr>
<td>internal</td>
<td></td>
</tr>
<tr>
<td>both</td>
<td></td>
</tr>
<tr>
<td>external</td>
<td></td>
</tr>
<tr>
<td>EXPIRED_AMOUNT</td>
<td>When resolving an external dependency, the time when the required job or batch was active plays a role. The expired amount defines for how many time units this may lie in the past.</td>
</tr>
<tr>
<td>EXPIRED_BASE</td>
<td>The expired base defines the time unit for the expired amount</td>
</tr>
<tr>
<td>SELECT_CONDITION</td>
<td>The select condition defines a condition that must be fulfilled so that a job or batch can be regarded as being a required job.</td>
</tr>
</tbody>
</table>
list environment

Purpose

The purpose of the list environment statement is to get a list of defined environments.

Syntax

The syntax for the list environment statement is

```
list environment
```

Description

The list environment statement is used to get a list of defined environments that are visible to the user.

Output

This statement returns an output structure of type table.

Output Description

The data items of the output are described in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>The repository object Id</td>
</tr>
<tr>
<td>NAME</td>
<td>The name of the environment</td>
</tr>
<tr>
<td>PRIVS</td>
<td>String containing the users privileges on the object</td>
</tr>
</tbody>
</table>

Table 14.4.: Description of the output structure of the list environment statement
**list event**

**Purpose**

*Purpose*  
The purpose of the *list event* statement is to get a list of all defined events.

**Syntax**

*Syntax*  
The syntax for the *list event* statement is

```
list event
```

**Description**

*Description*  
The *list event* statement creates a list of all the defined events.

**Output**

*Output*  
This statement returns an output structure of type table.

**Output Description**  
The data items of the output are described in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>The repository object Id</td>
</tr>
<tr>
<td>NAME</td>
<td>The object name</td>
</tr>
<tr>
<td>OWNER</td>
<td>The group owning the object</td>
</tr>
<tr>
<td>SCHEDULING_ENTITY</td>
<td>Batch or job that is submitted when this event occurs</td>
</tr>
<tr>
<td>PRIVS</td>
<td>String containing the users privileges on the object</td>
</tr>
</tbody>
</table>

Table 14.5.: Description of the output structure of the list event statement
list exit state definition

Purpose

The purpose of the list exit state definition statement is to get a list of all defined exit states.

Syntax

The syntax for the list exit state definition statement is

list exit state definition

Description

The list exit state definition statement gives you a list of all the Exit States.

Output

This statement returns an output structure of type table.

**Output Description**  The data items of the output are described in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>The repository object Id</td>
</tr>
<tr>
<td>NAME</td>
<td>The object name</td>
</tr>
<tr>
<td>PRIVS</td>
<td>String containing the users privileges on the object</td>
</tr>
</tbody>
</table>

Table 14.6.: Description of the output structure of the list exit state definition statement
**list exit state mapping**

**Purpose**

*Purpose* The purpose of the *list exit state mapping* statement is to get a list of all defined mappings.

**Syntax**

*Syntax* The syntax for the *list exit state mapping* statement is

```plaintext
list exit state mapping
```

**Description**

*Description* The *list exit state mapping* statement gives you a list of all the defined mappings.

**Output**

*Output* This statement returns an output structure of type table.

**Output Description** The data items of the output are described in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>The repository object Id</td>
</tr>
<tr>
<td>NAME</td>
<td>The object name</td>
</tr>
<tr>
<td>PRIVS</td>
<td>String containing the users privileges on the object</td>
</tr>
</tbody>
</table>

Table 14.7.: Description of the output structure of the *list exit state mapping* statement
**list exit state profile**

**Purpose**

The purpose of the `list exit state profile` statement is to get a list of all defined exit state profiles.

**Syntax**

The syntax for the `list exit state profile` statement is

```
list exit state profile
```

**Description**

The `list exit state profile` statement gives you a list of all the defined Exit State Profiles.

**Output**

This statement returns an output structure of type table.

**Output Description** The data items of the output are described in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>The repository object Id</td>
</tr>
<tr>
<td>NAME</td>
<td>The object name</td>
</tr>
<tr>
<td>DEFAULT_ESM_NAME</td>
<td>The default Exit State Mapping is active if the job itself does not define something else.</td>
</tr>
<tr>
<td>IS_VALID</td>
<td>Flag displayed showing the validity of this Exit State Profile</td>
</tr>
<tr>
<td>PRIVS</td>
<td>String containing the users privileges on the object</td>
</tr>
</tbody>
</table>

Table 14.8.: Description of the output structure of the list exit state profile statement
list exit state translation

Purpose

Purpose  The purpose of the list exit state translation is to get a list of all defined exit state translations.

Syntax

Syntax  The syntax for the list exit state translation statement is

list exit state translation

Description

Description  The list exit state translation statement gives you a list of all the defined Exit State Translations.

Output

Output  This statement returns an output structure of type table.

Output Description  The data items of the output are described in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>The repository object Id</td>
</tr>
<tr>
<td>NAME</td>
<td>The object name</td>
</tr>
<tr>
<td>PRIVS</td>
<td>String containing the users privileges on the object</td>
</tr>
</tbody>
</table>

Table 14.9.: Description of the output structure of the list exit state translation statement
**list folder**

**Purpose**

The purpose of the *list folder* statement is to get a (partial) list of all folders defined in the system.

**Syntax**

The syntax for the *list folder* statement is

\[
\text{list} \ [\text{condensed}] \ [\text{folder} \ \text{folderpath} \ [\text{with} \ \text{WITHITEM} \{, \ \text{WITHITEM}\}] ]
\]

\[
\text{list} \ [\text{condensed}] \ [\text{folder} \ id \ [\text{with} \ \text{WITHITEM} \{, \ \text{WITHITEM}\}] ]
\]

WITHITEM:

\[
\text{expand} = \text{none} \\
| \text{expand} = < ( id \{, id\} ) | \text{all} > \\
| \text{FILTERTERM} \{\text{or} \ \text{FILTERTERM}\}
\]

FILTERTERM:

\[
\text{FILTERITEM} \{\text{and} \ \text{FILTERITEM}\}
\]

FILTERITEM:

\[
( \text{FILTERTERM} \{\text{or} \ \text{FILTERTERM}\} ) \\
| \text{name like string} \\
| \text{not} ( \text{FILTERTERM} \{\text{or} \ \text{FILTERTERM}\} ) \\
| \text{owner in} ( \text{groupname} \{, \ \text{groupname}\} )
\]

**Description**

The *list folder* statement gives you a list for the specified folder with all the direct child folders.

**expand** The expand option can be used to make the hierarchy visible at children level. This is done by specifying in the list the IDs of the nodes whose children are to be made visible. If **none** is specified as an expand option, only the level below the requested node is made visible.
filter  The child folders can be selected by name. Refer to the official Java documentation for the exact syntax used for regular expressions. The various conditions can be combined with one another using and and or. The usual valuation order of the operators applies (and before or).

Output

Output  This statement returns an output structure of type table.

Output Description  The data items of the output are described in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>The repository object Id</td>
</tr>
<tr>
<td>NAME</td>
<td>The object name</td>
</tr>
<tr>
<td>OWNER</td>
<td>The group owning the object</td>
</tr>
<tr>
<td>TYPE</td>
<td>This states the type of object. The following options are available: Batch, Milestone, Job and Folder.</td>
</tr>
<tr>
<td>RUN_PROGRAM</td>
<td>A command line that starts the script or program can be specified in the Run_Program field.</td>
</tr>
<tr>
<td>RERUN_PROGRAM</td>
<td>The Rerun_Program field specifies the command that is to be executed when repeating the job following an error (rerun).</td>
</tr>
<tr>
<td>KILL_PROGRAM</td>
<td>The Kill_Program field determines which program is to be run to terminate a currently running job.</td>
</tr>
<tr>
<td>WORKDIR</td>
<td>This is the working directory of the current job.</td>
</tr>
<tr>
<td>LOGFILE</td>
<td>The Logfile field specifies the file in which all the normal outputs of the Run program are to be returned. These are usually all the outputs that use the standard output channel (STDOUT under UNIX).</td>
</tr>
<tr>
<td>TRUNC_LOG</td>
<td>Defines whether the log file is to be renewed or not</td>
</tr>
<tr>
<td>ERRLOGFILE</td>
<td>The Error Logfile field specifies the file in which all the error outputs from the Run_program are to be returned.</td>
</tr>
<tr>
<td>TRUNC_ERRLOG</td>
<td>Defines whether the Error log file is to be renewed or not</td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXPECTED_RUNTIME</td>
<td>The Expected_Runtime describes the anticipated time that will be required to execute a job.</td>
</tr>
<tr>
<td>EXPECTED_FINALTIME</td>
<td>The Expected_Finaltime describes the anticipated time that will be required to execute a job or batch together with its children.</td>
</tr>
<tr>
<td>GET_EXPECTED_RUNTIME</td>
<td>This is a reserved field for future extended functions.</td>
</tr>
<tr>
<td>PRIORITY</td>
<td>The Priority field indicates the urgency with which the process, if it is to be started, is to be considered by the Scheduling System.</td>
</tr>
<tr>
<td>MIN_PRIORITY</td>
<td>This is the minimum effective priority that can be achieved through natural aging.</td>
</tr>
<tr>
<td>AGING_AMOUNT</td>
<td>The number of time units after which the effective priority is incremented by 1.</td>
</tr>
<tr>
<td>AGING_BASE</td>
<td>The time unit that is used for the aging interval</td>
</tr>
<tr>
<td>SUBMIT_SUSPENDED</td>
<td>Flag that indicates whether the object is to be suspended after the submit.</td>
</tr>
<tr>
<td>MASTER_SUBMITTABLE</td>
<td>The job that is started by the trigger is submitted as its own Master Job and does not have any influence on the current Master Job run of the triggering job.</td>
</tr>
<tr>
<td>SAME_NODE</td>
<td>Obsolete</td>
</tr>
<tr>
<td>GANG_SCHEDULE</td>
<td>Obsolete</td>
</tr>
<tr>
<td>DEPENDENCY_MODE</td>
<td>The Dependency Mode states the context in which the list of dependencies has to be viewed. The following options are available: ALL and ANY.</td>
</tr>
<tr>
<td>ESP_NAME</td>
<td>This is the name of the Exit State Profile.</td>
</tr>
<tr>
<td>ESM_NAME</td>
<td>This is the name of the Exit State Mapping.</td>
</tr>
<tr>
<td>ENV_NAME</td>
<td>This is the name of the environment.</td>
</tr>
<tr>
<td>FP_NAME</td>
<td>This is the name of the footprint.</td>
</tr>
<tr>
<td>SUBFOLDERS</td>
<td>This is the number of folders below the folder.</td>
</tr>
<tr>
<td>ENTITIES</td>
<td>This is the number of jobs and batches below the folder.</td>
</tr>
<tr>
<td>HAS_MSE</td>
<td>The folder contains at least one job that can be executed as a Master Submittable job.</td>
</tr>
<tr>
<td>PRIVS</td>
<td>String containing the users privileges on the object.</td>
</tr>
</tbody>
</table>

Continued on next page
Continued from previous page

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDPATH</td>
<td>Id of the path to the object</td>
</tr>
<tr>
<td>HIT</td>
<td>Line is a search hit Y/N.</td>
</tr>
</tbody>
</table>

Table 14.10.: Description of the output structure of the list folder statement
**list footprint**

**Purpose**

The purpose of the *list footprint* statement is to get a list of all defined footprints.

**Syntax**

The syntax for the *list footprint* statement is

```
list footprint
```

**Description**

The *list footprint* statement gives you a list of all the defined footprints.

**Output**

This statement returns an output structure of type table.

**Output Description**

The data items of the output are described in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>The repository object Id</td>
</tr>
<tr>
<td>NAME</td>
<td>The object name</td>
</tr>
<tr>
<td>PRIVS</td>
<td>String containing the users privileges on the object</td>
</tr>
</tbody>
</table>

Table 14.11.: Description of the output structure of the list footprint statement
**list group**

**Purpose**

*Purpose*  The purpose of the *list group* statement is to get a list of all defined groups.

**Syntax**

*Syntax*  The syntax for the *list group* statement is

\[ \text{list group} \]

**Description**

*Description*  The *list group* statement gives you a list of all the defined groups.

**Output**

*Output*  This statement returns an output structure of type table.

**Output Description**  The data items of the output are described in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>The repository object Id</td>
</tr>
<tr>
<td>NAME</td>
<td>The object name</td>
</tr>
<tr>
<td>PRIVS</td>
<td>String containing the users privileges on the object</td>
</tr>
</tbody>
</table>

Table 14.12.: Description of the output structure of the *list group* statement
list interval

Purpose

The purpose of the list interval statement is to get a list of all defined intervals.

Syntax

The syntax for the list interval statement is

list interval

list interval all

Description

The list interval statement gives you a list of all the defined intervals.

Output

This statement returns an output structure of type table.

Output Description

The data items of the output are described in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>The repository object Id</td>
</tr>
<tr>
<td>NAME</td>
<td>The object name</td>
</tr>
<tr>
<td>OWNER</td>
<td>The group owning the object</td>
</tr>
<tr>
<td>STARTTIME</td>
<td>The beginning of the interval. No edges are generated before this time.</td>
</tr>
<tr>
<td>ENDTIME</td>
<td>The end of the interval. No edges are generated after this time.</td>
</tr>
<tr>
<td>BASE</td>
<td>The period of the interval</td>
</tr>
<tr>
<td>DURATION</td>
<td>The duration of a block</td>
</tr>
<tr>
<td>SYNCTIME</td>
<td>The time with which the interval is synchronised. The first period of the interval starts at this time.</td>
</tr>
<tr>
<td>INVERSE</td>
<td>The definition whether the selection list should be regarded as being positive or negative</td>
</tr>
<tr>
<td>EMBEDDED</td>
<td>The interval from which a selection is subsequently made</td>
</tr>
</tbody>
</table>

Continued on next page
Continued from previous page

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OBJ_TYPE</td>
<td>The object type is the type of object to which the interval belongs.</td>
</tr>
<tr>
<td>OBJ_ID</td>
<td>The object id is the ID of the object to which the interval belongs.</td>
</tr>
<tr>
<td>PRIVS</td>
<td>String containing the users privileges on the object.</td>
</tr>
<tr>
<td>SE_ID</td>
<td>If an interval has been created as part of a schedule for a Job Definition, the job ID is set in the interval.</td>
</tr>
</tbody>
</table>

Table 14.13.: Description of the output structure of the list interval statement
list job

Purpose

The purpose of the list job statement is to get a list of submitted entities based on the selection criteria specified.

Syntax

The syntax for the list job statement is

```
list [ condensed ] job [ jobid [, jobid] ] [ with WITHITEM {, WITHITEM} ]
```

WITHITEM:

- enabled only
- expand = none
- expand = < ( id [, id] ) | all >
- FILTERTERM {or FILTERTERM}
- mode = < list | tree >
- parameter = ( parametername {, parametername} )

FILTERTERM:

FILTERITEM [and FILTERITEM]

FILTERITEM:

( FILTERTERM {or FILTERTERM} )
- < enable | disable >
- < final | restartable | pending >
- exit state in ( statename {, statename} )
- < history | future > = period
- history between period and period
- job . identifier < cmpop | like | not like > RVALUE
- job in ( jobid [, jobid] )
- jobserver in ( serverpath {, serverpath} )
- job status in ( JOBSTATE {, JOBSTATE} )
- master
- master_id in ( jobid [, jobid] )
- merged exit state in ( statename {, statename} )
- name in ( folderpath {, folderpath} )
- name like string
- node in ( nodename {, nodename} )
- not ( FILTERTERM {or FILTERTERM} )
The list job statement gives you a list of Submitted Entities. The selection of the jobs can be finely specified as required that by defining a filter. Job parameter names can also be specified that are then visible in the output.

The statement list job without any further information is equivalent to the statement list job with master and therefore outputs the list of all the Master Jobs and Batches.

expand  The expand option can be used to make the hierarchy visible at children level. This is done by specifying in the list the IDs of the nodes whose children are to be made visible. If none is specified as an expand option, only the level below the requested node is made visible.
mode  list mode just outputs a list of selected jobs. If the tree mode is defined, however, all the parents for each selected job are outputted as well.

parameter  Additional information about the selected jobs can be outputted by specifying parameter names. The parameters are valued in the context of each job and the value of the parameter is displayed in the output. If this fails, the output is an empty string. This means that specifying non-existent parameter names does not have any adverse consequences. This allows state or progress details for jobs to be easily and clearly displayed.

filter  A large number of filters are available for filtering all the jobs present in the system. The individual filters can be combined with one another using Boolean operators. The usual order of priority operator applies here. The individual filter functions are briefly described here.

FINAL, RESTARTABLE, PENDING  This filter selects all the jobs in the state final respectively restartable or pending.

EXIT STATE  All jobs that are in an Exit State defined in the specified list are selected. This is the job’s own Exit State, and not the Merged Exit State which also takes the Exit States of the children into consideration.

HISTORY  By defining a history, only those jobs that have become final at the earliest before the given time are selected. All non-final jobs are selected.

FUTURE  Scheduled future jobs are also outputted by specifying a future. These events are determined based on Scheduled Events and calendar entries. “SCHEDULED” is outputted as the state of such jobs.

JOB.IDENTIFIER  This filter is used to select all those jobs whose defined parameters fulfil the condition. This allows all the jobs of a developer to be easily selected, for example. (This obviously assumes that each job has a parameter with the developer’s name).

The expr Function can be used to perform calculations. The expression

job.starttime < expr('job.sysdate - job.expruntime * 1.5')

determines those jobs, that exceeded their expected runtime by more than 50%.

JOB IN (ID, ...)  This filter option is equivalent to specifying Jobids after “list job”. Only those jobs with one of the specified IDs are selected.

JOBSERVER  Only those jobs running on the specified jobserver are selected.

JOB STATE  This filter selects only those jobs that have one of the specified job states. For example, it is then easy to find all the jobs in the state broken_finished.

MASTER  Only the Master Jobs and Batches are selected.

MASTER_ID  Only jobs that belong to the specified Master Jobs and Batches are selected.
MERGED EXIT STATE All jobs that are in a Merged Exit State defined in the specified list are selected. This is the Exit State that results from a job’s own Exit State in combination with the Exit States of the children.

NAME IN (FOLDERPATH, . . .) The jobs whose associated Scheduling Entity is included in the specified list are selected.

NAME LIKE STRING The jobs whose associated Scheduling Entity has the matching name are selected. Refer to the official Java documentation for more details about the syntax used for regular expressions.

NODE Jobs running on one of the specified nodes are selected. In this context, the node designates the entry for the node of the jobserver.

OWNER Only the jobs of the defined owners (groups) are selected.

SUBMITTING USER Only jobs that have been submitted by the specified user are selected.

Output

Output This statement returns an output structure of type table.

Output Description The data items of the output are described in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>The repository object Id</td>
</tr>
<tr>
<td>MASTER_ID</td>
<td>This is the ID of the Master Job.</td>
</tr>
<tr>
<td>HIERARCHY_PATH</td>
<td>The Hierarchy Path is the full path of the current entry. The single hierarchy levels are separated by a period.</td>
</tr>
<tr>
<td>SE_TYPE</td>
<td>This is the Scheduling Entity type (job, batch or milestone).</td>
</tr>
<tr>
<td>PARENT_ID</td>
<td>This is the Id of the parent.</td>
</tr>
<tr>
<td>OWNER</td>
<td>The group owning the object</td>
</tr>
<tr>
<td>SCOPE</td>
<td>The scope or jobserver to which the job is allocated</td>
</tr>
<tr>
<td>HTTPHOST</td>
<td>The host name of the scope for accessing log files via HTTP</td>
</tr>
<tr>
<td>HTTPPORT</td>
<td>The HTTP port number of the jobserver for accessing log files via HTTP</td>
</tr>
<tr>
<td>EXIT_CODE</td>
<td>The Exit_Code of the executed process</td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PID</td>
<td>The PID is the process identification number of the monitoring jobserver process on the respective host system.</td>
</tr>
<tr>
<td>EXTPID</td>
<td>The EXT_PID is the process identification number of the utility process.</td>
</tr>
<tr>
<td>STATE</td>
<td>The State is the current state of the job.</td>
</tr>
<tr>
<td>IS_DISABLED</td>
<td>Indicates whether the submitted entity is disabled</td>
</tr>
<tr>
<td>IS_CANCELLED</td>
<td>Indicates whether a Cancel operation was performed on the job</td>
</tr>
<tr>
<td>JOB_ESD</td>
<td>The Job_Esd is the Exit State of the job.</td>
</tr>
<tr>
<td>FINAL_ESD</td>
<td>The final_esd is the Merged Exit State of the job or batch job with all the Child Exit States.</td>
</tr>
<tr>
<td>JOB_IS_FINAL</td>
<td>This field defines whether the job itself is final.</td>
</tr>
<tr>
<td>CNT_RESTARTABLE</td>
<td>The number of children in a Restartable state</td>
</tr>
<tr>
<td>CNT_SUBMITTED</td>
<td>The number of children in a Submitted state</td>
</tr>
<tr>
<td>CNT_DEPENDENCY_WAIT</td>
<td>The number of children in a Dependcy.Wait state</td>
</tr>
<tr>
<td>CNT_SYNCHRONIZE_WAIT</td>
<td>The number of children in a Synchronize.Wait state</td>
</tr>
<tr>
<td>CNT_RESOURCE_WAIT</td>
<td>The number of children in a Resource.Wait state</td>
</tr>
<tr>
<td>CNT_RUNNABLE</td>
<td>The number of children in a Runnable state</td>
</tr>
<tr>
<td>CNT_STARTING</td>
<td>The number of children in a Starting state</td>
</tr>
<tr>
<td>CNT_STARTED</td>
<td>The number of children in a Started state</td>
</tr>
<tr>
<td>CNT_RUNNING</td>
<td>The number of children in a Running state</td>
</tr>
<tr>
<td>CNT_TO_KILL</td>
<td>The number of children in a To_Kill state</td>
</tr>
<tr>
<td>CNT_KILLED</td>
<td>The number of children in a Killed state</td>
</tr>
<tr>
<td>CNT_CANCELLED</td>
<td>The number of children in a Cancelled state</td>
</tr>
<tr>
<td>CNT_FINISHED</td>
<td>The number of children in a Finished state</td>
</tr>
<tr>
<td>CNT_FINAL</td>
<td>The number of children in a Final state</td>
</tr>
<tr>
<td>CNT_BROKEN_ACTIVE</td>
<td>The number of children in a Broken_Active state</td>
</tr>
<tr>
<td>CNT_BROKEN_FINISHED</td>
<td>The number of children in a Broken_Finished state</td>
</tr>
<tr>
<td>CNT_ERROR</td>
<td>The number of children in an Error state</td>
</tr>
<tr>
<td>CNT_UNREACHABLE</td>
<td>The number of children in a Unreachable state</td>
</tr>
<tr>
<td>CNT_WARN</td>
<td>The number of children with a warning</td>
</tr>
<tr>
<td>SUBMIT_TS</td>
<td>The time when the job was submitted</td>
</tr>
</tbody>
</table>

*Continued on next page*
<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RESUME_TS</td>
<td>The time when the job is automatically resumed</td>
</tr>
<tr>
<td>SYNC_TS</td>
<td>The time when the job switched to the state synchronize_wait</td>
</tr>
<tr>
<td>RESOURCE_TS</td>
<td>The time when the job switched to the state Resource_wait</td>
</tr>
<tr>
<td>RUNNABLE_TS</td>
<td>The time when the job reached the state Runnable</td>
</tr>
<tr>
<td>START_TS</td>
<td>The time when the job was reported by the job-server as having been started</td>
</tr>
<tr>
<td>FINISH_TS</td>
<td>This is the time when the job is finished</td>
</tr>
<tr>
<td>FINAL_TS</td>
<td>The time when the job reached the state Final</td>
</tr>
<tr>
<td>PRIORITY</td>
<td>The static priority of a job. This is derived from the defined priority and the nice values of the parent(s).</td>
</tr>
<tr>
<td>DYNAMIC_PRIORITY</td>
<td>The Dynamic_Priority of the job. This is the static priority that was corrected dependent on the delay time.</td>
</tr>
<tr>
<td>NICEVALUE</td>
<td>The nice value is the correction of the children’s priority.</td>
</tr>
<tr>
<td>MIN_PRIORITY</td>
<td>This is the minimum value for the dynamic priority.</td>
</tr>
<tr>
<td>AGING_AMOUNT</td>
<td>The Aging_Amount defines after how many time units the dynamic priority of a job is incremented by one point.</td>
</tr>
<tr>
<td>AGING_BASE</td>
<td>The Aging_Base defines the time unit for the Aging Amount.</td>
</tr>
<tr>
<td>ERROR_MSG</td>
<td>The error message describing why the job switched to the error state.</td>
</tr>
<tr>
<td>CHILDREN</td>
<td>The number of children of the job or batch</td>
</tr>
<tr>
<td>HIT</td>
<td>This field indicates whether the job was selected based on filter criteria or not.</td>
</tr>
<tr>
<td>HITPATH</td>
<td>This field indicates that the job is a direct or indirect parent of a selected job.</td>
</tr>
<tr>
<td>SUBMITPATH</td>
<td>This is the list of submitting parents. In contrast to the general parent-child hierarchy, this is always unequivocal.</td>
</tr>
<tr>
<td>IS_SUSPENDED</td>
<td>This field defines whether the job or batch itself is suspended.</td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IS_RESTARTABLE</td>
<td>This field defines whether the job is restartable.</td>
</tr>
<tr>
<td>PARENT_SUSPENDED</td>
<td>This field defines whether the job is suspended (True) or not (False) through one of its parents.</td>
</tr>
<tr>
<td>CHILDTAG</td>
<td>The tag that enables a differentiation to be made between multiple children</td>
</tr>
<tr>
<td>IS_REPLACED</td>
<td>This field defines whether the job or batch has been replaced by another one.</td>
</tr>
<tr>
<td>WARN_COUNT</td>
<td>This is the number of unattended warnings.</td>
</tr>
<tr>
<td>CHILD_SUSPENDED</td>
<td>The number of children that have been suspended</td>
</tr>
<tr>
<td>CNT_PENDING</td>
<td>The number of children in a Pending state</td>
</tr>
<tr>
<td>PRIVS</td>
<td>String containing the users privileges on the object</td>
</tr>
<tr>
<td>WORKDIR</td>
<td>Name of the working directory of the utility process</td>
</tr>
<tr>
<td>LOGFILE</td>
<td>Name of the utility process log file. The output to stdout is written in this log.</td>
</tr>
<tr>
<td>ERRLOGFILE</td>
<td>Name of the utility process error log file. The output to stderr is written in this log.</td>
</tr>
<tr>
<td>APPROVAL_PENDING</td>
<td>This field indicates whether an Approval or Review Request exists for the relevant Submitted Entity</td>
</tr>
</tbody>
</table>

Table 14.14.: Description of the output structure of the list job statement
list job definition hierarchy

Purpose

The purpose of the list job definition hierarchy statement is to get the complete jobtree of the specified job.

Syntax

The syntax for the list job definition hierarchy statement is

\[
\text{list} \ [ \text{condensed} \ | \ \text{job definition hierarchy} \ \text{folderpath} \ [ \text{with} \ \text{EXPAND} \ ] \\
\text{list} \ [ \text{condensed} \ | \ \text{job definition hierarchy} \ \text{id} \ [ \text{with} \ \text{EXPAND} \ ]
\]

\[
\text{EXPAND}:
\begin{align*}
& \text{expand} = \text{none} \\
| & \text{expand} = < (\text{id} , \text{id}) > \ | \ \text{all}
\end{align*}
\]

Description

The list job definition statement hierarchy gives you the complete tree structure of the specified job.

Output

This statement returns an output structure of type table.

Output Description

The data items of the output are described in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>The repository object Id</td>
</tr>
<tr>
<td>NAME</td>
<td>The object name</td>
</tr>
<tr>
<td>OWNER</td>
<td>The group owning the object</td>
</tr>
<tr>
<td>TYPE</td>
<td>This states the type of object. The following options are available: Batch,</td>
</tr>
<tr>
<td></td>
<td>Milestone, Job and Folder.</td>
</tr>
<tr>
<td>RUN_PROGRAM</td>
<td>A command line that starts the script or program can be specified in the Run</td>
</tr>
<tr>
<td></td>
<td>Program field.</td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RERUN_PROGRAM</td>
<td>The Rerun_Program field specifies the command that is to be executed when repeating the job following an error (rerun).</td>
</tr>
<tr>
<td>KILL_PROGRAM</td>
<td>The Kill_Program field determines which program is to be run to terminate a currently running job.</td>
</tr>
<tr>
<td>WORKDIR</td>
<td>This is the working directory of the current job.</td>
</tr>
<tr>
<td>LOGFILE</td>
<td>The Logfile field specifies the file in which all the normal outputs of the Run program are to be returned. These are usually all the outputs that use the standard output channel (STDOUT under UNIX).</td>
</tr>
<tr>
<td>TRUNC_LOG</td>
<td>Defines whether the log file is to be renewed or not</td>
</tr>
<tr>
<td>ERRLOGFILE</td>
<td>The Error Logfile field specifies the file in which all the error outputs from the Run_program are to be returned.</td>
</tr>
<tr>
<td>TRUNC_ERRLOG</td>
<td>Defines whether the Error log file is to be renewed or not</td>
</tr>
<tr>
<td>EXPECTED_RUNTIME</td>
<td>The Expected_Runtime describes the anticipated time that will be required to execute a job.</td>
</tr>
<tr>
<td>GETEXPECTED_RUNTIME</td>
<td>This is a reserved field for future extended functions.</td>
</tr>
<tr>
<td>PRIORITY</td>
<td>The Priority field indicates the urgency with which the process, if it is to be started, is to be considered by the Scheduling System.</td>
</tr>
<tr>
<td>SUBMIT_SUSPENDED</td>
<td>The Submit_Suspended parameter specifies the form in which the Child Object is delayed when being started or if it can be started immediately. The following options are available: Yes, No and Childsuspend.</td>
</tr>
<tr>
<td>MASTER_SUBMITTABLE</td>
<td>The job that is started by the trigger is submitted as its own Master Job and does not have any influence on the current Master Job run of the triggering job.</td>
</tr>
<tr>
<td>SAME_NODE</td>
<td>Obsolete</td>
</tr>
<tr>
<td>GANG_SCHEDULE</td>
<td>Obsolete</td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEPENDENCY_MODE</td>
<td>The Dependency Mode states the context in which the list of dependencies has to be viewed. The following options are available: ALL and ANY.</td>
</tr>
<tr>
<td>ESP_NAME</td>
<td>This is the name of the Exit State Profile.</td>
</tr>
<tr>
<td>ESM_NAME</td>
<td>This is the name of the Exit State Mapping.</td>
</tr>
<tr>
<td>ENV_NAME</td>
<td>This is the name of the environment.</td>
</tr>
<tr>
<td>FP_NAME</td>
<td>This is the name of the footprint.</td>
</tr>
<tr>
<td>CHILDREN</td>
<td>This is the number of direct children.</td>
</tr>
<tr>
<td>SH_ID</td>
<td>The Id of the Hierarchy Definition</td>
</tr>
<tr>
<td>IS_STATIC</td>
<td>Flag indicating the static or dynamic submits of this job</td>
</tr>
<tr>
<td>IS_DISABLED</td>
<td>Flag indicating the child should be executed or skipped</td>
</tr>
<tr>
<td>INT_NAME</td>
<td>The interval id is the ID of the interval used to check whether the child is enabled.</td>
</tr>
<tr>
<td>ENABLE_CONDITION</td>
<td>The interval id is the ID of the interval used to check whether the child is enabled.</td>
</tr>
<tr>
<td>ENABLE_MODE</td>
<td>The interval id is the ID of the interval used to check whether the child is enabled.</td>
</tr>
<tr>
<td>SH_PRIORITY</td>
<td>The Priority field indicates the urgency with which the process, if it is to be started, is to be considered by the Scheduling System.</td>
</tr>
<tr>
<td>SH_SUSPEND</td>
<td>The Submit Suspended switch can be used to delay the actual start of a job run.</td>
</tr>
<tr>
<td>SH_ALIAS_NAME</td>
<td>A child can be assigned a new logical name by entering it in the Alias field.</td>
</tr>
<tr>
<td>MERGE_MODE</td>
<td>The Merge Mode indicates whether a Child Object is started multiple times within a Master Job run or not. The following options are available: No Merge, Failure, Merge Local and Merge Global.</td>
</tr>
<tr>
<td>EST_NAME</td>
<td>This is the Exit State Translation.</td>
</tr>
<tr>
<td>IGNORED_DEPENDENCIES</td>
<td>Here you can add a list of dependencies which are to be ignored by the child in this parent-child relationship.</td>
</tr>
</tbody>
</table>
Field | Description
---|---
HIERARCHY_PATH | The Path describes the parent folder hierarchy of an object. All the parent folders are displayed separated by periods.
STATES | The State is the current state of the job.
PRIVS | String containing the users privileges on the object

Table 14.15.: Description of the output structure of the list job definition hierarchy statement
**list named resource**

**Purpose**

The purpose of the `list named resource` statement is to get a (partial) list of all defined named resources.

**Syntax**

The syntax for the `list named resource` statement is

```
list named resource [ identifier { , identifier } ] [ with WITHITEM { , WITHITEM } ]

list named resource id [ with WITHITEM { , WITHITEM } ]

WITHITEM:
   expand = none
   | expand = < ( id { , id } ) | all >
   | FILTERTERM { or FILTERTERM }

FILTERTERM:
   FILTERITEM { and FILTERITEM }

FILTERITEM:
   ( FILTERTERM { or FILTERTERM } )
   | name like string
   | not ( FILTERTERM { or FILTERTERM } )
   | usage in ( RESOURCE_USAGE { , RESOURCE_USAGE } )

RESOURCE_USAGE:
   category
   | static
   | synchronizing
   | system
```

**Description**

The `list named resource` statement gives you a list of all the defined Named Resources. If a resource is specified, this Named Resource and, if the Named Resource is a category, all the children are listed. The list of Named Resources can be shortened accordingly by specifying a filter.
**expand** The expand option can be used to make the hierarchy visible at children level. This is done by specifying in the list the IDs of the nodes whose children are to be made visible. If **none** is specified as an expand option, only the level below the requested node is made visible.

**filter** Named Resources can be filtered by name and/or usage by specifying filters. Refer to the official Java documentation for the syntax used for regular expressions.

**Output**

This statement returns an output structure of type table.

**Output Description** The data items of the output are described in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>The repository object Id</td>
</tr>
<tr>
<td>NAME</td>
<td>The object name</td>
</tr>
<tr>
<td>OWNER</td>
<td>The group owning the object</td>
</tr>
<tr>
<td>USAGE</td>
<td>The Usage field specifies the Resource type.</td>
</tr>
<tr>
<td>RESOURCE_STATE_PROFILE</td>
<td>This is the Resource State Profile assigned to the resource.</td>
</tr>
<tr>
<td>FACTOR</td>
<td>This is the default factor by which Resource Requirement Amounts are multiplied if nothing else has been specified for the resource.</td>
</tr>
<tr>
<td>SUBCATEGORIES</td>
<td>This is the number of categories that are present as children below the displayed Named Resources.</td>
</tr>
<tr>
<td>RESOURCES</td>
<td>These are the instances of the Named Resource.</td>
</tr>
<tr>
<td>PRIVS</td>
<td>String containing the users privileges on the object</td>
</tr>
<tr>
<td>IDPATH</td>
<td>The idpath is a list of parent IDs separated by periods. This value is mainly used for frontend programs.</td>
</tr>
</tbody>
</table>

Table 14.16.: Description of the output structure of the list named resource statement
list resource state definition

Purpose

The purpose of the list resource state definition is to get a list of all defined resource states.

Syntax

The syntax for the list resource state definition statement is

list resource state definition

Description

The list resource state definition statement gives you a list of all the defined Resource States.

Output

This statement returns an output structure of type table.

Output Description The data items of the output are described in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>The repository object Id</td>
</tr>
<tr>
<td>NAME</td>
<td>The object name</td>
</tr>
<tr>
<td>PRIVS</td>
<td>String containing the users privileges on the object</td>
</tr>
</tbody>
</table>

Table 14.17.: Description of the output structure of the list resource state definition statement
list resource state mapping

Purpose
The purpose of the *list resource state mapping* statement is to get a list of all defined resource state mappings.

