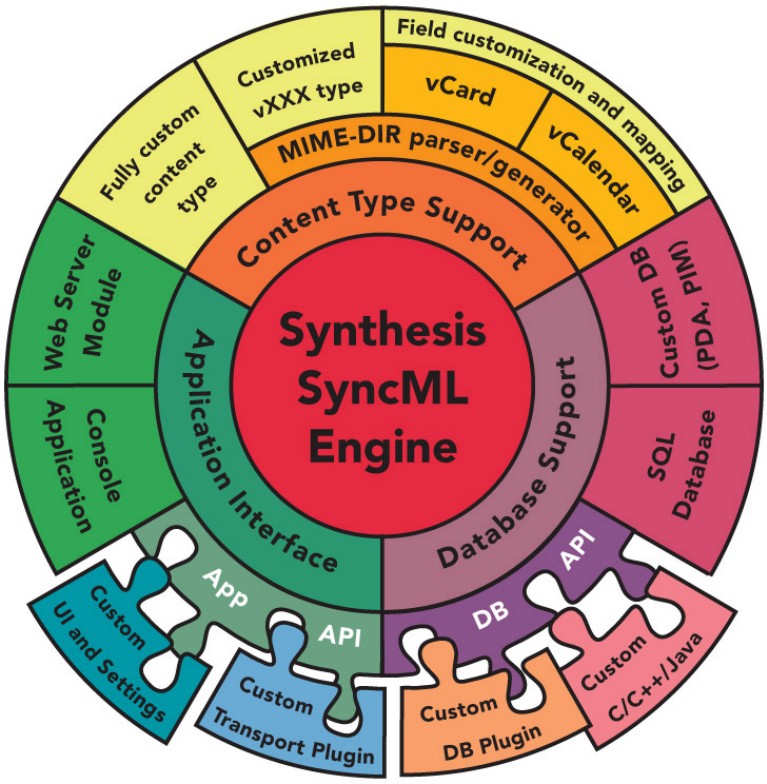
Sustainable
Software Concepts

SYNTHESIS AG

www.synthesis.ch

XML Configuration Reference for Synthesis SyncML Server & Client 3.4 Products

SyncML



This manual was written for Synthesis SyncML Engine Version 3.4.0.0

Synthesis SyncML products uses parts of the following software:

expat - XML parser - http://sourceforge.net/projects/expat

SyncML toolkit - http://sourceforge.net/projects/syncml-ctoolkit/

zlib compression library - http://www.zlib.net/

SQLite 3 database engine - http://www.sqlite.org/

PCRE Library - http://www.pcre.org/license.txt

Disclaimer

1. Introduction

All Synthesis AG SyncML products, clients as well as servers are based on our platform independent SyncML engine. This engine is configured using a single XML config file, which makes replicating or migration of client and server installations very simple.

If you already have worked with a previous version of the Synthesis SyncML engine configuration, please refer to the (new) chapter "What's New?" on page 12 of this manual. Please also consult the product-specific manuals (like Server Manual, Client Manual etc.) for product specific news and step-by-step migration guides.

As both clients and servers share the same core engine, large sections of the configuration is equal or similar in server und clients, different platforms and versions.

Therefore, this configuration reference covers all Synthesis SyncML products that are user-configurable with an XML configuration document. This includes all servers, command-line desktop clients and most versions of the Synthesis SyncML client engine that can be used to build custom clients with the client SDK.
Only ready-to use, device specific versions such as the PalmOS and Windows Mobile clients have no XML configuration).

In the description of a configuration option, the products for which the option is available is listed under the "Available" header line if it does not apply to all versions.

Please note also that this document is a reference manual. It is useful to get an overview of the entire functionality available and of course to create and adapt configuration files. However, it is not a guide for creating new configurations from scratch. We recommend to always use one of the tested and commented sample configuration files included in the product distributions as a starting point.

Note that this manual makes heavy use of cross references (references to related parts in the manual) - which are active links if this manual is viewed as a PDF document. You can just click on any of the cross-referenced chapter numbers to have the PDF viewer show the corresponding page of the manual. Using the "back" button in the PDF viewer, you can always jump back to the original page.

Contents

1. Introduction 3
Contents 4
2. What's New? 12
2.1 New in this manual 12
2.2 New in SyncML Engine 3.2 and newer compared to 3.0 12
2.2.1 General changes 12
2.2.2 New Features 13
2.2.3 How to migrate from 3.0 to 3.2 or newer (up to 3.4) 14
3. Overview 17
3.1 Basic Concepts 17
3.2 Configuration Structure 17
3.3 XML basics 19
3.4 Synthesis Sync Server Config specific XML usage 19
4. Configuration variables and conditional configuration 21
4.1 Sources for values of config variable 21
4.2 Using configuration variables 21
4.3 "expand" attribute 22
4.4 Predefined Configuration Variables 22
4.5 "ifdef/ifndef/if" conditional attributes 23
4.6 "platform" conditional attribute 23
5. Time zone handling 24
5.1 Timestamp representation 24
5.2 Timezone contexts 24
5.3 Time zone specifications 26
6. Scripting Language 27
6.1 What can be scripted? 27
6.2 Embedding script source code in XML 27
6.3 Comments 28
6.4 Statements and Statement Blocks 28
6.5 Identifiers 28
6.6 Data types 28
6.7 Constants/Literals 28
6.8 Script contexts 29
6.9 Variables 29
6.9.1 Context Variables 30
6.9.2 Local variables of a user-defined function 30
6.9.3 Field variables 30
6.9.4 Array variable references 31
6.10 Expressions 32
6.11 Flow control 32
6.12 Macros 33
6.12.1 Defining Macros 33
6.12.2 Marco arguments 34
6.12.3 Using Macros 34
6.13 Functions 35
6.13.1 User defined Functions 35
6.13.2 Built-in Functions 36
6.14 Global built-in Function Reference 36

6.14.1 String functions 37
6.14.2 Regular Expression functions 38
6.14.3 Date and Time functions 38
6.14.4 Time zone related functions 41
6.14.5 Debug log functions 42
6.14.6 Other functions 43
6.15 Debugging scripts 45
7. Filters 47
7.1 Test and Make-Pass modes 48
7.2 Basic filter syntax 48
7.3 Identifiers in filters 49
7.4 CGI Filter Syntax 50
7.5 Special options in CGI filters passed with database path 51
7.6 Filters in the configuration 52
8. General Global Configuration Options 53
8.1 <licensename>, <licensecode>: License 53
8.2 <maxconcurrentsessions>: concurrent sessions limit 53
8.3 <maxmsgsize>: max SyncML message size 53
8.4 <maxobjsize>: maximum object size 54
8.5 <configidstring>: text to identify config 54
8.6 <manufacturer>: text to identify product manufacturer 54
8.7 <model>: text to identify model/product name 54
8.8 <configvar>: define configuration variable 55
8.9 <configmsg>: define configuration variable 55
8.10 <scripting>: Global scripting definitions 55
8.10.1 <function>: User-defined function 55
8.10.2 <macro>: define macro 56
8.10.3 <looptimeout>: maximum loop execution time 56
8.11 <debug>: Debug Option Section 56
8.11.1 <logpath>: Directory path for debug log files 57
8.11.2 <enable>, <disable> 57
8.11.3 <logformat>: select log file format 59
8.11.4 <folding>: dynamic folding for HTML logs 59
8.11.5 <timestamp>, <timestampall>: show timestamps in logs 60
8.11.6 <showthreadid>: show thread ID in logs 60
8.11.7 <timedsessionlognames>: show timestamps in logs 60
8.11.8 <singlegloballog>, <singlesessionlog>: single file log option 61
8.11.9 <appendtoexisting>: append or overwrite existing session logs 61
8.11.10 <logflushmode>: select log file format 61
8.11.11 <subthreadmode>: if and how to show log output from subthreads 62
8.11.12 <fileprefix>, <filesuffix>: text to add at begin and end of logfiles 62
8.11.13 <indentstring>: string to be used for indenting blocks 63
8.11.14 <xmltranslate>: show traffic in XML 63
8.11.15 <msgdump>: dump SyncML traffic to files 64
8.11.16 <sessionlogs>: generate session logs 64
8.11.17 <sepsessionlogs>: No longer supported; use <singlesessionlog>instead 65
8.11.18 <globallogs>: generate global log 65
8.11.19 <logsessionstoglobal>: send session logs to global logfile 65
8.12 <configdate>: set timestamp for config file 65
8.13 <neverputdevinf>: avoid PUT of devinf 66
8.14 <systemtimezone>: override local system time zone 66
8.15 <definetimezone>: define custom time zone as VTIMEZONE 66

9. <transport>: Transport Configuration Section 68
9.1 <keepconnection>: HTTP 1.1 connection 68
9.2 <bufferretryanswer>: buffer last answer for retries 69
9.3 <protocol>: communication protocol 69
9.4 <httpport>: HTTP and OBEX/TCP server port number 70
9.5 <ipaddress>: listener IP address 70
9.6 <obexservice>: OBEX service name 70
9.7 <maxthreads>: Max number of session threads per server process 71
9.8 <maxsessionruns>: Max sessions to be run by a process 71
10. <datatypes>: Data Type Definitions 72
10.1 <fieldlist>: internal data field list 73
10.2 <field>: definition of an internal field 73
10.3 <mimeprofile>: definition of a mime-dir profile 75
10.3.1 <profile>: root profile definition 76
10.3.2 <subprofile>: nested subprofile definition 76
10.3.3 <property>: property definition 77
10.3.4 <value>: property or parameter value storage 79
10.3.5 <enum>: enumerated values 81
10.3.6 <parameter>: property parameter definition 82
10.3.7 <position>: control storage position and repetitions 84
10.3.8 <vtimezonegenmode>: VTIMEZONE generation mode 87
10.3.9 <unfloattimestamps>: handling of floating timestamps 87
10.4 <textprofile>: definition of a text format profile 88
10.4.1 <linemap>: mapping of text based formats to database fields 88
10.4.2 <numlines>: Number of lines to map 88
10.4.3 <inheader>: header lines 89
10.4.4 <allowempty>: empty field handling 89
10.4.5 <headertag>: tagged header handling 89
10.4.6 <valuetype>: type of text field 89
10.4.7 RFC822 email body options 90
10.5 <datatype>: definition of a datatype 91
10.5.1 <use>: MIME-DIR profile, text profile or field list to use for datatype 91
10.5.2 <version>: vCard or vCalendar version 92
10.5.3 <typestring>, <versionstring>: MIME type and version 92
10.5.4 <zippedbindata>: Enable/disable special compressed (non-standard) item format 93
10.5.5 <zipcompressionlevel>: Compression level for <zippedbindata> compression 93
10.5.6 <binaryparts>: Allow unencoded binary in content 93
10.5.7 <unicodedata>, <bigendian>: Unicode content 94
10.5.8 <initscript>: Initialisation of type-specific script context 94
10.5.9 <incomingscript>, <outgoingscript>: Custom pre- and postprocessing items 95
10.5.10 <filterinitscript>, <filterscript>: Script-based data filtering 95
10.5.11 <processitemscript>: Custom processing for incoming items 97
10.5.12 <comparescript>: Custom item comparison 98
10.5.13 <mergescript>: Custom item merge 99
10.5.14 <mimedirmode>: MIME-DIR conformance 100
10.6 RRULE field block 100
11. <server>, <client>: General Server and Client Settings 101
11.1 <maxsyncmlversion>,<minsyncmlversion>: SyncML version support 101
11.2 <sessiontimeout>: Timeout for unfinished sessions 102

11.3 <requestmaxtime>: max time for request processing 102
11.4 <requestmintime>: artifical slow down 102
11.5 <externalurl>: specify URL used to access the server 103
11.6 <requestedauth>,<requiredauth>: SyncML Authentication 103
11.7 <autononce>: MD5 nonce generation mode 104
11.8 <constantnonce>: constant nonce string 104
11.9 <sendrespuri>, <respurionlywhendifferent>: RespURI configuration 104
11.10 <simpleauthuser>, <simpleauthpw>: single user mode 105
11.11 <multithread>: Allow multi-threaded execution 105
11.12 <sessioninitscript>: Session init script 105
11.13 <sessionfinishscript>: Session finish script 106
11.14 <sentitemstatusscript>, <receiveditemstatusscript>: Session level status code handling 106
11.15 <customgetputscript>, <customendputscript>: Creation of custom SyncML Get and Put commands 106
11.16 <customgethandlerscript>: Custom handling of SyncML Get commands 107
11.17 <customputresulthandlerscript>: Custom handling of SyncML Put/Result commands 108
11.18 <waitforstatusofinterrupted>: SyncML command flow option 108
11.19 <relyonearlymaps>: Add resending policy 108
11.20 <debugchunkmaxsize>: LargeObject chunk size limit for testing 109
11.21 <deletinggoneok>: Handling of delete for non-existing items 109
11.22 <usertimezone>: Set user's default time zone 109
11.23 <autoenddateinclusive>: end date for allday events inclusive 109
11.24 <abortonallitemsfailed>: error handling option 110
11.25 <showctcapproperties>: show field support details in device information 110
11.26 <showtypesizeinctcap10>: show size and type in SyncML 1.0 devInf 110
11.27 <enumdefaultpropparams>: enumerate default property parameter's values as property names 111
11.28 <acceptserveralerted>: Acceptance of server alerted sync types 111
11.29 <logfile>: Activity log text file 111
11.30 <logenabled>: Activity log enable 112
11.31 <logformat>: Activity log format 112
11.32 <loglabels>: Activity log header 114
11.33 <logininitscript>, <loginfinishscript>: Pre- and post-login scripts 114
11.34 <datastore>: General Datastore settings 115
11.34.1 <alias>: alternate name for this datastore 116
11.34.2 <dbtypeid>: datastore type ID 116
11.34.3 <displayname>: decriptive name for a datastore 117
11.34.4 <readonly>: read-only datastore 117
11.34.5 <deletewins>: delete overrides replace 117
11.34.6 <tryupdatedeleted>: try to update "deleted" items 118
11.34.7 <reportupdates>: transmit updates to remote 118
11.34.8 <maxitemspermessage>: maximum number of data items per SyncML message 118
11.34.9 <alwayssendlocalid>: send localID (GUID) in all operations (not only adds) 119
11.34.10 <conflictstrategy>, <slowsyncstrategy>, <firsttimestrategy>: sync conflict resolution strategy 119
11.34.11 <typesupport>: datastore's supported types 120

11.34.12 <use>: use a datatype 120
11.34.13 <ds12filters>: enable SyncML DS 1.2 filtering 121
11.34.14 <daterangesupport>: enable date range filtering 121
11.34.15 <acceptfilter>: check incoming items 121
11.34.16 <silentdiscard>: discard not accepted items silently 122
11.34.17 <localdbfilter>: filter subset of datastore 122
11.34.18 <invisiblefilter>: filter invisible items 122
11.34.19 <makevisiblefilter>: make item visible 123
11.34.20 <makepassfilter>: make incoming items pass 123
11.34.21 <datastoreinitscript>: script called before accessing database 123
11.34.22 <datastorefinishscript>: script called after accessing database 126
11.34.23 <adminreadyscript>: script called when admin data (targets, maps) are read 127
11.34.24 <syncendscript>: script executed at end of sync 127
11.34.25 <alertscript>: script called at sync alert 127
11.34.26 <alertprepscript>: script called before sending sync alert 128
11.34.27 <sentitemstatusscript>: script to handle status codes for sent items 128
11.34.28 <receiveditemstatusscript>: script to handle status codes for received items 129
11.34.29 <resendfailing>: re-send failing items in next session 129
11.34.30 <timeutc>, <timestamputc>: type of database timestamp 130
11.34.31 <datatimezone>: timezone for database timestamps 130
11.34.32 <userzoneoutput>: output data in user zone 130
11.34.33 <datacharset>: character set to be used for database strings 131
11.34.34 <datalineends>: encoding of line ends within database strings 131
11.34.35 <updateallfields>: always update all fields 132
11.34.36 <fromremoteonlysupport>: Support for "one-way from remote sync" 132
11.34.37 <synctimestampatend>: How to determine "time of last sync" 132
11.34.38 <storesyncidentifiers> (or <storelastsyncidentifier>): custom "time of last sync" identifier 133
11.34.39 <resumesupport>: support for resuming interrupted sync session 133
11.34.40 <resumeitemsupport>: support for resuming half-transmitted data items after interrupted sync 133
11.34.41 <fieldmap>: mapping datatype's fields to database fields 134
11.34.41.1 <map>, <mapredefine>: mapping a datatype field to a database field 134
11.34.41.2 <automap>: auto-map internal to DB fields 137
11.34.41.3 <initscript>: initialize accessing database 137
11.34.41.4 <afterreadscript>: post-process item read from database 138
11.34.41.5 <beforewritescript>: prepare writing item to database 139
11.34.41.6 <finalisationscript>: finalize written items 140
11.34.41.7 <finishscript>: finish access to database 141
11.35 <superdatastore>: combined datastore definition 141
11.35.1 <contains>: Include a datastore in a superdatastore 142
11.35.2 <dispatchfilter>: filter to direct incoming items 142
11.35.3 <guidprefix>: prefix for item ID 142
11.36 <remoterule>: special rules for specific remotes 143
11.36.1 <finalrule> 143
11.36.2 device identification tags for <remoterule> 143
11.36.3 <descriptivename> 144
11.36.4 <limitedfieldlengths>: device has short fields 144
11.36.5 <noemptyproperties>: do not send empty properties 145

11.36.6 <updateclientinslowsync>: update client records during slowsync 145
11.36.7 <updateserverinslowsync>: update server records during slowsync 145
11.36.8 <noreplaceinslowsync>: never update client records during slowsync 146
11.36.9 <ignoredevinfmaxsize>: ignore maximum field size reported in client's devInf 146
11.36.10 <dspathindevinf>, <dscgiindevinf>: how to show datastore name in devInf sent to client. 146
11.36.11 <allowmessageretries>: allow client to send the same message twice 147
11.36.12 <completefromclientonly>: allow client to send the same message twice 147
11.36.13 <forcelocaltime>: always send time information as localtime 147
11.36.14 <forceutc>: always send time information as localtime 148
11.36.15 <treataslocaltime>: always treat received information as localtime 148
11.36.16 <treatasutc>: always treat received information as UTC 148
11.36.17 <nocontentfolding>: prevent folding long lines 148
11.36.18 <autoenddateinclusive>: end date for allday events inclusive 149
11.36.19 <outputcharset>: set default output character set 149
11.36.20 <inputcharset>: set default input character set 149
11.36.21 <legacymode>, <lenientmode>: use relaxed conformance modes 149
11.36.22 <rejectstatus>: reject sync with device 150
11.36.23 <requestmaxtime>: max time for request processing 150
11.36.24 <rulescript>: script to execute if rule applies 150

12. <server type="sql"/"odbc">, <client type="sql"/"odbc">: SQL/ODBC based Server or Client Config 151

12.1 SQL Statement processing 151
12.1.1 Placeholders for all SQL statements 152
12.1.2 Placeholders for SQL statements within <datastore> 152
12.1.3 Placeholders for SQL data access statements within <datastore> 153
12.1.4 Executing SQL statements from scripts 154
12.2 <datasource>: ODBC data source name 155
12.3 <dbuser>: ODBC database user name 156
12.4 <dbconnectionstring>: ODBC database connection string 156
12.5 <dbpass>: ODBC database password 157
12.6 <preventconnectattrs>: prevent setting connection attributes 157
12.7 <dbtimeout>: ODBC timeout 157
12.8 <afterconnectscript>: Script executed whenever new DB connection is opened. 157
12.9 <transactionmode>: Transaction isolation mode 158
12.10 <usecursorlib>: usage of ODBC cursor library 158
12.11 <textmap>, <textauth>, <textpath>: outdated - no longer available 158
12.12 <cleartextpw>: plain text password in database 159
12.13 <md5userpass>: MD5 digest passwort in database 159
12.14 <md5hex>: MD5 digest stored as hex string in database 159
12.15 <getdevicesql>, <newdevicesql>, <savenoncesql>, <saveinfosql>: Device management 160
12.16 <userkeysql>: query for user authentication 161
12.17 <logincheckscript>: custom login checking script 162
12.18 <timestampsql>: query for getting database time 163
12.19 <writelogsql>: SQL statement to write activity log entry 163
12.20 <datastore type="sql"/"odbc">: SQL and ODBC Datastore specific settings 164
12.20.1 <folderkeysql>: get data subselection key 164

12.20.2 <synctargetgetsql>, <synctargetnewsql>, <synctargetupdatesql>, <synctargetdeletesql>: Sync target management ... 165
12.20.3 <synctimestamp>: format for timestamps in target table ... 169
12.20.4 <lastmodfieldtype>: modified time stamp type ... 169
12.20.5 <selectmapallsql>, <insertmapsql>, <updatemapsql>, <deletemapsql>: Map table management ... 169
12.20.6 <sqlitefile>: SQLite database file name ... 171
12.20.7 <sqlitebusytimeout>: SQLite database file name ... 172
12.20.8 <quotingmode>: how ODBC strings must be escaped for the database ... 172
12.20.9 <dbcanfilter>: use filtering in WHERE clause ... 172
12.20.10 <earlycommit>: commit at end of SyncML message exchange ... 173
12.20.11 <multicursor>: no longer supported in version 3.0 ... 173
12.20.12 <commititems>: commit each item update ... 173
12.20.13 <modtimestamp>: combined date and time for modification timestamp ... 173
12.20.14 <selectidandmodifiedsql>: read IDs and timestamps ... 174
12.20.15 <selectdatasql>: read record from database ... 174
12.20.16 <insertdatasql>, <updatedatasql>, <deletedatasql>, <zapdatasql>: write records to database ... 175
12.20.17 <ignoreaffectedcount>: Ignore SQLRowCount ... 175
12.20.18 <obtainidafterinsert>, <obtainlocalidsql>, <determineidonce>, <minnextid>, <specialidmode>, <insertreturnsid>, <localidscript>: local object ID management ... 176
12.20.19 <map>: SQL specific field mapping features ... 178
12.20.20 <array>: definition of master - detail record structures ... 179
12.20.21 <maxrepeat>, <repeatinc>, <storeempty>: detail record storage options ... 181
12.20.22 <noitemsfilter>: detail record storage filter ... 181
12.20.23 <selectarraysql>, <deletearraysql>, <insertelementsql>: detail record SQL ... 182
12.20.24 <alwaysclean>: clean detail records on insert ... 182
12.20.25 <optionfilterscript>: prepare SQL filter according to options ... 182

13. <server type="textdb">, <client type="textdb">: Text File Based Server or Client ... 183

14. <server type="plugin">, <client type="plugin">: Plugin Based Server or Client Config ... 184

14.1 plugin module: global settings ... 184
14.1.1 <plugin_module> ... 184
14.1.2 <plugin_sessionauth> ... 185
14.1.3 <plugin_deviceadmin> ... 185
14.1.4 <plugin_params> ... 185
14.2 <datastore type="plugin">: Plugin Datastore specific settings ... 185
14.2.1 <plugin_datastoreadmin> ... 185
14.2.2 <plugin_module> ... 186
14.2.3 <plugin_params> ... 186
14.2.4 <plugin_debugflags> ... 186
14.2.5 <plugin_module_admin>,<plugin_params_admin>,.....<plugin_debugflags_admin 186
14.3 plugin module "SDK_textdb" ... 187
14.3.1 Files of the textdb ... 187
14.3.2 PluginParams of the textdb ... 187
14.4 plugin module "FILEOBJ" ... 188

14.4.1 Files of the fileobj modules 188
15. <client>, <server>: Synthesis SyncML Engine library only configuration tags 189
15.1 <binfilespath>: Path for persistent storage of client settings and admin data 189
15.2 <binfilesactive>: enable binfile based admin 189
15.3 <crcchangedetection>: enable CRC based change detection 189
16. <client>: Command line client-only configuration tags 191
16.1 <defaultsyncmlversion>: Set default SyncML Version to start a session 191
16.2 <defaultauth>: Set default auth method 191
16.3 <defaultauthencoding>: Set default auth encoding 191
16.4 <defaultauthnonce>: Set default nonce 192
16.5 <newsessionforretry>: Use a new sessionID for retries 192
16.6 <originaluriforretry>: Use original URI for retry 192
16.7 <smartauthretry>: Use smart retry attempt variations 192
16.8 <putdevinfatslowsync>: Always send Device Info at Slowsync 193
16.9 <localdbuser>, <localdbpassword>: Login to local database 193
16.10 <nolocaldblogin>: Prevent local DB login 193
16.11 <syncmlencoding>: SyncML encoding format 193
16.12 <serverurl>: Remote SyncML server URL 194
16.13 <serveruser>, <serverpassword>: Login to remote SyncML server 194
16.14 <sockshost>, <proxyhost>: Proxy servers 194
16.15 <proxyuser>, <proxypassword>: Proxy auth 194
16.16 <transportuser>, <transportpassword>: Login to remote SyncML server 195
16.17 <syncrequest>: Request to sync a datastore 195
16.17.1 <dbpath>: path of remote server's datastore 195
16.17.2 <syncmode>: Synchronisation mode 195
16.17.3 <slowsync>: Force a slow sync 196
16.17.4 <localpathextension>: local datastore options 196
16.17.5 <recordfilter>: define SyncML DS 1.2 record filter 197
16.17.6 <recordfilterinclusive>: define inclusive SyncML DS 1.2 record filter 197
17. List of built-in timezones 198
18. Error codes 199
18.1 SyncML Status Codes 199
18.2 Internal Error Codes 200
19. Index 202
19.1 Alphabetic Index of all config XML tags 202

2. What's New?

2.1 New in this manual

- This chapter – intended as a quick overview to see what is new in this release. **In particular, paragraph 2.2.3 "How to migrate from 3.0 to 3.2" details the (simple) steps to take to upgrade an existing installation.**
- An overview diagram "SySync_script_call_flow.pdf" (not in this manual itself, but as a separate PDF document in the product package) showing which scripts (PRO version only) are called when in the process of sync session, and what is the typical use of a script. This is intended as a reference card to quickly find out which scripts to use to accomplish a certain customisation task.
- A separate chapter (see chapter 5) about time zones and how these are handled in the SyncML engine. **We recommend to read this chapter, as correct handling of timezones is crucial for a successful calendar sync.**

2.2 New in SyncML Engine 3.2 and newer compared to 3.0

2.2.1 General changes

- New Version numbering convention (borrowed from Linux kernel numbering):
  Odd numbers are development versions, that may be released from time to time as beta or for solving very specific customer needs in a project.
  Even numbers are official release versions.

- Completely revised and greatly enhanced handling of **timezones**. Apart from a lot of new features and script functions to work efficiently with time zones, the **most important general change is the internal representation of time stamps**. In engine versions before 3.1, the internal timestamp value was always represented in UTC (Universal Time Coordinated) along with a time zone offset indicating the time zone context. In engine version starting with 3.1, the **internal timestamp value represents time in the context of the time zone identifier that is attached to it**. This is a small, but important change, which allows to handle all aspects of timestamps, including "floating" timestamps, in a consistent way throughout the entire data path from backend database to remote SyncML device. See chapter 5 for more details about timestamps and timezones.
  Most existing setups are not or only slightly affected by these changes –only configurations which were using the LOCALIZEDASUTC, RELATIVEASUTC, UTCASRELATIVE, LOCALZONEOFFSET, SETZONEOFFSET, ISRELATIVE and SETRELATIVE functions in scripts or a "zoneoffset_xxx" conversion mode need to be adapted – and normally the adapted version is simpler and much easier to understand as the new timezone system is more logical and consistent.
  Our sample config for the PRO servers until 3.1 used RELATIVEASUTC and "zoneoffset_secs" two times – please refer to instructions in paragraph **2.2.3 "How to migrate from 3.0 to 3.2" how to update your config**.

2.2.2 New Features

- **SyncML engine now available as library** – Starting with Synthesis SyncML engine 3.1, the core engine with all SyncML functionality, SQL/ODBC/SQLite and plugin database interfaces, complete XML configuration is now available as a dynamically linkable library for various platforms. Using the Synthesis SyncML SDK, custom SyncML applications can be built in native languages (C/C++/Delphi etc.) as well as in Java or .net.

- **Extended support for symbolic time zones** – which can be referenced and stored in the database by name. These symbolic time zones handle DST rules automatically (not only for the system's local zone, but any time zone). See chapter 5 for an overview of time zones. The PRO version also has many new built-in script functions for working with time zones: ZONEOFFSET(), TIMEZONE(), VTIMEZONE(), SETTIMEZONE(), SETFLOATING(), CONVERTTOZONE(), CONVERTTOUSERZONE(), USERTIMEZONE(), SETUSERTIMEZONE(), ISDATEONLY(), DATEONLY(), ISFLOATING() – see (6.14.4).

- **Per user time zones** – using <usertimezone> (see 11.22) and SETUSERTIMEZONE() (see 6.14.4) it is possible to assign a default time zone on a per-user level. This is important with client devices that do not support UTC, and must be server in a specific local time zone.

- **Per datastore time zone** – using the new <datatimezone> (see 11.34.31) which replaces former <timeutc> (still supported for compatibility).

- **Record level or field level time zones** e.g. for storing originating time zone of a calendar entry along with the entry, or to define floating timestamps (timestamps not bound to a time zone). To map the timezone of a timestamp field to a database string field, use the new "zonename" database field type (see 11.34.41.1). To create TZ, DAYLIGHT and TZID values in vCalendar/iCalendar data formats, new conversion modes "TZ", "DAYLIGHT", "TZID" have been added (see 10.3.4). To include full time zone specification in VTIMEZONE format, a predefined <subprofile mode="vtimezone"> (see 10.3.2) has been added.

- PRO only: Support for **maintaining relational links** between items by providing an optional post-processing step at the end of the session (when the data sets are known to be in sync) trough the new <finalisationscript> (see 11.34.41.6).

- PRO only: Support for **regular expression** search, replace and pattern split (using new script functions REGEX_FIND, REGEX_MATCH, REGEX_SPLIT, REGEX_REPLACE, see 6.14.2). These can greatly simplify value conversion scripts.

- PRO only: Script language now **supports the WHILE() statement** in addition to the LOOP statement.

- PRO only: **A lot of new built-in script functions**: SYNCMLVERS(), EXPLODE(), ABS(), SIGN(), NUMFORMAT(), DAYUNITS(), MONTHDAYS(), PARSEEMAILSPEC(), MAKEEMAILSPEC(), SLEEPMS(), TIMESTAMPTODBINT(), DBINTTOTIMESTAMP(), CONVERTTODATAZONE(), ADDTARGETCGI(), SETRECORDFILTER(), SETDAYSRANGE(), TARGETSETTING() - see (6.14).

- **Enhanced support for calendar content formats** – new conversion modes "valuetype", "tzid", "tz", "daylight", "autodate", "autoenddate" see (10.3.4), new script functions to detects/generate all-day events: ALLDAYCOUNT(), MAKEALLDAY() and to generate and expand recurring events: RECURRENCE_DATE(), RECURRENCE_COUNT(), see (6.14.3).

- **SyncML max message** and **object size limits** are now **configurable** using <maxmsgsize>, <maxobjsize> (see 8.3 and 8.4).

- **Debug logging enhanced** – now shows timestamps with millisecond resolution. Blocks now have a "enclosing" navigation link which allows jumping to the beginning or end of an enclosing block in the hierarchy.

- **SyncML message dumping enhanced** – messages are now saved on a per-session basis and message dumping can be switched on and off in scripts, e.g. based on what user is logged in or what device type is being synchronized. See <msgdump> (8.11.15), <xmltranslate> (8.11.14) and the related script functions SETMSGDUMP and SETXMLTRANSLATE (6.14.5).

- **Script execution logging enhanced** – colorized to recognize comments, executed code and conditionally skipped code at a glance. Condensed output to avoid too much detail by default (but new "expressions" debug option still allows in-detail expression debugging, see 8.11.2). New script functions DEBUGSHOWITEM() and DEBUGSHOWVARS() can be used to show the contents of a sync item or of all local script variables in the log, see 6.14.5.

- **Optional parameters for built-in script functions** – many of the new 3.1 engine's scripting functions and some of the existing functions now have optional parameters which can be omitted when no special non-default behaviour is needed.

- **Config variables**: These are variable strings that can be referenced in the XML config using $(varname) syntax (see 4.2) or for conditional config sections (see 4.5).
  Config variables are either preset by the operating environment (with values like engine version, device ID, file paths to standard config, temp, user directories etc., see 4.4), defined from the command line using the –D option (for executable program versions, use -h option to show syntax options), via the "/configvars" engine settings key (for library versions).

- **Conditional config**: All config XML tags now have generic "if", "ifdef" and "ifndef" attributes (see 4.5) that can be used to make certain config sections dependent on config variables (e.g. SyncML engine version). This simplifies using the same config file for different versions of the SyncML engine, different platform, different operating conditions.

2.2.3 How to migrate from 3.0 to 3.2 or newer (up to 3.4)

Existing 3.0 installations usually need some changes when based on our 3.0 configuration sample to run with SyncML engine 3.2 or later. Some configurations that did not make use of time-zone related features might need no changes at all.
Please check the following:

- **Obsolete time zone conversion modes:** zoneoffset_secs, zoneoffset_mins, zoneoffset_hours are no longer available. These have been replaced by the much more versatile "tz" mode (see 10.3.4).

If your configuration is based on the sample configuration as delivered with the 3.0 version, you'll likely have two occurrences of zoneoffset_secs for the "TZ" field. The way this field was defined in 3.0 and earlier was not correct vCalendar 1.0 usage anyway, and almost no client supported it, so it had no real meaning.
If you want to be prepared to save the originating time zone along with each calendar item, change the type of the TIMEZONE field in the "calendar" <fieldlist> from string:

```
<field name="TIMEZONE" type="string" compare="never"/>
```

to integer

```
<field name="TIMEZONE" type="integer" compare="never"/>
```

This will cause your TIMEZONE field in the database store a **minute** offset to UTC instead of a **seconds** offset as in 3.0.
Then move the first occurrence of "<property name="TZ">...</property>" one level up (out of "<subprofile name="VEVENT" …>" into "<profile name="VCALENDAR"…>"), and delete the second occurrence of "<property name="TZ">...</property>":

```
<profile name="VCALENDAR" nummandatory="1">

 <property name="VERSION" mandatory="yes">
   <value conversion="version"/>
 </property>

 <property name="TZ">
   <value field="TIMEZONE" conversion="tz"/>
 </property>

 <!-- sub-profile for events -->
 <subprofile name="VEVENT" nummandatory="1" field="KIND" ...>

   <property name="TZ">
     <value field="TIMEZONE" conversion="zoneoffset_secs"/>
   </property>

   ...
 </subprofile>

 <!-- sub-profile for events -->
 <subprofile name="VTODO" nummandatory="1" field="KIND" ...>

   <property name="TZ">
     <value field="TIMEZONE" conversion="zoneoffset_secs"/>
   </property>
```

Alternatively, if you do not need the TIMEZONE information in your application, just remove the two <property name="TZ">...</property> definitions entirely from the config file.

- **<timeutc> and <timestamputc> should be changed to <datatimezone>:** No change is needed, but the SyncML engine will show a warning when <timeutc> or <timestamputc > (see 11.34.30) are used as these should be replaced by the more versatile <datatimezone> (see 11.34.31).

- **PRO only – script functions no longer supported:** LOCALIZEDASUTC, RELATIVEASUTC, UTCASRELATIVE, LOCALZONEOFFSET, SETZONEOFFSET, ISRELATIVE and SETRELATIVE are not supported any more because they do not fit with the new enhanced timezone handling.
  If your configuration is based on the sample configuration as delivered with the 3.0 version,

you'll likely have two occurrences of RELATIVEASUTC in the <comparescript> of <datatype> vCalendar10. These can be simply removed. So the line that originally reads:

```
RES = COMPARE(DATEONLY(RELATIVEASUTC(TARGET.DTSTART)),
DATEONLY(RELATIVEASUTC(REFERENCE.DTSTART)));
```

can be replaced by

```
RES = COMPARE(DATEONLY(TARGET.DTSTART),
DATEONLY(REFERENCE.DTSTART));
```

3. Overview

In order to understand a server's or client's configuration, an overview of the basic building blocks and concepts in the Synthesis SyncML engine is helpful.

3.1 Basic Concepts

The Synthesis SyncML engine performs three conceptually more or less separate tasks:

- Running the SyncML protocol. SyncML is an open industry standard and therefore there are clear specifications about how the SyncML protocol must be implemented and run. **Therefore, there is not a lot of configuration needed for the SyncML protocol engine itself**.
- Encoding and decoding the data that is synchronized with the SyncML protocol. SyncML itself is designed to synchronize any type of data, even proprietary, customer-defined types. However, to make a SyncML server or client interoperable, it must support some standard datatypes. Today, this includes the widely used vCard format for contact information and vCalendar for events and tasklists, and a number of RFC(2)822 based email formats for email synchronisation. Synthesis SyncML products support these standard formats, but they give the user complete freedom about all the details (you can define a server or client that can handle 37 phone numbers per contact if this is important in your context). In addition, custom formats based on plain text, MIME-email or MIME-DIR can be defined. **Covering all the possible options of the vCard/vCalendar formats and even allowing to define new formats makes the datatype configuration quite complex and big - however in most applications, it is sufficient to slightly modify one of the provided sample datatypes.**
- Interfacing the SyncML data with a server's or client's database. The complexity of this task depends largely on the type and kind of database. Our text file based demo versions need almost no configuration, because the data is simply saved to tab-separated text files. On the other hand, our ODBC-based products are designed to interface with existing databases, which requires very flexible configuration options to handle field mapping and data conversions. **In most real-world applications, configuration of the database interface is what needs most attention and customisation.**

3.2 Configuration Structure

According to the basic tasks decribed above, the config for a Synthesis SyncML server or client is structured as follows:

- client and server: **General global options**, (see 8) such as:
  - a <debug> section for configuring level of debugging log information
  - a <scripting> section for defining global scripting functions and macros (only in PRO versions)
  - a <licence> section for entering license codes (not all versions need license codes)

- server only: Configuration of the **transport**, that is how clients can access the server (see 9)

- server and client: **Datatype definitions** (see 10). This consists of the following sub-sections:
  - one or multiple <fieldlist> sections, which define the internal representation of a data type as a list of typed fields (or array fields).

  - one or multiple <mimeprofile> and/or <textprofile> sections, which define a mapping between internal fields and a data format based on MIME-DIR (such as vCard or vCalendar) or plain text (such as email or notes).
  - one or multiple <datatype> sections, which define actual data types based on field lists and MIME-DIR or text profiles.

- server only: a <server> section which defines the **server databases** (see 11 and 12)
- client only: a <client> section which defines the **client databases** (see 11 and 12)
  - server and client: the <server> or <client> section contains one or multiple <datastore> sections which each define a database. The definition provides the necessary **mapping information between the internal fields from the field list and the database itself**. For the ODBC based products this includes all SQL statements needed to read, modify, insert and delete data as well as a mapping table assigning SQL-names to internal field names. For the plugin based datastores, the mapping is between internal field names and the plugin API data format's names. For both types of datastores, importants settings like **database character set** and **line end format** can be defined here.
- client only: one or multiple <syncrequest> sections which define what databases should be synced with a server when the client application is started (not available in all client versions).
- server and client: optional <remoterule> sections which define **special options for a certain type of remote SyncML client or server**. This is normally used in servers to control device-specific behaviour.

The order in which the elements appear in the config file does not matter unless a section refers to definitions in another sections (like <datastore> referencing <field>s, or <datatype>s referencing <mimeprofiles>) - in this case the defining section must appear before the referring section in the config file. Generally, we recommend using the order as outlined above (and also used in the sample config files).

An "empty" server config file looks like this (a client would be similar except that there was a <client> section instead of the <server> section):

```
<?xml version="1.0" ?>
<sysync_config version="1.0">

  <debug>
    <!-- debug options -->
  </debug>

  <transport type="xxx">
    <!-- transport related options -->
  </transport>

  <datatypes>
    <!-- definitions of data types -->
  </datatypes>

  <server type="odbc">
    <!-- definitions of server database access -->
  </server>

  <remoterule>
```

```
    <!-- special rule for some device -->
  </remoterule>

</sysync_config>
```

We recommend to use the sample config file as a starting point, because there are quite complex parts (especially datatype definitions) which are hard to create from scratch. In many applications, modest modifications to one of the sample files is sufficient anyway.

3.3 XML basics

The configuration file is formatted in XML, which is a tagged text format. Any text editor (including Windows Notepad) can be used to edit XML files. In addition, there are many XML-aware text editors or specialized XML editors. To view (but not edit) XML files neatly formatted and colorized, they can be opened with a web browser like Firefox.

We cannot give a real introduction to XML here, but here are just a few notes about XML syntax in case you are not familiar with it already:

- An XML tag consist of text enclosed in angle brackets like:
  `<this>`
- XML tags must always appear in pairs:
  `<this>something in between</this>`
- the "something in between" can be plain text or other paired tags:
  `<this><that>some text</that></this>`
- Instead of writing:
  `<this></this>`
  for a tag pair with "nothing in between" (tag with no contents), it can be abbreviated as:
  `<this/>`
- Tags can have attributes:
  `<this attribute="value" another="value2">`
  Attribute values must always be enclosed in double quotes.
  In Synthesis Sync Server config, tags with attributes are often tags with no contents, so many config tags might look like:
  `<this attr1="value1" attr2="value2"/>`
  (note the slash at the end)
- XML allows inserting comments. A comment starts with <!-- and ends with -->:
  `<!-- this is a comment -->`
- Formatting does not matter (except for string values, see below), but it makes XML much more readable when nested contents are indented like in the sample config files. Most XML enabled tools do this automatically or have an option for it.

3.4 Synthesis Sync Server Config specific XML usage

A few notes on the way XML is used in Synthesis Sync Server:

- **String values:** They are used exactly as written in the config file, except that leading and trailing whitespace is removed first. All other contained spaces, control characters and

line ends are preserved. So when formatting the XML source nicely, make sure you don't break strings into muliple lines that should be one line (such as directory paths).

- **C-String values:** These are strings that are parsed like in the C programming language as follows: Actual line ends are ignored, but the following escape sequences can be used to insert special characters into the string:

\t is used to insert a TAB character

\r is used to insert a CR character

\n is used to insert a LF (linefeed) character

\xXX is used to insert the character having an ASCII-code of XX (in hexadecimal). Note that the octal form \0XX available in the C language is NOT supported.

**** is used to insert a single backslash character.

- **Boolean values:** "yes", "true", "on", "1" can be used for true, "no", "false", "off", "0" can be used for false.

4. Configuration variables and conditional configuration

New in 3.1: Configuration variables (or short "*config variables*") are a new concept introduced with the 3.1 SyncML engine to allow parametrizing some values within a config file from "the outside" without needing to edit the config file itself. Using conditional sections in a config file, the same file can be used for different setups controlled by configuration variables.

4.1 Sources for values of config variable

There are four sources for these "outside" values:

- From the operating environment – these are values like file system paths to various platform specific directories (like temp dir, application dir etc.) or other values like current user name (see 4.4 for a list).
- From the SyncML engine itself – like the version of the SyncML engine
- Supplied from another program or the user: via the –D command line option for standalone SyncML applications or via the "/configvars" settings key when the SyncML engine is used as a libary with the client or server SDK API – see separate SDK docs for details.
- Finally, the <configvar> directive (see 8.8) can be used to define config variables in the config file itself.

The first two sources are predefined by the engine. See 4.4 for a list of commonly supported configuration variables. Depending on the platform or engine variant, there might be additional variables predefined (and documented in the specific product documentation).

4.2 Using configuration variables

Configuration variables can be used within many string literals in the configuration using the syntax **$(*configvarname*)**.

This syntax is generally recognized in strings that specify file system paths.

In all other tags, the *$(configvarname)* syntax is **not** recognized by default, but can be switched on using the "expand" attribute (see 4.3).

In XML tag attributes, the $(configvarname) syntax is usually **not** supported, however there are exceptions such as the <configvar> tag, see 8.8.

By default, expansion of config variables is recursive, which means that if the value of a config variable contains another *$(configvarname)*, this is expanded as well. To avoid recursive expansion, the "expand" attribute (see 4.3) can be set to "single".

Note: before version 3.2.0.11, expanding tags with pure numeric, enumerated or timestamp values was not supported. From 3.2.0.11 onwards, $(configname) expansion works for all tags (but for most tags only if the "expand" attribute is set, see above).

Config variables can also be used to control conditional configuration (see 4.5).

4.3 "expand" attribute

All tags that support configuration variable expansion (see 4.2) can have a "expand" attribute to control if and how to expand *$(configvarname)*. Possible values for "expand" are:

- "no" : do not expand
- "single" : expand once, but do not try to expand result again
- "yes" : expand recursively.

4.4 Predefined Configuration Variables

The following list contains the configuration variables generally available in most Synthesis SyncML products. Depending on the platform or engine variant, there might be additional variables predefined (and documented in the specific product documentation).

version	Version string of the Synthesis SyncML engine, like "3.4.0.0"
hexversion	Synthesis SyncML engine version as 32-bit hex like MMmmssbb (MM=major, mm=minor, ss=subversion, bb=build).
manufacturer	Manufacturer string that is also communicated to remote parties in the SyncML device Information. For Synthesis SyncML engine library products, this can be configured using the <manufacturer> tag (see 8.6).
model	Model (product name) string that is also communicated to remote parties in the SyncML device Information. For Synthesis SyncML engine library products, this can be configured using the <model> tag (see 8.7).
platformname	name of the current OS platform (like Windows, Linux, iPhoneOS…)
platformvers	version string of the current OS platform (like "5.1.1732")
globcfg_path	global system-wide config path (such as C:\Windows or /etc)
loccfg_path	local config path (such as exedir or user's dir)
defout_path	default path to writable directory to write logs and other output by default
temp_path	path where we can write temp files
exedir_path	path to directory where executable resides
userdir_path	path to the user's home directory for user-visible documents and files
appdata_path	path to the user's preference directory for this application
prefs_path	path to directory where all application prefs reside (not just mine)
device_uri	URI of the device (usually unique ID or URL identifying the device or server)
device_name	Name of the device (like a model or brand name)
user_name	name of the currently logged-in user
conferrpath	for Synthesis SyncML engine library only: path of the file to output configuration parsing error messages. Can be set to "console" to direct the error messages to the standard output (note that a usable standard output might not exist for certain platforms).

4.5 "ifdef/ifndef/if" conditional attributes

New in 3.1: These attributes are available in every XML tag and can be used in a similar way as the "platform" attribute (see 4.6) to make sections of the configuration dependent on certain conditions:

- "**ifdef**": used in the form <sometag ifdef="configvarname"....>. This will conditionally include <sometag> and all tags contained only if "configvarname" is an existing *config variable*.
- "**ifndef**": same as "ifdef", but condition reversed – *config variable* must not exist to include the tag in the config.
- "**if**": used in the form <sometag if="configvarname=value" ...>. This will compare the *config variable* with the specified value. Allowed comparison operators are "=", ">", "<", "!=", ">=", "<=". The comparison is a string comparison, except when comparing the "version" variable, which is compared such that "newer version > older version" is always true (which would not always be the case with string comparison).

4.6 "platform" conditional attribute

Usually, a Synthesis SyncML server or client configuration file is largely platform independent. However, some specifications, such as file paths, are always platform dependent. Since version 2.9.8.5, every tag can be made platform-specific by adding a *platform="xxx"* attribute. *xxx* can be "win32", "linux" or "macosx" at this time. Tags having a *platform* attribute are only evaluated on the specified platform.
This allows using a single config file for multiple platforms. For example, the debug log path usually varies depending on the platform:

```
<logpath platform="win32">C:\logs\syncml</logpath>
<logpath platform="linux">/var/log/syncml</logpath>
<logpath platform="macosx">/private/var/log/syncml</logpath>
```

5. Time zone handling

With the Synthesis SyncML engine 3.1, a completely revised time zone handling has been implemented that allows much more flexibility than before, and allows handling time zone association with timestamp information down to the single field level.

Note: For setups designed for SyncML engine before 3.1, the new engine behaves fully compatible when used with existing 3.0 configuration file, with the exception of a few rarely used scripting functions (LOCALIZEDASUTC, RELATIVEASUTC, UTCASRELATIVE, LOCALZONEOFFSET, SETZONEOFFSET, ISRELATIVE and SETRELATIVE) which are no longer available. In config the 3.0 engine config samples, RELAVTIVEASUTC is used twice, which can be simply removed as the new timestamp representation makes the use of RELATIVEASUTC functionally obsolete.

5.1 Timestamp representation

Timestamp fields consist of two parts:

- The **timestamp value** itself (internally represented as 64bit integer counting milliseconds passed since -4712-01-01 00:00:00 on most platforms). When a timestamp is converted e.g. from a string representation into internal format, the timestamp represents date and time exactly as found in the input, regardless of eventual time zone information. **This is the key difference between Synthesis SyncML engine 3.0 and 3.1 timezone handling – in 3.0 and earlier, timestamps were always converted to UTC.**

- The **time zone context** where a timestamp value belongs to or originates from. The context is either a plain numeric offset from UTC (like: 1 hour east of UTC, which applies for example for Zürich local winter time), or it can be in symbolic form which handles winter and summer (daylight savings) time (like "CET/CEST" meaning Central European Time and Central European Summer Time, which is 1 hour east of UTC in winter, and 2 hours in summer).
  Timestamps that are not associated with a specific time zone are called **floating timestamps**. **Date-only values** are normally floating, as they usually refer to a specific calendar day and not a absolute point in UTC time.
  External string representations for timestamps sometimes include time zone information (like the "Z" in ISO8601 UTC format: 20071212T110000**Z** or an explicit offset like in 20071212T120000+01) . Sometimes, external representation does not include time zone information directly, but timestamps are still implicitly meant in a specific time zone context, like the system's current time zone for example). Therefore it is important to understand what different implicit time zone contexts exist within a SyncML session and how timestamp values are affected by "travelling" through these contexts.

5.2 Timezone contexts

The Synthesis SyncML Engine 3.1 and later maintains the following time zone contexts, ordered starting with most general and global context and ending with most specific context:

- **System time zone context.** This is the time zone set in the operating system which runs the SyncML engine. It can be referenced by name by the string "SYSTEM". Usually, the parameters (offset and daylight saving switching rules) are obtained from the operating

system automatically. It can be overridden by using the <systemtimezone> configuration tag (see 8.14).

- **Datastore time zone context.** This is a per-datastore time zone context which is used for all timestamps stored in the database. This means that timestamp values are converted from their original timezone context to the *datastore time zone context* before storing them in the database. Vice versa, timestamps read from the database are implicitly treated as originating in the *datastore time zone context*. An exception to this are timestamps mapped to the database using the "f" (floating) mode flag in the <map> tag (see 11.34.41.1) – these are always stored as-is (which can make sense if the actual time zone is stored along with each timestamp in a separate database field using the "zonename" or " zoneoffset_xxx" mapping types).
  The *datastore time zone context* to be used for a datastore is specified using the <datatimezone> tag (see 11.34.31). For compatibility with pre-3.1 configuration files <timeutc> (see 11.34.30) is still supported but no longer recommended – use <datatimezone> (see 11.34.31) instead: timeutc=true is equivalent with datatimezone=UTC and timeutc=false is equivalent with datatimezone=SYSTEM).