Syntax
The syntax for the *list resource state mapping* statement is

```
list resource state mapping
```

```
list resource state mapping for profilename
```

Description
The *list resource state mapping* gives you a list of all the defined Resource State Mappings.

Output
This statement returns an output structure of type table.

**Output Description** The data items of the output are described in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>The repository object Id</td>
</tr>
<tr>
<td>NAME</td>
<td>The object name</td>
</tr>
<tr>
<td>PRIVS</td>
<td>String containing the users privileges on the object</td>
</tr>
</tbody>
</table>

Table 14.18.: Description of the output structure of the list resource state mapping statement
**list resource state profile**

**Purpose**

*Purpose*  The purpose of the `list resource state profile` statement is to get a list of all currently defined resource state profiles.

**Syntax**

*Syntax*  The syntax for the `list resource state profile` statement is

```
list resource state profile
```

**Description**

*Description*  The `list resource state profile` statement gives you a list of all the defined Resource State Profiles.

**Output**

*Output*  This statement returns an output structure of type table.

**Output Description**  The data items of the output are described in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>The repository object Id</td>
</tr>
<tr>
<td>NAME</td>
<td>The object name</td>
</tr>
<tr>
<td>INITIAL_STATE</td>
<td>This field defines the initial state of the resource. This Resource State does not have to be present in the list of valid Resource States.</td>
</tr>
<tr>
<td>PRIVS</td>
<td>String containing the users privileges on the object</td>
</tr>
</tbody>
</table>

Table 14.19.: Description of the output structure of the list resource state profile statement
list schedule

Purpose

The purpose of the list schedule statement is to get a (partial) list of all defined schedules.

Syntax

The syntax for the list schedule statement is

\[
\text{list schedule schedulepath [ with EXPAND ]}
\]

EXPAND:

\[
\begin{align*}
\text{expand} &= \text{none} \\
| \text{expand} &= < ( id | id ) | \text{all} >
\end{align*}
\]

Description

The list schedule statement delivers a list with the specified schedule and all its children.

expand The expand option can be used to make the hierarchy visible at children level. This is done by specifying in the list the IDs of the nodes whose children are to be made visible. If none is specified as an expand option, only the level below the requested node is made visible.

Output

This statement returns an output structure of type table.

Output Description The data items of the output are described in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>The repository object Id</td>
</tr>
<tr>
<td>NAME</td>
<td>The object name</td>
</tr>
<tr>
<td>OWNER</td>
<td>The group owning the object</td>
</tr>
<tr>
<td>INTERVAL</td>
<td>The name of the interval belonging to the schedule</td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIME_ZONE</td>
<td>The time zone in which the schedule is to be calculated</td>
</tr>
<tr>
<td>ACTIVE</td>
<td>This field defines whether the schedule is marked as being active.</td>
</tr>
<tr>
<td>EFF_ACTIVE</td>
<td>This field defines whether the schedule is actually active. This can deviate from “active” due to the hierarchical organisation.</td>
</tr>
<tr>
<td>PRIVS</td>
<td>String containing the users privileges on the object</td>
</tr>
</tbody>
</table>

Table 14.20.: Description of the output structure of the list schedule statement
**list scheduled**

**Purpose**

The *list scheduled* command shows a list of the batches and jobs that will be started in the specified period.

**Syntax**

The syntax for the *list scheduled* statement is

```plaintext
list scheduled with LC_WITHITEM {, LC_WITHITEM}

LC_WITHITEM:
  endtime = datetime
  | filter = LC_FILTERTERM {or LC_FILTERTERM}
  |starttime = datetime
  | time zone = string

LC_FILTERTERM:
  LC_FILTERITEM {and LC_FILTERITEM}

LC_FILTERITEM:
  ( LC_FILTERTERM {or LC_FILTERTERM} )
  | job . identifier < cmpop | like | not like > RVALUE
  | name like string
  | not ( LC_FILTERTERM {or LC_FILTERTERM} )
  | owner in ( groupname {, groupname} )

RVALUE:
  expr ( string )
  | number
  | string
```

**Description**

The *list scheduled* statement provides a list of all the scheduled start times for batches and jobs.

**Output**

This statement returns an output structure of type table.
**Output Description**  The data items of the output are described in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>The repository object Id</td>
</tr>
<tr>
<td>SE_NAME</td>
<td>Name of the Scheduling Entity</td>
</tr>
<tr>
<td>SE_TYPE</td>
<td>Type of the Scheduling Entity (job or batch)</td>
</tr>
<tr>
<td>SE_ID</td>
<td>Id of the Scheduling Entity</td>
</tr>
<tr>
<td>SE_OWNER</td>
<td>Owner of the Scheduling Entity</td>
</tr>
<tr>
<td>SE_PRIVS</td>
<td>Privileges for the Scheduling Entity</td>
</tr>
<tr>
<td>SCE_NAME</td>
<td>Name of the schedule</td>
</tr>
<tr>
<td>SCE_ACTIVE</td>
<td>Flag that indicates if the schedule is active</td>
</tr>
<tr>
<td>EVT_NAME</td>
<td>Name of the event</td>
</tr>
<tr>
<td>STARTTIME</td>
<td>Start time</td>
</tr>
<tr>
<td>EXPECTED_FINAL_TIME</td>
<td>Expected date and time the job or batch will reach a final state</td>
</tr>
<tr>
<td>TIME_ZONE</td>
<td>The used time zone for date and time display</td>
</tr>
</tbody>
</table>

Table 14.21.: Description of the output structure of the list scheduled statement
list scheduled event

Purpose

The purpose of the list scheduled event is to get a list of all defined scheduled events.

Syntax

The syntax for the list scheduled event statement is

list scheduled event

Description

The list scheduled event statement gives you a list of all the defined Scheduled Events.

Output

This statement returns an output structure of type table.

Output Description

The data items of the output are described in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>The repository object Id</td>
</tr>
<tr>
<td>OWNER</td>
<td>The group owning the object</td>
</tr>
<tr>
<td>SCHEDULE</td>
<td>The Schedule that determines when the Scheduled Event is to take place</td>
</tr>
<tr>
<td>EVENT</td>
<td>The event that is triggered</td>
</tr>
<tr>
<td>ACTIVE</td>
<td>This flag indicates whether the Scheduled Event is labelled as being active.</td>
</tr>
<tr>
<td>EFF_ACTIVE</td>
<td>This flag indicates whether the Scheduled Event is actually active.</td>
</tr>
<tr>
<td>BROKEN</td>
<td>The Broken field can be used to check whether an error occurred when the job was submitted.</td>
</tr>
<tr>
<td>ERROR_CODE</td>
<td>If an error occurred while the job was being executed in the Time Scheduling, the returned error code is displayed in the Error_Code field. If no error occurred, this field remains empty.</td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERROR_MSG</td>
<td>If an error occurred while the job was being executed in the Time Scheduling, the returned error message is displayed in the Error Message field. If no error occurred, this field remains empty.</td>
</tr>
<tr>
<td>LAST_START</td>
<td>The last time the job is to be executed by the Scheduling System is shown here.</td>
</tr>
<tr>
<td>NEXT_START</td>
<td>The next scheduled time when the task is to be executed by the Scheduling System is shown here.</td>
</tr>
<tr>
<td>NEXT_CALC</td>
<td>The next time when a recalculation is to take place.</td>
</tr>
<tr>
<td>PRIVS</td>
<td>String containing the users privileges on the object.</td>
</tr>
<tr>
<td>BACKLOGHandled</td>
<td>The Backlog_Handling describes how events that should have been triggered following a downtime are to be handled.</td>
</tr>
<tr>
<td>SUSPEND_LIMIT</td>
<td>The Suspend Limit defines the delay after which a job is submitted in a suspended state.</td>
</tr>
<tr>
<td>EFFECTIVE_SUSPEND_LIMIT</td>
<td>The Suspend Limit defines the delay after which a job is submitted in a suspended state.</td>
</tr>
<tr>
<td>CALENDAR</td>
<td>This flag indicates whether calendar entries are created.</td>
</tr>
<tr>
<td>CALENDAR_HORIZON</td>
<td>The defined length of the period in days for which a calendar is created.</td>
</tr>
<tr>
<td>EFFECTIVE_CALENDAR_</td>
<td>The effective length of the period in days for which a calendar is created.</td>
</tr>
<tr>
<td>HORIZON</td>
<td></td>
</tr>
</tbody>
</table>

Table 14.22.: Description of the output structure of the list scheduled event statement
**list scope**

**Purpose**

The purpose of the `list scope` statement is to get a (partial) list of all defined scopes.

**Syntax**

The syntax for the `list scope` statement is

```
list < scope serverpath | jobserver serverpath > [ with EXPAND ]
```

**Description**

The `list scope` statement displays a list with the requested scope together with its children.

**expand**  The expand option can be used to make the hierarchy visible at children level. This is done by specifying in the list the IDs of the nodes whose children are to be made visible. If none is specified as an expand option, only the level below the requested node is made visible.

**Output**

This statement returns an output structure of type table.

**Output Description**  The data items of the output are described in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>The repository object Id</td>
</tr>
<tr>
<td>NAME</td>
<td>The object name</td>
</tr>
<tr>
<td>OWNER</td>
<td>The group owning the object</td>
</tr>
<tr>
<td>TYPE</td>
<td>The type of scope</td>
</tr>
<tr>
<td>IS_TERMINATE</td>
<td>This flag indicates whether a termination order exists.</td>
</tr>
</tbody>
</table>

Continued on next page
### Table 14.23.: Description of the output structure of the list scope statement

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HAS_ALTERED_CONFIG</td>
<td>The configuration on the server does not match the current configuration on the server.</td>
</tr>
<tr>
<td>IS_SUSPENDED</td>
<td>Indicates whether the scope is suspended</td>
</tr>
<tr>
<td>IS_ENABLED</td>
<td>The jobserver can only log on to the server if the Enable flag is set to YES.</td>
</tr>
<tr>
<td>IS_REGISTERED</td>
<td>Defines whether the jobserver has sent a register command</td>
</tr>
<tr>
<td>IS_CONNECTED</td>
<td>Indicates whether the jobserver is connected</td>
</tr>
<tr>
<td>STATE</td>
<td>This is the current state of the resource in this scope.</td>
</tr>
<tr>
<td>PID</td>
<td>The PID is the process identification number of the jobserver process on the respective host system.</td>
</tr>
<tr>
<td>NODE</td>
<td>The Node specifies the computer on which the jobserver is running. This field has a purely documentary character.</td>
</tr>
<tr>
<td>IDLE</td>
<td>The time that has elapsed since the last command. This only applies for jobservers.</td>
</tr>
<tr>
<td>NOPDELAY</td>
<td>The time that a jobserver waits for NOP</td>
</tr>
<tr>
<td>ONLINE_SERVER</td>
<td>The online_server column indicates whether the job server is configured as such. The value is always set to True for scopes.</td>
</tr>
<tr>
<td>ERRMSG</td>
<td>This is the most recently outputted error message.</td>
</tr>
<tr>
<td>SUBSCOPES</td>
<td>The number of scopes and jobservers that are present under this scope</td>
</tr>
<tr>
<td>RESOURCES</td>
<td>The resources present in this scope are displayed here.</td>
</tr>
<tr>
<td>PRIVS</td>
<td>String containing the users privileges on the object</td>
</tr>
<tr>
<td>IDPATH</td>
<td>The idpath is a list of parent IDs separated by periods. This value is mainly used for frontend programs.</td>
</tr>
</tbody>
</table>
list session

Purpose

The purpose of the list session statement is to get a list of connected sessions.  

Syntax

The syntax for the list session statement is

```
list session
```

Description

The list session statement gives you a list of the connected sessions.

Output

This statement returns an output structure of type table.

**Output Description** The data items of the output are described in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>THIS</td>
<td>The current session is indicated in this field by an asterisk (*)</td>
</tr>
<tr>
<td>SESSIONID</td>
<td>The internal server Id for the session</td>
</tr>
<tr>
<td>PORT</td>
<td>The TCP/IP port number at which the session is connected</td>
</tr>
<tr>
<td>START</td>
<td>Time when the connection was set up</td>
</tr>
<tr>
<td>TYPE</td>
<td>Type of connection: user, jobserver or job</td>
</tr>
<tr>
<td>USER</td>
<td>Name of the connecting user, jobserver or job (Job Id)</td>
</tr>
<tr>
<td>UID</td>
<td>Id of the user, jobserver or job</td>
</tr>
<tr>
<td>IP</td>
<td>IP address of the connecting sessions</td>
</tr>
<tr>
<td>TXID</td>
<td>Number of the last transaction that was executed by the session</td>
</tr>
<tr>
<td>IDLE</td>
<td>The number of seconds since the last statement from a session</td>
</tr>
</tbody>
</table>

*Continued on next page*
Continued from previous page

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATE</td>
<td>The state of the session. This is one of the following: IDLE (no activity), QUEUED (statement is waiting to be executed), ACTIVE (statement is being executed), COMMITTING (changes to a write transaction are being written), CONNECTED (not yet authenticated).</td>
</tr>
<tr>
<td>TIMEOUT</td>
<td>The idle time after which the session is automatically disconnected</td>
</tr>
<tr>
<td>INFORMATION</td>
<td>Additional information about the session (optional)</td>
</tr>
<tr>
<td>STATEMENT</td>
<td>The statement that is currently being executed</td>
</tr>
<tr>
<td>WAIT</td>
<td>The wait flag shows if the session is waiting (for a lock).</td>
</tr>
</tbody>
</table>

Table 14.24.: Description of the output structure of the list session statement
**list trigger**

**Purpose**

The purpose of the `list trigger` statement is to get a list of defined triggers.

**Syntax**

The syntax for the `list trigger` statement is

```
list trigger [ < all | limit integer > ]
```

```
list trigger for folderpath [ < all | limit integer > ]
```

```
list trigger of folderpath [ < all | limit integer > ]
```

```
list trigger for CT_OBJECT [ < all | limit integer > ]
```

```
CT_OBJECT:
    job definition folderpath
    | named resource identifier {. identifier}
    | object monitor objecttypename
    | resource identifier {. identifier} in < folderpath | serverpath >
```

**Description**

The `list trigger` statement gives you a list of all the defined triggers.

**Output**

This statement returns an output structure of type table.

**Output Description** The data items of the output are described in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>The repository object Id</td>
</tr>
<tr>
<td>NAME</td>
<td>The object name</td>
</tr>
<tr>
<td>OBJECT_TYPE</td>
<td>The type of object in which the trigger is defined</td>
</tr>
</tbody>
</table>

*Continued on next page*
<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OBJECT_SUBTYPE</td>
<td>The subtype of the object in which the trigger is defined</td>
</tr>
<tr>
<td>OBJECT_NAME</td>
<td>Full path name of the object in which the trigger is defined</td>
</tr>
<tr>
<td>ACTIVE</td>
<td>The flag indicates whether the trigger is currently active.</td>
</tr>
<tr>
<td>ACTION</td>
<td>Type of triggered action: SUBMIT or RERUN</td>
</tr>
<tr>
<td>STATES</td>
<td>A list of states that cause the trigger to be activated</td>
</tr>
<tr>
<td>SUBMIT_TYPE</td>
<td>The object type that is submitted when the trigger is activated</td>
</tr>
<tr>
<td>SUBMIT_NAME</td>
<td>Name of the job definition that is submitted</td>
</tr>
<tr>
<td>SUBMIT_SE_OWNER</td>
<td>The owner of the object that is submitted</td>
</tr>
<tr>
<td>SUBMIT_PRIVS</td>
<td>The privileges for the object that is to be submitted</td>
</tr>
<tr>
<td>MAIN_TYPE</td>
<td>Type of main job (job/batch)</td>
</tr>
<tr>
<td>MAIN_NAME</td>
<td>Name of the main job</td>
</tr>
<tr>
<td>MAIN_SE_OWNER</td>
<td>Owner of the main job</td>
</tr>
<tr>
<td>MAIN_PRIVS</td>
<td>Privileges for the main job</td>
</tr>
<tr>
<td>PARENT_TYPE</td>
<td>Type of parent job (job/batch)</td>
</tr>
<tr>
<td>PARENT_NAME</td>
<td>Name of the parent job</td>
</tr>
<tr>
<td>PARENT_SE_OWNER</td>
<td>Owner of the parent job</td>
</tr>
<tr>
<td>PARENT_PRIVS</td>
<td>Privileges for the parent job</td>
</tr>
<tr>
<td>TRIGGER_TYPE</td>
<td>The trigger type that describes when it is activated</td>
</tr>
<tr>
<td>MASTER</td>
<td>Indicates whether the trigger submitted a master or a child</td>
</tr>
<tr>
<td>IS_INVERSE</td>
<td>In case of an inverse trigger, the trigger is regarded to belong to the triggered job. The trigger can be regarded as some kind of callback function. This flag has no effects on the trigger’s behaviour.</td>
</tr>
<tr>
<td>SUBMIT_OWNER</td>
<td>The owner group that is used with the Submitted Entity</td>
</tr>
<tr>
<td>IS_CREATE</td>
<td>Indicates whether the trigger reacts to create events</td>
</tr>
<tr>
<td>Field</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>IS_CHANGE</td>
<td>Indicates whether the trigger reacts to change events</td>
</tr>
<tr>
<td>IS_DELETE</td>
<td>Indicates whether the trigger reacts to delete events</td>
</tr>
<tr>
<td>IS_GROUP</td>
<td>Indicates whether the trigger handles the events as a group</td>
</tr>
<tr>
<td>MAX_RETRY</td>
<td>The maximum number of trigger activations in a single Submitted Entity</td>
</tr>
<tr>
<td>SUSPEND</td>
<td>Specifies whether the submitted object is suspended</td>
</tr>
<tr>
<td>RESUME_AT</td>
<td>Time of the automatic resume</td>
</tr>
<tr>
<td>RESUME_IN</td>
<td>Number of time units until the automatic resume</td>
</tr>
<tr>
<td>RESUME_BASE</td>
<td>Specified time unit for RESUME_IN</td>
</tr>
<tr>
<td>WARN</td>
<td>Specifies whether a warning has to be given when the activation limit is reached</td>
</tr>
<tr>
<td>LIMIT_STATE</td>
<td>This field specifies which state the triggering job acquires if the fire limit is reached. If the triggering job has a final state already, this specification is ignored. If the value is NONE, no state change takes place.</td>
</tr>
<tr>
<td>CONDITION</td>
<td>Conditional expression to define the trigger condition</td>
</tr>
<tr>
<td>CHECK_AMOUNT</td>
<td>The amount of CHECK_BASE units for checking the condition in the case of non-synchronised triggers</td>
</tr>
<tr>
<td>CHECK_BASE</td>
<td>Units for the CHECK_AMOUNT</td>
</tr>
<tr>
<td>PARAMETERS</td>
<td>The parameter clause can be used to define parameters for the job or batch that is to be submitted. The names of the parameters are taken over as such. The expressions are valuated in the context of the triggering job or batch.</td>
</tr>
<tr>
<td>PRIVS</td>
<td>String containing the users privileges on the object</td>
</tr>
<tr>
<td>TAG</td>
<td>Units for the CHECK_AMOUNT</td>
</tr>
</tbody>
</table>

Continued on next page
Continued from previous page

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMMENT</td>
<td>Comment if defined</td>
</tr>
<tr>
<td>COMMENTTYPE</td>
<td>Type of comment if a comment is defined</td>
</tr>
</tbody>
</table>

Table 14.25.: Description of the output structure of the list trigger statement
**list user**

**Purpose**

The purpose of the *list user* statement is to get a list of all defined users.  

**Syntax**

The syntax for the *list user* statement is  

```
list user
```

**Description**

The *list user* statement gives you a list of all the defined users.  

**Output**

This statement returns an output structure of type table.  

**Output Description**  
The data items of the output are described in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>The repository object Id</td>
</tr>
<tr>
<td>NAME</td>
<td>The object name</td>
</tr>
<tr>
<td>IS_ENABLED</td>
<td>Flag that shows whether the user is allowed to log on</td>
</tr>
<tr>
<td>DEFAULT_GROUP</td>
<td>The default group of the users who are being used by the owners of the object</td>
</tr>
</tbody>
</table>

*Continued on next page*
Field | Description |
--- | --- |
CONNECTION_TYPE | Indicates which security level of a connection is required. |
| | 1. **plain** – Every kind of connection is permitted |
| | 2. **ssl** – Only SSL-connections are permitted |
| | 3. **ssl_auth** – Only SSL-connections with client authentication are permitted |
PRIVS | String containing the users privileges on the object |

Table 14.26.: Description of the output structure of the list user statement
15. move commands
move folder

Purpose

Purpose  The purpose of the move folder statement is to rename the folder and/or to move it to some other place in the folder hierarchy.

Syntax

Syntax  The syntax for the move folder statement is

  move folder folderpath to folderpath

Description

Description  The move folder command either moves the specified folder to somewhere else or renames it.

Output

Output  This statement returns a confirmation of a successful operation.
move job definition

Purpose

The purpose of the move job definition statement is to rename a scheduling entity object, and/or move it into some other folder.

Syntax

The syntax for the move job definition statement is

move job definition folderpath to folderpath

Description

The move job definition command moves the specified job definition to the specified folder. If the destination folder does not exist, the last part of the fully qualified name is interpreted as being the new name for the job definition. The relationships to other objects are not changed.

Output

This statement returns a confirmation of a successful operation.
**move named resource**

**Purpose**

*Purpose* The purpose of the *move named resource* statement is to rename the named resource and/or to move the resource into another category.

**Syntax**

*Syntax* The syntax for the *move named resource* statement is

```
move named resource identifier { . identifier } to identifier { . identifier }
```

**Description**

*Description* The *move named resource* statement is used to rename a Named Resource or to reorganise categories. If a Named Resource is moved, the specified destination has to be a category or it must not exist and its parent must be a category.

**Output**

*Output* This statement returns a confirmation of a successful operation.
move schedule

Purpose

The purpose of the move schedule statement is to rename and/or to move the schedule to some other place in the hierarchy.

Syntax

The syntax for the move schedule statement is

move schedule schedulepath . schedulename to schedulepath

Description

The move schedule command either moves the specified schedule to somewhere else and/or renames it.

Output

This statement returns a confirmation of a successful operation.
move scope

Purpose

Purpose The purpose of the move scope statement is to rename a scope and/or to move it to some other place within the scope hierarchy.

Syntax

Syntax The syntax for the move scope statement is

\[ \text{move} < \text{scope serverpath} | \text{jobserver serverpath} > \text{ to serverpath} \]

Description

Description The move scope command either moves the specified scope to somewhere else and/or renames it.

Output

Output This statement returns a confirmation of a successful operation.
16. multicommend commands
multicommand

Purpose

Purpose
This statement is used to control the behaviour of the SDMS Server.

Syntax

Syntax
The syntax for the multicommand statement is

```
begin multicommand commandlist end multicommand

begin multicommand commandlist end multicommand rollback
```

Description

Description
The multicommands allow multiple SDMS commands to be executed together, i.e. in one transaction. This ensures that either all the statements are executed without any errors or nothing happens at all. Not only that, but the transaction is not interrupted by other write transactions.
If the rollback keyword is specified, the transaction is undone at the end of the processing. This means that you can test whether the statements can be correctly processed (technically speaking).

Output

Output
This statement returns a confirmation of a successful operation.
17. register commands
**register**

**Purpose**

The purpose of the `register` statement is to notify the server that the jobserver is ready to process jobs.

**Syntax**

The syntax for the `register` statement is

```
register serverpath . servername
with pid = pid [ suspend ]
```

**Description**

The first form is used by the operator to enable jobs to be executed by the specified jobserver. The second form is used by the jobserver itself to notify the server that it is ready to execute jobs. Jobs are scheduled for this jobserver (unless it is suspended) regardless of whether the server is connected or not. Refer to the 'deregister' statement on page 180.

- **pid** The pid option provides the server with information about the jobserver’s process Id at operating level.

- **suspend** The suspend option causes the jobserver to be transferred to a suspended state.

**Output**

This statement returns a confirmation of a successful operation.
18. rename commands
rename environment

Purpose

Purpose The purpose of the *rename environment* statement is to give the specified environment another name.

Syntax

Syntax The syntax for the *rename environment* statement is

```
rename environment *environmentname* to *environmentname*
```

Description

Description The *rename environment* statement is used to rename environments. Renaming an environment does not have any effect on the functionality and is only for purposes of clarity.

Output

Output This statement returns a confirmation of a successful operation.
rename event

Purpose

The purpose of the rename event is to give the specified event another name.  

Syntax

The syntax for the rename event statement is

\[
\text{rename event } \text{eventname to } \text{eventname}
\]

Description

The rename event statement is used to give a specified event a different name.

Output

This statement returns a confirmation of a successful operation.
rename exit state definition

Purpose

*Purpose*  The purpose of the *rename exit state definition* statement is to give the specified exit state definition another name.

Syntax

*Syntax*  The syntax for the *rename exit state definition* statement is

\[
\text{rename exit state definition statename to statename}
\]

Description

*Description*  The *rename exit state definition* statement is used to rename Exit State Definitions. Renaming an Exit State Definition does not have any effect on the functionality and is only for purposes of clarity.

Output

*Output*  This statement returns a confirmation of a successful operation.
rename exit state mapping

Purpose

The purpose of the `rename exit state mapping` statement is to give the specified mapping another name.

Syntax

The syntax for the `rename exit state mapping` statement is

```
rename exit state mapping mappingname to profilename
```

Description

The `rename exit state mapping` statement is used to rename Exit State Mappings. Renaming an Exit State Mapping does not have any effect on the functionality and is only for purposes of clarity.

Output

This statement returns a confirmation of a successful operation.
**rename exit state profile**

**Purpose**

*Purpose*  The purpose of the `rename exit state profile` statement is to give the specified profile another name.

**Syntax**

*Syntax*  The syntax for the `rename exit state profile` statement is

```
rename exit state profile  profilename to profilename
```

**Description**

*Description*  The `rename exit state profile` statement is used to rename Exit State Profiles. Renaming the Exit State Profiles does not have any effect on the functionality and is only for purposes of clarity.

**Output**

*Output*  This statement returns a confirmation of a successful operation.
rename exit state translation

Purpose

The purpose of the *rename exit state translation* statement is to give the specified
exit state translation another name.

Syntax

The syntax for the *rename exit state translation* statement is

```plaintext
rename exit state translation transname to transname
```

Description

The *rename exit state translation* statement is used to rename Exit State Transla-
tions. Renaming an Exit State Translation does not have any effect on the function-
ality and is only for purposes of clarity.

Output

This statement returns a confirmation of a successful operation.
rename folder

Purpose
Purpose The purpose of the rename folder statement is to give a folder another name.

Syntax
Syntax The syntax for the rename folder statement is

    rename folder folderpath to foldername

Description
Description The rename folder command renames the specified folder. This is done within the same parent folder. If an object with the new name already exists, this triggers an error message.

Output
Output This statement returns a confirmation of a successful operation.
rename footprint

Purpose

The purpose of the `rename footprint` statement is to give the specified footprint another name.

Syntax

The syntax for the `rename footprint` statement is

```
rename footprint footprintname to footprintname
```

Description

The `rename footprint` statement is used to give a specified footprint a different name.

Output

This statement returns a confirmation of a successful operation.
rename group

Purpose

Purpose The purpose of the rename group statement is to change the name of a group without affecting any other properties.

Syntax

Syntax The syntax for the rename group statement is

```
rename group groupname to groupname
```

Description

Description The rename group statement is used to rename groups. Renaming a group does not have any effect on the functionality and is only for purposes of clarity.

Output

Output This statement returns a confirmation of a successful operation.
rename interval

Purpose

The purpose of the rename interval statement is to give the specified interval another name.

Syntax

The syntax for the rename interval statement is

rename interval intervalname to intervalname

Description

The rename interval statement is used to give a specified interval a different name.

Output

This statement returns a confirmation of a successful operation.
rename job definition

Purpose

Purpose  The purpose of the `rename job definition` statement is to give the job definition another name.

Syntax

Syntax  The syntax for the `rename job definition` statement is

```
rename job definition folderpath to jobname
```

Description

Description  The `rename job definition` command renames the specified job definition.

Output

Output  This statement returns a confirmation of a successful operation.
rename named resource

Purpose

The purpose of the rename named resource statement is to give a named resource another name.

Syntax

The syntax for the rename named resource statement is

```
rename named resource identifier \{ identifier \} to resourcename
```

Description

The rename named resource statement is used to rename a Named Resource.

Output

This statement returns a confirmation of a successful operation.
rename resource state definition

Purpose

Purpose The purpose of the rename resource state definition statement is to rename the resource state.

Syntax

Syntax The syntax for the rename resource state definition statement is


rename resource state definition statename to statename

Description

Description The rename resource state definition statement is used to rename Resource State Definitions. Renaming a Resource State Definition does not have any effect on the functionality and is only for purposes of clarity.

Output

Output This statement returns a confirmation of a successful operation.
rename resource state mapping

Purpose

The purpose of the rename resource state mapping statement is to give the specified mapping a new name.

Syntax

The syntax for the rename resource state mapping statement is

```
rename resource state mapping mappingname to profilename
```

Description

The rename resource state mapping statement is used to rename Resource State Mappings. Renaming a Resource State Mapping does not have any effect on the functionality and is only for purposes of clarity.

Output

This statement returns a confirmation of a successful operation.

rename resource state profile

Purpose

Purpose The purpose of the rename resource state profile is to give the specified resource state profile a new name.

Syntax

Syntax The syntax for the rename resource state profile statement is

```
rename resource state profile profilename to profilename
```

Description

Description The rename resource state profile statement is used to rename Resource State Profiles. Renaming a Resource State Profile does not have any effect on the functionality and is only for purposes of clarity.

Output

Output This statement returns a confirmation of a successful operation.
rename schedule

Purpose
The purpose of the rename schedule statement is to give a schedule another name.

Syntax
The syntax for the rename schedule statement is

rename schedule schedulepath . schedulename to schedulename

Description
The rename schedule command renames the specified schedule.

Output
This statement returns a confirmation of a successful operation.
rename scope

Purpose
The purpose of the rename scope statement is to give a scope another name.

Syntax
The syntax for the rename scope statement is

```plaintext
rename < scope serverpath | jobserver serverpath > to scopename
```

Description
The rename scope command renames the specified scope.

Output
This statement returns a confirmation of a successful operation.
rename trigger

Purpose

The purpose of the `rename trigger` statement is to give the specified trigger another name.

Syntax

The syntax for the `rename trigger` statement is

```
rename trigger triggername on TRIGGEROBJECT [ < noinverse | inverse > ] to triggername
```

TRIGGEROBJECT:

```
resource identifier {. identifier} in folderpath
| job definition folderpath
| named resource identifier {. identifier}
| object monitor objecttypename
| resource identifier {. identifier} in serverpath
```

Description

The `rename trigger` statement is used to rename the trigger. Renaming a trigger does not have any effect on the functionality and is only for purposes of clarity.

Output

This statement returns a confirmation of a successful operation.
rename user

Purpose

Purpose   The purpose of the rename user statement is to change the name of a user without altering any other of its properties.

Syntax

Syntax   The syntax for the rename user statement is

   rename user username to username

Description

Description   The rename user statement is used to rename users. Renaming a user does not have any effect on the functionality and is only for purposes of clarity.

Output

Output   This statement returns a confirmation of a successful operation.
19. resume commands
**resume**

**Purpose**

*Purpose*  The purpose of the *resume* statement is to reactivate the jobserver. See also the *suspend* statement on page 418.

**Syntax**

*Syntax*  The syntax for the *resume* statement is

```
resume serverpath
```

**Description**

*Description*  The *resume* statement is used to reactivate a jobserver.

**Output**

*Output*  This statement returns a confirmation of a successful operation.
20. select commands
**select**

**Purpose**

The purpose of the `select` statement is to enable the user to issue (almost) arbitrary queries to the underlying RDBMS.

**Syntax**

The syntax for the `select` statement is

```
select-statement [ with WITHITEM {, WITHITEM} ]
```

**Description**

The `select` statement allows practically any number of database select statements to be executed by the Scheduling Server. Refer to the documentation of the database system you are using for information about the syntax that is used for the `select` statement.

Since executing arbitrary `select` statements generally represents a vulnerability, administrator privileges are required for this statement. This means that only users belonging to the `ADMIN` group are allowed to use this statement.

Using the `withitems` causes IDs to be translated into names. This function is available for all hierarchically structured object types since this operation is not always easy to perform using SQL means.

If the optional keyword `quoted` is specified, all elements will be quoted. This is especially useful when generating statements from the repository.

It is also possible to sort the set of results after replacing the IDs. The columns that are to be used for sorting are addressed according to their position in the set of results (zero-based, i.e. the first column has the number 0).

**Output**

This statement returns a confirmation of a successful operation.
21. set commands
**set parameter**

**Purpose**

*Purpose* The purpose of the *set parameter* statement is to set the value of the specified parameters within the context of the requesting job, respectively the specified job.

**Syntax**

*Syntax* The syntax for the *set parameter* statement is

```
set parameter parametername = string [, parametername = string]
```

```
set parameter < on | of > jobid parametername = string [, parametername = string] [ with comment = string ]
```

```
set parameter < on | of > jobid parametername = string [, parametername = string] identified by string [ with comment = string ]
```

**Description**

*Description* The *set parameter* statements can be used to set jobs or user parameter values in the context of the job.

If the *identified by* option is specified, the parameter is only set if the pair *jobid* and *string* would allow a logon.

**Output**

*Output* This statement returns a confirmation of a successful operation.
22. show commands
**show comment**

**Purpose**

The purpose of the *show comment* statement is to show the comment for the specified object.

**Syntax**

The syntax for the *show comment* statement is

```
show comment on OBJECTURL
```

**OBJECTURL:**

- `distribution distributionname for pool identifier [. identifier] in serverpath`
- `environment environmentname`
- `exit state definition statename`
- `exit state mapping mappingname`
- `exit state profile profilename`
- `exit state translation transname`
- `event eventname`
- `resource identifier [. identifier] in folderpath`
- `folder folderpath`
- `footprint footprintname`
- `group groupname`
- `interval intervalname`
- `job definition folderpath`
- `job jobid`
- `named resource identifier [. identifier]`
- `parameter parametername of PARAM_LOC`
- `resource state definition statename`
- `resource state mapping mappingname`
- `resource state profile profilename`
- `scheduled event schedulepath . eventname`
- `schedule schedulepath`
- `resource identifier [. identifier] in serverpath`
- `< scope serverpath | observer serverpath >`
- `trigger triggername on TRIGGEROBJECT [ < noinverse | inverse > ]`
- `user username`
Description

The *show comment* statement is used to display the saved comment for the specified object. If no comment on the object exists, this is *not* regarded as being an error; instead, an empty output structure is created and returned. This empty output structure naturally corresponds to the output structure described below, so that it can be easily evaluated by programs without any exception handling.

Output

This statement returns an output structure of type table.

**Output Description** The data items of the output are described in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>System-wide unique object number</td>
</tr>
<tr>
<td>TAG</td>
<td>The comment tag is a headline of the comment block. It is an optional field.</td>
</tr>
<tr>
<td>COMMENT</td>
<td>The comment on the specified object</td>
</tr>
<tr>
<td>COMMENTTYPE</td>
<td>Type of comment, text or URL</td>
</tr>
<tr>
<td>CREATOR</td>
<td>Name of the user who created this pool</td>
</tr>
<tr>
<td>CREATE_TIME</td>
<td>The creation time</td>
</tr>
<tr>
<td>CHANGER</td>
<td>Name of the last user who modified this pool</td>
</tr>
</tbody>
</table>

*Continued on next page*
<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHANGE_TIME</td>
<td>Time of the last modification</td>
</tr>
<tr>
<td>PRIVS</td>
<td>Abbreviation for the privileges for this object held by the requesting user</td>
</tr>
</tbody>
</table>

Table 22.1.: Description of the output structure of the show comment statement
**show environment**

**Purpose**

The purpose of the *show environment* statement is to get detailed information about the specified environment.

**Syntax**

The syntax for the *show environment* statement is

```
show environment environmentname [ with EXPAND ]
```

**Description**

The *show environment* statement gives you detailed information about the specified environment.

*expand*  
Since the number of job definitions in the table JOB_DEFINITIONS can become very large, by default they are not all displayed. If the option *expand = all* is used, all the job definitions as well as their parent folder and the folder hierarchy are outputted. Individual paths in the hierarchy can be selected by specifying individual (folder) IDs.

**Output**

This statement returns an output structure of type record.

**Output Description**  
The data items of the output are described in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>The repository object Id</td>
</tr>
<tr>
<td>NAME</td>
<td>The name of the environment</td>
</tr>
<tr>
<td>COMMENT</td>
<td>Comment if defined</td>
</tr>
<tr>
<td>COMMENTTYPE</td>
<td>Type of comment if a comment is defined</td>
</tr>
</tbody>
</table>

Continued on next page
**Field** | **Description**
--- | ---
**CREATOR** | Name of the user created the object
**CREATE_TIME** | Date and time of object creation
**CHANGER** | Name of the last user changed the object
**CHANGE_TIME** | Date and time of object change
**PRIVS** | String containing the users privileges on the object
**RESOURCES** | Table of static resources that form this environment
See also table 22.3 on page 314
**JOB_DEFINITIONS** | Table of jobs and folders that use this environment
See also table 22.4 on page 314

Table 22.2.: Description of the output structure of the show environment statement

**RESOURCES** | The layout of the RESOURCES table is shown in the table below.

<table>
<thead>
<tr>
<th><strong>Field</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ID</strong></td>
<td>The repository object Id</td>
</tr>
<tr>
<td><strong>NR_NAME</strong></td>
<td>Full path name of static Named Resources</td>
</tr>
<tr>
<td><strong>CONDITION</strong></td>
<td>The condition that has to be fulfilled for the allocation</td>
</tr>
<tr>
<td><strong>PRIVS</strong></td>
<td>String containing the users privileges on the object</td>
</tr>
</tbody>
</table>

Table 22.3.: Description of the output structure of the show environment subtable

**JOB_DEFINITIONS** | The layout of the JOB_DEFINITIONS table is shown in the table below.

<table>
<thead>
<tr>
<th><strong>Field</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ID</strong></td>
<td>The repository object Id</td>
</tr>
<tr>
<td><strong>SE_PATH</strong></td>
<td>Full folder path name of job definitions or folders</td>
</tr>
<tr>
<td><strong>TYPE</strong></td>
<td>The object type. The possible values are FOLDER and JOB_DEFINITION.</td>
</tr>
</tbody>
</table>

*Continued on next page*
Continued from previous page

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENV</td>
<td>An asterisk indicates that the current environment was specified here.</td>
</tr>
<tr>
<td>HAS_CHILDREN</td>
<td>True means that there are more environment users further down the tree.</td>
</tr>
<tr>
<td>PRIVS</td>
<td>String containing the users privileges on the object</td>
</tr>
</tbody>
</table>

Table 22.4.: Description of the output structure of the show environment subtable
show event

Purpose

Purpose The purpose of the show event statement is to get detailed information about the specified event.

Syntax

Syntax The syntax for the show event statement is

    show event eventname

Description

Description The show event statement gives you detailed information about the specified event.

Output

Output This statement returns an output structure of type record.

Output Description The data items of the output are described in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>The repository object Id</td>
</tr>
<tr>
<td>NAME</td>
<td>Name of the show event</td>
</tr>
<tr>
<td>OWNER</td>
<td>The group owning the object</td>
</tr>
<tr>
<td>SCHEDULING_ENTITY</td>
<td>Batch or job that is submitted when this event occurs</td>
</tr>
<tr>
<td>CREATOR</td>
<td>Name of the user created the object</td>
</tr>
<tr>
<td>CREATE_TIME</td>
<td>Date and time of object creation</td>
</tr>
<tr>
<td>CHANGER</td>
<td>Name of the last user changed the object</td>
</tr>
<tr>
<td>CHANGE_TIME</td>
<td>Date and time of object change</td>
</tr>
<tr>
<td>PARAMETERS</td>
<td>Parameters that are used when submitting the job or batch</td>
</tr>
<tr>
<td></td>
<td>See also table 22.6 on page 317</td>
</tr>
<tr>
<td>PRIVS</td>
<td>String containing the users privileges on the object</td>
</tr>
</tbody>
</table>

Continued on next page
Table 22.5.: Description of the output structure of the show event statement

**PARAMETERS** The layout of the PARAMETERS table is shown in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>The repository object Id</td>
</tr>
<tr>
<td>KEY</td>
<td>Name of the parameter</td>
</tr>
<tr>
<td>VALUE</td>
<td>Value of the parameter</td>
</tr>
</tbody>
</table>

Table 22.6.: Description of the output structure of the show event subtable
**show exit state definition**

**Purpose**

*Purpose* The purpose of the `show exit state definition` statement is to get detailed information about the specified exit state definition.

**Syntax**

*Syntax* The syntax for the `show exit state definition` statement is

```plaintext
show exit state definition statename
```

**Description**

*Description* The `show exit state definition` statement gives you detailed information about the specified Exit State Definition.

**Output**

*Output* This statement returns an output structure of type record.

**Output Description** The data items of the output are described in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>The repository object Id</td>
</tr>
<tr>
<td>NAME</td>
<td>Name of the Exit State Definition</td>
</tr>
<tr>
<td>COMMENT</td>
<td>Comment if defined</td>
</tr>
<tr>
<td>COMMENTTYPE</td>
<td>Type of comment if a comment is defined</td>
</tr>
<tr>
<td>CREATOR</td>
<td>Name of the user created the object</td>
</tr>
<tr>
<td>CREATE_TIME</td>
<td>Date and time of object creation</td>
</tr>
<tr>
<td>CHANGER</td>
<td>Name of the last user changed the object</td>
</tr>
<tr>
<td>CHANGE_TIME</td>
<td>Date and time of object change</td>
</tr>
<tr>
<td>PRIVS</td>
<td>String containing the users privileges on the object</td>
</tr>
</tbody>
</table>

Table 22.7.: Description of the output structure of the show exit state definition statement
show exit state mapping

Purpose

The purpose of the `show exit state mapping` statement is to get detailed information about the specified mapping.

Syntax

The syntax for the `show exit state mapping` statement is

```
show exit state mapping mappingname
```

Description

The `show exit state mapping` statement gives you detailed information about the specified mapping.

Output

This statement returns an output structure of type record.

Output Description

The data items of the output are described in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>The repository object Id</td>
</tr>
<tr>
<td>NAME</td>
<td>The object name</td>
</tr>
<tr>
<td>COMMENT</td>
<td>A comment that can be freely selected by the user</td>
</tr>
<tr>
<td>COMMENTTYPE</td>
<td>Type of comment if a comment is defined</td>
</tr>
<tr>
<td>CREATOR</td>
<td>Name of the user created the object</td>
</tr>
<tr>
<td>CREATE_TIME</td>
<td>Date and time of object creation</td>
</tr>
<tr>
<td>CHANGER</td>
<td>Name of the last user changed the object</td>
</tr>
<tr>
<td>CHANGE_TIME</td>
<td>Date and time of object change</td>
</tr>
<tr>
<td>PRIVS</td>
<td>String containing the users privileges on the object</td>
</tr>
<tr>
<td>RANGES</td>
<td>The assignment of the respective value ranges shown in a table</td>
</tr>
</tbody>
</table>

See also table 22.9 on page 320

Table 22.8.: Description of the output structure of the show exit state mapping statement
RANGES  The layout of the RANGES table is shown in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECR_START</td>
<td>Minimum limit of the range (inclusive)</td>
</tr>
<tr>
<td>ECR_END</td>
<td>Maximum limit of the range (inclusive)</td>
</tr>
<tr>
<td>ESD_NAME</td>
<td>Name of the Exit State to which this area is mapped</td>
</tr>
</tbody>
</table>

Table 22.9.: Description of the output structure of the show exit state mapping subtable
show exit state profile

Purpose

The purpose of the *show exit state profile* statement is to get detailed information about the specified profile.

Syntax

The syntax for the *show exit state profile* statement is

```plaintext
show exit state profile profilename
```

Description

The *show exit state profile* statement gives you detailed information about the specified profile.

Output

This statement returns an output structure of type record.

Output Description

The data items of the output are described in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>The repository object Id</td>
</tr>
<tr>
<td>NAME</td>
<td>The object name</td>
</tr>
<tr>
<td>DEFAULT_ESM_NAME</td>
<td>The default Exit State Mapping is active if the job itself does not define something else.</td>
</tr>
<tr>
<td>IS_VALID</td>
<td>Flag displayed showing the validity of this Exit State Profile</td>
</tr>
<tr>
<td>COMMENT</td>
<td>Comment if defined</td>
</tr>
<tr>
<td>COMMENTTYPE</td>
<td>Type of comment if a comment is defined</td>
</tr>
<tr>
<td>CREATOR</td>
<td>Name of the user created the object</td>
</tr>
<tr>
<td>CREATE_TIME</td>
<td>Date and time of object creation</td>
</tr>
<tr>
<td>CHANGER</td>
<td>Name of the last user changed the object</td>
</tr>
<tr>
<td>CHANGE_TIME</td>
<td>Date and time of object change</td>
</tr>
<tr>
<td>PRIVS</td>
<td>String containing the users privileges on the object</td>
</tr>
</tbody>
</table>

*Continued on next page*
Continued from previous page

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATES</td>
<td>Table contains Exit States that are valid for this profile</td>
</tr>
<tr>
<td></td>
<td>See also table 22.11 on page 322</td>
</tr>
</tbody>
</table>

Table 22.10.: Description of the output structure of the show exit state profile statement

STATES  The layout of the STATES table is shown in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>The repository object Id</td>
</tr>
<tr>
<td>PREFERENCE</td>
<td>The preference for controlling the connection of the Child Exit States</td>
</tr>
<tr>
<td>TYPE</td>
<td>Indicates whether the state is FINAL, PENDING or RESTARTABLE</td>
</tr>
<tr>
<td>ESD_NAME</td>
<td>Name of the Exit State Definition</td>
</tr>
<tr>
<td>IS_UNREACHABLE</td>
<td>Indicates that this Exit State is used when a job is unreachable</td>
</tr>
<tr>
<td>IS_DISABLED</td>
<td>Normally, a disabled job will take on the same Exit State as an empty batch. However, if a FINAL State is marked as Disabled, the default behaviour is disabled, and a disabled job will take on that state.</td>
</tr>
<tr>
<td>IS_BROKEN</td>
<td>Indicates that this Exit State is used when a job is broken</td>
</tr>
<tr>
<td>IS_BATCH_DEFAULT</td>
<td>Indicates that this Exit State is used when a batch or milestone does not have any children</td>
</tr>
<tr>
<td>IS_DEPENDENCY_DEFAULT</td>
<td>Indicates that this Exit State is used if the state selection DEFAULT was selected in the Dependency Definition</td>
</tr>
</tbody>
</table>

Table 22.11.: Description of the output structure of the show exit state profile subtable

322
**show exit state translation**

**Purpose**

The purpose of the `show exit state translation` statement is to get detailed information about the specified exit state translation.

**Syntax**

The syntax for the `show exit state translation` statement is

```plaintext
show exit state translation transname
```

**Description**

The `show exit state translation` statement gives you detailed information about the specified Exit State Translation.

**Output**

This statement returns an output structure of type record.

**Output Description**  The data items of the output are described in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>The repository object id</td>
</tr>
<tr>
<td>NAME</td>
<td>Name of the Exit State Translation</td>
</tr>
<tr>
<td>COMMENT</td>
<td>Comment if defined</td>
</tr>
<tr>
<td>COMMENTTYPE</td>
<td>Type of comment if a comment is defined</td>
</tr>
<tr>
<td>CREATOR</td>
<td>Name of the user created the object</td>
</tr>
<tr>
<td>CREATE_TIME</td>
<td>Date and time of object creation</td>
</tr>
<tr>
<td>CHANGER</td>
<td>Name of the last user changed the object</td>
</tr>
<tr>
<td>CHANGE_TIME</td>
<td>Date and time of object change</td>
</tr>
<tr>
<td>PRIVS</td>
<td>String containing the users privileges on the object</td>
</tr>
<tr>
<td>TRANSLATION</td>
<td>Table of Exit State translations from child to parent</td>
</tr>
</tbody>
</table>

| See also table 22.13 on page 324 |

Table 22.12.: Description of the output structure of the `show exit state translation` statement
The layout of the TRANSLATION table is shown in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FROM_ESD_NAME</td>
<td>Child exit state</td>
</tr>
<tr>
<td>TO_ESD_NAME</td>
<td>Parent exit state</td>
</tr>
</tbody>
</table>

Table 22.13.: Description of the output structure of the show exit state translation subtable
**show folder**

**Purpose**

The purpose of the *show folder* statement is to get detailed information about the specified folder.

**Syntax**

The syntax for the *show folder* statement is

```
show folder folderpath
```

**Description**

The *show folder* statement gives you detailed information about the specified folder.

**Output**

This statement returns an output structure of type record.

**Output Description** The data items of the output are described in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>The repository object Id</td>
</tr>
<tr>
<td>PARENT_ID</td>
<td>The ID of the parent.</td>
</tr>
<tr>
<td>NAME</td>
<td>Name of the folder</td>
</tr>
<tr>
<td>OWNER</td>
<td>The group owning the object</td>
</tr>
<tr>
<td>TYPE</td>
<td>This states the type of object. The following options are available: Batch, Milestone, Job and Folder.</td>
</tr>
<tr>
<td>ENVIRONMENT</td>
<td>The name of the optional environment</td>
</tr>
<tr>
<td>INHERIT_PRIVS</td>
<td>Privileges that are inherited from the parent folder</td>
</tr>
<tr>
<td>COMMENT</td>
<td>Comment if defined</td>
</tr>
<tr>
<td>COMMENTTYPE</td>
<td>Type of comment if a comment is defined</td>
</tr>
<tr>
<td>CREATOR</td>
<td>Name of the user created the object</td>
</tr>
<tr>
<td>CREATE_TIME</td>
<td>Date and time of object creation</td>
</tr>
</tbody>
</table>

Continued on next page
**Continued from previous page**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHANGER</td>
<td>Name of the last user changed the object</td>
</tr>
<tr>
<td>CHANGE_TIME</td>
<td>Date and time of object change</td>
</tr>
<tr>
<td>PRIVS</td>
<td>String containing the users privileges on the object</td>
</tr>
<tr>
<td>PARAMETERS</td>
<td>The parameters table shows all the defined constants for this folder.</td>
</tr>
<tr>
<td>DEFINED_RESOURCES</td>
<td>The Defined_Resources table shows all the resource instances that are defined for this folder.</td>
</tr>
</tbody>
</table>

Table 22.14.: Description of the output structure of the show folder statement
**show footprint**

**Purpose**

The purpose of the show footprint statement is to get detailed information about the specified footprint.

**Syntax**

The syntax for the show footprint statement is

```
show footprint footprintname [ with EXPAND ]
```

**Description**

The show footprint statement gives you detailed information about the specified footprint.

**expand** Since the number of job definitions in the table JOB_DEFINITIONS can become very large, by default they are not all displayed. If the option **expand = all** is used, all the job definitions as well as their parent folder and the folder hierarchy are outputted. Individual paths in the hierarchy can be selected by specifying individual (folder) IDs.

**Output**

This statement returns an output structure of type record.

**Output Description** The data items of the output are described in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>The repository object Id</td>
</tr>
<tr>
<td>NAME</td>
<td>Name of the footprint</td>
</tr>
<tr>
<td>COMMENT</td>
<td>Comment if defined</td>
</tr>
<tr>
<td>COMMENTTYPE</td>
<td>Type of comment if a comment is defined</td>
</tr>
</tbody>
</table>

*Continued on next page*
Continued from previous page

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATOR</td>
<td>Name of the user created the object</td>
</tr>
<tr>
<td>CREATE_TIME</td>
<td>Date and time of object creation</td>
</tr>
<tr>
<td>CHANGER</td>
<td>Name of the last user changed the object</td>
</tr>
<tr>
<td>CHANGE_TIME</td>
<td>Date and time of object change</td>
</tr>
<tr>
<td>PRIVS</td>
<td>String containing the users privileges on the object</td>
</tr>
<tr>
<td>RESOURCES</td>
<td>Table of system resources that form this footprint</td>
</tr>
<tr>
<td>JOB_DEFINITIONS</td>
<td>Table of job definitions that use this footprint</td>
</tr>
</tbody>
</table>

See also table 22.16 on page 328

Table 22.15.: Description of the output structure of the show footprint statement

**RESOURCES** The layout of the RESOURCES table is shown in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>The repository object Id</td>
</tr>
<tr>
<td>RESOURCE_NAME</td>
<td>Fully qualified path name of System Named Resources</td>
</tr>
<tr>
<td>AMOUNT</td>
<td>Amount of resource units that are allocated</td>
</tr>
<tr>
<td>KEEP_MODE</td>
<td>The Keep_Mode specifies the time at which the resource is released (FINISH, JOB_FINAL oder FINAL)</td>
</tr>
</tbody>
</table>

Table 22.16.: Description of the output structure of the show footprint suetable

**JOB_DEFINITIONS** The layout of the JOB_DEFINITIONS table is shown in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>The repository object Id</td>
</tr>
<tr>
<td>SE_PATH</td>
<td>Folder path name of the object</td>
</tr>
<tr>
<td>TYPE</td>
<td>Type of object</td>
</tr>
</tbody>
</table>

Continued on next page
Table 22.17.: Description of the output structure of the show footprint subtable

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HAS_CHILDREN</td>
<td>True means that there are more environment users further down the tree.</td>
</tr>
<tr>
<td>PRIVS</td>
<td>String containing the users privileges on the object</td>
</tr>
</tbody>
</table>
**show group**

**Purpose**

*Purpose*  The purpose of the *show group* statement is to get detailed information about the specified group.

**Syntax**

*Syntax*  The syntax for the *show group* statement is

```
show group groupname
```

**Description**

*Description*  The *show group* statement gives you detailed information about the specified group.

**Output**

*Output*  This statement returns an output structure of type record.

**Output Description**  The data items of the output are described in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>The repository object Id</td>
</tr>
<tr>
<td>NAME</td>
<td>Name of the group</td>
</tr>
<tr>
<td>COMMENTTYPE</td>
<td>Type of comment if a comment is defined</td>
</tr>
<tr>
<td>COMMENT</td>
<td>Comment if defined</td>
</tr>
<tr>
<td>CREATOR</td>
<td>Name of the user created the object</td>
</tr>
<tr>
<td>CREATE_TIME</td>
<td>Date and time of object creation</td>
</tr>
<tr>
<td>CHANGER</td>
<td>Name of the last user changed the object</td>
</tr>
<tr>
<td>CHANGE_TIME</td>
<td>Date and time of object change</td>
</tr>
<tr>
<td>PRIVS</td>
<td>String containing the users privileges on the object</td>
</tr>
<tr>
<td>MANAGE_PRIVS</td>
<td>Table of the manage privileges</td>
</tr>
<tr>
<td>USERS</td>
<td>Table of the user groups</td>
</tr>
</tbody>
</table>

Table 22.18.: Description of the output structure of the *show group* statement

330
**MANAGE_PRIVS**  The layout of the MANAGE_PRIVS table is shown in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRIVS</td>
<td>String containing the users privileges on the object</td>
</tr>
</tbody>
</table>

Table 22.19.: Description of the output structure of the show group subtable

**USERS**  The layout of the USERS table is shown in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>The repository object Id</td>
</tr>
<tr>
<td>UID</td>
<td>Id of the user</td>
</tr>
<tr>
<td>NAME</td>
<td>The object name</td>
</tr>
<tr>
<td>IS_ENABLED</td>
<td>This flag tells the user whether he can be connected.</td>
</tr>
<tr>
<td>DEFAULT_GROUP</td>
<td>The default group of this user</td>
</tr>
<tr>
<td>PRIVS</td>
<td>String containing the users privileges on the object</td>
</tr>
</tbody>
</table>

Table 22.20.: Description of the output structure of the show group subtable
**show interval**

**Purpose**  
The purpose of the `show interval` statement is to get detailed information about the interval.

**Syntax**  
The syntax for the `show interval` statement is

```
show interval intervalname [ ( id ) ] [ with day between datetime and
datetime [ , limit = integer ] ]
```

**Description**  
The `show interval` statement displays detailed information about an interval. By specifying a period, the generated blocks are returned for this period. The rising edges (start times) of the blocks are the times at which jobs would be submitted if the interval is used as a driver. If the interval is used as a filter, however, the rising edges generated by a driver interval are let through if they lie between a rising edge (inclusive) and the associated end of the block (exclusive).

**Output**  
This statement returns an output structure of type record.

**Output Description** The data items of the output are described in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>The repository object Id</td>
</tr>
<tr>
<td>NAME</td>
<td>The object name</td>
</tr>
<tr>
<td>OWNER</td>
<td>The group owning the object</td>
</tr>
<tr>
<td>STARTTIME</td>
<td>The beginning of the interval. No edges are gen-</td>
</tr>
<tr>
<td></td>
<td>erated before this time.</td>
</tr>
<tr>
<td>ENDTIME</td>
<td>The end of the interval. No edges are generated</td>
</tr>
<tr>
<td></td>
<td>after this time.</td>
</tr>
<tr>
<td>BASE</td>
<td>The period of the interval</td>
</tr>
<tr>
<td>DURATION</td>
<td>The duration of a block</td>
</tr>
</tbody>
</table>

*Continued on next page*
<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNCTIME</td>
<td>The time with which the interval is synchronised. The first period of the interval starts at this time.</td>
</tr>
<tr>
<td>INVERSE</td>
<td>The definition whether the selection list should be regarded as being positive or negative</td>
</tr>
<tr>
<td>EMBEDDED</td>
<td>The interval from which a selection is subsequently made</td>
</tr>
<tr>
<td>SELECTION</td>
<td>Single blocks are selected using Selection. [See also table 22.22 on page 334]</td>
</tr>
<tr>
<td>FILTER</td>
<td>Name(s) of the intervals that filter (multiplication) the output of this interval more finely [See also table 22.23 on page 334]</td>
</tr>
<tr>
<td>DISPATCHER</td>
<td>The Dispatch table is only relevant for Dispatch intervals. It gives detailed information about the Dispatch functionality. [See also table 22.24 on page 334]</td>
</tr>
<tr>
<td>HIERARCHY</td>
<td>The Hierarchy table shows the hierarchical structure of an interval. [See also table 22.25 on page 335]</td>
</tr>
<tr>
<td>REFERENCES</td>
<td>This field is not yet documented [See also table 22.26 on page 337]</td>
</tr>
<tr>
<td>CREATOR</td>
<td>Name of the user created the object</td>
</tr>
<tr>
<td>CREATE_TIME</td>
<td>Date and time of object creation</td>
</tr>
<tr>
<td>CHANGER</td>
<td>Name of the last user changed the object</td>
</tr>
<tr>
<td>CHANGE_TIME</td>
<td>Date and time of object change</td>
</tr>
<tr>
<td>PRIVS</td>
<td>String containing the users privileges on the object</td>
</tr>
<tr>
<td>OWNER_OBJ_TYPE</td>
<td>If an interval belongs to another object, the type of the parent object is stated in this field.</td>
</tr>
<tr>
<td>OWNER_OBJ_ID</td>
<td>If an interval belongs to another object, the ID of the parent object is stated in this field.</td>
</tr>
<tr>
<td>SE_ID</td>
<td>If an interval has been created as part of a schedule for a Job Definition, the job ID is set in the interval.</td>
</tr>
<tr>
<td>COMMENT</td>
<td>Comment if defined</td>
</tr>
<tr>
<td>COMMENTTYPE</td>
<td>Type of comment if a comment is defined</td>
</tr>
</tbody>
</table>

*Continued on next page*
Continued from previous page

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EDGES</td>
<td>If requested, a list of trigger times is returned.</td>
</tr>
</tbody>
</table>

See also table 22.27 on page 337

Table 22.21.: Description of the output structure of the show interval statement

**SELECTION**  
The layout of the SELECTION table is shown in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>The repository object Id</td>
</tr>
<tr>
<td>VALUE</td>
<td>Number of the selected edge</td>
</tr>
<tr>
<td>PERIOD_FROM</td>
<td>Beginning of the period in which all the occurring edges are considered to have been selected</td>
</tr>
<tr>
<td>PERIOD_TO</td>
<td>End of the period in which all the occurring edges are considered to have been selected</td>
</tr>
</tbody>
</table>

Table 22.22.: Description of the output structure of the show interval subtable

**FILTER**  
The layout of the FILTER table is shown in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>The repository object Id</td>
</tr>
<tr>
<td>CHILD</td>
<td>Name of the filtering interval</td>
</tr>
</tbody>
</table>

Table 22.23.: Description of the output structure of the show interval subtable

**DISPATCHER**  
The layout of the DISPATCHER table is shown in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>The repository object Id</td>
</tr>
<tr>
<td>SEQNO</td>
<td>The seqno field defines the sequence for the Dispatch rules.</td>
</tr>
<tr>
<td>NAME</td>
<td>To make the Dispatch rules easier to understand, each rule has a name.</td>
</tr>
</tbody>
</table>

Continued on next page
Continued from previous page

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT_INTERVAL_ID</td>
<td>The ID of the interval that defines the time periods in which the rule applies.</td>
</tr>
<tr>
<td>SELECT_INTERVAL_NAME</td>
<td>The name of the interval that defines the time periods in which the rule applies.</td>
</tr>
<tr>
<td>FILTER_INTERVAL_ID</td>
<td>The ID of the interval to be valuated at the times defined by the Select interval.</td>
</tr>
<tr>
<td>FILTER_INTERVAL_NAME</td>
<td>The name of the interval to be valuated at the times defined by the Select interval.</td>
</tr>
<tr>
<td>IS_ENABLED</td>
<td>This field specifies whether the rule is to be valuated or not.</td>
</tr>
<tr>
<td>IS_ACTIVE</td>
<td>This field defines whether the Filter interval is valuated or not. If the Filter interval is not valuated, nothing is let through.</td>
</tr>
</tbody>
</table>

Table 22.24.: Description of the output structure of the show interval subtable

**HIERARCHY**  The layout of the HIERARCHY table is shown in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>The repository object Id</td>
</tr>
<tr>
<td>LEVEL</td>
<td>The <strong>level</strong> specifies the hierarchy level on which the described object is located.</td>
</tr>
<tr>
<td>ROLE</td>
<td>The <strong>role</strong> field specifies the object’s role. The following possibilities are available:</td>
</tr>
<tr>
<td></td>
<td>Role</td>
</tr>
<tr>
<td></td>
<td>HEAD</td>
</tr>
<tr>
<td></td>
<td>FILTER</td>
</tr>
<tr>
<td></td>
<td>EMBEDDED</td>
</tr>
<tr>
<td></td>
<td>DISPATCH</td>
</tr>
<tr>
<td></td>
<td>DISPATCH_SELECT</td>
</tr>
<tr>
<td></td>
<td>DISPATCH_FILTER</td>
</tr>
<tr>
<td>PARENT</td>
<td>The <strong>parent</strong> field specifies which object is the parent object in the hierarchy.</td>
</tr>
<tr>
<td>NAME</td>
<td>The name of the interval.</td>
</tr>
<tr>
<td>SEQNO</td>
<td>The <strong>seqno</strong> field defines the sequence for the Dispatch rules.</td>
</tr>
</tbody>
</table>

Continued on next page
Continued from previous page

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT_INTERVAL_NAME</td>
<td>The name of the interval that defines the time periods in which the rule applies.</td>
</tr>
<tr>
<td>FILTER_INTERVAL_NAME</td>
<td>The name of the interval to be evaluated at the times defined by the Select interval.</td>
</tr>
<tr>
<td>IS_ENABLED</td>
<td>This field specifies whether the rule is to be evaluated or not.</td>
</tr>
<tr>
<td>IS_ACTIVE</td>
<td>This field defines whether the Filter interval is evaluated or not. If the Filter interval is not evaluated, nothing is let through.</td>
</tr>
<tr>
<td>OWNER</td>
<td>The group owning the object</td>
</tr>
<tr>
<td>STARTTIME</td>
<td>The beginning of the interval. No edges are generated before this time.</td>
</tr>
<tr>
<td>ENDTIME</td>
<td>The end of the interval. No edges are generated after this time.</td>
</tr>
<tr>
<td>BASE</td>
<td>The period of the interval</td>
</tr>
<tr>
<td>DURATION</td>
<td>The duration of a block</td>
</tr>
<tr>
<td>SYNCTIME</td>
<td>The time with which the interval is synchronised. The first period of the interval starts at this time.</td>
</tr>
<tr>
<td>INVERSE</td>
<td>The definition whether the selection list should be regarded as being positive or negative</td>
</tr>
<tr>
<td>EMBEDDED</td>
<td>The interval from which a selection is subsequently made</td>
</tr>
<tr>
<td>SELECTION</td>
<td>Single blocks are selected using Selection.</td>
</tr>
<tr>
<td>FILTER</td>
<td>Name(s) of the intervals that filter (multiplication) the output of this interval more finely</td>
</tr>
<tr>
<td>DISPATCHER</td>
<td>Name(s) of the intervals that filter (multiplication) the output of this interval more finely</td>
</tr>
<tr>
<td>OWNER_OBJ_TYPE</td>
<td>If an interval belongs to another object, the type of the parent object is stated in this field.</td>
</tr>
<tr>
<td>OWNER_OBJ_ID</td>
<td>If an interval belongs to another object, the ID of the parent object is stated in this field.</td>
</tr>
</tbody>
</table>

Table 22.25.: Description of the output structure of the show interval subtable

REFERENCES

The layout of the REFERENCES table is shown in the table below.
## Table 22.26.: Description of the output structure of the show interval subtable

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>REFERER_ID</td>
<td>This field is not yet documented</td>
</tr>
<tr>
<td>REFERER_NAME</td>
<td>This field is not yet documented</td>
</tr>
<tr>
<td>REFERER_TYPE</td>
<td>This field is not yet documented</td>
</tr>
<tr>
<td>REFERENCE_TYPE</td>
<td>This field is not yet documented</td>
</tr>
<tr>
<td>CHILD_ID</td>
<td>This field is not yet documented</td>
</tr>
<tr>
<td>CHILD_NAME</td>
<td>This field is not yet documented</td>
</tr>
<tr>
<td>CHILD_TYPE</td>
<td>This field is not yet documented</td>
</tr>
<tr>
<td>REFERENCE_PATH</td>
<td>This field is not yet documented</td>
</tr>
</tbody>
</table>

**EDGES**  The layout of the EDGES table is shown in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRIGGER_DATE</td>
<td>Time stamp of the rising edge. If the interval is used as a driver, triggering takes place at such times.</td>
</tr>
<tr>
<td>BLOCK_END</td>
<td>The end of the block. If the interval is used as a filter, all trigger times between trigger_date and block_end are let through. The value has no significance for drivers.</td>
</tr>
</tbody>
</table>

Table 22.27.: Description of the output structure of the show interval subtable
**show job**

**Purpose**

*Purpose* The purpose of the *show job* statement is to get detailed information about the specified job.

**Syntax**

*Syntax* The syntax for the *show job* statement is

```
show job jobid [ with WITHITEM {, WITHITEM} ]

show job submittag = string [ with WITHITEM {, WITHITEM} ]
```

*WITHITEM:*

```
filter = ( FILTERITEM {, FILTERITEM} )
| recursive audit
```

*FILTERITEM:*

```
approval REQUEST
| approve
| cancel
| change priority
| clear warning
| clone
| comment
| disable
| enable
| ignore named resource
| ignore resource
| ignore dependency [ recursive ]
| job in error
| kill
| reject
| renice
| rerun [ recursive ]
| restartable
| resume
| review REQUEST
| set exit state
| set parameter
```
Description

The `show job` statement gives you detailed information about the specified job. The job can be specified using either its Id or, if a submit tag was specified during the submit, the submit tag. The filter option is used for selecting audit entries. If the filter option is not specified, all the audit entries are shown. Otherwise, only entries of the type specified in the filter are outputted. The `recursive audit` option collects all the audit messages for the displayed job and its direct or indirect children.

Output

This statement returns an output structure of type record.