- **User time zone context.** Each user of a SyncML application thinks of his or her calendar entries in the context of a time zone. Calendar applications and web sites use this time zone to display timestamps, and usually input of new calendar entries is meant in that time zone as well. This general fact gains technical relevance with SyncML devices that are not capable of receiving and sending timestamps in a time-zone independent way (usually UTC). When communicating with such a device, the SyncML application must know implicitly in what timezone context transmitted timestamp values are meant. By default (and generally in Synthesis SyncML engines before 3.1), this *user time zone context* is the same as the local time zone of the operating system. While this is usually correct for single user mobile devices, it might not be sufficient for a multi-user server. To allow individual time zone context per user, Synthesis SyncML Engine 3.1 adds the <usertimezone> configuration tag (see 11.22) and especially the SETUSERTIMEZONE() (see 6.14.4) script function, which allows setting the *user time zone context* based on user-level information retrieved at login (see <logininitscript> and <loginfinishscript> in 11.33 or <logincheckscript> in 12.17).

- **Item time zone context.** For each item (such as a vCalendar item) processed by the SyncML engine, the *item time zone context* is initialized with the *user time zone context*, which means that timestamp data which has no timezone information attached is treated as related to the *user time zone context*. However, if a item carries time zone information (for example vCalendar TZ/DAYLIGHT), this modifies the *item time zone context* accordingly, and any timestamp found in the item which does not have its own specific time zone will be subsequently treated in the *item time zone context*. Vice versa, the *item time zone context* might be used (depending on rules defined by the content format) to represent timestamps when generating items like vCalendar.

- **Field level time zone context.** Each timestamp which "travels" from SyncML end to the database end trough the SyncML engine has its own time zone context associated. When reading a item from SyncML content formats like vCalendar, the *field level time zone context* is either read as part of the timestamp string representation (e.g. TZID parameter in iCalendar) or copied from the *item time zone context*. When writing the timestamp to the database, it is usually converted to the *datastore time zone context* (if <map> mode flag "f" is not used, see 11.34.41.1) Vice versa, timestamps read from the database receive either a individual time zone from a "zonename" <map> (see 11.34.41.1) or are put into *datastore time zone context*. Before data is converted to SyncML content formats like vCalendar, timestamps are converted to *user time zone context* (except if <userzoneoutput> is set to false, see 11.34.32). The built-in script language offers various built-in functions to access and manipulate the *field level time zone context*.

5.3 Time zone specifications

Time zones can be specified in different ways:

- **By name:** The SyncML engine has a built-in list of world-wide time zones which can be referenced by name. Well known timezone name examples are "UTC", "PST", "CET". If a timezone has daylight savings, it can be referenced either by the standard time zone name, the daylight savings zone name or a combination of both: "PST", "PDT" and "PST/PDT" all reference Pacific time; "CET", "CEST" and "CET/CEST" all reference Central European Time. Some zones have more descriptive aliases like "Pacific", some have variants with differing daylight savings rules like "Pacific_Mexico" etc. The TIMEZONE script function (see 6.14.4) returns the time zone name of a given timestamp.
  For a complete list of built-in time zones see chapter 17.
  A number of special time zone names are supported as follows:
  - **SYSTEM** – means the local time zone of the operating system (eventually overridden with <systemtimezone> see 8.14).
  - **DATE** – means a floating date-only value.
  - **FLOATING** – means that the timestamp is not related to any time zone in particular.
  - **USERTIMEZONE** – can be used in script functions like SETTIMEZONE (see 6.14.4) to apply the *user time zone context* active for the current user in the current sync session.
- **By VTIMEZONE specification:** New timezones can be added to the built-in list using the VTIMEZONE format (as defined in iCalendar, RFC 2445). This can be done statically in the configuration (<definetimezone>, see 8.15) or dynamically in scripts using script functions like SETTIMEZONE (see 6.14.4) which can accept VTIMEZONE input to specify a time zone. New timezones are also created implicitly when receiving vCalendar items containing VTIMEZONE specifications that do not match one of the already defined time zones. The VTIMEZONE script function (see 6.14.4) returns the time zone of a given timestamp as a VTIMEZONE record.
- **By TZ/DAYLIGHT specification:** New timezones are also added implicitly when receiving TZ/DAYLIGHT properties in vCalendar 1.0 items (using the special "tz" and "daylight" conversion modes, see 10.3.4).
- **As numeric offset:** This is generally not recommended, as most time zones do not have the samm offset all year long but change between standard and daylight savings time, so a numeric offset only applies for a single specific time stamp value and cannot be used generally for other timestamps in the same zone.
  Numeric time zone offsets are accepted as part of timestamp formats like ISO8601 or RFC822 (email time stamps), or as input to script functions like SETTIMEZONE (see 6.14.4) when used with numeric arguments.

6. Scripting Language

The PRO versions of the Synthesis SyncML engine feature a built-in, highly efficient, C-style syntax scripting language that extends flexibility in adapting to even exotic database layouts far beyond what is possible with the standard version. It allows for example to code value translations that are more complex than simple 1:1 translations (which can be done with <enum>s, see 10.3.5).

This chapter describes the script language in general. Scripts can be defined at many points within the configuration to customize many aspects of handling data, database access, data matching algorithms etc. These places where a script can be used within the configuration is described together with the related configuration section.

Note that this chapter assumes basic knowledge of C or a C-like syntax language (for example JavaScript).

6.1 What can be scripted?

There are various possibilities to use scripts to customize processing of a synchronisation operation (a so called sync session). For each possibility, a "hook" exists to insert your custom code in the form of a <*xxxxx*script> configuration tag, where *xxxxx* describes the action or process that can be customized. These scripts are executed in various different *contexts* (see 6.8 for details). This is important to understand as every context has it's own scope (local variables, lifetime, context script functions that can be accessed).

To get an overview of what scripts exist in what contexts, please refer to the "SySync_script_call_flow.pdf" diagram (separate PDF document).

6.2 Embedding script source code in XML

All script source is embedded in the configuration file as text between XML tags like:

```
<testscript>
  // this is a script
  integer x;
</testscript>
```

However, as scripts often contain greater-than and less-than signs (< and >) and maybe ampersands (&) which have a special meaning in XML, we strongly recommend to use the XML CDATA bracket to enclose scripts, as follows:

```
<testscript><![CDATA[
  // this is a script that can safely contain <, > and &
  integer x,y,z;
  x = y > z && z<100;
]]></testscript>
```

6.3 Comments

The script engine supports both forms of ANSI-C style comments, that is

- Any sequence of characters starting with /* and ending with */
- Any sequence of characters starting with // and ending with a line end

6.4 Statements and Statement Blocks

A statement is either a simple statement terminated with a semicolon (;) or it is a statement block. Statement blocks are multiple statements enclosed in { and }. Note that empty statements are allowed (consisting of a semicolon only)

```
// simple statement
a = b;

// statement block
{
  a = b; // first simple statement
  c = d; // another simple statement
}

// empty statement
;
```

6.5 Identifiers

Identifiers identify language keywords (such as IF, ELSE etc.), symbolic constants, variables and functions. Identifiers always start with an alphabethic character, and otherwise consists of any number of alphanumeric characters or underscores (_).

Note that identifiers in scripts are not case sensitive (unlike in C)!

6.6 Data types

The built-in script language uses exactly the same base datatypes that are also available for fields in a <fieldlist> definition (see 10.2 for details). All datatypes are either integer or string based. There is no floating point data type. Note also that despite the similarity with C, the basic types known from C are not available (neither char nor int).

6.7 Constants/Literals

There are three basic types of literal constants:

- **Integer constants**: an optional minus sign "-" followed by digits "0".."9". From version 3.1 onwards C-style hexadecimal (0x...) integer syntax is supported (but not octal (0...) integer syntax).
- **Quoted string constants**: Any text surrounded by double quotes. Within the quotes, only "\" (backslash) has a special meaning: it causes the following character to be added to the

string unprocessed, for example a doublequote or the backslash itself. Note the following special escape sequences:
  - "\n" stands for a line end
  - "\t" stands for a tab
  - "\xNN (where NN is a two-digit hexadecimal number) stands for the character with the ASCII-Code equivalent to NN.
- **Date and time constants** in ISO8601 basic format such as 20030604T164922 or 20030604144922Z (see ISO8601 specs if you need more details). Note that if the ISO8601 constant expression contains + or -, it must be quoted like a string constant.

There are also the following special **symbolic constant** values:
- **TRUE** (synonymous to a constant integer of 1)
- **FALSE** (synonymous to a constant integer of 0)
- **EMPTY**: This is a special "value" all variables can have and means that the variable has no value.
- **UNASSIGNED**: This is almost the same as EMPTY, but for data item fields, it has the additional meaning of "this value has not been assigned, not even with an EMPTY value". This is useful to distinguish values that were transmitted from the remote SyncML device with an empty value (==EMPTY) from values that were not transmitted at all (==UNASSIGNED).

6.8 Script contexts

All scripts run in a specific *script context*. For example, there is a "login context", a "session context", "datatype contexts" etc. A script context can be thought of as an execution environment that is (mostly) isolated from other script contexts and has the following properties:
- One or several scripts that belong to the script context and are executed within the environment that this context represents. Which scripts belong to what context is described where the script's tag in other chapters of this reference manual.
- A lifetime. For example, the "session context"s life time is the entire sync session, whereas a database mapping script context's life is only as long as database access takes place.
- *Context variables* (these are defined by the scripts, see 6.9 below). These are values that are local to the script context and can be accessed only by scripts in the same context.
- None, one or two (but not more) *data items*. A data item represents the data in an object being synced by the SyncML engine, such as a vCard or vCalendar. However, the data item is *not* the vCard itself, but its internal representation which consists of a list of fields as defined by a <fieldlist> (see 10.1).
- A number of *context built-in functions*. These are built-in functions that do make sense only within this special script context and are not available in other script contexts (as opposed to *global built-in functions* (see 6.13.2) which are available in all scripts).

6.9 Variables

Variables in scripts are technically the same internal objects as fields in a <fieldlist> (see 10.2). Variables can be assigned values (using the = assignment operator) and values stored in variables can be used in *expressions* (see 6.10)

Within scripts, there are three basically different kinds of variables - where the difference is in the way they come into existence (scope) and the syntax used to access them.

6.9.1 Context Variables

These are variables *declared* in one of the scripts that belong to a script context. Note that all variables must be *declared* before being used. Variables can be of any of the basic datatypes (see 6.6) or dynamic arrays thereof (always with empty square brackets, specification of a size is neither needed nor allowed):

```
// declaration of simple variables
integer mynumber,yournumber; // two integers
string s; // a dynamic string
telephone tel; /* also a dynamic string, but with tele-
phone-number comparison rules */
timestamp t; // a timestamp

// declaration of dynamic arrays
string mynamelist[]; // an array of strings
integer l[]; // an array of integers
```

Note that all *context variables* declared in *any* of the scripts belonging to the same context are accessible in *all* scripts belonging to this context regardless of when (or whether at all) the scripts containing the declarations are executed. Therefore, it is sufficient to declare context variables in *one* script belonging to a context (altough redeclaration is allowed as long as the type of a variable is the same in all declarations).
To explicitly access context variables in script expressions when there are *field variables* having the same name the name can be qualified with a "local." prefix, but this is only for documentation purposes (as a *context variable* always override *field variables* with the same name):

```
// accessing a context variable
integer a,b; // two integers

// normal access
a = b;

// with qualifier prefix for documentation
local.a = local.b;
```

6.9.2 Local variables of a user-defined function

Variables declared in user-defined functions (see 6.13) are local to the function, and not to the script context where the function is called. Otherwise, local variables in user-defined functions are declared and used like script context variables.

6.9.3 Field variables

In script contexts that have associated *data items* (see 6.8 above) the fields of the data items can be accessed as follows:

- The field of the so called *target* data item (also called *new* or *winning* item, depending on the context) can be accessed by the field name alone (if there is no *context variable* with the same name) or with any of the (synonymous) prefixes "new.", "winning.", "target.".
- The field of the so called *reference* data item (also called *old* or *loosing* item, depending on the context) can only be accessed by using one of the (synonymous) prefixes "old.", "loosing." or "reference.".

```
/* assuming there is no context variable DTSTART we can
access the DTSTART of a vCalendar like: */

t = DTSTART;

/* to make sure we get the field, and not a context vari-
able: */

t = TARGET.DTSTART;

/* if there are two data items involved (for example when
comparing or merging items: */

if (NEW.DTSTART<>OLD.DTSTART) { /* do something */ }
```

Note that in some contexts, it might be that data items (one or both) cannot be written, such that assigning values to field variables is not allowed (for example in the <comparescript>).

6.9.4 Array variable references

Array variables are always one-dimensional, dynamically sized arrays of simple variables. The elements can be accessed by a zero-based index expression in square brackets like:

```
// declaration
integer a,myarray[];

// accessing array elements
a = myarray[0]; // first element
b = myarray[7]; // 8th element
```

In addition there is a special form of array index that can be used to access fields in a field list (see 10.1) by index instead of by name. This special form of array index starts with a + sign as the first character after the opening [as shown in the following example:

```
<!-- a sample field list -->
<fieldlist>
  <field name="NAME" type="string"/>
  <field name="TEL_1" type="telephone"/>
  <field name="TEL_2" type="telephone"/>
  <field name="TEL_3" type="telephone"/>
</fieldlist>
```

```
/* sample script to access the telephone numbers by index
instead of by name */

integer i;
telephone a,b,c;

a = TEL_1[+0]; // this is the same as: a=TEL_1
b = TEL_1[+1]; // this is the same as: b=TEL_2
c = TEL_1[+2]; // this is the same as: c=TEL_3
```

As this special form is dependent on the field order in the field list, it's use is only recommended when this can be guaranteed.

6.10 Expressions

Expressions are built as follows (much like C, but not all operators of C available):

- An *expression* consists of a single *term* or multiple *terms* linked together with one of the *operators*.
- A *term* is either an *expression* enclosed in parantheses, a *constant* (see 6.7), a *variable reference* (see 6.9 and 6.9.4) or a *function call* (see 6.13). A *term* can be optionally preceeded by a *typecast*.
- A *typecast* is a type name enclosed in parantheses, like: (integer)a. It's effect is that the term it preceedes is converted to the type specified.
- *Operators* are the following, in the order of precedence:
  - Unary minus (-), unary logical NOT (!) and unary bitwise NOT (~).
  - Multiply (*), divide (/) and modulus (%)
  - Add for numbers or concatenate for strings (+) and subtract (-)
  - Shift left (<<) and shift right (>>)
  - Comparison operators (>, <, >=, <=, ==, !=)
  - Bitwise AND (&)
  - Bitwise XOR (^)
  - Bitwise OR (|)
  - Logical AND (&&)
  - Logical OR (||)

  Note that all operators except + need numeric (integer or timestamp) operands. If operands are not numeric, they will be implicitly converted.

6.11 Flow control

The script engine supports the following flow control mechanisms:

- **IF (conditional_expression) *statement***: *statement* is executed if *conditional_expression* returns non-zero result.
- **IF (*conditional_expression*) *statement1* ELSE *statement2***: *statement1* is executed if *conditional_expression* returns non-zero result, otherwise, *statement2* is executed.
- **IF (*conditional_expression1*) *statement1* ELSE IF (*conditional_expression2*) *statement2*.... ELSE *statement***: Chained if-else; the last *statement* is only executed if none of the previous IF conditions were true.
- **LOOP *statement*.** This is a general loop mechanism (there is also a *while* statement, but no *for*/*do* as in C). It causes *statement* (which is normally a *statement block*) to be repeated forever. Therefore, the *statement (block)* **must** contain at least one **BREAK** statement to exit the loop.

It can also contain **CONTINUE** statements to jump to the beginning of the loop. To prevent the SyncML engine to hang in case there is an infinite LOOP in a script, any LOOP that takes more than the number of seconds defined in <looptimeout> (see 8.10.3, default is 5 seconds) aborts the script execution with an error.

- **WHILE (*conditional_expression*) *statement*.** New in 3.1: This causes *statement* (which is normally a *statement block*) to be repeated as long as *conditional_expression* is true. The *statement (block)* may contain **BREAK** statements to exit the while loop or **CONTINUE** statements to jump to the beginning of the while loop. To prevent the SyncML engine to hang in case there is an infinite WHILE in a script, any WHILE that takes more than the number of seconds defined in <looptimeout> (see 8.10.3, default is 5 seconds) aborts the script execution with an error.
- **RETURN** or **RETURN *expression*.** This statement can be used to terminate the script and optionally return the specified *expression* to the caller (whether this makes sense, depends on the script's purpose and is described in the other configuration chapters).

Some examples:

```
integer a,b;
string s;
a=5;
b=2;

// exit script if a is equal b
if (a==b) return false;

// loop 5 times
a=5;
loop {
  if (a<=0) break;
  a=a-1;
}

// chained if/else
if (a==1) s="one";
else if (a==2) s="two";
else if (a==3) s="three";
else s="out of range";
```

6.12 Macros

Since version 2.9.8.12, the scripting engine also supports macros. Macros are texts (usually consisting of one or more script statements) that are defined once in the <scripting> section of the config and can then be inserted into any script by name.
The sample config makes use of macros to avoid duplication of some lengthy scripts required in serveral different email datatypes.

6.12.1 Defining Macros

Macros are defined in the <scripting> section using the <macro name="macroname"> tag. The *macroname* must be unique among all macros and is used to reference the macros in scripts. Like

with scripts, using the XML <![CDATA[…]]> bracket around the actual macro text prevents problems with special XML characters like <,> and & and is strongly recommended.

6.12.2 Marco arguments

New in 3.4: Macros can have up to 9 arguments. Use $1, $2 … $9 in the macro definiton as placeholders where arguments should be substituted.

6.12.3 Using Macros

Macros can be used (inserted) in scripts simply by a dollar sign followed by the macro name. See the following example:

```
<scripting>
   <macro name="MYMACRO"><![CDATA[
     integer a,b,c;
     a = b;
     c = b*5;
  ]]></macro>
</scripting>

… other config tags …

<initscript>
  $MYMACRO
  d = c + 1;
</initscript>
```

This is equivalent to the following script:

```
<initscript>
  integer a,b,c;
  a = b;
  c = b*5;
  d = c + 1;
</initscript>
```

If the macro definition contains placeholders for arguments ($1,$2...$9), see 6.12.2, the macro invocation should provide arguments in parantheses:

```
<scripting>
   <macro name="MYMACRO"><![CDATA[
     f = b*$1; // macro argument
  ]]></macro>
</scripting>

… other config tags …

<initscript>
  integer b,f;
  $MYMACRO(7) // macro call with one argument
</initscript>
```

6.13 Functions

Functions are called using their name, followed by a comma-separated list of parameters in parantheses. Functions that have no parameters are called just with empty parantheses. Some functions return a value, which can be used in expressions:

```
integer l;
string s;
timestamp t;

// function that returns a value and has one parameter
l = length(s);

// function that returns a value and has no parameter
t = now();

// function with more than one parameter
l = find(s,"x",0);
```

6.13.1 User defined Functions

User-defined functions must be defined in the <scripting> section of the config file using a <function> tag for every function to be defined.

The *function definition* starts with the type of the return value (if the function has a return value), followed by the function name, followed by a *parameter list* in parantheses, followed by a *statement block* that contains the function's code. Functions can return a value to the caller using the RETURN statement.

The *parameter list* can be empty for functions without parameters, or contains one or multiple comma separated *parameter declarations*.

A *parameter declaration* is like a *variable declaration* (see 6.9.1) and consists of a type name followed by the parameter name. This declares a parameter that is passed by value (which means that the value specified when calling the function is stored in a local variable of the function, which can be modified by the function code, but does not affect any variables of the caller).

Optionally, parameter passing can also be declared as "by reference" by preceeding the parameter name with an ampersand. This means that the caller of the function must specify a variable (rather than a constant or an expression) for the parameter. If the function code assigns a new value to the parameter, the caller's variable will be changed.

```
<scripting>

  <function><![CDATA[
    // function with one by-value parameter
    integer decremented(integer a)
    {
      // this only changes the local variable "a".
      a=a-1;
```

```
      // changed local "a" is returned as function result
      return a;
    }
  ]]></function>

  <function><![CDATA[
    // function with one by-reference parameter
    decrement(integer &a)
    {
      // this changes the caller's variable that
      // was passed for the parameter "a"
      a=a-1;
    }
  ]]></function>

  <function><![CDATA[
    // function with two parameters
    string rightmost(string s, integer n)
    {
      // call built-in function substr to extract
      // n rightmost characters from s
      return substr(s,length(s)-n,n);
    }
  ]]></function>

</scripting>
```

6.13.2 Built-in Functions

A number of useful functions are built into the script engine. They can be called exactly like user-defined functions.
There are two types of built-in functions

- **Global built-in functions**: These are generally useful functions that are available in all scripts. See 6.14 for a list.
- **context built**-in functions: These are special functions that are only available in a specific context. These functions are described along with the scripts where they are valid. See section 0 for a list of all function names (global and context-specific).

6.14 Global built-in Function Reference

This section lists script functions that are available in all script contexts. Note that there are many more script functions available in some specific script contexts – these are described with the <xxxxscript> tag they apply to. See section 0 for a list of all function names (global and context-specific).

6.14.1 String functions

integer LENGTH(expression): returns the length of *expression*. The result is the number of bytes in the string representation of *expression*. Note that due to UTF-8 encoding (multi-byte representation of characters outside 7bit ASCII range) the number of characters might be less than the number of bytes. **Note that (unlike stated in earlier versions of this document) LENGTH cannot be used to determine the size of an array. Use SIZE() instead.**

integer SIZE(var): returns the size of *var*. If *var*is an array, the result is the number of elements in the array. If *var* is a single value (a non-array or an array element), the result is the number of bytes in the string representation of *var*. Note that due to UTF-8 encoding (multi-byte representation of characters outside 7bit ASCII range) the number of characters might be less than the number of bytes. **Note that SIZE cannot be used to determine the length of a string expression. User LENGTH instead.**

string LOWERCASE(string str): returns an all-lowercase version of *str*.

string UPPERCASE(string str): returns an all-uppercase version of *str*.

string NORMALIZED(string str): returns normalized version of *str*. Normalized has only a meaning for string-based types such as simple string (New in 3.1: normalized form is spaces trimmed off at start and end) telephone number (normalized form is number without all separator and spacing characters) or multiline (normalized form is text without leading or trailing empty lines and spaces).

integer FIND(string *str*, string *pattern*, integer *start*): searches for first occurrence of *pattern* in *str*, starting the search at *start* (0=first character). Returns UNASSIGNED if *pattern* not found, otherwise returns the position of *pattern* in *str* (0=at the beginning)

integer RFIND(string *str*, string *pattern*, integer *start*): searches backwards for first occurrence of *pattern* in *str* before *start* (0=first character). Returns UNASSIGNED if *pattern* not, otherwise returns the position of *pattern* in *str* (0=at the beginning)

string SUBSTR(string str, integer start, integer count): returns substring of *str* which starts at position *start* in *str* and has at most *count* characters.

string NUMFORMAT(integer *num*, integer *digits* [,string *filler*=" " [, string *opts*=""]]): formats the number specified in *num* as a string with *digits* number of digits or spaces. If *digits* is negative, the output is left justified, otherwise it is right justified. If *filler* is specified empty, no padding will occur, otherwise *filler* is used to pad unused space to make the result *digit* characters long. *Opts* can be set to '+' to force a positive sign to be shown, or to ' ' to force a space to be shown for positive numbers. With no *opts*, negative numbers are prefixed by a '-', positive number have no prefix. If *opts*contains 'x', the number is formatted in hexadecimal.

string EXPLODE(string *glue*, &*parts*[]): returns the elements of the *parts* array passed concatenated as a string, elements separated by *glue*.

integer PARSEEMAILSPEC(string *emailspec*, string &*name*, string &*email*): Parses *emailspec* as a RFC2822 email address and puts the descriptive name into *name* and the plain email address into *email*.

string MAKEEMAILSPEC(string *name*, string *email*): Makes a RFC2822 email address out of *name* and *email* (quoting *name* if required).

6.14.2 Regular Expression functions

Note that regular expression support may not be compiled into all versions of the Synthesis SyncML engine, and therefore the following functions may not be available.
The *pattern* string in the following functions can be of the "/rrrr/oo" form, where rrrr is the regular expression and oo are one or multiple options (i,m,s,x,U supported). Alternatively, a regex can be specified directly without delimiters, as long as it does not start with a /.
Please refer to general documentation about regular expressions for information about how to work with regular expressions. The regular expression engine used in Synthesis SyncML engine is PCRE which is documented in the internet at http://www.pcre.org/.

integer REGEX_FIND(string *subject*, string *pattern* [, integer *startat*]): searches for first occurrence of *pattern* in *subject*, starting the search at *startat* (0=first character). Returns UNASSIGNED if no match is found, otherwise returns the position of where *pattern* matches in *subject* (0=at the beginning).

integer REGEX_MATCH(string *subject*, string *pattern*, integer *startat*, string &*matches[]*): searches for the first match of *pattern* within *subject*, starting the search at *startat* (0=first character). Returns UNASSIGNED if no match is found, otherwise returns the position of the match. Additionally, the *matches* array will contain the entire matched string in its first element, and paranthesized subpattern matches within *pattern* in the subsequent elements. If a non-array is passed for *matches*, that variable is assigned either the entire matched string (if *pattern* does not specify subpatterns) or the first subpattern (if *pattern* does specify at least one subpattern).

integer REGEX_SPLIT(string *subject*, string *separatorpattern*, string &*elements* [, boolean *emptyElements*]): Splits *subject* into string elements and store them in *elements*. The string is split where *separatorpattern* matches in *subject*. If *emptyElements* is set to ture, elements consisting of nothing (i.e. two separators in succession) will be stored as such in *elements*, otherwise empty elements will be ignored.

string REGEX_REPLACE(string *subject*, string *pattern*, string *replacement* [, integer *startat* [, integer *repeat*]]): Replaces matches of *pattern* within *subject* with *replacement*. Pattern matching starts at *startat* in *subject* (default = 0) and continues *repeat* times (default = 0 which means all occurrences are replaced).

6.14.3 Date and Time functions

timestamp NOW(): returns the current time of the SyncML server in UTC. Note that the database time might not be fully in sync with this. Use DBNOW() if you need accurate database time.

timestamp DBNOW(): returns database's current time in UTC. Note that calling this function will cause a database access, so use it with care to avoid performance degradation.

timestamp SYSTEMNOW(): returns the current time of the SyncML server in the system's local time. Note: while this is usually a shortcut for CONVERTTOZONE(NOW(),"SYSTEM"), there might be implementations that do not have time zone support at all. In these implementations, NOW() would not return a proper value, but SYSTEMNOW() will.

SLEEPMS(integer milliseconds): suspends the current thread for the given number of milliseconds. Accuray depends on the platform, not all platforms support millisecond resolution for sleeping, so actual time might differ.

timestamp DATEONLY(timestamp *ts*): returns the date part of *ts* as a date-only value (which corresponds to a floating timestamp with a 0:00:00 time part). However, a date-only value has a special flag set to differentiate it from a timestamp, which is used for example when rendering time/date values in ISO8601 (for example, in the "autodate" conversion mode for vCalendar items, see 10.3.4).

timestamp TIMEONLY(timestamp *ts*): returns the time part of *ts* as a time-only value (which is the number of time units since midnight).

integer ISDATEONLY(timestamp *ts*); returns true (1) if *ts*is a date-only value.

timestamp DURATION(timestamp *ts*): returns the timestamp as a duration. Duration timestamps are rendered in the ISO8601 duration format. The timestamp's internal value will not be affected by the conversion to duration format.

timestamp POINTINTIME(timestamp *ts*): returns the timestamp as a point in time. This is useful to convert duration timestamps back into timestamps that represent a point in absolute time, and are rendered in ISO8601 date/time format. The timestamp's internal value will not be affected by the conversion to a point in time.

integer ISDURATION(timestamp *ts*); returns true (1) if *ts*is a duration value.

integer WEEKDAY(timestamp *ts*): returns weekday of day represented by *ts* (0=sunday, 1=monday ... 6=saturday).

integer SECONDS(timestamp *ts*): returns number of seconds corresponding to *ts* (which makes most sense if *ts* is a difference between two timestamps, that is, a duration).

integer MILLISECONDS(timestamp *ts*): returns number of milliseconds corresponding to *ts* (which makes most sense if *ts* is a difference between two timestamps, that is, a duration).

integer TIMEUNITS(integer *seconds*): returns number of time units (normally milliseconds, but can be another unit depending on platform capabilities) corresponding to the specified number of seconds. Time units can be added or subtracted from time stamps.

integer DAYUNITS(integer *days*): returns number of internal time units (normally milliseconds, but can be another unit depending on platform capabilities) corresponding to the specified number of days. Time units can be added or subtracted from time stamps.

integer MONTHDAYS(timestamp *date*): returns number of days of the month *date* is in.

integer ALLDAYCOUNT(timestamp *start*, timestamp *end* [, boolean *checkinusercontext* [, boolean *onlyutcinusercontext*]]): This functions examines the *start* and *end* timestamps to check if these represent an all-day event, and if so, how many days it spans. This function is designed to operate on vCalendar 1.0 and iCalendar 2.0 DTSTART and DTEND values, and takes into account that in vCalendar 1.0 All-day events cannot be represented as such, but just as events starting at midnight, and ending at next midnight or 23:59. If the result is 0, *start* and *end* do not specify an all-day event, otherwise, the result is the number of days. The input timestamps must be in the context in which they are to be checked for midnight, 23:59:xx etc. except if *checkutcinusercontext* ist set. If so all non-floating timestamps (*onlyutcinusercontext* = false) or all

UTC timestamps (*onlyutcinusercontext* = true) will be converted to user time zone before checking for all day boundaries.

MAKEALLDAY(timestamp &*start*, timestamp &*end* [,integer *days*]): This function is designed to adjust *start* and *end* suitable for using it with the "autodate" and "autoenddate" conversion modes (see 10.3.4). If *days* is omitted or set to <=0, the difference between *end* and *start* determines the number of days. If *days* is set to >0, the input value of *end* is ignored, and an all-day of *days* days length is created starting at *start*. On input, timestamps must already represent local day times. On output, the timestamps are made floating.

timestamp RECURRENCE_DATE(timestamp *start*, string *rr_freq*, integer *interval*, integer *fmask*, integer *lmask*, boolean *occurrencecount*, integer *count*): Returns the date of the *count*th iteration of a recurrence rules. If *occurrencecount* is true, the *count*th occurrence is calculated, otherwise the *count*th repetition of the entire rule interval (the latter is relevant for vCalendar 1.0 RRULE #n repetition value). See 10.6 for a description of the *start*, *rr_freq*, *interval*, *fmask* and *lmask* parameters.

integer RECURRENCE_COUNT(timestamp *start*, string *rr_freq*, integer *interval*, integer *fmask*, integer *lmask*, boolean *occurrencecount*, timestamp *occurrence*): Returns the count of a given *occurrence* date relative to the beginning of a recurrence rule. If *occurrencecount* is true, the count returned is the occurrence count, otherwise it is the repetition count of the entire rule interval (the latter is relevant for vCalendar 1.0 RRULE #n repetition value). See 10.6 for a description of the *start*, *rr_freq*, *interval*, *fmask* and *lmask* parameters. If no recurrence count can be calculated for *occurrence*, the function returns UNASSIGNED.

string MAKE_RRULE(boolean *rrule2*, string *rr_freq*, integer *interval*, integer *fmask*, integer *lmask*, timestamp *until*): Creates a RRULE from the RRULE block parameters specified (see 10.6 for a description of the *start*, *rr_freq*, *interval*, *fmask, lmask* and *until* parameters). If *rrule2* is set to true, a iCalendar 2.0 style RRULE will be created, otherwise, a vCalendar 1.0 RRULE is returned.

boolean PARSE_RRULE(boolean *rrule2*, string *rrule*, timestamp *start*, string &*rr_freq*, integer &*interval*, integer &*fmask*, integer &*lmask*, timestamp &*until*): Parses a RRULE string into RRULE block parameters specified (see 10.6 for a description of the *rr_freq*, *interval*, *fmask*, *lmask* and *until* parameters). If *rrule2* is set to true, a iCalendar 2.0 style RRULE is expected in *rrule*, otherwise, a vCalendar 1.0 RRULE is expected. The *start* parameter must be set to the starting point of the recurring calendar entry. The function returns true if *rrule* could be successfully parsed, false otherwise.

integer ISRELATIVE(timestamo *ts*): No longer supported in 3.1. Usually ISFLOATING() provides the same functionality, but please read 5.1 about the general changes in the way timestamps are represented in 3.1 vs. 3.0.

SETRELATIVE(timestamp &*ts*): No longer supported in 3.1. Usually SETFLOATING() provides the same functionality, but please read 5.1 about the general changes in the way timestamps are represented in 3.1 vs. 3.0.

timestamp LOCALIZEDASUTC(timestamp *ts*): No longer supported in 3.1. The functionality itself is no longer needed as the new 3.1 way of representing timestamps (see 5.1) makes it obsolete.

integer LOCALZONEOFFSET():No longer supported in 3.1. Replacements are usually USERTIMEZONE() or TIMEZONE(SYSTEMNOW()).

timestamp RELATIVEASUTC(timestamp *ts*): No longer supported in 3.1. The functionality itself is no longer needed as the new 3.1 way of representing timestamps (see 5.1) makes it obsolete.

SETZONEOFFSET(timestamp &*ts*, integer *zoneoffset*): No longer supported in 3.1. Usually SETTIMEZONE() provides the same functionality, but please read 5.1 about the general changes in the way timestamps are represented in 3.1 vs. 3.0..

timestamp UTCASRELATIVE(timestamp *ts*): No longer supported in 3.1. The functionality itself is no longer needed as the new 3.1 way of representing timestamps (see 5.1) makes it obsolete.

6.14.4 Time zone related functions

Some of the time zone related functions have a *timezonespec* parameter. This parameter specifies a time zone in one of the following ways:

- when a timestamp type is passed, the time zone is copied from the specified timestamp.
- when a integer type is passed, the time zone is set to the given number of seconds east of GMT/UTC
- when an empty value or the string value "FLOATING" is passed, the result is no time zone (i.e floating timestamp).
- when a string value of "USERTIMEZONE" is passed, the current user time zone (see 5.2) is used.
- when a string value of "SYSTEM" is passed, the current system time zone is used.
- when a string value beginning with "BEGIN:VTIMEZONE" is passed, it is paresed as a timezone specification in vTIMEZONE format.
- when the string value names one of the defined zone names (see list in chapter 17), the corresponding zone is used.
- finally, the string can specify a time zone offset specified in ISO8601 format.

Script functions that return a *timezonestring* either return:

- the name of the time zone (see list in chapter 17) as string
- an empty string for floating timestamps
- a time zone offset in ISO8601 format for fixed UTC offset time zones

timezonestring TIMEZONE(timestamp *atime*): returns the time zone associated with *atime*.

string VTIMEZONE(timestamp *atime*): returns the time zone associated with *atime* formatted as a vTIMEZONE entry.

SETTIMEZONE(timestamp &*atime*, timezonespec *zone*): sets the time zone of *atime* to the specified zone. Note that the local time value of *atime* does not change (which means that the absolute time value changes by the difference of the old and new time zone).

integer ISFLOATING(timestamp *ts*); returns true (1) if *ts* is a floating timestamp (i.e. a timestamp not associated with a time zone).

SETFLOATING(timestamp &*atime*): this is a shortcut for SETTIMEZONE(*atime*, "FLOATING"), and removes time zone information from *atime*, thus making *atime* a floating timestamp (not associated to any time zone).

string USERTIMEZONE(): returns the current user time zone (see 5.2), as set by SETUSERTIMEZONE() or <usertimezone>. The default is "SYSTEM".

SETUSERTIMEZONE(timezonespec *tz*): sets the user time zone (the time zone context used to evaluate local time specifications which do not include an originating time zone information, see 5.2). The default user time zone can be configured using <usertimezone> (see 11.22) and defaults to "SYSTEM".

timestamp CONVERTTOZONE(timestamp *atime*, timezonespec *zone* [,boolean *doUnfloat*]): returns *atime* converted to the specified *zone*. This means that the absolute value (UTC time) of the result will be the same as that of *atime*, but represented in a different time zone and therefore having a different local time value. A special case are floating time stamps – these cannot be actually converted to a different time zone, as they have no zone to begin with. If *doUnfloat* is set to true, floating time stamps will be made local time of the specified zone, without changing their time value. Otherwise, floating timestamps will be returned as-is.

timestamp CONVERTTOUSERZONE(timestamp *atime* [,boolean *doUnfloat*]): this is a shortcut for CONVERTTOZONE(*atime*, "FLOATING", *doUnfloat*), and returns *atime* converted to the current user time zone.

integer ZONEOFFSET(timestamp *ts*): returns offset in number of seconds, east of UTC/Greenwich of *ts*. If *ts* is a floating timestamp, this function will return UNASSIGNED.

6.14.5 Debug log functions

DEBUGMESSAGE(string *message*): writes *message* to the debug log. Note that these messages are only shown in the log if the "hot" option in <debug> (see 8.11.2) is enabled.

DEBUGSHOWITEM(boolean *refItem*): In scripts that have access to a data item (a collection of fields as defined in a field list, see 6.9.3), this function can be called to dump the contents of the data item into the log file (if the debug options are set such that user data can be shown in logs at all, see "userdata" option in 8.11.2). For scripts that operate on two data items (for example: <comparescript>, see 10.5.12), setting *refItem* to true shows the reference item (depending on the context, also called "loosing" or "old" item) instead of the normal (sometimes called "winning") item.

DEBUGSHOWVARS(): This dumps a list of all local variables of the current script and their current values to the debug log.

integer GETDEBUGMASK(): returns the currently active debug options for the current sync session as 32 bit integer value representing flags for each possible debug option, see 8.11. This is useful to temporarily change the debug options in a script (see SETDEBUGOPTIONS and SETDEBUGMASK) and later restore the original options.

SETDEBUGMASK(integer *mask*): should be used only to restore a debug channel configuration previously saved from GETDEBUGMASK. To enable or disable debug channels, please use SETDEBUGOPTIONS.

SETDEBUGOPTIONS(string *optionname*, boolean *enable*): This is the equivalent of the <enable> (when *enable* is true) and <disable> (when *enable* is false) tags in the <debug> section, see 8.11.2. The same names that are valid for the "option" attribute in <enable> and <disable> can be used for *optionname*.

SETXMLTRANSLATE(boolean *enable*): This function allows switching on or off writing an XML transcript of the SyncML messages for the current session. This can be used for example in the <logininitscript> (see 11.33) to selectively switch on XML loggin based on device or user name. See <xmltranslate> in 8.11.14 for details about the files created.

SETMSGDUMP (boolean *enable*): This function allows switching on or off dumping SyncML messages to files for the current session. This can be used for example in the <logininitscript> (see 11.33) to selectively switch on SyncML message dumping based on device or user name. See <msgdump> in 8.11.15 for details about the files created.

6.14.6 Other functions

integer ABS(integer *value*): returns the absolute of *value*.

integer SIGN(integer *value*): returns the sign of *value*, that is, 0 if *value* is 0, 1 if *value*>0 and -1 if *value*<0.

integer RANDOM(integer *range* [, integer *seed*]): returns a random number between 0 and *range*-1. The optional *seed* can be specified to seed the random generator.

string SYNCMLVERS(): This function returns the SyncML version number (currently one of "1.0", "1.1" or "1.2") of the running session. This can be useful to implement version dependent behaviour.

ABORTSESSION(integer *statuscode*): aborts the current session and reports *statuscode* as the reason for aborting the session. Note that *statuscode* can be 0 to abort silently.

integer COMPARE(*value1*, *value2*): returns 0 if *value1* equals *value2*, -1 if *value1* < *value2*, 1 if *value1* > *value2* and -999 if values cannot be compared.

integer CONTAINS (&variable, *value* [, bool *caseinsensitive*]): returns 1 (true) if *value* is contained in *variable*. "Contain" means that *value* is a substring of *variable*, or in case *value* is an array,.it means that *value* is a substring of one of *variable*'s elements. *caseinsenstive* can be set to make the comparison case insensitive.

APPEND (&variable, *value*): appends *value* to the contents of *variable*. If *variable* is an array, appending means adding a new element. If *value* is an array, all elements of *value* will be appended to *variable* one by one.

boolean ISAVAILABLE(*field*): checks if *field* (which must be a field from the <fieldlist>, see 10.1, of the item processed in the current script's context) is explicitly available for the current sync. Explicitly available means that the remote's device information was received and contained a list of fields supported by the remote party. If this is the case, this function returns true or false. If a non-field (e.g. a script local variable) is specified for *field*, UNASSIGNED is returned. If a *field* is specified, but no explicit

availability is known from the remote, the function returns EMPTY. This function is useful in <incomingscript> and <outgoingscript> (see 10.5.9) to format data differently depending on what fields are supported – for example including some data from unsupported fields into the description text.

REQUESTMAXTIME(integer *maxtime*): this allows to set the max time in seconds the server should spend processing a request before it should return an answer to the client. See <requestmaxtime> in 11.3 for details.

REQUESTMINTIME(integer *mintime*): this allows to artificially delay server responses to be sent not earlier than *mintime* seconds after the request reached the server. See <requestmintime> for details (11.4).

FORCELOCALTIME(boolean *flagvalue*): this allows modifying the flag that is controlled by the <forcelocaltime> directive in <remoterule>, see 11.36.13.

FORCEUTC(boolean *flagvalue*): this allows modifying the flag that is controlled by the <forceutc> directive in <remoterule>, see 11.36.14.

SHOWCTCAPPROPERTIES(boolean *flagvalue*): this allows modifying the flag that is controlled by the <showctcapproperties> directive (see 11.25).

ENUMDEFAULTPROPPARAMS(boolean *flagvalue*): this allows modifying the flag that is controlled by the <enumdefaultpropparams> directive (see).

string LOCALURI(): returns the original URI used by the client to start the server session.

string REMOTERULENAME(): returns the name of the active <remoterule> (see 11.36) or EMPTY if no remoterule is active in the session. This can be used to implement device-specific behaviour.

SETREADONLY(integer *readonly*): If *readonly* is set to TRUE, the session will be read-only (clients cannot apply any changes to the server's database). See also per-datastore read-only option in 11.34.4 and per-datastore version of SETREADONLY in 11.34.21.

SETDEBUGLOG(integer *enabled*): Enables or disables the session log for this session.

SETLOG(integer *enabled*): Enables or disables logging this session's result in the log file/log table.

any_type SESSIONVAR(string *varname*): Returns the value of the session context variable *varname* (Session context variables can be declared for example in <sessioninitscript>, see 11.11). This allows to access context variables (see 6.9.1) of the session's context from any script (and not only from those running in session context). This allows using session context variables as a kind of global variables. Note that accessing variables this way is less efficient (access by name needs string search) than normal variable references (access through precompiled index). If *varname* does not exists, UNASSIGNED is returned.

SETSESSIONVAR(string *varname, value*): Assigns a new *value* to the session context variable *varname*.

integer SHELLEXECUTE(string *command*, string *params*, integer *backgroundflag*): Calls the operating system's shell to execute the *command* with the specified *params*. If *backgroundflag* is true, the shell process is started in background and SHELLEXECUTE immediately returns with result=0 (**Note that background execution might not be available on all platforms!**). Otherwise,

SHELLEXECUTE waits until the shell command completes and then returns the exit code of the *command* executed. The exit code is operating system specific. In case that the *command* could not be started at all, SHELLEXCUTE returns -1. **Note: using SHELLEXECUTE often makes the configuration file platform-dependent**. Therefore, using the "platform" attribute (see 4) is recommended in scripts using SHELLEXECUTE to make sure the script only runs on the right platform, or to provide multiple variants of the script for each platform.

SWAP(*variable1*, *variable2*): swaps the contents of *variable1* and *variable2*.

UPDATECLIENTINSLOWSYNC(boolean *flagvalue*): this allows modifying the flag that is controlled by the < updateclientinslowsync > directive in <remoterule>, see 11.36.6.

UPDATESERVEINSLOWSYNC(boolean *flagvalue*): this allows modifying the flag that is controlled by the <updateserverinslowsync> directive in <remoterule>, see 11.36.7.

TREATASLOCALTIME(boolean *flagvalue*): this allows modifying the flag that is controlled by the <treataslocaltime> directive in <remoterule>, see 11.36.15.

TREATASUTC(boolean *flagvalue*): this allows modifying the flag that is controlled by the <treatasutc> directive in <remoterule>, see 11.36.16.

string TYPENAME(*value*): returns the name of the type of *value*.

string ITEMDATATYPE (): returns the internal name of the datatype of the current item (only usable in context that are processing a data item at all). The datatype name is the name specified in the <datatype> tag, see 10.5

string ITEMTYPENAME (): returns the MIME type name of the datatype of the current item (only usable in context that are processing a data item at all). The datatype name is the name specified in the <typestring> tag, see 10.5.3 (or implicitly set by the basetype attribute of <datatype>, see 10.5)

string ITEMTYPEVERS (): returns the type version string of the datatype of the current item (only usable in context that are processing a data item at all). The datatype version is the name specified in the <versionstring> tag, see 10.5.3 (or implicitly set by the basetype attribute of <datatype>, see 10.5)

6.15 Debugging scripts

Scripts can be debugged by switching on the "*scripts*" option in the <debug> directive (see 8.11.2). This will cause that every script line processed to be shown in the debug log along with some information about the expressions evaluated and variables assigned (see "*expressions*" option in the <debug> directive for enabling more in-depth expression debugging). If you use HTML formatted logs, the script source will be colored gray for script lines that were skipped unexecuted due to flow control.
Note that using this debug option not only **can generate huge logfiles and degrades performance**, but also needs slightly more memory per sync session as the script engine must keep the script source code in memory (when debugging is off, scripts are stored in a compressed, tokenized form only).

When the "*exotic*" debug option is enabled as well, the script processing is logged in extensive detail – this is only recommended for hard core debugging.

The script engine is designed for efficiency, and is not meant to be a general-purpose programming language, and therefore there is no real debugger available. If your scripts get too large and complicated to be debugged and tested with the simple debug log feature, you should probably re-think your application design in general. Scripts are provided to add more flexibilty to adapt SyncML to your application, but not to implement things that should be done in the application itself.

If you feel that the adaption to your database exceeds what can be done reasonably with scripts, please consider using the **plugin API for database adapters** available in the PRO products. This allows you to separate all database access code into an external plugin project written in C, C++, Java or .net. See 14 for details.

7. Filters

There are three types of Filters that can be used:

- ***Inclusive* or *Temporary* Filter expressions:** This type of filter defines conditions for content to be transmitted to a remote party, that is, *included* into the *sync set* (the set of items that are being synced). However, a *inclusive* filter does *not exclude items from the sync set*. This means that if for instance the client already contains items (from a previous sync session) that do not pass an inclusive filter, they will not be deleted on the client (in contrast, with a dynamic or static *exclusive filter*, see below, these would be deleted).
  - In SyncML 1.0 and 1.1 terminology, these type of filters were called *Target Address Filters (TAF)*, and can be specified as part of the database path using a CGI syntax.
  - In SyncML 1.2 and later, these filters are called *Inclusive Filters* and are transmitted from client to server using the <Filter> and <FilterType> SyncML elements in the CGI syntax (see 7.4).
- ***Dynamic exclusive* Filter expressions**: These allow clients to request synchronizing only a subset of the database by specifying constraints.
  - In Synthesis SyncML servers *dynamic exclusive filters* can be specified similar to TAF in the database path. This is a Synthesis-specific option and not part of the SyncML standard. An events path could look like "./events?/fi(DTSTART>20030630T000000Z)" which would restrict the sync set to events starting after July 2003.
  - In SyncML 1.2, the <Filter> and <FilterType> SyncML elements are used to specify exclusive filters in the CGI syntax (see 7.4). Synthesis SyncML servers still support the SyncML 1.1 methods (TAF and /fi, even for SyncML 1.2).
- ***Static* Filter expressions used in the config file:** Filters are also used internally, for example to split a common "calendar" database into "events" and "tasks" or to implement visibility control for records based on a special database field or the type of device connected (as an example some devices cannot handle dates before a certain date, so these can be filtered out by setting a static filter. See the sample config files for examples.

The *inclusive* filters are simply applied just before sending data to the client – only data passing the filter is included, other data it is just ignored and not sent to the client.

The term *dynamic* is used with *exclusive* filters to specify that the filter might change between sync sessions and therefore some records which were filtered out (= *excluded*) in one session get visible in the next session and vice versa. *Dynamic* filters can put considerable load to the SyncML server as applying them might require the server to load all records from the database instead of the changed ones. However, if the *dynamic* filter is such that it can be translated to a SQL WHERE clause, the performance penalty is much smaller (see <dbcanfilter> in 12.20.9).

A *static* filter is a filter that is guaranteed not to change between sync sessions with a particluar device, such as filter specified in the server configuration or set depending on the device that is being synced. *Static* filters are much more efficient because only those records that have changed or added need to be filtered.

There is one important reason why the Synthesis SyncML engine supports filters in parallel with scripting (which might look like the same thing was implemented twice): The filter syntax is such (much simpler than script expression syntax) that the SyncML engine can translate most filter expressions directly to SQL WHERE clause expressions. This is a huge performance benefit, because this way, only needed data gets fetched from the database at all (while otherwise, as explained above for *dynamic* filters all records need to be fetched from the database only to be

checked against a filter. This might be needed for complex situations where a filter is not flexible enough (see <filterscript> in 10.5.10), but should be avoided whenever possible.

7.1 Test and Make-Pass modes

Filters are used by the SyncML engine in two modes:

- **Test Mode:** This is the normal mode, and means that a filter expression is applied to a data item, which gives a result of true (item passes) or false (item does not pass).
- **Make-Pass Mode**: Sometimes, the engine must make sure that an internally generated item will pass a certain filter. In this case, the filter expression is applied to the data item first like in test mode, and if the result is true, nothing more happens. If the result is false (item does not pass), the *assign-to-pass modifiers* (see below, 7.2) in the filter expressions are applied from left to right until the data item passes the filter.

7.2 Basic filter syntax

The filter syntax might look a little unusual - however it is modeled after the TAF (Target Address Filter) syntax proposed by the SyncML standard for the *temporary* filters. The SyncML standard does not specify *dynamic* or *static* filters, but we use the same syntax (with some extensions) for all three filter types.