Output Description

The data items of the output are described in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>The repository object Id</td>
</tr>
<tr>
<td>SE_NAME</td>
<td>The full path name of the object</td>
</tr>
<tr>
<td>SE_OWNER</td>
<td>Owner of the object</td>
</tr>
<tr>
<td>SE_TYPE</td>
<td>The Se_Type is the object type (JOB, BATCH or MILESTONE).</td>
</tr>
<tr>
<td>SE_RUN_PROGRAM</td>
<td>The Run_Program line in the job definition</td>
</tr>
<tr>
<td>SE_RERUN_PROGRAM</td>
<td>The Rerun_Program line in the job definition</td>
</tr>
<tr>
<td>SE_KILL_PROGRAM</td>
<td>The Kill_Program line in the job definition</td>
</tr>
<tr>
<td>SE_WORKDIR</td>
<td>The Workdir of the job definition</td>
</tr>
<tr>
<td>SE_LOGFILE</td>
<td>The log file of the job definition</td>
</tr>
<tr>
<td>SE_TRUNC_LOG</td>
<td>Defines whether the log file is to be truncated before the process starts</td>
</tr>
<tr>
<td></td>
<td>or if the log information is to be appended</td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SE_ERRLOGFILE</td>
<td>The error log file of the job definition</td>
</tr>
<tr>
<td>SE_TRUNC_ERRLOG</td>
<td>Defines whether the log file is to be truncated before the process starts or if the log information is to be appended</td>
</tr>
<tr>
<td>SEEXPECTED_RUNTIME</td>
<td>The anticipated runtime of the job definition</td>
</tr>
<tr>
<td>SE_PRIORITY</td>
<td>Priority/nice value of the job definition</td>
</tr>
<tr>
<td>SE_SUBMIT_SUSPENDED</td>
<td>The Suspend Flag of the object</td>
</tr>
<tr>
<td>SE_MASTER_SUBMITTABLE</td>
<td>The Master_Submittable Flag of the object</td>
</tr>
<tr>
<td>SE_DEPENDENCY_MODE</td>
<td>The Dependency_Mode of the object</td>
</tr>
<tr>
<td>SE_ESP_NAME</td>
<td>The Exit State Profile of the object</td>
</tr>
<tr>
<td>SE_ESM_NAME</td>
<td>The Exit State Mapping the job definition</td>
</tr>
<tr>
<td>SE_ENV_NAME</td>
<td>The environment of the job definition</td>
</tr>
<tr>
<td>SE_FP_NAME</td>
<td>The footprint of the job definition</td>
</tr>
<tr>
<td>MASTER_ID</td>
<td>This is the ID of the Master Job</td>
</tr>
<tr>
<td>TIME_ZONE</td>
<td>The applicable time zone for the job</td>
</tr>
<tr>
<td>CHILD_TAG</td>
<td>Tag for exclusive identifying jobs that have been submitted several times as children of the same job</td>
</tr>
<tr>
<td>SE_VERSION</td>
<td>The version of definitions that are valid for this Submitted Entity</td>
</tr>
<tr>
<td>OWNER</td>
<td>The group owning the object</td>
</tr>
<tr>
<td>PARENT_ID</td>
<td>This is the Id of the parent.</td>
</tr>
<tr>
<td>SCOPE_ID</td>
<td>The scope or jobserver to which the job is allocated</td>
</tr>
<tr>
<td>HTTPHOST</td>
<td>The host name of the scope for accessing log files via HTTP</td>
</tr>
<tr>
<td>HTTPPORT</td>
<td>The HTTP port number of the jobserver for accessing log files via HTTP</td>
</tr>
<tr>
<td>IS_STATIC</td>
<td>Flag indicating the static or dynamic submits of this job</td>
</tr>
<tr>
<td>MERGE_MODE</td>
<td>Indicates how multiple submits of the same defined object are handled in the current Master Run</td>
</tr>
<tr>
<td>STATE</td>
<td>The State is the current state of the job.</td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IS_DISABLED</td>
<td>Indicates whether the submitted entity is disabled</td>
</tr>
<tr>
<td>IS_PARENT_DISABLED</td>
<td>Indicates whether the submitted entity is disabled</td>
</tr>
<tr>
<td>IS_CANCELED</td>
<td>Indicates whether a Cancel operation was performed on the job</td>
</tr>
<tr>
<td>JOB_ESD_ID</td>
<td>The Job_Esd is the Exit State of the job</td>
</tr>
<tr>
<td>JOB_ESD_PREF</td>
<td>The preference for merging the Job Exit States with the Child States</td>
</tr>
<tr>
<td>JOB_IS_FINAL</td>
<td>This field defines whether the job itself is final.</td>
</tr>
<tr>
<td>JOB_IS_RESTARTABLE</td>
<td>A flag indicating that this job is restartable</td>
</tr>
<tr>
<td>FINAL_ESD_ID</td>
<td>The final (merged) Exit State of the object</td>
</tr>
<tr>
<td>EXIT_CODE</td>
<td>The Exit_Code of the executed process</td>
</tr>
<tr>
<td>COMMANDLINE</td>
<td>The created command line that is used for the first execution</td>
</tr>
<tr>
<td>RR_COMMANDLINE</td>
<td>Created rerun command line that is used for the last executed rerun</td>
</tr>
<tr>
<td>WORKDIR</td>
<td>Name of the working directory of the utility process</td>
</tr>
<tr>
<td>LOGFILE</td>
<td>Name of the utility process log file. The output to stdout is written in this log.</td>
</tr>
<tr>
<td>ERRLOGFILE</td>
<td>The created error log file</td>
</tr>
<tr>
<td>PID</td>
<td>The PID is the process identification number of the monitoring jobserver process on the respective host system.</td>
</tr>
<tr>
<td>EXT_PID</td>
<td>The EXT_PID is the process identification number of the utility process.</td>
</tr>
<tr>
<td>ERROR_MSG</td>
<td>The error message describing why the job switched to the error state.</td>
</tr>
<tr>
<td>KILL_ID</td>
<td>The Submitted Entity Id of the submitted Kill Job</td>
</tr>
<tr>
<td>KILL_EXIT_CODE</td>
<td>The Exit Code of the last executed Kill Program</td>
</tr>
<tr>
<td>IS_SUSPENDED</td>
<td>This field defines whether the job or batch itself is suspended.</td>
</tr>
<tr>
<td>IS_SUSPENDED_LOCAL</td>
<td>Flag indicating whether the object is locally suspended (for restart trigger with suspend)</td>
</tr>
</tbody>
</table>
**Field** | **Description**
--- | ---
PRIORITY | The static priority of a job. This is derived from the defined priority and the nice values of the parent(s).
RAW_PRIORITY | The raw priority value of the job. Unlike the priority, this value is practically unbounded. This is required in order to be able to restore the correct priority after Nice Profile manipulations.
NICEVALUE | The current nice value of the job
NP_NICEVALUE | The np_nicevalue is the nice value which is the effect of activating (and deactivating) nice profiles.
MIN_PRIORITY | This is the minimum value for the dynamic priority.
AGING_AMOUNT | The Aging_Amount defines after how many time units the dynamic priority of a job is incremented by one point.
AGING_BASE | The Aging_Base defines the time unit for the Aging Amount.
DYNAMIC_PRIORITY | The Dynamic_Priority of the job. This is the static priority that was corrected dependent on the delay time.
PARENT_SUSPENDED | This field defines whether the job or batch is suspended through one of its parents.
CANCEL_APPROVAL | Effective Approval setting for the Cancel operation
RERUN_APPROVAL | Effective Approval setting for the cwRerunCancel operation
ENABLE_APPROVAL | Effective Approval setting for the Enable or Disable operation
SET_STATE_APPROVAL | Effective Approval setting for the Set State operation
IGN_DEPENDENCY_APPROVAL | Effective Approval setting for the Ignore Dependency operation
<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IGN_RESOURCE_APPROVAL</td>
<td>Effective Approval setting for the Ignore Resource operation</td>
</tr>
<tr>
<td>CLONE_APPROVAL</td>
<td>Effective Approval setting for the Clone operation</td>
</tr>
<tr>
<td>EDIT_PARAMETER_APPROVAL</td>
<td>Effective Approval setting for the Edit Parameter operation</td>
</tr>
<tr>
<td>KILL_APPROVAL</td>
<td>Effective Approval setting for the Kill operation</td>
</tr>
<tr>
<td>SET_JOB_STATE_APPROVAL</td>
<td>Effective Approval setting for the Set Job State operation</td>
</tr>
<tr>
<td>SUBMIT_TS</td>
<td>This is the time when the job is submitted</td>
</tr>
<tr>
<td>RESUME_TS</td>
<td>The time when the job is automatically resumed</td>
</tr>
<tr>
<td>SYNC_TS</td>
<td>The time when the job switched to the state synchronize_wait</td>
</tr>
<tr>
<td>RESOURCE_TS</td>
<td>The time when the job switched to the state Resource_wait</td>
</tr>
<tr>
<td>RUNNABLE_TS</td>
<td>The time when the job reached the state Runnable</td>
</tr>
<tr>
<td>START_TS</td>
<td>The time when the job was reported by the jobserver as having been started</td>
</tr>
<tr>
<td>FINISH_TS</td>
<td>This is the time when the job is finished</td>
</tr>
<tr>
<td>FINAL_TS</td>
<td>The time when the job reached the state Final</td>
</tr>
<tr>
<td>CNT_SUBMITTED</td>
<td>The number of children in a Submitted state</td>
</tr>
<tr>
<td>CNT_DEPENDENCY_WAIT</td>
<td>The number of children in a Dependency_Wait state</td>
</tr>
<tr>
<td>CNT_SYNCHRONIZE_WAIT</td>
<td>The number of children in a Synchronize_Wait state</td>
</tr>
<tr>
<td>CNT_RESOURCE_WAIT</td>
<td>The number of children in a Resource_Wait state</td>
</tr>
<tr>
<td>CNT_RUNNABLE</td>
<td>The number of children in a Runnable state</td>
</tr>
<tr>
<td>CNT_STARTING</td>
<td>The number of children in a Starting state</td>
</tr>
<tr>
<td>CNT_STARTED</td>
<td>The number of children in a Started state</td>
</tr>
<tr>
<td>CNT_RUNNING</td>
<td>The number of children in a Running state</td>
</tr>
<tr>
<td>CNT_TO_KILL</td>
<td>The number of children in a To_Kill state</td>
</tr>
</tbody>
</table>

*Continued on next page*
### Field Description

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CNT_KILLED</td>
<td>The number of children in a Killed state</td>
</tr>
<tr>
<td>CNT_CANCELLED</td>
<td>The number of children in a Cancelled state</td>
</tr>
<tr>
<td>CNT_FINISHED</td>
<td>The number of children in a Finished state</td>
</tr>
<tr>
<td>CNT_FINAL</td>
<td>The number of children in a Final state</td>
</tr>
<tr>
<td>CNT_BROKEN_ACTIVE</td>
<td>The number of children in a Broken_Active state</td>
</tr>
<tr>
<td>CNT_BROKEN_FINISHED</td>
<td>The number of children in a Broken_Finished state</td>
</tr>
<tr>
<td>CNT_ERROR</td>
<td>The number of children in an Error state</td>
</tr>
<tr>
<td>CNT_RESTARTABLE</td>
<td>The number of children in a Restartable state</td>
</tr>
<tr>
<td>CNT_UNREACHABLE</td>
<td>The number of children in a Unreachable state</td>
</tr>
<tr>
<td>CNT_WARN</td>
<td>The number of children with a warning</td>
</tr>
<tr>
<td>WARN_COUNT</td>
<td>This is the number of unattended warnings.</td>
</tr>
<tr>
<td>IDLE_TIME</td>
<td>The time the job was idle respectively waiting</td>
</tr>
<tr>
<td>DEPENDENCY_WAIT_TIME</td>
<td>The time the job resided in the Dependency_Wait state</td>
</tr>
<tr>
<td>SUSPEND_TIME</td>
<td>The time the job was suspended</td>
</tr>
<tr>
<td>SYNC_TIME</td>
<td>The time the job resided in the Synchronize_Wait state</td>
</tr>
<tr>
<td>RESOURCE_TIME</td>
<td>The time the job resided in the Resource_Wait state</td>
</tr>
<tr>
<td>JOBSERVER_TIME</td>
<td>The time the job was under control of a jobserver</td>
</tr>
<tr>
<td>RESTARTABLE_TIME</td>
<td>The time the job was in a Restartable state (waiting for a Rerun or Cancel)</td>
</tr>
<tr>
<td>CHILD_WAIT_TIME</td>
<td>The time the job waited for its children to reach a final state</td>
</tr>
<tr>
<td>PROCESS_TIME</td>
<td>The time a job was running or could have been running if enough resources would have been available. Hence the time from submit until final without the time it was waiting for dependencies.</td>
</tr>
<tr>
<td>ACTIVE_TIME</td>
<td>The time the job was active</td>
</tr>
<tr>
<td>IDLE_PCT</td>
<td>The percentage of the total time that a job was considered to be active</td>
</tr>
<tr>
<td>CHILDREN</td>
<td>The number of children of the job or batch</td>
</tr>
<tr>
<td>PARENTS</td>
<td>Table of the parents</td>
</tr>
</tbody>
</table>

See also table 22.29 on page 346
**Field** | **Description**
--- | ---
PARAMETER | See also table 22.30 on page 347
| Table of the parameters
| See also table 22.31 on page 347
REQUIRED_JOBS | Table of objects upon which the following objects are dependent
| See also table 22.32 on page 348
DEPENDENT_JOBS | Table of the dependent jobs
| See also table 22.33 on page 350
REQUIRED_RESOURCES | Table of the required resources
| See also table 22.34 on page 352
SUBMIT_PATH | The path from the job to the master via the submit hierarchy
IS_REPLACED | This field defines whether the job or batch has been replaced by another one.
TIMEOUT_AMOUNT | The maximum time that the job will wait for its resource
TIMEOUT_BASE | The unit that is used to specify the timeout in seconds, minutes, hours or days
TIMEOUT_STATE | The timeout of the Scheduling Entity
RERUN_SEQ | The rerun order
AUDIT_TRAIL | Table of the log entries
| See also table 22.35 on page 354
CHILD_SUSPENDED | The number of children that have been suspended
CNT_PENDING | The number of children in a Pending state
CREATOR | Name of the user created the object
CREATE_TIME | Date and time of object creation
CHANGER | Name of the last user changed the object
CHANGE_TIME | Date and time of object change
PRIVS | String containing the users privileges on the object
SE_PRIVS | Privileges for the Scheduling Entity
SUBMITTAG | Unique marker that is given at the submit time
APPROVAL_PENDING | This field indicates whether one or more approvals or reviews are currently waiting to be processed

*Continued on next page*
**Field** | **Description**
--- | ---
UNRESOLVED_HANDLING | Defines what to do if the required object cannot be found
DEFINED_RESOURCES | Table of the Defined Resources of the object
RUNS | Table of the Defined Resources of the object

See also table 22.36 on page 354
See also table 22.37 on page 355

Table 22.28.: Description of the output structure of the show job statement

**CHILDREN** The layout of the CHILDREN table is shown in the table below.

**Field** | **Description**
--- | ---
CHILDID | The Submitted Entity Id of the child
CHILDPRIVS | The privileges for the child object
CHILDSENAME | The name of the child object
CHILDSETYPE | The type of child object
CHILDSEPRIVS | The privileges for the child object
PARENTID | The Id of the parent
PARENTPRIVS | The privileges for the parent object
PARENTSENAME | The name of the parent object
PARENTSETYPE | The type of parent object
PARENTSEPRIVS | The privileges for the job definition that belong to the parent
IS_STATIC | Static flag of the hierarchy definition
PRIORITY | The priority of the hierarchy definition
SUSPEND | The suspend mode of the hierarchy definition
MERGE_MODE | The merge mode of the hierarchy definition
EST_NAME | The name of the Exit State Translation of the hierarchy definition
IGNORED_DEPENDENCIES | Ignored Dependencies flag of the hierarchy definition

Table 22.29.: Description of the output structure of the show job subtable

**PARENTS** The layout of the PARENTS table is shown in the table below.
### Field Description

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHILDID</td>
<td>The Submitted Entity Id of the child</td>
</tr>
<tr>
<td>CHILDPRIVS</td>
<td>The privileges for the child object</td>
</tr>
<tr>
<td>CHILDSERENAME</td>
<td>The name of the child object</td>
</tr>
<tr>
<td>CHILDSERETYPE</td>
<td>The type of child object</td>
</tr>
<tr>
<td>CHILDSERPRIVS</td>
<td>The privileges for the child object</td>
</tr>
<tr>
<td>PARENTID</td>
<td>The Id of the parent</td>
</tr>
<tr>
<td>PARENTPRIVS</td>
<td>The privileges for the parent object</td>
</tr>
<tr>
<td>PARENTSERENAME</td>
<td>The name of the parent object</td>
</tr>
<tr>
<td>PARENTSERETYPE</td>
<td>The type of parent object</td>
</tr>
<tr>
<td>PARENTSERPRIVS</td>
<td>The privileges for the job definition that belong to the parent</td>
</tr>
<tr>
<td>IS_STATIC</td>
<td>Static flag of the hierarchy definition</td>
</tr>
<tr>
<td>PRIORITY</td>
<td>The priority of the hierarchy definition</td>
</tr>
<tr>
<td>SUSPEND</td>
<td>The suspend mode of the hierarchy definition</td>
</tr>
<tr>
<td>MERGE_MODE</td>
<td>The merge mode of the hierarchy definition</td>
</tr>
<tr>
<td>EST_NAME</td>
<td>The name of the Exit State Translation of the hierarchy definition</td>
</tr>
<tr>
<td>IGNORED_DEPENDENCIES</td>
<td>Ignored Dependencies flag of the hierarchy definition</td>
</tr>
</tbody>
</table>

Table 22.30.: Description of the output structure of the show job subtable

### PARAMETER

The layout of the PARAMETER table is shown in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>The repository object Id</td>
</tr>
<tr>
<td>NAME</td>
<td>The name of the parameter, variable or expression</td>
</tr>
<tr>
<td>TYPE</td>
<td>The type of parameter, variable or expression</td>
</tr>
<tr>
<td>VALUE</td>
<td>The value of the parameter, variable or expression</td>
</tr>
</tbody>
</table>

Table 22.31.: Description of the output structure of the show job subtable

### REQUIRED_JOBS

The layout of the REQUIRED_JOBS table is shown in the table below.
<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>The repository object Id</td>
</tr>
<tr>
<td>DEPENDENT_ID</td>
<td>Id of the dependent Submitted Entity</td>
</tr>
<tr>
<td>DEPENDENT_PATH</td>
<td>The path from the job to the master via the submit hierarchy</td>
</tr>
<tr>
<td>DEPENDENT_PRIVS</td>
<td>The privileges for the dependent object</td>
</tr>
<tr>
<td>DEPENDENT_ID_ORIG</td>
<td>Id of the original dependent Submitted Entity on which the dependency is defined for dependencies that have been inherited from the parents</td>
</tr>
<tr>
<td>DEPENDENT_PATH_ORIG</td>
<td>The path from the dependent object to the master via the submit hierarchy</td>
</tr>
<tr>
<td>DEPENDENT_PRIVS_ORIG</td>
<td>The privileges for the original dependent object</td>
</tr>
<tr>
<td>DEPENDENCY_OPERATION</td>
<td>Defines whether all or only some dependencies of the original object have to be fulfilled</td>
</tr>
<tr>
<td>REQUIRED_ID</td>
<td>Id of the required Submitted Entity</td>
</tr>
<tr>
<td>REQUIRED_PATH</td>
<td>The path from the required object to the master via the submit hierarchy</td>
</tr>
<tr>
<td>REQUIRED_PRIVS</td>
<td>The privileges for the required object</td>
</tr>
<tr>
<td>STATE</td>
<td>The state of the dependency (OPEN, FULL-FILLED or FAILED)</td>
</tr>
<tr>
<td>DD_ID</td>
<td>Id of the Dependency Definition object</td>
</tr>
<tr>
<td>DD_NAME</td>
<td>Name of the Dependency Definition</td>
</tr>
<tr>
<td>DD_DEPENDENTNAME</td>
<td>The full path name of the object</td>
</tr>
<tr>
<td>DD_DEPENDENTTYPE</td>
<td>The type of dependent object</td>
</tr>
<tr>
<td>DD_DEPENDENTPRIVS</td>
<td>Privileges for the dependent object</td>
</tr>
<tr>
<td>DD_REQUIREDNAME</td>
<td>Path name of the definition of the dependent object</td>
</tr>
<tr>
<td>DD_REQUIREDTYPE</td>
<td>The type of required object</td>
</tr>
<tr>
<td>DD_REQUIREDPRIVS</td>
<td>The privileges for the required object definition</td>
</tr>
<tr>
<td>DD_UNRESOLVED_HANDLING</td>
<td>Specifies how to handle unresolvable dependencies during a submit</td>
</tr>
<tr>
<td>DD_STATE_SELECTION</td>
<td>The State Selection defines how the required Exit States are determined. The options here are FINAL, ALL_REACHABLE, UNREACHABLE and DEFAULT. In the case of FINAL, the required Exit States can be explicitly listed.</td>
</tr>
</tbody>
</table>

Continued on next page
### Field Description

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DD_MODE</td>
<td>Defines whether just the required job itself or the required job together with its children have to be final</td>
</tr>
<tr>
<td>DD_STATES</td>
<td>List of Exit States that the required object have to achieve to fulfil the dependency</td>
</tr>
<tr>
<td>JOB_STATE</td>
<td>In the Job State list, you can filter for jobs that have the entered Job State.</td>
</tr>
<tr>
<td>IS_SUSPENDED</td>
<td>This field defines whether the job or batch itself is suspended.</td>
</tr>
<tr>
<td>PARENT_SUSPENDED</td>
<td>This field defines whether the job is suspended (True) or not (False) through one of its parents.</td>
</tr>
<tr>
<td>CNT_SUBMITTED</td>
<td>The number of children in a Submitted state</td>
</tr>
<tr>
<td>CNT_DEPENDENCY_WAIT</td>
<td>The number of children in a Dependency_Wait state</td>
</tr>
<tr>
<td>CNT_SYNCHRONIZE_WAIT</td>
<td>The number of children in a Synchronize_Wait state</td>
</tr>
<tr>
<td>CNTRESOURCE_WAIT</td>
<td>The number of children in a Resource_Wait state</td>
</tr>
<tr>
<td>CNT_RUNNABLE</td>
<td>The number of children in a Runnable state</td>
</tr>
<tr>
<td>CNT_STARTING</td>
<td>The number of children in a Starting state</td>
</tr>
<tr>
<td>CNT_STARTED</td>
<td>The number of children in a Started state</td>
</tr>
<tr>
<td>CNT_RUNNING</td>
<td>The number of children in a Running state</td>
</tr>
<tr>
<td>CNT_TO_KILL</td>
<td>The number of children in a To_Kill state</td>
</tr>
<tr>
<td>CNT_KILLED</td>
<td>The number of children in a Killed state</td>
</tr>
<tr>
<td>CNT_CANCELLED</td>
<td>The number of children in a Cancelled state</td>
</tr>
<tr>
<td>CNT_FINISHED</td>
<td>The number of children in a Finished state</td>
</tr>
<tr>
<td>CNT_FINAL</td>
<td>The number of children in a Final state</td>
</tr>
<tr>
<td>CNT_BROKEN_ACTIVE</td>
<td>The number of children in a Broken_Active state</td>
</tr>
<tr>
<td>CNT_BROKEN_FINISHED</td>
<td>The number of children in a Broken_Finished state</td>
</tr>
<tr>
<td>CNT_ERROR</td>
<td>The number of children in an Error state</td>
</tr>
<tr>
<td>CNT_RESTARTABLE</td>
<td>The number of children in a Restartable state</td>
</tr>
<tr>
<td>CNT_UNREACHABLE</td>
<td>The number of children in an Unreachable state</td>
</tr>
<tr>
<td>JOB_IS_FINAL</td>
<td>The number of Child Jobs in an Is_Final state</td>
</tr>
<tr>
<td>CHILD_TAG</td>
<td>Tag for exclusive identifying jobs that have been submitted several times as children of the same job</td>
</tr>
<tr>
<td>FINAL_STATE</td>
<td>The final state of a job</td>
</tr>
</tbody>
</table>

*Continued on next page*
Table 22.32.: Description of the output structure of the show job subtable

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHILDREN</td>
<td>The number of children of the job or batch</td>
</tr>
<tr>
<td>IGNORE</td>
<td>Flag indicating whether the Resource Allocation is bring ignored</td>
</tr>
<tr>
<td>CHILD_SUSPENDED</td>
<td>The number of children that have been suspended</td>
</tr>
<tr>
<td>CNT_PENDING</td>
<td>The number of children in a Pending state</td>
</tr>
<tr>
<td>DD_CONDITION</td>
<td>The condition that has to be additionally fulfilled for the dependency to be fulfilled</td>
</tr>
</tbody>
</table>

**DEPENDENT_JOBS**  The layout of the DEPENDENT_JOBS table is shown in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>The repository object Id</td>
</tr>
<tr>
<td>DEPENDENT_ID</td>
<td>Id of the dependent Submitted Entity</td>
</tr>
<tr>
<td>DEPENDENT_PATH</td>
<td>The path from the job to the master via the submit hierarchy</td>
</tr>
<tr>
<td>DEPENDENT_PRIVS</td>
<td>The privileges for the dependent object</td>
</tr>
<tr>
<td>DEPENDENT_ID_ORIG</td>
<td>Id of the original dependent Submitted Entity on which the dependency is defined for dependencies that have been inherited from the parents</td>
</tr>
<tr>
<td>DEPENDENT_PATH_ORIG</td>
<td>The path from the dependent object to the master via the submit hierarchy</td>
</tr>
<tr>
<td>DEPENDENT_PRIVS_ORIG</td>
<td>The privileges for the original dependent object</td>
</tr>
<tr>
<td>DEPENDENCY_OPERATION</td>
<td>Defines whether all or only some dependencies of the original object have to be fulfilled</td>
</tr>
<tr>
<td>REQUIRED_ID</td>
<td>Id of the required Submitted Entity</td>
</tr>
<tr>
<td>REQUIRED_PATH</td>
<td>The path from the required object to the master via the submit hierarchy</td>
</tr>
<tr>
<td>REQUIRED_PRIVS</td>
<td>The privileges for the required object</td>
</tr>
<tr>
<td>STATE</td>
<td>The state of the dependency (OPEN, FUL-FILLED or FAILED)</td>
</tr>
<tr>
<td>DD_ID</td>
<td>Id of the Dependency Definition object</td>
</tr>
<tr>
<td>DD_NAME</td>
<td>Name of the Dependency Definition Definition</td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DD_DEPENDENTNAME</td>
<td>The full path name of the object</td>
</tr>
<tr>
<td>DD_DEPENDENTTYPE</td>
<td>The type of dependent object</td>
</tr>
<tr>
<td>DD_DEPENDENTPRIVS</td>
<td>Privileges for the dependent object</td>
</tr>
<tr>
<td>DD_REQUIREDNAME</td>
<td>Path name of the definition of the dependent object</td>
</tr>
<tr>
<td>DD_REQUIREDTYPE</td>
<td>The type of required object</td>
</tr>
<tr>
<td>DD_REQUIREDPRIVS</td>
<td>The privileges for the required object definition</td>
</tr>
<tr>
<td>DD_UNRESOLVEDHANDLING</td>
<td>Specifies how to handle unresolvable dependencies during a submit</td>
</tr>
<tr>
<td>DD_STATE_SELECTION</td>
<td>The State Selection defines how the required Exit States are determined. The options here are FINAL, ALL_REACHABLE, UNREACHABLE and DEFAULT. In the case of FINAL, the required Exit States can be explicitly listed.</td>
</tr>
<tr>
<td>DD_MODE</td>
<td>Defines whether just the required job itself or the required job together with its children have to be final</td>
</tr>
<tr>
<td>DD_STATES</td>
<td>List of Exit States that the required object have to achieve to fulfil the dependency</td>
</tr>
<tr>
<td>JOB_STATE</td>
<td>In the Job State list, you can filter for jobs that have the entered Job State.</td>
</tr>
<tr>
<td>IS_SUSPENDED</td>
<td>This field defines whether the job or batch itself is suspended.</td>
</tr>
<tr>
<td>PARENT_SUSPENDED</td>
<td>This field defines whether the job is suspended (True) or not (False) through one of its parents.</td>
</tr>
<tr>
<td>CNT_SUBMITTED</td>
<td>The number of children in a Submitted state</td>
</tr>
<tr>
<td>CNT_DEPENDENCY_WAIT</td>
<td>The number of children in a Dependency_Wait state</td>
</tr>
<tr>
<td>CNT_SYNCHRONIZE_WAIT</td>
<td>The number of children in a Synchronize_Wait state</td>
</tr>
<tr>
<td>CNT_RESOURCE_WAIT</td>
<td>The number of children in a Resource_Wait state</td>
</tr>
<tr>
<td>CNT_RUNNABLE</td>
<td>The number of children in a Runnable state</td>
</tr>
<tr>
<td>CNT_STARTING</td>
<td>The number of children in a Starting state</td>
</tr>
<tr>
<td>CNT_STARTED</td>
<td>The number of children in a Started state</td>
</tr>
<tr>
<td>CNT_RUNNING</td>
<td>The number of children in a Running state</td>
</tr>
<tr>
<td>CNT_TO_KILL</td>
<td>The number of children in a To_Kill state</td>
</tr>
<tr>
<td>CNT_KILLED</td>
<td>The number of children in a Killed state</td>
</tr>
</tbody>
</table>
Continued from previous page

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CNT_CANCELLED</td>
<td>The number of children in a Cancelled state</td>
</tr>
<tr>
<td>CNT_FINISHED</td>
<td>The number of children in a Finished state</td>
</tr>
<tr>
<td>CNT_FINAL</td>
<td>The number of children in a Final state</td>
</tr>
<tr>
<td>CNT_BROKEN_ACTIVE</td>
<td>The number of children in a Broken_Active state</td>
</tr>
<tr>
<td>CNT_BROKEN_FINISHED</td>
<td>The number of children in a Broken_Finished state</td>
</tr>
<tr>
<td>CNT_ERROR</td>
<td>The number of children in an Error state</td>
</tr>
<tr>
<td>CNT_RESTARTABLE</td>
<td>The number of children in a Restartable state</td>
</tr>
<tr>
<td>CNT_UNREACHABLE</td>
<td>The number of children in an Unreachable state</td>
</tr>
<tr>
<td>JOB_IS_FINAL</td>
<td>The number of Child Jobs in an Is_Final state</td>
</tr>
<tr>
<td>CHILD_TAG</td>
<td>Tag for exclusive identifying jobs that have been submitted several times as</td>
</tr>
<tr>
<td></td>
<td>children of the same job</td>
</tr>
<tr>
<td>FINAL_STATE</td>
<td>The final state of a job</td>
</tr>
<tr>
<td>CHILDREN</td>
<td>The number of children of the job or batch</td>
</tr>
<tr>
<td>IGNORE</td>
<td>Flag indicating whether the Resource Allocation is bring ignored</td>
</tr>
<tr>
<td>CHILD_SUSPENDED</td>
<td>The number of children that have been suspended</td>
</tr>
<tr>
<td>CNT_PENDING</td>
<td>The number of children in a Pending state</td>
</tr>
<tr>
<td>DD_CONDITION</td>
<td>The condition that has to be additionally fulfilled for the dependency to be</td>
</tr>
<tr>
<td></td>
<td>fulfilled</td>
</tr>
</tbody>
</table>

Table 22.33.: Description of the output structure of the show job subtable

REQUIRED_RESOURCES The layout of the REQUIRED_RESOURCES table is shown in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCOPE_ID</td>
<td>Id of the scope that allocated the resource</td>
</tr>
<tr>
<td>SCOPE_NAME</td>
<td>The fully qualified name of the scope</td>
</tr>
<tr>
<td>SCOPE_TYPE</td>
<td>The type of scope (SCOPE or SERVER, FOLDER, BATCH or JOB)</td>
</tr>
<tr>
<td>SCOPE_PRIVS</td>
<td>The privileges for the scope</td>
</tr>
<tr>
<td>RESOURCE_ID</td>
<td>Id of the Requested Resource</td>
</tr>
<tr>
<td>RESOURCE_NAME</td>
<td>Categorical path name of the Requested Resource</td>
</tr>
</tbody>
</table>

Continued on next page
Continued from previous page

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RESOURCE_USAGE</td>
<td>The usage of the required resource (STATIC, SYSTEM or SYNCHRONIZING)</td>
</tr>
<tr>
<td>RESOURCE_OWNER</td>
<td>Name of the owner of the Requested Resource</td>
</tr>
<tr>
<td>RESOURCE_PRIVS</td>
<td>The privileges for the Requested Resource</td>
</tr>
<tr>
<td>RESOURCE_STATE</td>
<td>The state of the Requested Resource</td>
</tr>
<tr>
<td>RESOURCE_TIMESTAMP</td>
<td>Date time of last time state was set for the requested resource</td>
</tr>
<tr>
<td>REQUESTABLE_AMOUNT</td>
<td>The maximum amount of resources that can be requested by a job</td>
</tr>
<tr>
<td>TOTAL_AMOUNT</td>
<td>The complete amount that can be allocated</td>
</tr>
<tr>
<td>FREE_AMOUNT</td>
<td>The Free_Amount that can be allocated</td>
</tr>
<tr>
<td>REQUESTED_AMOUNT</td>
<td>This is the requested amount</td>
</tr>
<tr>
<td>REQUESTED_LOCKMODE</td>
<td>The requested lockmode</td>
</tr>
<tr>
<td>REQUESTED_STATES</td>
<td>The requested Resource State</td>
</tr>
<tr>
<td>RESERVED_AMOUNT</td>
<td>The amount that is reserved by the Requested Resource</td>
</tr>
<tr>
<td>ALLOCATED_AMOUNT</td>
<td>The amount that was allocated by the Requested Resource</td>
</tr>
<tr>
<td>ALLOCATED_LOCKMODE</td>
<td>The lockmode currently allocated by the Requested Resource</td>
</tr>
<tr>
<td>IGNORE</td>
<td>Flag indicating whether the Resource Allocation is bring ignored</td>
</tr>
<tr>
<td>STICKY</td>
<td>Flag indicating whether it is a Sticky Resource Allocation</td>
</tr>
<tr>
<td>STICKY_NAME</td>
<td>Optional name of the sticky resource request</td>
</tr>
<tr>
<td>STICKY_PARENT</td>
<td>Parent job within which the sticky request is evaluated</td>
</tr>
<tr>
<td>STICKY_PARENT_TYPE</td>
<td>Type of the parent within which the sticky requirement is evaluated</td>
</tr>
<tr>
<td>ONLINE</td>
<td>Flag indicating whether the resource is available for an allocation</td>
</tr>
<tr>
<td>ALLOCATE_STATE</td>
<td>The state of the allocation (RESERVED, ALLOCATED, AVAILABLE or BLOCKED)</td>
</tr>
<tr>
<td>EXPIRE</td>
<td>Time defining the maximum or minimum age of a resource depending on whether the expire is positive or negative</td>
</tr>
</tbody>
</table>

Continued on next page
Continued from previous page

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXPIRE_SIGN</td>
<td>Defines the expiration condition, +/- indicating younger/older than</td>
</tr>
<tr>
<td>IGNORE_ON_RERUN</td>
<td>This flag indicates if the expire condition should be ignored in case of a rerun.</td>
</tr>
<tr>
<td>DEFINITION</td>
<td>Where the Resource Definition is saved</td>
</tr>
</tbody>
</table>

Table 22.34.: Description of the output structure of the show job subtable

**AUDIT_TRAIL**  The layout of the AUDIT_TRAIL table is shown in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>The repository object Id</td>
</tr>
<tr>
<td>USERNAME</td>
<td>User name that causes this audit record</td>
</tr>
<tr>
<td>TIME</td>
<td>The time when this audit record was created</td>
</tr>
<tr>
<td>TXID</td>
<td>Transaction number of the change</td>
</tr>
<tr>
<td>ACTION</td>
<td>Action that causes this audit record</td>
</tr>
<tr>
<td>ORIGINID</td>
<td>The original object Id that causes this audit record</td>
</tr>
<tr>
<td>JOBID</td>
<td>The Id of the job for which this audit entry is written</td>
</tr>
<tr>
<td>JOBNAME</td>
<td>The name of the job for which this audit entry is written</td>
</tr>
<tr>
<td>COMMENT</td>
<td>Comment if defined</td>
</tr>
<tr>
<td>INFO</td>
<td>Additional system information about the Action Event that caused the audit record</td>
</tr>
</tbody>
</table>

Table 22.35.: Description of the output structure of the show job subtable

**DEFINED_RESOURCES**  The layout of the DEFINED_RESOURCES table is shown in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>Id of the Defined Resource</td>
</tr>
<tr>
<td>RESOURCE_NAME</td>
<td>Full path name of the Defined Object</td>
</tr>
</tbody>
</table>

Continued on next page
### Field Description

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RESOURCE_USAGE</td>
<td>The usage of the required resource (STATIC, SYSTEM or SYNCHRONIZING)</td>
</tr>
<tr>
<td>RESOURCE_OWNER</td>
<td>The owner of the resource</td>
</tr>
<tr>
<td>RESOURCE_PRIVS</td>
<td>The privileges for the resource</td>
</tr>
<tr>
<td>RESOURCE_STATE</td>
<td>The current state of the resource</td>
</tr>
<tr>
<td>RESOURCE_TIMESTAMP</td>
<td>Date time of last time state was set for the requested resource</td>
</tr>
<tr>
<td>REQUESTABLE_AMOUNT</td>
<td>The maximum amount of resources that can be requested by a job</td>
</tr>
<tr>
<td>TOTAL_AMOUNT</td>
<td>The complete amount that can be allocated</td>
</tr>
<tr>
<td>FREE_AMOUNT</td>
<td>The Free_Amount that can be allocated</td>
</tr>
<tr>
<td>ONLINE</td>
<td>Indicates whether the resource can be allocated or not</td>
</tr>
</tbody>
</table>

Table 22.36.: Description of the output structure of the show job subtable

### RUNS

The layout of the RUNS table is shown in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RERUN_SEQ</td>
<td>The rerun order</td>
</tr>
<tr>
<td>SCOPE_ID</td>
<td>The scope or jobserver to which the job is allocated</td>
</tr>
<tr>
<td>HTTPHOST</td>
<td>The host name of the scope for accessing log files via HTTP</td>
</tr>
<tr>
<td>HTTPPORT</td>
<td>The HTTP port number of the jobserver for accessing log files via HTTP</td>
</tr>
<tr>
<td>JOB_ESD_ID</td>
<td>The Job_Esd is the Exit State of the job.</td>
</tr>
<tr>
<td>EXIT_CODE</td>
<td>The Exit_Code of the executed process</td>
</tr>
<tr>
<td>COMMANDLINE</td>
<td>The created command line that is used for the first execution</td>
</tr>
<tr>
<td>WORKDIR</td>
<td>Name of the working directory of the utility process</td>
</tr>
<tr>
<td>LOGFILE</td>
<td>Name of the utility process log file. The output to stdout is written in this log.</td>
</tr>
<tr>
<td>ERRLOGFILE</td>
<td>The created error log file</td>
</tr>
<tr>
<td>EXT_PID</td>
<td>The EXT_PID is the process identification number of the utility process.</td>
</tr>
</tbody>
</table>

Continued on next page
Continued from previous page

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNC_TS</td>
<td>The time when the job switched to the state synchronize_wait</td>
</tr>
<tr>
<td>RESOURCE_TS</td>
<td>The time when the job switched to the state Resource_wait</td>
</tr>
<tr>
<td>RUNNABLE_TS</td>
<td>The time when the job reached the state Runnable</td>
</tr>
<tr>
<td>START_TS</td>
<td>The time when the job was reported by the job-server as having been started</td>
</tr>
<tr>
<td>FINISH_TS</td>
<td>This is the time when the job is finished.</td>
</tr>
</tbody>
</table>

Table 22.37.: Description of the output structure of the show job subtable
show job definition

Purpose
The purpose of the show job definition statement is to get detailed information about the specified job definition.

Syntax
The syntax for the show job definition statement is

```
show job definition folderpath
```

Description
The show job definition statement gives you detailed information about the specified job definition.

Output
This statement returns an output structure of type record.

**Output Description** The data items of the output are described in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>The repository object Id</td>
</tr>
<tr>
<td>PARENT_ID</td>
<td>The ID of the parent.</td>
</tr>
<tr>
<td>NAME</td>
<td>The full path name of the job definition</td>
</tr>
<tr>
<td>OWNER</td>
<td>The group owning the object</td>
</tr>
<tr>
<td>TYPE</td>
<td>This states the type of object. The following options are available: Batch, Milestone, Job and Folder.</td>
</tr>
<tr>
<td>INHERIT_PRIVS</td>
<td>Privileges that are inherited from the parent folder</td>
</tr>
<tr>
<td>RUN_PROGRAM</td>
<td>A command line that starts the script or program can be specified in the Run_Program field.</td>
</tr>
<tr>
<td>RERUN_PROGRAM</td>
<td>The Rerun_Program field specifies the command that is to be executed when repeating the job following an error (rerun).</td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th><strong>Field</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>KILL_PROGRAM</td>
<td>The Kill_Program field determines which program is to be run to terminate a currently running job.</td>
</tr>
<tr>
<td>WORKDIR</td>
<td>This is the working directory of the current job.</td>
</tr>
<tr>
<td>LOGFILE</td>
<td>The Logfile field specifies the file in which all the normal outputs of the Run program are to be returned. These are usually all the outputs that use the standard output channel (STDOUT under UNIX).</td>
</tr>
<tr>
<td>TRUNC_LOG</td>
<td>Defines whether the log file is to be renewed or not</td>
</tr>
<tr>
<td>ERRLOGFILE</td>
<td>The Error Logfile field specifies the file in which all the error outputs from the Run_program are to be returned.</td>
</tr>
<tr>
<td>TRUNC_ERRLOG</td>
<td>Defines whether the Error log file is to be renewed or not</td>
</tr>
<tr>
<td>EXPECTED_RUNTIME</td>
<td>The Expected_Runtime describes the anticipated time that will be required to execute a job.</td>
</tr>
<tr>
<td>EXPECTED_FINALTIME</td>
<td>The Expected_Finaltime describes the anticipated time that will be required to execute a job or batch together with its children.</td>
</tr>
<tr>
<td>PRIORITY</td>
<td>The Priority field indicates the urgency with which the process, if it is to be started, is to be considered by the Scheduling System.</td>
</tr>
<tr>
<td>MIN_PRIORITY</td>
<td>This is the minimum effective priority that can be achieved through natural aging.</td>
</tr>
<tr>
<td>AGING_AMOUNT</td>
<td>The number of time units after which the effective priority is incremented by 1.</td>
</tr>
<tr>
<td>AGING_BASE</td>
<td>The time unit that is used for the aging interval</td>
</tr>
<tr>
<td>SUBMIT_SUSPENDED</td>
<td>Flag that indicates whether the object is to be suspended after the submit</td>
</tr>
<tr>
<td>RESUME_AT</td>
<td>If the job is to be submitted as being suspended, an automatic resume takes place at the given time.</td>
</tr>
<tr>
<td>RESUME_IN</td>
<td>If the job is to be submitted as being suspended, an automatic resume takes place after the given number of time units.</td>
</tr>
<tr>
<td>RESUME_BASE</td>
<td>Specified time unit for RESUME_IN</td>
</tr>
</tbody>
</table>

*Continued on next page*
<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MASTER_SUBMITTABLE</td>
<td>The job that is started by the trigger is submitted as its own Master Job and does not have any influence on the current Master Job run of the triggering job.</td>
</tr>
<tr>
<td>TIMEOUT_AMOUNT</td>
<td>The number of time units for the delay until the timeout occurs</td>
</tr>
<tr>
<td>TIMEOUT_BASE</td>
<td>The unit that is used to specify the timeout in seconds, minutes, hours or days</td>
</tr>
<tr>
<td>TIMEOUT_STATE</td>
<td>The timeout of the Scheduling Entity</td>
</tr>
<tr>
<td>DEPENDENCY_MODE</td>
<td>The Dependency Mode states the context in which the list of dependencies has to be viewed. The following options are available: ALL and ANY.</td>
</tr>
<tr>
<td>ESP_NAME</td>
<td>This is the name of the Exit State Profile.</td>
</tr>
<tr>
<td>ESM_NAME</td>
<td>This is the name of the Exit State Mapping.</td>
</tr>
<tr>
<td>ENV_NAME</td>
<td>This is the name of the environment.</td>
</tr>
<tr>
<td>CANCEL_LEAD_FLAG</td>
<td>The Lead flag indicates whether the Approval setting should also apply to all children. The Approval settings cannot be relaxed but only tightened up.</td>
</tr>
<tr>
<td>CANCEL_APPROVAL</td>
<td>The Approval setting for the Cancel operation</td>
</tr>
<tr>
<td>RERUN_LEAD_FLAG</td>
<td>The Lead flag indicates whether the Approval setting should also apply to all children. The Approval settings cannot be relaxed but only tightened up.</td>
</tr>
<tr>
<td>RERUN_APPROVAL</td>
<td>The Approval setting for the Rerun operation</td>
</tr>
<tr>
<td>ENABLE_LEAD_FLAG</td>
<td>The Lead flag indicates whether the Approval setting should also apply to all children. The Approval settings cannot be relaxed but only tightened up.</td>
</tr>
<tr>
<td>ENABLE_APPROVAL</td>
<td>The effective Approval setting for the Enable or Disable operation</td>
</tr>
<tr>
<td>Field</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>SET_STATE_LEAD_FLAG</td>
<td>The Lead flag indicates whether the Approval setting should also apply to all children. The Approval settings cannot be relaxed but only tightened up.</td>
</tr>
<tr>
<td>SET_STATE_APPROVAL</td>
<td>The Approval setting for the Set State operation</td>
</tr>
<tr>
<td>IGN_DEPENDENCY_LEAD_FLAG</td>
<td>The Lead flag indicates whether the Approval setting should also apply to all children. The Approval settings cannot be relaxed but only tightened up.</td>
</tr>
<tr>
<td>IGN_DEPENDENCY_APPROVAL</td>
<td>The Approval setting for the Ignore Dependency operation</td>
</tr>
<tr>
<td>IGN_RESOURCE_LEAD_FLAG</td>
<td>The Lead flag indicates whether the Approval setting should also apply to all children. The Approval settings cannot be relaxed but only tightened up.</td>
</tr>
<tr>
<td>IGN_RESOURCE_APPROVAL</td>
<td>The Approval setting for the Ignore Resource operation</td>
</tr>
<tr>
<td>CLONE_LEAD_FLAG</td>
<td>The Lead flag indicates whether the Approval setting should also apply to all children. The Approval settings cannot be relaxed but only tightened up.</td>
</tr>
<tr>
<td>CLONE_APPROVAL</td>
<td>The Approval setting for the Clone operation</td>
</tr>
<tr>
<td>EDIT_PARAMETER_LEAD_FLAG</td>
<td>The Lead flag indicates whether the Approval setting should also apply to all children. The Approval settings cannot be relaxed but only tightened up.</td>
</tr>
<tr>
<td>EDIT_PARAMETER_APPROVAL</td>
<td>The Approval setting for the Edit Parameter operation</td>
</tr>
</tbody>
</table>
### Field Description

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>KILL_LEAD_FLAG</td>
<td>The Lead flag indicates whether the Approval setting should also apply to all children. The Approval settings cannot be relaxed but only tightened up.</td>
</tr>
<tr>
<td>KILL_APPROVAL</td>
<td>The Approval setting for the Kill operation</td>
</tr>
<tr>
<td>SET_JOB_STATE_LEAD_FLAG</td>
<td>The Lead flag indicates whether the Approval setting should also apply to all children. The Approval settings cannot be relaxed but only tightened up.</td>
</tr>
<tr>
<td>SET_JOB_STATE_APPROVAL</td>
<td>The Approval setting for the Set Job State operation</td>
</tr>
<tr>
<td>FP_NAME</td>
<td>This is the name of the footprint.</td>
</tr>
<tr>
<td>COMMENT</td>
<td>Comment if defined</td>
</tr>
<tr>
<td>COMMENTTYPE</td>
<td>Type of comment if a comment is defined</td>
</tr>
<tr>
<td>PRIVS</td>
<td>String containing the users privileges on the object</td>
</tr>
<tr>
<td>CREATOR</td>
<td>Name of the user created the object</td>
</tr>
<tr>
<td>CREATE_TIME</td>
<td>Date and time of object creation</td>
</tr>
<tr>
<td>CHANGER</td>
<td>Name of the last user changed the object</td>
</tr>
<tr>
<td>CHANGE_TIME</td>
<td>Date and time of object change</td>
</tr>
<tr>
<td>CHILDREN</td>
<td>Table of the children</td>
</tr>
<tr>
<td>PARENTS</td>
<td>Table of the parents</td>
</tr>
<tr>
<td>PARAMETER</td>
<td>Table of the parameters and variables that are defined for this object</td>
</tr>
<tr>
<td>REFERENCES</td>
<td>Table of parameter references to this object</td>
</tr>
<tr>
<td>REQUIRED_JOBS</td>
<td>Table of objects upon which the following objects are dependent</td>
</tr>
<tr>
<td>DEPENDENT_JOBS</td>
<td>Table of objects that are dependent upon the following objects</td>
</tr>
<tr>
<td></td>
<td>See also table 22.39 on page 362</td>
</tr>
<tr>
<td></td>
<td>See also table 22.40 on page 364</td>
</tr>
<tr>
<td></td>
<td>Table of objects upon which the following objects are dependent</td>
</tr>
<tr>
<td></td>
<td>See also table 22.41 on page 365</td>
</tr>
<tr>
<td></td>
<td>Table of objects that are dependent upon the following objects</td>
</tr>
<tr>
<td></td>
<td>See also table 22.42 on page 367</td>
</tr>
</tbody>
</table>
### Table of resource requirements that are not included in the environment and footprint

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>REQUIRED.Resources</td>
<td>Table of resource requirements that are not included in the environment and footprint</td>
</tr>
<tr>
<td></td>
<td>See also table 22.43 on page 368</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEFINED.Resources</td>
<td>Table of resources to be instantiated at the submit time, visible for submitting children</td>
</tr>
</tbody>
</table>

Table 22.38.: Description of the output structure of the show job definition statement

### CHILDREN

The layout of the CHILDREN table is shown in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>The repository object Id</td>
</tr>
<tr>
<td>CHILD_ID</td>
<td>Full path name of the child object</td>
</tr>
<tr>
<td>CHILDNAME</td>
<td>Full path name of the child object</td>
</tr>
<tr>
<td>CHILDTYPE</td>
<td>The child type (JOB, BATCH or MILESTONE)</td>
</tr>
<tr>
<td>CHILDPRIVS</td>
<td>A string containing the user privileges of the child object</td>
</tr>
<tr>
<td>PARENT_ID</td>
<td>The name of the parent object</td>
</tr>
<tr>
<td>PARENTNAME</td>
<td>The name of the parent object</td>
</tr>
<tr>
<td>PARENTTYPE</td>
<td>The parent type (JOB, BATCH or MILESTONE)</td>
</tr>
<tr>
<td>PARENTPRIVS</td>
<td>A string containing the user privileges of the parent object</td>
</tr>
<tr>
<td>ALIAS_NAME</td>
<td>Name for referencing to child definitions with dynamic submits</td>
</tr>
<tr>
<td>IS_STATIC</td>
<td>The is_static flag defines whether the job is to be statically or dynamically submitted.</td>
</tr>
<tr>
<td>IS_DISABLED</td>
<td>Flag indicating the the child should be executed or skipped</td>
</tr>
<tr>
<td>INT_NAME</td>
<td>The interval id is the ID of the interval used to check whether the child is enabled.</td>
</tr>
</tbody>
</table>

Continued on next page
The enable condition, if completed, determines whether a child is enabled or disabled. The condition is evaluated at the time of the submit; any parameter values must therefore already be known at this time. The basic idea is to enable parameter-controlled process variants with the help of the condition.

The enable mode determines how the results of the enable condition and the enable interval are linked to one another. The possibilities are AND and OR. In the first case, a child will only be enabled if both the enable interval and the condition give cause for this. In the latter case, only one of the two has to give the go-ahead. If either condition is missing, the value for enable mode is irrelevant.

The nice value that has been added to the children.

Determines whether the child is to be suspended for the submit.

If the job is to be submitted as being suspended, an automatic resume takes place at the given time.

If the job is to be submitted as being suspended, an automatic resume takes place after the given number of time units.

Specified time unit for RESUME_IN.

Determines how the condition handles the same object that occurs more than once in the submission hierarchy.

An Exit State Translation that is used to translate the Exit States of the children to the Exit States of the parents.

List with the names of the dependencies for ignoring the dependencies of the parents.

Table 22.39.: Description of the output structure of the show job definition subtable
The layout of the PARENTS table is shown in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>The repository object Id</td>
</tr>
<tr>
<td>CHILD_ID</td>
<td>Full path name of the child object</td>
</tr>
<tr>
<td>CHILDNAME</td>
<td>Full path name of the child object</td>
</tr>
<tr>
<td>CHILDTYPE</td>
<td>The child type (JOB, BATCH or MILESTONE)</td>
</tr>
<tr>
<td>CHILDPRIVS</td>
<td>A string containing the user privileges of the child object</td>
</tr>
<tr>
<td>PARENT_ID</td>
<td>The name of the parent object</td>
</tr>
<tr>
<td>PARENTNAME</td>
<td>The name of the parent object</td>
</tr>
<tr>
<td>PARENTTYPE</td>
<td>The parent type (JOB, BATCH or MILESTONE)</td>
</tr>
<tr>
<td>PARENTPRIVS</td>
<td>A string containing the user privileges of the parent object</td>
</tr>
<tr>
<td>ALIAS_NAME</td>
<td>Name for referencing to child definitions with dynamic submits</td>
</tr>
<tr>
<td>IS_STATIC</td>
<td>The is_static flag defines whether the job is to be statically or dynamically submitted.</td>
</tr>
<tr>
<td>IS_DISABLED</td>
<td>Flag indicating the the child should be executed or skipped</td>
</tr>
<tr>
<td>INT_NAME</td>
<td>The interval id is the ID of the interval used to check whether the child is enabled.</td>
</tr>
<tr>
<td>ENABLE_CONDITION</td>
<td>The enable condition, if completed, determines whether a child is enabled or disabled. The condition is evaluated at the time of the submit; any parameter values must therefore already be known at this time. The basic idea is to enable parameter-controlled process variants with the help of the condition.</td>
</tr>
<tr>
<td>ENABLE_MODE</td>
<td>The enable mode determines how the results of the enable condition and the enable interval are linked to one another. The possibilities are AND and OR. In the first case, a child will only be enabled if both the enable interval and the condition give cause for this. In the latter case, only one of the two has to give the go-ahead. If either condition is missing, the value for enable mode is irrelevant.</td>
</tr>
<tr>
<td>PRIORITY</td>
<td>The nice value that has been added to the children</td>
</tr>
</tbody>
</table>
### Field Description

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUSPEND</td>
<td>Determines whether the child is to be suspended for the submit</td>
</tr>
<tr>
<td>RESUME_AT</td>
<td>If the job is to be submitted as being suspended, an automatic resume takes place at the given time.</td>
</tr>
<tr>
<td>RESUME_IN</td>
<td>If the job is to be submitted as being suspended, an automatic resume takes place after the given number of time units.</td>
</tr>
<tr>
<td>RESUME_BASE</td>
<td>Specified time unit for RESUME_IN</td>
</tr>
<tr>
<td>MERGE_MODE</td>
<td>Determines how the condition handles the same object that occurs more than once in the submission hierarchy</td>
</tr>
<tr>
<td>EST_NAME</td>
<td>An Exit State Translation that is used to translate the Exit States of the children to the Exit States of the parents</td>
</tr>
<tr>
<td>IGNORED_DEPENDENCIES</td>
<td>List with the names of the dependencies for ignoring the dependencies of the parents</td>
</tr>
</tbody>
</table>

Table 22.40.: Description of the output structure of the show job definition subtable

### REQUIRED_JOBS

The layout of the REQUIRED_JOBS table is shown in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>The repository object Id</td>
</tr>
<tr>
<td>NAME</td>
<td>The object name</td>
</tr>
<tr>
<td>DEPENDENT_ID</td>
<td>The full path name of the dependent object</td>
</tr>
<tr>
<td>DEPENDENTNAME</td>
<td>The full path name of the dependent object</td>
</tr>
<tr>
<td>DEPENDENTTYPE</td>
<td>The type of dependent object (JOB, BATCH or MILESTONE)</td>
</tr>
<tr>
<td>DEPENDENTPRIVS</td>
<td>String containing the user privileges of the dependent object</td>
</tr>
<tr>
<td>REQUIRED_ID</td>
<td>The full path name of the required object</td>
</tr>
<tr>
<td>REQUIREDNAME</td>
<td>The full path name of the required object</td>
</tr>
<tr>
<td>REQUIREDTYPE</td>
<td>The type of required object (JOB, BATCH or MILESTONE)</td>
</tr>
</tbody>
</table>

Continued on next page
Continued from previous page

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>REQUIREDPRIVS</td>
<td>String containing the user privileges of the required object</td>
</tr>
<tr>
<td>UNRESOLVED_HANDLING</td>
<td>Defines what to do if the required object cannot be found</td>
</tr>
<tr>
<td>MODE</td>
<td>The Dependency Mode states the context in which the list of dependencies has to be viewed. The following options are available: ALL and ANY.</td>
</tr>
<tr>
<td>STATE_SELECTION</td>
<td>The State Selection defines how the required Exit States are determined. The options here are FINAL, ALL_REACHABLE, UNREACHABLE and DEFAULT. In the case of FINAL, the required Exit States can be explicitly listed.</td>
</tr>
<tr>
<td>CONDITION</td>
<td>The additional conditions must be fulfilled.</td>
</tr>
<tr>
<td>STATES</td>
<td>Comma-separated list of permitted Exit States that the required object has to achieve to fulfil the dependencies</td>
</tr>
<tr>
<td>RESOLVE_MODE</td>
<td>The Resolve Mode defines the context in which the dependency is to be resolved. The possible values are:</td>
</tr>
<tr>
<td></td>
<td>Value</td>
</tr>
<tr>
<td></td>
<td>internal</td>
</tr>
<tr>
<td></td>
<td>both</td>
</tr>
<tr>
<td></td>
<td>external</td>
</tr>
<tr>
<td>EXPIRED_AMOUNT</td>
<td>When resolving an external dependency, the time when the required job or batch was active plays a role. The expired amount defines for how many time units this may lie in the past.</td>
</tr>
<tr>
<td>EXPIRED_BASE</td>
<td>The expired base defines the time unit for the expired amount</td>
</tr>
<tr>
<td>SELECT_CONDITION</td>
<td>The select condition defines a condition that must be fulfilled so that a job or batch can be regarded as being a required job.</td>
</tr>
</tbody>
</table>

Table 22.41.: Description of the output structure of the show job definition subtable
**DEPENDENT_JOBS**  The layout of the DEPENDENT_JOBS table is shown in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>The repository object Id</td>
</tr>
<tr>
<td>NAME</td>
<td>The object name</td>
</tr>
<tr>
<td>DEPENDENT_ID</td>
<td>The full path name of the dependent object</td>
</tr>
<tr>
<td>DEPENDENTNAME</td>
<td>The full path name of the dependent object</td>
</tr>
<tr>
<td>DEPENDENTTYPE</td>
<td>The type of dependent object (JOB, BATCH or MILESTONE)</td>
</tr>
<tr>
<td>DEPENDENTPRIVS</td>
<td>String containing the user privileges of the dependent object</td>
</tr>
<tr>
<td>REQUIRED_ID</td>
<td>The full path name of the required object</td>
</tr>
<tr>
<td>REQUIREDNAME</td>
<td>The full path name of the required object</td>
</tr>
<tr>
<td>REQUIREDTYPE</td>
<td>The type of required object (JOB, BATCH or MILESTONE)</td>
</tr>
<tr>
<td>REQUIREDPRIVS</td>
<td>String containing the user privileges of the required object</td>
</tr>
<tr>
<td>UNRESOLVED_HANDLING</td>
<td>Defines what to do if the required object cannot be found</td>
</tr>
<tr>
<td>MODE</td>
<td>The Dependency Mode states the context in which the list of dependencies has to be viewed. The following options are available: ALL and ANY.</td>
</tr>
<tr>
<td>STATE_SELECTION</td>
<td>The State Selection defines how the required Exit States are determined. The options here are FINAL, ALL_REACHABLE, UNREACHABLE and DEFAULT. In the case of FINAL, the required Exit States can be explicitly listed.</td>
</tr>
<tr>
<td>CONDITION</td>
<td>The additional conditions must be fulfilled.</td>
</tr>
<tr>
<td>STATES</td>
<td>Comma-separated list of permitted Exit States that the required object has to achieve to fulfil the dependencies</td>
</tr>
</tbody>
</table>

*Continued on next page*
Continued from previous page

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RESOLVE_MODE</td>
<td>The Resolve Mode defines the context in which the dependency is to be resolved. The possible values are:</td>
</tr>
<tr>
<td></td>
<td>Value</td>
</tr>
<tr>
<td></td>
<td>internal</td>
</tr>
<tr>
<td></td>
<td>both</td>
</tr>
<tr>
<td></td>
<td>external</td>
</tr>
<tr>
<td>EXPIRED_AMOUNT</td>
<td>When resolving an external dependency, the time when the required job or batch was active plays a role. The expired amount defines for how many time units this may lie in the past.</td>
</tr>
<tr>
<td>EXPIRED_BASE</td>
<td>The expired base defines the time unit for the expired amount</td>
</tr>
<tr>
<td>SELECT_CONDITION</td>
<td>The select condition defines a condition that must be fulfilled so that a job or batch can be regarded as being a required job.</td>
</tr>
</tbody>
</table>

Table 22.42.: Description of the output structure of the show job definition subtable

**REQUIRED_RESOURCES**  The layout of the REQUIRED_RESOURCES table is shown in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>The repository object Id</td>
</tr>
<tr>
<td>RESOURCE_ID</td>
<td>Full path name of the required Named Resource</td>
</tr>
<tr>
<td>RESOURCE_NAME</td>
<td>Full path name of the required Named Resource</td>
</tr>
<tr>
<td>RESOURCE_USAGE</td>
<td>The usage of the required resource (STATIC, SYSTEM or SYNCHRONIZING)</td>
</tr>
<tr>
<td>RESOURCE_PRIVS</td>
<td>String containing the user privileges of the Named Resource</td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMOUNT</td>
<td>The required amount with System or Synchronizing Resources</td>
</tr>
<tr>
<td>KEEP_MODE</td>
<td>The Keep_Mode specifies the time at which the resource is released (FINISH, JOB_FINAL oder FINAL)</td>
</tr>
<tr>
<td>IS_STICKY</td>
<td>Indicates whether the resource allocation for subsequent jobs is retained</td>
</tr>
<tr>
<td>STICKY_NAME</td>
<td>Optional name of the sticky resource request</td>
</tr>
<tr>
<td>STICKY_PARENT</td>
<td>Parent Job Definition within which the sticky requirement is handled</td>
</tr>
<tr>
<td>RESOURCE_STATE_MAPPING</td>
<td>The Resource State Mapping defines how and whether the state of the resource is to be changed after the job has finished.</td>
</tr>
<tr>
<td>EXPIRED_AMOUNT</td>
<td>Die number of units. If the Expired Amount is positive, the state change must have occurred within the specified period. If it is negative, the state change must have occurred earlier than the specified period.</td>
</tr>
<tr>
<td>EXPIRED_BASE</td>
<td>The time unit for specifying the operation</td>
</tr>
<tr>
<td>IGNORE_ON_RERUN</td>
<td>This flag indicates if the expire condition should be ignored in case of a rerun.</td>
</tr>
<tr>
<td>LOCKMODE</td>
<td>The lockmode for allocating Synchronizing Resources (N, S, SX, X)</td>
</tr>
<tr>
<td>STATES</td>
<td>Comma-separated list of permitted Exit States that the required object has to achieve to fulfil the dependencies</td>
</tr>
<tr>
<td>DEFINITION</td>
<td>(REQUIREMENT, FOOTPRINT, FOLDER or ENVIRONMENT)</td>
</tr>
<tr>
<td>ORIGIN</td>
<td>Name of the Resource Request Definition, invalid in the case of a complete request</td>
</tr>
<tr>
<td>CONDITION</td>
<td>The optional condition that can be defined for requests for Static Resources</td>
</tr>
<tr>
<td>RESOURCE_STATE_PROFILE_ID</td>
<td>Id of the Resource State Profile of the Named Resource</td>
</tr>
<tr>
<td>RESOURCE_STATE_PROFILE_NAME</td>
<td>Name of the used Resource State Mapping</td>
</tr>
</tbody>
</table>

Table 22.43.: Description of the output structure of the show job definition subtable
show named resource

Purpose

The purpose of the show named resource statement is to get detailed information about the named resource.

Syntax

The syntax for the show named resource statement is

\[
\text{show} \ [ \text{condensed} ] \ \text{named resource} \ \text{id1, id2} \ [ \text{with} \ \text{EXPAND} ]
\]

\[
\text{EXPAND:}
\begin{align*}
\text{expand} &= \text{none} \\
| & \text{expand} = < ( \text{id} \{, \text{id} ) | \text{all} >
\end{align*}
\]

Description

The show named resource statement gives you detailed information about the Named Resource.

\textbf{expand} Since the number of job definitions in the table JOB_DEFINITIONS can become very large, by default they are not all displayed. If the option \textbf{expand = all} is used, all the job definitions as well as their parent folder and the folder hierarchy are outputted. Individual paths in the hierarchy can be selected by specifying individual (folder) IDs.

Output

This statement returns an output structure of type record.

\textbf{Output Description} The data items of the output are described in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>The repository object Id</td>
</tr>
<tr>
<td>NAME</td>
<td>Name of the Named Resource</td>
</tr>
<tr>
<td>OWNER</td>
<td>Owner of the Named Resource</td>
</tr>
<tr>
<td>USAGE</td>
<td>The Usage field specifies the Resource type.</td>
</tr>
</tbody>
</table>

Continued on next page
### Field Description

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INHERIT_PRIVS</td>
<td>Privileges that are inherited from the parent folder</td>
</tr>
<tr>
<td>RESOURCE_STATE_PROFILE</td>
<td>This is the Resource State Profile assigned to the resource.</td>
</tr>
<tr>
<td>FACTOR</td>
<td>This is the default factor by which Resource Requirement Amounts are multiplied if nothing else has been specified for the resource.</td>
</tr>
<tr>
<td>COMMENT</td>
<td>Comment if defined</td>
</tr>
<tr>
<td>COMMENTTYPE</td>
<td>Type of comment if a comment is defined</td>
</tr>
<tr>
<td>CREATOR</td>
<td>Name of the user created the object</td>
</tr>
<tr>
<td>CREATE_TIME</td>
<td>Date and time of object creation</td>
</tr>
<tr>
<td>CHANGER</td>
<td>Name of the last user changed the object</td>
</tr>
<tr>
<td>CHANGE_TIME</td>
<td>Date and time of object change</td>
</tr>
<tr>
<td>PRIVS</td>
<td>String containing the users privileges on the object</td>
</tr>
<tr>
<td>RESOURCES</td>
<td>These are the instances of the Named Resource.</td>
</tr>
<tr>
<td></td>
<td>See also table 22.45 on page 371</td>
</tr>
<tr>
<td>PARAMETERS</td>
<td>These are the defined parameters of the Named Resource.</td>
</tr>
<tr>
<td></td>
<td>See also table 22.46 on page 372</td>
</tr>
<tr>
<td>JOB_DEFINITIONS</td>
<td>These are the job definitions that request the Named Resource.</td>
</tr>
<tr>
<td></td>
<td>See also table 22.47 on page 373</td>
</tr>
</tbody>
</table>

Table 22.44.: Description of the output structure of the show named resource statement

### RESOURCES

The layout of the RESOURCES table is shown in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>The repository object Id</td>
</tr>
<tr>
<td>SCOPE</td>
<td>The names of the Scopes, Submitted Entities, Scheduling Entities or folders that offer the respective Named Resource are shown here.</td>
</tr>
<tr>
<td>TYPE</td>
<td>This is the resource type.</td>
</tr>
<tr>
<td>OWNER</td>
<td>The group owning the object</td>
</tr>
</tbody>
</table>

Continued on next page
Continued from previous page

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATE</td>
<td>Indicates the state of the resource</td>
</tr>
<tr>
<td>REQUESTABLE_AMOUNT</td>
<td>The maximum amount of resources that can be requested by a job</td>
</tr>
<tr>
<td>AMOUNT</td>
<td>The amount states the current number of instances of the Named Resource for this scope or jobserver.</td>
</tr>
<tr>
<td>FREE_AMOUNT</td>
<td>The Free Amount designates the total number of instances of a resource in the selected scope or jobserver that have not yet been allocated to jobs.</td>
</tr>
<tr>
<td>IS_ONLINE</td>
<td>Indicates whether the resource is online or not</td>
</tr>
<tr>
<td>PRIVS</td>
<td>String containing the users privileges on the object</td>
</tr>
</tbody>
</table>

Table 22.45.: Description of the output structure of the show named resource subtable

PARAMETERS  

The layout of the PARAMETERS table is shown in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>The repository object Id</td>
</tr>
<tr>
<td>NAME</td>
<td>Name of the parameter</td>
</tr>
<tr>
<td>TYPE</td>
<td>This is the parameter type. Local or Local Constant</td>
</tr>
<tr>
<td>DEFAULT_VALUE</td>
<td>With the Default Value, we differentiate between Constants and Local Constants. It is the value of the parameter for Constants and the default value for Local Constants.</td>
</tr>
<tr>
<td>TAG</td>
<td>The tag serves as a kind of heading for the comment and is optional.</td>
</tr>
<tr>
<td>COMMENT</td>
<td>Comment if defined</td>
</tr>
<tr>
<td>COMMENTTYPE</td>
<td>Type of comment if a comment is defined</td>
</tr>
</tbody>
</table>

Table 22.46.: Description of the output structure of the show named resource subtable
**JOB DEFINITIONS**  The layout of the JOB DEFINITIONS table is shown in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>The repository object Id</td>
</tr>
<tr>
<td>NAME</td>
<td>Name of the job definition</td>
</tr>
<tr>
<td>AMOUNT</td>
<td>The amount of the resource that is required by the job</td>
</tr>
<tr>
<td>KEEP_MODE</td>
<td>The value of the Keep parameter for the resource request from the job</td>
</tr>
<tr>
<td>IS_STICKY</td>
<td>Indicates whether it is a Sticky Request or not</td>
</tr>
<tr>
<td>STICKY_NAME</td>
<td>Optional name of the sticky resource request</td>
</tr>
<tr>
<td>STICKY_PARENT</td>
<td>Parent Job Definition within which the sticky requirement is handled</td>
</tr>
<tr>
<td>RESOURCE_STATE_MAPPING</td>
<td>If a Resource State Mapping was specified in the resource request, it is displayed here.</td>
</tr>
<tr>
<td>EXPIRED_AMOUNT</td>
<td>The number of units. If the Expired Amount is positive, this means that the state change cannot have taken place longer ago than the given maximum time. If the amount is negative, it must have taken place at least as long ago as the given minimum time.</td>
</tr>
<tr>
<td>EXPIRED_BASE</td>
<td>The unit in minutes, hours, days, weeks, months and years</td>
</tr>
<tr>
<td>IGNORE_ON_RERUN</td>
<td>This flag indicates if the expire condition should be ignored in case of a rerun.</td>
</tr>
<tr>
<td>LOCKMODE</td>
<td>The lockmode describes the mode for accessing this resource (exclusive, shared, etc.).</td>
</tr>
<tr>
<td>STATES</td>
<td>Multiple states that are acceptable for this job are separated by commas.</td>
</tr>
<tr>
<td>CONDITION</td>
<td>The condition that can be defined for requests for Static Resources</td>
</tr>
<tr>
<td>PRIVS</td>
<td>String containing the users privileges on the object</td>
</tr>
</tbody>
</table>

Table 22.47.: Description of the output structure of the show named resource subtable
**show resource**

**Purpose**

*Purpose* The purpose of the *show resource* statement is to get detailed information about the resource.

**Syntax**

*Syntax* The syntax for the *show resource* statement is

```
show RESOURCE_URL

RESOURCE_URL:
  resource identifier [, identifier] in folderpath
| resource identifier [, identifier] in serverpath
```

**Description**

*Description* The *show resource* statement gives you detailed information about the resource.

**Output**

*Output* This statement returns an output structure of type record.