- A *filter expression* either consists of a single *filter term* or multiple *filter terms* concatenated with *logical operators*.
- *Logical operators* are: & (and), | (or). Note that there is no "NOT" operator.
- A *filter term* consists of a *filter expression* enclosed in parantheses or of an *identifier* followed by a *comparison operator* followed by a *constant*.
- A *identifier* is usually the name of an property or header field in a content format like vCard or RFC2822 email, but can also directly reference internal fields from the <fieldlist>. There are also a number of predefined special identifiers. Details see 7.3.
- *Comparison operators* are the usual =,<>, >, <, >=, <=. In addition % means "contains" and $ means "does not contain" (for strings). *Comparison operators* can be preceeded by three optional prefix characters, only in the following order:
  - a colon, called the *assign-to-pass modifier*, which means "assign to make true". This modifier is ignored in *test mode* (see 7.1), but used when by the sync engine in *make-pass mode* (when it needs to make a data item pass a filter and signals that assigning the value on the right of the operator to the field on the left of the operator will make the *filter term* evaluate to true). Note that, obviously, this makes only sense for comparisons where such an assignment actually makes the expression true: "FIELD := 2" will assign the value 2 to FIELD, which makes the comparison FIELD=2 true. However, "FIELD :<> 2" will not work, as assigning 2 to FIELD will obviously make the expression evaluate to false.
  - an asterisk, which means that the following *constant* is a special value (see below).
  - a '^' character, which makes the comparison case insensitive.
- *Constants* are string representations of the values to be compared. For strings, this is simply the characters the string consists of (no quotes around the string, no "&" or "|" or ")" might be contained). Numbers must be entered as decimal integers. Date and time values must be entered in ISO8601 format (yyyymmddThhmmssZ, for example 20030724T120000Z). If the *Comparison operator* is immediately preceeded by an asterisk, the *Constant* is treated as a special value: *E* meaning "empty" or *N* meaning "not assigned".

A few examples:

```
FIELD:*=E
```

Means: test if FIELD is empty. If the sync engine needs to make a data item pass the filter, it will assign an empty value to FIELD.

```
FIELD:>=2
```

Means: test if FIELD greater than or equal 2. If the sync engine needs to make a data item pass the filter, it will assign the value 2 to FIELD.

```
FIELD^=a simple text
```

Means:case insensitive test if FIELD contains the string "a simple text" ("A Simple Text" would match as well, as the comparison is case insensitive). There is no colon prefix, so the sync engine will not assign anything to FIELD even in case a data item needs to be made pass the filter (which means that the item cannot be made pass the filter, if the shown filter term is the entire filter expression).

```
FIELD1*=E|FIELD2:=4
```

Means: test if FIELD1 is empty or FIELD2 equals 4. If the sync engine needs to make a data item pass the filter, it will assign 4 to FIELD2, but only if FIELD1 is not empty - otherwise the expression is already true without modifying anything.

7.3 Identifiers in filters

Filter identifiers reference values that are to be compared. The following identifiers can be used in filter expressions:

- **a property name:** for MIME-DIR based formats like vCard and vCalendar, properties of the format (like TITLE, DTSTART, SUBJECT, etc) can be referenced directly by name.
- **a header line name:** for textprofile-based formats like email, named headers like "Cc", "From", "To" can be referenced directly.
- **a filter keyword:** textprofile and dataobj based formats allow defining keywords for some headers and content fields.
- **a field name from the <fieldlist>:** This gives direct access to the value of an internal field. This is useful e.g. to access elements of multi-value properties like N or ADR in vCard. To make sure an identifier is used to access the field list (and not a predefined special identifier), it can be prefixed by "F.", like "F.N_FIRST".
- **for vCalendar: START, END** – these are aliases for DTSTART and DTEND.
- **for vCard: FAMILY, GIVEN** – these are aliases for the lastname and firstname components of the N property.
- **for vCard: GROUP** – this is an alias for the CATEGORIES property.
- ***LOCALID*** – this means the local identifier (e.g. database key) of an item.
- ***REMOTEID*** – available in servers only. This means the identifier the client uses for a certain item.
- ***GUID*** –available in servers only. This is equivalent to the server's LOCALID.
- ***LUID*** or **&LUID;**– in a server, this is equivalent to REMOTEID, in a client it is equivalent to LOCALID.
- ***SINCE, BEFORE*** – these can only be used in CGI filters and SyncML DS 1.2 <filter>, but not in *static* filter expressions in the configuration file. These are pseudo-identifiers for defining a date range filter.
  `SINCE&EQ;20061206T120000&AND;BEFORE&EQ;20061231T090000`
  for example means the range from December 6th 12PM to December 31th 9AM, 2006. See also */dr()* below.

- ***MAXSIZE, MAXCOUNT*** – these can only be used in CGI filters and SyncML DS 1.2 <filter>, but not in *static* filter expressions in the configuration file. These are pseudo-identifiers for defining a maximum item size (in bytes) or a maximum item count, resp. See also */limit()* and */max()* below
- ***NOATT*** - this can only be used in CGI filters and SyncML DS 1.2 <filter>, but not in *static* filter expressions in the configuration file. Setting this equal to 1 (or "true" or "yes") suppresses attachments. See also */na* below.
- ***DBOPTIONS*** – this can only be used in CGI filters and SyncML DS 1.2 <filter>, but not in *static* filter expressions in the configuration file. It can be used to pass a string of implementation specific DB options. See also */o ()* below.

7.4 CGI Filter Syntax

When filter expressions are passed in the database path for SyncML 1.0 and SyncML 1.1, or when passing them via the <Filter> element in SyncML 1.2, they must be formatted like CGI parameters. The Synthesis SyncML engine also accepts most of the operators literally (useful when entering in phone clients with difficult input methods) , however to follow the SyncML standard, the following entities must be used:

&EQ;	=	equal (case sensitive)
&iEQ;	^=	equal (case insensitive)
>	>	greater than (case sensitive)
&iGT;	^>	greater than (case insensitive)
&GE;	>=	greater or equal (case sensitive)
&iGE;	^>=	greater or equal (case insensitive)
<	<	less than (case sensitive)
&iLT;	^<	less than (case insensitive)
&LE;	<=	less or equal (case sensitive)
&iLE;	^<=	less or equal (case insensitive)
&NE;	<>	not equal (case sensitive)
&iNE;	^<>	not equal (case insensitive)
&CON;	%	contains (case sensitive)
&iCON;	^%	contains (case insensitive)
&NCON;	$	does not contain (case sensitive)
&iNCON;	^$	does not contain (case insensitive)
&	&	and
&AND;	&	and
&OR;	|	or
&LUID;		is an alias for the LUID identifier.
&NULL;		can be used as value and means "no value" or "empty value".
&UNASSIGNED;		can be used as value and means "no value assigned".

In addition, in the value part of a filter term (the part following after the operator), a % sign followed by two digit hex number is interpreted as the the character with the ASCII-code or UTF-8 sequence element corresponding to the hex number, to allow including any char into values.

Therefore, the following two filter expressions are synonymous:

```
FIELD1*&EQ;&NULL;&or;FIELD2:&iEQ;test
FIELD1*=E|FIELD2:^=test
```

7.5 Special options in CGI filters passed with database path

As the SyncML 1.0 and 1.1 standards only provide TAF filters (*temporary inclusive* filters), Synthesis SyncML engine also parses some special options in the CGI to allow for additional flags and for differentiating between *excklusive* and *inclusive* filters:

/dr(-*before*,*after*)
: This specifies a date range relative to today, starting *before* days before today and ending *after* days after today. Note that actually implementing a date range filter for a datastore in a server needs scripting (see <filterscript> in <datatypes> in 10.5.4). Alternatively, the SINCE and BEFORE pseudo-identifiers can be used for defining a absolute date range, see 7.3.

/li(*kbytes*)
: This specifies a limit for example when dealing with possibly large data objects such as email. Note that actually implementing a limit for a datastore in a server needs scripting (see <processitemscript> in <datatypes> in 10.5.11). Alternatively, the MAXSIZE pseudo-identifier can be used for defining a size limit.

/o(*string*)
: This specifies an option string. The option string can be parsed in scripts, such as the <dbinitscript> to archieve special user-defined behaviour. Alternatively, the DBOPTIONS pseudo-identifier can be used for setting database options.

/slow Server only: this forces a slow sync even if neither client nor server have requested it. This is useful for clients which have no GUI to force a slow sync (such as Nokia 9210)

/na This specifies that no attachments should be transmitted to the remote party. This is useful for clients which cannot handle them to reduce traffic. Alternatively, the NOATT pseudo-identifier can be set to true to suppress attachments.

/max(*items*)
: This specifies a limit for the number of items to be sent to the remote (for example how many of the most recent email messages are to be sent). Note that actually implementing a limit for a datastore in a server needs scripting (see <datastore-initscript> in 11.34.21). Alternatively, the MAXCOUNT pseudo-identifier can be used for setting the item count limit.

/fi(*filter expression*)
: This option is provided to allow to specify *dynamic* filters (rather than *temporary*) in the CGI.

/tf(*filter expression*)
: This option is provided to allow to specify *temporary* filters mixing other options with filters in the same CGI. If only a filter expression is used, it can be specified in the CGI without the */fi* option. So the following two examples are synonymous:

```
FIELD1=3
/tf(FIELD1=3)
```

Note that while our own SyncML client products may provide user interface for these options, they work with any SyncML client that allows entering the database paths.

7.6 Filters in the configuration

Filters used in the configuration must always be specified in the basic filter syntax (see 7.2 above). None of the extra GCI options are allowed (nor would they make sense) in filters in the configuration. For details, see the description of the filtering tags (such as <acceptfilter> etc.).

8. General Global Configuration Options

8.1 <licensename>, <licensecode>: License

Contained in: <sysync_config>
Can contain: nothing
Attributes: none
Available: in non-demo versions

These tags are used to enable the server or client according to a license purchased. Please fill in the license name and code you have received with your purchase of the product as shown in the following sample:

```
<licensename>joe tester  joe@company.com</licensename>
<licensecode>2HRY-23LU-45AG-ORN5</licensecode>
```

If license name and code are correct, the product will work according to what you have licensed. The license information affects the number of simultaneous connections a server can handle, see 8.2. Note that there are permanent licenses (work forever) and time-limited evaluation licenses (stop working after a certain date). The license may also grant or deny access to some features, like using external plugin modules etc.

8.2 <maxconcurrentsessions>: concurrent sessions limit

Contained in: <sysync_config>
Can contain: integer value
Attributes: none
Available: in evaluation version only
Default: off

This tag is available in some versions to specify how many simultaneous sync sessions the server will allow. Note that the upper limit is defined by what your license allows (see 8.1).

8.3 <maxmsgsize>: max SyncML message size

Contained in: <sysync_config>
Can contain: integer value (number of bytes)
Attributes: none
Default: 20000 bytes for clients, 50000 bytes for servers.

Usually there is no need to change the message size, but in rare cases (with implementations not respecting message size limits, or testing message size related features like chunking) this allows to set the maximum size. Note that the engine allocates twice the maxmsgsize per session in memory, so setting huge values here might cause memory shortage.

8.4 <maxobjsize>: maximum object size

Contained in: <sysync_config>
Can contain: integer value (number of bytes)
Attributes: none
Default: 4000000 bytes

This tag determines maximum size an object (i.e. item, like a contact, a calendar entry etc.) can have. This is set to approx 4MBytes by default. If your application has larger data items to synchronize, this should be increased accordingly. Note that unlike with <maxmsgsize> (see 8.3), a high <maxobjsize> number here does not cause any memory usage per se. Memory is allocated only when actually needed for a large object.

8.5 <configidstring>: text to identify config

Contained in: <sysync_config>
Can contain: string
Attributes: none

This tag is useful to specify a string which is output to all debug log files and helps to identify what config file was in use for a particular session. It is recommended to put a text here that uniquely identifies your config version, such as "Config V2.7 for myserver.com, last edited by ME 2003-11-02".

8.6 <manufacturer>: text to identify product manufacturer

Contained in: <sysync_config>
Can contain: string
Attributes: none
Available: in Synthesis SyncML library products only

This tag can be used to set the manufacturer string which will be transmitted to remote parties in the SyncML device information as <man>. If this string is not defined or empty, "Synthesis AG" will be used. The actual manufacturer string can be read as a config variable (see 4.4).

8.7 <model>: text to identify model/product name

Contained in: <sysync_config>
Can contain: string
Attributes: none
Available: in Synthesis SyncML library products only

This tag can be used to set the model (product name) string which will be transmitted to remote parties in the SyncML device information as <mod>. If this string is not defined or empty, Synthesis' product name will be used. The actual model string can be read as a config variable (see 4.4).

8.8 <configvar>: define configuration variable

Contained in: <sysync_config>
Attributes: name, value

This tag can be used to define config variables (see chapter 4 for general information about config variables) in the configuration file, and use the defined value in subsequent configuration sections using the $(varname) syntax.

<configvar> has the following attributes:

- "**name**": This name of the variable to define (or redefine), without leading $ and paranthesis.
- "**value**": The value (string) to assign to the variable. Note that if the "expand" attribute (see 4.3) is specified before "value" and enables config variable expansion, $(varname) within the assigned string is expanded before assigning the value.

Defining config variables in a config file is useful for things like base paths for logs, binary files, sqlite files etc.

8.9 <configmsg>: define configuration variable

Contained in: <sysync_config>
Attributes: error, warning

This tag can be used to cause a configuration parsing error. This is useful together with conditional attributes (see 4.5), for example to generate an error message to the configuration error output path (usually the console stdout, or a special log file for servers - for syncml engine library also see "conferrpath" in 4.4) when a configuration file is used with an unsupported platform. The available attributes are:

- "**error**": shows a config error (which makes startup of the sync application fail).
- "**warning**": shows a warning on the config error output (but does not fail startup).

8.10 <scripting>: Global scripting definitions

Contained in: <sysync_config>
Can contain: <function>, <macro>, <looptimeout>
Attributes: none

This section contains global definitions for scripts contained in other sections.

8.10.1 <function>: User-defined function

Contained in: <scripting>
Can contain: user-defined function, see 6.13.1
Attributes: none

This is used to define functions that then can be called from scripts in other sections of the configuration. See 6.13.1 for description of function syntax.

8.10.2 <macro>: define macro

Contained in: <scripting>
Can contain: script text to be used in other scripts, see 6.12.1.
Attributes: name

This is used to define macros that then can be used in scripts in other sections of the configuration. See 6.12.1 for details.
The *name* attribute defines the macro's name, which is used to reference it in scripts, see 6.12.3.

8.10.3 <looptimeout>: maximum loop execution time

Contained in: <scripting>
Can contain: maximum loop execution time in seconds, 0 for unlimited.
Attributes: none
Default: 5 (seconds)

This is used to define the maximum time a loop statement (see 6.11) in a script might take. This is to prevent sessions to completely hang in a scripting loop.

8.11 <debug>: Debug Option Section

Contained in: <sysync_config>
Can contain: <logpath>, <enable>, <disable>, <xmltranslate>, <sepsessionlogs>, <msgdump>
Attributes: none

This tag contains all options for debug logs. Since the 2.1 versions of the Synthesis SyncML engine, debug logging has been significantly enhanced. In 2.1, debug log files were simple plain text without much structure, and therefore somewhat hard to read.
The new 3.0 engine has the following new options that make debug logs much more easily readable:

- Logs are now hierachically structured in indented blocks with timestamps at beginning and end. This groups all log output for a certain operation, item or message.
- Standard log format is now nicely colored HTML, which can be viewed in any web browser. Blocks can be collapsed/expanded while browsing (built-in Javascript). Links to jump between SyncML commands sent and status received are automatically added.
- Log writing is much faster, because it does no longer open and close the logfile for every log output line as in version 2.1.
- Alternative log formats are XML or plain text like in version 2.1, with or without indented blocks.

8.11.1 <logpath>: Directory path for debug log files

Contained in: <debug>
Can contain: full path to directory where to store log files
Attributes: none
Default: none (no debug output)

This tag specifies the directory where all debug-related log files are stored. For the ISAPI and Apache versions, please make sure that the server process' user (normally called IUSR_xxx under IIS or something like "www-data" or "wwwrun" under Apache) has write access to this directory. Note that the "platform" attribute (see 4) can be used to define different log paths for use on different platforms.

Example:

```
<logpath platform="win32">C:\sync\logs</logpath>
<logpath platform="linux">/var/log/syncml</logpath>
```

8.11.2 <enable>, <disable>

Contained in: <debug>
Can contain: nothing
Attributes: option
Default: by default, "normal" debug is enabled

The option attribute specifies which type of information should be enabled or disabled for logging. There are a number of separate debug topic and category options, and some useful groups of multiple topics that can be selected with a single <enable> or <disable> tag.

Separate debug topics:

- **"error"**: error messages. The standard HTML formatting **shows these in bold red**.
- **"hot"** : most important information (of all topics). This should never be switched of (except when switching off debug logging completely). The standard HTML formatting shows important information in **boldface**, using the color of the debug topic related.
- **"proto"**: SyncML protocol related information. The standard HTML formatting **shows syncml protocol related information in olive**.
- **"session"**: Session management related information. No special formatting.
- **"admin"**: Everything that has to do with administrative data (anchors, targets, map table). No special formatting.
- **"data"**: Everything that has to do with handling user data (data objects). Actual user data will however be shown only if "userdata" option is on as well (see below). No special formatting.
- **"remoteinfo"**: This shows information delivered in the remote party's device information, such as manufacturer name, datatypes supported, fields supported etc. The standard HTML formatting **shows remote's device information related messages in grass green**.
- **"parse"**: This shows information related to parsing and processing incoming data from the remote party. Actual user data will however be shown only if "userdata" option is on as well

(see below). The standard HTML formatting **shows parsing related messages in dark green**.

- **"generate"**: This shows information related to generating outgoing data for the remote party. Actual user data will however be shown only if "userdata" option is on as well (see below). The standard HTML formatting **shows generation related messages in dark blue**.
- **"transp"**: Shows transport (http and TCP communication) related information. Note that this type of log messages only appear in the global logs (see 8.11.18).
- **"syncml_rtk"**: Messages generated by the SyncML Toolkit code.
- **"rest"**: Any other debug log message that does not fit in any of the above topics.

Detail categories (these are combined with the topics above to determine the level of detail to be shown for the above topics):

- **"userdata"**: Anything that is user data. To create anonymized logs that do not show user's data, disable this category (and, depending on the database interface, "dbapi" as well, as it might show SQL commands revealing user data as well).
- **"dbapi"**: Information related to accessing the database. For ODBC, this enables showing SQL statements issued to the database, for plugin datastores, this includes all communication with the plugins and also messages generated by the plugin itself (see "plugin" below). The standard HTML formatting **shows database API message in dark pink**.
- **"plugin"**: Messages generated by database adapter plugins. These are messages **shows database plugin messages in mauve**.
- **"scripts"**: This is useful to debug scripts, and shows each line of executed scripts (but only for enabled debug topics!). Switching this on can generate huge log files, so it should normally be switched off in productive environments. The standard HTML formatting generally **shows script execution in brown**, but colorizes **executed code in bright blue**, **comments in light green**, **conditionally skipped code in grey** and expression results in red.
- **"expressions"**: (New in 3.1) Together with "scripts" this causes detailed step-by-step logging of script expression evaluation.
- **"filter"**: Information about data item filtering. The standard HTML formatting **shows filter processing in light brown**.
- **"match"**: Information about matching data in slow sync. **Note that together with "exotic" this can produce extremely large logs as matching is an O(N^2) operation, so use with care.** The standard HTML formatting **shows slow sync matching in a brownish orange**.
- **"conflict"**: Information about conflict resolution and data merging. The standard HTML formatting **shows conflict resolution and data merge information in dark red**.
- **"details"**: Enabling this option adds generally some more detail to the debug output.
- **"exotic"**: Enabling this adds the highest level of exotic detail possible. This is usually only required to track down device interoperability issues or bugs in the server/config. **If "exotic" and "match" are both enabled, extremely large logs can be produced as all slow sync matching is shown in full field by field detail. Use with care!** The standard HTML formatting **shows exotic details in orange**.

Groups of multiple topics/categories

- **"minimal"**: Just "error" and "hot"
- **"normal"**: This is the default, and shows "hot", "error", "data", "admin", "proto" and "remoteinfo". This gives an overview what is happening during a sync, but no details and no user data.
- **"extended"**: This shows extended info – almost everything except "script" and "exotic".
- **"maximal"**: Everything switched on, including "exotic", however "match" is disabled because together with "exotic" it would produce enormously large logs.
- **"db"**: Everything related to database access ("data", "admin", "dbapi", "plugin").
- **"all"**: Really everything. Should not be used normally with <enable>, but only for <disable>..

For compatibility with 2.1 version:

- **"items"**: alias for "data".
- **"cmd"**: alias for "proto".
- **"devinf"**: alias for "remoteinfo".
- **"dataconv"**: same as "parse" and "debug" together.

Note that enabling and disabling is done in the order specified, so to enable extended debugging topics but no user data, specify:

```
<enable option="extended"/>
<disable option="userdata"/>
```

8.11.3 <logformat>: select log file format

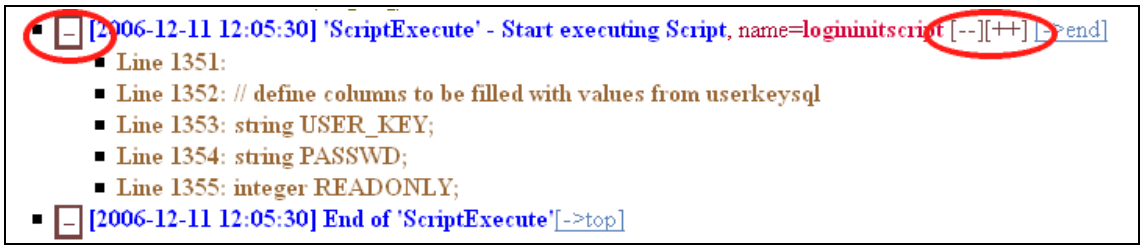
Contained in: <debug>
Can contain: "html", "xml", "text"
Attributes: none
Default: "html"

Selects the debug log format. Default is nicely colored HTML, but XML (useful for post-processing with XLT or other XML tools) and plain text (similar to version 2.1 logs) are possible.

8.11.4 <folding>: dynamic folding for HTML logs

Contained in: <debug>
Can contain: "none", "collapsed", "expanded", "auto"
Attributes: none
Default: "auto"

If not set to "none", HTML logs will contains some JavaScript to allow dynamically "fold" the block structure of the debug log. Every block can be collapsed or expanded individually. Using the "[++] and [--]" buttons on the right, a block and all of it's contained blocks can be expanded or collapsed with a single click.



The <folding> mode determines the initial folding state: "collapsed" means that all blocks are collapsed, "expanded" means that all are expanded and "auto" will have most blocks expanded, but some possibly lengthy, but for normal use not essential blocks will be shown collapsed. **Please note that the find function in browsers only searches the visible part of a HTML document, so messages in collapsed blocks cannot be found.** So if your log debugging mainly relies on searching in the browser, "expanded" is probably the best mode to make sure everything can be found. Alternatively, there is a button named "expand all" at the beginning of the log file which can be used to expand everything before using the browser's search function.

8.11.5 <timestamp>, <timestampall>: show timestamps in logs

Contained in: <debug>
Can contain: boolean value
Attributes: none
Default: true (for timestamp), false (for timestampall).

These boolean flags control if start and end of each block (<timestamp>) and every single log message (<timestampall>) should be prefixed with the current date and time (including milliseconds, if platform supports it). This is useful to see where a lot of time is spent during processing of SyncML messages. Usually, timestamps at start and end of each block is sufficient (the default).

8.11.6 <showthreadid>: show thread ID in logs

Contained in: <debug>
Can contain: boolean value
Attributes: none
Default: false for session logs, true for global logs.

These boolean flags control if every log messge should be prefixed with the ID of the current thread.

8.11.7 <timedsessionlognames>: show timestamps in logs

Contained in: <debug>
Can contain: boolean value
Attributes: none
Default: true.

These boolean flags control if the name of session logs contain the date of creation in their name. With this option, session log files are of the form:

pipe_odbc_**20061211T150022**_sXXXXXXXXXX.html

(where XXXXXXX is the session ID). Otherwise, session log files ar simply named like

pipe_odbc_sXXXXXXXXXX.html

8.11.8 <singlegloballog>, <singlesessionlog>: single file log option

Contained in: <debug>
Can contain: boolean value
Attributes: none
Default: false.

These boolean flags can be set to true when all global or session-related logging should be written into a single file, rather than creating a new file for each start of the server or new session, resp. Note that in case of the session log, the <appendtoexisting> flag (see 8.11.9) controls if the file is overwritten for every new session (erasing the previous log) or if new information is appended to an existing file when <singlesessionlog> is enabled.

8.11.9 <appendtoexisting>: append or overwrite existing session logs

Contained in: <debug>
Can contain: boolean value
Attributes: none
Default: false.

This flag controls if in <singlesessionlog> mode (see 8.11.8), an existing file is overwritten when a new session starts, or if new information is appended to the existing session log file. Note that the global log is always appended to.

8.11.10 <logflushmode>: select log file format

Contained in: <debug>
Can contain: "buffered", "flush", "openclose"
Attributes: none
Default: "buffered"

Selects how log information is written to the log file:

- **"buffered"** is the fastest mode – log file is only written when internal file buffer is full, that is ususally a few kilobytes at a time. This is the best mode for normal operation, but to debug crashes the contents of the internal buffer is lost when the application crashes, so the log will not show exactly where the crash occurred. In this case, use one of the other modes.

- **"flush"** is a good compromise between "buffered" and "openclose". The log file is kept open, but log information is flushed to the file after every log message, so even in case of a crash, the last message before the crash should be visible in the log file. **Note however that using flush mode when writing logs to a network volume might degrade performance significantly.**
- **"openclose"** is the slowest, but also safest mode for writing logs. The log file is opened for every log line to be written and then closed again. This mode was the only mode available in version 2.1.

8.11.11 <subthreadmode>: if and how to show log output from subthreads

Contained in: <debug>
Can contain: "suppress", "separate"
Attributes: none
Default: "suppress"

This determines how to handle debug output from simultaneously running subthreads of a session. Subthreads are created for loading the sync set (reading all items that are required in a sync session – in a slow sync this can be all items, so this can take a while, that's why this is done in a separate thread so the main thread can continue communicating with the clients to prevent them timing out).
If <subthreadmode> is set to "suppress", debug output from these threads is discarded and not stored at all.
If <subthreadmode> is set to "separate", a separate file is created for each thread, having the same file name as the main session log but with a "_xxxx" suffix (xxxx = thread ID). In HTML logs, the thread logs are linked from the main log at the point in the log where the subthread is started, which allows easy viewing.

8.11.12 <fileprefix>, <filesuffix>: text to add at begin and end of logfiles

Contained in: <debug>
Can contain: any text
Attributes: none
Default: standard prefix/suffix suitable for selected <logformat> (see 8.11.3)

Here you can define the text that is inserted at the beginning and end of a log file. This can be used to use a custom style sheet for HTML instead of the built-in, or to reference a XLT for XML logfiles.

8.11.13 <indentstring>: string to be used for indenting blocks

Contained in: <debug>
Can contain: any text
Attributes: none
Default: two spaces

This string is used to indent messages contained in blocks visibly. For HTML and XML logs, this should be whitespace in all cases, but for text logs any character can be used. The indent string is inserted at beginning of lines once for every indendation level.

8.11.14 <xmltranslate>: show traffic in XML

Contained in: <debug>
Can contain: boolean value
Attributes: none
Default: off

If set to on, incoming and outgoing messages (which are often in binary WBXML format) will be translated to XML format and written to the path specified with <logpath> with the following file naming scheme:

xxxx_sYYYYYYYYYY_trmNNN_MMM_outgoing.xml
xxxx_sYYYYYYYYYY_trmNNN_MMM_incoming.xml

Where

- "xxxx" is an identifier for the product being used such as "isapi_odbc" for Windows ISAPI version, "xpt_odbc" for the standalone versions, "pipe_odbc" for Apache based servers, "demo" for demo versions etc.
- YYYYYYYYYY is the internal session ID
- "trm" means "translated message"
- NNN is the SyncML message number
- MMM is a sequence number, which is just incremented for every message translation saved in a particular session. This is because sometimes messages are resent due to problems, so more than one dump with the same NNN value might exist. MMM ensures that these are saved in separate files.

Note: This option should never be switched on permanently in productive environment, as it requires a lot of additional memory, degrades performance and can lead to problems when completely malformed messages are processed. It is only for debugging problems with specific SyncML clients.
A good option in the PRO version is to use XML translation selectively – for example only for the first sync with a device. This is possible using the SETXMLTRANSLATE script function (see 6.14.5) for example in the <logininitscript> (see 11.33).

Example:

```
<xmltranslate>on</xmltranslate>
```

8.11.15 <msgdump>: dump SyncML traffic to files

Contained in: <debug>
Can contain: boolean value
Attributes: none
Default: off

If set to yes, messages received from and sent to client will be dumped 1:1 to files in the path specified with <logpath> with the following file naming scheme:

xxxx_sYYYYYYYYYY_msgNNN_MMM_outgoing.wbxml (or .xml)
xxxx_sYYYYYYYYYY_msgNNN_MMM_incoming.wbxml (or .xml)

Where

- "xxxx" is an identifier for the product being used such as "isapi_odbc" for Windows ISAPI version, "xpt_odbc" for the standalone versions, "pipe_odbc" for Apache based servers, "demo" for demo versions etc.
- YYYYYYYYYY is the internal session ID
- "msg" means "message", i.e. 1:1 dump of the actual message
- NNN is the SyncML message number
- MMM is a sequence number, which is just incremented for every message translation saved in a particular session. This is because sometimes messages are resent due to problems, so more than one dump with the same NNN value might exist. MMM ensures that these are saved in separate files.

The file will usually have .wbxml suffix (WBXML is a binary, space saving encoding method for XML). These can't be viewed with a text editor (consider <xmltranslate> to save a XML translation along with the dumped original, see 8.11.14) If the communication is in plain text XML, the dump files will have .xml suffix.

Note: This option creates a lot of small files for each message of a session, usually dozens or hundreds per session, depending on the amout of data transferred.
A good option in the PRO version is to use message dumping selectively – for example only for the first sync with a device. This is possible using the SETMSGDUMP script function (see 6.14.5) for example in the <logininitscript> (see 11.33).

8.11.16 <sessionlogs>: generate session logs

Contained in: <debug>
Can contain: boolean value
Attributes: none
Default: yes

If set to yes, Synthesis Sync Server writes session-specific logs. This is the normal case under almost every circumstance.

8.11.17 <sepsessionlogs>: No longer supported; use <singlesessionlog>instead

This tag is no longer supported in Version 3.x. Use <singlesessionlog> instead (see 8.11.8).

8.11.18 <globallogs>: generate global log

Contained in: <debug>
Can contain: boolean value
Attributes: none
Default: no

If set to no, no global log is written. This is recommended for productive environments, as for one the global log can get very large quickly, and the information contained is only useful to track down very low-level problems on the data transport level. In addition, writing the global log is slow because it must always use "openclose" <logflushmode> (see 8.11.10) as multiple threads are writing to the session log.

8.11.19 <logsessionstoglobal>: send session logs to global logfile

Contained in: <debug>
Can contain: boolean value
Attributes: none
Default: no

If set to yes, session logs will be sent to the global log file instead of creating separate log files for each session.

8.12 <configdate>: set timestamp for config file

Contained in: <sysync_config>
Can contain: date/time in ISO8601 format (yyyymmddThhmmss)
Attributes: none
Available: server only
Default: none

If this tag is specified, the server will assume that the last modification of the config file has happened at the date specified. Without this option, the server will use the file's modification date (as set by the operating system) to determine when the config has last changed.
The date/time of last config change is needed by the server to determine if a client must be sent updated "devinf" - which is always the case when a client's last sync date is before the last change to the config. See also <neverputdevinf> (8.13).

8.13 <neverputdevinf>: avoid PUT of devinf

Contained in: <sysync_config>
Can contain: boolean value
Attributes: none
Available: server only
Default: off

If this tag is set to on, the server will never send its device information "devinf" to the client if not specifically asked (client sending GET command).

8.14 <systemtimezone>: override local system time zone

Contained in: <sysync_config>
Can contain: time zone specification (see □)
Attributes: none
Default: system local time zone as retrieved from the operating system

This can be used (usually required for testing only) to set the system time zone to a specific time zone rather than retrieving it from the operating system.

8.15 <definetimezone>: define custom time zone as VTIMEZONE

Contained in: <sysync_config>
Can contain: time zone definition in VTIMEZONE format (as specified in iCalendar, RFC 2445)
Attributes: none

This can be used to extend the built-in set of named time zones (see chapter 17 for a list) by custom time zones specified in the VTIMEZONE format. Note that not all exotic features of VTIMEZONE as specified in RFC 2445 are supported). For general information on time zone handling please refer to chapter 5.

The following example shows how to define a new time zone named "ZUERICH" which is one hour east of UTC in winter and 2 hours east of UTC during daylight savings which is active since 1987 between last sunday in march and last sunday in october:

```
<definetimezone><![CDATA[
BEGIN:VTIMEZONE
TZID:ZUERICH
BEGIN:STANDARD
DTSTART:19671029T000000
RRULE:FREQ=MONTHLY;INTERVAL=12;BYDAY=-1SU
TZOFFSETFROM:0200
TZOFFSETTO:0100
END:STANDARD
BEGIN:DAYLIGHT
DTSTART:19870329T000000
RRULE:FREQ=MONTHLY;INTERVAL=12;BYDAY=-1SU
TZOFFSETFROM:0100
```

```
TZOFFSETTO:0200
END:DAYLIGHT
END:VTIMEZONE
]]></ definetimezone>
```

9. <transport>: Transport Configuration Section

Contained in: <sysync_config>
Can contain: <protocol>,<httpport>
Attributes: type

This tag encloses all transport-related configuration.
The "type" attribute is required and must have one of the following values:

- "xpt" for the standalone server (all platforms)
- "isapi" for the ISAPI-based server (Windows only)
- "pipe" for Apache/pipe based server (Linux, Mac OS X)

Note that for SyncML Engine V 1.0.8.50 and later the config file is allowed to have multiple <transport> sections. All transport sections that do not match the server type (isapi, xpt or pipe) will simply be ignored (before V1.0.8.50, this caused a config error). The advantage is that you now can use the same config file for all type of servers - simply include a <transport> section for each type. This is handy to test config files for a production ISAPI or Apache server using the standalone server first, without the need to modify *anything* in the file before using it with the production version later.

Example:

```
<transport type="xpt">
  <!-- options for use by the standalone server -->
  <protocol>HTTP</protocol>
  <httpport>80</httpport>
</transport>

<transport type="isapi">
  <!-- options for use by the ISAPI server -->
  <keepconnection>yes</keepconnection>
</transport>

<transport type="pipe">
  <!-- options for Apache/pipe based server -->
  <maxthreads>0</maxthreads>
  <maxsessionruns>200</maxsessionruns>
</transport>
```

9.1 <keepconnection>: HTTP 1.1 connection

Contained in: <transport>
Can contain: boolean value
Attributes: none
Available: standalone and ISAPI server only
Default: yes

This tag is used to enable or disable HTTP 1.1 "keep alive" / "keep connection" feature. If set, the server signals the client to not close the HTTP connection for every request but re-use it until the entire sync session is done.
Notes:

- **Supporting keep-alive is important for synchronizing with mobile devices – these often can open only a limited number of connections in a given time frame. So it is essential not to re-open a connection for every SyncML request, but use a single connection for the entire session.**
- In the Apache version of the server keep-alive cannot be controlled by the sync server config, but is a setting the apache configuration.

9.2 <bufferretryanswer>: buffer last answer for retries

Contained in: <transport type="isapi">
Can contain: boolean value
Attributes: none
Available: ISAPI server only
Default: yes

If this option is enabled, the server will buffer the last answer message sent for every session in progress. This will allow the server to re-send a message in case a client did not receive it and re-sends a particular request again.
This can reduce probability for aborted sessions when the connection is not stable (as sometimes the case in mobile environments). However, it only works with clients that can actually resend messages in case of transmission problems. A few recent client implementation will do that (including the Synthesis clients for PocketPC and Palm), but many others don't.
Note that switching this on will cause the server to require around 30-100k more memory per session in progress.

9.3 <protocol>: communication protocol

Contained in: <transport type="xpt">
Can contain: protocol name
Attributes: none
Available: in standalone version only
Default: HTTP

This tag is used to specify the transport protocol to be used. The following options are supported (attention, these are case sensitive!):

- "HTTP" : This is the default and specifies HTTP (Web) protocol.
- "OBEX/IR": This works only under Windows 2000. It defines that the server is listening for connections on the infrared port. Note that most existing clients (even those devices that have builtin infrared) don't support SyncML over infrared yet. Synthesis Sync Client 1.1 however supports infrared connections (under Windows 2000).
- "OBEX/TCP": This is a very seldom used connection mode. Don't use it unless you have distinct reasons to use this instead of HTTP.

9.4 <httpport>: HTTP and OBEX/TCP server port number

Contained in: <transport type="xpt">
Can contain: port number
Attributes: none
Available: in standalone version only
Default: 80

This tag specifies the TCP/IP port number for the HTTP server or for the OBEX/TCP server. If no other HTTP (Web) server is running on the same machine, the default of 80 is best (standard HTTP port). Note that some older SyncML clients might have problems accessing servers on other ports than 80. For OBEX/TCP, this should normally be set to 650.

9.5 <ipaddress>: listener IP address

Contained in: <transport type="xpt">
Can contain: IP address
Attributes: none
Available: in standalone version only
Default: 0.0.0.0 (=all)

This tag specifies the TCP/IP address for the HTTP server to listen. On machines with muliple IP addresses, this can be used to have the server listen on only one specific address instead of all (0.0.0.0)

9.6 <obexservice>: OBEX service name

Contained in: <transport type="xpt">
Can contain: OBEX service name
Attributes: none
Available: in standalone version only
Default: SYNCML-SYNC

This tag is relevant only when <protocol> is set to "OBEX/IR". It specifies the OBEX service name. This is conceptually similar to the HTTP port number. The default value of "SYNCML-SYNC" is the official standard service for doing SyncML over OBEX, so normally <obexservice> does not need to be specified.

9.7 <maxthreads>: Max number of session threads per server process

Contained in: <transport type="pipe">
Can contain: number
Attributes: none
Available: Apache pipe based server only
New in: Server version 3.0.2.0
Default: 0

This tag is used to control multi-threaded operation (multiple sessions run as threads within one single server process):

- **maxthreads=0:** single-threaded server, one process per sync session (default, and only mode available in servers before 3.0.2.0).
- **maxthreads=1:** only one session at a time per process, but after session finishes, server process keeps running and can process another session.
- **maxthreads>1:** multi-threaded server, can run the specified number of sessions in parallel threads. This is the recommended mode when starting a new server process is expensive in terms of memory or cpu, such as with Java based plugins (with a multithreaded server, the Java VM will be loaded only once for all sessions, in a single-threaded server, each session will instantiate a new Java VM).

Note that for multithreaded operation, mod_sysync/mod_sysync2 must be updated to the version included with server 3.0.2.0 and later. Otherwise, the server will still work, but will use a new process for every session (as with older servers that have no multithread support).

9.8 <maxsessionruns>: Max sessions to be run by a process

Contained in: <transport type="pipe">
Can contain: number
Attributes: none
Available: Apache pipe based server only
New in: Server version 3.0.2.0
Default: 0

If set to >0, this is the max number of sessions a server process should run before exiting. This can be useful to make sure the server environment is restarted once in a while to avoid eventual memory leaks in plugins etc. to accumulate too much

10. <datatypes>: Data Type Definitions

Contained in: <sysync_config>
Can contain: <fieldlist>, <mimeprofile>, <datatype>
Attributes: none

This section defines the content data types that are used in synchronisation. Most SyncML clients today support vCard (versions 2.1 and 3.0) and vCalendar (version 1.0).

Of course Synthesis Sync Server supports these as well. However, it can do much more as it has a fully configurable parser/generator for MIME-DIR formatted data. So Synthesis Sync Server can be used with most MIME-DIR based formats (see RFC2425 for details about MIME-DIR).

This allows you to customize existing formats like vCard and vCalendar exactly how you need them (they both allow a lot of variants depending on what information is required). In addition, you can define your own MIME-DIR based formats.

To change or create datatype config, it is important to understand how Synthesis Sync Server's datatype architecture works. There are three basic building blocks, that are needed to build a datatype:

1. **A field list**. Field lists are one-dimensional lists of data fields. For every possible value in a vCard, vCalendar or other MIME-DIR type, there should be an appropriate field in the field list. For plain text types (see below), a field list is also required. For example to store the vCard "N" property, the underlying field list should define 5 separate fields, as "N" contains 5 different name parts (first, middle, last names, suffix and prefix). Of course, if some of theses values are not relevant to your application, the field list does not need to include them. If your vCard should allow to repeat some values (such as 10 different telephone numbers), the underlying field list should provide a separate field for each repetition (and list them all in sequence) or use an array field (PRO version only, see 10.2).

2. **A mime-dir or text profile**. A mime-dir profile defines a format like vCard. Such a format is not just a list of values, but can also have the following features:
   - it may contain structured values (like the "N" property),
   - certain properties (such as "TEL" for telephone numbers) can occur more than just once
   - certain properties can have attributes (e.g. LANGUAGE) that need to be stored in the database as well
   - certain properties can have attributes (eg. TYPE) that qualify the contents of the property itself. For example, a "TEL" property having a "TYPE=WORK" attribute should be stored in another data field than a "TEL" property without an attribute.
   - a profile can contain subprofiles (like VTODO and VEVENT are subprofiles of the VCALENDAR profile)

   All of these cases can be handled in the MIME-DIR profile configuration. A MIME-DIR profile always relates to an existing field list (see 1.), as it specifies which fields are used to store the information contained in a MIME-DIR object.
   Likewise, a text profile defines header fields and types in a text base format like email or notes.

3. **A datatype specification**. A datatype specification finally defines a complete content type. It usually refers to a mime-dir or text profile to define the structure of the datatype, but provides some additional information such as the version of the datatype.

It is a common case that the same profile is referenced by different datatypes: in the sample files, there is one generic "vcard" mime-profile, which is used in two datatypes "vcard21" and "vcard30" to provide the different versions of the vCard type.
Datatypes are finally referenced by datastores in the <typesupport> section (see "11.34.11") to make them available for accessing a datastore.

10.1 <fieldlist>: internal data field list

Contained in: <datatypes>
Can contain: <field>
Attributes: name

A field list defines all data fields used by a certain data type. The name attribute is required to give the field list a name under which it can be referenced later in MIME-DIR profile definitons (see "10.3") and mapping lists (see "11.34.41").

For each field, the <fieldlist> must contain a <field> tag (see below).

Note that the names of the fields are used internally only and need not to be the same as the fields in your database. To associate fields from a field list with actual database fields, the <fieldmap> tag in <datastore> (see "11.34.41") is used.

10.2 <field>: definition of an internal field

Contained in: <fieldlist>
Can contain: nothing
Attributes: name, array, type, compare, age, merge

The <field> tag has the following attributes:

- "**name**": this is the internal name of the field. This name is used to reference the field in MIME-DIR-profiles and <fieldmap> tags, but it needs not to be the same as the database's field name (but of course, it can have the same name).
- "**array**" (PRO versions only): if this is set to true, the field is created as an array and will be able to store a list of values of the type specified in "type". This can be useful for storing values that can occur more than once in a record (such as EXDATE in vCalendar).
- "**type**" specifies the field's type:
  - "**string**" : string field.
  - **"multiline"** : string field, but intended for use with strings that consist of multiple lines. Comparisons of multilines ignores leading or trailing line ends.
  - "**telephone**": This is like string, except that when comparing two telephone fields, only the following characters are compared: 0..9, *, # and +. So, the following two telephone values will be considered equal: "+ 41 1 440 66 00" and "+4114406600". The reason for using telephone fields is that some clients do store telephone numbers including formatting spaces, some other clients don't.
  - "**url**": This is like string, except that it is intended to store an URL. To allow the widespread habit of entering WWW URLs without the "http://" prefix, this field type automatically adds the "http://" if no other service specifier (such as "ftp://") is specified.
  - "**timestamp**" : date and/or time field including time zone context information (see chapter 5 for details about timestamps and time zone handling).Standard string representation

is ISO8601 (useful for all combined date and time values in vCard and vCalendar). Depending on the context, other string representations are used (like RFC822 in emails). See chapter 5 for general information about timestamps and timezones.
    - "**date**" : date-only value. The only difference to "timestamp" is that the output is always a date-only value. On input, it can be assigned date and combined date/time values. Standard string representation is ISO8601 (useful for date-only values like BDAY in vCard). Depending on the context, other string representations are used (like RFC822 in emails). . See chapter 5 for general information about dates and timezones.
    - **"integer"** : Integer number (64 bit for servers and most clients, could 32 bit on some limited client platforms).
    - **"blob"**: This is a "binary large object" and can be used to store any chunk of binary data. This is useful for contact pictures or email attachments.
- "**compare**": This attribute controls how fields are compared in sync conflict, slow sync and first time sync cases. Note that in slowsync, only actually assigned (that is, transmitted) field values are compared, whereas when resolving conflicts during normal sync, all fields supported by both server and client are compared.
    - "**never**": field is not compared at all. This is for fields that do not contain user data, such as "REV" in vCard. It would not make sense to compare these fields, as they are not relevant for finding out if two objects have the same data or not.
    - "**conflict**": field is compared only when a sync conflict occurs (that is, when both client and server have modified versions of the same object). This mode should be set for all fields that contain user-entered data and which do not use "slowsync" or "always", see below.
    - "**slowsync**": field is compared in conflict case (like "conflict"), but in addition, it is also compared during slow-sync to match client objects with existing server objects. Therefore, "slowsync" should be set only on data fields that are important for identifying objects (such as name, company, country, but probably not details that might differ in server and client like telephone numbers, notes etc.). **Setting too many fields to "slowsync" carries the risk of creating duplicates during slow sync, because the matching criteria is too tight and small differences between client and server versions of a data record will prevent them to match.**
    - "**always**": field is always used in comparisons, not only in conflict and slow-sync cases, but also in "first time sync" case. This is the special case when a client and a server perform sync for the first time. This is different from slow-sync as in a first-time sync situation, it is often desirable to have relatively loose matching criteria (for example only compare first and last name) to match and union server and client objects. Use this only for fields that are absolutely essential for identifying an object.
- "**age**": This optional attribute can be set to "yes" for fields that are relevant in age comparison (i.e. finding out which one of two objects is more recent). If more than one field has the "age" attribute set, the fields that are defined first have precedence when comparing. In vCard, this attribute is normally used for the "REV" field, in vCalendar for the "DATEMODIFIED" field. If a format does not have a timestamp value that can be used to compare ages, no field must have the "age" attribute set. Note that for fields with "age" set, compare mode should be "never" normally.
- "**merge**": This optional attribute is used to define a merging mode for the field. Merging is used when resolving conflicts, that is when two versions of the same object exist and must be unified into one without loosing data. The following merge modes are available:
    - "**no**" : do not use merge with this field (this is the default when a field has no "merge" attribute.
    - "**fillempty**": If a field is empty in one object and has a value in the other one, the value is copied to the object which has no value. Note that this implies that "more data is better

than no data". While this is a sensible strategy in most cases, there might be cases where this is not the case.

- "**addunassigned**": This is like "fillempty" with the following difference: a value is only copied if it has a value in one object and *does not exist at all* in the other object. A value not existing is different from an empty value. For example, a vCard that includes an empty "TEL" property is explicitly saying: "there is no telephone number *at all*". A vCard that does not include a "TEL" property is saying "there is no telephone number *stored yet*". So in the second case it is ok to supply a telephone number once one gets available, while in the first case it might be unwanted to fill in one against the explicit statement of the sender that there is none. This is what "addunassigned" is for.
- "**lines**": This is very useful for multi-line fields (like NOTE in vCard). If field values differ, they will be merged on a line-by-line basis as follows: the resulting value will have all lines from both objects, but without duplicating identical lines. So, if one object has just a new line appended to the original note field, this line will simply be added to the other object as well.
- **any single character** (for example a comma, or space or any other): This works like the "lines" option, but instead of operating on lines, it operates on values separated by the specified character.
- "**append**": If a field's value is different between the two objects, the contents of both fields will be concatenated to build a new field value for both objects. For example, if one field contained "this", and the other "that", after conflict resolution both objects will have the value "thisthat" (or "thatthis").

Example (fieldlist for a simple vCard with name, private and work phone number and a notes field only):

```
<fieldlist name="Contact">
  <field name="REV" type="timestamp"
   compare="never" age="yes"/>
  <field name="N_LAST" type="string"
   compare="always"/>
  <field name="N_FIRST" type="string"
   compare="always"/>
  <field name="N_PREFIX" type="string"
   compare="slowsync"/>
  <field name="TEL_HOME" type="telephone"
   compare="conflict"/>
  <field name="TEL_WORK" type="telephone"
   compare="conflict"/>
  <field name="NOTE" type="string"
   compare="slowsync" merge="lines"/>
</fieldlist>
```

10.3 <mimeprofile>: definition of a mime-dir profile

Contained in: <datatypes>
Can contain: <profile>
Attributes: name, fieldlist

A MIME-DIR profile defines the entire structure of a MIME-DIR based datatype such as vCard or vCalendar. It must contain one <profile> tag (see "10.3.1").

<mimeprofile> has the following attributes:

- "**name**": This must be specified to name the MIME-DIR profile. In <datatype>, these names are used to reference a profile (see 10.5).
- "**fieldlist**": This must be the name of an already defined fieldlist (see 10.1). This fieldlist is the base on which a MIME-DIR type can be defined. All field names used in the definition of the MIME-DIR profile will reference fields from this fieldlist.

10.3.1 <profile>: root profile definition

Contained in: <mimeprofile>
Can contain: <property>,<subprofile>
Attributes: name, nummandatory

This tag defines the root of a MIME-DIR profile. MIME-DIR profiles can be multi-level, that is the root profile can contain several sub-profiles (such as VCALENDAR contains VTODO and VEVENT subprofiles). To define sub-profiles, use the <subprofile> tag (see "10.3.2").

<profile> has the following attributes:

- "**name**": This is the name of the profile. This is the name that appears in the BEGIN and END lines. For vCard, the name must be "VCARD", for vCalendar it must be "VCALENDAR".
- "**nummandatory**". This attribute is an integer and specifies how many properties of this profile are mandatory, that is, must be present in a valid data item. Normally, this is the number of properties in the profile that are flagged "mandatory" (see "10.3.3"). It can however a lower number, for example if the profile is valid if either one or another value is present: then, there will be two properties flagged "mandatory", but "nummandatory" would be set to 1. Note that this count relates only to properties of this profile, not of eventually contained subprofiles.

10.3.2 <subprofile>: nested subprofile definition

Contained in: <profile>
Can contain: <property>
Attributes: name, nummandatory, field, value, showlevel, showprops

This tag defines a sub-profile (such as VTODO, VEVENT or VTIMEZONE in VCALENDAR).