**Output Description** The data items of the output are described in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>The repository object Id</td>
</tr>
<tr>
<td>NAME</td>
<td>Name of the resource</td>
</tr>
<tr>
<td>SCOPENAME</td>
<td>Name of the scope in which the pool was created</td>
</tr>
<tr>
<td>SCOPE_TYPE</td>
<td>This field indicates whether it is a scope or job server.</td>
</tr>
<tr>
<td>OWNER</td>
<td>The group owning the object</td>
</tr>
<tr>
<td>LINK_ID</td>
<td>Id of the referenced resource</td>
</tr>
<tr>
<td>LINK_SCOPE</td>
<td>Scope name of the referenced resource</td>
</tr>
<tr>
<td>LINK_SCOPE_TYPE</td>
<td>Scope name of the referenced resource</td>
</tr>
<tr>
<td>BASE_ID</td>
<td>Id of the ultimately referenced resource</td>
</tr>
</tbody>
</table>

*Continued on next page*
<table>
<thead>
<tr>
<th><strong>Field</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>BASE_SCOPE</td>
<td>Scope name of the ultimately referenced resource</td>
</tr>
<tr>
<td>MANAGER_ID</td>
<td>Id of the Managing Pool</td>
</tr>
<tr>
<td>MANAGER_NAME</td>
<td>Name of the Managing Pool</td>
</tr>
<tr>
<td>MANAGER_SCOPENAME</td>
<td>Name of the scope in which the Managing Pool was created</td>
</tr>
<tr>
<td>MANAGER_SCOPE_TYPE</td>
<td>This field indicates whether it is a scope or job server.</td>
</tr>
<tr>
<td>USAGE</td>
<td>The Usage field specifies the Resource type.</td>
</tr>
<tr>
<td>RESOURCE_STATE_PROFILE</td>
<td>This is the Resource State Profile assigned to the resource.</td>
</tr>
<tr>
<td>COMMENT</td>
<td>Comment if defined</td>
</tr>
<tr>
<td>COMMENTTYPE</td>
<td>Type of comment if a comment is defined</td>
</tr>
<tr>
<td>TAG</td>
<td>The tag is an optional short name for the resource.</td>
</tr>
<tr>
<td>STATE</td>
<td>The state is the current state of the resource in this scope or jobserver.</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>The timestamp indicates the last time the Resource State changed.</td>
</tr>
<tr>
<td>REQUESTABLE_AMOUNT</td>
<td>The maximum amount of resources that can be requested by a job</td>
</tr>
<tr>
<td>DEFINED_AMOUNT</td>
<td>The amount that is available if the resource is not pooled</td>
</tr>
<tr>
<td>AMOUNT</td>
<td>The actual available amount</td>
</tr>
<tr>
<td>FREE_AMOUNT</td>
<td>The Free_Amount designates the total number of instances of a resource that have not yet been allocated to jobs.</td>
</tr>
<tr>
<td>IS_ONLINE</td>
<td>Is_Online is an indicator that states whether the resource is online or not.</td>
</tr>
<tr>
<td>FACTOR</td>
<td>This is the correction factor by which the requested amount is multiplied.</td>
</tr>
<tr>
<td>TRACE_INTERVAL</td>
<td>The Trace_Interval is the minimum time in seconds between when Trace Records are written.</td>
</tr>
<tr>
<td>TRACE_BASE</td>
<td>The Trace_Base is the basis for the valuation period.</td>
</tr>
</tbody>
</table>

*Continued on next page*
Continued from previous page

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRACE_BASE_MULTIPLIER</td>
<td>The Base_Multiplier determines the multiplication factor of the Trace_Base.</td>
</tr>
<tr>
<td>TD0_AVG</td>
<td>The average resource allocation of the last $B \times M^0$ seconds</td>
</tr>
<tr>
<td>TD1_AVG</td>
<td>The average resource allocation of the last $B \times M^1$ seconds</td>
</tr>
<tr>
<td>TD2_AVG</td>
<td>The average resource allocation of the last $B \times M^2$ seconds</td>
</tr>
<tr>
<td>LW_AVG</td>
<td>The average allocation since the last time a Trace Record was written</td>
</tr>
<tr>
<td>LAST_WRITE</td>
<td>The time the last Trace Record was written</td>
</tr>
<tr>
<td>CREATOR</td>
<td>Name of the user created the object</td>
</tr>
<tr>
<td>CREATE_TIME</td>
<td>Date and time of object creation</td>
</tr>
<tr>
<td>CHANGER</td>
<td>Name of the last user changed the object</td>
</tr>
<tr>
<td>CHANGE_TIME</td>
<td>Date and time of object change</td>
</tr>
<tr>
<td>PRIVS</td>
<td>String containing the users privileges on the object</td>
</tr>
<tr>
<td>ALLOCATIONS</td>
<td>This is a table of resource allocations.</td>
</tr>
<tr>
<td>PARAMETERS</td>
<td>Additional information about a resource can be saved in the Parameters tab</td>
</tr>
</tbody>
</table>

Table 22.48.: Description of the output structure of the show resource statement

**ALLOCATIONS** The layout of the ALLOCATIONS table is shown in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>The repository object Id</td>
</tr>
<tr>
<td>JOBID</td>
<td>This is the Id of the job instance that was started with either a direct submit of the job or by a submit of the Master Batch or Master Job.</td>
</tr>
<tr>
<td>MASTERID</td>
<td>This is the Id of the job or batch instance that was started as a Master Job and contains the current job as a child.</td>
</tr>
<tr>
<td>JOBTYPE</td>
<td>This is the Id of the job.</td>
</tr>
</tbody>
</table>

Continued on next page
Continued from previous page

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>JOBNAME</td>
<td>This is the name of the job.</td>
</tr>
<tr>
<td>AMOUNT</td>
<td>This is the available amount.</td>
</tr>
<tr>
<td>KEEP_MODE</td>
<td>The Keep parameter defines when the job releases the resource. The following options are available: KEEP, NO KEEP and KEEP FINAL.</td>
</tr>
<tr>
<td>IS_STICKY</td>
<td>The resource is only released if there are no other Sticky Requests for this Named Resource in the same batch.</td>
</tr>
<tr>
<td>STICKY_NAME</td>
<td>Optional name of the sticky resource request</td>
</tr>
<tr>
<td>STICKY_PARENT</td>
<td>Parent job within which the sticky request is evaluated</td>
</tr>
<tr>
<td>STICKY_PARENT_TYPE</td>
<td>Type of the parent within which the sticky requirement is evaluated</td>
</tr>
<tr>
<td>LOCKMODE</td>
<td>The lockmode defines which access mode is used to allocate the resource to the current job.</td>
</tr>
<tr>
<td>RSM_NAME</td>
<td>The name of the Resource State Mapping</td>
</tr>
<tr>
<td>TYPE</td>
<td>The type of allocation: Available, Blocked, Allocations, Master_Reservation, Reservation</td>
</tr>
<tr>
<td>TYPESORT</td>
<td>Aid for sorting the allocations</td>
</tr>
<tr>
<td>P</td>
<td>The priority of the job</td>
</tr>
<tr>
<td>EP</td>
<td>The effective priority of the job</td>
</tr>
<tr>
<td>PRIVS</td>
<td>String containing the users privileges on the object</td>
</tr>
</tbody>
</table>

Table 22.49.: Description of the output structure of the show resource subtable

PARAMETERS The layout of the PARAMETERS table is shown in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>The repository object Id</td>
</tr>
<tr>
<td>NAME</td>
<td>Name of the parameter</td>
</tr>
<tr>
<td>EXPORT_NAME</td>
<td>The export name defines the name under which the value of the parameter is exported to the process’s environment.</td>
</tr>
<tr>
<td>TYPE</td>
<td>This is the parameter type</td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IS_LOCAL</td>
<td>True for local parameters that are only visible for the job itself</td>
</tr>
<tr>
<td>EXPRESSION</td>
<td>Expression for the parameter type expression</td>
</tr>
<tr>
<td>DEFAULT_VALUE</td>
<td>The default value of the parameter</td>
</tr>
<tr>
<td>REFERENCE_TYPE</td>
<td>Type of object that is being referenced</td>
</tr>
<tr>
<td>REFERENCE_PATH</td>
<td>The path to the object that is being referenced</td>
</tr>
<tr>
<td>REFERENCE_PRIVS</td>
<td>The user’s privileges for the object that is being referenced</td>
</tr>
<tr>
<td>REFERENCE_PARAMETER</td>
<td>Name of the parameter that is being referenced</td>
</tr>
<tr>
<td>COMMENT</td>
<td>Comment if defined</td>
</tr>
<tr>
<td>COMMENTTYPE</td>
<td>Type of comment if a comment is defined</td>
</tr>
<tr>
<td>ID</td>
<td>The repository object Id</td>
</tr>
<tr>
<td>NAME</td>
<td>Name of the parameter</td>
</tr>
<tr>
<td>EXPORT_NAME</td>
<td>The export name defines the name under which the value of the parameter is exported to the process’s environment.</td>
</tr>
<tr>
<td>TYPE</td>
<td>This is the parameter type</td>
</tr>
<tr>
<td>IS_LOCAL</td>
<td>True for local parameters that are only visible for the job itself</td>
</tr>
<tr>
<td>EXPRESSION</td>
<td>Expression for the parameter type expression</td>
</tr>
<tr>
<td>DEFAULT_VALUE</td>
<td>The default value of the parameter</td>
</tr>
<tr>
<td>REFERENCE_TYPE</td>
<td>Type of object that is being referenced</td>
</tr>
<tr>
<td>REFERENCE_PATH</td>
<td>The path to the object that is being referenced</td>
</tr>
<tr>
<td>REFERENCE_PRIVS</td>
<td>The user’s privileges for the object that is being referenced</td>
</tr>
<tr>
<td>REFERENCE_PARAMETER</td>
<td>Name of the parameter that is being referenced</td>
</tr>
<tr>
<td>COMMENT</td>
<td>Comment if defined</td>
</tr>
<tr>
<td>COMMENTTYPE</td>
<td>Type of comment if a comment is defined</td>
</tr>
<tr>
<td>DEFINITION</td>
<td>This column indicates where a visible parameter is defined.</td>
</tr>
<tr>
<td>DEFINITION_TYPE</td>
<td>The type of the defining object</td>
</tr>
<tr>
<td>ID</td>
<td>The repository object Id</td>
</tr>
<tr>
<td>NAME</td>
<td>Name of the parameter</td>
</tr>
<tr>
<td>TYPE</td>
<td>This is the parameter type</td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IS_LOCAL</td>
<td>True for local parameters that are only visible for the job itself</td>
</tr>
<tr>
<td>REFERENCE_TYPE</td>
<td>Type of object that is referencing the parameter</td>
</tr>
<tr>
<td>REFERENCE_PATH</td>
<td>The path to the object that is referencing the parameter</td>
</tr>
<tr>
<td>REFERENCE_ID</td>
<td>The ID of the referenced job</td>
</tr>
<tr>
<td>REFERENCE_PRIVS</td>
<td>The user’s privileges for the object that is referencing the parameter</td>
</tr>
<tr>
<td>REFERENCE_PARAMETER</td>
<td>Name of the parameter that is being referenced</td>
</tr>
<tr>
<td>COMMENT</td>
<td>Comment if defined</td>
</tr>
<tr>
<td>COMMENTTYPE</td>
<td>Type of comment if a comment is defined</td>
</tr>
</tbody>
</table>

Table 22.50.: Description of the output structure of the show resource subtable
**show resource state definition**

**Purpose**

*Purpose* The purpose of the *show resource state definition* is to get detailed information about the specified resource state definition.

**Syntax**

*Syntax* The syntax for the *show resource state definition* statement is

```
show resource state definition statename
```

**Description**

*Description* The *show resource state definition* statement gives you detailed information about the Resource State Definition.

**Output**

*Output* This statement returns an output structure of type record.

**Output Description** The data items of the output are described in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>The repository object Id</td>
</tr>
<tr>
<td>NAME</td>
<td>The object name</td>
</tr>
<tr>
<td>COMMENT</td>
<td>Comment if defined</td>
</tr>
<tr>
<td>COMMENTTYPE</td>
<td>Type of comment if a comment is defined</td>
</tr>
<tr>
<td>CREATOR</td>
<td>Name of the user created the object</td>
</tr>
<tr>
<td>CREATE_TIME</td>
<td>Date and time of object creation</td>
</tr>
<tr>
<td>CHANGER</td>
<td>Name of the last user changed the object</td>
</tr>
<tr>
<td>CHANGE_TIME</td>
<td>Date and time of object change</td>
</tr>
<tr>
<td>PRIVS</td>
<td>String containing the users privileges on the object</td>
</tr>
</tbody>
</table>

Table 22.51.: Description of the output structure of the *show resource state definition* statement
show resource state mapping

Purpose

The purpose of the `show resource state mapping` statement is to get detailed information about the specified mapping.

Syntax

The syntax for the `show resource state mapping` statement is

```
show resource state mapping profilename
```

Description

The `show resource state mapping` statement gives you detailed information about the specified mapping.

Output

This statement returns an output structure of type record.

Output Description

The data items of the output are described in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>The repository object Id</td>
</tr>
<tr>
<td>NAME</td>
<td>Name of the Resource State Mapping</td>
</tr>
<tr>
<td>COMMENT</td>
<td>Comment if defined</td>
</tr>
<tr>
<td>COMMENTTYPE</td>
<td>Type of comment if a comment is defined</td>
</tr>
<tr>
<td>CREATOR</td>
<td>Name of the user created the object</td>
</tr>
<tr>
<td>CREATE_TIME</td>
<td>Date and time of object creation</td>
</tr>
<tr>
<td>CHANGER</td>
<td>Name of the last user changed the object</td>
</tr>
<tr>
<td>CHANGE_TIME</td>
<td>Date and time of object change</td>
</tr>
<tr>
<td>PRIVS</td>
<td>String containing the users privileges on the object</td>
</tr>
<tr>
<td>MAPPINGS</td>
<td>A table with translations from the Exit State to the Resource State</td>
</tr>
</tbody>
</table>

See also table 22.53 on page 382

Table 22.52.: Description of the output structure of the `show resource state mapping` statement
The layout of the MAPPINGS table is shown in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESD_NAME</td>
<td>Name of the Exit State Definition</td>
</tr>
<tr>
<td>RSD_FROM</td>
<td>The original state of the resource</td>
</tr>
<tr>
<td>RSD_TO</td>
<td>The current state of the resource</td>
</tr>
</tbody>
</table>

Table 22.53.: Description of the output structure of the show resource state mapping subtable
show resource state profile

Purpose

The purpose of the *show resource state profile* is to get detailed information about the specified resource state profile.

Syntax

The syntax for the *show resource state profile* statement is

```
show resource state profile profilename
```

Description

The *show resource state profile* statement gives you detailed information about the specified Resource State Profile.

Output

This statement returns an output structure of type record.

**Output Description**  The data items of the output are described in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>The repository object Id</td>
</tr>
<tr>
<td>NAME</td>
<td>The object name</td>
</tr>
<tr>
<td>INITIAL_STATE</td>
<td>This field defines the initial state of the resource.</td>
</tr>
<tr>
<td>COMMENT</td>
<td>Comment if defined</td>
</tr>
<tr>
<td>COMMENTTYPE</td>
<td>Type of comment if a comment is defined</td>
</tr>
<tr>
<td>CREATOR</td>
<td>Name of the user created the object</td>
</tr>
<tr>
<td>CREATE_TIME</td>
<td>Date and time of object creation</td>
</tr>
<tr>
<td>CHANGER</td>
<td>Name of the last user changed the object</td>
</tr>
<tr>
<td>CHANGE_TIME</td>
<td>Date and time of object change</td>
</tr>
<tr>
<td>PRIVS</td>
<td>String containing the users privileges on the object</td>
</tr>
</tbody>
</table>

*Continued on next page*
The valid Resource States are shown in the Resource State column in the States table. See also table 22.55 on page 384.

Table 22.54.: Description of the output structure of the show resource state profile statement

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATES</td>
<td>The valid Resource States are shown in the Resource State column in the States table.</td>
</tr>
</tbody>
</table>

The layout of the STATES table is shown in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>The repository object Id</td>
</tr>
<tr>
<td>RSD_NAME</td>
<td>Name of the Resource State Definition</td>
</tr>
<tr>
<td>PRIVS</td>
<td>String containing the users privileges on the object</td>
</tr>
</tbody>
</table>

Table 22.55.: Description of the output structure of the show resource state profile subtable
show schedule

Purpose

The purpose of the show schedule statement is to get detailed information about the specified schedule.

Syntax

The syntax for the show schedule statement is

show schedule schedulepath

Description

The show schedule statement gives you detailed information about the specified schedule.

Output

This statement returns an output structure of type record.

Output Description The data items of the output are described in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>The repository object Id</td>
</tr>
<tr>
<td>NAME</td>
<td>The object name</td>
</tr>
<tr>
<td>OWNER</td>
<td>The group owning the object</td>
</tr>
<tr>
<td>INHERIT_PRIVS</td>
<td>Privileges that are inherited from the parent folder</td>
</tr>
<tr>
<td>INTERVAL</td>
<td>The name of the interval belonging to the schedule</td>
</tr>
<tr>
<td>TIME_ZONE</td>
<td>The time zone in which the schedule is to be calculated</td>
</tr>
<tr>
<td>ACTIVE</td>
<td>This field defines whether the schedule is marked as being active.</td>
</tr>
<tr>
<td>EFFACTIVE</td>
<td>This field defines whether the schedule is actually active. This can deviate from “active” due to the hierarchical organisation.</td>
</tr>
<tr>
<td>CREATOR</td>
<td>Name of the user created the object</td>
</tr>
<tr>
<td>CREATE_TIME</td>
<td>Date and time of object creation</td>
</tr>
</tbody>
</table>

Continued on next page
### Table 22.56: Description of the output structure of the show schedule statement

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHANGER</td>
<td>Name of the last user changed the object</td>
</tr>
<tr>
<td>CHANGE_TIME</td>
<td>Date and time of object change</td>
</tr>
<tr>
<td>PRIVS</td>
<td>String containing the users privileges on the object</td>
</tr>
<tr>
<td>COMMENT</td>
<td>Comment if defined</td>
</tr>
<tr>
<td>COMMENTTYPE</td>
<td>Type of comment if a comment is defined</td>
</tr>
</tbody>
</table>

Continued from previous page
show scheduled event

Purpose

The purpose of the show scheduled event is to get detailed information about the specified event.

Syntax

The syntax for the show scheduled event statement is

    show scheduled event schedulepath . eventname

Description

The show scheduled event statement gives you detailed information about the specified event.

Output

This statement returns an output structure of type record.

Output Description The data items of the output are described in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>The repository object Id</td>
</tr>
<tr>
<td>OWNER</td>
<td>The group owning the object</td>
</tr>
<tr>
<td>SCHEDULE</td>
<td>The Schedule that determines when the Scheduled Event is to take place</td>
</tr>
<tr>
<td>EVENT</td>
<td>The event that is triggered</td>
</tr>
<tr>
<td>ACTIVE</td>
<td>This flag indicates whether the Scheduled Event is labelled as being active.</td>
</tr>
<tr>
<td>EFF_ACTIVE</td>
<td>This flag indicates whether the Scheduled Event is actually active.</td>
</tr>
<tr>
<td>BROKEN</td>
<td>The Broken field can be used to check whether an error occurred when the job was submitted.</td>
</tr>
<tr>
<td>ERROR_CODE</td>
<td>If an error occurred while the job was being executed in the Time Scheduling, the returned error code is displayed in the Error_Code field. If no error occurred, this field remains empty.</td>
</tr>
</tbody>
</table>

Continued on next page
Continued from previous page

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERROR_MSG</td>
<td>If an error occurred while the job was being executed in the Time Scheduling, the returned error message is displayed in the Error Message field. If no error occurred, this field remains empty.</td>
</tr>
<tr>
<td>LAST_START</td>
<td>The last time the job is to be executed by the Scheduling System is shown here.</td>
</tr>
<tr>
<td>NEXT_START</td>
<td>The next scheduled time when the task is to be executed by the Scheduling System is shown here.</td>
</tr>
<tr>
<td>NEXT_CALC</td>
<td>The next time when a recalculation is to take place.</td>
</tr>
<tr>
<td>CREATOR</td>
<td>Name of the user created the object.</td>
</tr>
<tr>
<td>CREATE_TIME</td>
<td>Date and time of object creation.</td>
</tr>
<tr>
<td>CHANGER</td>
<td>Name of the last user changed the object.</td>
</tr>
<tr>
<td>CHANGE_TIME</td>
<td>Date and time of object change.</td>
</tr>
<tr>
<td>PRIVS</td>
<td>String containing the users privileges on the object.</td>
</tr>
<tr>
<td>BACKLOG_HANDLING</td>
<td>The Backlog_Handling describes how events that should have been triggered following a downtime are to be handled.</td>
</tr>
<tr>
<td>SUSPEND_LIMIT</td>
<td>The Suspend Limit defines the delay after which a job is submitted in a suspended state.</td>
</tr>
<tr>
<td>EFFECTIVE_SUSPEND_LIMIT</td>
<td>The Suspend Limit defines the delay after which a job is submitted in a suspended state.</td>
</tr>
<tr>
<td>CALENDAR</td>
<td>This flag indicates whether calendar entries are created.</td>
</tr>
<tr>
<td>CALENDAR_HORIZON</td>
<td>The defined length of the period in days for which a calendar is created.</td>
</tr>
<tr>
<td>EFFECTIVECALENDAR_HORIZON</td>
<td>The effective length of the period in days for which a calendar is created</td>
</tr>
<tr>
<td>COMMENT</td>
<td>Comment if defined.</td>
</tr>
<tr>
<td>COMMENTTYPE</td>
<td>Type of comment if a comment is defined.</td>
</tr>
<tr>
<td>CALENDAR_TABLE</td>
<td>The table with the next start times.</td>
</tr>
</tbody>
</table>

Table 22.57.: Description of the output structure of the show scheduled event statement
**show scope**

**Purpose**

The purpose of the *show scope* statement is to get detailed information about a scope.

**Syntax**

The syntax for the *show scope* statement is

```
show < scope serverpath | jobserver serverpath > [ with EXPAND ]
```

**Description**

The *show scope* statement gives you detailed information about the scope.

**Output**

This statement returns an output structure of type record.

**Output Description** The data items of the output are described in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>The repository object Id</td>
</tr>
<tr>
<td>NAME</td>
<td>The object name</td>
</tr>
<tr>
<td>OWNER</td>
<td>The group owning the object</td>
</tr>
<tr>
<td>TYPE</td>
<td>The type of scope</td>
</tr>
<tr>
<td>INHERIT_PRIVS</td>
<td>Privileges that are inherited from the parent folder</td>
</tr>
<tr>
<td>IS_TERMINATE</td>
<td>This flag indicates whether a termination order exists.</td>
</tr>
<tr>
<td>IS_SUSPENDED</td>
<td>Indicates whether the scope is suspended</td>
</tr>
<tr>
<td>IS_ENABLED</td>
<td>The jobserver can only log on to the server if the Enable flag is set to YES.</td>
</tr>
</tbody>
</table>

*Continued on next page*
<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IS_REGISTERED</td>
<td>Defines whether the jobserver has sent a register command.</td>
</tr>
<tr>
<td>IS_CONNECTED</td>
<td>Indicates whether the jobserver is connected. The configuration on the server does not match the current configuration on the server.</td>
</tr>
<tr>
<td>HAS_ALTERED</td>
<td>The configuration on the server does not match the current configuration on the server.</td>
</tr>
<tr>
<td>STATE</td>
<td>This is the current state of the resource in this scope.</td>
</tr>
<tr>
<td>PID</td>
<td>The PID is the process identification number of the jobserver process on the respective host system.</td>
</tr>
<tr>
<td>NODE</td>
<td>The Node specifies the computer on which the jobserver is running. This field has a purely documentary character.</td>
</tr>
<tr>
<td>IDLE</td>
<td>The time that has elapsed since the last command. This only applies for job servers.</td>
</tr>
<tr>
<td>ERRMSG</td>
<td>This is the most recently outputted error message.</td>
</tr>
<tr>
<td>COMMENT</td>
<td>Comment if defined.</td>
</tr>
<tr>
<td>COMMENTTYPE</td>
<td>Type of comment if a comment is defined.</td>
</tr>
<tr>
<td>CREATOR</td>
<td>Name of the user who created the object.</td>
</tr>
<tr>
<td>CREATE_TIME</td>
<td>Date and time of object creation.</td>
</tr>
<tr>
<td>CHANGER</td>
<td>Name of the last user who changed the object.</td>
</tr>
<tr>
<td>CHANGE_TIME</td>
<td>Date and time of object change.</td>
</tr>
<tr>
<td>PRIVS</td>
<td>A string containing the user's privileges on the object.</td>
</tr>
<tr>
<td>RESOURCES</td>
<td>The resources present in this scope are displayed here. The configuration of the jobserver is described in the Config tab. The configuration variables are visible in the Config tab. See also table 22.60 on page 392.</td>
</tr>
<tr>
<td>CONFIG_ENVMAPPING</td>
<td>Whether and under which name the environment variables are visible is configured in this scope. The configuration of the jobserver is described in the Config tab. Whether and under which name the environment variables are visible is configured in this scope. See also table 22.61 on page 393.</td>
</tr>
<tr>
<td>CONFIG</td>
<td>The configuration of the jobserver is described in the Config tab. See also table 22.59 on page 391.</td>
</tr>
<tr>
<td>CONFIG_ENVMAPPING</td>
<td>See also table 22.61 on page 393. Continue on next page.</td>
</tr>
</tbody>
</table>
Continued from previous page

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PARAMETERS</td>
<td>Additional information about a resource can be saved in the Parameters tab</td>
</tr>
<tr>
<td></td>
<td>See also table 22.62 on page 393</td>
</tr>
</tbody>
</table>

Table 22.58.: Description of the output structure of the show scope statement

**RESOURCES** The layout of the RESOURCES table is shown in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>The repository object Id</td>
</tr>
<tr>
<td>NR_ID</td>
<td>Id of the Named Resource</td>
</tr>
<tr>
<td>NAME</td>
<td>Name of the Named Resource</td>
</tr>
<tr>
<td>USAGE</td>
<td>It is the usage of the Named Resource (STATIC, SYSTEM or SYNCHRONISING)</td>
</tr>
<tr>
<td>NR_PRIVS</td>
<td>String containing the abbreviations for the user privileges for this Named Resource</td>
</tr>
<tr>
<td>TAG</td>
<td>The tag is an optional short name for the resource.</td>
</tr>
<tr>
<td>OWNER</td>
<td>The group owning the object</td>
</tr>
<tr>
<td>LINK_ID</td>
<td>Id of the referenced resource</td>
</tr>
<tr>
<td>LINK SCOPE</td>
<td>Scope name of the referenced resource</td>
</tr>
<tr>
<td>LINK SCOPE TYPE</td>
<td>This field indicates whether it is a scope or job server.</td>
</tr>
<tr>
<td>STATE</td>
<td>The Resource State of the resource</td>
</tr>
<tr>
<td>REQUESTABLE_AMOUNT</td>
<td>The maximum amount of resources that can be requested by a job</td>
</tr>
<tr>
<td>AMOUNT</td>
<td>The actual amount that is available</td>
</tr>
<tr>
<td>FREE_AMOUNT</td>
<td>The Free_Amount that can be allocated</td>
</tr>
<tr>
<td>TOTAL_FREE_AMOUNT</td>
<td>Free_Amount available for allocations including the free amount of pooled resources if it is a pool</td>
</tr>
<tr>
<td>IS ONLINE</td>
<td>This is the availability status of the resource.</td>
</tr>
<tr>
<td>FACTOR</td>
<td>This is the correction factor by which the requested amount is multiplied.</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>The last time the Resource State changed.</td>
</tr>
</tbody>
</table>

Continued on next page
### Field Description

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCOPE</td>
<td>The scope in which the resource was created</td>
</tr>
<tr>
<td>MANAGER_ID</td>
<td>Id of the Managing Pool</td>
</tr>
<tr>
<td>MANAGER_NAME</td>
<td>Name of the Managing Pool</td>
</tr>
<tr>
<td>MANAGER_SCOPENAME</td>
<td>Name of the scope in which the Managing Pool was created</td>
</tr>
<tr>
<td>MANAGER_SCOPE_TYPE</td>
<td>This field indicates whether it is a scope or job server.</td>
</tr>
<tr>
<td>HAS_CHILDREN</td>
<td>Flag indicating whether a Pool Child has managed resources/pools. If it is not a pool, this is always FALSE.</td>
</tr>
<tr>
<td>POOL_CHILD</td>
<td>This flag indicates whether the displayed resource is a child of the pool.</td>
</tr>
<tr>
<td>TRACE_INTERVAL</td>
<td>The Trace_Interval is the minimum time in seconds between when Trace Records are written.</td>
</tr>
<tr>
<td>TRACE_BASE</td>
<td>The Trace_Base is the basis for the valuation period (B).</td>
</tr>
<tr>
<td>TRACE_BASE_MULTIPLIER</td>
<td>The Base Multiplier determines the multiplication factor (M) of the Trace_Base.</td>
</tr>
<tr>
<td>TD0_AVG</td>
<td>The average resource allocation of the last $B \times M^0$ seconds</td>
</tr>
<tr>
<td>TD1_AVG</td>
<td>The average resource allocation of the last $B \times M^1$ seconds</td>
</tr>
<tr>
<td>TD2_AVG</td>
<td>The average resource allocation of the last $B \times M^2$ seconds</td>
</tr>
<tr>
<td>LW_AVG</td>
<td>The average allocation since the last time a Trace Record was written</td>
</tr>
<tr>
<td>LAST_WRITE</td>
<td>The time the last Trace Record was written</td>
</tr>
<tr>
<td>PRIVS</td>
<td>String containing the users privileges on the object</td>
</tr>
</tbody>
</table>

Table 22.59.: Description of the output structure of the show scope subtable

### CONFIG

The layout of the CONFIG table is shown in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>KEY</td>
<td>The name of the configuration variable</td>
</tr>
</tbody>
</table>

Continued on next page
### Table 22.60.: Description of the output structure of the show scope subtable

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VALUE</td>
<td>The value of the configuration variable</td>
</tr>
<tr>
<td>LOCAL</td>
<td>Indicates whether the Key Value Pair is defined at local or parent level</td>
</tr>
<tr>
<td>ANCESTOR_SCOPE</td>
<td>This is the scope in which the Key Value Pair is defined.</td>
</tr>
<tr>
<td>ANCESTOR_VALUE</td>
<td>This is the value that is defined at parent level.</td>
</tr>
</tbody>
</table>

### CONFIG_ENVMAPPING

The layout of the CONFIG_ENVMAPPING table is shown in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>KEY</td>
<td>Name of the environment variable</td>
</tr>
<tr>
<td>VALUE</td>
<td>Name of the environment variable that is to be set</td>
</tr>
<tr>
<td>LOCAL</td>
<td>Indicates whether the Key Value Pair is defined at local or parent level</td>
</tr>
<tr>
<td>ANCESTOR_SCOPE</td>
<td>This is the scope in which the Key Value Pair is defined.</td>
</tr>
<tr>
<td>ANCESTOR_VALUE</td>
<td>This is the value that is defined at parent level.</td>
</tr>
</tbody>
</table>

### PARAMETERS

The layout of the PARAMETERS table is shown in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>The repository object Id</td>
</tr>
<tr>
<td>NAME</td>
<td>Name of the parameter</td>
</tr>
<tr>
<td>EXPORT_NAME</td>
<td>The export name defines the name under which the value of the parameter is exported to the process’s environment.</td>
</tr>
<tr>
<td>TYPE</td>
<td>This is the parameter type</td>
</tr>
<tr>
<td>IS_LOCAL</td>
<td>True for local parameters that are only visible for the job itself</td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXPRESSION</td>
<td>Expression for the parameter type expression</td>
</tr>
<tr>
<td>DEFAULT_VALUE</td>
<td>The default value of the parameter</td>
</tr>
<tr>
<td>REFERENCE_TYPE</td>
<td>Type of object that is being referenced</td>
</tr>
<tr>
<td>REFERENCE_PATH</td>
<td>The path to the object that is being referenced</td>
</tr>
<tr>
<td>REFERENCE_PRIVS</td>
<td>The user’s privileges for the object that is being referenced</td>
</tr>
<tr>
<td>REFERENCEPARAMETER</td>
<td>Name of the parameter that is being referenced</td>
</tr>
<tr>
<td>COMMENT</td>
<td>Comment if defined</td>
</tr>
<tr>
<td>COMMENTTYPE</td>
<td>Type of comment if a comment is defined</td>
</tr>
<tr>
<td>ID</td>
<td>The repository object Id</td>
</tr>
<tr>
<td>NAME</td>
<td>Name of the parameter</td>
</tr>
<tr>
<td>EXPORT_NAME</td>
<td>The export name defines the name under which the value of the parameter is exported to the process’s environment.</td>
</tr>
<tr>
<td>TYPE</td>
<td>This is the parameter type</td>
</tr>
<tr>
<td>IS_LOCAL</td>
<td>True for local parameters that are only visible for the job itself</td>
</tr>
<tr>
<td>EXPRESSION</td>
<td>Expression for the parameter type expression</td>
</tr>
<tr>
<td>DEFAULT_VALUE</td>
<td>The default value of the parameter</td>
</tr>
<tr>
<td>REFERENCE_TYPE</td>
<td>Type of object that is being referenced</td>
</tr>
<tr>
<td>REFERENCE_PATH</td>
<td>The path to the object that is being referenced</td>
</tr>
<tr>
<td>REFERENCE_PRIVS</td>
<td>The user’s privileges for the object that is being referenced</td>
</tr>
<tr>
<td>REFERENCEPARAMETER</td>
<td>Name of the parameter that is being referenced</td>
</tr>
<tr>
<td>COMMENT</td>
<td>Comment if defined</td>
</tr>
<tr>
<td>COMMENTTYPE</td>
<td>Type of comment if a comment is defined</td>
</tr>
<tr>
<td>DEFINITION</td>
<td>This column indicates where a visible parameter is defined.</td>
</tr>
<tr>
<td>DEFINITION_TYPE</td>
<td>The type of the defining object</td>
</tr>
<tr>
<td>ID</td>
<td>The repository object Id</td>
</tr>
<tr>
<td>NAME</td>
<td>Name of the parameter</td>
</tr>
<tr>
<td>TYPE</td>
<td>This is the parameter type</td>
</tr>
<tr>
<td>IS_LOCAL</td>
<td>True for local parameters that are only visible for the job itself</td>
</tr>
<tr>
<td>REFERENCE_TYPE</td>
<td>Type of object that is referencing the parameter</td>
</tr>
</tbody>
</table>

Continued on next page
Continued from previous page

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>REFERENCE_PATH</td>
<td>The path to the object that is referencing the parameter</td>
</tr>
<tr>
<td>REFERENCE_ID</td>
<td>The ID of the referenced job</td>
</tr>
<tr>
<td>REFERENCE_PRIVS</td>
<td>The user’s privileges for the object that is referencing the parameter</td>
</tr>
<tr>
<td>REFERENCE_PARAMETER</td>
<td>Name of the parameter that is being referenced</td>
</tr>
<tr>
<td>COMMENT</td>
<td>Comment if defined</td>
</tr>
<tr>
<td>COMMENTTYPE</td>
<td>Type of comment if a comment is defined</td>
</tr>
</tbody>
</table>

Table 22.62.: Description of the output structure of the show scope subtable
show session

Purpose

Purpose  The purpose of the show session statement is to get more detailed information about the specified or the current session.

Syntax

Syntax  The syntax for the show session statement is

```
show session [ sid ]
```

Description

Description  The show session statement gives you detailed information about the specified or current session.

Output

Output  This statement returns an output structure of type record.

Output Description  The data items of the output are described in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>THIS</td>
<td>The current session is indicated in this field by an asterisk (*).</td>
</tr>
<tr>
<td>SESSIONID</td>
<td>The internal server Id for the session</td>
</tr>
<tr>
<td>START</td>
<td>Time when the connection was set up</td>
</tr>
<tr>
<td>USER</td>
<td>Name of the user name used for the session login</td>
</tr>
<tr>
<td>UID</td>
<td>Id of the user, jobserver or job</td>
</tr>
<tr>
<td>IP</td>
<td>IP address of the connecting sessions</td>
</tr>
<tr>
<td>IS_SSL</td>
<td>Indicates if the connection is an SSL/TLS connection</td>
</tr>
<tr>
<td>IS_AUTHENTICATED</td>
<td>Indicates if the client has been authenticated</td>
</tr>
<tr>
<td>TXID</td>
<td>Number of the last transaction that was executed by the session</td>
</tr>
<tr>
<td>IDLE</td>
<td>The number of seconds since the last statement from a session</td>
</tr>
</tbody>
</table>

Continued on next page
Continued from previous page

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIMEOUT</td>
<td>The idle time after which the session is automatically disconnected</td>
</tr>
<tr>
<td>STATEMENT</td>
<td>The statement that is currently being executed</td>
</tr>
</tbody>
</table>

Table 22.63.: Description of the output structure of the show session statement
**show system**

*Purpose*

*Purpose* The purpose of the *show system* statement is to get information about the actual configuration of the running server.

*Syntax*

*Syntax* The syntax for the *show system* statement is

```
show system
```

```
show system with lock
```

*Description*

*Description* The *show system* statement gives you detailed information about the current configuration of the running server.

*Output*

*Output* This statement returns an output structure of type record.

**Output Description** The data items of the output are described in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VERSION</td>
<td>The current version of the software</td>
</tr>
<tr>
<td>MAX_LEVEL</td>
<td>The maximum compatibility level of the software</td>
</tr>
<tr>
<td>BUILD_DATE</td>
<td>Date and time of the build</td>
</tr>
<tr>
<td>BUILD_HASH</td>
<td>The build hash is a unique hash value that indicates an exact development status. This value is often required when handling support requests.</td>
</tr>
<tr>
<td>NUM_CPU</td>
<td>The number of processors present in the system</td>
</tr>
<tr>
<td>MEM_USED</td>
<td>The amount of used memory</td>
</tr>
<tr>
<td>MEM_FREE</td>
<td>The amount of free memory</td>
</tr>
<tr>
<td>MEM_MAX</td>
<td>The maximum amount of memory that the server can use</td>
</tr>
</tbody>
</table>

*Continued on next page*
Field Description
STARTTIME The time when the server was started
UPTIME The time when the server started running
HITRATE The hit rate in the environment cache of the Scheduling Thread
LOCK_HWM The Lock_HWM shows the high water mark of active locks in the system. This field is only relevant if multiple writer threads are active.
LOCKS_REQUESTED The Locks_Requested field shows the total number of locks requested since server startup. This field is only relevant in case of multiple writer threads.
LOCKS_USED This field shows the number of locks currently in use. It is only relevant in case of multiple writer threads.
LOCKS_DISCARDED The field Locks_Discarded shows the number of locks removed from the system.
CNT_RW_TX The number of R/W Transactions since server startup
CNT_DL The number of deadlocks since server startup
CNT_WL The number of single threaded write worker transactions since server startup
WORKER A table with a list of the Worker Threads
See also table 22.65 on page 399
LOCKING STATUS The locking state provides information about the state of the internal locking. This field is only displayed if the with locks option is specified.

Table 22.64.: Description of the output structure of the show system statement

WORKER The layout of the WORKER table is shown in the table below.

Field Description
ID The repository object Id
TYPE The type of worker thread, Read/Write (RW) or Read Only (RO)
NAME The object name

Continued on next page
Table 22.65.: Description of the output structure of the show system subtable

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATE</td>
<td>The state of the worker</td>
</tr>
<tr>
<td>TIME</td>
<td>The time from which the worker is in a state</td>
</tr>
</tbody>
</table>
show trigger

Purpose

The purpose of the show trigger statement is to get detailed information about the specified trigger.

Syntax

The syntax for the show trigger statement is

```
show trigger triggername on TRIGGEROBJECT [ < noinverse | inverse > ]
```

TRIGGEROBJECT:

- resource identifier [. identifier] in folderpath
- job definition folderpath
- named resource identifier [. identifier]
- object monitor objecttypename
- resource identifier [. identifier] in serverpath

Description

The show trigger statement gives you detailed information about the specified trigger.

Output

This statement returns an output structure of type record.