<subprofile> has the following attributes:

- "**name**": This is the name of the subprofile. This is the name that appears in the BEGIN and END lines..
- "**mode**": This defaults to "custom" for subprofiles which include <property> tags to define the supported set of properties. Special modes are:
  - "**vtimezones**" : This mode automatically creates or parses the VTIMEZONE records referenced by TZID parameters (<value conversion="tzid">, see see 10.3.4) of time-

stamp properties in iCalendar 2.0 based records. A <subprofile mode="vtimezones"> must not contain any <property> tags – the needed properties are implicitly created.

- "**onlyformode**": If set to "old" or "standard", this restricts the subprofile to exist in pre-MIME-DIR ("old", e.g. vCard 2.1, vCalendar 1.0) or MIME-DIR ("standard", e.g. vCard 3.0, iCalendar 2.0) datatypes (see 10.5.14 and 10.5.1) only.
- "**nummandatory**": This works as described for <profile>, see "10.3.1".
- "**field**": This optional attribute can specify the name of a field in the referenced field list, which is used to control this subprofile. See "value" below for details.
- "**value**": If "field" is set, this attribute specifies a value that...
  - ...must be contained in the specified field in order to generate that subprofile when generating an object.
  - ...will be written to the specified field when an object is received that contains this subprofile.
- "**showlevel**": obsolete since 3.2 – no longer needed as engine automatically calculates if a subprofile nees to be shown in devInf.
- "**showprops**": obsolete since 3.2 – no longer needed as engine automatically calculates if a subprofile nees to be shown in devInf.
- "**showifselectedonly**": If this attribute is set, the subprofile (and all of its subprofiles) are shown in devInf only if the subprofile is the selected subprofile, or if no subprofile is specifically selected. This is useful for vCalendar profiles, which contain both vEvent and vTodo subprofiles, but in a event-only datastore only vEvent is "selected".
- "**useproperties**": This optional attribute can be used to specify the name of an already defined profile or subprofile to use the same set of properties without the need to define them again. When this attribute is used, "showprops" defaults to false because normally the same properties should not be shown twice in the device information.

Example: skeleton profile and subprofiles for VCALENDAR (the field named KIND can contain "TODO" or "EVENT" depending on the type of VCALENDAR that is received or to be sent):

```
<profile name="VCALENDAR" nummandatory="1">
  <subprofile name="VEVENT" nummandatory="1"
   field="KIND" value="EVENT">
    <!-- add VEVENT properties here -->
  </subprofile>
  <subprofile name="VEVENT" nummandatory="1"
   field="KIND" value="TODO">
    <!-- add VTODO properties here -->
  </subprofile>
</profile>
```

10.3.3 <property>: property definition

Contained in: <profile>,<subprofile>
Can contain: <parameter>, <value>, <position>
Attributes: name, suppressempty, values, mandatory, show

This tag defines a MIME-DIR property. A property contains one or multiple data items and eventually some parameters.

<property> has the following attributes:

- "**name**": This is the name of the property. This is the name that appears at the beginning of a MIME-DIR object, such as "TEL:" for a telephone number.
- "**onlyformode**": If set to "old" or "standard", this restricts the property to exist in pre-MIME-DIR ("old", e.g. vCard 2.1, vCalendar 1.0) or MIME-DIR ("standard", e.g. vCard 3.0, iCalendar 2.0) datatypes (see 10.5.14 and 10.5.1) only.
- **"rule"**: This is the name of a remote rule (see 11.36) that must be active in order to activate this property. This allows to specify multiple property definitions for the same property "name". All adjacent <property> definitions having the same name are treated as a group. The group can contain alternatives for different remote rules, and it can also contain a default property definition (specify rule = "other" in the definition) that is used when no remote rule is active or the group does not contain a specific <property> for the currently active remote rule. Properties that have no "rule" attribute are unconditional, that is they are always active (this is the default).
- "**suppressempty**": This optional boolean value. can be set to true if this property must never be sent empty (without values). This is the case for most vCalendar fields, for example.
- "**delayedparsing**": This optional integer value can be set to indicate that the property must be parsed after all other properties with a smaller "delayedparsing" value have been parsed. This is useful for example for properies like RRULE where parsing may depend on values like DTSTART. The default value is 0 (means immediate processing).
- "**values**": This optional attribute specifyes how many values the property consists of. For example, the "N:" property in vCard consists of 5 values (first, last, middle, prefix, suffix). The default is 1. For each value, the property can contain a <value> tag that defines where and how to store the value (see "10.3.4"). The special values "list" or "expandedlist" can be used instead of a number for properties which contain multiple values of the same type, such as the EXDATE property in vCalendar. This will cause the values to be handled as if the property was occurring multiple times (allowing the use of all the <position> repeating mechanisms for the values). While "list" will generate a single property with multiple values on output, "expandedlist" will generate multiple instances of the same property, one for each value (some servers expect EXDATEs in that form).
- "**valueseparator**": This optional attribute can be used to specify the separator used for multi-valued properties. Normally, this is the semicolon (as in N), but for example to read CATEGORIES into elements of an array, "valueseparator" can be set to another separator, like the comma.
- "**altvalueseparator**": This optional attribute can be used to specify a alternate value separator that is also recognized when parsing properties. This is useful for properties like EXDATE or CATEGORIES where many implementations use ; instead of , or vice versa.
- "**mandatory**": This optional boolean attribute can be set to specify that this property is to be counted as mandatory (see "nummandatory" in <profile> and <subprofile>). Default is "false".
- "**show**": obsolete since 3.2 – use "showindefinv" instead.
- "**showindevinf** ": This optional boolean attribute specifies if the property should be shown in the device information. Default is "true".
- "**groupfield**" New in 3.4: This optional attribute can be used to specify a string field (usually a array or a repeating field) which represents the vCard or vCalendar group tag (a prefix to the property name, separated by a dot). The group tag can be used to link properties together which can occur multiple times. For example, some vCards might contain more than one ORG and TITLE. Now each title belongs to a particular organisation, so the group tag is used to represent that:

```
A.ORG:myOwnCompany
B.ORG:myEmployer
```

```
B.TITLE:employee
A.TITLE:boss
```

The groupfield mechanism makes sure that TITLE and ORG repetitions will be stored in the same repetition index (array position if ORG and TITLE are mapped to arrays) according to their group tag, even if occuring out of order in the incoming vCard.

10.3.4 <value>: property or parameter value storage

Contained in: <property>,<parameter>
Can contain: <enum>
Attributes: index, field, conversion, combine

This tag defines if and how a value of a property or parameter should be stored, and defines conversions that should be applied before storing it.

<value> has the following attributes:

- "**index**": This optional attribute specifies which one of multiple values (see "values" attribute in <property>) of a property this <value> tag applies to. The default is 1, so for properties with only one value it can be omitted. For parameters, "index" cannot be specified as parameters always have only one value.
- "**field**": This optional attribute can specifiy a field name (from the fieldlist referenced in <mimeprofile>) where the corresponding property value should be stored. If there is no "field" attribute, the corresponding property value will not be stored and an empty value will be generated when the object is generated by the server.
- "**conversion**": This optional attribute can specify a special conversion mode to be applied to the value. It can be one of the following:
  - "**version**": This special conversion mode means that the property value is the profile's version number. Its only use is for the VERSION property of profiles like vCard or vCalendar.
  - "**none**": This is the standard conversion (just copy). This is also the default if no "conversion" is specified
  - "**emptyonly**": This is the like "none", but value is only assigned to fields that are empty. This can be useful to only assign the first value of a value list.
  - "**tz**" : New in 3.1: When used with a timestamp field, the value is the standard (non-DST) minute offset for the timestamp (indended for vCalendar TZ property). Together with "daylight" (see below), this can be used to represent time zone in vCalendar 1.0 formats. It can be used also with a string field (representing a time zone name or UTC offset in minutes) or a integer field (representing the UTC offset in minutes).
  - "**daylight**": New in 3.1: Intended for vCalendar 1.0 DAYLIGHT property. When used with a timestamp field, the value is the assiociated time zone's daylight savings rule (for FALSE if no DST defined for the time zone) in the DAYLIGHT format.
  - "**tzid**": New in 3.1: Intended for iCalendar 2.0 TZID parameter. When used with a timestamp field, the value is a time zone identifier. Appropriate time zone descriptions can be included using a <subprofile mode="vtimezones"> (see 10.3.2).
  - "**zoneoffset_hours**" : No longer supported in 3.1.
  - "**zoneoffset_mins**": No longer supported in 3.1 – equivalent is using "tz" with an integer field.
  - "**zoneoffset_secs**": No longer supported in 3.1.

- "**timestamp**": Forces the output to be a timestamp (date + time), even if the referenced field is a date-only field.
- "**valuetype**": This is a special conversion mode to be used with VALUE parameters of some properties (e.g. date/time fields in iCalendar 2.0).
  For timestamps, if the referenced field is a time-only value, the conversion result is "TIME", if the referenced field is a date-only value, the conversion result is "DATE". Otherwise, the conversion result is empty which causes no VALUE parameter to be used, which denotes a timestamp value.
- "**date**": Render and parse as date-only, even if actual value is a datetime.
- "**autodate**": New in 3.1: This is for properties like DTSTART that can be either date-only or timestamp values in MIME-DIR formats (like iCalendar 2.0), but must always be timestamps in vCalendar 1.0. A date-only value is rendered as timestamp with 0:00 local-time in vCalendar 1.0, whereas it is rendered as a real date-only in MIME-DIR conformant formats.
- "**autoenddate**": New in 3.1: Similar to "autodate", however a date-only value is rendered as a timestamp at 23:59:59 localtime in the previous day when <autoenddateinclusive> (see 11.36.18) is set (otherwise, the end date is rendered as-is, which is 0:00 the next day). This is useful for DTEND properties.
- "**bitmap**": This is a special conversion mode for integer fields that represent a number of flags by their individual bits. It converts an integer number into a list of bit numbers - for example, the decimal integer value 17 (hex: 0x11) will be represented as "0,4" as Bit0 and Bit4 are set. This is very useful in conjunction with <enum>s (see 10.3.5), as these allow mapping list of numbers to list of identifiers. If the property had <enum>s relating WORK=0, HOME=1,FAX=2,PAGER=3,MOBILE=4, the above example "0,4" would correspond with "WORK,MOBILE", whereas 6 = 0x06 would correspond to the bitmap "1,2" which would represent "HOME,FAX". This can be very useful to efficiently store TYPE attributes of telephone number or email addresses.
- "**multimix**": New in 3.1: This is a very powerful conversion mode including the functionality of "bitmap", but additionally allows mixing bits from more than one field and also using literal values mixed with bitmapped values. This is useful for complex TYPE parameters like that from our iPhone client which has WORK, HOME etc. but also custom labels in the form "X-CustomLabel-xxxx" and IDs in the form "X-Synthesis-Ref-y" (see example below)
  - When the input string into the conversion is of the form "Bx" or "n.Bx", multimix works like "bitmap", i.e. it maps to the bit number x in an integer field.
  - When the input string into the conversion is of the form "Lyyyyy" or "n.Lyyyyy", the yyyyy part is stored in the referenced field.
  - In both variants, the "n." prefix defines an offset in the field list relative to to the main field selected by the 'name' parameter, such that more than one field can be targeted by a multimix conversion.
- "**rrule**": This is a special conversion mode for vCalendar RRULE, it is available only in types based on "vcalendar". "rrule" conversion mode requires that the "field" attribute references not a single field, but the first field of a so-called RRULE field block. See 10.6 for details.
- "**blob_b64**": the contents of the associated field (usually a BLOB) is represented as a base 64 encoded binary value. This is for example required for vCard PHOTO.

Note: For converting enumerated values, the <value> tag can contain <enum> tags, see below.

- "**combine**": This option can be used to combine the values of multiple properties into a single field. It can be one of the following:
  - "**no**" : do not combine values. This is the default.

- "**lines**" : combine values by storing each value on a new line
- **any single character:** combine values by storing them separated by the specified character.

The following example shows how to use the "multimix" mode in combination with the "prefix" <enum> mode (see 10.3.5). For each telephone number in TEL_NUMBERS array, the TEL_FLAGS holds a bitmap coding CELL, HOME and WORK in any combination, TEL_IDS holds a numeric ID transferred as "X-Synthesis-RefX" and TEL_LABELS can hold a custom label. The 1 in "1.L" is the offset between the value's specified field (TEL_FLAGS) and the field that should actually be used to store (TEL_IDS). Same for the "2.L". Note that if there is no field offset, the value must be specified without a "0." prefix, like the "B0", "B1" and "B2" values.

```
<!-- these fields are part of the <fieldlist> -->

  <!-- the actual telephone numbers -->
  <field name="TEL_NUMBERS" array="yes" type="telephone"/>
  <!-- CELL, WORK, HOME flags for each number encoded as bits 0, 1, 2 -->
  <field name="TEL_FLAGS" array="yes" type="integer"/>
  <!-- a numeric ID used to identify TEL occurrences uniquely -->
  <field name="TEL_IDS" array="yes" type="integer"/>
  <!-- custom label strings a TEL might have -->
  <field name="TEL_LABELS" array="yes" type="string"/>

...

<property name="TEL">
  <value field="TEL_NUMBERS"/>
  <parameter name="TYPE" default="yes" show="yes">
    <value field="TEL_FLAGS" conversion="multimix" combine=",">
      <!-- CELL, HOME, WORK map to bits 0,1,2 in TEL_FLAGS -->
      <enum name="CELL" value="B0"/>
      <enum name="HOME" value="B1"/>
      <enum name="WORK" value="B2"/>
      <!-- from the ID sent as X-Synthesis-RefX, X is to be stored
       literally in the field with offset 1 from TEL_FLAGS (=TEL_IDS) -->
      <enum mode="prefix" name="X-Synthesis-Ref" value="1.L"/>
      <!-- from the label sent as X-CustomLabel-yyyy, yyyy is to be stored
       literally in the field with offset 2 from TEL_FLAGS (=TEL_LABELS) -->
      <enum mode="prefix" name="X-CustomLabel-" value="2.L"/>
    </value>
    <!-- accept any number of TEL properties, store them in array fields -->
    <position field="TEL_NUMBERS" repeat="array" minshow="1"/>
  </parameter>
</property>
```

10.3.5 <enum>: enumerated values

Contained in: <value>
Can contain: nothing
Attributes: name, value, positional

This tag defines a name/value pair. All <enum> tags contained in a <value> tag form an enumeration list which will be used to convert values contained in the MIME-DIR based format into values more convenient for internal use: If a value in the MIME-DIR object matches the "name" of an <enum>, the "value" of that <enum> is used as value to store into the internal field (and in the database, finally).

<enum> has the following attributes:

- "**name**": This is how the value is shown in the MIME-DIR object.
- "**value**": This is how the value is shown internally (and stored in the database).

- "**positional**": This is an optional boolean attribute which can be used to exclude or include a certain value into the set of values that control the storage position of data (see "10.3.7"). Normally, this needs not to be specified, but to have <enum> lists with mixed positional and non-positional values, it can be specified.
- "**mode**": This attribute can be used to specify special enum modes as follows (the default is "translate"):
  - "**translate**": default mode, translate 1:1 between name and value
  - "**defaultname**": when translating from values to names, and no matching value is found, the result is taken from the name of the <enum> with mode="defaultname" (if no such <enum> exists, the value is passed as name without translation).
  - "**defaultvalue**": when translating from names to values, and no matching name is found, the result is taken from the value of the <enum> with mode="defaultvalue" (if no such <enum> exists, the name is passed as value without translation).
  - "**ignore**": if a value or name matches an <enum> with mode="ignore", it will be ignored, i.e. the result generated is empty.
  - "**prefix**": this works similar to "translate", however the translation is applied even if only the beginning of the name or value string matches. If so, the remainder of the string are appended to the output string. As an example: a <enum mode="translate" value="B" name="X-BitNumber-"/> would translate "X-BitNumber-2" to "B2" or "X-BitNumber-3456" to "B3456" and vice versa. See 10.3.4 for an example using prefix enum mode together with multimix conversion mode.

Example: Usage of an <enum> list to convert the STATUS attribute of VCALENDAR into an internal numeric representation:

```
<property name="STATUS">
  <value field="NUMERIC_STATUS">
    <enum name="ACCEPTED"     value="1"/>
    <enum name="NEEDS ACTION" value="2"/>
    <enum name="SENT"         value="3"/>
    <enum name="TENTATIVE"    value="4"/>
    <enum name="CONFIRMED"    value="5"/>
    <enum name="DECLINED"     value="6"/>
    <enum name="COMPLETED"    value="7"/>
    <enum name="DELEGATED"    value="8"/>
  </value>
</property>
```

10.3.6 <parameter>: property parameter definition

Contained in: <property>
Can contain: <value>,<position>
Attributes: name, default, show, positional,shownonempty

This tag defines an attribute to a property. Parameters are supplementary information to a property which often defines things like language or type of the information stored in the property's value(s).

There are two main uses of parameters:

- non-positional use: using them as additional values and store them like any other property value.
- positional use: use the value of a parameter to determine in which field the property's value should be stored. An example of this is the TYPE parameter of the TEL property in vCard: If TYPE=WORK, the number should be stored in a TEL_WORK field, but when TYPE=HOME, the number should be store in TEL_HOME. Positional parameters must always have some <enum> tags in their <value>; these will be used with the <position> tag (see "10.3.7") to define the rules for storing property values according to parameter values.

<parameter> has the following attributes:

- "**name**": This is how the parameter is shown in the MIME-DIR object. For example, the "TYPE" can be used to specify the type parameter in the vCard TEL property (which look like: "TEL;TYPE=WORK:123456").
- "**onlyformode**": If set to "old" or "standard", this restricts the property to exist in pre-MIME-DIR ("old", e.g. vCard 2.1, vCalendar 1.0) or MIME-DIR ("standard", e.g. vCard 3.0, iCalendar 2.0) datatypes (see 10.5.14 and 10.5.1) only.
- "**default**" is an optional boolean attribute. If it is set, the parameter is treated as default parameter. This has significance in non-standard (pre-MIME-DIR) formats like vCard 2.1 only, and means that the parameter value(s) appear without the parameter name (such as "TEL;WORK:123456" which is vCard 2.1 format for "TEL;TYPE=WORK:123456"). By default, this is "false".
- "**showindevinf** " (also "**show**" is allowed for backward compatibility, but is deprecated in 3.2 and later) is an optional boolean attribute. If set to true, this parameter and (if any) its defined <enum> values will be shown in the device information. This should usually be set on "TYPE" parameters for properties like "TEL" or "ADDR", to show the client what kind of telephone numbers/postal addresses the server supports. Some phone device clients will not send more than a single tel number if the devInf does not contain this information. Default is "false". Note that this option has effect only if the containing profile/subprofile has not set "showprops" to "false".
- "**positional**" is an optional boolean attribute. It specifies if the parameter is used to determine where to store property values (see above). The default is "false".
- "**shownonempty**" is an optional boolean attribute. It specifies that the property which contains this parameter is to be show when this parameter contains a value, even if the property itself contains no value(s). The default is "false" which means that the property is only shown if it has main values(s).

Example: make sure that the value of the LANGUAGE attribute of the NOTE property gets stored into the field "NOTE_LANG" (non-positional parameter):

```
<property name="NOTE">
  <value field="NOTE"/>
  <parameter name="LANGUAGE">
    <value field="NOTE_LANG"/>
  </parameter>
</property>
```

10.3.7 <position>: control storage position and repetitions

Contained in: <property>,<parameter>
Can contain: nothing
Attributes: has, hasnot, shows, field, repeat, increment, minshow

This tag is used together with positional parameters (or generally with repeating properties to define how and how many times these are stored). It can be used to define rules how parameter values influence in what field the enclosing property's data is stored. It also handles the case of repeating properties, such as multiple TEL properties in a single vCard object.

Note that while this is a very powerful option, it is also rather complex to understand and use. So we recommend to look at the sample config files distributed with the server before defining your own <mimeprofile>. They show how <position> can be used in some standard cases for vCard and vCalendar.

<position> can appear in a <parameter> or in <property>. If the position (field(s) where to store property value(s) depend on a *single* parameter's value, <position> should be put inside that <parameter>. Otherwise, <position> should be inside <property> (especially when the <position> is used only to define a repeat count for a property).

<position> has the following attributes:

- "**has**", "**hasnot**": These attributes both specify one or several (comma separated) <enum> name(s), either from the <enum> list of the enclosing <parameter>, or from any other <enum> list of a parameter in the <property>. If the <enum> being referenced is not in the same <parameter> as the <position> tag, the <enum> name must be prefixed with the parameter name. For example, to reference the "WORK" <enum> from the "TYPE" parameter, you should write "TYPE.WORK".
  - "has" specifies the attribute values that must be present for the <position> to apply to a property's values. For example, `has="TYPE.WORK"` specifies a property that must have a TYPE-attibute with a value of "WORK".
  - "hasnot" specifies the attribute values that prevent the <position> to apply to a property's values. For example, `hasnot="TYPE.MODEM,TYPE.FAX"` specifies that the property may not be of TYPE MODEM or FAX.
- "**shows**": This attribute is like "has", but can be used to specify additional values that a parameter should have when the object is sent to the remote party. For example, on voice telephone numbers it does not make sense to specify "VOICE" in "has" (because not all clients tag voice numbers as such), but it would be a good idea to specify `shows="TYPE.VOICE"`, so voice numbers in outgoing vCards will have the "VOICE" type set.
- "**field**": Simply said, this attribute defines the field where the property value is stored when the <position> applies according to its "has" and "hasnot" attributes. However, this simple description is only true if the property has a single value. The whole complicated truth is: The field specified is used to calculate a difference between the numeric positions in the fieldlist of that field and the first field specified in a <value> of the enclosing property. This offset is then added to *all* the fields that store values of this property and *all* its non-positional parameters (so the entire block of fields is offset). In order to use this feature, you need to make sure that the field list is designed appropriately for offsetting fields or blocks of fields.
- "**repeat**": This attribute specifies how many times this <position> can be applied. For example, a database might provide 3 fields for work telephone numbers, so the <position> that

catches work numbers (using has="TYPE.WORK") can have a "repeat" attribute of 3. Repeat can also be set to the following special values:

- "**rewrite**": this means that if the property occurs more than once, the last occurrence will be stored. Note that this is different from the default value "1" for "repeat": This will store the first occurrence, but if the property occurs again, it will not match this <position> again, but eventually a subsequent one.
- "**array**" (PRO versions only): this can be used when the target field is an array field. In this case, all repeated occurrences will be stored in the elements of the array (instead of using field offsets). **Important Note:** before version 3.1.x, when an empty array element is encountered when generating properties from an array, this stops generation, even if subsequent array elements exist and are non-empty. For properties with parameters, these are not checked for being empty by default, but they can be included by using the "shownonempty" attribute of <parameter> (see 10.3.6).

Note that a <position> specifying "repeat" (and probably "increment" as well) can also be used completely independently of any positional parameters (no "has", "hasnot" etc.), for a property that is simply allowed to repeat several times.

- "**increment**": This is needed only when "repeat" is used. It specifies the increment (offset) that is added to the numeric position of the field(s) after each repeated occurrence. So, if a database has 3 fields for work telephone numbers, and they are listed one after the other in the fieldlist, the <position> will have a "repeat" of 3 and an "increment" of 1.
  For a setup where the database has 5 blocks of 2 fields each, one for the telephone number itself and one for the TYPE parameter value, "repeat" would be 5 and "increment" would be 2.
- "**minshow**": This specifies how many times a property is shown minimally in a generated object. For example, in a database with 3 fields for work telephone numbers, you might not want to show 3 "TEL" properties unless they really contain data. But probably you still want one "TEL" property to show in all cases, even if the database has no work telephone numbers stored at all. In this case, you could set "minshow" to 1. By default, "minshow" is equal to "repeat", meaning that all possible repetitions of a property will be shown in generated objects, even if they are empty. Please note that if the "suppressempty" attribute of <property> (see "10.3.3") is set, empty properties are generally suppressed. There is also a device-specific setting <noemptyproperties> in <remoterule> (see "11.36.5") which overrides "suppressempty" and "minshow" when set (it causes that empty properties are *never* generated for that specific device - some devices just can't handle empty properties at all).
- **"readonly"**: This specifies that this position rule is only used for parsing data, not for generating data. This allows specifying more than one <position> rules for the same values. The parser will try to apply them in the order of appearance. Default is "false".
- **"overwriteempty"**: If this is set to true, the parser will overwrite empty occurrences of a repeating <position> specifications with subsequent occurrences. This avoids wasting storage with storing empty values. Default is "true".

Simple example: a vCard for a database with 2 fields for email addresses:

```
<!-- assuming a fieldlist with two email fields
     defined in sequence:
     EMAIL_1, EMAIL_2
-->

<property name="EMAIL">
  <value field="EMAIL_1"/>
  <position field="EMAIL_1" repeat="2" increment="1"/>
</property>
```

Full featured example: a vCard with 8 telephone numbers, 4 of them for specific purposes, and 4 more for additional numbers (note the comments):

```
<!-- assuming a fieldlist with the following fields
     defined in sequence:
     TEL_HOME, TEL_WORK, TEL_MOBILE, TEL_FAX,
     TEL_AUX_1, TEL_AUX_2, TEL_AUX_3, TEL_AUX_4
-->

<property name="TEL">

  <!-- TEL_HOME is the first field in the fieldlist -->
  <value field="TEL_HOME"/>

  <!-- TYPE is positional, as its value determines in
       which field the number will be stored -->
  <parameter name="TYPE" default="yes" positional="yes">

    <!-- these are the values that are of interest for
         positioning -->
    <value>
      <enum name="HOME"/>
      <enum name="WORK"/>
      <enum name="CELL"/>
      <enum name="FAX"/>
      <enum name="VOICE"/>
    </value>

    <!-- a telephone number that is a HOME number,
         but not a FAX or CELL number, is stored
         in the TEL_HOME field. When generating
         the object, VOICE is also added to the
         TYPE parameter -->
    <position has="HOME" hasnot="FAX,CELL"
     shows="VOICE" field="TEL_HOME"/>

    <!-- a telephone number that is a WORK number,
         but not a FAX or CELL number, is stored
         in the TEL_WORK field. -->
    <position has="WORK" hasnot="FAX,CELL"
     shows="VOICE" field="TEL_WORK"/>

    <!-- a telephone number that is a CELL number
         is stored in the TEL_MOBILE field -->
    <position has="CELL" shows="VOICE"
     field="TEL_MOBILE"/>

    <!-- a telephone number that is a FAX number
         is stored in the TEL_FAX field -->
    <position has="FAX" field="TEL_FAX"/>

    <!-- up to 4 additional telephone numbers
         that do not match any of the <position>
         specifications above
         (such as one with different TYPE or
         a SECOND work, home or mobile number)
         will be stored in TEL_AUX_1 up to
         TEL_AUX_4.
         Note that if there are more than one
```

```
          Home, Work, Fax or Cell number, the extra
          number are also stored in the TEL_AUX fields.
          When generating the object, and no
          additional telephone numbers are
          assigned, minshow="0" prevents that
          any empty TEL properties are generated -->
      <position field="TEL_AUX_1" repeat="4"
       increment="1" minshow="0"/>

    </parameter>
</property>
```

10.3.8 <vtimezonegenmode>: VTIMEZONE generation mode

Contained in: <mimeprofile>
Attributes: none
Default: none

This tag defines how VTIMEZONE records should be generated in a mimeprofile which contains datetime values specifying a TZID parameter. It can have one of the following values

- "**current**": VTIMEZONE is generated for descibing the current time only.
- "**start**": VTIMEZONE is generated for descibing the earliest date used in the record.
- "**end**": VTIMEZONE is generated for descibing the latest date used in the record.
- "**range**": VTIMEZONE is generated for descibing all dates used in the record.
- "**openend**": VTIMEZONE is generated for describing all dates starting from the earliest date used in the record up to the latest time zone defintion known.

10.3.9 <unfloattimestamps>: handling of floating timestamps

Contained in: <mimeprofile>
Attributes: none
Default: false

If set, parsed datetime values that do not specify a time zone (i.e. have no TZID or no Z suffix) will be assigned to the item's time zone (which is usually the user's time zone). See 5.2 for details about time zone contexts. This is useful for DB configurations that cannot store floating time stamps and must interpret them in a fixed time zone context.

10.4 <textprofile>: definition of a text format profile

Contained in: <datatypes>
Can contain: <linemap>,
in PRO version only: <mimemail>, <sizelimitfield>, <bodymimetypesfield>, <bodycountfield>, <maxattachments>, <attachmentcountfield>, <attachmentmimetypesfield>, <attachmentsfield>, <encodedattachments>, <attachmentsizesfield>, <attachmentnamesfield>
Attributes: name, fieldlist

A text profile relates internal fields to (header) lines of a plain-text format like email. It contains one or multiple <linemap> tags (see 10.4.1).

<textprofile> has the following attributes:

- "**name**": This must be specified to name the text profile. In <datatype>, these names are used to reference a profile (see 10.5) with the <use> tag.
- "**fieldlist**": This must be the name of an already defined fieldlist (see 10.1). This fieldlist is the base on which a text type can be defined. All field names used in the definition of the text profile will reference fields from this fieldlist.

10.4.1 <linemap>: mapping of text based formats to database fields

Contained in: <textprofile> (or <datatype>, only for compatibility with version 2.1)
Can contain: <numlines>, <inheader>, <allowempty>, <headertag>, <filterkeyword>
Attributes: field

Note: In version 2.1, <linemaps> were defined directly within <datatype>. This is still possible in version 3.x for compatibility, **but no longer recommended**. Please create <textprofile> (see 10.4) sections for your <linemaps> and reference them from <datatype> via <use>.

This tag is used to map one or multiple lines of the text based data format to a field of the fieldlist. The "field" attribute is required:

- "**field**" specifies the name of the field from the fieldlist which is to be mapped to one or multiple lines of the text format.

The tag contains further tags to define how the lines are mapped

10.4.2 <numlines>: Number of lines to map

Contained in: <linemap>
Can contain: integer number
Default: 0 (all lines)

This tag is used to specify how many lines are to be mapped to the field specified in <linemap>. If the value is 0, this means that all remaining lines should be mapped to the specified field. Note that for linemaps with <headertag>, <numlines> is irrelevant (a header is always decoded into a single line – even if present as multiple lines with RFC822 folding).

10.4.3 <inheader>: header lines

Contained in: <linemap>
Can contain: boolean value
Default: false

This tag is used to specify if the line(s) in the <linemap> belong to a email-style header or not. E-mail headers are separated by body text by a single empty line. E-Mail header style lines can be represended on multiple lines using header folding mechanism. The folding/unfolding takes place automatically – the referenced internal field will always contain a single line.

10.4.4 <allowempty>: empty field handling

Contained in: <linemap>
Can contain: boolean value
Default: false

If this option is set, one empty line will be generated if the mapped field is empty, otherwise, empty fields will just be ignored when creating the text output.

10.4.5 <headertag>: tagged header handling

Contained in: <linemap>
Can contain: header name
Default: empty

This option is used to parse and generate RFC822-type headers (for example, email messages). The *header name* is the name of the header **including the colon**! If an input line begins with the *header name*, it is assigned to the *field* specified in the *linemap*. When generating output, the output text is preceeded by the *header name*. Note that long header lines are folded according to RFC822 specs.

Example for the "From." header of email messages:

```
<linemap field="SENDER">
  <headertag>From:</headertag>
  <inheader>true</inheader>
</linemap>
```

10.4.6 <valuetype>: type of text field

Contained in: <linemap>
Can contain: value type name
Default: text

This option allows four type options:

- **text** (default): contents are treated as plain text
- **date**: contents are treated as RFC822 date/time specification

- **body**: contents are treated as a RFC822/MIME email body (that can include multiple body variants and attachments). Note that this option is only available in the PRO versions. Use the RFC822 email body options (see 10.4.7) to specify details for storing the body elements. The body text itself will be stored in the field specified by the enclosing <linemap>. If that field is an array, multiple format variants of the body will be stored in subsequent array elements (also see <bodymimetypesfield> and <bodycountfield> below in 10.4.7). Otherwise, only the plain text variant of the body will be stored.
- **rfc2047**: contents are treated as RFC2047 encododed. This is the encoding used to represent non-ASCII characters in RFC2822 style email headers.

10.4.7 RFC822 email body options

Contained in: <textprofile> (<datatype>, only for compatibility with version 2.1)
Available: in PRO version only

Note: In version 2.1, the following tags were defined directly within <datatype>. This is still possible in version 3.x for compatibility, **but no longer recommended**. Please create <textprofile> (see 10.4) sections and reference them from <datatype> via <use>.

These tags are used to control parsing/generation of RFC(2)822-type email bodies and are applied to *linemap*s that have a *valuetype*of *body*:

- **<mimemail>*boolean*</mimemail>:** If set to false (default), no RFC(2)822 email body parsing/generating is done at all. For email format support, this must be set to true.
- **<maxattachments>*number*</maxattachments>:** Specifies the maximum *number* of attachments that will be processed
- **<attachmentcountfield>*field*</attachmentcountfield>:** Specifies a *field* that will be assigned the number of attachments in a parsed email message
- **<attachmentmimetypesfield>*field*</attachmentmimetypesfield>**: specifies the *field* (normally an array) that is used to store the attachment MIME-types.
- **<attachmentsfield>*field*</attachmentsfield>:** specifies the *field* (normally an array) that is used to store the attachment contents.
- **<attachmentsizesfield>*field*</attachmentsizesfield>:** specifies the *field* (normally an array) that is used to store the size (in bytes) of the attachments.
- **<attachmentnamesfield>*field*</attachmentnamesfield>:** specifies the *field* (normally an array) that is used to store the file names of the attachments.
- **<sizelimitfield>*field*</sizelimitfield>:** specifies the *field* that is used by Synthesis SyncML Server and clients to transmit email size constraints with the special "X-Sync-Message-Limit" header. Note that this is a very advanced setting, and needs a lot of scripting in a config file to make it work with a server database. See implementation for email in the sample config.
- **<bodymimetypesfield>*field* </bodymimetypesfield>**: specifies the *field* (normally an array) that is used to store the MIME-types of the possibly multiple body variants of the email message (such as text, html, rtf).
- **<bodycountfield>*field* </bodycountfield>:** Specifies a *field* that will be assigned the number of body variants found in a parsed email message.

10.5 <datatype>: definition of a datatype

Contained in: <datatypes>
Can contain: <use>,
If basetype is "vcard" or "vcalendar": <version>,
If basetype is "mimedir": <mimedirmode>, <typestring>, <versionstring>
If basetype is "text": <typestring>, <versionstring>, (<linemap> for compatibility with version 2.1, but these should be moved to a <textprofile> if possible)
Attributes: name, basetype

This tag is used to define a datatype that can then be referenced by a datastore as content format (see "11.34.12").

The "name" and "basetype" attributes are required:

- "**name**" specifies the name under which the datatype can be referenced by a datastore. For example, vCard 2.1 is named "vcard21" in the sample config files.
- "**basetype**" can be one of the following:
  - "**vcard**": The type is a vCard. Note that a vCard could also be defined with basetype "mimedir" (as it is a MIME-DIR based format), however using the "vcard" base type allows the server to optimize some vcard specific things.
  - "**vcalendar**": The type is a vCalendar.
  - "**mimedir**": The type is a generic MIME-DIR type. This basetype is intended for defining custom formats which are not vCard nor vCalendar.
  - "**text**": The type is a text-based format, such as notes or email. **The <use> tag should reference a <textprofile> (new in version 3.0)** containing <linemap> (see 10.4).For compatibility with version 2.1, the <linemap>s can be specified in the <datatype> directly, but you will get warnings when starting the server, recommending moving <linemap> to a separate <textprofile>.

10.5.1 <use>: MIME-DIR profile, text profile or field list to use for datatype

Contained in: <datatype>
Can contain: nothing
Attributes: for basetypes "mimedir", "vcard", "vcalendar": mimeprofile
for basetype "text": textprofile (or fieldlist for compatibility with version 2.1)

This tag is used to define the MIME-DIR profile (for basetypes "mimedir", "vcard" and "vcalendar"), textprofile or fieldlist (for basetype "text") the datatype is based on.

The attribute must specify a name of a previously defined <mimedirprofile> (see 10.3), <textprofile> (see 10.4) or <fieldlist> (see 10.1) resp.

Examples:

```
<datatype name="vcard21" basetype="vcard">
  <version>2.1</version>
  <use mimeprofile="vcard"/>
</datatype>
```

```
<datatype name="note" basetype="text">
  <use fieldlist="note"/>
  <typestring>text/plain</typestring>
  <versionstring>1.0</versionstring>
  <linemap field="SUBJECT">
    <numlines>1</numlines>
    <inheader>false</inheader>
    <allowempty>true</allowempty>
  </linemap>
  <linemap field="TEXT">
    <numlines>0</numlines>
    <inheader>false</inheader>
    <allowempty>true</allowempty>
  </linemap>
</datatype>
```

10.5.2 <version>: vCard or vCalendar version

Contained in: <datatype basetype="vcard"> or <datatype basetype="vcalendar">
Can contain: vCard or vCalendar version
Attributes: none

This tag is used to define the vCard or vCalendar version. It controls how data is interpreted and generated (as this is different between versions) as well as how the type is named in SyncML communication (for example, the correct name for vCard 2.1 is "text/x-vcard", while for vCard 3.0 it is "text/vcard").

For vCard, versions available are "2.1" and "3.0".
For vCalendar, versions available are "1.0" and "2.0".

10.5.3 <typestring>, <versionstring>: MIME type and version

Contained in: <datatype basetype="mimedir">
Can contain: MIME type and version strings, resp.
Attributes: none

These tags are used to define a MIME type and a version for <datatype> which base on the generic "mimedir" type. This should be used only for custom types.

Example of a custom type definition:

```
<datatype name="mytype" basetype="mimedir">
  <typestring>text/x-mytype</typestring>
  <versionstring>1.7</versionstring>
  <use mimeprofile="myprofile"/>
</datatype>
```

10.5.4 <zippedbindata>: Enable/disable special compressed (non-standard) item format

Contained in: <datatype>
Available: might not be available in all standard products
Can contain: boolean value
Attributes: none
Default: false

If this flag is set to true for a <datatype> definition, it enables a (non-standard) per-item data compression as follows: The raw data (for example a vCard) will be compressed with the zip compressor (compression level can be set using <zipcompressionlevel>, see 10.5.5) and the resulting binary data will be sent as "bin" opaque WBXML data. The <item> <meta> will contain <format> set to <bin> and will have a <MaxObjSize> tag which contains the original (uncompressed) size of the item data. This will happen if and only if the SyncML communication is in WBXML format and if the SyncML Version is 1.1 or higher.
Note: This allows to define special types (in addition to the standard data types) to optimize network bandwith usage, such as for large emails. Of course, this does only work if both ends (server and client) support this **non-standard** compression scheme. Synthesis SyncML client for Windows Mobile for example support this kind of compressed format for emails as "application/x-zip-message" custom type, and therefore the sample config supports this format as well, in addition to the standard "text/message".

10.5.5 <zipcompressionlevel>: Compression level for <zippedbindata> compression

Contained in: <datatype>
Available: might not be available in all standard products
Can contain: number 0..9
Attributes: none
Default: empty

This tag specifies the compression level (0=no compression, 1=low compression level, fast, 9=high compression level, slow) used when the <zippedbindata> feature is enabled (see 10.5.4).

10.5.6 <binaryparts>: Allow unencoded binary in content

Contained in: <datatype>
Available: might not be available in all standard products
Can contain: boolean value
Attributes: none
Default: false

If this flag is set to true for a <datatype> definition, it enables a (non-standard) format extension for certain content formats (RFC 2822 email attachments at this time) to allow more efficient bandwidth usage than base 64 encoding by including binary parts as such (plain binary). This takes place with WBXML format only – XML cannot transfer unencoded binary.

Synthesis SyncML client for Windows Mobile support binary parts for email attachments in the "application/x-zip-message" (1.1) custom type, and therefore the sample config supports this as well.

10.5.7 <unicodedata>, <bigendian>: Unicode content

Contained in: <datatype>
Available: might not be available in all standard products
Can contain: boolean value
Attributes: none
Default: false
New in: 3.0.2.0

If <unicodedata> is set, the content of the <data> parts is interpreted as Unicode (16-bit characters) rather than UTF-8 which is the standard encoding for SyncML. <bigendian> can be set in case the Unicode content is in Big Endian order (Motorola style, MSB first) rather than Little Endian order (Intel style, LSB first).
Note that this makes only sense in WBXML. There are some clients on the market that use unicode encoded data in special, semi-proprietary formats like Nokia's VMSG (for SMS sync).

10.5.8 <initscript>: Initialisation of type-specific script context

Contained in: <datatype>
Available: in PRO versions only
Can contain: script
Script context: datatype context
Attributes: none
Default: no script

This script is executed once before the datatype is first used for receiving or sending data. It can be used to initialize the datatype context. Note that because it is possible for a SyncML session to use different datatypes for sending and receiving (for example vCard 2.1 and vCard 3.0), the context for sending items is not necessarily the same as for receiving items - although it will be in most cases.

<initscript> is the place to declare and initialize variables that are used for processing incoming and outgoing items in <incomingscript>, <outgoingscript> (see 10.5.9), <filterinitscript>, <filterscript> (see 0), <processitemscript> (see 10.5.11), <comparescript> (see 10.5.12) and <mergescript> (see 10.5.13)

10.5.9 <incomingscript>, <outgoingscript>: Custom pre- and postprocessing items

Contained in: <datatype>
Available: in PRO versions only
Can contain: script
Script context: datatype context
Attributes: none
Default: no script

These scripts are executed for every data item just after decoding the incoming data (<incomingscript>) and just before encoding the outgoing data (<outgoingscript>) according to the encoding/decoding defined for the <datatype>.

These scripts are the place to customize the endcoding/decoding process between internal field list representation and the SyncML datatype. Note that this is *not* the place to implement database specific conversions, because this is better done in the <beforewritescript> and <afterreadscript> in the <datastore> section (see 12.20).

10.5.10 <filterinitscript>, <filterscript>: Script-based data filtering

Contained in: <datatype>
Available: in PRO versions only
Can contain: script returning boolean value
Script context: datatype context
Attributes: none
Default: no script

With these two scripts, custom filtering that cannot be represented with the standard flters (see 7) can be implemented. For example, for making the special date range limit (see 7.4) work for a datatype, an appropriate <filterinitscript> and <filterscript> must be implemented (see example below).

The <filterinitscript> is executed once per sync session. It must analyze the filter parameters (such as STARTDATE(), ENDDATE() and SIZELIMIT(), see below) and then return TRUE if the <filterscript> must be called for every item modified or added in the database since the last sync session. In addition, if the <filterscript> must be called for *all* items in the sync set (not only those that have changed or were added), the <filterinitscript> must use SETFILTERALL(TRUE). Note that this "filter all" mode is required for filter conditions that are dynamic, that is, which change from sync session to sync session (like a "10 days before and 30 day after" rule, which affects a different date range every day. For static conditions (like "only tasks that are not completed"), it is sufficient to have only the new and changed records filtered, which can make a huge performance difference (with SETFILTERALL(TRUE), *all records of the database must be read and checked*, which might need quite some processing for large databases. If <filterinitscript> returns FALSE, this means that no filtering is needed which is the most efficient case.

The <filterscript> is executed once per data item read from the database. It must return TRUE if the data item passes the filter and FALSE otherwise.

The following functions are available in the datatype context only:

SETFILTERALL(integer *all*): For <filterinitscript>. If *all* is TRUE, this causes all records of the datastore being passed through the <filterscript>.

integer SIZELIMIT(): returns the size limit that was set for the item being processed. This limit defaults to the size limit set for the datastore which can be obtained by the DEFAULTSIZELIMIT function (see 11.34.21), for example when set with the /li() CGI option (see 7.4). If no size limit was set, SIZELIMIT() returns UNASSIGNED.

SETSIZELIMIT(integer *limit*): sets a new size limit for the item being processed. This can be used to override the default size limit on a item-by-item basis. If *limit* is UNASSIGNED or -1, this disables the limit for the item being processed.

In addition, all datastore context functions as listed in 11.34.21 are also available.

The following example shows a <filterinitscript> and a <filterscript> which offer date-range filtering for events and todos (if inserted into a vCalendar-based <datatype>).

```
<filterinitscript><![CDATA[
  // check if we need to filter
  INTEGER NEEDFILTER;

  // we need to filter if STARTDATE() and/or ENDDATE() are set...
  NEEDFILTER =
    !DBHANDLESOPTS() && // ...but only if DB itself does not handle it directly
    (STARTDATE()!=EMPTY || ENDDATE()!=EMPTY); // start or end date set
  SETFILTERALL(NEEDFILTER); // as the filter conditions change dynamically with
                            // time, we need to filter ALL records
  RETURN NEEDFILTER; // we need to filter
]]></filterinitscript>

<filterscript><![CDATA[
  INTEGER PASSES;

  // check if item passes filter
  PASSES=FALSE;
  // as vCalendar handles events and tasks, we need to differentiate here
  IF (KIND=="EVENT") {
    // Events pass if they start within the range or
    // have a recurrence that has started, but not yet ended
    PASSES =
      (DTSTART<=ENDDATE()) &&
      (
        (DTSTART>=STARTDATE()) ||
        ((RR_FREQ!=EMPTY) && ((RR_END==EMPTY) || (RR_END>STARTDATE())))
      );
  }
  ELSE {
    // Todo pass if they have no DUE or one within the range
    PASSES =
      (DUE==EMPTY) ||
      (DUE>=STARTDATE() && DUE<=ENDDATE());
  }
  RETURN PASSES;
]]></filterscript>
```

10.5.11 <processitemscript>: Custom processing for incoming items

Contained in: <datatype>
Available: in PRO versions only
Can contain: script returning boolean value
Script context: datatype context
Attributes: none
Default: no script

This script is executed before an incoming item is processed (that is, stored in the database), but after an eventual <incomingscript>. This is the place to implement non-standard behaviour (such as rejecting certain items or ignoring updates etc.) for item processing. For "standard" datatypes this script is normally not used, but it is needed to implement special-behaviour datatypes such as email.

This script can influence how the item will be processing by using the following context functions (note that the context functions described in 10.5.10 are also available):

ECHOITEM(string *syncop*): This will cause the incoming item being echoed back to the sender with the specified *syncop* (which can be "add", "replace", "delete").

REJECTITEM(integer *statuscode*): This will cause the incoming item being rejected with the *statuscode*. When*statuscode* is set to zero, the item will be silently rejected, without returning a error code.

string SYNCOP(): Returns the sync operation requested for the current item ("add", "replace" or "delete".

string LOCALID(): Returns the current item's local ID.

SETLOCALID(string *localid*): Sets the current item's local ID. Note that this is only required in very special cases. Usually, modifying the local ID will lead to messed up sync sessions.

string REMOTEID(): Returns the current item's remote ID.

SETREMOTEID(string *remoteid*): Sets the current item's remote ID. Note that this is only required in very special cases, such as clients requiring extra information (such as an email folder path) embedded in the remote ID string.

CONFLICTSTRATEGY(string *strategy*): Sets the conflict strategy ("duplicate","newer-wins", "server-wins", "client-wins") for the current item and overrides the default strategy set by <conflictstrategy>, <slowsyncstrategy> and <firsttimestrategy> (see 11.34.10) or by SETCONFLICTSTRATEGY() in <datastoreinitscript> (see 11.34.21).

FORCECONFLICT(): Forces a conflict even if there is none. This means that the item coming from the remote party is made conflict the item already in the database (if any) and have it being processed by the normal conflict resolution procedure.

DELETEWINS(): Normally, when a delete and a replace operation conflict, the replace always wins to preserve data (unless datastore-wide <deletewins> option is set, see 11.34.5). When DELETEWINS() is used, this makes delete win over replace.

PREVENTADD(): Prevent that the item is added to the datastore. Note that just checking SYNCOP() for "add" is not enough, as clients are free to send new items using the "replace" command, which will cause an implicit add if the item does not already exist. PREVENTADD() prevents this implicit add as well.

IGNOREUPDATE(): Causes replace operations to be ignored. Note that just checking SYNCOP() for "replace" is not enough, as clients are free to send new items using the "replace" command, causing implicit adds in the server. IGNOREUPDATE() makes sure only "replace" operations are executed that cause an implicit add.

10.5.12 <comparescript>: Custom item comparison

Contained in: <datatype>
Available: in PRO versions only
Can contain: script returning 0 if *target* equals *reference*, 1 if *target* is newer than *reference*, -1 if *target* is older than *reference*,-999 if items are not comparable or not equal but age of items is not known.
Script context: datatype context
Attributes: none
Default: no script

This script can be used to implement customized comparison rules. Comparison is important when trying to match existing items from client and server in slow sync, or when a conflict occurs. The fields of the two items to be compared can be accessed as field variables with the prefix *target* or *reference* (as described in 6.9.3)

The following special context function is available for comparison (note that the context functions described in 10.5.10 are also available):

integer COMPAREFIELDS(): This performs the default comparison (according to the compare rules defined in the <fieldlist> for the fields). It returns 0 if *target* equals *reference*, -1 if *target* is older than *reference*, 1 if *target* is newer than *reference* and -999 if *target* and *reference* cannot be compared at all or are not equal and no age of the items is known.

The following example shows how to implement a relaxed comparison for vCalendar (only comparing dates, not times to avoid that time zone shift problems will generate duplicates in slow syncs):

```
<comparescript><![CDATA[
  INTEGER RES;

  // do standard compare (WHICH MUST NOT
  // INCLUDE DTSTART/DTEND!)
  RES = COMPAREFIELDS();
  // do specially relaxed date checking
  if (RES==0) {
    // only check for same local date (not time)
    // of DTSTART(=DUE for vTODO)
    // to increase probability that time zone problems
    // get masked out
    RES =
      COMPARE(
```

```
          DATEONLY(RELATIVEASUTC(TARGET.DTSTART)),
          DATEONLY(RELATIVEASUTC(REFERENCE.DTSTART))
        );
    }
    RETURN RES;
]]></comparescript>
```

10.5.13 <mergescript>: Custom item merge

Contained in: <datatype>
Available: in PRO versions only
Can contain: script
Script context: datatype context
Attributes: none
Default: no script

This script can be used to implement custom merge algorithms. Merge is used when resolving conflicts, that is when two items need to be updated from each other to both contain all relevant data. The fields of the two items to be merged can be accessed as field variables with the prefix *winning* (for the item that wins the conflict, that is, contains the "better" or "never" data) or *loosing* (for the item that looses the conflict), as described in 6.9.3.

After the execution of <mergescript>, both items (*loosing* and *winning*) should be equal. The <mergescript> **must call the SETWINNINGCHANGED()** and **SETLOOSINGCHANGED()** functions (see below) when it modifies one of the items - this is required by the SyncML engine to know if corresponding updates must be sent to the remote party or applied to the local database.

The following special context functions are available for comparison (note that the context functions described in 10.5.10 are also available):

integer MERGEFIELDS(): This performs the default merge (according to the merge options defined in the <fieldlist> for the fields). Calling this function also updates the **WINNINGCHANGED()** and **LOOSINGCHANGED()** (see below) flags to show if either of the items were modified.

integer WINNINGCHANGED() and **integer LOOSINGCHANGED()**: return TRUE if the corresponding (*winning* or *loosing*) item is flagged changed (which will cause it to be sent to the local database or the remote party for update).

SETWINNINGCHANGED(integer changed) and

integer SETLOOSINGCHANGED(integer *changed*): used to set the change flags of the *winning* or *loosing* item, resp.

10.5.14 <mimedirmode>: MIME-DIR conformance

Contained in: <datatype basetype="mimedir">
Can contain: "old" or "standard"
Attributes: none

This tag is used to define the behaviour of a <datatype> which bases on the generic "mimedir" type. The following values can be specified:

- **"standard"** : The type behaves according to MIME-DIR (folding of long lines, encoding of multi-line fields, escaping of characters)
- **"old"**: The type behaves like in the pre-MIME-DIR formats vCard 2.1 and vCalendar 1.0. Using this setting makes a type not to conform to MIME-DIR specifications, so it should be used with care.

10.6 RRULE field block

Datatypes based on "vcalendar" can make use of the "rrule" conversion mode (see 10.3.4), which requires a special block of subsequent fields in the field list as follows:

1. String field, containing 2 chars to code the type of recurrence:
   - First character is the frequency code: (0 = none, D = daily, W = weekly, M = monthly, Y = yearly)
   - Second char is the frequency modifier (space = none, W = by weekday list, D = by monthday list, M = by monthlist)
2. Integer field, containing the interval (expressed as number of units specified in the frequency code)
3. Integer field, containing a bit mask which codes the frequency modifications as follows:
   - for weekly: Bit0=Sun, Bit6=Sat
   - for monthly by weekday: Bit0=first Sun, Bit7=2nd Sun... Bit35=5th Sun
   - for monthly by day: Bit0=1st, Bit1=2nd, Bit31=31st.
   - for yearly by month: Bit0=jan, Bit1=feb,...
4. Integer field, containing a bit mask like the previous field, but coding ocurrence from the end of the interval, not the beginning. This makes sense only as follows:
   - for monthly by weekday: Bit0=last Sun, Bit7=2nd last Sun...
   - for monthly by day: Bit0=last, Bit1=2nd last...
5. Timestamp field, containing the date/time when the recurrence ends. If this is empty, the recurrence has no end and repeats forever.
6. Timestamp field, containing the start of the entire event (this is normally the field used for DTSTART property). Note that this field will never be written by the RRULE conversion, but is only used as reference context for parsing/generating RRULEs.

Note that this scheme covers most, but not all options possible in RRULEs. However, it covers all of the RRULE implementations commonly used in real-world applications. Please also note that not all combinations of frequency code and frequency modifier are valid.

See sample config files to see how a RRULE block is defined and used.

11. <server>, <client>: General Server and Client Settings

Contained in: <sysync_config>
Can contain: tags described in this chapter (see also 12 for datastore-specific settings and 16 specifically for clients)
Attributes: type

This tag encloses all server (or client) database related information. This includes three kinds of information:

- **general (Database type independent) configuration tags**. These are described in this chapter and can be used in any type of <server> or <client> tag.
- **database type specific configuration**. These are described inchapter 12 for SQL/ODBC or SQLite based servers and clients and in chapter 14 for the plugin based servers and clients. Note that the version 3.0 DEMO server is now a plugin-based configuration (see 14) with a single, built-in text-file plugin (see 14.3).
- **For clients built on the Synthesis SyncML client engine library** a few special configuration directives exist which are described in chapter 15.
- **For command line clients only, configuration of the Sync Request** (information about the SyncML server to be contacted and synchronized with). These client-specific tags are described in 16.

The "type" attribute is required. It specifies the type of the server database.

- For the SQL/SQLite/ODBC type must be set to "odbc" or "sql".
- For the plugin version (which includes the DEMO server), type must be set to "plugin".

Other values are reserved for future versions of the server with DB interfaces other than SQL/SQLite/ODBC or plugin)

11.1 <maxsyncmlversion>,<minsyncmlversion>: SyncML version support

Contained in: <server>
Can contain: SyncML version string, currently: "1.2", "1.1" or "1.0"
Attributes: none
Default: all versions implemented are supported

These tags specify what SyncML versions the server or client should support. Normally, the server allows clients to connect with all known SyncML versions (1.2, 1.1 and 1.0 at this time) and the client tries all SyncML versions when connecting to a server. For special testing it might be useful to limit the range of SyncML versions accepted. For clients, also see <defaultsyncmlversion> in 16.1.

11.2 <sessiontimeout>: Timeout for unfinished sessions

Contained in: <server>
Can contain: integer value, specifying timeout in seconds
Attributes: none
Default: 300 seconds (5 minutes)

This tag specifies how long (minimally) a started, but not finished sync session will be kept alive in the server's memory. After this time, the server might abort the session to free ressources (depending on the platform and version, the server might however keep the session alive longer if there is no need to free resources).

11.3 <requestmaxtime>: max time for request processing

Contained in: <server>
Can contain: max duration (in seconds) of a single request processing, 0=no limit
Attributes: none
Default: 0 (=no minimum)

This can be used to specify how quickly a client request should be processed by the server. This can be important if the server database is slow or the number of synced data items is large. In these cases, request processing could take so much time that the SyncML client would time out and abort the session before the server sends an answer.
With <requestmaxtime> it is possible to indicate to the SyncML server engine how long processing of a single request may take. If normal sequential processing would exceed the specified amount of time, the server takes measures to answer the request in time such as postponing execution of some commands or executing them in the background in a separate thread.

Note that <requestmaxtime> cannot always be met exactly - some datastore implementations or server platforms might not allow multi-threaded execution, or might have uninterruptable operations that could prevent <requestmaxtime> from being met.

By using <requestmaxtime> in a <remoterule> (see 11.36.23) it is possible to specify different request processing times on a device by device basis - however, as it takes one or two message exchanges until <remoterule>s can be evaluated and applied, make sure that <requestmaxtime> in the <session> is set to a reasonable time. On the other hand, operations performed before <remoterule>s can be applied are almost never time consuming (usually only a database login).

Finally, using the REQUESTMAXTIME() script function (see 6.14.6), maximum request time can also be controlled from scripts.

11.4 <requestmintime>: artifical slow down

Contained in: <server>
Can contain: minimal duration (in seconds) of a single request processing, 0=no minimum
Attributes: none
Default: 0 (=no minimum)

This is only intended for debugging purposes. Setting this to non-zero causes the server to wait for each response before at least the specified number of seconds have passed since the last re-

quest arrived. This can help to slow down sync sessions such that testing things like manually aborting sessions in the middle, disconnecting network etc. gets possible. Using the REQUESTMINTIME() script function (see 6.14.6), such slow-down behaviour can also be applied to certain sessions only (like depending on user options).

11.5 <externalurl>: specify URL used to access the server

Contained in: <server>
Can contain: fully specified URL
Attributes: none
Available: server only
Default: not set

This tag is used to set the URL which is used by external clients to access the server. **Note that this is normally neither required nor recommended, so please use this option with care in special situations only (exotic proxy or redirection situations).** If this option is set, the server will use this hard-coded URL in its <Source><LocURI> tags rather than the URL which was used by the client to address the server in the first place.

11.6 <requestedauth>,<requiredauth>: SyncML Authentication

Contained in: <server>
Can contain: SyncML authentication method name
Attributes: none
Default: md5

NOTE: in Server versions before 1.0.5.3 the <requiredauth> tag was misspelt as <requireauth>. So if you are upgrading from an older server, this spelling error should be corrected in your existing config file.
These tags specify what type of server-level authentication the SyncML server requests (or requires minimally, resp.) from a SyncML client that want to connect. In most but very special test cases both tags should have the same value.

The following values are allowed:

- "**none**" : the server does not require the client to authenticate itself. This is not a common case, as on a multi-user server authentication is needed to login to the user's data.
- "**basic**" : the client can use the SyncML "basic" authentication method (or md5, see below). This is not recommended generally as in this method the password is sent unencrypted over the net.
- "**md5**": the client must use the SyncML "md5" method. MD5 is an algorithm that encrypts the password in a way that is cannot be recovered from the data that travels over the net. This is the recommended settings, because all compliant SyncML clients are required to support this anyway.

If your database does store passwords in MD5-digest form (see "12.13"), you must set <requiredauth> to "basic", as it is not possible with SyncML 1.0 to check MD5 digests (it will be possible with SyncML 1.1 and later).
Note that <requestedauth> should always be set to the same or higher "strength" method than <requiredauth> (none < basic < md5 in terms of "strength"). Otherwise the server would request something "weaker" from the client than what it actually requires for authentication - which makes no sense and will block client login.

11.7 <autononce>: MD5 nonce generation mode

Contained in: <server>
Can contain: boolean value
Attributes: none
Default: on

This is a special option that controls the behaviour of MD5 authentication. It should be set to on under normal circumstances. The technical background is that the MD5 authentication scheme uses a so-called nonce string to increase security. The nonce string is generated by the server. When autononce is on, this happens automatically. When autononce is off, the server does not send a nonce string or uses a constant string that can be specified with the <constantnonce> tag (see below). This compromises security, but might be useful for testing or other special circumstances.

11.8 <constantnonce>: constant nonce string

Contained in: <server>
Can contain: nonce string
Attributes: none
Default: empty

This tag should be used when <autononce> is not set to specify a constant nonce string to be used in MD5 authentication.

11.9 <sendrespuri>, <respurionlywhendifferent>: RespURI configuration

Contained in: <server>
Can contain: boolean value
Default: true (for both options)

<sendrespuri> controls if the server should create a RespURI including a sessionID CGI string or not. Default is true, because for HTTP servers this is required for session tracking.
For transports which have a transport-level session context, this may not be required so it can be turned off.
By default, the RespURI is only generated when it is different from the URI used for the current request. <respurionlywhendifferent> can be set to false to always generate a RespURI.

11.10 <simpleauthuser>, <simpleauthpw>: single user mode

Contained in: <server>
Can contain: username, password
Attributes: none
Default: not specified

For single-user situations, it might be that the overhead of having a separate table for authenticating users is not desired. For this special case, these two tags allow to provide a single username and password. A SyncML client trying to authenticate with a matching user/password combination will be granted access. In this case, the <userkeysql> (see "12.13") will never be executed, and therefore there will be no userkey value that can be used as parameter (%U) in subsequent queries. **Do not use these tags unless you have a true single-user situation.**

Example:

```
<simpleauthuser>test</simpleauthuser>
<simpleauthpw>test</simpleauthpw>
```

11.11 <multithread>: Allow multi-threaded execution

Contained in: <server>
Can contain: boolean value
Default: false (for Linux) / true (for all other operating systems)

Allows multi-threaded execution for the server, if set to true.
As there are problems in early 2.4 Linux kernels, multi-threading was switched off completely for Linux in the past. For compatibility reasons the default value is set accordingly.
When <multithread> is on, the server uses a separate thread for each datastore to load the sync set, as this can take a long time for large sync sets and the server needs to continue communicating with the client to prevent the client to time out.

11.12 <sessioninitscript>: Session init script

Contained in: <server>, <client>
Available: in PRO versions only
Can contain: script
Script context: session context
Attributes: none
Default: no script

This script is executed whenever a new session is started. It can be used to declare and initialize variables in the session script context (which can be accessed from other script contexts using the SESSIONVAR and SETSESSIONVAR built-in functions, see 6.14.6).

11.13 <sessionfinishscript>: Session finish script

Contained in: <server>, <client>
Available: in PRO versions only
Can contain: script
Script context: session context
Attributes: none
Default: no script

This script is executed whenever a session (successfully) comes to its end. All variables of the session script context can be used. See also <sessioninitscript> for details.

11.14 <sentitemstatusscript>, <receiveditemstatusscript>: Session level status code handling

Contained in: <server>, <client>
Available: in PRO versions only
Can contain: script
Script context: session context
Attributes: none
Default: no script

These scripts can be inserted in the session level as well as in the datastore level. See 11.34.27 and 11.34.28 for details.

11.15 <customgetputscript>, <customendputscript>: Creation of custom SyncML Get and Put commands

Contained in: <server>, <client>
Available: in PRO versions only
Can contain: script
Script context: session context
Attributes: none
Default: no script

This allows issuing custom SyncML <Get> and <Put> commands - custom <Alert> commands, such as Alert 100 to display a message to the user) are possible as well.

The <customgetputscript> is executed at the beginning of the session when <Get> and <Put> for device information are sent. The <customendputscript> is executed at the end of the session to allow sending custom session summary or status information to the remote. For example, a Alert 100 could transmit information about conflicts etc. to the client.

The script must return TRUE if it has handled the item. If it returns nothing or FALSE, other handlers (such as the default handler for devInf) are checked.

The following functions are available to these scripts:

SETSTATUS(integer *status*): Sets the *status* code to be returned for the get/put/result command. If no status is set, the engine assumes the command execution was successful.

string ITEMURI(): Returns the Target LocURI of the item. This should be checked for known URIs to decide if the script should handle the command or not.

SETITEMURI(string *locURI*): Sets the Target *locURI* of the item for sending a command.

string ITEMDATA(): Returns the <Data> of the item.

SETITEMDATA(string *itemData*): Sets the <Data> of the item to the string *itemData*.

string METATYPE(): Returns the <Meta><Type> information of the item.

SETMETATYPE(string *metaType*): Sets the <Meta><Type> information of the item to the string *metaType*.

ISSUEPUT(boolean *allowfailure*, boolean *noresp*): Creates and issues a <Put> command using the values set by SETITEMURI(), SETMETATYPE() and SETITEMDATA(). If *allowfailure* is set, the status returned for the <Put> by the remote is ignored and will not cause the session to abort if it is not ok. If *noresp* is set, the remote is told not to respond to this command.

ISSUEGET(boolean *allowfailure*): Creates and issues a <Get> command using the values set by SETITEMURI() and SETMETATYPE(). If *allowfailure* is set, the status returned for the <Get> by the remote is ignored and will not cause the session to abort if it is not ok.

ISSUEALERT(boolean *allowfailure*, integer *alertcode*): Creates and issues a <Alert> command with *alertcode* and one item containing the string set by SETITEMDATA(). If *allowfailure* is set, the status returned for the <Alert> by the remote is ignored and will not cause the session to abort if it is not ok.

11.16 <customgethandlerscript>: Custom handling of SyncML Get commands

Contained in: <server>, <client>
Available: in PRO versions only
Can contain: script
Script context: session context
Attributes: none
Default: no script

This allows specifying a script to handle items of SyncML <Get> commands. This can be useful for implementing custom functionality.

The script must return TRUE if it handles the command and produces any data to be returned as <Result> to the sender of the <Get> command. If it returns nothing or FALSE, other handlers (such as the default handler for devInf) are checked.

The same functions as in <customgetputscript> (see 11.15) are available to this script.

11.17 <customputresulthandlerscript>: Custom handling of SyncML Put/Result commands

Contained in: <server>, <client>
Available: in PRO versions only
Can contain: script
Script context: session context
Attributes: none
Default: no script

This allows specifying a script to handle items of SyncML <Put> and <Result> commands. This can be useful for implementing custom functionality.

The script must return TRUE if it has handled the item. If it returns nothing or FALSE, other handlers (such as the default handler for devInf) are checked.

In addition to the functions available in <customgetputscript> (see 11.15), the following special function is available in <customputresulthandlerscript>:

boolean ISPUT (): Returns TRUE if the command that called the script is a SyncML <Put> command, false if it is a SyncML <Result> command.

11.18 <waitforstatusofinterrupted>: SyncML command flow option

Contained in: <server>, <client>
Can contain: boolean value
Attributes: none
Default: no

This is a highly technical setting. If set to yes, a interrupted command (for example a <sync> command that had to be terminated to fit into one message and needs to be continued in the next message) will not be continued until the status for the previous part has arrived.

11.19 <relyonearlymaps>: Add resending policy

Contained in: <server>
Can contain: boolean value
Attributes: none
Default: yes

This is a highly technical setting. If set to no, the server does not re-send <Add> commands again until a subsequent session has successfully completed. This is to avoid duplicates for clients that do not reliably re-send unconfirmed <Map> items left over from the previous session at the beginning of the next session. For properly SyncML conformant clients, this can be set to yes (the default).

11.20 <debugchunkmaxsize>: LargeObject chunk size limit for testing

Contained in: <server>, <client>
Can contain: max size of chunk for LargeObject in bytes
Attributes: none
Default: 0 (disabled)

This is a highly technical setting. If set to non-zero, all objects larger than the specified number of bytes will be sent in chunks of no more than the specified size. The only use of this is to test LargeObjects implementation in SyncML DS 1.1 and 1.2 remote parties. Usually, the <MaxMsgSize> reported by the remote party determines when objects need to be sent in multiple chunks.

11.21 <deletinggoneok>: Handling of delete for non-existing items

Contained in: <server>, <client>
Can contain: boolean value
Attributes: none
Default: yes

This is a highly technical setting. If set to yes, receiving a delete command for an item that does not exist is not considered an error. However, some implementation (especially the SCTS test tool) requires an error message for that, hence this option.

11.22 <usertimezone>: Set user's default time zone

Contained in: <server>, <client>
Can contain: time zone name (see list of all time zones in 17)
Attributes: none
Default: system time zone

This defines the user's default time zone. This is important for synchronizing with client that only support local time specifications (rather than UTC). For installations supporting multiple user time zones, the USERTIMEZONE() script function (see 6.14.4) can be used to set the time zone specifically for a sync session, usually derived from the user account.

11.23 <autoenddateinclusive>: end date for allday events inclusive

Contained in: <server>, <client>
Can contain: boolean value
Attributes: none
Default: true for clients, false for servers.

If this option is set, date values with conversion mode "autoenddate" (see 10.3.4).will be rendered as 23:59:59 of the previous day in old formats like vCalendar 1.0. If it is not set, "autoenddate"s will be rendered as 0:00 of the next day (which usually is the first day not included in the all-day event when "autoenddate" is used for DTEND properties).
Note that this setting can be modified on a per-device basis using a remote rule (see 11.36.18)

11.24 <abortonallitemsfailed>: error handling option

Contained in: <server>, <client>
Can contain: boolean value
Attributes: none
Default: yes

This determines if receiving an error for all items sent *in a slow sync* to the remote party is considered a complete failure of the entire sync for that datastore, or if the failing items are simply marked for being re-tried in the next sync. (Note that before 3.1.x, this applied to normal sync as well – which is no longer the case as in a normal sync with few upates a failure of these usually does not indicate a general problem).

11.25 <showctcapproperties>: show field support details in device information

Contained in: <server>, <client>
Can contain: boolean value
Attributes: none
Default: yes

This determines if the list of supported fields should be listed in the device information sent to the remote party. Normally, this should be switched on – but sometimes if a device does not send all fields one would expect it should send, disabling sending the field information might help as then the device cannot do (possibly wrong) decisions on the fields it should send based on the device information. This can help to debug IOT problems with such devices. Note that there is a scripting function SHOWCTCAPPROPERTIES (see 6.14.6) that allows switching this option on and off on a session by session base (for example in the <logininitscript>, see 11.33, maybe depending on AUTHDEVICEID()).

11.26 <showtypesizeinctcap10>: show size and type in SyncML 1.0 devInf

Contained in: <server>, <client>
Can contain: boolean value
Attributes: none
Default: no

This determines if in the list of supported fields, the type and (max)size information should be included when sending devInf to SyncML 1.0 devices. By default, this is set to "no", because even though type and (max)size is valid information for SyncML 1.0, some of these (old) client

implementations cannot handle it and would crash. This option can be set to "yes" in the unlikely case the type and (max)size information should be included for SyncML 1.0 clients.
SyncML 1.1 and later clients are not affected (and always receive type and (max)size information when available server side).

11.27 <enumdefaultpropparams>: enumerate default property parameter's values as property names

Contained in: <server>, <client>
Can contain: true, false, default
Attributes: none
Default: default (automatic)

This determines how the default parameter of a MIME-DIR property (for example TYPE for TEL in vCard 2.1) is shown in the device information. If it is set to true, all possible values for the parameter are enumerated as parameter *names* (*propParam* tag). This is required e.g. by some Nokia DS 1.1 clients. If it is set to false, the possible parameter values are shown as *valEnum* tags, as required by DS 1.2 clients.
If set to "default", the format is automatically chosen depending on the SyncML version of the current session.
Except for very special cases, usually in debugging new clients, this setting should be left at its default.
Note that there is a scripting function ENUMDEFAULTPROPPARAMS (see 6.14.6) that allows switching this option on and off on a session by session base (for example in the <login-initscript>, see 11.33, maybe depending on AUTHDEVICEID()).

11.28 <acceptserveralerted>: Acceptance of server alerted sync types

Contained in: <server>, <client>
Can contain: boolean value
Attributes: none
Default: yes

If yes, "server alerted" sync types are accepted by the server. They operate exactly like the normal client-initiated sync types, but indicate that the sync was triggered by the server (for example via SAN using the Synthesis PushAlerter tool). If for some reason a server should generally reject server alerted syncs, this can be set to false.

11.29 <logfile>: Activity log text file

Contained in: <server>, <client>
Can contain: complete path for log file
Attributes: none
Default: none

This tag can contain a path to a text file (which must be writable). For each datastore involved in a sync session, a log entry as specified with <logformat> (see 11.31) will be appended to this file (usually a single line). Note that in addition to this logfile, you can also specify a SQL statement to store log information in a SQL database table (see 12.19).

11.30 <logenabled>: Activity log enable

Contained in: <server>, <client>
Can contain: boolean value
Attributes: none
Default: true

This tag enables or disables recording activity log information in the <logfile> text file (see 11.21), in the SQL database table (see 12.19) and in a eventual database adapter plugin (see 14.2). Note that this setting is used as the default setting for all sessions, but using the SETLOG() function in <logininitscript> or <logincheckscript> (see 11.33 and 12.17) activity log can be enabled or disabled on a per-session basis (for example, prevent logging specific test users etc.).

11.31 <logformat>: Activity log format

Contained in: <server>, <client>
Can contain: string (C-string)
Attributes: none
Default: standard logfile format, depends on server/client version

This string is used to format log entries. It is a string in the C language syntax (see 3.4),

To actually insert useful log information, the following sequences can be used:

%T Time of sync as plain text (synonymous with %seT)

%ssT Sync start time in plain text (when this sync attempt, successful or not, has started)

%seT Sync end time in plain text (when this sync attempt, successful or not, has ended)

%nD Datastore name (the name as defined in <datastore name="*name*">, see 11.34)

%rD Datastore remote path (identification of the remote party's datastore)

%lD Datastore local path (how the remote party has addressed the local datastore, this might include subfolder information and CGI options)

%iR Remote Device ID (URI of the remote device, usually a IMEI number for phones)

%nR Remote Device name (if a <remoterule> is applied which has a <descriptivename> part, this shows the <descriptivename> instead of the real name sent by the device. See 11.36 for details)

%vR Remote Device version information as provided by the device's device information in the following format:
DeviceType (HardwareVersion, FirmwareVersion, SoftwareVersion) OEMname

%U User name

%sS Status code for this datastore (0 if successful, 408 for timeout, all other errors are SyncML status codes). Note that other datastores in the same sync session might have different status.

%ssS Session status code (0 if successful, 408 for timeout). Note that the session status can be successful even if some datastores have non-successful status.

%syV SyncML protocol version string ("1.2", "1.1" or "1.0").

%syVn Numeric code for SyncML protocol version: 1=1.0, 2=1.1, 3=1.2

%mS Sync Mode (0=twoway, 1=fromclient 2=fromserver)

%tS Synctype (0=normal,1=slow,2=firsttime slow, 10=normal resumed, 11=slowsync resumed, 12=first time resumed)

%laI number of locally added Items

%raI number of remotely added Items

%ldI number of locally deleted Items

%rdI number of remotely deleted Items

%luI number of locally updated Items

%ruI number of remotely updated Items

%leI number of locally failed Items

%reI number of remotely failed Items

%diB number of incoming (received) content data bytes for this datastore (net content data excluding any SyncML protocol overhead)

%doB number of outgoing (sent) content data bytes for this datastore (net content data excluding any SyncML protocol overhead)

For servers only, the following sequences are also available:

%iS Server Session ID (useful to find a corresponding debug logfile, see 8.11)

%smI number of items matched in slow sync

%scI number of conflicts won by the server

%ccI number of conflicts won by the client

%dcI number of conflicts resolved by duplicating the conflicting item

%tiB session total of incoming (received) bytes (SyncML messages, not including transport protocol's header or other envelope data)

%toB session total of outgoing (sent) bytes (SyncML messages, not including transport protocol's header or other envelope data)

11.32 <loglabels>: Activity log header

Contained in: <server>, <client>
Can contain: text (C-string)
Attributes: none
Default: header for standard logfile format, depends on server/client version

This tag specifies a text (in C-String format, see 3.4) that is written as the first line in the file specified as <logfile>. This can be useful to write a header for a log file in tab separated text form, for example.

11.33 <logininitscript>, <loginfinishscript>: Pre- and post-login scripts

Contained in: <server>
Available: in PRO versions only
Can contain: script returning boolean value
Script context: login context
Attributes: none
Default: no script

The <logininitscript> is executed before testing login details with the database. It can be used to pre-process the user name for example. If this script returns TRUE as result, this means that login is granted without any further database lookup. If the script returns FALSE as result, this means that login is rejected without any further database lookup. If the script does not return a value, this means that futher checks are needed to validate the login data (such as executing <userkeysql>, see 12.16, and <logincheckscript>, see 12.17).

The <loginfinishscript> is executed after database level authorisation is done (regardless of the result) and has the final say about actually allowing or denying access. If this script does not return a value, the result of the database level authorisation decides about granting access. Otherwise, if this script returns TRUE as result, this means that login is granted, otherwise access is denied. The <loginfinishscript> is a good place to apply user-specific parameters (like setting the *user time zone context*, see 5.2).

These scripts have access to the SQL execution functions described in 12.1.4 and the following special script functions:

integer AUTHOK(): returns true if the standard checking thinks that login is ok.

string AUTHUSER(): name of the user that tries to log-in

SETUSERNAME(string *username*): sets the *username* that should be used to perform login checking (can be used in SQL with %U)

SETDOMAIN(string *domain*): Sets the "*domain*", can be used in SQL with %D

string AUTHSTRING(): The auth string sent by the remote (clear text password or MD5 digest)

integer AUTHTYPE(): 0=anonymous (no credentials), 1=password in clear text, 2=SyncML 1.0-type MD5 digest, 3=SyncML 1.1-type MD5 digest.

string AUTHDEVICEID(): The device ID sent by the remote

integer UNKNOWNDEVICE(): returns TRUE if this is the first time this device tries to connect this server.

string USERKEY(): Gets the current userkey (as set by SETUSERKEY() or retrieved by the <userkeysql> query, see 12.16)

SETUSERKEY(string *userkey*): Sets the value that should be used as "*userkey*".

string DEVICEKEY(): Gets the current device key (as set by SETDEVICEKEY() or retrieved by the <getdevicesql> query, see 12.15)

SETDEVICEKEY(string *devicekey*): Sets the value that should be used as "*devicekey*".

integer CHECKAUTH(string *user*, string *secret*, integer *secretismd5*): This function allows implementing completely custom checking of credentials in the <logininitscript>. It checks if the credentials sent by the remote device matches the specified *user* and *secret*. *Secret* must be either the password in clear text (if secretismd5==FALSE) or the b64(md5(user:password)) hash (if secretismd5==TRUE). Usually a custom credential checking involves getting the secret for a user from the local database (EXECSQL etc., see 12.1.4), possibly converting/decrypting it to obtain either a clear-text password or the b64(md5(user:password)) hash. If furthe database accesses depend on a user key, SETUSERKEY() should be used to set it appropriately. Finally, the <logininitscript> would return the result of CHECKAUTH() to accept or reject the login request.

timestamp CONVERTTODATAZONE(timestamp *atime* [,boolean *doUnfloat*]): this returns *atime* converted to the data time zone (the time zone set for the datastore or the session using <datatimezone>, see 11.34.31). If *doUnfloat* is set to true, floating time stamps will be fixed into local time of the data time zone, without changing their time value.

integer TIMESTAMPTODBINT(timestamp *ts*,string *dbfieldtype*): Converts the timestamp *ts*'s value to a integer representation. *Dbfieldtype* specifies the integer representation type to use as a database field type (see 11.34.41.1). Note that not all database field types can be used, but only "lineartime", "lineardate", "unixtime_s", "unixtime_ms", "unixtime_us", "unixdate_s", "unixdate_ms", "unixdate_us".

timestamp DBINTTOTIMESTAMP(integer *dbint*,string *dbfieldtype*): Converts the integer representation of a date/time value *dbint* into a timestamp value. *Dbfieldtype* specifies the integer representation type to use as a database field type (see 11.34.41.1). Note that not all database field types can be used, but only "lineartime", "lineardate", "unixtime_s", "unixtime_ms", "unixtime_us", "unixdate_s", "unixdate_ms", "unixdate_us".

11.34 <datastore>: General Datastore settings

Contained in: <server>
Can contain: <conflictstrategy>, <slowsyncstrategy>, <typesupport> and server-type specific tags, see 12.20.
Attributes: name, type
Default: not specified

This tag specifies all the details for one datastore. A datastore is the SyncML concept for a collection (table) of objects (records) of the same type (e.g. vCard or vCalendar).

A <server> can contain multiple <datastore> sections to support multiple datatypes. Note that the sample config file defines two datastores, one for contacts (vCard) and one for events and tasks (vCalendar).

A datastore must have a *name* attribute, which specifies the name under which the datastore will be accessible from the SyncML client. For example, if the name is set to "mytest", the datastore will be accessible under "./mytest" (some clients allow just "mytest"). Note that the folder concept (see 12.20.1) allows datastores to contain multiple folders that are addressed like "./mytest/foldername".

A datastore can also have a *type* attribute. It has no relevance in SyncML engines that support only one type of datastore (such as textfile or odbc) - which is the case for our current standard products. However, future products and customized products might need the *type* attribute to select among different types of datastores (that is, database interfaces).

Note that this section only covers the settings that are not dependent on the database type. See 12.20 and for <datastore> settings specific to ODBC or for text (demo) datastores.

Example (skeleton for a contact and a task/event datastore):

```
<datastore name="Contact">
   <!-- insert datastore definition here -->
</datastore>

<datastore name="Calendar">
   <!-- insert datastore definition here -->
</datastore>
```

11.34.1 <alias>: alternate name for this datastore

Contained in: <datastore> of <server>
Can contain: alternate data store name
Attributes: none
Default: none

This can be used to make the same datastore accessible with an alternate name. In scripts the LOCALDBNAME() function (see 11.34.21) can be used to find out what name was used by the remote (client) to address the datastore – and possibly behave differently depending on what name was used.

11.34.2 <dbtypeid>: datastore type ID

Contained in: <datastore>
Can contain: 32-bit unique identifier number for datastore, must not be zero
Attributes: none
Default: none

This is required for SyncML engine library based applications. The <dbtypeid> is a 32-bit number which is used to identify the datastore when accessing its user settings (see SDK manual for details about accessing settings via the *settings key* mechanisms). It is also used to identify datastore related progress events.
The <dbtypeid> must be unique for each <datastore> in the config, may not be zero and must not change once assigned (if it changes, related user settings will get detached). Otherwise, you are free to choose any number for this ID.

11.34.3 <displayname>: decriptive name for a datastore

Contained in: <datastore>
Can contain: string
Attributes: none
Default: none

This option allows to set a descriptive name for a datastore. This descriptive text will be transmitted to the remote party and might be used instead of the real name in communication with the end user (to select a database from a list, for example).
In Synthesis SyncML engine library, this string can be queried as "dispName" value.

11.34.4 <readonly>: read-only datastore

Contained in: <datastore>
Can contain: boolean value
Attributes: none
Default: off

This option allows a datastore to be defined as read-only. This will cause that the remote party cannot write to that datastore (attempts to do so will simply be ignored). Note that in PRO versions, it is also possible to switch to read-only mode based on login information; see session-level SETREADONLY() script function in 11.33, and datastore-level SETREADONLY() script function in 11.34.21.

11.34.5 <deletewins>: delete overrides replace

Contained in: <datastore>
Can contain: boolean value
Attributes: none
Default: off
New in: 3.0.2.2

Normally, when a delete and a replace operation conflict, the replace always wins to preserve data. When <deletewins> is set, this makes delete win over replace. Note that in <processitem-script> a function DELETEWINS is available (see 10.5.11) to set delete override on a item-per item basis.

11.34.6 <tryupdatedeleted>: try to update "deleted" items

Contained in: <datastore>
Can contain: boolean value
Attributes: none
Default: off
New in: 3.0.2.2

This is a special option relevant only for the case of a client update conflicting with a server delete (and <deletewins> not set, see 11.34.5). Server deletes can be caused by the item really deleted from the server, but also if a item falls out of a filter (such as a date range). In the latter case, the item still exists, but is no longer visible in the syncset. If the client tries to update such an item, the update will be converted to an add, because that item is not in the syncset. This creates a duplicate in the server database. With <tryupdatedeleted>, before adding the item again, the server will try to update the item. Only if that fails, this means that the item is really deleted, not only filtered out and should be added again.

11.34.7 <reportupdates>: transmit updates to remote

Contained in: <datastore>
Can contain: boolean value
Attributes: none
Default: on

Setting this option to "no" will suppress reporting changes to existing records to the remote party. This can make sense for datastores with objects that cannot change per definition (such as received email) and will prevent updates for such objects in all cases (even if the modification date is updated).

11.34.8 <maxitemspermessage>: maximum number of data items per SyncML message

Contained in: <datastore>
Can contain: max number of items per SyncML message (0=no limit)
Attributes: none
Default: 0 (no limit)

This can be used to specify how many items from this datastore are sent to the remote party in a single SyncML message maximally. Usually, there is no need to set a maximum, as the maximum SyncML message size implicitly limits the number of items anyway. However, if a datastore is exceptionally slow in fetching even small data items, setting a <maxitemspermessage> limit can ensure composing a SyncML message does not take too long.

11.34.9 <alwayssendlocalid>: send localID (GUID) in all operations (not only adds).

Contained in: <datastore> (in <server> only)
Can contain: boolean value
Attributes: none
Default: off
New in: 3.0.2.0

Setting this option to "yes" will cause the server to send the server-side ID (localID, GUID) to be sent to the client not only in SyncML "Add" commands, but also for "Replace" and "Delete". **Note that this is not strictly according to the SyncML standard –** however it is accepted by real-world clients and **helps avoiding problems** when client items addressed by a "Replace" command do not exist any more. Without the localID, a client cannot add items instead of replace, which can cause sessions to abort.

11.34.10 <conflictstrategy>, <slowsyncstrategy>, <firsttimestrategy>: sync conflict resolution strategy

Contained in: <datastore>
Can contain: conflict strategy name
Attributes: none
Default: newer-wins

These tags specify how to handle conflict situations (an object was modified on both server an client). <conflictstrategy> defines the strategy that is used during normal synchronisation, where <slowsyncstrategy> defines the strategy that is used in so-called slow sync (not first time synchronisation after some problem or data loss in either client or server). <firsttimestrategy> defines the strategy that is used in first-time sync, and is normally set to the same strategy as <slowsyncstratgey>

The following values can be specified:

- "**duplicate**" : The conflicting objects are duplicated such that both client and server will have both versions of the object. The user then decides which one is the "right" one or to keep both.
- "**newer-wins**": If the objects in question both carry a timestamp when they were last modified, the object that was more recently modified will "win" the conflict. If no timestamps are available, this mode works like "duplicate".
- "**server-wins**": Server's version always wins the conflict.
- "**client-wins**": Client's version always wins the conflict.

Note that "winning" does not necessarily mean that winning side's data simply overwrites loosing side's data. Synthesis Sync Server has powerful merging features that will combine data from both objects. The merging is contolled by the "merge" attribute of the <field> tag (see 10.2) and can be customized with <mergescript> (PRO version only).

The default setting of "newer-wins" is the most "smart" mode; it reduces unnecessary duplicates while still avoiding newer data to be overwritten with older data.

Note that the actual strategy used for a sync session can also be defined in runtime (depending on user settings for example) by using the SETCONFLICTSTRATEGY() script function in the <datastoreinitscript> (see 11.34.21).

11.34.11 <typesupport>: datastore's supported types

Contained in: <datastore>
Can contain: <use>
Attributes: none

This tag defines what SyncML content data types (as defined in the <datatypes> config section, see 10) are used for sending data to and receiving data from a SyncML client.

The tag must contain a <use> tag for every data type that is to be supported by the datastore. A datastore can support multiple (similar) types such as different versions of vCard for a contact datastore.

11.34.12 <use>: use a datatype

Contained in: <typesupport>
Can contain: nothing
Attributes: datatype, mode, preferred

This tag adds support for a datatype (such as a vCard or vCalendar version, or any other customer-defined datatype).

The <use> tag has the following attributes:

- "**name**": this must be a name of a previously defined datatype (using the <datatype> tag in the <datatypes> section of the config file (see 10.5).
- "**mode**": this optional attribute specifies if the datatype is to be used for receiving data from the client ("r") or for sending data to the client ("w") or both ("rw"). The default is "rw".
- "**preferred**": this optional attribute must be present for exactly one write-enabled and one read-enabled (or one combined read-write-enabled) <use> tag. It is used to specify which datatypes are preferred for reading and writing. The special value "legacy" can be specified instead of "no" to specify a type that should be used as the preferred type when the engine runs ins so-called legacy mode (i.e. when a not fully conformant remote party is detected automatically or settings or remote rule (see 11.36.21) demand it). For example, in configurations that support both vCard 2.1 and vCard 3.0, usually vCard 3.0 should be marked preferred=yes, and vCard 2.1 as preferred=legacy. So fully compliant remote parties will profit from the newer vCard 3.0 format, while bad implementations that might not work properly with the new format will use the older vCard 2.1.
- **"rulematch"**: (3.0.2.0 and newer) this optional attribute is used to define device-specific data types. The contents of "rulematch" can be a single name of a <remoterule> (see 11.36) or a comma separated list of remote rule names. The names might also contain wildcards (* and ?). If "rulematch" matches the currently active remoterule, the specified datatype will be used **instead of the preferred type** and **overriding normal transfer type negotiation**.

Example (a contact datastore supporting vCard 2.1 and vCard 3.0 for both reading and writing, while preferring vCard 2.1, and a special datatype exclusively for a exotic device):

```
<typesupport>
```

```
  <use datatype="vcard21" mode="rw" preferred="yes"/>
  <use datatype="vcard30" mode="rw"/>
  <use datatype="vcard21_exotic" rulematch="Exotic*"/>
</typesupport>
```

11.34.13 <ds12filters>: enable SyncML DS 1.2 filtering

Contained in: <datastore>
Can contain: boolean value
Attributes: none
Default: on

If this is set to true, the sync engine accepts SyncML DS 1.2 style <Filter> specifications (see 7.4 for syntax) in the <Alert> commands, and reports filter capabilities in the device information (SyncML DS 1.2 only).

11.34.14 <daterangesupport>: enable date range filtering

Contained in: <datastore>
Can contain: boolean value
Attributes: none
Default: on

If this is set to true, this datastore supports date range filtering (see /dr() option in 7.5 and SINCE/BEFORE keywords in 7.3). Note that actual support must be implemented using appropriate <filterscript> in the datatype (see 10.5.10) or <optionfilterscript> in SQL (see 12.20.25) to actually make date ranges functional. Setting this option only makes date range filtering keywords appear in the device information (SyncML DS 1.2 only).

11.34.15 <acceptfilter>: check incoming items

Contained in: <datastore>
Can contain: filter expression (see 7)
Attributes: none

This filter is applied (in *test mode*) to incoming items to check if the datastore can process them. If not, the item is rejected with SyncML status code 415 (unsupported type or format).

This filter is also applied in *make-pass mode* (see 7.1) to any item sent from the datastore to the remote party to make sure it meets the <acceptfilter> conditions.

This filter is useful for example to avoid vTODO items to mess up an event-only datastore and vice versa. Note that more complex filtering (or filtering that causes items to be ignored rather than rejected with status 415) can be implemented with the <processitemscript> (see 10.5.11). To simply make <acceptfilter> discard items silently instead of returning status 415, use <silentdiscard> (see 11.34.16).

11.34.16 <silentdiscard>: discard not accepted items silently

Contained in: <datastore>
Can contain: boolean value
Attributes: none

If this is set to true, incoming items that do not pass the <acceptfilter> (see 11.34.15) are silently discarded and successful status is returned, rather than 415.

11.34.17 <localdbfilter>: filter subset of datastore

Contained in: <datastore>
Can contain: filter expression (see 7)
Attributes: none

This filter is applied (in *test mode*) to all items read from the local database. If an item does not pass the filter, it will be simply ignored for the synchronisation.

This filter is also applied in *make-pass mode* (see 7.1) to any item sent from the remote party before storing it in the local database.

This filter basically does the same thing (actually with reversed logic) as the <invisiblefilter>. However, while visibility is something that is also changed on a per-item basis, the <localdbfilter> is meant to implement a first-level subselection of items. An example for this would be when a server database that contains both events and tasks, but we need to create a <datastore> for events only.
Normally, this can be archieved by appropriate WHERE clauses in SQL statements directly, but with <localdbfilter> this can be implemented for text-based datastores as well.

11.34.18 <invisiblefilter>: filter invisible items

Contained in: <datastore>
Can contain: filter expression (see 7)
Attributes: none

This filter is applied (in *test mode*) to items to be sent to the remote party. If the filter result is true, the item is **not** sent to the remote party (it is considered *invisible* for the remote party).

This filter is also applied in *make-pass mode* (see 7.1) when the SyncML engine must make an item invisible for the remote party. For example, when a SyncML client sends an archive-delete command, the SyncML engine will apply <invisiblefilter> to the item - which looks like deleting the item from the client (it is not there any more - *invisible*) but still exists in the database.

11.34.19 <makevisiblefilter>: make item visible

Contained in: <datastore>
Can contain: filter expression (see 7)
Attributes: none

This filter is never used in *test mode*, but only in *make-pass mode* (see 7.1) as follows: When an item must be made visible for the remote party, and <invisiblefilter> test returns true (meaning that the item is invisible), then the <makevisiblefilter> is applied in *make-pass mode*.
This happens for items that are added to a server from a client to make sure they will be visible when they are read back from the server database later.

11.34.20 <makepassfilter>: make incoming items pass

Contained in: <datastore>
Can contain: filter expression (see 7)
Attributes: none

This filter is never used in *test mode*, but only in *make-pass mode* (see 7.1) as follows: Before an incoming items is added to the database, the <makepassfilter> is applied in *make-pass mode*.
This primarily makes sense in a server that has visibility control to make sure that items added from clients without a visibility level specified in the database path CGI are added to the database with a defined visibility level.

11.34.21 <datastoreinitscript>: script called before accessing database

Contained in: <datastore>
Available: in PRO versions only
Can contain: script
Script context: data**store** context
Attributes: none
Default: no script

This script is executed just before the datastore contents (that is, the sync set's data itself) is accessed for the first time in a sync session. Note that access to administrative data (like targets and maps) has already taken place at this time.
This is is good place to modify filters depending on the remote device (see *remote rules* in 11.36 and REMOTERULENAME() function in 6.14.6) or depending on options passed by the client (such as the string in DBOPTIONS(), see below). It is also the place to apply options like readonly folders that have been detected for example in <adminreadyscript> (see 11.34.23).

This script has access to the following special script functions:

string GETCGITARGETFILTER(): returns the current *temporary* filter string as sent by client, see CGI options in 7.4. Note that this might change during the sync session, depending on the client. The complete filter expression used as *temporary* filter consists of GETCGITARGETFILTER() *and* GETTARGETFILTER() together.

string GETTARGETFILTER(): returns the current *temporary* target filter string (as internally set by SETTARGETFILTER or ADDTARGETFILTER, see below). The complete filter expression used as *temporary* filter consists of GETCGITARGETFILTER() *and* GETTARGETFILTER() together

SETTARGETFILTER(string *targetfilter*): sets the current *temporary* filter to *targetfilter*, overwriting any existing filter string.

ADDTARGETFILTER(string *targetfilter*): adds an additional *temporary* filter expression to the existing *dynamic* target filter (See 7 for difference between *dynamic* and *static* filters). This function automatically inserts parantheses and an "and" operator such that the resulting filter expression will be true only if both existing and new added filter expressions

string GETFILTER(): returns the current dynamic target filter string (as eventually sent by client, see CGI options in 7.4)

SETFILTER(string *filter*): sets the current dynamic target filter to *filter*, overwriting any existing filter string.

ADDFILTER(string *filter*): adds an additional *dynamic* filter expression to the existing *dynamic* target filter (See 7 for difference between *dynamic* and *static* filters). This function automatically inserts parantheses and an "and" operator such that the resulting filter expression will be true only if both existing and new added filter expressions are true.

ADDSTATICFILTER(string *filter*): adds an additional filter expression to the existing *static* database filter (defined by <localdbfilter>, see 11.34.17). This function automatically inserts parantheses and an "and" operator such that the resulting filter expression will be true only if both existing and new added filter expressions are true.

string DBOPTIONS(): returns the option string specified by the remote party using the /o() option as CGI parameter in the database path (see CGI options in 7.4).

string DBNAME(): returns the name of the datastore. This might be useful for example in the vCalendar datatype to check if the vCalendar item is being used in the "tasks" or the "events" datastore.

string LOCALDBNAME(): returns the name of the local datastore as used in the <sync> target locuri specification from the remote party. This can be used in datastores with <alias>es (see 11.34.1) to find out how the datastore was addressed.

string REMOTEDBNAME(): returns the name of the remote datastore as used in the <sync> source locuri specification from the remote party. Note that the remote datastore name might be a path containing multiple elements and even CGI.

timestamp STARTDATE(): returns the start date if a date range was set for the datastore (for example with the /dr() CGI option, see 7.4). If no range was set, STARTDATE() returns EMPTY.

SETSTARTDATE(timestamp *date*): sets the start date (as it can also be set by the /dr() CGI option (see 7.4) according to *date*.

timestamp ENDDATE(): returns the end date if a date range was set for the datastore (for example with the /dr() CGI option, see 7.4). If no range was set, ENDDATE () returns EMPTY.

SETENDDATE(timestamp *date*): sets the end date (as it can also be set by the /dr() CGI option (see 7.4) according to *date*.

integer NOATTACHMENTS(): returns TRUE if the /na GCI option (see 7.4) is set.

SETNOATTACHMENTS(integer *flag*): set the /na option (see 7.4) according to *flag*.

integer MAXITEMCOUNT(): returns the number specified with the /max GCI option (see 7.4) or zero if no /max option was used.

SETMAXITEMCOUNT(integer *maxcount*): set the the /max GCI option (see 7.4) according to *maxcount* (0 = no item count limit).

integer DEFAULTSIZELIMIT(): returns the default size limit set with the /li() CGI option, (see 7.4). If no size limit was set, DEFAULTSIZELIMIT() returns EMPTY. Note that individual items might have a size limit differing from the default (see SIZELIMIT and SETSIZELIMIT functions in 10.5.10)

SETDEFAULTSIZELIMIT(integer *limit*): set the the /li() CGI option (see 7.4) according to *limit* (EMPTY = no size limit, 0 = only header information).

integer DBHANDLESOPTS(): This returns TRUE if the database implementation does handle the needed filtering for date range and limit options automatically. For the currently available ODBC and text based datastores, this always returns FALSE, but future implementations might be able to handle the filtering natively.

integer SLOWSYNC(): returns true if session is a slow sync

FORCESLOWSYNC(): can be used in <alertscript> (see 11.34.25) to force a slow sync even if the sync engine would do a normal sync. This has the same effect as using the /slow option in the database path (see 7.4).

integer FIRSTTIMESYNC(): returns true if session is a first-time (slow) sync

SETCONFLICTSTRATEGY(string *strategy*): allows to define a conflict strategy for the datastore for example depending on logged in user or device. *strategy* must be a valid strategy name as described in 11.34.10.

integer ALERTCODE(): returns the alert code for the current synchronisation (see SyncML standard for details). This can be useful in <alertscript> (see 11.34.25) to determine what kind of synchronisation was requested.

integer SETALERTCODE(integer *alertcode*): allows to set the *alertcode* to something different than received from the remote party in <alertscript> (see 11.34.25). Note that doing so requires knowledge of the SyncML protocol - arbitrarily modifiying the alert code is likely to make sync sessions fail.

integer READONLY(): returns true if the current sync is a read-only sync, that is, data from the remote is ignored, and the local database is only read (never written) to send updates to the remote party. Note that the initial state of this flag is determined by the <readonly> tag (see 11.34.4) and by the session-level SETREADONLY() function (see 11.33).

SETREADONLY(integer *readonly*): sets the *readonly* flag. This is useful to force a read-only sync (modifications from the remote party will be ignored) with this datastore when the remote actually requests a two-way sync in the <alertscript> (see 11.34.25). Note that there is also a session-level version of SETREADONLY(), see 11.33.

integer REFRESHONLY(): returns true if the current sync is a refresh-only sync, that is, only data from the remote is received and stored, but no updates or adds are sent to the remote party.

SETREFRESHONLY(integer *refreshonly*): sets the *refreshonly* flag. This is useful to force a refresh-from-remote sync when the remote actually requests a two-way sync in the <alertscript> (see 11.34.25). Make sure that you do not clear the refreshonly flag when the remote actually requests a refresh-only sync, as this will likely make the sync session fail (the remote does not expect data coming from the server).

In SyncML client configurations, additionally the following functions are available:

ADDTARGETCGI(string *cgi*): adds the string *cgi* as CGI to the database path sent to the server. This can be used to add TAF expressions or proprietary server options to the base server database path. The function automatically adds a "?" delimiter between the database path before appending *cgi* if the delimiter is not already part of the database path. The function also checks if *cgi* is already part of the database path and if so, does not add it a second time.

SETRECORDFILTER(string *filterexpression*, boolean *inclusive*): sets a record level filter for the alert to be sent to the server. If the session is run in SyncML DS 1.2 or later, the filter expression is sent to the server using the <filter> and <FilterType> tags. Otherwise, inclusive and exclusive filters are added using the /tf() and /fi() option syntax, resp. see 7.5.

SETDAYSRANGE(integer *daysbefore*, integer *daysafter*): sets a relative day range from *daysbefore* days into the past and *daysafter* days into the future for the alert to be sent to the server. If the session is run in SyncML DS 1.2 or later, this range is represented as a filter expression using the BEFORE and SINCE filter keywords. Otherwise, the date range is added as CGI using the /dr(-x,y) option syntax, see 7.5.

variant TARGETSETTING(string *settingsfieldname*): Only for client configurations which are based on the Synthesis SyncML client engine library, this function can be used to query certain (not all) fields of the target settings. See the SDK_manual.pdf for more information on target settings. Currently supported *settingsfieldname*s are: "extras", "limit1", "limit2", "remoteFilters". Others might be supported depending on the client engine library version. This is useful for example to use the limit1 and limit2 fields in the settings for user-settable date range, which can then be queried in <alertprepscript> (see 11.34.26) using TARGETSETTING() and applied to the outgoing alert using SETDAYSRANGE().