Output Description

The data items of the output are described in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>The repository object Id</td>
</tr>
<tr>
<td>NAME</td>
<td>The object name</td>
</tr>
<tr>
<td>OBJECTTYPE</td>
<td>The type of object in which the trigger is defined</td>
</tr>
<tr>
<td>OBJECTNAME</td>
<td>Full path name of the object in which the trigger is defined</td>
</tr>
<tr>
<td>ACTIVE</td>
<td>The flag indicates whether the trigger is currently active.</td>
</tr>
<tr>
<td>ACTION</td>
<td>Type of triggered action: SUBMIT or RERUN</td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUBMIT_TYPE</td>
<td>The object type that is submitted when the trigger is activated</td>
</tr>
<tr>
<td>SUBMIT_NAME</td>
<td>Name of the job definition that is submitted</td>
</tr>
<tr>
<td>SUBMIT_SE_OWNER</td>
<td>The owner of the object that is submitted</td>
</tr>
<tr>
<td>SUBMIT_PRIVS</td>
<td>The privileges for the object that is to be submitted</td>
</tr>
<tr>
<td>MAIN_TYPE</td>
<td>Type of main job (job/batch)</td>
</tr>
<tr>
<td>MAIN_NAME</td>
<td>Name of the main job</td>
</tr>
<tr>
<td>MAIN_SE_OWNER</td>
<td>Owner of the main job</td>
</tr>
<tr>
<td>MAIN_PRIVS</td>
<td>Privileges for the main job</td>
</tr>
<tr>
<td>PARENT_TYPE</td>
<td>Type of parent job (job/batch)</td>
</tr>
<tr>
<td>PARENT_NAME</td>
<td>Name of the parent job</td>
</tr>
<tr>
<td>PARENT_SE_OWNER</td>
<td>Owner of the parent job</td>
</tr>
<tr>
<td>PARENT_PRIVS</td>
<td>Privileges for the parent job</td>
</tr>
<tr>
<td>TRIGGER_TYPE</td>
<td>The trigger type that describes when it is activated</td>
</tr>
<tr>
<td>MASTER</td>
<td>Indicates whether the trigger submitted a master or a child</td>
</tr>
<tr>
<td>IS_INVERSE</td>
<td>In case of an inverse trigger, the trigger is regarded to belong to the triggered job. The trigger can be regarded as some kind of callback function. This flag has no effects on the trigger’s behaviour.</td>
</tr>
<tr>
<td>SUBMIT_OWNER</td>
<td>The owner group that is used with the Submitted Entity</td>
</tr>
<tr>
<td>IS_CREATE</td>
<td>Indicates whether the trigger reacts to create events</td>
</tr>
<tr>
<td>IS_CHANGE</td>
<td>Indicates whether the trigger reacts to change events</td>
</tr>
<tr>
<td>IS_DELETE</td>
<td>Indicates whether the trigger reacts to delete events</td>
</tr>
<tr>
<td>IS_GROUP</td>
<td>Indicates whether the trigger handles the events as a group</td>
</tr>
<tr>
<td>MAX_RETRY</td>
<td>The maximum number of trigger activations in a single Submitted Entity</td>
</tr>
<tr>
<td>SUSPEND</td>
<td>Specifies whether the submitted object is suspended</td>
</tr>
</tbody>
</table>
### Description of the output structure of the show trigger statement

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RESUME_AT</td>
<td>Time of the automatic resume</td>
</tr>
<tr>
<td>RESUME_IN</td>
<td>Number of time units until the automatic resume</td>
</tr>
<tr>
<td>RESUME_BASE</td>
<td>Specified time unit for RESUME_IN</td>
</tr>
<tr>
<td>WARN</td>
<td>Specifies whether a warning has to be given when the activation limit is reached</td>
</tr>
<tr>
<td>LIMIT_STATE</td>
<td>This field specifies which state the triggering job acquires if the fire limit is reached. If the triggering job has a final state already, this specification is ignored. If the value is NONE, no state change takes place.</td>
</tr>
<tr>
<td>CONDITION</td>
<td>Conditional expression to define the trigger condition</td>
</tr>
<tr>
<td>CHECK_AMOUNT</td>
<td>The amount of CHECK_BASE units for checking the condition in the case of non-synchronised triggers</td>
</tr>
<tr>
<td>CHECK_BASE</td>
<td>Units for the CHECK_AMOUNT</td>
</tr>
<tr>
<td>COMMENT</td>
<td>Comment if defined</td>
</tr>
<tr>
<td>COMMENTTYPE</td>
<td>Type of comment if a comment is defined</td>
</tr>
<tr>
<td>CREATOR</td>
<td>Name of the user created the object</td>
</tr>
<tr>
<td>CREATE_TIME</td>
<td>Date and time of object creation</td>
</tr>
<tr>
<td>CHANGER</td>
<td>Name of the last user changed the object</td>
</tr>
<tr>
<td>CHANGE_TIME</td>
<td>Date and time of object change</td>
</tr>
<tr>
<td>STATES</td>
<td>A list of states that cause the trigger to be activated</td>
</tr>
<tr>
<td></td>
<td>See also table 22.67 on page 403</td>
</tr>
<tr>
<td>PARAMETERS</td>
<td>A list of states that cause the trigger to be activated</td>
</tr>
<tr>
<td></td>
<td>See also table 22.68 on page 404</td>
</tr>
</tbody>
</table>

Table 22.66.: Description of the output structure of the show trigger statement

### STATES

The layout of the STATES table is shown in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>The repository object Id</td>
</tr>
</tbody>
</table>

Continued on next page
**Field** | **Description**
--- | ---
FROM_STATE | The trigger is activated if this is the old Resource State
TO_STATE | The trigger is activated if this is the new Resource State or the Exit State of the object.

Table 22.67.: Description of the output structure of the show trigger subtable

**PARAMETERS** The layout of the PARAMETERS table is shown in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>The repository object Id</td>
</tr>
<tr>
<td>NAME</td>
<td>Name of the parameter that is set at the submit time.</td>
</tr>
<tr>
<td>EXPRESSION</td>
<td>An expression that is valuated in the context of the triggering object. The syntax is the same as the syntax in the trigger condition, except that here general expressions are allowed, and not just Boolean expressions.</td>
</tr>
</tbody>
</table>

Table 22.68.: Description of the output structure of the show trigger subtable
**show user**

**Purpose**

The purpose of the *show user* statement is to show detailed information about the user.

**Syntax**

The syntax for the *show user* statement is

```plaintext
show user [ username ]
```

**Description**

The *show user* statement gives you detailed information about the user.

**Output**

This statement returns an output structure of type record.

**Output Description**  The data items of the output are described in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>The repository object Id</td>
</tr>
<tr>
<td>NAME</td>
<td>The object name</td>
</tr>
<tr>
<td>IS_ENABLED</td>
<td>Flag that shows whether the user is allowed to log on</td>
</tr>
<tr>
<td>DEFAULT_GROUP</td>
<td>The default group of the users who are being used by the owners of the object</td>
</tr>
<tr>
<td>CONNECTION_TYPE</td>
<td>Indicates which security level of a connection is required.</td>
</tr>
<tr>
<td></td>
<td>1. <strong>plain</strong> – Every kind of connection is permitted</td>
</tr>
<tr>
<td></td>
<td>2. <strong>ssl</strong> – Only SSL-connections are permitted</td>
</tr>
<tr>
<td></td>
<td>3. <strong>ssl_auth</strong> – Only SSL-connections with client authentication are permitted</td>
</tr>
<tr>
<td>CREATOR</td>
<td>Name of the user created the object</td>
</tr>
</tbody>
</table>

*Continued on next page*
Continued from previous page

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE_TIME</td>
<td>Date and time of object creation</td>
</tr>
<tr>
<td>CHANGER</td>
<td>Name of the last user changed the object</td>
</tr>
<tr>
<td>CHANGE_TIME</td>
<td>Date and time of object change</td>
</tr>
<tr>
<td>PRIVS</td>
<td>String containing the users privileges on the object</td>
</tr>
<tr>
<td>MANAGE_PRIVS</td>
<td>Table of the manage privileges</td>
</tr>
<tr>
<td></td>
<td>See also table 22.70 on page 406</td>
</tr>
<tr>
<td>GROUPS</td>
<td>Table of groups to which the user belongs</td>
</tr>
<tr>
<td></td>
<td>See also table 22.71 on page 406</td>
</tr>
<tr>
<td>EQUIVALENT_USERS</td>
<td>Table of users and jobervers that count as equivalent</td>
</tr>
<tr>
<td></td>
<td>See also table 22.72 on page 407</td>
</tr>
<tr>
<td>PARAMETERS</td>
<td>It is possible to save key value pairs for a user. Although these values are</td>
</tr>
<tr>
<td></td>
<td>not used by the server itself, they allow user-related settings to be</td>
</tr>
<tr>
<td></td>
<td>centrally stored by a frontend application.</td>
</tr>
<tr>
<td>COMMENTTYPE</td>
<td>Type of comment if a comment is defined</td>
</tr>
<tr>
<td>COMMENT</td>
<td>Comment if defined</td>
</tr>
<tr>
<td></td>
<td>See also table 22.73 on page 407</td>
</tr>
</tbody>
</table>

Table 22.69.: Description of the output structure of the show user statement

**MANAGE_PRIVS**  The layout of the MANAGE_PRIVS table is shown in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRIVS</td>
<td>String containing the users privileges on the object</td>
</tr>
</tbody>
</table>

Table 22.70.: Description of the output structure of the show user subtable

**GROUPS**  The layout of the GROUPS table is shown in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>The repository object Id</td>
</tr>
</tbody>
</table>

Continued on next page

406
Continued from previous page

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME</td>
<td>The object name</td>
</tr>
<tr>
<td>PRIVS</td>
<td>String containing the users privileges on the ob-</td>
</tr>
<tr>
<td></td>
<td>ject</td>
</tr>
</tbody>
</table>

Table 22.71.: Description of the output structure of the show user subtable

**EQUIVALENT_USERS**  The layout of the EQUIVALENT_USERS table is shown in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TYPE</td>
<td>The type of user (server or user)</td>
</tr>
<tr>
<td>EQUIVALENT_USER</td>
<td>Name of the equivalent user</td>
</tr>
</tbody>
</table>

Table 22.72.: Description of the output structure of the show user subtable

**COMMENT**  The layout of the COMMENT table is shown in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TAG</td>
<td>The tag serves as a kind of heading for the com-</td>
</tr>
<tr>
<td></td>
<td>ment and is optional.</td>
</tr>
<tr>
<td>COMMENT</td>
<td>Comment if defined</td>
</tr>
</tbody>
</table>

Table 22.73.: Description of the output structure of the show user subtable
23. shutdown commands
**shutdown**

**Purpose**

*Purpose*  The purpose of the *shutdown* statement is to instruct the addressed jobserver to terminate.

**Syntax**

*Syntax*  The syntax for the *shutdown* statement is

```plaintext
shutdown serverpath
```

**Description**

*Description*  The *shutdown* statement is used to shut down the addressed jobserver.

**Output**

*Output*  This statement returns a confirmation of a successful operation.
24. stop commands
**stop server**

**Purpose**

*Purpose:* The purpose of the *stop server* statement is to instruct the server to terminate.

**Syntax**

*Syntax:* The syntax for the *stop server* statement is

```
stop server

stop server kill
```

**Description**

*Description:* The *stop server* statement is used to shut down the server. If this should not function correctly for any reason, the server can also be forced to shut down using kill.

**Output**

*Output:* This statement returns a confirmation of a successful operation.
25. submit commands
**submit**

**Purpose**

The purpose of the `submit` statement is to execute a master batch or job as well as all defined children.

**Syntax**

The syntax for the `submit` statement is

```plaintext
submit < folderpath | id > [ with WITHITEM {, WITHITEM} ]

submit aliasname [ with WITHITEM {, WITHITEM} ]
```

**WITHITEM:**

- `check only`
- `childtag = string`
- `< enable | disable >`
- `master`
- `nicevalue = signed_integer`
- `parameter = none`
- `parameter = ( PARAM {, PARAM} )`
- `< noresume | resume in period | resume at datetime >`
- `submittag = string`
- `< nosuspend | suspend >`
- `time zone = string`
- `unresolved = JRQ_UNRESOLVED`
- `group = groupname`

**PARAM:**

`parametername = < string | number >`

**JRQ_UNRESOLVED:**

- `defer`
- `defer ignore`
- `error`
- `ignore`
- `suspend`
**Description**

The *submit* statement is used to submit a job or batch. There are two kinds of submit command:

- The first kind is used by users, who can also be programs, and the Time Scheduling Module. This form submits Master Jobs and Batches.

- The second form of the statement is used by jobs to submit dynamic children.

**check only**  The check only option is used to verify whether a Master Submitable Batch or Job can be submitted. This means that a check is run to ascertain whether all the dependencies can be fulfilled and all the referenced parameters are defined.

Whether the jobs can be executed in any scope or not is not verified. This is a situation that can arise at any point during the runtime.

Positive feedback means that, from the system’s perspective, the job or batch can be submitted.

**childtag**  The childtag option is used by jobs to submit several instances of the same Scheduling Entity and to be able to differentiate between them.

An error is triggered if the same Scheduling Entity is submitted twice using the same childtag. The content of the childtag has no further significance for the Scheduling System.

The maximum length for a childtag is 70 characters. The childtag option is ignored in the case of a Master Submit.

**group**  The group option is used to set the owner group to the specified value.

The user must belong to this group unless he belongs to the ADMIN privileged group. In this case, any group can be specified.

**nicevalue**  The nicevalue option defines a correction that is used for the calculation of the priorities for the job and its children. Values between -100 and 100 are permitted.

**parameter**  The parameter option is used to specify the value of Job Parameters for the submit. The parameters are set in the scope of the Master Batch or Job. This means that if parameters are specified that are not defined in the Master Batch or Job, these parameters are invisible to any children.
**submittag**  If the submittag is specified, it must have a unique name for the Submitted Entity. This tag was introduced to be able to programmatically submit jobs and batches and to resubmit the job or batch with the same tag following a crash of one of the components. If the job submit was successful the first time, the second submit will report an error. If not, the second submit will succeed.

**unresolved**  The unresolved option defines how the server is to react to unresolved dependencies. This option is mainly used if parts of a batch are submitted following repair work. The faulty part is normally cancelled and then resubmitted as a Master Run. In this case the previous dependencies have to be ignored otherwise the submit will fail.

**suspend**  The suspend option is used to submit jobs or batches and to suspend them at the same time. If nothing is defined, they are not suspended. This can be explicitly specified at the submit time.
If a job or batch was suspended, neither it nor its children are started. If a job is already running, it will not reach a Final State if it is suspended.

**resume**  The resume option can be used together with the suspend option to cause a delayed execution. There are two ways to do this. A delay can be achieved by specifying either the number of time units for the delay the time when the job or batch is to be activated.
This option can be used to reproduce the at functionality without creating a schedule.

**Output**

*Output*  This statement returns an output structure of type record.

**Output Description**  The data items of the output are described in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>Id of the Submitted Entity</td>
</tr>
</tbody>
</table>

Table 25.1.: Description of the output structure of the submit statement

416
26. suspend commands
**suspend**

**Purpose**

*Purpose*  The purpose of the *suspend* statement is to prevent further jobs to be executed by this jobserver. See also the *resume* statement on page 304.

**Syntax**

*Syntax*  The syntax for the *suspend* statement is

```
suspend serverpath
```

**Description**

*Description*  The *suspend* statement prevents further jobs from being executed by this jobserver.

**Output**

*Output*  This statement returns a confirmation of a successful operation.
Part III.

Jobserver Commands
27. Jobserver Commands
alter job

Purpose

The purpose of the alter job statement is to change properties of the specified job. This statement is used by job administrators, job servers, and by the job itself.

Syntax

The syntax for the alter job statement is

```plaintext
alter job jobid
with WITHITEM {, WITHITEM}  

alter job
with WITHITEM {, WITHITEM}
```

WITHITEM:

```plaintext
< disable | enable >
| < suspend | suspend restrict | suspend local | suspend local restrict >
| cancel
| clear warning
| clone [ < resume | suspend > ]
| comment = string
| error text = string
| exec pid = pid
| exit code = signed_integer
| exit state = statename [ force ]
| ext pid = pid
| ignore resource = ( id {, id} )
| ignore dependency = ( jobid [ recursive ] {, jobid [ recursive ]} )
| kill [ recursive ]
| nicevalue = signed_integer
| priority = integer
| renice = signed_integer
| rerun [ recursive ]
| resume
| < noresume | resume in period | resume at datetime >
| run = integer
| state = JOBSTATE
| timestamp = string
| warning = string
```
Description

The alter job command is used for several purposes. Firstly, job servers use this command to document the progress of a job. All the state transitions a job undergoes during the time when the job is the responsibility of a job server are performed using the alter job command.

Secondly, some changes such as ignoring dependencies or resources, as well as changing the priority of a job, are carried out manually by an administrator.

The Exit State of a job in a Pending State can be set by the job itself or by a process that knows the job ID and key of the job that is to be changed.

**cancel**  The cancel option is used to cancel the addressed job and all non-Final Children. A job can only be cancelled if neither the job itself nor one of its children is active. Cancelling a running job will set the job in a cancelling state. The effective cancel is postponed until the job is finished.

If a Scheduling Entity is dependent upon the cancelled job, it can become unreachable. In this case the dependent job does not acquire the Unreachable Exit State defined in the Exit State Profiles, but is set as having the Job State “Unreachable”.

It is the operator’s task to restore this job back to the job state “Dependency Wait” by ignoring dependencies or even to cancel it.

Cancelled jobs are considered to be just like Final Jobs without a Final Exit. This means that the parents of a cancelled job become final without taking into consideration the Exit State of the cancelled job. In this case the dependent jobs of the parents continue running normally.

The cancel option can only be used by users.

**comment**  The comment option is used to document an action or to add a comment to the job. Comments can have a maximum length of 1024 characters. Any number of comments can be saved for a job.

Some comments are saved automatically. For example, if a job attains a Restartable State, a log is written to document this fact.
**error text**  The error text option is used to write error information about a job. This can be done by the responsible jobserver or a user. The server can write this text itself as well. This option is normally used if the jobserver cannot start the corresponding process. Possible cases are where it is not possible to switch to the defined working directory, if the executable program cannot be found, or when opening the error log file triggers an error.

**exec pid**  The exec pid option is used exclusively by the jobserver to set the process ID of the control process within the server.

**exit code**  The exit code option is used by the jobserver to tell the repository server with which Exit Code the process has finished. The repository server now calculates the matching Exit State from the Exit State Mapping that was used.

**exit state**  The exit state option is used by jobs in a pending state to set their state to another value. This is usually a Restartable or Final State. Alternatively, this option can be used by administrators to set the state of a non-final job. If the Force Flag is not being used, the only states that can be set are those which are theoretically attainable by applying the Exit State Mapping to any Exit Code. The set state must exist in the Exit State Profile.

**ext pid**  The ext pid option is used exclusively by the jobserver to set the process ID of the started user process.

**ignore resource**  The ignore resource option is used to revoke individual Resource Requests. The ignored resource is then no longer requested. If the parameters of a resource are being referenced, that resource cannot be ignored. If invalid IDs have been specified, it is skipped. All other specified resources are ignored. Invalid IDs in this context are the IDs of resources that are not requested by the job. The ignoring of resources is logged.

**ignore dependency**  The ignore dependency option is used to ignore defined dependencies. If the recursive flag is used, not only do the job or batch ignore the dependencies, but its children do so as well.
The kill option is used to submit the defined Kill Job. If no Kill Job has been defined, it is not possible to forcibly terminate the job from within BICsuite. The job obviously has to be active, that means it must be running, killed or broken_active. The last two states are not regular cases. When a Kill Job has been submitted, the Job State is to_kill. After the Kill Job has terminated, the Job State of the killed job is set to killed unless it has been completed, in which case it is finished or final. This means that the job with the Job State killed is always still running and that at least one attempt has been made to terminate it.

The nicevalue option is used to change the priority or the nicevalue of a job or batch and all of its children. If a child has several parents, any changes you make can, but do not necessarily have to, affect the priority of the child in the nicevalue of one of the parents. Where there are several parents, the maximum nicevalue is searched for. This means that if Job C has three Parents P1, P2 and P3, whereby P1 sets a nice value of 0, P2 sets a nicevalue of 10 and P3 a nicevalue of -10, the effective nicevalue is -10. (The lower the nicevalue the better). If the nicevalue for P2 is changed to -5, nothing happens because the -10 of P3 is better than -5. If the nicevalue of P3 falls to 0, the new effective nicevalue for Job C is -5. The nicevalues can have values between -100 and 100. Values that exceed this range are tacitly adjusted.

The priority option is used to change the (static) priority of a job. Because batches and milestones are not executed, priorities are irrelevant to them. Changing the priority only affects the changed job. Valid values lie between 0 and 100. In this case, 100 corresponds to the lowest priority and 0 is the highest priority. When calculating the dynamic priority of a job, the scheduler begins with the static priority and adjusts it according to how long the job has already been waiting. If more than one job has the same dynamic priority, the job with the lowest job ID is scheduled first.

The renice option is similar to the nicevalue option with the difference that the renice option functions relatively while the nicevalue option functions absolutely. If some batches have a nicevalue of 10, a renice of -5 causes the nicevalue to rise to 5. (It rises because the lower the number, the higher the priority).

The rerun option is used to restart a job in a Restartable State. If you attempt to restart a job that is not restartable, an error message is displayed. A job is restartable if it is in a Restartable State or it has the Job State error or broken_finished. If the recursive flag has been specified, the job itself and all its direct and indirect children that are in a Restartable State are restarted. If the job itself is final, this is
not considered to be an error. It is therefore possible to recursively restart batches.

**resume** The resume option is used to reactivate a suspended job or batch. There are two ways to do this. The suspended job or batch can either be reactivated immediately or a delay can be set. A delay can be achieved by specifying either the number of time units for the delay the time when the job or batch is to be activated. For details about specifying a time, refer to the overview on page 20. The resume option can be used together with the suspend option. Here, the job is suspended and then resumed again after (or at) a specified time.

**run** The run option is used by the jobserver to ensure that the modified job matches the current version. Theoretically, the computer could crash after a job has been started by a jobserver. To complete the work, the job is manually restarted from another jobserver. After the first system has been booted, the jobserver can attempt to change the job state to **broken_finished** without knowing anything about what happened after the crash. Using the run option then prevents the wrong state from being set.

**state** The state option is mainly used by jobservers, but it can also be used by administrators. It is not recommended to do so unless you know exactly what you are doing. The usual procedure is that the jobserver sets the state of a job from **starting** to **started**, from **started** to **running**, and from **running** to **finished**. In the event of a crash or any other problems, it is possible for the jobserver to set the job state to **broken_active** or **broken_finished**. This means that the Exit Code of the process is not available and the Exit State has to be set manually.

**suspend** The suspend option is used to suspend a batch or job. It always functions recursively. If a parent is suspended, its children are all suspended as well. The resume option is used to reverse the situation. The effect of the **restrict** option is that cwa resume can be done by members of the group **ADMIN** only.

**timestamp** The timestamp option is used by the jobserver to set the timestamps of the state transition in keeping with the local time from the perspective of jobserver.

**Output**

*Output* This statement returns a confirmation of a successful operation.
**alter jobserver**

**Purpose**

The purpose of the *alter jobserver* statement is to alter properties of a jobserver.

**Syntax**

The syntax for the *alter jobserver* statement is

```
alter [existing] jobserver
with < fatal | nonfatal > error text = string

alter [existing] jobserver
with dynamic PARAMETERS

PARAMETERS:
  parameter = none
  | parameter = (PARAMETERSPEC {, PARAMETERSPEC})

PARAMETERSPEC:
  parametername = < string | number >
```

**Description**

The *alter jobserver* command is both a user command and a jobserver command. It is used as a user command to change the configuration or other properties of a scope or jobserver. Further details are described in the *create scope* command on page 164.

The syntax of the user command corresponds to the first form of the *alter scope* command. As a jobserver command, it is used to notify the server about any errors.

If the Fatal Flag is used, this means that the jobserver is shutting down. In the other case, the jobserver continues running.

The third form of the *alter jobserver* command is also used by the jobserver. The jobserver publishes the values of its dynamic parameter. The server uses published values to resolve parameters in the command line and log file information when retrieving a job.

**Output**

This statement returns a confirmation of a successful operation.
connect

Purpose

Purpose  The purpose of the `connect` statement is to authenticate a jobserver to the server.

Syntax

Syntax  The syntax for the `connect` statement is

```
connect jobserver serverpath , servername identified by string | with
WITHITEM {, WITHITEM} |
```

WITHITEM:

```
command = ( sdms-command {; sdms-command} )
| method = string
| protocol = PROTOCOL
| session = string
| timeout = integer
| token = string
| < trace | notrace >
| trace level = integer
```

PROTOCOL:

```
json [ ZERO TERMINATED ]
| line
| perl [ ZERO TERMINATED ]
| python [ ZERO TERMINATED ]
| serial
| xml
```

Description

Description  The `connect` command is used to authenticate the connected process on the server to. A communication protocol can be optionally specified. The default protocol is `line`.

The selected protocol defines the output format. All protocols except for `serial` return ASCII output. The protocol `serial` returns a serialized Java object.

An executable command can also be returned when the connection is established. In this case, the output of the accompanying command is used as the output for the `connect` command. If the command fails, but the `connect` was successful, the connection remains active.

An example for all protocols except the `serial` protocol is given below.
**line protocol**  The line protocol only returns an ASCII text as the result from a command.

```sql
connect donald identified by 'duck' with protocol = line;
```

Connect

```
CONNECT_TIME : 19 Jan 2005 11:12:43 GMT
Connected
```

**XML protocol**  The XML protocol returns an XML structure as the result from a command.

```xml
<OUTPUT>
<Data>
<TITLE>Connect</TITLE>
<RECORD>
<CONNECT_TIME>19 Jan 2005 11:15:16 GMT</CONNECT_TIME></RECORD>
</DATA>
<FEEDBACK>Connected</FEEDBACK>
</OUTPUT>
```

**python protocol**  The python protocol returns a Python structure that can be evaluated using the **Python eval** function.

```python
connect donald identified by 'duck' with protocol = python;
{
'DATA': {
'TITLE': 'Connect',
'DESC': [
'CONNECT_TIME'
],
'RECORD': {
'CONNECT_TIME': '19 Jan 2005 11:16:08 GMT'
}
,'FEEDBACK': 'Connected'
}
```

**perl protocol**  The perl protocol returns a Perl structure that can be evaluated using the **Perl eval** function.
connect donald identified by 'duck' with protocol = perl;
{
'DATA' =>
{
'TITLE' => 'Connect',
'DESC' => [
'CONNECT_TIME'
],
'RECORD' => {
'CONNECT_TIME' => '19 Jan 2005 11:19:19 GMT'}
},
'FEEDBACK' => 'Connected'
}

Output

This statement returns a confirmation of a successful operation.
deregister

Purpose

The purpose of the `deregister` statement is to notify the server that the jobserver is not to process jobs anymore. See also the `register` statement on page 282.

Syntax

The syntax for the `deregister` statement is

```
  deregister serverpath . servername
```

Description

The `deregister` statement is used to notify the server about a more or less permanent failure of a jobserver. This message prompts different server actions. Firstly, all the running jobs on the jobserver (i.e. jobs in the state `started`, `running`, `to_kill` and `killed`) are set to the state `broken_finished`. Jobs in the state `starting` are reset to `runnable`. The jobserver is then removed from the list of jobservers that are able to process jobs so that this jobserver is consequently no longer allocated any more jobs. A side effect of this is that jobs that can only run on this server due to their resource requirements are set to the state `error` with the message “Cannot run in any scope because of resource shortage”. Finally, a complete reschedule is executed so that jobs are redistributed among the jobservers. The jobserver is added to the list of job-processing jobservers again by re-registering it (refer to the `register` statement on page 282).

Output

This statement returns a confirmation of a successful operation.
**disconnect**

**Purpose**

*Purpose*  The purpose of the *disconnect* statement is to terminate the server connection.

**Syntax**

*Syntax*  The syntax for the *disconnect* statement is

```
disconnect
```

**Description**

*Description*  The connection to the server can be shut down using the *disconnect* statement.

**Output**

*Output*  This statement returns a confirmation of a successful operation.
get next job

Purpose

The purpose of the get next job command is to fetch the next assignment from the server.

Syntax

The syntax for the get next job statement is

   get next job

Description

The jobserver uses the get next job statement to fetch the next command to be executed from the server.

Output

This statement returns an output structure of type table.

Output Description  The data items of the output are described in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMMAND</td>
<td>The command to be executed by the jobserver (NOP, ALTER, SHUTDOWN, STARTJOB)</td>
</tr>
<tr>
<td>CONFIG</td>
<td>Changed configuration. This value is only present in the case of an ALTER command.</td>
</tr>
<tr>
<td>ID</td>
<td>The Id of the job to be started; only present for the STARTJOB command.</td>
</tr>
<tr>
<td>DIR</td>
<td>The working directory of the job to be started; only present for the STARTJOB command.</td>
</tr>
<tr>
<td>LOG</td>
<td>The log file of the job to be started; only present for the STARTJOB command.</td>
</tr>
<tr>
<td>LOGAPP</td>
<td>Indicator showing whether the log file is to be opened with Append; only present for the STARTJOB command.</td>
</tr>
<tr>
<td>ERR</td>
<td>The error log file of the job to be started; only present for the STARTJOB command.</td>
</tr>
</tbody>
</table>

Continued on next page
Continued from previous page

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERRAPP</td>
<td>Indicator showing whether the error log file is to be opened with Append; only present for the STARTJOB command.</td>
</tr>
<tr>
<td>CMD</td>
<td>File name of the executable to be started; only present for the STARTJOB command.</td>
</tr>
<tr>
<td>ARGS</td>
<td>The command line parameter of the executable to be started; only present for the STARTJOB command.</td>
</tr>
<tr>
<td>ENV</td>
<td>Additional entries for the environment of the executable to be started; only present for the STARTJOB command.</td>
</tr>
<tr>
<td>RUN</td>
<td>Number of the run. Refer also to the alter job statement on page 69; only present for the STARTJOB command.</td>
</tr>
<tr>
<td>JOBENV</td>
<td>Vector of key value pairs defining the job defined environment variables to set before job execution</td>
</tr>
</tbody>
</table>

Table 27.1.: Description of the output structure of the get next job statement
**multicommand**

**Purpose**

This statement is used to control the behaviour of the SDMS Server.

**Syntax**

The syntax for the *multicommand* statement is

```
begin multicommand commandlist end multicommand
```

```
begin multicommand commandlist end multicommand rollback
```

**Description**

The *multicommands* allow multiple SDMS commands to be executed together, i.e. in one transaction. This ensures that either all the statements are executed without any errors or nothing happens at all. Not only that, but the transaction is not interrupted by other write transactions.

If the *rollback* keyword is specified, the transaction is undone at the end of the processing. This means that you can test whether the statements can be correctly processed (technically speaking).

**Output**

This statement returns a confirmation of a successful operation.
**reassure**

**Purpose**

*Purpose*  The purpose of the *reassure* job statement is to get a confirmation from the server about the necessity of starting a job after a jobserver was started.

**Syntax**

*Syntax*  The syntax for the *reassure* statement is

```
reassure jobid [with run = integer]
```

**Description**

*Description*  With the *reassure* statement a jobserver gets a confirmation from the server as to whether a job should be started. This statement is used when a jobserver boots up and there is a job in the *starting* state.

**Output**

*Output*  This statement returns a confirmation of a successful operation.
**register**

**Purpose**

The purpose of the `register` statement is to notify the server that the jobserver is ready to process jobs.

**Syntax**

The syntax for the `register` statement is

```
register serverpath . servername
with pid = pid [ suspend ]
```

```
register with pid = pid
```

**Description**

The first form is used by the operator to enable jobs to be executed by the specified jobserver. The second form is used by the jobserver itself to notify the server that it is ready to execute jobs. Jobs are scheduled for this jobserver (unless it is suspended) regardless of whether the server is connected or not. Refer to the 'deregister' statement on page 180.

**pid** The pid option provides the server with information about the jobserver’s process Id at operating level.

**suspend** The suspend option causes the jobserver to be transferred to a suspended state.

**Output**

This statement returns a confirmation of a successful operation.
Part IV.

Job Commands
28. Job Commands
**alter job**

**Purpose**

*Purpose* The purpose of the *alter job* statement is to change properties of the specified job. This statement is used by job administrators, job servers, and by the job itself.

**Syntax**

*Syntax* The syntax for the *alter job* statement is

```
alter job jobid
with WITHITEM {, WITHITEM}

alter job
with WITHITEM {, WITHITEM}
```

**WITHITEM:**

```
< disable | enable >
| < suspend | suspend restrict | suspend local | suspend local restrict >
cancel
clear warning
clone [ < resume | suspend > ]
comment = string
error text = string
exec pid = pid
exit code = signed_integer
exit state = statename [ force ]
ext pid = pid
ignore resource = ( id [, id] )
ignore dependency = ( jobid [ recursive ] [, jobid [ recursive ]] )
kill [ recursive ]
nicevalue = signed_integer
priority = integer
renice = signed_integer
rerun [ recursive ]
resume
< noresume | resume in period | resume at datetime >
run = integer
state = JOBSTATE
timestamp = string
warning = string
```
JOBSTATE:

broken active
broken finished
dependency wait
error
finished
resource wait
running
started
starting
synchronize wait

Description

The alter job command is used for several purposes. Firstly, jobservers use this command to document the progress of a job. All the state transitions a job undergoes during the time when the job is the responsibility of a jobserver are performed using the alter job command.

Secondly, some changes such as ignoring dependencies or resources, as well as changing the priority of a job, are carried out manually by an administrator.

The Exit State of a job in a Pending State can be set by the job itself or by a process that knows the job ID and key of the job that is to be changed.

cancel  The cancel option is used to cancel the addressed job and all non-Final Children. A job can only be cancelled if neither the job itself nor one of its children is active. Cancelling a running job will set the job in a cancelling state. The effective cancel is postponed until the job is finished.

If a Scheduling Entity is dependent upon the cancelled job, it can become unreachable. In this case the dependent job does not acquire the Unreachable Exit State defined in the Exit State Profiles, but is set as having the Job State “Unreachable”.

It is the operator’s task to restore this job back to the job state “Dependency Wait” by ignoring dependencies or even to cancel it.

Cancelled jobs are considered to be just like Final Jobs without a Final Exit. This means that the parents of a cancelled job become final without taking into consideration the Exit State of the cancelled job. In this case the dependent jobs of the parents continue running normally.

The cancel option can only be used by users.

cancel  The comment option is used to document an action or to add a comment to the job. Comments can have a maximum length of 1024 characters. Any number of comments can be saved for a job.

Some comments are saved automatically. For example, if a job attains a Restartable State, a log is written to document this fact.
**error text**  The error text option is used to write error information about a job. This can be done by the responsible jobserver or a user. The server can write this text itself as well. This option is normally used if the jobserver cannot start the corresponding process. Possible cases are where it is not possible to switch to the defined working directory, if the executable program cannot be found, or when opening the error log file triggers an error.

**exec pid**  The exec pid option is used exclusively by the jobserver to set the process ID of the control process within the server.

**exit code**  The exit code option is used by the jobserver to tell the repository server with which Exit Code the process has finished. The repository server now calculates the matching Exit State from the Exit State Mapping that was used.

**exit state**  The exit state option is used by jobs in a pending state to set their state to another value. This is usually a Restartable or Final State. Alternatively, this option can be used by administrators to set the state of a non-final job. If the Force Flag is not being used, the only states that can be set are those which are theoretically attainable by applying the Exit State Mapping to any Exit Code. The set state must exist in the Exit State Profile.

**ext pid**  The ext pid option is used exclusively by the jobserver to set the process ID of the started user process.

**ignore resource**  The ignore resource option is used to revoke individual Resource Requests. The ignored resource is then no longer requested. If the parameters of a resource are being referenced, that resource cannot be ignored. If invalid IDs have been specified, it is skipped. All other specified resources are ignored. Invalid IDs in this context are the IDs of resources that are not requested by the job. The ignoring of resources is logged.

**ignore dependency**  The ignore dependency option is used to ignore defined dependencies. If the recursive flag is used, not only do the job or batch ignore the dependencies, but its children do so as well.
**kill**  The kill option is used to submit the defined Kill Job. If no Kill Job has been defined, it is not possible to forcibly terminate the job from within BICsuite. The job obviously has to be active, that means it must be **running**, **killed** or **broken_active**. The last two states are not regular cases. When a Kill Job has been submitted, the Job State is **to_kill**. After the Kill Job has terminated, the Job State of the killed job is set to **killed** unless it has been completed, in which case it is **finished** or **final**. This means that the job with the Job State **killed** is always still running and that at least one attempt has been made to terminate it.

**nicevalue**  The nicevalue option is used to change the priority or the nicevalue of a job or batch and all of its children. If a child has several parents, any changes you make can, but do not necessarily have to, affect the priority of the child in the nicevalue of one of the parents. Where there are several parents, the maximum nicevalue is searched for. This means that if Job C has three Parents P1, P2 and P3, whereby P1 sets a nice value of 0, P2 sets a nicevalue of 10 and P3 a nicevalue of -10, the effective nicevalue is -10. (The lower the nicevalue the better). If the nicevalue for P2 is changed to -5, nothing happens because the -10 of P3 is better than -5. If the nicevalue of P3 falls to 0, the new effective nicevalue for Job C is -5. The nicevalues can have values between -100 and 100. Values that exceed this range are tacitly adjusted.