11.34.22 <datastorefinishscript>: script called after accessing database

Contained in: <datastore>
Available: in PRO versions only
Can contain: script
Script context: data**store** context
Attributes: none
Default: no script

This script is executed just after the datastore has been (successfully) accessed. See also <datastoreinitscript> for details.

11.34.23 <adminreadyscript>: script called when admin data (targets, maps) are read

Contained in: <datastore>
Available: in PRO versions only
Can contain: script
Script context: data**base** context
Attributes: none
Default: no script

This script is executed when datastore related administrative data (target information, folder information, map data) has been fetched.
This is is good place to implement folder specific behaviour (e.g. making a folder read-only).

Unlike <datastoreinitscript>, the <adminreadyscript> has access to the *database context* specific functions (such as those described in 11.34.41.3 or in 12.1.4 for ODBC).

11.34.24 <syncendscript>: script executed at end of sync

Contained in: <datastore>
Available: in PRO versions only
Can contain: script
Script context: data**base** context
Attributes: none
Default: no script

This script is called once after all operations related to a datastore in a sync session are completed. This is the script to place special application specific operations that must be done after completing sync with a datastore.
Note that the syncendscript is executed at end of both successful and failing sync sessions.

Unlike <datastorefinishscript>, the <syncendscript> has access to the *database context* specific functions (such as those described in 11.34.41.3 or in 12.1.4 for ODBC).

11.34.25 <alertscript>: script called at sync alert

Contained in: <datastore>
Available: in PRO versions only
Can contain: script
Script context: data**store** context
Attributes: none
Default: no script

This script is executed when the server **receives** an <alert> command from the client. This is the place to install custom behaviour (like switching to refresh-from-remote only using the SETREFRESHONLY() function or update-remote only using SETREADONLY()). All script functions available in <datastoreinitscript> are available (see 11.34.21), but not all might make sense at this stage of the sync process.
Note: Don't confuse this with <alertprepscript> (see 11.34.26).

11.34.26 <alertprepscript>: script called before sending sync alert

Contained in: <datastore> within <client>
Available: in PRO version **clients only**
Can contain: script
Script context: data**store** context
Attributes: none
Default: no script

This script is executed before a client sends an <alert> to the server. This is the place to add extra parameters for the datastore sync like filter expressions (see 7) or date range options (see 7.5). This can be done using script functions like ADDTARGETCGI(), SETRECORDFILTER(), SETDAYSRANGE(), TARGETSETTING()
Note: Don't confuse this with <alertscript> (see 11.34.25).

11.34.27 <sentitemstatusscript>: script to handle status codes for sent items

Contained in: <datastore> or <server>/<client> (see text)
Available: in PRO versions only
Can contain: script
Script context: data**store** context
Attributes: none
Default: no script

This script is executed whenever a status for a sent data item is received. Note that this script can be specified on both the datastore level (will catch status for that datastore only) or on the session level (will catch all status responses).
This script must return *true* if it has taken all actions required to handle the status code, otherwise it should return nothing or *false* which will cause the engine to apply default processing for the status code.

This script has access to the following special script functions:

integer STATUS(): returns the current status code. See 18.1 for a list of SyncML error codes.

SETSTATUS(integer *statuscode*): sets a new *statuscode* (this will be used by all subsequent processing of the status instead of the original status code, including the Sync engine's default processing.

SETRESEND(boolean *resend*): This can be used to override the error case behaviour set by <resendfailing> (see 11.34.29) on a per-item basis. If *resend* is set to true, the item will be marked for resend in the next session. If *resend* is set to false, a non-OK status will cause an error and abort the sync with that datastore. **Note that marking items for resend only works in datastores with <resumesupport> (see 11.34.39) switched on.**

ABORTDATASTORE(integer *statuscode*): aborts syncing the current datastore (but continues the sync session if it includes other datastores) and reports *statuscode* as the reason for aborting the sync. Note that *statuscode* can be 0 to abort silently.

STOPADDING(): Stops adding more data to the remote datastore. This can be used to continue a sync after the remote party signals that its datastore has no capacity to accept further records.

string SYNCOP(): Returns the sync operation related to the status code being processed. Possible return values are: "add", "replace", "archive+delete", "soft-delete", "delete", "copy" and "map".

11.34.28 <receiveditemstatusscript>: script to handle status codes for received items

Contained in: <datastore> or <server>/<client> (see text)
Available: in PRO versions only
Can contain: script
Script context: data**store** context
Attributes: none
Default: no script

This script is executed before sending a status commad for a received data item to the remote party. Note that this script can be specified on both the datastore level (will catch status for that datastore only) or on the session level (will catch all status responses).
This script can return *true* if it wants to mark the status code as "regular" processing (fully ok, no workarounds applied) or *false*to signal "irregular" processing.

This script has access to the same special script functions as <sentitemstatusscript>, see 11.34.27.

11.34.29 <resendfailing>: re-send failing items in next session

Contained in: <datastore>
Can contain: boolean value
Attributes: none
Default: true

If this option is set to true, items sent to the remote that receive a non-ok status will not cause the sync session to abort, but will be marked such that the items get re-sent in the next sync session. This is new functionality of version 3.0, and helps to overcome temporary failures of writing items to the remote party (such as outgoing email that can't be sent at the first attempt will not break the session, but simply re-tried in the next session).
This option can be set to false to restore pre-version-3.0 behaviour.
Note that resending items requires that map entries have map flags (<resumesupport> turned on, see 11.34.39 and 12.20.5).

11.34.30 <timeutc>, <timestamputc>: type of database timestamp

Contained in: <datastore> (<timeutc>)
<server> or <client> (<timeutc> or <timestamputc> for version 2.1 compatibility)
Can contain: boolean value
Attributes: none
Default: false (operating system's local time)
Usage: deprecated in 3.1 onwards, use <datatimezone> (see 11.34.31) instead

If set to true, timestamp results returned by the database API layer (ODBC or plugin) are interpreted as UTC (former Greenwich Mean Time, GMT) and timestamps sent to the database API layer are sent in UTC.

Note: when used in context of a <datastore>, this setting only affects the actual accesses to this datastore - so it is possible to have different timestamp settings for different datastores. If used in context of <server> or <client>, the setting is used for all accesses that are not related to a particular datastore, such as reading the database time (for example see 12.18).

11.34.31 <datatimezone>: timezone for database timestamps

Contained in: <datastore>; <server> or <client>
Can contain: time zone specification (see 5.3)
Attributes: none
Default: SYSTEM (operating system's local time zone)
Usage: New in 3.1 onwards, use instead of deprecated <timeutc> and <timestamputc> (see 11.34.30)

Timestamp results returned by the database API layer (ODBC or plugin) are interpreted using the time zone specified, and timestamps to be written to the database are converted to this zone before writing (except for timestamps explicitly mapped as floating using the "f" mode option in the <map> / <fieldmap>, see 11.34.41.1).

Note: when used in context of a <datastore>, this setting only affects the actual accesses to this datastore - so it is possible to have different time zone settings for different datastores. If used in context of <server> or <client>, the setting is used for all accesses that are not related to a particular datastore, such as reading the database time (for example see 12.18).

11.34.32 <userzoneoutput>: output data in user zone

Contained in: <datastore>
Can contain: boolean value
Attributes: none
Default: true
Usage: New in 3.1 onwards

If this is set to true (the default), timestamps are converted to *user time zone context* (see 5.2) before data is converted to SyncML content formats like vCalendar. For cases where the original timezone as obtained from the database must be retained, this can be set to false.

11.34.33 <datacharset>: character set to be used for database strings

Contained in: <datastore>, <server> or <client>
Can contain: name of character set
Attributes: none
Default: "ANSI"

This defines the character to be used for ODBC strings:

- **"ASCII"**: plain 7-bit ASCII, ANSI/ISO-8859-1 characters are converted to nearest ASCII-equivalent, for example 'ä' to 'a' etc.
- **"ANSI"**: standard window character set
- **"ISO-8859-1"**: ISO-8859-1 character set
- **"UTF-8"**: UTF-8 character set
- **"GB2312"**: Simplified Chinese standard character set. Note that this character set is not supported in all client and server versions.
- **"CP936"**: Simplified Chinese Windows codepage (Multi-Byte character set). Note that this character set is not supported in all client and server versions.

Note: when used in context of a <datastore>, this setting only affects the actual accesses to this datastore - so it is possible to have different character sets for different datastores. If used in context of <server> or <client>, the setting is used for all accesses that are not related to a particular datastore.

11.34.34 <datalineends>: encoding of line ends within database strings

Contained in: <datastore>, <server> or <client>
Can contain: name of line-end mode
Attributes: none
Default: "dos"

This defines how line ends within strings are encoded:

- **"dos"**: line ends are DOS/Windows compatible CRLF (0x0D followed by a 0x0A character)
- **"mac"**: line ends are Apple Macintosh compatible CR (single 0x0D)
- **"unix"**: line ends are Unix/Linux compatible LF (single 0x0A)
- **"cstr"**: line ends are compatible with platform's encoding for lineends in C strings (normally 0x0A like "unix")
- **"filemaker"**: line ends are encoded as single 0x0B character (which is used by the Filemaker desktop database)

Note: when used in context of a <datastore>, this setting only affects the actual accesses to this datastore - so it is possible to have different line ends for different datastores. If used in context

of <server> or <client>, the setting is used for all accesses that are not related to a particular datastore.

11.34.35 <updateallfields>: always update all fields

Contained in: <datastore>
Can contain: boolean value
Attributes: none
Default: false

If this option is set to true, the server always updates all fields that are <map>ed when updating a record. If it is set to false, the server might update only those field that have actually changed. Note that this should be set to false for efficiency reasons except if the database really does not support updating fewer than all fields.

11.34.36 <fromremoteonlysupport>: Support for "one-way from remote sync"

Contained in: <datastore>
Can contain: boolean value
Attributes: none
Default: off

This flag determines if the datastore can perform the "one-way from remote sync" mode. This is the only sync mode that needs one (or two, if <synctimestampatend> is enabled) extra time-stamps to be maintained per sync target (in ODBC this is in the *sync targets* table, see 12.20.2). Synthesis SyncML engine versions before 2.0.7.2 did not support that extra mode, so some existing installations may not have the appropriate fields in their SYNC_TARGETS tables yet.

11.34.37 <synctimestampatend>: How to determine "time of last sync"

Contained in: <datastore>
Can contain: boolean value
Attributes: none
Default: off

This flag determines how the "time of last sync" is determined. This value is very important for subsequent sync session to find out which records have been changed since last sync session. Normally, this should be left to the default (off), meaning that the "time of last sync" is the time when the sync session begins. This time is used as modification timestamp for all records touched during a synchronisation.
For some (desktop) databases, it might not be possible to set modification timestamps when inserting or updating records, but those databases always assign the current time to the last-modified field. In this case, the "time of last sync" must be taken AFTER all modifications have been applied. This can be done by setting <synctimestampatend> to on.
Note that in ODBC based datastores, SQL statements (see 12.20.2) must be formed according to this setting.

Note that setting <synctimestampatend> is only safe in true single-user situation. During sync, no other modifications (neither by another sync session nor by another database user) may occur!

11.34.38 <storesyncidentifiers> (or <storelastsyncidentifier>): custom "time of last sync" identifier

Contained in: <datastore>
Can contain: boolean value
Attributes: none
Default: off

This flag is provided for plugin based datastores that don't use timestamps (or a different kind of timestamp) to determine modifications since last sync. If this flag is set to yes, a separate datastore/plugin dependent identifier is saved in the sync target administrative data. In case of ODBC based admin data, the SYNC_TARGETS table must have the appropriate fields and the SQL statements to read and write target records must be adapted accordingly (see 12.20.2).

11.34.39 <resumesupport>: support for resuming interrupted sync session

Contained in: <datastore>
Can contain: boolean value
Attributes: none
Default: off

To enable SyncML DS 1.2 Suspend & Resume feature, this flag must be set. If this flag is set, some additional data will be saved for each sync target at the end of each SyncML message processed (for servers) or at the end of the session (for clients) that allows resuming interrupted sync sessions without starting over. In case of ODBC based admin data, the SYNC_TARGETS table must have the appropriate fields and the SQL statements to read and write target records must be adapted accordingly (see 12.20.2).

11.34.40 <resumeitemsupport>: support for resuming half-transmitted data items after interrupted sync

Contained in: <datastore>
Can contain: boolean value
Attributes: none
Default: off

This flag is only meaningful if <resumesupport> is switched on (see 11.34.39). If it is set to yes, it enables resuming transfer of partially transmitted items without re-transmitting the entire item. It requires some additional data to be saved for each sync target. Especially, the already transmitted fragment of a partially transmitted item will need to be saved. This is a block of binary data of arbitrary size. A BLOB database field can be used to store this block. In case of ODBC based

admin data, the SYNC_TARGETS table must have the appropriate fields and the SQL statements to read and write target records must be adapted accordingly (see 12.20.2).

11.34.41 <fieldmap>: mapping datatype's fields to database fields

Contained in: <datastore>
Can contain: <map>, <initscript>,<beforewritescript>,<afterreadscript>, <array> (ODBC only, see 12.20.20)
Attributes: fieldlist

This tag defines how the internal fields of a datatype's field list (see 10.1) are mapped to the fields in a SQL table.

The "fieldlist" attribute is required and specifies the field list containing the fields to be mapped.

The tag must contain a <map> tag (see 11.34.41.1 and for ODBC 12.20.19) for every field that is to be mapped.

In the PRO version the <fieldmap> tag can also contain scripts to perform conversions between internal field data and database data while reading or writing.

11.34.41.1 <map>, <mapredefine>: mapping a datatype field to a database field

Contained in: <fieldmap>, <array> (ODBC only, see 12.20.20)
Can contain: nothing
Attributes: name, references, type, mode, size, truncate
plus database API specific attributes (see 12.20.19 for ODBC)

This tag establishes a link between an internal field as defined in a <fieldlist>'s <field> tag (see 10.2) and a field in the datastore's user data table (SQL database table field or plugin data field).

The <mapredefine> tag works exactly as <map> except that the field must be already mapped before. This makes sense after using <automap> (see 11.34.41.2) to create a 1:1 default mapping, and then override some mappings with <mapredefine>.

The <map> tag has the following attributes:

- **"name"**: this is the database field name of the field to be mapped
- **"references"**: this is the name of a field in the <fieldlist> specified with the "fieldlist" attribute of the enclosing <fieldmap> (or <array>) or it is the name of a local variable of the *database context* (see script descriptions in 11.34.41.3, 11.34.41.4, 11.34.41.5 and 11.34.41.7)
- **"type"** specifies how the field should be treated when accessing it in the database. Note that most fields can just be accessed as string, even if they contain numeric data. Other types than string are only required when there is no unambiguous string representation (such as for date and time fields). The following types are supported:

- "**string**" : string field. Values are copied unmodified (except for appropriate character set and line feed conversion and truncation to the maximum field length as specified with the "size" attribute, see below).
- "**blob**" : BLOB (binary large object) field. Values are treated as opaque binary data, and are copied byte by byte without *any* modification (except for truncation to the maximum field length if specified with the "size" attribute, see below).
- **"numeric"** : numeric field. Values are assumed to be valid numeric strings. If not, this will cause database errors to occur, so use this type only for values that are really numeric. Empty fields will be stored as NULL
- "**date**" : date field. The referenced <field> must be a timestamp or date value.
- "**time**" : time field. The referenced <field> must be a timestamp or time value.
- "**timefordate**" : This special type is used when a referenced timestamp <field> must be stored as separate date/time fields in the database. In this case, create a <map type="date"> *and* a <map type="timefordate"> referencing the *same* <field>. **Note that the "date" field must be listed in the <map> before the "timefordate" field!**
- "**timestamp**" : timestamp field.
- **"zonename"** (New in 3.1): This special type is used to store time zone information from a referenced timestamp <field> in symbolic form (zone name) in the database. Unlike "zoneoffset_xxxx" (see below), the time zone name identifies a time zone including its daylight savings rule. Therefore this is the preferred way to store time zones, as it is not dependent on the time of the year. **Note that the "time" or "timestamp" field must be listed in the <map> before the "zonename" field!** For general information on time zone handling please refer to chapter 5.
- **"zoneoffset_hours", "zoneoffset_mins", "zoneoffset_secs"** : These are deprecated in versions 3.1 onwards of the SyncML engine, but still available for compatibility. They should no longer be used as a mere offset from UTC cannot specify a time zone completely. Use the "zonename" type instead (see above). **Note that the "time" or "timestamp" field must be listed in the <map> before the "zoneoffset_xxx" field!** For general information on time zone handling please refer to chapter 5.
- "**lineartime**" : integer representation of a timestamp in the SyncML engine's internal time format which is milliseconds elapsed since January 1st, 4712 BC, midnight.
- "**lineardate**" : integer representation of a date in the SyncML engine's internal date format which is days elapsed since January 1st, 4712 BC.
- "**unixtime_s**" : integer representation of a timestamp in Unix Epoch Time, which is seconds elapsed since January 1st, 1970, midnight.
- "**unixtime_ms**" : Same as unixtime_s, but in milliseconds
- "**unixtime_us**" : Same as unixtime_s, but in microseconds
- "**unixdate_s**" : integer representation of a date-only value in Unix Epoch Time, which is seconds elapsed since January 1st, 1970.
- "**unixdate_ms**" : Same as unixdate_s, but in milliseconds
- "**unixdate_us**" : Same as unixdate_s, but in microseconds
- "**nsdate**" : integer representation of a timestamp in Apple Cocoa NSDate scale, which is seconds elapsed since the "reference Date" of January 1st, 2001, midnight.

- **"mode"**: this optional attribute specifies when the database field being mapped is used. In ODBC datastores this influences the field/value lists created by the %v,%V,%N placeholders, see 12.1.3). The default for "mode" is "rw" if not specified explicitly. Mode can consist of one or multiple of the following flag characters:
  - "**r**": the field is included when reading values from the database (in ODBC usually when executing a SELECT statement).

- "**i**": the field is included when inserting new records into the database (in ODBC usually executing an INSERT statement)
- "**u**": the field is included when updating existing records in the database (in ODBC usually executing an UPDATE statement)
- "**w**": same as specifying "i" and "u" together ("iu") - the field is included when writing to the database.
- "**p**": This flag means that the field should not be literally mapped, but inserted as a parameter. BLOBs are always inserted as parameters (even without the "p" flag), but for long strigs it might make sense as well. For ODBC, "p" means using the ODBC parameter mechanism; for plugin datastores, it causes the BLOBID-mechanism to be used (see SDK docs).
- "**f**" (New in 3.1): This flag has only a meaning for time and timestamp fields. It means that the database fields represents the time as a **floating (time zone independent) value** rather than in a fixed time zone context (as specified with <datatimezone>, see 11.34.31). This is useful in combination with the "zonename" field type, see above. For general information on time zone handling please refer to chapter 5. Note: do *not set* the "f" flag in a <map> that maps a timestamp's timezone name to a DB field.
- "**x**" (New in 3.1): This flag indicates that the value of the field directly referenced (with the "references" attribute, see above) must be kept for later finalisation using <finalisationscript> (see 11.34.41.6 for details). This is useful to create links between records in relational setups.

- **"size"**: this optional attribute specifies a maximum number of characters for string fields. It should always be set to avoid database errors when a client has longer strings than the database can store. Note for ODBC datastores especially that *size* is used to specify the maximum column size for binding SQL parameters (see "p" mode flag above) - in case you get 'HY104' SQL state errors, this is most likely caused by an invalid column size value.

- **"truncate"**: this attribute can be set to "no" (default is "yes") if the data field must not be truncated. This is usually the case for binary data, where truncation would mean corrupting the data. For string data, truncation is usually acceptable. If "truncate" is set to "no", this signals to the remote party that it should not send truncated data for this field. Note that this mechanism is a new feature of SyncML DS 1.2 and therefore does not work with 1.1 and 1.0 implementations.

- **"set_no"**:This optional attribute (which defaults to 0) can be used to assign this mapping entry to a numbered set of mappings. The default set is 0 and will normally be used for accessing the database. For ODBC, <map>s that have a *set_no* other than zero can be used in SQL statements that are prefixed with the %GO(*set_no*) sequence. See 12.1.3 for details. In Plugin datastores, *set_no* should always be 0.

- **Other attributes specific to the datastore type:** ODBC has additional attributes for <map>, see 12.20.19.

Example (fieldmap for a mimimalistic name/phone number + photo (new for 3.0) datastore)

```
<fieldmap fieldlist="Contact">
  <map name="MODIFIED" references="REV"
   type="timestamp" mode="r" size="0"/>
  <map name="LASTNAME" references="N_LAST"
   type="string" mode="rw" size="63"/>
  <map name="FIRSTNAME" references="N_FIRST"
   type="string" mode="rw" size="63"/>
```

```
  <map name="TEL_H" references="TEL_HOME"
   type="string" mode="rw" size="31"/>
  <map name="PHOTO" references="PHOTO"
   type="blob" mode="prw" size="16000" truncate="no"/>
</fieldmap>
```

11.34.41.2 <automap>: auto-map internal to DB fields

Contained in: <fieldmap>
Attributes: fieldlist, indexasname

This tag automatically maps all fields in the fieldlist to database fields. The database fields are assumed to have the same names as the internal fields in the fieldlist (unless indexasname is set to true, see below). All fields are mapped as "string", except for timestamps which are mapped as "timestamp". Note that <mapredefine> (see 11.34.41.1) can be used after <automap> to change some of the automapped fields when needed.

The "fieldlist" attribute is required and specifies the field list containing the fields to be mapped.

The "indexasname" attribute can be set to "true" to make the database field "names" to be the numeric index of the field in the fieldlist (0,1,2….n).

11.34.41.3 <initscript>: initialize accessing database

Contained in: <fieldmap> or <array> (SQL only, see 12.20.20)
Available: in PRO versions only
Can contain: script returning boolean value
Script context: data**base** context
Attributes: none
Default: no script

If the <initscript> tag is directly contained in the <fieldmap> tag (see 11.34.41), this script is executed once before the SyncML engine starts accessing the database and can be used to declare and initialize variables in the *database context*.

ODBC only: If the <initscript> tag is contained in an <array> tag (see 12.20.20 for details), the script is executed once every time before the SyncML engine starts accessing the detail records for a master record (which is always *after* the master record is read or written). If the script returns false, the SyncML engine will not read or write detail records. This is useful if the master record contains an indication if there are detail records at all which can be tested in the <initscript> to increase performance by avoiding unneeded SQL queries. The <initscript> of an <array> can access all fields of the item being read or written - as the arrays are always read or written *after* the master record is read or written, the fields mapped in the <fieldmap> are already read from the database when <initscript> is called for an <array>.

In ODBC datastores, the <initscript> has access to the SQL execution functions described in 12.1.4 and the following functions specific to the *database context*:

integer ARRAYINDEX(): ODBC only: Returns the current array index (zero based) when reading or writing an array. In the <finishscript> (see 11.34.41.7) the return value is the number of array records read or written.

string PARENTKEY(): Returns the key (*localID*) of the master record (same value as %k represents in SQL queries, see 12.1.3)

string NEWKEY(): Returns a new key (*localID*) for the master data table (same value as %X represents in SQL queries, see 12.1.3). Note that this can only be used when <obtainidafterinsert> is false, that is, when keys can be generated independent from actually inserting records into the master data table.

string LASTKEY(): Returns the last key generated for the master data table (same value as %X represents in SQL queries, see 12.1.3).

integer WRITING(): Returns true if script is called while writing to the database, returns false if script is called while reading from the database.

integer INSERTING(): Returns true if script is called while inserting new records to the database, returns false if script is called while reading or updating existing record in the database. Note that even while UPDATEing a record, it's <array> detail records may get INSERTed.

SETSQLFILTER(string *sqlfilter*): ODBC only: This can be used to specify a WHERE clause expression which is then used as part of the %AF and %WF placeholders, see 12.1.3.

string LOGSUBST(string logtext): Returns *logtext* with all placeholders which are valid in <writelogsql> (see 12.19) substituted with the appropriate value.

11.34.41.4 <afterreadscript>: post-process item read from database

Contained in: <fieldmap> or <array> (ODBC only, see 12.20.20)
Available: in PRO versions only
Can contain: script
Script context: data**base** context
Attributes: none
Default: no script

This script is called after a database record has been read according to the <map> definitions into the internal item's fields (as defined in the <fieldlist>, see 10.1) or local variables of the *database context*. Note that for ODBC, the sync engine may call <afterreadscript> before or after reading all detail records from <array>s (see 12.20.20), therefore any scripting that refers to <array> fields should not be placed in the <afterreadscript> (but in the <finishscript> of the array). This script has access to all fields of the data item just read and may examine and change them before the data is processed any further.

This script is where custom data conversions between data in the database and <field> contents can be implemented. To do that, the fields that need special conversion are mapped (with <map>, see 11.34.41.1) to a local variable of the *database context*. Then, the <afterreadscript> reads the value from that variable and stores it in the actual <field>.

The <afterreadscript> has access to the *database context* specific functions described in 11.34.41.3.

11.34.41.5 <beforewritescript>: prepare writing item to database

Contained in: <fieldmap> or <array> (ODBC only, see 12.20.20)
Available: in PRO versions only
Can contain: script
Script context: data**base** context
Attributes: none
Default: no script

This script is called before starting to write a database record and has access to all fields of the data item that will be written and may examine and change them before the data is actually written to the database.

This script is where custom data conversions between data in the database and <field> contents can be implemented. To do that, the fields that need special conversion are mapped (with <map>, see 11.34.41.1) to a local variable of the *database context*. Then, the <beforewritescript> reads the value from the <field> and stores the converted value in that local variable. As the local variable is <map>ed to a database column, the converted value will be stored in the database.

The <beforewritescript> has access to the *database context* specific functions described in 11.34.41.3.

ODBC only: when used in an <array> (see 12.20.20), <beforewritescript> can return false to stop writing detail records explicitly (before any of the implicit criteria - maxrepeat, sizefrom and all-empty data - stops writing detail records).

The following example shows a <initscript>, a <afterreadscript> and a <beforewritescript> implementing a custom conversion between priority values in the database ("low", "normal", "high") and priority values needed for the vCalendar <field> (3,2,1):

```
<!-- the init script is executed once before reading or writing
     the record. It is normally used to define common variables
     for afterreadscript and beforewritescript -->
<initscript><![CDATA[
  // we define a string that will hold our converted PRIORITY value
  STRING PRIORITY_TEXT;
]]></initscript>

<!-- this script is called after reading one record from the
     database. PRIORITY_TEXT will contain the value as read
     from the database -->
<afterreadscript><![CDATA[
  IF (PRIORITY_TEXT==EMPTY) PRIORITY=EMPTY; // we have no priority
  ELSE IF (PRIORITY_TEXT=="LOW") PRIORITY=3;
  ELSE IF (PRIORITY_TEXT=="HIGH") PRIORITY=1;
  ELSE PRIORITY=2;
]]></afterreadscript>

<!-- this script is called before writing one record to the
     database. PRIORITY_TEXT must be assigned the value to be
     written to the database -->
<beforewritescript><![CDATA[
  IF (PRIORITY==EMPTY) PRIORITY_TEXT=EMPTY;
```

```
  ELSE IF (PRIORITY>2) PRIORITY_TEXT="LOW";
  ELSE IF (PRIORITY<2) PRIORITY_TEXT="HIGH";
  ELSE PRIORITY_TEXT="NORMAL";
]]></beforewritescript>
```

11.34.41.6 <finalisationscript>: finalize written items

Contained in: <fieldmap>
Available: new in 3.1, in PRO versions only
Can contain: script
Script context: data**base** context
Attributes: none
Default: no script

This is a special script that is called at the end of all user data accesses, right before <finish-script> (see 11.34.41.7). It is called once for every item written to the database (inserted or updated), but only if at least one of the field <map>s (see 11.34.41.1) has a "x" flag in its mode attribute.

In each call, the script has access to an item (like in the <beforewritescript> 11.34.41.5), but this item is empty except for the fields directly referenced in a <map> which has a "x" flag set in its mode attribute.

This mechanism allows to create inter-item relational links, for example when one incoming item references another (such as tasks and subtasks) by a string identifier (like an UID), but in the database this relation should be stored as a direct relational link from one table referencing another table's (or the same table's) identity key column.
In such a case, the key of the to-be-referenced item may not be known when the referencing item is written to the database (because the referenced item might be new in this session and thus only added later in the session's progress).
By setting the "x" flag in the <map> (see 11.34.41.1) referencing the string identfier field (say, an UID), this means that this field's value is saved until the end of the sync session, and will be available to the <finalisationscript>. On the other hand, fields in the item that don't have a "x" mode flag in their <map> will not be included in the item the <finalisationscript> can access. This is to limit the amount of data that must be kept in memory down to what is really needed at finalisation (most data content does *not* need to be kept, because it can be stored right away when the item is arriving during the sync session).
If no <map> item has the "x" flag set, the <finalisationscript> is not called at all.

The <finalisationscript> has access to the *database context* specific functions described in 11.34.41.3.

11.34.41.7 <finishscript>: finish access to database

Contained in: <fieldmap> or <array> (ODBC only, see 12.20.20)
Available: in PRO versions only
Can contain: script
Script context: data**base** context
Attributes: none
Default: no script

If the <finishscript> tag is directly contained in the <fieldmap> tag (see 11.34.41), this script is executed once after the SyncML engine has accessed the database and can be used to do some cleanup work in the *database context*. See also <initscript> for details.

If the <finishscript> tag is contained in an <array> tag, it is called once after reading or writing all records of an <array> is complete. It can be used for example to store the number of <array> elements read (ARRAYINDEX() function, see 11.34.41.3) in a <field>.

The <finishscript> has access to the *database context* specific functions described in 11.34.41.3.

11.35 <superdatastore>: combined datastore definition

Contained in: <server>, <client>
Can contain: <contains>
Attributes: name

Superdatastores are a very simple to use concept to combine two or more datastore into one "superdatastore" that can be accessed by the remote party as a single datastore. Most obvious use of a superdatastore is in any SyncML server that supports events and tasks. Some SyncML clients (mostly those based on Symbian OS, like Nokia9210, P800 etc.) do not access events and tasks as separate entieties, but as a single "calendar" datastore.
With superdatastores, creating this "calendar" data store is as easy as grouping the existing events and tasks datastores using a <superdatastore> tag.

A superdatastore must have a *name* attribute, which specifies the name under which the datastore will be accessible from the SyncML client (such as "calendar" for the combined events/tasks).

A <superdatastore> must contain one or multiple <contains> tags (see 11.35.1) to specifiy which datastores it groups together to form a superdatastore. Note that the grouped datastores must be defined in the configuration file before the <superdatastore> tag.

The following example shows a typical "calendar" superdatastore, grouping "events" and "tasks" together:

```
<superdatastore name="calendar">

  <!-- sub-datastores contained in this superdatastore -->

  <contains datastore="events">
    <dispatchfilter>F.KIND:=EVENT</dispatchfilter>
    <guidprefix>e</guidprefix>
  </contains>
```

```
  <contains datastore="tasks">
    <dispatchfilter>F.KIND:=TODO</dispatchfilter>
    <guidprefix>t</guidprefix>
  </contains>

  <!-- datatypes supported by this datastore -->

  <typesupport>
    <use datatype="vcalendar10" mode="rw" preferred="yes"/>
  </typesupport>

</superdatastore>
```

11.35.1 <contains>: Include a datastore in a superdatastore

Contained in: <superdatastore>
Can contain: <dispatchfilter>, <guidprefix>
Attributes: datastore

This tag is used to include the datastore specified with the *datastore* attribute into a superdatastore.

11.35.2 <dispatchfilter>: filter to direct incoming items

Contained in: <contains>
Can contain: filter expression (see 7)
Attributes: none

This filter must be specified to allow the SyncML engine to dispatch incoming data items to the correct datastore within the superdatastore. In case of the event/task datastore, the filter must check if the incoming item is a vEVENT (for the <contains datastore="events") or a vTODO (for the <contains datastore="tasks").

When an item is received from the remote party for the superdatastore, all <dispatchfilter>s of all the contained datastores are checked to see which contained datastore can handle it.

When an item is sent to the remote party from the superdatastore, the <dispatchfilter> is used in *make-pass mode* (see 7.1) to make sure the outgoing item meets the criteria set by the filter for incoming items.

11.35.3 <guidprefix>: prefix for item ID

Contained in: <contains>
Can contain: string (normally only one character)
Attributes: none

Each <contains> in a <superdatastore> must specify a different <guidprefix>. This string (single character recommended, more only if absolutely needed) is used as a prefix to the contained datastore's item IDs to form an ID that is unique within the entire superdatastore and allows the SyncML engine to find the correct datastore from the ID.

11.36 <remoterule>: special rules for specific remotes

Contained in: <server>, <client>
Can contain: see below
Attributes: name

Sometimes it is desirable to treat specific remote devices different from others. A common case is when experiencing compatibility problems with a client.
Synthesis Sync Server provides the <remoterule> config option to specify rules for certain clients.

<remoterule> has the following attributes:

- "**name**": This optional attribute is used to specify a name for the rule. The name can be used to check in scripts (PRO version only) which (if any) rule is active using the REMOTERULENAME() script function. The name can also be used to define device-dependent datatypes using datastore's rulematch="xxx" attribute in <typesupport><use> (see 11.34.12) and in MIME-DIR based datatypes to define properties depending on a certain remote rule (see 10.3.3)

A <server> or <client> tag can contain any number of <remoterule> tags, one for each device (or set of devices) that need special treatment.

A <remoterule> tag contains none, one or more tags to identify the device(s), and one or more options that control the special behaviour:

11.36.1 <finalrule>

Contained in: <remoterule>
Can contain: boolean value
Attributes: none
Default: no

Remote Rules are searched in the order of their definition. By default, the first rule that matches will be applied and *no more rules (that eventually would match as well) will be searched*. This option can be set in a remoterule to have the server *continue searching for more matching rules* after applying the rule.

11.36.2 device identification tags for <remoterule>

Contained in: <remoterule>
Can contain: text
Attributes: none

The following tags are available for matching devices:

- <manufacturer>: device manufacturer name
- <model>: device model name
- <oem>: OEM
- <firmware>: firmware version string
- <software>: software version string
- <hardware>: hardware version string

- <deviceid>: device ID (this should be unique, so this can be used to make a rule for one single specific device)
- <devicetype>: SyncML device type

If all of these tags appearing in a <remoterule> match with the connecting remote device, the rule is applied. Note that <manufacturer>, <model> and <oem> might contain wildcards (* and ?).
Note that default rules can be specified that is applied to *any* device by not specifying any of the device identification tags. As rules are searched in the order of their definition, such a "catch all" rule must either:

- be defined as the last remoterule (and will be applied to *all devices not handled by another rule*)
- or it must have a <finalrule>no</finalrule> special option (see) to allow subsequent rules to be processed (thus applying the settings to *all devices, even those which are caught by another rule*)

11.36.3 <descriptivename>

Contained in: <remoterule>
Can contain: text
Attributes: none

This option allows to set a more descriptive name for the device than what the device information provides (especially old devices do not even include the phone model name there). The descriptive name will be used when generating activity logs, see 11.31 and 12.19.

11.36.4 <limitedfieldlengths>: device has short fields

Contained in: <remoterule>
Can contain: boolean value
Attributes: none

If this option is set in a remoterule, it means that the device has limited field lengths, but does not identify them in the device information. This is the case for example with the T39m mobile phone. Without a special rule for this device (contained in all sample config files), the limited field lengths would not be detected by the server and long strings could get lost during sync.

Example (to switch on limited field length handling for the Ericsson T39m):

```
<remoterule>
  <!-- Rule for Ericsson T39m client -->
  <manufacturer>Ericsson</manufacturer>
  <software>R1A</software>

  <limitedfieldlengths>yes</limitedfieldlengths>
</remoterule>
```

11.36.5 <noemptyproperties>: do not send empty properties

Contained in: <remoterule>
Can contain: boolean value
Attributes: none
Default: off

If this option is set in a remoterule, the vCard/vCalendar (MIME-DIR) generator will suppress sending properties with no value. This is because some clients do have problems when encountering empty properties in a vCard or vCalendar. Normally, this option is not required.

11.36.6 <updateclientinslowsync>: update client records during slowsync

Contained in: <remoterule> or <server> (settings in <server> are defaults for sessions without a remote rule applied or with a remote rule applied that does not specify a value for this particular option)
Can contain: boolean value
Attributes: none
Default: off

If this option is set in a remoterule, the SyncML server will try to update client records with server data if comparison shows that the server has additional data that the client does not have *during non-first time slow sync. Note that this updating always takes place in first-time sync, regardless of this options's setting.* It is off by default because many clients cannot store some data but also cannot inform the server what they can store exactly, so turning this option on will cause much unneeded client updates at slow sync. For clients that store everything they report in devinf however, this feature can be switched on resulting in better data consistency after a slow sync. See also corresponding function in 6.14.6.

11.36.7 <updateserverinslowsync>: update server records during slowsync

Contained in: <remoterule> or <server> (settings in <server> are defaults for sessions without a remote rule applied or with a remote rule applied that does not specify a value for this particular option)
Can contain: boolean value
Attributes: none
Default: off

If this option is set in a remoterule, the SyncML server will try to update server records with client data if comparison shows that the client has additional data that the server does not have *during non-first time slow sync. Note that this updating always takes place in first-time sync, regardless of this options's setting.* It is off by default because differences in client and server database layouts often cause uneeded updates which will cause other clients to be updated on the next sync as well. For clients that store everything they report in devinf however, this feature can be switched on resulting in better data consistency after a slow sync. See also corresponding function in 6.14.6.

11.36.8 <noreplaceinslowsync>: never update client records during slowsync

Contained in: <remoterule>
Can contain: boolean value
Attributes: none
Default: off

If this option is set in a remoterule, the SyncML server will never send a REPLACE command to the client during slow sync. This is for clients that cannot handle replace command in slow sync (some older SyncML clients had problems with this) - this option should be set for those clients only. Note that this overrides <updateclientinslowsync> (see 11.36.6) as it effectively blocks *any* updates during slow-sync, including first-time sync.

11.36.9 <ignoredevinfmaxsize>: ignore maximum field size reported in client's devInf

Contained in: <remoterule>
Can contain: boolean value
Attributes: none
Default: off

If this option is set in a remoterule, the SyncML server will ignore the maximum field size reported by the client in it's devInf. This is needed for some DS 1.2 clients which report wrong field sizes, especially a way too low value (like 256 bytes) for the PHOTO field in contacts. This would cause that the server can never send a photo to these clients if it would respect the maximum field size. So for these clients, <ignoredevinfmaxsize> should be set in the appropriate remote rule.

11.36.10 <dspathindevinf>, <dscgiindevinf>: how to show datastore name in devInf sent to client.

Contained in: <remoterule>
Can contain: boolean values
Attributes: none
Default: true (for both options)

These options can be set to true or false in a remoterule to control a special workaround which is normally active for all clients as follows: If a client starts a sync for a datastore using not only the datastore name alone (like "contacts"), but includes a subfolder path (like in "contacts/private") or some CGI filters (like in "contacts/private?CATEGORY&iCON;Family"), and then queries devInf, the server by default puts the entire string as used by the client to address the datastore as the datastore name. This is because many phone clients will fail when the devInf does not fully match the string sent. More elaborate clients will accept both the full string or only the datastore's name. In case a client does not accept the full string, <dscgiindevinf> can be set to false to suppress the CGI part, and <dspathindevinf> to suppress all but the datastore's name.
These are highly technical and only needed in exotic cases.

11.36.11 <allowmessageretries>: allow client to send the same message twice

Contained in: <remoterule> or <server> (settings in <server> are defaults for sessions without a remote rule applied or with a remote rule applied that does not specify a value for this particular option)
Can contain: boolean value
Attributes: none
Default: off

This is a highly technical setting. Some clients try to re-send SyncML messages when the answer from the server does not reach the client (for example, on unstable connections). Normally, the server would reject this. Setting this option makes the server accept re-sent messages - however there is no guarantee that reprocessing a message is possible in all contexts.

11.36.12 <completefromclientonly>: allow client to send the same message twice

Contained in: <remoterule> or <server> (settings in <server> are defaults for sessions without a remote rule applied or with a remote rule applied that does not specify a value for this particular option)
Can contain: boolean value
Attributes: none
Default: no

This is a compatibility settings for clients that expect a from-client-only session to work exactly like any other sync type, that is, include a (empty) <Sync> from the server to the client and a <Map> phase. To conform to the standard, this option must be disabled.

11.36.13 <forcelocaltime>: always send time information as localtime

Contained in: <remoterule>
Can contain: boolean value
Attributes: none

If this option is set in a remoterule, time information sent to the remote party is always specified in local time. Many SyncML clients with bad time zone implementation need that this flag is set in order to avoid time shifts. Note that with SyncML 1.1.1 conformant devices this option is not needed as the device reports wether it supports UTC or not. See also corresponding function in 6.14.6.

11.36.14 <forceutc>: always send time information as localtime

Contained in: <remoterule>
Can contain: boolean value
Attributes: none

If this option is set in a remoterule, time information sent to the remote party is always specified in UTC. Note that this is the default case for SyncML 1.0, but with SyncML 1.1.1 conformant devices this option is not needed as the device reports wether it supports UTC or not. See also corresponding function in 6.14.6.

11.36.15 <treataslocaltime>: always treat received information as localtime

Contained in: <remoterule>
Can contain: boolean value
Attributes: none

If this option is set in a remoterule, time information received from the remote party is always interpreted as localtime, even if it looks as if it was UTC ("Z" suffix). This is needed for some Symbian based clients (P800) which send local time suffixed with a "Z" which would make the receiver interpret these as UTC if this option is not set. See also corresponding function in 6.14.6.

11.36.16 <treatasutc>: always treat received information as UTC

Contained in: <remoterule>
Can contain: boolean value
Attributes: none

If this option is set in a remoterule, time information received from the remote party is always interpreted as UTC, even if it looks as if it was local time (no "Z" suffix). This is needed for some Nokia Series 80 based clients (9500, 9300) which send UTC time without the "Z" suffix which would make the receiver interpret these as localtime if this option is not set. See also corresponding function in 6.14.6.

11.36.17 <nocontentfolding>: prevent folding long lines

Contained in: <remoterule>
Can contain: boolean value
Attributes: none

If this option is set in a remoterule, lines longer than 72 characters in MIME-DIR properties are not folded into multiple lines as requested by the MIME-DIR standard, but transferred as a single, long line. This is for buggy SyncML client implementation that crash or misbehave when processing folded lines.

11.36.18 <autoenddateinclusive>: end date for allday events inclusive

Contained in: <remoterule>
Can contain: boolean value
Attributes: none

If this option is set in a remoterule, date values with conversion mode "autoenddate" (see 10.3.4).will be rendered as 23:59:59 of the previous day in old formats like vCalendar 1.0. If it is not set, "autoenddate"s will be rendered as 0:00 of the next day (which usually is the first day not included in the all-day event when "autoenddate" is used for DTEND properties).
Note that there is a session-level setting for this option (see 11.23)

11.36.19 <outputcharset>: set default output character set

Contained in: <remoterule>
Can contain: character set name (choices see 11.34.33)
Attributes: none

This optioncan be set in a remote rule to use a non-default character set for output formats, such as in vCard or vCalendar. Usually, the character set in SyncML is UTF-8. However there are buggy SyncML client implementations that misbehave on UTF-8 strings, but work for example with ANSI charset.

11.36.20 <inputcharset>: set default input character set

Contained in: <remoterule>
Can contain: character set name (choices see 11.34.33)
Attributes: none

This optioncan be set in a remote rule to use a non-default character set for interpreting input formats when these do not implicitly specify a character set, such as in vCard. Usually, the character set in SyncML is UTF-8. However there are buggy SyncML client implementations that send data e.g. with ANSI charset.

11.36.21 <legacymode>, <lenientmode>: use relaxed conformance modes

Contained in: <remoterule>
Can contain: boolean value
Attributes: none

These options can be set for remote rules that handle buggy and ill-behaving SyncML implementations:

- If <legacymode> is set, the SyncML engine will use different data types (usually old vCard 2.1 and vCalendar 1.0 instead of modern MIME-DIR, but this is configurable, see 11.34.12).

- If <lenientmode> is set, the SyncML engine will be more tolerant to certain types of non-conformant behaviour. Especially, with servers which do not handle client side anchors correctly, <lenientmode> can disable client side anchor comparison.

11.36.22 <rejectstatus>: reject sync with device

Contained in: <remoterule>
Can contain: SyncML status code
Attributes: none

If this option is set in a remoterule, it will abort any attempt to synchronize with the specified status code. This is an option which can be used to prevent sync with some specific type of client completely (for example if a client is known to have bugs that could affect data integrity).

11.36.23 <requestmaxtime>: max time for request processing

Contained in: <remoterule>
Can contain: max duration (in seconds) of a single request processing
Attributes: none

This can be used to override the session-default for maximum request processing time (see 11.3 for details about limiting request time processing) - for example if a device is known to be especially impatient and timing out quickly.

11.36.24 <rulescript>: script to execute if rule applies

Contained in: <remoterule>
Available: in PRO versions only
Can contain: script (without variable declarations!)
Script context: session context
Attributes: none
Default: no script

This script is executed when the remote rule is applied to the session. It can be used to assign values to variables in the session context, which then can be used to control device-specific behaviour in all other scripts (using the SESSIONVAR built-in functions, see 6.14.6).
Note: it is not allowed to declare variables in <rulescript> - all variables to be accessed must be declared in the <sessioninitscript> (see 11.12).

12. <server type="sql"/"odbc">, <client type="sql"/"odbc">: SQL/ODBC based Server or Client Config

Contained in: <sysync_config>
Can contain: tags described in chapters 12 (see also 11, and 16 specifically for clients)
Attributes: type="sql" (or as an alias for backward compatibility: type="odbc")

This is where Synthesis Sync Server can be configured to work with almost any existing SQL database. It requires firm knowledge about SQL, your target database system and your database layout to do these changes successfully.
Please refer to the sample configuration available in the product distribution package to see complete configurations for different types of SQL databases.

Example:

```
<server type="sql">
  <!-- many contained tags, see section 12.2 ff -->
</server>
```

12.1 SQL Statement processing

In the 1.0.5 version of the Synthesis SyncML engine, SQL statements were generated by the server automatically from a few pieces of information in the config. While this was convenient for database layouts that resembled our standard sample layout, it was not flexible enough to adapt to any database. So since version 1.0.8, we have changed the basic mechanism how SQL statements are generated and processed:

- All SQL statements used to access the database are now fully customizable
- The engine provides a number of *placeholder sequences* (a % sign followed by one or multiple characters) that are replaced by variable data when the statement is executed.
- In most tags, where a SQL statement is expected, the engine also allows specifying multiple statements separated by the special sequence %GO
- For backward compatibility, the engine still allows using the 1.0.5-style configuration. However, for new projects, we strongly recommend using the new directives. The old style configuration might get unavailable in a future version of the Synthesis SyncML engine. In the following paragraphs, all old-style configuration tags are marked as such and mention which new tag should be used as a replacement.

The following paragraphs describe all the *placeholder sequences* that can be generally used in SQL statements in the configuration. There are additional placeholders that are only valid for specific SQL statemements (such as <writelogsql>, see 12.19); those are described with the statement they belong to.

Note that placeholder for strings always only represent the value, but not the eventually required quoting characters around the string value. This means that when using a placeholder that represents a string value in an SQL statement, the placeholder must be enclosed in quotes itself.

12.1.1 Placeholders for all SQL statements

%u represents the current *userkey*. The *userkey*is obtained with the <userkeysql> statement (see 12.16) and/or using the SETUSERKEY() script function in <logininitscript> (see 11.33) or <logincheckscript> (see 12.17). Note that the *userkey* has a meaningful value only after a successful login.

%d represents the current *devicekey*. The *devicekey*is obtained with the <getdevicesql> statement (see 12.14) and/or using the SETDEVICEKEY() script function in <logininitscript> (see 11.33) or <logincheckscript> (see 12.17).

%C PRO version only: represents the *domain*, which can be set in in <logininitscript> (see 11.33) or <logincheckscript> (see 12.17) with the SETDOMAIN() script function.

%sv(*var*) PRO version only: represents the value of the *session context variable* with the name *var*. Note that this works like the built-in function SESSIONVAR(), see 6.14.6. Note also that the %sv() placeholder simply represents the text string in the session variable as-is. If you need to insert timestamps or strings containing non-ASCII-characters, the contents of the session variable should be generated using the DBLITERAL() function, see 12.1.4.

%p(*mode,field_or_var [,dbfieldtype[,colsize]]*)
This is used to insert SQL parameters into a statement (normally a ODBC {call ...} statement to a stored procedure). *mode* can be "i" (for input-only parameters), "o" (for output-only parameters) or "io" for (input-output parameters). *field_or_var* is the name of an internal <field> (from the datatype's <fieldlist>, see 10.1) or script variable to be used for the parameter's value. The optional *dbfieldtype* must be the name of a database field type (same values as for "type" attribute in <map>, see 11.34.41.1) - if not specified, *dbfieldtype* defaults to "string". The optional *colsize* can be used to explicitly specify the column size for the parameter.
Note: **Make sure not to add extra spaces within the paranthesis, only the needed parameters and the separating commas!**

12.1.2 Placeholders for SQL statements within <datastore>

%GO This separates two SQL statements that need to be executed in sequence. Note that only the last SQL statement's result is evaluated (in case the SQL statement needs to return data at all).

%f represents the current *folderkey*. The *folderkey* is obtained with the <folderkeysql> statement (see 12.20.1). Note that the *folderkey* has a meaningful value only after a successful login.

%t represents the current *targetkey*. The *targetkey* is obtained with the <synctargetgetsql> statement (see 12.20.2)

%X represents a new record ID. Note that this only works if there is a method to create new IDs before actually inserting records (meaning that <obtainidafterinsert> must be false and there must be a <obtainlocalidsql> statement or a <specialidmode> set - see 12.20.18)

%x represents the most recently generated record ID. This is especially useful in layouts with one master record and several detail records that need to be linked to the master record by ID.

12.1.3 Placeholders for SQL data access statements within <datastore>

These placeholders are valid for all statements that access actual data records.

%GO(*set_no*) this special form of %GO can be used in front of any SQL statement. It modifies the operation of %N, %aN, %V, %aV, %v and %av (see below) as follows: From the list of mapped fields (see <map>, 11.34.41.1) only those will be included that have a corresponding *set_no*. The default *set_no* is 0 for all <map> entries that do not specifiy the *set_no* attribute. This allows for example to execute database update in multiple parts, each using a different SQL statement with a different *set_no* and therefore different fields.

%k represents the data key (local ID) of the master record inserted. Note that this is not available when inserting new records when <obtainidafterinsert> is true (as the local ID is only known after insert has taken place).

%pkos represents a string-type output parameter that receives the data key (local ID) of the record inserted. This is useful if inserting is not done with INSERT, but using a stored procedure which returns the generated ID as an SQL output Parameter.

%pkoi same as %pkos, but for integer-type output parameters.

%N represents the database field list for a SELECT or an INSERT statement. For data writing, the list will only contain the fields which need to be written.

%aN same as %N, but list contains all mapped fields, even if they do not need to be changed.

%V represents the fieldname = value pair list for an UPDATE statement. The list will only contain the fields/values which need to be updated. Note that the values might be represented as literals (normally) but als as ODBC parameters (for fields where no literal representation exists, such as BLOBs).

%aV same as %V, but list contains all mapped fields, even if they do not need to be updated.

%v represents the list of values for an INSERT statement's VALUE part, matching the field names generated with %N.

%av represents the list of values for an INSERT statement's VALUE part, matching the field names generated with %aN. (Note that in servers before 2.1.1.7 there was a bug that requires to write "%av(" instead of "%av").

%d([*opts*]*fieldname,dbfieldtype*) or %d([*opts*]*fieldname#arrindex,dbfieldtype*) represents the value of an internal field (from the datatype's <fieldlist>, see 10.1) or context variable contents (see 6.9.1) named *fieldname*, where *opts* can be "l" for lowercase, "u" for uppercase and "a" for ASCII-only. If the field is an array field, the second form allows accessing the *arrindex*-1 th element from the array. *dbfieldtype* is optional and specifies how to format the field for the database (same values as for "type" attribute in <map>, see 11.34.41.1) - if not specified, *dbfieldtype* defaults to

"numeric" (which means that the value of the field is inserted 1:1 into the SQL without surrounding quotes). This is to be compatible with the behaviour of %d in versions before 2.1.1.15, where the *dbfieldtype* parameter did not yet exist, and %d was often used inside quotes to insert string values.
Note: **Make sure not to add extra spaces within the paranthesis, only the needed parameters and the separating commas!**

%S represents the number of bytes (not necessarily equal to the number of characters for character sets with multi-byte characters such as UTF-8) of all strings in a writing statement. This might be useful for tracking database space requirements.

%dM,%tM,%M
represents the date (%dM), time (%tM) or datetimestamp (%M) of the last modification of the record. Note that for a 100% multi-user SyncML server, the SyncML server must be able to explicitly set the modification date for all records it writes (as all modifications of one sync session should have exactly the same date). This is only important if it is possible that records are being changed on the server while a sync session is in progress.

%AF this expands either to an empty string or to a WHERE clause condition preceeded by an AND. The condition is derived from the active filter settings (see 7). This placeholders can be used in SELECT statements that already have a WHERE clause, so the WHERE conditions will be extended with the filter conditions if there are any. Note that automatic conversion from filter settings to an SQL WHERE clause might not work in all cases. Therefore, the ability to filter at the DB level using %AF and %WF is disabled by default. This will cause the filters to be applied to the data after fetching it - which is less efficient but works in all cases. Set <dbcanfilter> to true (see 12.20.9) to enable filtering at the database level.

%WF this is similar to %AF, but instead of AND, a WHERE is inserted if there is a filter condition at all. This placeholders can be used in SELECT statements that do not have a WHERE clause, so a WHERE clause will be added only when there are filter conditions (and <dbcanfilter>, see 12.20.9, is true).

12.1.4 Executing SQL statements from scripts

In PRO versions of the SyncML server and client, it is also possible to programmatically execute arbitrary SQL statements. This is especially useful for implementing special multi-step login procedures or similar tasks.

The following SQL processing script functions are available in the <logininitscript> (see 11.33), the <logincheckscript> (see 12.17) and all scripts in the SQL datastore context (e.g. <initscript>, see 11.34.41.3).

string DBLITERAL(variant *value*, string *dbfieldtype*): This is used to convert a *value* into a string that is suitable as literal value in SQL statements to be sent to the database, as required for example in SETSQLFILTER(). *dbfieldtype* specifies the database field type (as in "type" attribute of <map>, see 11.34.41.1).

integer SQLEXECUTE(string *sql*): This can be used to execute a SQL statement. Note that the statement is executed in a transaction context global to all datastores. Make sure that you use SQLCOMMIT or SQLROLLBACK to avoid unfinished transactions.

Returns 0 if statement executed ok, or a non-zero ODBC error code in case of failure.

integer SQLFETCHROW(): Fetches the next row from a result set of a preceeding SQLEXECUTE call. If no (more) rows are available, SQLFETCHROW returns 0, otherwise it returns 1.

integer SQLGETCOLUMN(integer *colindex*, variant &*value*, string *dbfieldtype*): This reads the *colindex*-th column from the currently fetched row from a result set into the variable *value*. *dbfieldtype* specifies the database field type expected (as in "type" attribute of <map>, see 11.34.41.1). Note that if SQLGETCOLUMN is called multiple times for a single row, *colindex* must be increasing for each call (this is an ODBC requirement).

integer SQLITELASTID(): For SQLite only: returns the ROWID created by the last INSERT statement.

SQLCOMMIT(): Commits the current transaction. Make sure you call SQLCOMMIT or SQLROLLBACK before ending scripts that use SQLEXECUTE, especially when the SQL statemement was a DML statement (modifiying data in the database).

SQLROLLBACK(): Performs a rollback on the current transaction. Make sure you call SQLCOMMIT or SQLROLLBACK before ending scripts that use SQLEXECUTE, especially when the SQL statemement was a DML statement (modifiying data in the database).

The following two functions are available only for ODBC, but not for SQLite:

SETDBCONNECTSTRING(string *dbconnectstring*): This can be used to programmatically set the ODBC database connection string (which is normally set using <dbconnectionstring>, see 12.4). Note that while this is technically possible in many scripts, it usually makes sense only in <logininitscript> (see 11.33) to eventually choose different databases depending on login information. Using SETDBCONNECTSTRING closes the currently open session level ODBC connection (if any), but does not open a new connection immediately. The new connection string and password are used to open a new connection as soon as the next SQL statement needs to be executed.

SETDBPASSWORD(string *dbpassword*): This can be used together with SETDBCONNECTSTRING to set the password for opening a DB connection.

12.2 <datasource>: ODBC data source name

Contained in: <server>, <client>
Can contain: name of ODBC data source
Attributes: none

This tag specifies the name of the ODBC data source. This is the name that was set when creating the ODBC data source (see chapter 2).
Note that for some data sources that need additional parameters (or if you want to connect without a data source by directly specifying ODBC driver parameters), you might need to use <dbconnectionstring> (see 12.4) instead of <datasource> and <dbuser>.

12.3 <dbuser>: ODBC database user name

Contained in: <server>, <client>
Can contain: user name for accessing the ODBC data source
Attributes: none

This tag specifies the name of the user that is used to access the ODBC data source.
Note that for some data sources that need additional parameters (or if you want to connect without a data source by directly specifying ODBC driver parameters), you might need to use <dbconnectionstring> (see 12.4) instead of <datasource> and <dbuser>.

12.4 <dbconnectionstring>: ODBC database connection string

Contained in: <server>, <client>
Can contain: ODBC connection string
Attributes: none

This tag can be used as an alternative to <datasource> and <dbuser> in cases when you need to pass more parameters to the ODBC engine than just datasource name and user name. For example some versions of the MyODBC driver need the "DATABASE" parameter to be able to connect correctly. Using <dbconnectionstring> it is also possible to connect directly to a ODBC driver without using a datasource by including all the driver-specific parameters in the <dbconnectionstring> (this is for adcanced ODBC users only - see ODBC documentation).

Important note: the <dbconnectionstring> is shown in debug logs for reference. Therefore we recommend to *not* including the PWD parameter in the <dbconnectionstring> but specifying it with <dbpass> (see 12.5) - which causes the PWD parameter to be automatically appended to the <dbconnectionstring> before it is sent to the ODBC engine (but *after* it is shown in the log).

The following three examples are functionally identical:

```
<!-- using datasource, dbuser, dbpass: -->
<datasource>mydatasource</datasource>
<dbuser>test</dbuser>
<dbpass>secret</dbpass>

<!-- using dbconnectionstring and dbpass: -->
<dbconnectionstring>DSN=mydatasource; UID=test;
DATABASE=syncml; </dbconnectionstring>
<dbpass>secret</dbpass>

<!-- using only dbconnectionstring, not recommended be-
cause password will be shown in logfiles -->
<dbconnectionstring>DSN=mydatasource; UID=test;
DATABASE=syncml; PWD=secret;</dbconnectionstring>
```

12.5 <dbpass>: ODBC database password

Contained in: <server>, <client>
Can contain: password for accessing the ODBC data source
Attributes: none

This tag specifies the password required to access the ODBC data source with the user name specified in <dbuser> (or in the <dbconnectionstring>, see 12.4). Please make sure that the config file is not accessible from the outside in order not to compromise your database's security.

12.6 <preventconnectattrs>: prevent setting connection attributes

Contained in: <server>, <client>
Can contain: boolean value
Attributes: none

This can be set to true to prevent that any attempt to set ODBC connection attributes is made at all (some ODBC drivers exists which will crash when trying to modify connection attributes).

12.7 <dbtimeout>: ODBC timeout

Contained in: <server>, <client>
Can contain: timeout value in seconds, 0 for no timeout
Attributes: none
Default: 30 seconds

Defines the timeout for ODBC.
Please note that this does not work with every ODBC driver, so this setting might have no effect at all with some ODBC databases.

12.8 <afterconnectscript>: Script executed whenever new DB connection is opened.

Contained in: <server>, <client>
Available: in PRO versions only
Can contain: script
Script context: afterconnect context (private only for this script)
Attributes: none
Default: no script

This script is executed once immediately after opening a new database connection. This can be used to execute extra SQL statements (such as access authorisation for SyBase) using the SQLxxx() functions (see 12.1.4).

12.9 <transactionmode>: Transaction isolation mode

Contained in: <server>, <client>
Can contain: transaction isolation mode
Attributes: none
Default: "default"

This tag is used to defined the transaction isolation mode to be used. It can be set as follows:

- "**none**" : no isolation mode
- "**default**": use default isolation mode of the ODBC driver
- "**read-uncommitted**": other transactions see all changes done. This is the least restrictive mode. **If you experience locked table problems, try using this mode.**
- "**read-committed**": other transactions see committed changes
- "**repeatable**": reads are repeatable with same results
- "**serializable**": full serializable isolation

Please note that not all ODBC drivers support all these modes. Setting a mode that is not supported may result in fatal errors preventing the sync server from working at all.

12.10 <usecursorlib>: usage of ODBC cursor library

Contained in: <server>
Can contain: boolean value
Attributes: none
Default: false

This tag is used to select if the ODBC cursor library should be used or not. Under normal circumstances, for the access patterns generated by a SyncML client or server, the cursor library is not required and only generates overhead if switched on.
Please note that some ODBC drivers may not work properly with cursor library switched on or off, so in case of strange ODBC behaviour, you might want to try changing this flag.

12.11 <textmap>, <textauth>, <textpath>: outdated - no longer available

Contained in: <datastore>
Available: no longer available in 3.0 version

These tags were used to control text-file based administration tables and/or user auth. This has been replaced by the plugin mechanism (see 14) and the built-in SDK_textdb plugin module (see 14.3).
To use SQL for the data tables, but have all administrative data stored in text files (as it is for example required for SQLite based setups, see 12.20.6), just use the SDK_textdb plugin in <plugin_module> and enable <plugin_deviceadmin>/ <plugin_sessionauth> at the session level (see 14.1) and use the SDK_textdb plugin in <plugin_module_admin> at the datastore level (see 14.2).

12.12 <cleartextpw>: plain text password in database

Contained in: <server>
Can contain: boolean value
Attributes: none
Default: on

This option (together with <md5userpass> selects how Synthesis Sync Server authenticates with the database:

- If <cleartextpw> is set, the database must be able to return the password for a user in clear text. The Sync Server then uses that password, together with an eventual nonce string (see 11.7) to calculate the MD5 digest to compare with the client's authentication attempt.
- If <cleartextpw> and <md5userpass> is not set, the database must be able to check authentication (and perform MD5 calcualtions required) for a given combination of username, MD5 digest and nonce string.

In most cases, <cleartextpw> should be set. **Even if passwords are transferred in clear text between database and sync server, they are NOT transferred in clear text over the net.** So if your database and web server is properly secured, <cleartextpw> set does not compromise security. Setting <md5userpass> prevents clear text passwords to be stored in the database, however due to a design problem in SyncML 1.0 (SyncML 1.1 has solved that), this will prevent that clients can use the MD5 digest authentification. **This is more of a security risk than using <cleartextpw>, because when clients cannot use MD5 authentification they must use basic authentification which means sending passwords almost clear text (only B64 encoded!) over the internet.**

12.13 <md5userpass>: MD5 digest passwort in database

Contained in: <server>
Can contain: boolean value
Attributes: none
Default: on

This option selects another mode of how Synthesis Sync Server can authenticates with the database:

- If <md5userpass> is set, the database must be able to return a string which is either the B64 encoding of the MD5 digest of "user:password" (<md5hex> set to false, see 12.14) or a 32-char hex string encoding the MD5 digest (<md5hex> set to true) for each user. The Sync Server then uses that information to verify client's credentials. See "0" for security considerations.

12.14 <md5hex>: MD5 digest stored as hex string in database

Contained in: <server>
Can contain: boolean value
Attributes: none
Default: false
New in: 3.0.0.17

This option selects the format of the MD5 digest when stored in the database (see 12.13). With md5hex set to true, the format is a 32-char string of hexadecimal digits (for example compatible with the md5() function in PHP). Otherwise, the MD5 digest must be stored in B64 encoding.

12.15 <getdevicesql>, <newdevicesql>, <savenoncesql>, <saveinfosql>: Device management

Contained in: <client>, <server>
Can contain: SQL statements
Attributes: none
Default: none

These tags can hold SQL statements to manage a *device table* in the database. If these statements are not defined, the SyncML engine can also work without a *device table*, but it is recommended to have a device table. The device table will receive a single entry for every device that has ever connected to the server.

The table for device management must have at least the following fields:

- a ***device key*** as a primary key to the devices table.
- a ***device id*** field, which should be a string of at least 50 characters. For better performance, this field should be indexed.
- a ***last nonce*** field, which should be a string of at least 20 characters.

Optionally, the following fields are recommended

- a ***device name*** field, which should be a string of about 60 characters, which will be assigned the descriptive name of the device.
- a ***device info*** field, which should be a string of about 120 characters, which will be assigned some additional information about the device (type, version, OEM manufacturer).

The four SQL statements related to device management must do the following:

- **<getdevicesql>** must be a SELECT statement returning for a given *device id* a result set with a single row and two columns: first column must be the *device key*, second column must be the *last nonce*. If the *device id* is not known yet, this statement must return an empty result set. To include the *device id* in the SELECT's WHERE clause, %D can be used. **Note that all other % placeholders described in 12.1 do not apply here!**

- **<newdevicesql>** must create a new record for the *device id* with a new, unique *device key* (by means of an autoincrementing device key field in the database or another database mechanism such as a generator, sequence or trigger). Normally, this is a simple INSERT statement. To insert the *device id* into the statement, %D can be used. **Note that all other placeholders described in 12.1 do not apply here!**

- **<savenoncesql>** must update the *last nonce* for a specified *device key*. To insert the device key (for example in the WHERE clause), %d can be used. To insert the *last nonce*, %N can be used. In addition, placeholders described in 12.1.1 can also be used.

- **<saveinfosql>** is an optional statement that can be used to save *device name* and *device info* information. To insert the device key (for example in the WHERE clause), %d can be used. To insert the *device name* %nR can be used, to insert the *device info*, %vR can be used. In addition, placeholders described in 12.1.1 can also be used.

The following example shows a complete set of SQL statements to manage devices in a table called SYNC_DEVICES:

```
<getdevicesql>
SELECT DEVICE_KEY,LASTNONCE FROM SYNC_DEVICES WHERE DEVICEID='%D'
</getdevicesql>

<newdevicesql>
INSERT INTO SYNC_DEVICES (DEVICEID) VALUES ('%D')
</newdevicesql>

<savenoncesql>
UPDATE SYNC_DEVICES SET LASTNONCE='%N' WHERE DEVICE_KEY=%d
</savenoncesql>

<saveinfosql>
UPDATE SYNC_DEVICES SET DEVICENAME='%nR', DEVICEINFO='%vR' WHERE DEVICE_KEY=%d
</saveinfosql>
```

12.16 <userkeysql>: query for user authentication

Contained in: <client>, <server>
Can contain: SQL query statement
Attributes: none

This tag specifies the SQL statement that is used to check authorisation of a user in the database.

The SQL string specified can contain placeholders to insert values into the query. In addition to the general placeholders (see 12.1.1), the following special placeholders are available in <userkeysql>:

%U represents the original user name as sent by the remote device.

%dU represents the user name that has been set with the SETUSERNAME() function in <logininitscript>.

%D represents the "domain" value that has been set with the SETDOMAIN() function in <logininitscript>.

%M represents the credential string (plain text password or MD5 digest, depending on login type)

%N represents the nonce string.

Depending on how <cleartextpw> is set (see 12.12), this SQL statement must do the following:

- If <cleartextpw> is set, the statement must return a row for every user with a given name having the following result columns (in that order):
  - a "user key": This is a value (normally a primary key into the user table) that uniquely identifies the user and which can be used in further queries to subselect this user's data folder(s) (see 12.20.1). This value is treated as a string, but can be of any data type that has a unambiguous string representation.
  - the password for that user in clear text.

  If no user with the given name exist, an empty result set (no rows) must be returned.
- If <md5userpass> is set, the statement must return a row for every user with a given name having the following result columns (in that order):
  - a "user key": This is a value (normally a primary key into the user table) that uniquely identifies the user and which can be used in further queries to subselect this user's data

folder(s) (see 12.20.1). This value is treated as a string, but can be of any data type that has a unambiguous string representation.
- the B64-encoded MD5-digest of the string "user:password".

If no user with the given name exist, an empty result set (no rows) must be returned.

- If neither <cleartextpw> nor <md5userpass> are set, the statement must return exactly one row as a result when there is a user that matches the given username, MD5-digest and nonce string, having the following result column:
  - a "user key": This is a value (normally a primary key into the user table) that uniquely identifies the user and which can be used in further queries to subselect this user's data folder(s) (see 12.20.1). This value is treated as a string, but can be of any data type that has a unambiguous string representation.

  If the given combination of username, MD5 digest and nonce are not valid, the query must return an empty result set (no rows).

Note: When using <logincheckscript> (see 12.17), the columns returned by <userkeysql> are assigned to the local variables of the logincheckscript in the order of their definition. Therefore, the <userkeysql> may be written such that it returns more columns than those described above. This allows fetching extra user data needed to decide about login in the logincheckscript.

Example with <cleartextpw> set:

```
<userkeysql>SELECT USER_KEY,PASSWD FROM SYNC_USERS WHERE
USERID='%U'</userkeysql>
```

Example with <md5userpass> set:

```
<userkeysql>SELECT USER_KEY,MD5DIGEST FROM SYNC_USERS
WHERE USERID='%U'</userkeysql>
```

Example with neither <cleartextpw> nor <md5userpass> set (note that this assumes the presence of a non-standard SQL function "B64MD5").

```
<userkeysql>SELECT USER_KEY FROM SYNC_USERS WHERE
B64MD5('%U',PASSWD,'%N')='%M'</userkeysql>
```

12.17 <logincheckscript>: custom login checking script

Contained in: <server>
Available: in PRO versions only
Can contain: script returning boolean value
Script context: login context
Attributes: none
Default: no script

This script is executed for every row returned by <userkeysql> (see 12.16). Local variables of this script will be initialized with values returned by <userkeysql>.
The script has access to the same special script functions as <logininitscript> (see 12.13 for details).

If this script returns TRUE, login is granted.

12.18 <timestampsql>: query for getting database time

Contained in: <server> or <client>
Can contain: SQL query statement
Attributes: none
Default: not specified

This tag specifies the SQL statement that is used to get a timestamp of the database server's current time.
If this tag is not specified, Synthesis Sync Server uses the local time of the machine it is run as current time.
It is recommended not to leave this unspecified, because time differences between database and sync server can lead to data consistency problems.
The query must return a single row with a single column that represents a timestamp (combined date and time) value.

Note that <datatimezone> (see 11.34.31) determines in what time zone context the result of this query is to be treated.

Example for Microsoft SQL server:

```
<timestampsql>SELECT GETDATE() AS
CURRENTDATETIME</timestampsql>
```

12.19 <writelogsql>: SQL statement to write activity log entry

Contained in: <server> or <client>
Can contain: SQL statement
Attributes: none
Default: false

This statement is executed for each datastore involved in a sync session after the sync session has either completed (with or without errors) or timed out (see 11.2 how to set session timeout). Usually, this is an SQL INSERT statement into a global log table.

In the <writelogsql> statement, all escape sequences described in <logformat> (see 11.31) plus the following special escape sequences can be used to insert information from the current session into the log:

%dT Sync date

%tT Sync time

%T Sync timestamp (reference time for comparisons of changed/unchanced decisions of last session)

%ssT Sync start timestamp (when this sync attempt, successful or not, has started)

%seT Sync end timestamp (when this sync attempt, successful or not, has ended)

%f folder key (see 12.20.1)

%t target key (see 12.20.2)

%u user key (see 12.16)

%d device key (see 12.14)

12.20 <datastore type="sql"/"odbc">: SQL and ODBC Datastore specific settings

Contained in: <server type="sql"/"odbc"> or <client type="sql"/"odbc">
Can contain: All tags in chapter 12.20
Attributes: name, type="sql" (or as an alias for backward compatibility: type="odbc")
Default: not specified

This chapter describes the tags that are specific to SQL and ODBC datastores. See 11.34 for a description of the <datastore> tag in general.

Note: Starting with version 3.0.3.0, the SQL datastore type also supports direct access to SQLite database files instead of using ODBC for accessing the user data. Admin data (SYNC_TARGETS, SYNC_MAPS, SYNC_DEVICES, SYNC_LOGS in the sample config) however is not supported for SQLite at this time. See 12.20.6 for details about SQLite related config. **Please also note that SQLite support may not be included in some products.**

12.20.1 <folderkeysql>: get data subselection key

Contained in: <datastore>
Can contain: SQL query string
Available for: ODBC only (not SQLite)
Attributes: none
Default: empty

This tag specifies the SQL statement that is used to obtain a key value that can be used in subsequent SQL statements to sub-select data in a so-called folder.
Folders are subdivisions of a user's data. For example, a user might have two separate database for work and private use on his SyncML-enabled PDA. To allow this on the server side, this user can be given two separate "folders" to store work and private contacts separated from each other.

On the SyncML client, the folder name is specified in the target database path. For example, if the database path for contacts is "./contacts", then a folder named "private" will be addressed as "./contacts/private".

This tag can be omitted or left empty if the server does not support multiple folders per user. In this case, every user will only have a single folder. In this case, the user key (or user name when using <textauth> (see 12.11) or <simpleauthuser>, <simpleauthpw> (see 11.10)) is used as folder key, allowing to differentiate records in the data table by user.

The SQL string specified can contain placeholders to insert the following values into the query:

%F represents the folder name. If the SyncML client specifies no folder name (eg. just "./contacts"), the folder name is an empty string.

%u represents the user key. The user key is the value that was returned by the <userkeysql> statement (see 12.16).

The query must return:

- An empty result set when the folder does not exist or the user is not allowed to access it
- At least one row with one single column containing the folder key. This value is treated as a string, but can be of any data type that has a unambiguous string representation.

Example (using a link table between folders and users to allow users sharing folders):

```
<folderkeysql>SELECT FOLDER_KEY FROM SYNC_FOLDERS F JOIN
SYNC_PERM P ON P.FOLDERKEY=F.FOLDER_KEY AND P.USERKEY=%u
WHERE F.FOLDERID='%F'</folderkeysql>
```

12.20.2 <synctargetgetsql>, <synctargetnewsql>, <synctargetupdatesql>, <synctargetdeletesql>: Sync target management

Contained in: <datastore>
Can contain: SQL query strings
Available for: ODBC only (not SQLite)
Attributes: none

These four SQL statements are used to manage *sync target* information. The sync server needs to remember some information for every client database (called *sync target*) it does a sync session with.
This information is stored in an auxiliary table which can exist once for the entire server (as in the config samples, where a separate field DSCODE is used to separate targets from different datastores, see example below) or in a separate table for every datastore (for example if varying custom data needs to be in the same tables for each datatype).

The SQL strings specified can contain the following special placeholders (in addition to the standard placeholders as described in 12.1.1) :

%f represents the folder key. The folder key is the value that was returned by the <folderkey> statement (see 12.20.1).

%u: represents the user key. The user key is the value that was returned by the <userkeysql> statement (see 12.16).

%t: represents the target key (only for <synctargetupdatesql>).

%D: represents the device ID of the SyncML client (client URI).

%P: represents the client's remote database path (client datastore URI).

%L: represents the timestamp of last sync (only for <synctargetupdatesql>).

%dL: represents the date of last sync (only for <synctargetupdatesql>).

%tL: represents the time of last sync (only for <synctargetupdatesql>).

%S: represents the timestamp of last server anchor (only for <synctargetupdatesql>). This (and the following two) is needed only when <synctimestampatend> (see 11.34.37) is set.

%dS: represents the date of last last server anchor (only for <synctargetupdatesql>).

%tS: represents the time of last last server anchor (only for <synctargetupdatesql>).

%RL: represents the timestamp of last sync where remote party was updated (only for <synctargetupdatesql>. This (and the following five) is needed only if <fromremoteonlysupport> (see 11.34.36) is set.

%dRL: represents the date of last sync (only for <synctargetupdatesql>).

%tRL: represents the time of last sync (only for <synctargetupdatesql>).

%iRL: represents the custom identifier for the last sync where remote party was updated. This is needed only when <storesyncidentifiers> (see 11.34.38) is set.

%RS: represents the timestamp of last server anchor (only for <synctargetupdatesql>). This (and the following two) is needed only when <synctimestampatend> (see 11.34.37) is set.

%dRS: represents the date of last last server anchor (only for <synctargetupdatesql>).

%tRS: represents the time of last last server anchor (only for <synctargetupdatesql>).

New for Version 3.0 (only needed if <resumesupport> is set – see 11.34.39)

%SUA: represents the resume alert code (only for <synctargetupdatesql>).

%SU: represents the timestamp of last suspend (only for <synctargetupdatesql>)..

%dSU: represents the date of last last suspend (only for <synctargetupdatesql>).

%tSU: represents the time of last last suspend (only for <synctargetupdatesql>).

%iSU: represents the custom identifier for the last suspend. This is needed only when <storesyncidentifiers> (see 11.34.38) is set.

New for Version 3.0 (only needed if <resumeitemsupport> is set – see 11.34.40).

%pSU: represents the Source URI string of the item that must be resumed in the next session.

%pTU: represents the Target URI string of the item that must be resumed in the next session.

%pSt: represents the partial item status code (numeric).

%pM: represents the internal partial item mode (numeric).

%pTS: represents the partial item total size (numeric).

%pUS: represents the partial item unconfirmed size (numeric).

%pSS: represents the partial item stored size (numeric), which is the number of bytes that will be stored into the partial item data field.

%pDAT: represents the partial item date (which is a BLOB of the size indicated by %pSS). This value is not literally inserted into the SQL statement, but as a parameter.

The four SQL statements are used to look up, create, modify and delete sync target records.

1. The statement specified in **<synctargetgetsql>** must look up if there is already a sync target record matching user, folder, deviceID, remote database path and if yes, return a single row with the following columns (in that order!):
   - **target key**: this is a value uniquely identifying the target. Normally this is the primary key of the sync target table. It is used to subselect map entries (see below) in the map table. This value is treated as a string, but can be of any data type that has a unambiguous string representation.
   - **sync anchor**: this is a string value which is set by the sync server. The database table should simply provide a string field of 40 chars size.
   - **timestamp of last successful sync**: This is set by the sync server. The database table must provide a column that can hold timestamp values when <synctimestamp> (see 12.20.3) is set, if <synctimestamp> is not set, it must provide a date and a time column).
   - Only if <synctimestampatend> (see 11.34.37) is set, a **separate timestamp used for last server anchor** (if <synctimestampatend> is not set, server anchor and last sync timestamps are identical and need to to be saved separately). The database table must provide a column that can hold timestamp values when <synctimestamp> is set, if <synctimestamp> is not set, it must provide a date and a time column) .
   - Only if <fromremoteonlysupport> (see 11.34.36) is set, a **timestamp of last successful sync when remote was updated**. This is needed to allow one-way from remote sync sessions, and is set by the sync server. The database table must provide a column that can hold timestamp values when <synctimestamp> (see 12.20.3) is set, if <synctimestamp> is not set, it must provide a date and a time column) .
   - Only if <fromremoteonlysupport> and <synctimestampatend> (see 11.34.37) is set, a **separate timestamp used for last server anchor** (if <synctimestampatend> is not set, server anchor and last sync timestamps are identical and need to to be saved separately). The database table must provide a column that can hold timestamp values when <synctimestamp> is set, if <synctimestamp> is not set, it must provide a date and a time column) .
   - Only if <storesyncidentifiers> (see 11.34.38) is set, a **string identifying the point in time of the last sync for the datastore implementation (plugin)**. The format of this string is free and only depends on the datastore implementation in the plugin. This is used when the plugin uses a private count or timestamp for detecting changes since last sync.

   New for version 3.0: For databases supporting SyncML DS 1.2 Suspend & Resume (those that have <resumesupport> enabled, see 11.34.39), the following additional columns must be returned:
   - **resume alert code:** this must be at least a 16-bit integer.
   - **timestamp of last suspend**: This is set by the sync server. The database table must provide a column that can hold timestamp values when <synctimestamp> (see 12.20.3) is set, if <synctimestamp> is not set, it must provide a date and a time column).
   - Only if <storesyncidentifiers> (see 11.34.38) is set, a **string identifying the point in time of the last suspend for the datastore implementation (plugin)**. The format of this string is free and only depends on the datastore implementation in the plugin. This is used when the plugin uses a private count or timestamp for detecting changes since last suspend.

   New for version 3.0: Only if <resumesupport> and <resumeitemsupport> (see 11.34.40) are enabled, the following additional columns must be returned:

- **last source URI:** a string used by the sync server for resuming partial items.
- **last target URI:** a string used by the sync server for resuming partial items.
- **last item Status:** this must be at least a 16-bit integer.
- **partial item State:** this must be at least a 8-bit integer.
- **total item size:** this must be at least a 32-bit integer.
- **unconfirmed item size:** this must be at least a 32-bit integer.
- **stored item size:** this must be at least a 32-bit integer, and represents the size of the data stored in the "partial item" column (see below).
- **partial item:** this must be a BLOB column that can hold an arbitrary block of binary data, with potentially 2^32 bytes of size. In reality, this block will never exceed the maximum size of data objects supported. But for example for emails with attachments, the partial item can get quite large.

2. The statement specified in **<synctargetnewsql>** must insert a new record into the sync target table, which is uniquely defined by user, folder, deviceID, and remote database path.

3. The statement specified in **<synctargetupdatesql>** must update all values fetched by <synctargetgetsql> for a specified targetkey.

4. The statement specified in **<synctargetdeletesql>** must delete an existing sync target record identified by a target key value. In addition, it should make sure that all related map entries (see 12.20.5) are also deleted (by an implicit ON DELETE CASCADE constraint or explicitly by including more than one DELETE statement into this tag, which can be done by using the %GO special sequence, see 12.1.2).

Example (what is new for version 3.0 is marked red):

```
<synctargetgetsql>
SELECT TARGET_KEY, ANCHOR, LASTSYNC, LASTTOREMOTESYNC,
RESUMEALERT, LASTSUSPEND, LISOURCE, LITARGET, LISTATUS,
PISTATE, PITOTALSZ, PIUNCONFSZ, PISTOREDSZ, PIDATA FROM
SYNC_TARGETS WHERE DSCODE='no' AND USERKEY=%u AND
FOLDERKEY=%f AND DEVICEKEY=%d AND DEVICEDBPATH='%P'
</synctargetgetsql>

<synctargetnewsql>
INSERT INTO SYNC_TARGETS (DSCODE, USERKEY, FOLDERKEY,
DEVICEKEY, DEVICEDBPATH) VALUES ('no', '%u', %f, %d,
'%P')
</synctargetnewsql>

<synctargetupdatesql>
UPDATE SYNC_CONTACTS_TARGETS SET ANCHOR='%A', LASTSYNC=%L
WHERE TARGET_KEY='%t'
UPDATE SYNC_TARGETS SET ANCHOR='%A', LASTSYNC=%L,
LASTTOREMOTESYNC=%RL, RESUMEALERT=%SUA, LASTSUSPEND=%SU,
LISOURCE='%pSU', LITARGET='%pTU', LISTATUS=%pSt,
PISTATE=%pM, PITOTALSZ=%pTS, PIUNCONFSZ=%pUS,
PISTOREDSZ=%pSS, PIDATA=%pDAT WHERE TARGET_KEY=%t
</synctargetupdatesql>
```

```
<synctargetdeletesql>
DELETE FROM SYNC_TARGETS WHERE TARGET_KEY=%t
%GO DELETE FROM SYNC_MAPS WHERE TARGET_KEY=%t
</synctargetdeletesql>
```

12.20.3 <synctimestamp>: format for timestamps in target table

Contained in: <datastore>
Can contain: boolean value
Available for: ODBC only (not SQLite)
Default: yes

If this tag is set to yes, timestamp values in the sync target information are stored in timestamp columns (one column containing combined date/time value). Otherwise, timestamp values are stored as a pair of separate date and time columns. Note that SQL statements (see 12.20.2) must be formed according to this setting.

12.20.4 <lastmodfieldtype>: modified time stamp type

Contained in: <datastore>
Can contain: database field type for the modified time stamp field
Attributes: none
Default: timestamp

This tag specifies the database field type used as modified timestamp. For SQL/ODBC databases this is usually "timestamp" (the default), but for SQLite databases which do not have a native timestamp format, this can be set to one of the integer timestamp formats (such as UNIX epoch time). See description of "type" attribute in 11.34.41.1 for possible values **but note that only timestamp-related field types makes sense here!**

12.20.5 <selectmapallsql>, <insertmapsql>, <updatemapsql>, <deletemapsql>: Map table management

Contained in: <datastore>
Can contain: SQL statements
Available for: ODBC only (not SQLite)
Attributes: none
Default: none

These four SQL statements are used to manage *object ID mapping* information. The sync server needs to create a map entry for every object synchronized between the server and a specific device. A map entry links the unique *localID* - normally a primary key of the data table in the local server database - to the corresponding unique *remoteID* for the same object in the remote SyncML client's database. As there are separate map entries for each database on each device, the map table entries must be related to a *sync target* (see 12.20.2).

Note that map entries may not be related to the data records via localID in a way that would cause automatic deletion of the map entry when the data record is deleted (such as with a ON DELETE constraint)! It is essential that map entries remain existing after the corresponding data records are deleted - this is required by the sync engine to detect and propagate delete operations.

The SQL strings specified can contain the following special placeholders (except selectmapallsql) in addition to the standard placeholders described in 12.1.2:

%k represents the *localID*. The *localID* is normally the primary key into the database table containing the actual data records (such as contact records or events). Depending on the type of key, this might be a string or a numeric value.

%r: represents the *remoteID*. This must always be treated as a string - normally it is not more than 30 characters in length, but some clients (mostly those that have to deal with Microsoft Exchange) use very long IDs so we recommend reserving 64 or even 128 characters here.

New for version 3.0: (only needed if <resumesupport> is set – see 11.34.39)

%e: represents the *entryType*. This is a small integer (8 bits are sufficient) and determines the type of the map entry – because for some SyncML DS 1.2 features, different types of map entries are required. **It is important to include *entryType* into the primary key for the map table** (and no longer only *localID* and *targetKey*) because multiple map entries with the same *localID*, but different *entryType* can exist for the same *target-Key*).

%x: represents the *mapFlags*. This is a 32 bit integer and stores extra flags for each map item required for SyncML DS 1.2 Suspend&Resume and other advanced features.

The four SQL statements are used to read the entire map table and to add, update and delete single map entries.

1. The statement specified in <selectmapallsql> must return a result set containing the entire map table for the current *sync target* (normally there is a WHERE clause including %t to restrict the SELECT to the map entries of one *sync target*) in two columns (in that order!):
   - **localID**: this column must contain the *localID*, which can be a string or a numeric value, depending on the type of primary key the data table has.
   - **remoteID:** this column must contain the *remoteID*, which always is a string.

   New for version 3.0: For databases supporting SyncML DS 1.2 Suspend & Resume (those that have <resumesupport> enabled, see 11.34.39), the following additional columns must be returned:
   - **entryType:** this column must contain the *entryType*, a small integer (8 bit is sufficient).
   - **mapFlags:** this column must contain the *mapFlags*, which is at least a 32 bit integer value.

2. The statement specified in <insertmapsql> must insert a new record into the map table, which is related to a *sync target* and contains both *localID* and *remoteID*.

3. The statement specified in <updatemapsql> must update the *remoteID* for a given *localID* related to a *sync target*.

4. The statement specified in <deletemapsql> must delete the map entry with a given *localID* related to a *sync target*.

Example SQL definition of a map table (what is new for version 3.0 is marked red):

```
CREATE TABLE SYNC_CONTACTS_MAP
(
  LOCALID        INTEGER NOT NULL,
  REMOTEID       VARCHAR(63),
  TARGETKEY      INTEGER NOT NULL,
  ENTRYTYPE      INTEGER DEFAULT 1,
  FLAGS          INTEGER DEFAULT 0,
  PRIMARY KEY    (LOCALID,ENTRYTYPE,TARGETKEY)
)
```

Example config to access the map table defined above (new for version 3.0 in red):

```
<selectmapallsql>
SELECT LOCALID, REMOTEID, ENTRYTYPE, FLAGS FROM SYNC_MAPS
WHERE TARGETKEY=%t
</selectmapallsql>

<insertmapsql>
INSERT INTO SYNC_MAPS (LOCALID, REMOTEID, TARGETKEY,
ENTRYTYPE, FLAGS) VALUES (%k,'%r',%t, %e, %x)
</insertmapsql>

<updatemapsql>
UPDATE SYNC_MAPS SET REMOTEID='%r', FLAGS=%x WHERE
LOCALID=%k AND ENTRYTYPE=%e AND TARGETKEY=%t
</updatemapsql>

<deletemapsql>
DELETE FROM SYNC_MAPS WHERE LOCALID=%k AND ENTRYTYPE=%e
AND TARGETKEY=%t
</deletemapsql>
```

12.20.6 <sqlitefile>: SQLite database file name

Contained in: <datastore>
Can contain: path to SQLite 3 database file (usually has extension .sdb)
Attributes: none
Default: none

If this tag contains a file path, the datastore uses the specified SQLite 3 database file to access user data rather than using ODBC to access a SQL server.
Note that SQLite databases can only be used for user data (contacts, calendar, notes etc.), but not for SyncML administrative data like *users, devices, folders, targets* and *maps* (see 12.20.2, 12.20.5, 12.15, 12.16).
In a SQLite based setup, administrative data is either transparently handled by the SyncML engine itself (this is the case for custom clients built with the Synthesis SyncML client library / SDK) or must be stored using a database plugin (see 14 – "<server type="plugin">, <client

type="plugin">: Plugin Based Server or Client Config"). The built-in text-file based plugin is usually sufficient and can be used as-is (see 14.3).

12.20.7 <sqlitebusytimeout>: SQLite database file name

Contained in: <datastore>
Can contain: SQLite timeout when data is busy (in seconds)
Attributes: none
Default: 15

This specifies the time the SQLite engine waits for data becoming ready for access before it returns a "database busy" error.

12.20.8 <quotingmode>: how ODBC strings must be escaped for the database

Contained in: <datastore>, <server> or <client>
Can contain: name of quoting mode
Attributes: none
Default: "singlequote"

This defines how line ends within strings are encoded:

- **"singlequote"**: This is the default, and this was the only mode supported before version 2.1.1.5: single quotes must be duplicated (ok for many SQL DBs like Oracle, Interbase, MS-SQL) in string literals
- **"doublequote"**: double quotes must be duplicated in string literals
- **"backslash"**: Backslash is an escape char, and CR,LF,TAB,BS,\," and ' must be backslash-escaped (MySQL mode).
- **"none"**: No quoting, usually not recommended as string containing the single quote string delimiter can not be used then.

Note: when used in context of a <datastore>, this setting only affects the actual accesses to this datastore - so it is possible to have different quoting modes for different datastores. If used in context of <server> or <client>, the setting is used for all accesses that are not related to a particular datastore.

12.20.9 <dbcanfilter>: use filtering in WHERE clause

Contained in: <datastore>
Can contain: boolean value
Attributes: none
Default: false

If this option is set to true, the server tries to convert active filters into an SQL WHERE clause that can be included in SELECT statements using the %AF and %WF placeholders. This is efficient as it prevents the database from fetching data that is not needed for a sync session, however not all filter expressions can be converted to a WHERE clause.

12.20.10 <earlycommit>: commit at end of SyncML message exchange

Contained in: <datastore>
Can contain: boolean value
Attributes: none
Default: true (false in 1.0.5 versions and earlier)

If this option is set to true, the server always commits all updates to the database at the end of every message exchange with the client. This is important for database setups where an unfinished transaction could lock other transactions. By setting this option, no transaction will be kept active for a longer period of time.

12.20.11 <multicursor>: no longer supported in version 3.0

Contained in: <datastore>
Can contain: boolean value
Attributes: none
Default: false

This option is no longer supported in version 3.0.

12.20.12 <commititems>: commit each item update

Contained in: <datastore>
Can contain: boolean value
Attributes: none
Default: false

If this option is set to true, the server separately commits all updates to the database as they occur. This might be needed depending on the database design.

12.20.13 <modtimestamp>: combined date and time for modification timestamp

Contained in: <datastore>
Can contain: boolean value (for <modtimestamp>)
field names (for <moddatefield>, <modtimefield>).
Available for: ODBC only (not SQLite)
Attributes: none
Default: true

This tag specifies if the modification timestamp in the data table consists of a single timestamp value (<modtimestamp> true) or if it consists of a date field and a time field (<modtimestamp> false). This setting is used when reading modification timestamps with <selectidandmodifiedsql>, see 12.20.14.

12.20.14 <selectidandmodifiedsql>: read IDs and timestamps

Contained in: <datastore>
Can contain: SQL statements
Available for: ODBC and SQLite
Attributes: none
Default: none

This SQL statement must return a result set containing *localID* and *modification timestamp* for all records in the database for the current *sync target* as follows (in that order!):

- **local id**: this column must contain the local ID, which can be a string or a numeric value.
- **modification timestamp** or **modification date:** this column must contain the modification timestamp (if <modtimestamp> is true) or the modification date (if <modtimestamp> is false).
- **modification time**: this third column must be returned only if <modtimestamp> is false (modification timestamp consisting of a date field and a time field).

This SQL statement must only return the rows that belong to the current user (therefore possibly including %u in the WHERE clause, see 12.1.1) and in the current folder (therefore possibly including %f in the WHERE clause, see 12.1.2) and passing the currently set filters (therefore including %AF or %WF in the WHERE clause, see 12.1.3).

Example (note that records are selected by folder key with %f, and %AF is included to extend the WHERE clause with AND plus a filter expression in case there are filters defined for the current sync session (see 7 for details about filters)

```
<selectidandmodifiedsql>
SELECT CONTACTS_KEY,MODIFIED FROM SYNC_CONTACTS WHERE
FOLDERKEY=%f %AF
</selectidandmodifiedsql>
```

12.20.15 <selectdatasql>: read record from database

Contained in: <datastore>
Can contain: SQL statements
Available for: ODBC and SQLite
Attributes: none
Default: none

This SQL statement is used to read all fields of a record identified by its *localID* from the database. The statement must return a result set with a single row containing all columns (fields) in the <fieldmap> that are enabled for read (see "mode" attribute of <map> in 11.34.41.1) in the order as they appear in the <fieldmap>. Normally, this consists of a SELECT statement which uses %N as the list of columns to be selected (%N automatically contains the list of all read-enabled fields in the <fieldlist>, see 12.1.3).

Example:

```
<selectdatasql>
SELECT %N FROM SYNC_CONTACTS WHERE CONTACTS_KEY=%k
</selectdatasql>
```

12.20.16 <insertdatasql>, <updatedatasql>, <deletedatasql>, <zapdatasql>: write records to database

Contained in: <datastore>
Can contain: SQL statements
Available for: ODBC and SQLite
Attributes: none
Default: none

These four SQL statements are used to modify the data table.

1. The statement specified in **<insertdatasql>** must insert a new record into the data table. If <obtainidafterinsert> (see 12.20.18) is false, the ID for the new record will be made available *before* the execution of <insertdatasql> (by executing <obtainlocalidsql>) and can be included in <insertdatasql> using the %k placeholder (see 12.1.3). If <obtainidafterinsert> is true, the execution of <insertdatasql> must automatically generate an ID, which is then obtained with the <obtainlocalidsql> statement *after* the insert. The %N and %v placeholders can be used to easily include the list of column names and values that must be INSERTed. See example below.

2. The statement specified in **<updatedatasql>** must update a record identified by a *localID* (which can be inserted into the statement using %k, see 12.1.3). The %V placeholder can be used to easily include a list of column name / value pairs for an UPDATE statement. %N and %v can be used as well, for example when not using UPDATE, but a stored procedure call.
   Note that %V, %N and %v only include the columns that need to be changed, unless <updateallfields> is set to true. To include all columns that are write-enabled (see "mode" attribute of <map> in 11.34.41.1) the placeholders %aV, %aN and %av can be used.

3. The statement specified in **<deletedatasql>** must delete a record identified by a *localID* (which can be inserted into the statement using %k, see 12.1.3). Note that this needs not necessarily be a physical DELETE statement, but could also be an UPDATE statement that updates a flag in the records such that it disappears from the list of records returned by the <selectidandmodifiedsql> (see 12.20.14).

4. The optional statement specified in **<zapdatasql>** must delete all records that are part of the synchronized data set (that is, all records that are returned by the <selectidandmodifiedsql> statement, see 12.20.14). This statement is used in a server when a SyncML client requests "refresh from client only" sync or in a client when "refresh from server" sync mode is used. If it is not specified, the server will repeatedly use <deletedatasql> to delete all the records one by one.

12.20.17 <ignoreaffectedcount>: Ignore SQLRowCount

Contained in: <datatype>
Can contain: boolean value
Attributes: none
Default: false
New in: 3.0.2.2

If < ignoreaffectedcount> is set, the SQLRowCount (number of rows affected by an UPDATE or INSERT statement) is ignored for <insertdatasql> and <updatedatasql> (see 12.20.16). This might be needed when update statements are implemented as stored procedure which might not set SQLRowCount correctly.

12.20.18 <obtainidafterinsert>, <obtainlocalidsql>, <determineidonce>, <minnextid>, <specialidmode>, <insertreturnsid>, <localidscript>: local object ID management

Contained in: <datastore>
Can contain: boolean value (for <determineidonce>, <obtainidafterinsert>),
SQL query (for <obtainlocalidsql>),
integer value (for <minnextid>),
one of "none" or "unixmsrnd6" (for <specialidmode>),
script for <localidscript> (in PRO version only)
Attributes: none
Defaults: <obtainidafterinsert>: false
<obtainlocalidsql>: empty
<determineidonce>: false
<insertreturnsid>: false
<minnextid>: 1000000
<specialidmode>: none
<localidscript>: none

These seven tags specify how Synthesis Sync Server obtains a new local ID (unique key into the data table) when inserting new data records.

Basically, depending on the database used, the next ID for an INSERT statement must be either obtained before doing the insert (e.g. in Interbase/Firebird or Oracle by using a generator, or by using a random generator) or the database generates a key automatically when the INSERT occurs, and this key must be obtained afterwards (e.g. with MS SQL server's identity columns or MySQL auto_increment).

Unfortunately, some desktop databases like Filemaker Pro do not provide proper support for obtaining the ID of a new records, or are terribly slow in doing so. For these, the very special <determineidonce> and <minnextid> options can provide a solution that is usable, but not fully multi-user proof (see example).

The options are used as follows:

- **<obtainidafterinsert>** must be set to "no" when the key value must be obtained before doing an INSERT statement (e.g. Interbase/Firebird or Oracle case). If so, the <insertdatasql> statement (see 12.20.16) must contain a statement (usually a INSERT) that sets a value for the key field (normally using the %k placeholder, see 12.1.3).
  <obtainidafterinsert> must be set to "yes" if the key value is generated automatically by the database at INSERT and can be obtained afterwards (SQlite, MS SQL or MySQL case). If so, the <insertdatasql> statement should not write to the key field but rely on the database to fill it appropriately.

- **<insertreturnsid>** can be set to "yes" if the <insertdatasql> statement (see 12.20.16) returns the new ID in a result set (single row, first column). This can be useful if insert is implemented using a stored procedure returning the new local ID. If this is set to true, the <obtainlocalidsql> is not executed.
- **<obtainlocalidsql>** must be an SQL query that returns a single row with a single column containing the local ID (data table key value). Depending on the setting of <obtainidafterinsert>, this SQL query is executed before or after doing an INSERT. Note that for SQLite, this is not available – the ROWID of the last insert is always obtained using sqlite3_last_insert_rowid() SQLite API function.
- **<determineidonce>** should never be set in multi-user environments. If set, the sync server executes the <obtainlocalidsql> only once at the beginning of the sync session, expecting a numeric starting value for keys. Keys for new records are then generated by incrementing this number. This can be useful for accessing desktop databases which do not have a proper mechanism for generating unique keys. The <determineidonce> SQL could be something like "SELECT MAX(ID) FROM DATA_TABLE". Of course, this will not work when the database is accessed by more than one application simultaneously.
- **<minnextid>** is only relevant when <determineidonce> is set. It can be used to specify a minimum integer value to be used for the next id. If the result returned by executing the <obtainlocalidsql> statement is numerically less than <minnextid>, the value from <minnextid> will be used as starting value for generating IDs. This is useful for desktop databases that have some auto-increment feature for generating IDs, but no way to properly obtain the ID via ODBC (Filemaker Pro 5.0 for example). With specifying a high <minnextid>, one can guarantee in a single user environment that IDs generated by the internal autoincrement feature and those generated by the sync engine do not conflict. Inserts done by the desktop database itself will have autoincrement IDs (say starting at 1), and because of a high <minnextid> (say 1000000), inserts done by the sync server will get IDs starting at 1000000. With "SELECT MAX(ID) FROM DATA_TABLE" as <obtainlocalidsql>, the sync server is able to properly continue numbering new items, while the database itself uses some internal counter for the autoincrement field which is way below 1000000.
- **<specialidmode>** can be set to "unixmsrnd6" to have pseudo-random IDs generated as follows: UNIX-style time() is multiplied by 1000000 and then a 6-digit random number is appended. This gives a very high probability that all IDs generated this way are unique. If <specialidmode> is not "none", the <obtainlocalidsql> is *not* used for obtaining an ID.
- **<localidscript>** can be used to implement a completely custom method of obtaining a new ID for a new record. This script is executed before <obtainlocalidsql> is executed. If it has a return value, it is used as the local ID for the record to be added, unless <obtainlocalidsql> is specified as well (in this case, the result of <obtainlocalidsql> will override the return value of the <localidscript> - the script might still be useful in case some preparation is required before executing <obtainlocalidsql>). Usually however, *either* <localidscript> *or* <obtainlocalidsql> will be used.