**priority**  The priority option is used to change the (static) priority of a job. Because batches and milestones are not executed, priorities are irrelevant to them. Changing the priority only affects the changed job. Valid values lie between 0 and 100. In this case, 100 corresponds to the lowest priority and 0 is the highest priority. When calculating the dynamic priority of a job, the scheduler begins with the static priority and adjusts it according to how long the job has already been waiting. If more than one job has the same dynamic priority, the job with the lowest job ID is scheduled first.

**renice**  The renice option is similar to the nicevalue option with the difference that the renice option functions relatively while the nicevalue option functions absolutely. If some batches have a nicevalue of 10, a renice of -5 causes the nicevalue to rise to 5. (It rises because the lower the number, the higher the priority).

**rerun**  The rerun option is used to restart a job in a Restartable State. If you attempt to restart a job that is not restartable, an error message is displayed. A job is restartable if it is in a Restartable State or it has the Job State **error** or **broken_finished**. If the recursive flag has been specified, the job itself and all its direct and indirect children that are in a Restartable State are restarted. If the job itself is final, this is
not considered to be an error. It is therefore possible to recursively restart batches.

**resume**  The resume option is used to reactivate a suspended job or batch. There are two ways to do this. The suspended job or batch can either be reactivated immediately or a delay can be set. A delay can be achieved by specifying either the number of time units for the delay or the time when the job or batch is to be activated. For details about specifying a time, refer to the overview on page 20. The resume option can be used together with the suspend option. Here, the job is suspended and then resumed again after (or at) a specified time.

**run**  The run option is used by the jobserver to ensure that the modified job matches the current version. Theoretically, the computer could crash after a job has been started by a jobserver. To complete the work, the job is manually restarted from another jobserver. After the first system has been booted, the jobserver can attempt to change the job state to broken\_finished without knowing anything about what happened after the crash. Using the run option then prevents the wrong state from being set.

**state**  The state option is mainly used by jobservers, but it can also be used by administrators. It is not recommended to do so unless you know exactly what you are doing. The usual procedure is that the jobserver sets the state of a job from starting to started, from started to running, and from running to finished. In the event of a crash or any other problems, it is possible for the jobserver to set the job state to broken\_active or broken\_finished. This means that the Exit Code of the process is not available and the Exit State has to be set manually.

**suspend**  The suspend option is used to suspend a batch or job. It always functions recursively. If a parent is suspended, its children are all suspended as well. The resume option is used to reverse the situation. The effect of the restrict option is that cwa resume can be done by members of the group ADMIN only.

**timestamp**  The timestamp option is used by the jobserver to set the timestamps of the state transition in keeping with the local time from the perspective of jobserver.

**Output**

*Output*  This statement returns a confirmation of a successful operation.
connect

Purpose

The purpose of the connect statement is to authenticate a job to the server.

Syntax

The syntax for the connect statement is

\[
\text{connect job } \text{jobid identified by } \text{string [ with WITHITEM {, WITHITEM} ]}
\]

WITHITEM:

\[
\begin{align*}
\text{command} &= ( \text{sdms-command [; sdms-command]} ) \\
\text{method} &= \text{string} \\
\text{protocol} &= \text{PROTOCOL} \\
\text{session} &= \text{string} \\
\text{timeout} &= \text{integer} \\
\text{token} &= \text{string} \\
<\text{trace}|\text{notrace}> \\
\text{trace level} &= \text{integer}
\end{align*}
\]

PROTOCOL:

\[
\begin{align*}
\text{json} &\quad \text{ZERO TERMINATED} \\
\text{line} \\
\text{perl} &\quad \text{ZERO TERMINATED} \\
\text{python} &\quad \text{ZERO TERMINATED} \\
\text{serial} \\
\text{xml}
\end{align*}
\]

Description

The connect command is used to authenticate the connected process on the server to. A communication protocol can be optionally specified. The default protocol is line. The selected protocol defines the output format. All protocols except for serial return ASCII output. The protocol serial returns a serialized Java object. An executable command can also be returned when the connection is established. In this case, the output of the accompanying command is used as the output for the connect command. If the command fails, but the connect was successful, the connection remains active. An example for all protocols except the serial protocol is given below.
**line protocol**  The line protocol only returns an ASCII text as the result from a command.

```plaintext
connect donald identified by 'duck' with protocol = line;
Connect
CONNECT_TIME : 19 Jan 2005 11:12:43 GMT
Connected
SDMS>
```

**XML protocol**  The XML protocol returns an XML structure as the result from a command.

```xml
connect donald identified by 'duck' with protocol = xml;
<OUTPUT>
<Data>
<TITLE>Connect</TITLE>
<Record>
<CONNECT_TIME>19 Jan 2005 11:15:16 GMT</CONNECT_TIME>
</Record>
</Data>
<FEEDBACK>Connected</FEEDBACK>
</OUTPUT>
```

**python protocol**  The python protocol returns a Python structure that can be valuated using the *Python eval* function.

```plaintext
connect donald identified by 'duck' with protocol = python;
{
'DATA' :
{
'TITLE' : 'Connect',
'DESC' : [
'CONNECT_TIME'
],
'RECORD' : {
'CONNECT_TIME' : '19 Jan 2005 11:16:08 GMT'}
},'FEEDBACK' : 'Connected'
}
```

**perl protocol**  The perl protocol returns a Perl structure that can be valuated using the *Perl eval* function.
connect donald identified by 'duck' with protocol = perl;
{
'DATA' =>
{
'TITLE' => 'Connect',
'DESC' => [
'CONNECT_TIME'
],
'RECORD' => {
'CONNECT_TIME' => '19 Jan 2005 11:19:19 GMT'}
},
'FEEDBACK' => 'Connected'
}

Output

This statement returns a confirmation of a successful operation.

Output
disconnect

Purpose

Purpose The purpose of the disconnect statement is to terminate the server connection.

Syntax

Syntax The syntax for the disconnect statement is

disconnect

Description

Description The connection to the server can be shut down using the disconnect statement.

Output

Output This statement returns a confirmation of a successful operation.
get parameter

Purpose

The purpose of the `get parameter` statement is to get the value of the specified parameter within the context of the requesting job, respectively the specified job.

Syntax

The syntax for the `get parameter` statement is

```
get parameter parametername [ < strict | warn | liberal > ]
```

```
get parameter of jobid parametername [ < strict | warn | liberal > ]
```

Description

The `get parameter` statement is used to get the value of the specified parameter within the context of a job. The additional option has the following meaning:

<table>
<thead>
<tr>
<th>Option</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>strict</td>
<td>The server returns an error if the requested parameter is not explicitly declared in the job definition.</td>
</tr>
<tr>
<td>warn</td>
<td>A message is written to the server’s log file when an attempt is made to determine the value of an undeclared parameter.</td>
</tr>
<tr>
<td>liberal</td>
<td>An attempt to query an undeclared parameter is tacitly allowed. The default behaviour depends on the configuration of the server.</td>
</tr>
</tbody>
</table>

Output

This statement returns an output structure of type record.

**Output Description** The data items of the output are described in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VALUE</td>
<td>Value of the requested parameter</td>
</tr>
</tbody>
</table>

Table 28.1.: Description of the output structure of the get parameter statement
**get submittag**

**Purpose**

*Purpose* The purpose of the `get submittag` statement is to get a (server local) unique identifier from the server. This identifier can be used to avoid *race conditions* between frontend and backend when submitting jobs.

**Syntax**

*Syntax* The syntax for the `get submittag` statement is

```
get submittag
```

**Description**

*Description* The `get submittag` statement is used to acquire an identification from the server. This prevents race conditions between the front end and back end when jobs are submitted.

Such a situation arises when feedback about the submit does not reach the front end due to an error. By using a submittag, the front end can safely start a second attempt. The server recognises whether the job in question has already been submitted and responds accordingly. This reliably prevents the job from being submitted twice.

**Output**

*Output* This statement returns an output structure of type record.

**Output Description** The data items of the output are described in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VALUE</td>
<td>The requested Submit Tag</td>
</tr>
</tbody>
</table>

Table 28.2.: Description of the output structure of the `get submittag` statement
multicommand

Purpose

This statement is used to control the behaviour of the SDMS Server.

Syntax

The syntax for the multicommand statement is

\begin{verbatim}
begin multicommand commandlist end multicommand
\end{verbatim}

\begin{verbatim}
begin multicommand commandlist end multicommand rollback
\end{verbatim}

Description

The multicommands allow multiple SDMS commands to be executed together, i.e. in one transaction. This ensures that either all the statements are executed without any errors or nothing happens at all. Not only that, but the transaction is not interrupted by other write transactions.

If the rollback keyword is specified, the transaction is undone at the end of the processing. This means that you can test whether the statements can be correctly processed (technically speaking).

Output

This statement returns a confirmation of a successful operation.
**set parameter**

*Purpose*

The purpose of the *set parameter* statement is to set the value of the specified parameters within the context of the requesting job, respectively the specified job.

*Syntax*

The syntax for the *set parameter* statement is

\[
\text{set parameter parametername} = \text{string} \{, \text{parametername} = \text{string} \}
\]

\[
\text{set parameter} \ < \text{on} | \text{of} > \text{jobid parametername} = \text{string} \{, \text{parametername} = \text{string} \} \ [\text{with comment} = \text{string} ]
\]

\[
\text{set parameter} \ < \text{on} | \text{of} > \text{jobid parametername} = \text{string} \{, \text{parametername} = \text{string} \} \text{identified by string} \ [\text{with comment} = \text{string} ]
\]

*Description*

The *set parameter* statements can be used to set jobs or user parameter values in the context of the job.

If the *identified by* option is specified, the parameter is only set if the pair *jobid* and *string* would allow a logon.

*Output*

This statement returns a confirmation of a successful operation.
set state

Purpose

The purpose of the set state statement is to set the exit state of a job in a pending exit state.

Syntax

The syntax for the set state statement is

\[ \text{set state} = \text{statename} \]

Description

The set state statement is used to set the Exit State of a job to a Pending Exit State.

Output

This statement returns a confirmation of a successful operation.
submit

Purpose

The purpose of the `submit` statement is to execute a master batch or job as well as all defined children.

Syntax

The syntax for the `submit` statement is

```plaintext
submit < folderpath | id > [ with WITHITEM {, WITHITEM} ]

submit aliasname [ with WITHITEM {, WITHITEM} ]
```

WITHITEM:

- `check only`
- `childtag = string`
- `< enable | disable >`
- `master`
- `nicevalue = signed_integer`
- `parameter = none`
- `parameter = ( PARAM {, PARAM} )`
- `< noresume | resume in period | resume at datetime >`
- `submittag = string`
- `< nosuspend | suspend >`
- `time zone = string`
- `unresolved = JRQ_UNRESOLVED`
- `group = groupname`

PARAM:

- `parametername = < string | number >`

JRQ_UNRESOLVED:

- `defer`
- `defer ignore`
- `error`
- `ignore`
- `suspend`
**Description**

The submit statement is used to submit a job or batch. There are two kinds of submit command:

- The first kind is used by users, who can also be programs, and the Time Scheduling Module. This form submits Master Jobs and Batches.

- The second form of the statement is used by jobs to submit dynamic children.

**check only**  The check only option is used to verify whether a Master Submitable Batch or Job can be submitted. This means that a check is run to ascertain whether all the dependencies can be fulfilled and all the referenced parameters are defined.

Whether the jobs can be executed in any scope or not is not verified. This is a situation that can arise at any point during the runtime.

Positive feedback means that, from the system’s perspective, the job or batch can be submitted.

**childdtag**  The childdtag option is used by jobs to submit several instances of the same Scheduling Entity and to be able to differentiate between them.

An error is triggered if the same Scheduling Entity is submitted twice using the same childdtag. The content of the childdtag has no further significance for the Scheduling System.

The maximum length for a childdtag is 70 characters. The childdtag option is ignored in the case of a Master Submit.

**group**  The group option is used to set the owner group to the specified value. The user must belong to this group unless he belongs to the ADMIN privileged group. In this case, any group can be specified.

**nicevalue**  The nicevalue option defines a correction that is used for the calculation of the priorities for the job and its children. Values between -100 and 100 are permitted.

**parameter**  The parameter option is used to specify the value of Job Parameters for the submit. The parameters are set in the scope of the Master Batch or Job. This means that if parameters are specified that are not defined in the Master Batch or Job, these parameters are invisible to any children.
**submittag** If the submittag is specified, it must have a unique name for the Submitted Entity. This tag was introduced to be able to programmatically submit jobs and batches and to resubmit the job or batch with the same tag following a crash of one of the components. If the job submit was successful the first time, the second submit will report an error. If not, the second submit will succeed.

**unresolved** The unresolved option defines how the server is to react to unresolved dependencies. This option is mainly used if parts of a batch are submitted following repair work. The faulty part is normally cancelled and then resubmitted as a Master Run. In this case the previous dependencies have to be ignored otherwise the submit will fail.

**suspend** The suspend option is used to submit jobs or batches and to suspend them at the same time. If nothing is defined, they are not suspended. This can be explicitly specified at the submit time. If a job or batch was suspended, neither it nor its children are started. If a job is already running, it will not reach a Final State if it is suspended.

**resume** The resume option can be used together with the suspend option to cause a delayed execution. There are two ways to do this. A delay can be achieved by specifying either the number of time units for the delay the time when the job or batch is to be activated. This option can be used to reproduce the `at` functionality without creating a schedule.

**Output**

*Output* This statement returns an output structure of type record.

**Output Description** The data items of the output are described in the table below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>Id of the Submitted Entity</td>
</tr>
</tbody>
</table>

Table 28.3.: Description of the output structure of the submit statement
Part V.

Programming Examples
29. Programming examples

This section contains some simple examples of how to communicate with the Scheduling Server in several different programming languages. The examples are intended to show the essential structures. The error handling is extremely rudimentary, and the processing of the server responses is also kept to a minimum.

As usual, some details are required to log on to the Scheduling Server: Host name or IP address of the system on which the Scheduling Server is running, the port to which it responds (usually 2506), a user name and a password. In our examples this data is defined as constants. It may be obvious that a serious implementation should use another method such as evaluating the .sdmshrc.

All the programs shown are available as source code under $BICSUITEHOMExamples.

Java

Since schedulix is itself written in Java, the BICsuite.jar can be used for Java developing utilities in Java.

In the example below, the SDMSServerConnection is used to set up the connection to the Scheduling Server. To do this, first of all an object is created using the standard information. The connection is then established using the connect() method. The finish() method is used to terminate the connection. As long as the connection is active, any number of statements can be executed with the help of the execute() method. In the example below, the list sessions; command is executed.

An object of the type SDMSOutput is returned as the result. If the Member Variable error is not null, an error occurred while the command was being processed. The Member Variables error.code and error.message give more details about the error.

In our example, the class SDMSLineRenderer is used to return the formatted result of the command at stdout.

The error handling is kept extremely simple. If an error occurs, the program is exited with a Return Code 1.

1 import de.independit.scheduler.shell.SDMSServerConnection;
2 import de.independit.scheduler.server.output.SDMSOutput;
3 import de.independit.scheduler.server.output.SDMSLineRenderer;
4 import java.io.IOException;
public class SimpleAccess {

    private static SDMSServerConnection sc = null;
    private static SDMSLineRenderer lr = null;

    public static void main(String argv[])
    {
        sc = new SDMSServerConnection(
            "localhost",  // host
            2506,        // port
            "SYSTEM",    // user
            "G0H0ME",    // password
            0,           // connection timeout disabled
            false         // no TLS
        );
        lr = new SDMSLineRenderer();

        try {
            SDMSOutput o = sc.connect(null);  // no special options
            if (o.error != null) {
                System.err.println("Connect Error: ", o.error.code + ", " + o.error.message);
                System.exit(1);
            }
            o = sc.execute("LIST SESSIONS;");
            try {
                lr.render(System.out, o);
            } catch (Exception e) {
                System.err.println("Something went wrong: ", e.toString());
            }
            sc.finish();
        } catch (IOException ioe) {
            System.err.println("Something went wrong : ", ioe.toString());
            System.exit(1);
        }

        System.exit(0);
    }
}

To convert the Java program, the the BICsuite.jar should be included in the
CLASSPATH. Under Unix or Linux, that could look like this (the output lines have
been shortened for reasons of clarity):

$ CLASSPATH=$CLASSPATH:$BICSUITEHOME/lib/BICsuite.jar

462
$ export CLASSPATH
$ javac SimpleAccess.java
$ java SimpleAccess

List of Sessions

<table>
<thead>
<tr>
<th>SESSIONID</th>
<th>PORT</th>
<th>START TIME</th>
<th>TYPE</th>
<th>USER</th>
</tr>
</thead>
<tbody>
<tr>
<td>1001</td>
<td>2506</td>
<td>Mon Oct 12 11:25:47</td>
<td>JOBSERVER</td>
<td>GLOBAL.EXAMPLES</td>
</tr>
<tr>
<td>1002</td>
<td>2506</td>
<td>Mon Oct 12 11:25:47</td>
<td>JOBSERVER</td>
<td>GLOBAL.EXAMPLES</td>
</tr>
<tr>
<td>1003</td>
<td>2506</td>
<td>Mon Oct 12 11:25:47</td>
<td>JOBSERVER</td>
<td>GLOBAL.EXAMPLES</td>
</tr>
<tr>
<td>1043</td>
<td>2506</td>
<td>Wed Oct 21 15:00:12</td>
<td>USER</td>
<td>SYSTEM</td>
</tr>
<tr>
<td>1234321</td>
<td>0</td>
<td>Mon Oct 12 11:25:22</td>
<td>USER</td>
<td>SchedulingThread</td>
</tr>
<tr>
<td>1234322</td>
<td>0</td>
<td>Mon Oct 12 11:25:22</td>
<td>USER</td>
<td>GarbageThread</td>
</tr>
<tr>
<td>1234323</td>
<td>0</td>
<td>Mon Oct 12 11:25:22</td>
<td>USER</td>
<td>TriggerThread</td>
</tr>
<tr>
<td>1234324</td>
<td>0</td>
<td>Mon Oct 12 11:25:22</td>
<td>USER</td>
<td>PoolThread</td>
</tr>
<tr>
<td>19630127</td>
<td>0</td>
<td>Mon Oct 12 11:25:22</td>
<td>USER</td>
<td>TimerThread</td>
</tr>
</tbody>
</table>

9 Session(s) found

A second example shows how attributes from the output structure can be queried. In this example, two commands are executed after the connection has been established, and data is then selectively extracted and outputted from the two results. The result of a `SHOW SYSTEM` command is always a record with a table. In line 41, the version information is extracted from the record data. In lines 44 to 47, the names of the worker threads from the table named `WORKER` are determined. The result of a `LIST SESSIONS` command is always a pure table. In lines 58 to 61, the names of the logged-on users, job servers and internal threads are determined and outputted.

```
import java.io.IOException;
import de.independit.scheduler.shell.SDMSServerConnection;
import de.independit.scheduler.server.output.SDMSOutput;
import de.independit.scheduler.server.output.SDMSOutputUtil;

public class testJavaApi {
    public static void main(String[] args) {
        SDMSServerConnection sc = new SDMSServerConnection("localhost", /* host */
                                                                2506, /* port */
                                                                "SYSTEM", /* user */
                                                                "G0H0ME", /* password */
                                                                0, /* connection timeout disabled */
                                                                false /* no TLS */);
        SDMSOutput output = null;
        try {
            output = sc.connect(null);
        } catch (IOException ioe) {
```
System.err.println("Error '" + ioe.toString() + ")
    System.exit(1);
}
if (output.error != null) {
    System.err.println("Error '" + output.error.code + ": " + 
        output.error.message + "' connecting to BICsuite server");
    System.exit(1);
}
String command = "SHOW SYSTEM";
output = sc.execute(command);
if (output.error != null) {
    System.err.println("Error '" + output.error.code + ": " + 
        output.error.message + "' executing command: " + 
        command);
    System.exit(1);
}
System.out.println("Version: " + SDMSOutputUtil.getFromRecord(
    output,"VERSION"));
    int workers = SDMSOutputUtil.getTableLength(output,"WORKER");
    System.out.println("Workers: " + workers);
    for (int i = 0; i < workers; i++) {
        System.out.println(" Name: " + 
            SDMSOutputUtil.getFromTable(output, "WORKER", i, "NAME");
    }
}
command = "LIST SESSIONS";
output = sc.execute(command);
if (output.error != null) {
    System.err.println("Error '" + output.error.code + ": " + 
        output.error.message + "' executing command: " + 
        command);
    System.exit(1);
}
    int sessions = SDMSOutputUtil.getTableLength(output);
    System.out.println("Sessions: " + sessions);
    for (int i = 0; i < sessions; i++) {
        System.out.println(" User: " + 
            SDMSOutputUtil.getFromTable(output, i, "USER"));
    }
}
try {
    sc.finish();
} catch (IOException ioe) {
    System.err.println("Error '" + ioe.toString() + ")
        "' closing BICsuite server connection");
    System.exit(1);
}
Converting and executing the program obviously functions in the same way as in
the first example. The CLASSPATH obviously does not have to be set again before
every conversion or execution.

```bash
$ CLASSPATH=$CLASSPATH:$BICSUITEHOME/lib/BICsuite.jar
$ export CLASSPATH
$ javac testJavaApi.java
$ java testJavaApi
```

Version: 2.10
Workers: 6
Name: Worker0
Name: Worker1
Name: Worker2
Name: Worker3
Name: Worker4
Name: Worker5
Sessions: 9
User: GLOBAL.EXAMPLES.HOST_1.SERVER
User: GLOBAL.EXAMPLES.LOCALHOST.SERVER
User: GLOBAL.EXAMPLES.HOST_2.SERVER
User: SYSTEM
User: SchedulingThread
User: GarbageThread
User: TriggerThread
User: PoolThread
User: TimerThread

**Python 2**

Access with Python 2 is also pretty simple. After all, the Zope application server
was written in Python and uses the file `sdms.py` as an extension to handle com-
unication with the Scheduling Server. This file can obviously also be used by any other Python script.
The `SDMSConnectionOpenV2()` method is used to set up the connection to the
Scheduling Server. This method requires a dictionary with a specified host and port
as the first parameter. Two other parameters specify the user and the password.
The last parameter is optional and is only used to give the session a meaningful
name. If the connection attempt fails, a dictionary is returned instead of a socket object.
This can be easily checked using the `has_key` method in a `try`-`except` block. In
the code fragment below, this is shown in lines 11 to 16.
As soon as the connection has been established, any commands can be executed us-
ing `SDMSCommandWithSoc`. The result is always an `SDMSOutput` data structure.
If an error has occurred, it contains an `ERROR` entry.
The `close()` method terminates the connection.

```python
1 import sdms
```
Try:
  if conn.has_key('ERROR'):
    print str(conn)
    exit(1)
  except:
    pass

stmt = "LIST SESSIONS;"
result = sdms.SDMSCommandWithSoc(conn, stmt)
if result.has_key('ERROR'):
  print str(result['ERROR'])
else:
  for row in result['DATA']['TABLE']:
    print "{0:3} {1:8} {2:32} {3:9} {4:15} {5:>15} {6}".format(
      row['THIS'],
      row['UID'],
      row['USER'],
      row['TYPE'],
      row['START'],
      row['IP'],
      row['INFORMATION'])

Connected

To execute the program, it is only necessary to set the PYTHONPATH accordingly. The output has been shortened for reasons of clarity.

$ PYTHONPATH=$PYTHONPATH:$BICSUITEHOME/..../schedulixweb/Extensions
$ export PYTHONPATH
$ python2 SimpleAccess.py
1047 GLOBAL.EXAMPLES.HOST_1.SERVER JOBSERVER Mon Oct 12 11:25:47 CEST 20...
1037 GLOBAL.EXAMPLES.LOCALHOST.SERVER JOBSERVER Mon Oct 12 11:25:47 CEST 20...
1057 GLOBAL.EXAMPLES.HOST_2.SERVER JOBSERVER Mon Oct 12 11:25:47 CEST 20...
  0 SYSTEM USER Wed Oct 21 14:20:40 CEST 20...
  2 SchedulingThread USER Mon Oct 12 11:25:22 CEST 20...
  2 GarbageThread USER Mon Oct 12 11:25:22 CEST 20...
  2 TriggerThread USER Mon Oct 12 11:25:22 CEST 20...
  2 PoolThread USER Mon Oct 12 11:25:22 CEST 20...
  2 TimerThread USER Mon Oct 12 11:25:22 CEST 20...

Python 3

In a Python 3 environment, everything runs analogue to the Python 2 environment while obviously taking into account the differences between the two lan-
guages. The Python 3 module is located in the Zope 4 tree under Extensions.

```python
import sdms

server = {'HOST': 'localhost',
          'PORT': '2506',
          'USER': 'SYSTEM',
          'PASSWORD': 'G0H0ME'}

conn = sdms.SDMSConnectionOpenV2(server,
                                 server['USER'],
                                 server['PASSWORD'],
                                 "Simple Access Example")

try:
    if 'ERROR' in conn:
        print(str(conn))
        exit(1)
    except:
        pass

stmt = "LIST SESSIONS;"
result = sdms.SDMSCommandWithSoc(conn, stmt)
if 'ERROR' in result:
    print(str(result['ERROR']))
else:
    for row in result['DATA']['TABLE']:
        print("{0:3} {1:8} {2:32} {3:9} {4:15} {5:>15} {6}".format(
            str(row['THIS']),
            str(row['UID']),
            str(row['USER']),
            str(row['type']),
            str(row['START']),
            str(row['IP']),
            str(row['INFORMATION'])))

conn.close()
```

The execution method is exactly the same as with Python 2:

```
$ PYTHONPATH=$PYTHONPATH:$BICSUITEHOME/..//schedulixweb4/Extensions
$ export PYTHONPATH
$ python3 SimpleAccess3.py
1047 GLOBAL.EXAMPLES.HOST_1.SERVER JOBSERVER Mon Oct 12 11:25:47 CEST 20...
1037 GLOBAL.EXAMPLES.LOCALHOST.SERVER JOBSERVER Mon Oct 12 11:25:47 CEST 20...
* 0 SYSTEM USER Wed Oct 21 15:33:31 CEST 20...
   2 SchedulingThread USER Mon Oct 12 11:25:22 CEST 20...
   2 GarbageThread USER Mon Oct 12 11:25:22 CEST 20...
   2 TriggerThread USER Mon Oct 12 11:25:22 CEST 20...
   2 PoolThread USER Mon Oct 12 11:25:22 CEST 20...
   2 TimerThread USER Mon Oct 12 11:25:22 CEST 20...
```

467
Our C API is used for access from a C program. This can be found at $BICSUITEHOME/src/capi. C is, of course, a relatively hardware-oriented programming language in which aspects such as memory management are largely left to the developer. That is why handling with the output structures is also more complex than in Java or Python. However, we have attempted to make the whole operation as simple as possible.

The prototypes of the available functions stand in the sdms_api.h header file. The relevant part of the file is shown below.

```c
extern int sdms_connection_open(SDMS_CONNECTION **connection, char *host, int port, char *user, char *password);
extern int sdms_command(SDMS_OUTPUT **output, SDMS_CONNECTION *connection, SDMS_STRING *command);
extern int sdms_connection_close(SDMS_CONNECTION **connection);
extern int sdms_string(SDMS_STRING **sdms_string, char *s);
extern int sdms_string_append(SDMS_STRING *string, char *text);
extern void sdms_string_clear(SDMS_STRING *string);
extern void sdms_string_free(SDMS_STRING **string);
extern int sdms_vector(SDMS_VECTOR **vector);
extern int sdms_vector_append(SDMS_VECTOR *vector, void *data);
extern void sdms_vector_free(SDMS_VECTOR **vector);
extern void sdms_output_free(SDMS_OUTPUT **output);
extern void sdms_error_print(char *text);
extern void sdms_error_clear(void);
extern int sdms_output_data_get_table_size(SDMS_OUTPUT_DATA *output_data, int *size);
extern int sdms_vector_find(SDMS_VECTOR *vector, char *name);
extern int sdms_output_data_get_by_name(SDMS_OUTPUT_DATA *output_data, SDMS_OUTPUT_DATA **value, char *name);
extern int sdms_output_data_get_string(SDMS_OUTPUT_DATA *output_data, char **value);
extern int sdms_output_data_get_row(SDMS_OUTPUT_DATA *output_data, SDMS_VECTOR **row, int index);
```

The functions `sdms_connection_open()` and `sdms_connection_close()` are self-explanatory. The function `sdms_command()` executes the command specified in `command`. The result is returned in the parameter `output`. Since a parameter of the type `SDMS_STRING` is required to execute commands, a number of functions are provided for handling this data type. A normal string in C
can be converted into a SDMS_STRING with the help of the function sdms_string(). The function sdms_string_append() is used to create an SDMS_STRING to expand the specified text. The function sdms_string_clear() deletes the contents of the string. Since dynamically allocated memory is required for working with strings, finally there is the sdms_string_free() function for freeing up the memory again in a controlled manner.

In many cases, data is returned as a list of values or even lists. In Java, this is done using a vector. Based on this, an SDMS_VECTOR is provided in the C interface. The functions for manipulating this data structure are roughly comparable to the SDMS_STRING functions. Normally, however, these functions are not used in applications because the vectors are not built by the application, but rather by the interface. Much more interesting, though, are the functions that extract elementary data from the vectors.

The data structure SDMS_OUTPUT is the comprehensive container in which the results of commands are returned. This container is made up of different data types which are usually stored in dynamically allocated memory blocks. To enable this memory to be freed up again, the function sdms_output_free() is called. This function also correctly takes into account the dynamic internal data structure.

True to the motto "a picture says more than a thousand words", in the program below a SHOW USER; as well as a LIST SESSIONS; are executed after the connection has been established, and the results are displayed on the screen.

```c
#include <stdio.h>
#include <stdlib.h>

#ifdef _WIN32
#include <winsock.h>
#endif

#include "sdms_api.h"

/* some constants / literals */
/* default values */
char * LOCALHOST = (char *) "localhost";
char * PORT = (char *) "2506";
char * SYSTEM = (char *) "SYSTEM";
char * PASSWD = (char *) "GOHOME";

/* column names */
char * NAME = (char *) "NAME";
char * GROUPS = (char *) "GROUPS";
char * SESSIONID = (char *) "SESSIONID";
char * USER = (char *) "USER";

/* used commands */
char * SHOW_USER = (char *) "SHOW USER;";
char * LIST_SESSION = (char *) "LIST SESSIONS;";
```
void do_exit (int exit_code);

/* sdms_connection_open() requires initialized pointer */
SDMS_CONNECTION *sdms_connection = NULL;

int main (int argc, char *argv[])
{
    char *host;
    char *port;
    char *user;
    char *pass;
    if (argc >= 2)
        host = argv[1];
    else
        host = LOCALHOST;
    if (argc >= 3)
        port = argv[2];
    else
        port = PORT;
    if (argc >= 4)
        user = argv[3];
    else
        user = SYSTEM;
    if (argc >= 5)
        pass = argv[4];
    else
        pass = PASSWD;

#ifndef _WIN32
    WSADATA wsaData;
    if (WSAStartup (MAKEWORD(1, 1), &wsaData) != 0) {
        fprintf (stderr, "WSAStartup(): Can't initialize Winsock.
");
        do_exit (1);
    }
#endif

    if (sdms_connection_open(&sdms_connection, host,
            atoi(port), user, pass) != SDMS_OK) {
        sdms_error_print((char *) "Error opening sdms connection");
        do_exit(1);
    }

    int size;
    int i;
    printf("--------------------------------------------------------
");
    printf("sdms_string() requires initialized pointer */
SDMS_STRING *command = NULL;

    /* sdms_command() requires initialized pointer */
SDMS_OUTPUT *sdms_output = NULL;
If (sdms_string (&command, SHOW_USER) != SDMS_OK) {
    sdms_error_print((char *) "Error allocating command SDMS_STRING")
    do_exit(1);
}

if (sdms_command (&sdms_output, sdms_connection, command) != SDMS_OK) {
    sdms_error_print((char *) "Error executing command");
    do_exit(1);
}

/* sdms_output_dump(sdms_output); */

SDMS_OUTPUT_DATA *name;
sdms_output_data_get_by_name(sdms_output->data, &name, NAME);
fprintf(stderr, "User %s is in the groups", (char *)(name->data));

SDMS_OUTPUT_DATA *groups;
sdms_output_data_get_by_name(sdms_output->data, &groups, GROUPS);
int groupname_idx = sdms_vector_find(groups->desc, NAME);
size = sdms_output_data_get_table_size(groups, &size);
char sep = ' ';
for (i = 0; i < size; i++) {
    SDMS_VECTOR *row;
    sdms_output_data_get_row(groups, &row, i);
    SDMS_OUTPUT_DATA *groupname = (SDMS_OUTPUT_DATA *) (row->buf[groupname_idx]);
    printf(stderr, "%c%s", sep, (char *)(groupname->data));
    sep = ', ';
}
fprintf(stderr, 
"");
sdms_output_free(&sdms_output);
printf("--------------------------------------------------------\n");

sdms_string_clear(command);
if (sdms_string_append(command, LIST_SESSION) != SDMS_OK) {
    sdms_error_print((char *) "Error building command");
    do_exit(1);
}

if (sdms_command (&sdms_output, sdms_connection, command) != SDMS_OK) {
    sdms_error_print((char *) "Error executing command");
    do_exit(1);
}

/* sdms_output_dump(sdms_output); */

SDMS_OUTPUT_DATA *data = sdms_output->data;
int sessionid_idx = sdms_vector_find(data->desc, SESSIONID);
int user_idx = sdms_vector_find(data->desc, USER);
Converting and executing the program are comparatively simple. A Make file is available for this, which should at least work on all Linux systems without any problems. The line breaks have been added for reasons of clarity.

```bash
$ cd $BICSUITEHOME/src/capi
$ make sdms_test
cc sdms_api.o sdms_test.o -o sdms_test
$ ./sdms_test localhost 2506 SYSTEM G0H0ME
User SYSTEM is in the groups ADMIN,PUBLIC
```

User SYSTEM is in the groups ADMIN,PUBLIC
As in the previous examples, this example follows a simple approach: Either it works or it terminates with exit code 1.

A modular design was also deliberately not used here. The fact that this is indispensable for large projects should be undisputed. In this simple example, however, it would be more of a distraction from what is to be shown.

The command line parameters are processed in lines 34 to 54. Missing parameters are replaced with the default parameters.

The WinSock library is initialised in lines 56 to 62 (this means that the example should also work under Windows). The symbol _WIN32 must be set to do this.

A connection with the Scheduling Server is then set up in lines 64 to 68. The program can now communicate with the server.

The first command should be a SHOW USER. Accordingly, the command is packed into an SDMS_STRING in line 81, and this data structure in line 86 (and line 87) is sent to the server.

This returns a data structure of the type SDMS_OUTPUT.

The received data is outputted at stderr in lines 94 to 111. First of all, the data item NAME is extracted from the output in line 95. The table containing groups is then fetched in line 99. From this table, the position of the group name is determined first (line 100) and the size of the table is queried (line 101).

This is followed by a simple loop to output the group names. The name is extracted in lines 106 and 107 using the previously determined index.

This completes the processing of this output structure, and the allocated memory is freed up again in line 113.

Since another command is to be executed, the memory for the old command is also freed up in line 117.

Now everything starts from the beginning all over again. The difference between the two commands is that a Show command always returns a record with perhaps one or more tables. A List command, on the other hand, always returns just one table.

Other commands, except for a few exceptions, do not return any data. In this case, it suffices to check the return value for SDMS_OK. If an SDMS_OK is returned, it is guaranteed that the command was also processed correctly.

The directory $BICSUITEHOME/src/capi contains a few more files. One of these is jsstub.c. This is a small C program which, from the Scheduling Server’s point
of view, acts as a job server. It obediently fetches new jobs and reports them as being finished after 10 seconds with exit code 0. It does not execute anything, however. This small program is used by developers for running stress tests. A large number of such dummy job servers can be started with no problems at all without putting a heavy load on the PC. However, the Scheduling Server has to work hard to push these windbags to their absolute limits.

It is an application written in C which is used productively in development environments. Knowing the above, it is now possible to see how this program communicates with the server and then processes data.