Example for MS SQL Server (ID must be obtained after insert):

```
<obtainidafterinsert>yes</obtainidafterinsert>
<obtainlocalidsql>SELECT @@IDENTITY AS
ID</obtainlocalidsql>
```

Example for Interbase / Firebird (ID must be generated before insert by calling a stored procedure):

```
<obtainidafterinsert>no</obtainidafterinsert>
<obtainlocalidsql>SELECT * FROM
NEXT_CONTACT_KEY</obtainlocalidsql>
```

Example for Filemaker Pro (non-multiuser, lacks proper ID mechanisms)

```
<determineidonce>yes</determineidonce>
<obtainlocalidsql>SELECT MAX(CONTACT_KEY)+1 FROM
SYNC_CONTACTS</obtainlocalidsql>
<minnextid>1000000</minnextid>
```

12.20.19 <map>: SQL specific field mapping features

Contained in: <fieldmap>, <array>
Can contain: nothing
Attributes: SQL specific: readblobsql, keyfield
plus standard attributes: name, references, type, mode, size, truncate (see 11.34.41.1)

This tag establishes a link between an internal field as defined in a <fieldlist>'s <field> tag (see 10.2) and a field in the datastore's user data table (SQL database table field or plugin data field).

The <map> tag in SQL datastores can have the following extra attributes (in addition to the standard <map> attributes explained in 11.34.41.1):

- **"readblobsql"**: This allows specifying a SQL statement for fetching only the mapped field from the database. It must contain a SQL statement (usually a SELECT) that returns exactly one row with exactly one column. This mechanism is intended to read large text and BLOB fields from the database only when needed. Unlike most small strings and other fields that are always required for slow sync comparison etc., large BLOBs and text field's contents are often only needed when actually transmitting them to the remote party. So specifying "readblobsql" prevents unneeded reading of large data chunks and thus increases memory efficiency and performance. **Note that the "readblobsql" will usually not be executed when reading other fields, but probably much later in the process of the sync session.**
- **"keyfield"**: this must be specified if the "readblobsql" needs a special key value to be retrieved independently from the main record. If "keyfield" is specified, the %N list of fields when fetching the main record will contain the name of the keyfield (instead of the name of the content field itself). The value returned from the keyfield can then be used in the "readblobsql" statement using the %K placeholder. Note that for BLOBs that are contained in the main record, "keyfield" can usually be left empty, as the main record key %k can be used to address them. However if the BLOBs are in a detail table (see <array> 12.20.20), the "keyfield" mechanism is important to obtain the correct key for individually reading the BLOBs of an array.

Example 1: the PHOTO field of a vCard (key is the main record key, so no "keyfield" is required:

```
<map name="PHOTO" references="PHOTO" type="BLOB"
 mode="prw"
 readblobsql="SELECT PHOTO FROM SYNC_CONTACTS WHERE CONTACTS_KEY=%k"/>
```

Example 2: the CONTENTS for email Attachments, which are in a detail table (<array> map, see 12.20.20). Here a "keyfield" is required, as the key for fetching the contents is NOT the main record key, but the key in the secondary table (here: ATTS_KEY):

```
<array sizefrom="ATT_CONTENTS">
 … other maps …
  <map name="CONTENTS" references="ATT_CONTENTS" type="blob" mode="prw"
   keyfield="ATTS_KEY"
   readblobsql="SELECT CONTENTS FROM SYNC_EMAIL_ATTS WHERE ATTS_KEY=%K"/>
</array>
```

12.20.20 <array>: definition of master - detail record structures

Contained in: <fieldmap>
Available: in PRO versions only
Can contain: <maxrepeat>, <repeatinc>, <storeempty>, <selectarraysql>, <deletearraysql>, <insertelementsql>, <noitemsfilter>, <initscript>, <beforewritescript>, <afterreadscript>, <finishscript>, <alwaysclean>
Attributes: sizefrom

The <array> tag has the following optional attribute:

- **"sizefrom"**: this can be used to specify either an array field (usually one of the detail fields <map>ed in the <array>) which is used to determine the size of the array when writing detail records. Alternatively, this can specify an integer field, which is then used as array size - it will receive the number of detail records read after a read operation and will be used to determine how many detail records must be written.

An array is a powerful means to map an internal item (consisting of the fields defined in a <fieldlist>, see 10.1) not only to a single database record, but to a *master record* having one or several *detail records*. For example, a database for contact information could store the contact's name in the *master record*, but have an unlimited number of attached *detail records* for each phone number relevant to that contact. Another example are calendar databases that store alarm information in a table separate from the main event table.

An array can contain several tags that control how and if detail records are accessed (see 12.20.21 and 12.20.22) and also contain tags for SQL statement to actually access the detail table 12.20.23).

An array must also contain <map> tags that map internal fields (and local variables of the *ODBC database context*) to the columns of the detail table. As an array might have multiple elements, <map> must either map array columns to array fields or it must map to the first <field> in a block of fields with same type (such that the second and further array elements can be stored in the second and further <field>s in that block).

The following step-by step description shows how <array> works for reading and writing.

When an item is read from the database, the following happens:

1. The master record is read using the <selectdatasql> statement (see 12.20.15)
2. If defined, the <initscript> of the <array> is executed. If this script returns false, processing continues at 13.
3. The <selectarraysql> statement (see 12.20.23) is executed to fetch all detail records related to the current master record.
4. An internal array index is reset to zero.
5. If the result set contains no more rows, processing continues at 11.
6. A row from the result set returned by the <selectarraysql> statement is stored in the fields as defined with the <map>s of the <array>. If a mapped field is an array field, the internal array index is used as array index for the field. If the mappe field is a non-array field, the internal array index is used as an offset which will be added to the position of the mapped field in the <fieldlist> (therefore, second and further elements of the array will be stored in the <field>s following the mapped <field> in the <fieldlist>.
   Note that if a <map> within an <array> references a local script variable rather than a field

from the <fieldlist>, the array index is irrelevant if the local variable referenced is not an array variable.
7. If defined, the <afterreadscript> of the <array> is executed
8. The internal array index is incremented by the value defined with <repeatinc> (default = 1).
9. If the number of array elements processed is less than the number defined with <maxrepeat> (and <maxrepeat> is not 0, meaning unlimited), processing continues at 5.
10. If the "sizefrom" attribute was set to a non-array field, the number of array elements read will be stored in it.
11. If the <selectarraysql> has returned no rows, the <noitemsfilter> (see 12.20.22) is applied to the data item in *make-pass mode* (see 7.1 for details about filters).
12. If defined, the <finishscript> of the <array> is executed
13. If defined, the <afterreadscript> of the <fieldmap> is executed
14. Now, reading the item is complete.

When an item is written to the database, the following happens:
1. If defined, the <beforewritescript> of the <fieldmap> is executed
2. The master record is written to the database using the <insertdatasql> or <updatedatasql> statements (see 12.20.16).
3. If defined, the <initscript> of the <array> is executed. If this script returns false, processing continues at 14. **Note that before Version 2.1.1.24**, the <initscript> was executed only after issuing the <deletearraysql> (see step 4), so returning false from the <initscript> for an update always caused deleting all existing detail records. **From Version 2.1.1.24 onwards, returning false from the <initscript> causes not touching the details records at all.**
4. If the write operation is an update, the <deletearraysql> statement of the <array> is executed (If <alwaysclean> is set to true, the <deletearraysql> is also executed for inserts, see 12.20.24). Note that updating the detail records always includes deleting all existing detail records and then creating new ones with updated detail data.
5. If the <noitemsfilter> is defined, it is applied in *test mode* to the data item. If the item passes the filter, this means that no detail records are needed and therefore processing continues at 14.
6. An internal array index is reset to zero.
7. If defined, the <beforewritescript> of the <array> is executed. If this script returns false, processing continues at 13.
8. If all fields that are mapped with <map> for write in the <array> are empty and the number of detail records is neither defined by a "sizefrom" field in the <array> nor with a <maxrepeat>, processing continues at 11.
9. If all fields that are mapped with <map> for write in the <array> are empty and <storeempty> is false, processing continues at 11.
10. The <insertelementsql> statement (see 12.20.23) is executed to insert one array element. The insert statement must make sure that the detail records are somehow related to the master record, for example by linking them with the master record's key (%k placeholder).
    The placeholders %v and %V will contain values as defined with the <map>s of the <array>. If a mapped field is an array field, the internal array index is used as array index for the field. If the mapped field is a non-array field, the internal array index is used as an offset which will be added to the position of the mapped field in the <fieldlist> (therefore, second and further elements of the array will be stored in the <field>s following the mapped <field> in the <fieldlist>.
    Note that if a <map> within an <array> references a local script variable rather than a field from the <fieldlist>, the array index is irrelevant if the local variable referenced is not an array variable. **As a consequence, mapping a local variable in an <array> means that the array has no "natural" size (as any array index will at least get a value from the local**

variable), so it is essential to use <maxrepeat> or the "sizefrom" attribute in <array> to limit the number of array elements written to the database!

11. The internal array index is incremented by the value defined with <repeatinc> (default = 1).
12. If the number of array elements processed is less than the number defined by the "sizefrom" field and less than specified with <maxrepeat>, processing continues at 7.
13. If defined, the <finishscript> of the <array> is executed
14. Now, writing the item is complete.

12.20.21 <maxrepeat>, <repeatinc>, <storeempty>: detail record storage options

Contained in: <array>
Available: in PRO versions only
Can contain: number (for maxrepeat and repeatinc), boolean value (for storeempty)
Attributes: none
Default: maxrepeat=1, repeatinc=1, storeempty=false

These tags control how <array> stores data in detail tables:

- **<maxrepeat>** defines the maximum number of detail records for the <array>. If this is set to 0, this means that the number is not fixed, but rather depending on how many non-empty detail records can be written or on the "sizefrom" attribute in <array>. We recommend specifying the "sizefrom" attribute (see 12.20.20) when using <maxrepeat>=0.
- **<repeatinc>** defines the increment for the array index (default is 1). This makes sense if the <map>ed fields are not array fields and therefore the array index is used as a offset in the field list.
- **<storeempty>** can be set to true to have even <array> elements with all <map>ed fields empty stored as array element in the detail table. If <storeempty> is set to false, all empty elements will not be stored in the database.

12.20.22 <noitemsfilter>: detail record storage filter

Contained in: <array>
Available: in PRO versions only
Can contain: filter expression (see 7)
Attributes: none
Default: none

This filter is applied in *test mode* to check if an <array> contains any elements. If the data item passes this filter, the SyncML engine assumes the item has no array elements. See <array> in 12.20.20 for details.

12.20.23 <selectarraysql>, <deletearraysql>, <insertelementsql>: detail record SQL

Contained in: <array>
Available: in PRO versions only
Can contain: SQL statements
Attributes: none
Default: none

These statements are used to read, delete or insert array elements. See <array> in 12.20.20 for details.

12.20.24 <alwaysclean>: clean detail records on insert

Contained in: <array>
Available: in PRO versions only
Can contain: boolean value
Attributes: none
Default: false

If this is set to true, the <deletearraysql> will be executed even after inserting a new master record. Usually, when a master record is inserted, no child records related to it already exist and therefore executing <deletearraysql> in these cases is not necessary (but was always performed before version 3.1.6.12).

12.20.25 <optionfilterscript>: prepare SQL filter according to options

Contained in: <fieldmap>
Available: in PRO versions only
Can contain: script
Script context: database context
Attributes: none
Default: no script

This script is called once before any reading or writing takes place. It should check the user options that might restrict the data to be fetched from the database, such as STARTDATE(), ENDDATE(), NOATTACHMENTS() and similar. If it is possible to generate an SQL expression that can be used in a WHERE clause (see %AF and %WF placeholders in 12.1.3), the script should use SETSQLFILTER() to specify the expression and return TRUE. If it is not possible to create an appropriate filter expression, the script must return FALSE or nothing.

The <optionfilterscript> has access to the *database context* specific functions described in 11.34.41.3.

13. <server type="textdb">, <client type="textdb">: Text File Based Server or Client

Contained in: <sysync_config>
Available: Is no longer available in version V3.0, replaced by „textdb“ plugin, see 14. <datafilepath> and <mapfilepath> can be defined as plugin params of the plugin module

14. <server type="plugin">, <client type="plugin">: Plugin Based Server or Client Config

Contained in: <sysync_config>
Can contain: tags described in this chapter (see also 11, and 16 specifically for clients)
Attributes: type="plugin"

This chapter describes the tags available for the configuration of a plugin based server or client. Plugins are additional modules, which will be accessed thru a standardized interface.
The details of this interface are described in the **SDK_manual.pdf** (Reference Manual for Software Development Kit and Plugin Interface of the Synthesis Sync Engine).

One member of the plugin family is the „textdb“, which is the replacement for the implementation in the 2.1.X servers/clients.

Plugins are either
- built-in (e.g.: „**textdb**“; replacement for the former server type „textdb“)
- external modules (DLLs / shared libaries; e.g.: samples of the **SDK** package)

14.1 plugin module: global settings

Contained in: <server type="plugin">, <client type="plugin">
Can contain: <plugin_module>
<plugin_sessionauth>
<plugin_deviceadmin>
<plugin_params>
Default: empty built-in module called „no_dbapi“ w/o functionality

14.1.1 <plugin_module>

The plugin name can either refer to an external module, which will be linked dynamically (example: `<plugin_module>SDK_textdb</plugin_module>`), or as built-in module using the bracket notation (example: `<plugin_module>[SDK_textdb]</plugin_module>`).

While external plugin modules can be added individually by the customer, the built-in modules are integrated by Synthesis AG only.

Built-in modules are:
- the empty default adapter „**no_dbapi**“
- the text db plugin „**SDK_textdb**“ , available in demo/STD/PRO server/desktop client, see also 14.3
- the Java/JNI interface „**JNI**“ (PRO server/desktop client only).

14.1.2 <plugin_sessionauth>

By default, the session authenitification (login / logout) will be done by the underlying layer (ODBC for the STD and PRO server, see 12.16). By setting <plugin_sessionauth> to yes, the authentification of the plugin module will be activated:

```
<plugin_sessionauth>yes</plugin_sessionauth>
```

14.1.3 <plugin_deviceadmin>

By default, the device administration (nonce handling, getting DB time, saving device info) will be done by the underlying layer (ODBC for the STD and PRO server). By setting the tag <plugin_deviceadmin> to yes, the device administration of the plugin module will be activated:

```
<plugin_deviceadmin>yes</plugin_deviceadmin>
```

14.1.4 <plugin_params>

All **plugin specific** parameters must be passed as within <plugin_params>. They will be passed as a string to the plugin module. See description of the used plugin module for the details of these parameters.

Example:

```
<plugin_params>
    <datafilepath>/var/log/sysync</datafilepath>
 </plugin_params>
```

14.2 <datastore type="plugin">: Plugin Datastore specific settings

Contained in: <server type="plugin">
Attributes: name, type
Can contain: <plugin_module>
<plugin_datastoreadmin>
<plugin_params>
<plugin_moduleadmin>
<plugin_paramsadmin>
Default: not specified

This chapter describes the tags that are specific to plugin based datastores. See 11.34 for a description of the <datastore> tag in general.

14.2.1 <plugin_datastoreadmin>

By default, the datastore administration (ADM / MAP tables) will be done by the underlying layer (ODBC for the STD and PRO server).

By setting the tag <plugin_datastoreadmin> to yes, the datastore administration of the plugin module will be activated: <plugin_ datastoreadmin>yes</plugin_ datastoreadmin>
NOTE: If this flag is set, the params of 14.2.5 will not be considered.

14.2.2 <plugin_module>

For details see 14.1.1

14.2.3 <plugin_params>

All **plugin specific** parameters for this datastore must be passed within <plugin_params>.
For details see 14.1.4.

14.2.4 <plugin_debugflags>

16 plugin specific flags can be defined for plugin debug logging.
Two of these bits (bit 0 and bit 1) are predefined and reserved: bit 0 is used for all plugin interface logging (which is part of the SyncML engine), bit 1 is an unspecific DB flag, which will be used in all the plugin examples. The flags must be defined in hexadecimal representation.

Example: <plugin_debugflags>0x0501</plugin_debugflags>
➔ Bits 0, 8 and 10 are set

If <plugin_debugflags> is not defined, ALL plugin specific flags are set (0xffff).

NOTE: To activate any plugin logging, the global <debug> flag "plugin" must be set. "plugin" will be activated also together with some combined groups. For details see the <debug> description (8.11):

```
<debug>
  ...
   <enable option="plugin"/>
  ...
</debug>
```

14.2.5 <plugin_module_admin>,<plugin_params_admin>, <plugin_debugflags_admin

It is possible to have a different plugin module for datastore data handling and administration, when defining <plugin_module_admin>. For this case <plugin_params_admin> and <plugin_debugflags_admin> will be used.

If only <plugin_module_admin> is defined, but not <plugin_module>, the datastore data will be handled by the underlying layer (which is ODBC for the STD and PRO server).
An overview is given in the table below:

<plugin_ module>	<plugin_ module_admin>	<plugin_ datastoreadmin>	DATA	ADMIN

-	-	false	ODBC	ODBC
aa	-	false	aa	ODBC
-	bb	false	ODBC	bb
aa	bb	false	aa	bb
-	-	true	ODBC	ODBC
aa	-	true	aa	aa
-	bb	true	ODBC	ODBC
aa	bb	true	aa	aa

14.3 plugin module "SDK_textdb"

The built-in plugin module „SDK_textdb" acts as the db interface for the „demo" server/desktop client, but is also available in the STD and PRO servers/desktop clients

14.3.1 Files of the textdb

The „textdb" creates 5 different files:

- **TDB**_uuu_datastorename.txt — uuu = userKey
- **BLB**_uuu_datastorename_itemid_fieldid
- **DEV**_ddd.txt — ddd = deviceKey
- **ADM**_ddd_uuu_datastorename.txt
- **MAP**_ddd_uuu_datastorename.txt

The most important one is the **TDB_*.txt** file, which contains the data for each datastore. The file format has the same structure as the former textdb data file:

- The first column is the GUID (object identifier on the server), which is an integer number in the SDK_textdb implementation (starting at 10000).
- The second column is the modification timestamp in ISO8601 format, UTC (you would need to update this in order to mark a record modified).
- All following columns contain the data fields in the order as defined in the corresponding <fieldlist> (see 10.1).

uuu is the userKey, resulting from Login(username,password). It must be unique for each user. The **ddd** part of the name is derived from the client's DeviceID (which should be, in theory, unique). Login(„test","test") is hardwired in this textdb and results in uuu="test".

The **DEV_*.txt** file holds the device information, **ADM_*.txt** / **MAP_*.txt** contain the administration data for each device and user and datastore (if datastore admin is switched on). Normally no reason exists to edit these files.

14.3.2 PluginParams of the textdb

Plugin specific params are defined for the (see 14.2.3):

<datafilepath>: The data files TDB_* can be chosen specifically. If this tag is not defined, data files will be stored at the current path of the SyncML engine. This plugin param replaces the former <datafilepath> of the old textdb.

<blobfilepath>: The BLOB files BLB_* can be chosen specifically. If this tag is not defined, BLOB files will be stored at <datafilepath>. This plugin param was not available for the old textdb (because BLOBs were not supported there).

<mapfilepath>: The map/admin files MAP_*, ADM_* and DEV_* can be chosen specifically. If this tag is not defined, data files will be stored at <datafilepath>. This plugin param replaces the former <mapfilepath> of the old textdb.

Some legacy params, do not use them any more for new applications:
<unixpath> : path for TDB/BLB/MAP/ADM/DEV files (Linux, MacOSX)
<winpath> : path for TDB/BLB/MAP/ADM/DEV files (Windows)

If platform specific data/map file paths are needed, use the platform attribute (see 4):
Example: <mapfilepath platform="win32">D:\projects\maps</mapfilepath>

14.4 plugin module "FILEOBJ"

14.4.1 Files of the fileobj modules

The FILEOBJ plugin is able to handle OMA DS 1.2 file objects. The items will be stored as real files, when possible with their correct name, file date and attributes. So with this plugin module, a file sync via SyncML is possible.

NOTE: Using OMA DS 1.2 FILEOBJ do not require necessarily a FILEOBJ plugin, the objects can alternatively be stored as BLOBs within an ODBC data structure. Even the textdb implementation is able to store these file objects.

15. <client>, <server>: Synthesis SyncML Engine library only configuration tags

Contained in: <sysync_config> of a Synthesis SyncML Engine library based client or server.
Can contain: tags described in this chapter (see also 11, 12 and 14 for settings not specific to SyncML engine library based clients)
Attributes: type (see 11 for details)

15.1 <binfilespath>: Path for persistent storage of client settings and admin data

Contained in: <client>
Can contain: path of a directory where client engine library can read and write
Default: platform's default location for storing application data

The Synthesis SyncML Client Engine Library usually comes with built-in management for persistent, but user changeable settings, organized in so-called "profiles" and "targets" (for details about the Synthesis SyncML Client Engine Library, please refer to the SDK documentation (SDK_manual.pdf). This setting information, along with some internal SyncML administration data like changelogs is stored in binary files.
The <binfilespath> tag is used to specify a directory where the client engine library can store these files.
If <binfilespath> is not explicitly set, the platform's default path for storing an application's data or preferences is used (the same path as represented by the $(appdata_path) config variable, see 4.4)

15.2 <binfilesactive>: enable binfile based admin

Contained in: <client>
Can contain: boolean value
Default: true for clients (false for server)

This can be set to false to disable the binfile admin data handling, and use admin storage implemented as SQL tables or in a plugin (like textdb). This can make sense when using the engine as client and server for the same database. Disabling the binfiles layer makes the engine behave like the command line client products (where remote server URL, login etc. is all defined in the XML config (see 16).

15.3 <crcchangedetection>: enable CRC based change detection

Contained in: <client>
Can contain: boolean value
Default: false

This can be set to true to make the SyncML engine detect changes by calculating CRC checksums of all records. If this option is enabled, the underlying database (SQL or plugin) does not need to have a reliable last-change date. The engine will always query all records from the database before each sync to calculate CRCs, which then are compared with stored CRCs to detect changes.

16. <client>: Command line client-only configuration tags

Contained in: <sysync_config> of a command line Synthesis SyncML client or Synthesis SyncML engine library with <binfilesactive> set to false (see 15.2).
Can contain: tags described in this chapter (see also 11, 12 and 14 for settings not specific to command line clients)
Attributes: type (see 11 for details)

16.1 <defaultsyncmlversion>: Set default SyncML Version to start a session

Contained in: <client>
Can contain: SyncML version string, currently "1.0", "1.1" or "1.2"
Default: Higest version supported by client.

This specifies the SyncML version that is used in the first attempt to contact a SyncML server. By default, this is the highest version supported by the client (1.2 at this time). When session initialisation fails with a SyncML version, the client automatically re-tries with the next lower SyncML version. Only if you e.g. want to prevent attempts for SyncML 1.2 or 1.1 completely, this can be set to 1.1 or 1.0, resp.

16.2 <defaultauth>: Set default auth method

Contained in: <client>
Can contain: SyncML auth method: "none", "basic", "md5"
Default: "none".

This specifies the SyncML auth method that is used in the first attempt to login with a SyncML server. When login fails, the server will return a challenge for the required auth method, and the client will login according to that challenge. Another setting than "none" should be used only for special testing situations. In particular, using "basic" comprises a security risk as this means the client will send credentials in decodable form.

16.3 <defaultauthencoding>: Set default auth encoding

Contained in: <client>
Can contain: SyncML encoding: "chr", "bin", "b64"
Default: "chr".

This specifies the encoding for the SyncML auth credentials that is used in the first attempt to login with a SyncML server. When login fails, the server will return a challenge for the required auth encoding, and the client will login according to that challenge. This setting should be changed only for special testing situations.

16.4 <defaultauthnonce>: Set default nonce

Contained in: <client>
Can contain: nonce to use for first MD5 login attempt
Default: "chr".

This is relevant only if <defaultauth> is set to "md5". Then the specified string will be used as nonce for the MD5 login attempt. In normal use, the nonce is always sent by the server in the challenge, and is not known in advance. Therefor, this setting is useful only for special testing situations.

16.5 <newsessionforretry>: Use a new sessionID for retries

Contained in: <client>
Can contain: boolean value
Default: true

If this is set, retries (due to auth failure) are done with a completely new SyncML session (different session ID). If this is set to no, the retry is done with the same session ID as the initial attempt (so it looks like continuing the session). Only some special testing tools may require this.

16.6 <originaluriforretry>: Use original URI for retry

Contained in: <client>
Can contain: boolean value
Default: true

If this is set, retries (due to auth failure) are done to the original URI as specified in the <serverurl> (see 16.12). If this is set to no, the retry is done with the URI returned by the server in the <RespURI> element. Only some special testing tools may require this.

16.7 <smartauthretry>: Use smart retry attempt variations

Contained in: <client>
Can contain: boolean value
Default: true

If this is set, after doing the regular retries in the way configured with <newsessionforretry> (see 16.5) and <originaluriforretry> (see 16.6), the client also does some more retry attempts with varied behaviour. In particular, <newsessionforretry> is inverted for the additional retries, and <originaluriforretry> will be set to the same (inverted) value of <newsessionforretry>. Future versions might add more elaborate retry attempts when <smartauthretry> is set. Note that this helps to get authorized with servers that have problem with either type of retry (in-session or fresh-session).

16.8 <putdevinfatslowsync>: Always send Device Info at Slowsync

Contained in: <client>
Can contain: boolean value
Default: true

If set to true, the client will always send it's Device Info (DevInf) to the server when a slow sync is performed. If set to false, the client will send the DevInf only in the first sync session with a server or when explicitly requested by the server.

16.9 <localdbuser>, <localdbpassword>: Login to local database

Contained in: <client>
Can contain: user, password
Default: none

This information is used to login the client locally with the database. This is needed because the SyncML client can connect to a multi-user-database, but will usually synchronize only one user's data in a given sync session. These tags are required to specify which user's data the sync session operates on.

16.10 <nolocaldblogin>: Prevent local DB login

Contained in: <client>
Can contain: boolean value
Default: false

If set to false, the client will not perform the login steps to the local database. This can be useful when the underlying database is a single-user database. Note that the sample database layouts delivered with the Synthesis SyncML client are designed for multiple users and therefore need local login.

16.11 <syncmlencoding>: SyncML encoding format

Contained in: <client>
Can contain: "xml" or "wbxml"
Default: "xml"

This can be used to select either the plain text (but bandwidth wasting) XML format or the much more efficient, but not human-readable binary WBXML format. For any application but server debugging, this should be set to WBXML.

16.12 <serverurl>: Remote SyncML server URL

Contained in: <client>
Can contain: URL
Default: none

This is the remote SyncML server's URL for use with command line clients which specify the server details in the configuration. For clients based on Synthesis SyncML client engine, the URL and other settings are specified in profile settings, not the config file. Still, this tag can be used to "hard-code" the server URL so the client can only be used for a specific server.

All clients support http URLs, some clients also support the following special URLS:

- SSL-enabled clients also support URLs beginning with "https://" instead of "http://"
- OBEX-over-infrared enabled clients support the special "obex:irda" URL.
- OBEX-over-TCP enabled clients support the special "obex:xxxx" URL, where xxx is the name or IP address of the OBEX server.

16.13 <serveruser>, <serverpassword>: Login to remote SyncML server

Contained in: <client>
Can contain: user, password
Default: none
Available in: command line based clients

This information is used to login the client with the remote SyncML server.

16.14 <sockshost>, <proxyhost>: Proxy servers

Contained in: <client>
Can contain: proxy server addresses
Default: none
Available in: command line based clients

These can be used to specify a SOCKS or HTTP proxy server to be used.

16.15 <proxyuser>, <proxypassword>: Proxy auth

Contained in: <client>
Can contain: proxy server user and password (if required)
Default: none
Available in: command line based clients

These can be used to specify a SOCKS or HTTP proxy server to be used.

16.16 <transportuser>, <transportpassword>: Login to remote SyncML server

Contained in: <client>
Can contain: user, password
Default: none
Available in: command line based clients

This information is used for HTTP authentification if the server addressed by <serverurl> needs this.

16.17 <syncrequest>: Request to sync a datastore

Contained in: <client>
Can contain: <localpathextension>, <dbpath>, <syncmode>, <slowsync>
Attributes: datastore
Default: none
Available in: command line based clients

This tag specifies a Sync request for one of the <datastore>s (see 11.34) available to the client. The "datastore" attribute must reference one of the <datastore>s defined further up in the configuration file.
Multiple <syncrequest> tags are allowed to synchronize multiple datastores in a single session (however, each datastore may only be synced once per session).

16.17.1 <dbpath>: path of remote server's datastore

Contained in: <syncrequest>
Can contain: remote server db path string
Default: empty
Available in: command line based clients

This tag specifies the path that is used to address the remote server's datastore that should be synchronized with the local datastore identfied by the "datastore" attribute in the enclosing <syncrequest>. Usually, this is a simple string like "contact", "events", "emails" etc. But some server support structured paths like "contacts/private", "emails/office" or even CGI-style filtering like "contacts?LAST=Miller". If the server happens to be a Synthesis SyncML server, you can refer to 7.4 to see what options are possible.

16.17.2 <syncmode>: Synchronisation mode

Contained in: <syncrequest>
Can contain: "twoway", "fromclient" or "fromserver"
Default: "twoway"
Available in: command line based clients

This can be used to select between three basic sync modes:

- "twoway" means that changes made on the client are sent to the server and changes made to the server are sent to the client.
- "fromclient" means that only changes made to the client are sent to the server. Changes made on the server will be ignored. Sometimes called "update server only" mode.
- "fromserver" means that only changes made to the server are sent to the client. Changes made on the client will be ignored. Sometimes called "update client only" mode.

Note: "fromclient" in conjunction with <slowsync> (see 16.17.3) set to true will cause the server database to be completely overwritten with the contents of the client database (also called "refresh from client" or "reload server"). Likewise, "fromserver" in conjunction with <slowsync> set to true will cause the client database to be completely overwritten with the contents of the server database (also called "refresh from server" or "reload client"/"reload device").

16.17.3 <slowsync>: Force a slow sync

Contained in: <syncrequest>
Can contain: boolean value
Default: automatically determined
Available in: command line based clients

This can be used to explicitly request a slow sync. Slow sync means that all data on the client and the server are exchanged. In a two-way slow sync, the result will be the union of all client and all server data. In a one-way (from server or from client) sync, the result is that either side (client or server) is totally overwritten with the data from the other side. See <syncmode> (16.17.2) for more information.

16.17.4 <localpathextension>: local datastore options

Contained in: <syncrequest>
Can contain: local db path extension
Default: empty
Available in: command line based clients

This tag specifies extra information for addressing the local datastore. Usually, this is empty. But if your local database makes use of multiple folders per user (see description of our standard database layout for SyncML in the client or server installation manual delivered with the software), you can specify the folder here. Likewise, if your local datastores support filtering (see 7.4), you can specify the required CGI to activate the filters here. Note that the same kind of options might be available at the remote server's side, but then they need to be specified as part of <dbpath>.

16.17.5 <recordfilter>: define SyncML DS 1.2 record filter

Contained in: <syncrequest>
Can contain: Filter expression
Default: none
Available in: command line based clients

This tag can be used to specify a SyncML DS 1.2 exclusive filter expression (see 7.4 for syntax and 7 for general information about filters). This is relevant only for sync sessions with SyncML DS 1.2 capable servers.

16.17.6 <recordfilterinclusive>: define inclusive SyncML DS 1.2 record filter

Contained in: <syncrequest>
Can contain: Filter expression
Default: none
Available in: command line based clients

This tag can be used to specify a SyncML DS 1.2 inclusive filter expression (see 7.4 for syntax and 7 for general information about filters). This is relevant only for sync sessions with SyncML DS 1.2 capable servers.

17. List of built-in timezones

Special names

SYSTEM
DATE
FLOATING
USERTIMEZONE

Name	east of UTC
Afghanistan	4.5
AKST/AKDT	-9
HNY/NAY	-9
Alaskan	-9
Arab	3
Arabian	4
Arabic	3
AST/ADT	-4
HNA/HAA	-4
Atlantic	-4
AUS_Central	9.5
AUS_Eastern	10
Azerbaijan	4
Azores	-1
Canada_Central	-6
Cape_Verde	-1
Caucasus	4
ACST/ACDT	9.5
Central_Australia	9.5
Central_America	-6
Central_Asia	6
Central_Brazilian	-4
CET/CEST	1
MEZ/MESZ	1
Central_Europe	1
Central_European	1
Central_Pacific	11
CST/CDT	-6
HNC/HAC	-6
Central	-6
Central_Mexico	-6
China	8
Dateline	-12
East_Africa	3
AEST/AEDT	10
East_Australia	10
EET/EEST	2
East_Europe	2
East_South_America	-3
EST/EDT	-5
HNE/HAE	-5
Eastern	-5
Egypt	2
Ekaterinburg	5
Fiji	12
FLE	2
Georgian	3
GMT	0
Greenland	-3
Greenwich	0
GTB	2
HAST/HADT	-10
Hawaiian	-10
India	5.5
Iran	3.5
Israel	2
Jordan	2
Korea	9
Mexico	-6
Mexico_2	-7
Mid_Atlantic	-2
Middle_East	2
Montevideo	-3
MST/MDT	-7
HNR/HAR	-7
Mountain	-7
Mountain_Mexico	-7
Myanmar	6.5
North_Central_Asia	6
Namibia	2
Nepal	5.75
New_Zealand	12
NST/NDT	-3.5
HNT/HAT	-3.5
Newfoundland	-3.5
North_Asia_East	8
North_Asia	7
Pacific_SA	-4
PST/PDT	-8
HNP/HAP	-8
Pacific	-8
Pacific_Mexico	-8
Romance	1
Russian	3
SA_Eastern	-3
SA_Pacific	-5
SA_Western	-4
Samoa	-11
SE_Asia	7
Singapore	8
South_Africa	2
Sri_Lanka	5.5
Taipei	8
Tasmania	10
Tokyo	9
Tonga	13
US_Eastern	-5
US_Mountain	-7
Vladivostok	10
West_Australia	8
West_Central_Africa	1
WET/WEST	1
West_Europe	1
West_Asia	5
West_Pacific	10
Yakutsk	9
A	1
ACDT	10.5
ACST	9.5
ADT	-3
AEDT	11
AEST	10
AKDT	-8
AKST	-9
AST	-4
AWST	8
B	2
BST	1
C	3
CDT	-5
CEST	2
CET	1
CST	-6
CXT	7
D	4
E	5
EDT	-4
EEST	3
EET	2
EST	-5
F	6
G	7
H	8
HAA	-3
HAC	-5
HADT	-9
HAE	-4
HAP	-7
HAR	-6
HAST	-10
HAT	-1.5
HAY	-8
HNA	-4
HNC	-6
HNE	-5
HNP	-8
HNR	-7
HNT	-2.5
HNY	-9
I	9
IST	1
K	10
L	11
M	12
MDT	-6
MESZ	2
MEZ	1
MST	-7
N	-1
NDT	-1.5
NFT	11.5
NST	-2.5
O	-2
P	-3
PDT	-7
PST	-8
Q	-4
R	-5
S	-6
T	-7
U	-8
UTC	0
V	-9
W	-10
WEST	1
WET	0
WST	8
X	-11
Y	-12
Z	0

18. Error codes

This section lists the error codes that can occur (normally visible in the logs or on the console).

18.1 SyncML Status Codes

These codes are defined by the SyncML standard. For details, see http://www.openmobilealliance.org/release_program/ds_v12.html. Note that this list is not complete, but only contains the codes that are important for the SyncML engine.

101	Server is busy (session limit reached, see 8.2)
200	OK, successful operation
201	Item added
207	Conflict resolved with merge
208	Conflict resolved - client wins
209	Conflict resolved by duplicating item
210	Deleted without archive
211	Item not deleted
212	Authentication accepted for entire session
213	Chunked item accepted and buffered (this status is sent for each non-final part of a data item that has been split across multiple SyncML messages)
400	Bad request
401	Unauthorized (bad credentials)
403	Forbidden (e.g. attempt to write to a read-only database)
404	Object not found
405	Command not allowed
406	Optional feature not supported
407	Authentication required (no credentials found)
408	Timeout
409	Conflict, operation failed
410	Gone, requested object not here any more
412	Incomplete command
415	Unsupported media type or format
418	Object already exists
419	Conflict resolved with server data
420	Device full
500	Command failed
501	Command not implemented
503	Service unavailable
505	DTD version not supported
508	Slow sync required
509	Authentication required
510	Database error
511	Server error
512	Synchronisation failed
513	SyncML Version not supported

514	Cancelled

18.2 Internal Error Codes

0	No error
10000..10999	These have the same meaning as the SyncML Status Codes (see 18.1), but they are offset by 10000 to make clear that they were generated internally, and not sent or received via SyncML.
20001	Bad or unknown transport protocol
20002	Fatal problem with SyncML encoder/decoder
20003	Cannot open communication
20004	Cannot send data
20005	Cannot receive data
20006	Bad content type (message received with an unknown MIME-type)
20007	Error processing incoming SyncML message (for example invalid XML or WBXML formatting)
20008	Cannot close communication
20009	Transport layer authorisation (e.g. HTTP auth) failed
20010	Error parsing XML config file
20011	Error reading config file
20012	No configuration found at all, or not enough for requested operation (client)
20013	Config file could not be found
20014	License expired or no license found
20015	Internal fatal error
20016	Bad handle
20017	Session aborted by user
20018	Invalid license
20019	Limited trial version
20020	Connection timeout
20021	Connection SSL certificate expired
20022	Connection SSL certificate invalid
20023	incomplete sync session (some datastores failed, some completed)
20025	Out of memory
20026	Connection impossible (e.g. no network available)
20027	Establishing connection failed (e.g. network layer login failure)
20028	element is already installed
20029	this build is too new for this license (need upgrading license)
20030	function not implemented
20031	this license code is valid, but not for this product (e.g. STD license used in PRO product, or client license in server product)
20032	Explicitly suspended by user
20033	this build is too old for this SDK/plugin
20034	unknown subsystem
20036	local datastore not ready
20037	session should be restarted from scratch
20038	internal pipe communication problem

20039	buffer too small for requested value
20040	value truncated to fit into field
20041	bad parameter
20042	out of range
20043	external transport failure (no details known in engine)
20044	class not registered
20500..20599	These represent SIG_xxx codes in Linux versions of the SyncML engine. Unexpected SIG_xxx will generate a error code of 20500+signal_code.
20998	Internal unkown exception
20999	Unknown error
21000...21999	Database plugin module specific error codes

19. Index

19.1 Alphabetic Index of all config XML tags

abortonallitemsfailed 110
acceptfilter 121
acceptserveralerted 111
adminreadyscript 127
afterconnectscript 157
afterreadscript 138
alertprepscript 128
alertscript 127
allowempty 89
allowmessageretries 147
alwaysclean 182
alwayssendlocalid 119
appendtoexisting 61
array 179
attachmentcountfield 90
attachmentmimetypesfield 90
attachmentnamesfield 90
attachmentsfield 90
attachmentsizesfield 90
autoenddateinclusive 109, 149
automap 137
autononce 104
beforewritescript 139
bigendian 94
binaryparts 93
binfilesactive 189
binfilespath 189
bodycountfield 90
bodymimetypesfield 90
bufferretryanswer 69
cleartextpw 159
client 101, 189, 191
client type= 151
client type=odbc 151
client type=plugin 171, 184
client type=textdb 183
commititems 173
comparescript 98
completefromclientonly 147
configdate 65
configidstring 54
configvar 55
conflictstrategy 119
constantnonce 104
contains 142
crcchangedetection 189
customendputscript 106
customgethandlerscript 107
customgetputscript 106
customputresulthandlerscript 108
datacharset 131
datalineends 131
datasource 155
datastore 115
datastore type= 164
datastore type=odbc 164
datastore type=plugin 185
datastoreinitscript 123, 126
datatimezone 130
datatype 91
datatypes 72
dbcanfilter 172
dbconnectionstring 156
dbpass 157
dbpath 195
dbtimeout 157
dbtypeid 116
dbuser 156
debug 56
debugchunkmaxsize 109
defaultauth 191
defaultauthencoding 191
defaultauthnonce 192
defaultsyncmlversion 191
definetimezone 66
deletearraysql 182
deletedatasql 175
deletemapsql 169
deletewins 117
deletinggoneok 109
descriptivename 144
determineidonce 176
deviceid 144
devicetype 144
disable 57
dispatchfilter 142
displayname 117
ds12filters 121
dscgiindevinf 146
dspathindevinf 146
earlycommit 173
enable 57
enum 81
enumdefaultpropparams 111
externalurl 103
field 73
fieldlist 73
fieldmap 134
fileprefix 62
filesuffix 62
filterinitscript 95
filterscript 95
finalisationscript 140
finalrule 143
finishscript 141
firmware 143
firsttimestrategy 119
folderkeysql 164
folding 59
forcelocaltime 147
forceutc 148
fromremoteonlysupport 132
function 55
getdevicesql 160
globallogs 65
guidprefix 142
hardware 143
headertag 89
httpport 70
ignoreaffectedcount 175
ignoredevinfmaxsize 146
incomingscript 95
indentstring 63
inheader 89
initscript 94, 137
inputcharset 149

insertdatasql 175
insertelementsql 182
insertmapsql 169
insertreturnsid 176
invisiblefilter 122
ipaddress 70
keepconnection 68
lastmodfieldtype 169
legacymode 149
lenientmode 149
licensecode 53
licensename 53
limitedfieldlengths 144
linemap 88
localdbfilter 122
localdbpassword 193
localdbuser 193
localidscript 176
localpathextension 196
logenabled 112
logfile 111
logflushmode 61
logformat 59, 112
logincheckscript 162
loginfinishscript 114
logininitscript 114
loglabels 114
logpath 57
logsessionstoglobal 65
looptimeout 56
macro 56
makepassfilter 123
makevisiblefilter 123
manufacturer 54, 143
map 134, 178
maxattachments 90
maxconcurrentsessions 53
maxitemspermessage 118
maxmsgsize 53
maxobjsize 54
maxrepeat 181
maxsessionruns 71
maxsyncmlversion 101
maxthreads 71
md5hex 159
md5userpass 159
mergescript 99
mimedirmode 99
mimemail 90
mimeprofile 75
minnextid 176
minsyncmlversion 101
model 54, 143
modtimestamp 173
msgdump 64
multicursor 173
multithread 105
neverputdevinf 66
newdevicesql 160
newsessionforretry 192
nocontentfolding 148
noemptyproperties 145
noitemsfilter 181
nolocaldblogin 193
noreplaceinslowsync 146
numlines 88
obexservice 70
obtainidafterinsert 176
obtainlocalidsql 176
oem 143
optionfilterscript 127, 182
originaluriforretry 192
outgoingscript 95
outputcharset 149
parameter 82
plugin_debugflags 186
plugin_deviceadmin 185
plugin_module 184, 186, 187, 188
plugin_moduleadmin 186
plugin_params 185, 186
plugin_paramsadmin 186
plugin_sessionauth 185
position 84
preventconnectattrs 157
processitemscript 97
profile 76
property 77
protocol 69
proxyhost 194
proxypassword 194
proxyuser 194
putdevinfatslowsync 193
quotingmode 172
readonly 117
receiveditemstatusscript 106, 129
recordfilter 197
recordfilterinclusive 197
rejectstatus 150
remoterule 143
repeatinc 181
reportupdates 118
requestedauth 103
requestmaxtime 102, 150
requestmintime 102
requiredauth 103
resendfailing 129
respurionlywhendifferent 104
resumeitemsupport 133
resumesupport 133
rulescript 150
saveinfosql 160
savenoncesql 160
scripting 55
selectarraysql 182
selectdatasql 174
selectidandmodifiedsql 174
selectmapallsql 169
sendrespuri 104
sentitemstatusscript 106, 128
server 101, 189
server type= 151
server type=odbc 151
server type=plugin 184
server type=textdb 183
serverpassword 194
serverurl 194
serveruser 194
sessioninitscript 105
sessionlogs 64, 65
sessiontimeout 102
showctcapproperties 110
showthreadid 60
showtypesizeinctcap10 110
silentdiscard 122
simpleauthpw 105
simpleauthuser 105
singlegloballog 61
singlesessionlog 61
sizelimitfield 90
slowsync 196
slowsyncstrategy 119
smartauthretry 192
sockshost 194
software 143
specialidmode 176
sqlitebusytimeout 172
sqlitefile 171
storeempty 181
storelastsyncidentifier 133
storesyncidentifiers 133

subprofile 76
subthreadmode 62
superdatastore 141
syncmlencoding 193
syncmode 195
syncrequest 195
synctargetgetsql 165
synctargetnewsql 165
synctargetupdatesql 165
synctimestamp 169
synctimestampatend 132
systemtimezone 66
textauth 158
textmap 158
textpath 158
textprofile 88
timedsessionlognames 60
timestamp 60
timestampall 60
timestampsql 163
timestamputc 130, 163
timeutc 130
transactionmode 158
transport 68
transportpassword 195
transportuser 195
treataslocaltime 148
treatasutc 148
tryupdatedeleted 118
typestring 92
typesupport 120
unfloattimestamps 87
unicodedata 94
updateallfields 132
updateclientinslowsync 145
updatedatasql 175
updatemapsql 169
updateserverinslowsync 145
use 91, 120
userkeysql 161
usertimezone 109
userzoneoutput 130
value 79
valuetype 89
version 92
versionstring 92
vtimezonegenmode 87
waitforstatusofinterrupted 108
xmltranslate 63
zapdatasql 175
zipcompressionlevel 93
zippedbindata 93

Alphabetic Index of all built-in script functions

ABORTDATASTORE 128
ABORTSESSION 43
ABS 43
ADDFILTER 124
ADDSTATICFILTER 124
ADDTARGETFILTER 124
ALERTCODE 125
ALLDAYCOUNT 39
APPEND 43
ARRAYINDEX 138
AUTHDEVICEID 114
AUTHOK 114
AUTHSTRING 114
AUTHTYPE 114
AUTHUSER 114
CHECKAUTH 115
COMPARE 43
COMPAREFIELDS 98
CONFLICTSTRATEGY 97
CONTAINS 43
CONVERTTOUSERZONE 42, 115
CONVERTTOZONE 42
DATEONLY 39
DAYUNITS 39
DBHANDLESOPTS 125
DBLITERAL 154
DBNAME 124
DBNOW 38
DBOPTIONS 124
DEBUGMESSAGE 42
DEBUGSHOWITEM 42
DEBUGSHOWVARS 42
DEFAULTSIZELIMIT 96, 125
DELETEWINS 97
DEVICEKEY 115
DURATION 39
ECHOITEM 97, 107, 108
ENDDATE 124
ENUMDEFAULTPROPPARAMS 44
EXPLODE 37
FIND 37
FIRSTTIMESYNC 125
FORCECONFLICT 97
FORCELOCALTIME 44
FORCEUTC 44
GETCGITARGETFILTER 123
GETDEBUGMASK 42
GETFILTER 124
GETTARGETFILTER 124
IGNOREUPDATE 98
ISAVAILABLE 43
ISDATEONLY 39
ISDURATION 39
ISFLOATING 41
ISRELATIVE 40
ITEMDATATYPE 45
ITEMTYPENAME 45
ITEMTYPEVERS 45
LASTKEY 138
LENGTH 37
LOCALDBNAME 124
LOCALID 97
LOCALIZEDASUTC 40
LOCALURI 44
LOCALZONEOFFSET 40
LOGSUBST 138
LOOSINGCHANGED 99
LOWERCASE 37
MAKE_RRULE 40
MAKEALLDAY 40
MAKEEMAILSPEC 37
mapredefine 134
MAXITEMCOUNT 125
MERGEFIELDS 99
MILLISECONDS 39
MONTHDAYS 39
NEWKEY 138
NOATTACHMENTS 125
NORMALIZED 37
NOW 38
NUMFORMAT 37
PARENTKEY 138
PARSE_RRULE 40
PARSEEMAILSPEC 37
POINTINTIME 39
PREVENTADD 98
RANDOM 43
RECURRENCE_COUNT 40
RECURRENCE_DATE 40
REGEX_FIND 38
REGEX_MATCH 38
REGEX_REPLACE 38
REGEX_SPLIT 38
REJECTITEM 97
RELATIVEASUTC 41
REMOTEDBNAME 124
REMOTEID 97
REMOTERULENAME 44
REQUESTMAXTIME 44
REQUESTMINTIME 44
RFIND 37
SECONDS 39

SESSIONVAR 44
SETALERTCODE 125
SETCONFLICTSTRATEGY 125
SETDBCONNECTSTRING 155
SETDBPASSWORD 155
SETDEBUGLOG 44
SETDEBUGMASK 43
SETDEBUGOPTIONS 43
SETDEFAULTSIZELIMIT 125
SETDEVICEKEY 115
SETDOMAIN 114
SETENDDATE 124
SETFILTER 124
SETFILTERALL 96
SETFLOATING 42
SETLOCALID 97
SETLOG 44
SETLOOSINGCHANGED 99
SETMAXITEMCOUNT 125
SETNOATTACHMENTS 125
SETREADONLY 44
SETRELATIVE 40
SETREMOTEID 97
SETSESSIONVAR 44
SETSIZELIMIT 96
SETSQLFILTER 138
SETSTARTDATE 124
SETSTATUS 128
SETTARGETFILTER 124
SETTIMEZONE 41
SETUSERKEY 115
SETUSERNAME 114
SETUSERTIMEZONE 42
SETWINNINGCHANGED 99
SETXMLTRANSLATE 43
SETZONEOFFSET 41
SHELLEXECUTE 44
SHOWCTCAPPROPERTIES 44
SIGN 43
SIZE 37
SLEEPMS 39
SLOWSYNC 125
SQLCOMMIT 155
SQLEXECUTE 154
SQLFETCHROW 155
SQLGETCOLUMN 155
SQLROLLBACK 155
STARTDATE 124
STATUS 128
STOPADDING 129
SUBSTR 37
SWAP 45
SYNCMLVERS 43
SYNCOP 97, 129
SYSTEMNOW 38
TIMEONLY 39
TIMEUNITS 39
TIMEZONE 41
TREATASLOCALTIME 45
TREATASUTC 45
TYPENAME 45
UNKNOWNDEVICE 115
UPDATECLIENTINSLOWSYNC 45
UPDATESERVEINSLOWSYNC 45
UPPERCASE 37
USERKEY 115
USERTIMEZONE 42
UTCASRELATIVE 41
VTIMEZONE 41
WEEKDAY 39
WINNINGCHANGED 99
WRITING 138
ZONEOFFSET 42