

1

2 **Auto-ID Savant Specification 1.0**

3 Version of 1 September 2003

4

5 This version:

6 <http://...>

7 Latest version:

8 <http://...>

9 Previous versions:

10 [http://develop.autoidcenter.org/archives/sag-savant/att-0034/WD-savant-1_0-](http://develop.autoidcenter.org/archives/sag-savant/att-0034/WD-savant-1_0-20030724.doc)
11 [20030724.doc](http://develop.autoidcenter.org/archives/sag-savant/att-0034/WD-savant-1_0-20030724.doc)

12 [http://develop.autoidcenter.org/archives/sag-savant/att-0028/WD-savant-1_0-](http://develop.autoidcenter.org/archives/sag-savant/att-0028/WD-savant-1_0-20030714.doc)
13 [20030714.doc](http://develop.autoidcenter.org/archives/sag-savant/att-0028/WD-savant-1_0-20030714.doc)

14

15 **Authors:**

16 Sean Clark (Sun Microsystems) sean.clark@sun.com

17 Ken Traub (Connecterra, Inc.) kt@connecterra.com

18 Dipan Anarkat (Uniform Code Council) danarkat@uc-council.org

19 Ted Osinski (Uniform Code Council) tosinski@uc-council.org

20

21 Copyright ©2003 [Auto-ID Center](#)[®], All Rights Reserved.

22

22 **Abstract**

23 This document defines Version 1.0 of the Savant. Savant is software that sits between
24 tag readers and enterprise applications, providing a variety of computational functions on
25 behalf of applications.

26 **Status of this document**

27 This section describes the status of this document at the time of its publication. Other
28 documents may supersede this document. The latest status of this document series is
29 maintained at the Auto-ID Center. This document is the first public working draft.

30 This is an Auto-ID Center Working Draft for review by Auto-ID Members and other
31 interested parties. It is a draft document and may be updated, replaced or made obsolete
32 by other documents at any time. It is inappropriate to use Auto-ID Working Drafts as
33 reference material or to cite them as other than "work in progress". This is work in
34 progress and does not imply endorsement by the Auto-ID membership.

35 Comments on this document should be sent to the Auto-ID Software Action Group
36 Savant Working Group mailing list sag-savant@develop.autoidcenter.org.

37 **Table of Contents**

38	1	Background Information	8
39	2	Introduction	8
40	3	Terminology	8
41	4	EPC System Network Architecture.....	8
42	4.1	EPC Network Software Architecture Components	9
43	4.1.1	Readers.....	9
44	4.1.2	Savant.....	9
45	4.1.3	EPC Information Service	10
46	4.1.4	ONS – Object Name Service	10
47	4.1.5	ONS local cache.....	10
48	4.2	EPC Network Data Standards.....	10
49	4.2.1	Electronic Product Code (EPC)	11
50	4.2.2	Physical Markup Language (PML).....	11
51	4.3	EPC Network Architecture – across Enterprises	11
52	5	Savant Overview	12
53	6	Reader Interface	14

54	6.1	Support for the Auto-ID Reader Protocol 1.0.....	14
55	6.2	Support for other Reader protocols	15
56	6.3	Support for Vendor-specific Reader Protocol Extensions.....	15
57	6.4	Number of Readers.....	15
58	7	Processing Modules.....	15
59	7.1	Definition.....	15
60	7.2	Access to Standard External Interfaces	16
61	7.3	Interaction among Processing Modules.....	16
62	7.4	Interaction with Other External Services.....	16
63	7.5	Registration of Capabilities	17
64	7.6	Support for Standard Processing Modules	17
65	7.7	Processing Module Naming Conventions	17
66	8	Application Interface – Structure	18
67	8.1	Layered Design.....	18
68	8.2	Message Channels	19
69	8.3	Command Structure.....	20
70	8.4	Control Channel Request Message.....	20
71	8.5	Control Channel Response Message	20
72	8.5.1	noSuchModule Response Message	21
73	8.5.2	noSuchCommand Response Message.....	21
74	8.5.3	Error Response Message.....	21
75	8.6	Notification Channel Message.....	21
76	9	Standard Processing Modules	21
77	9.1	The <code>autoid.core</code> Standard Processing Module.....	22
78	9.1.1	GetSavantID Message.....	22
79	9.1.1.1	Request.....	22
80	9.1.1.2	Normal Response	22
81	9.1.1.3	Error Response	22
82	9.1.2	GetCapabilities Message.....	23
83	9.1.2.1	Request.....	23
84	9.1.2.2	Normal Response	23
85	9.1.2.3	Error Response	23

86	9.1.3	Shutdown Message	23
87	9.1.3.1	Request	24
88	9.1.3.2	Normal Response	24
89	9.1.3.3	Error Response	24
90	9.1.4	ResetAll Message.....	24
91	9.1.4.1	Request	24
92	9.1.4.2	Normal Response	24
93	9.1.4.3	Error Response	24
94	9.2	The autoid.readerproxy Standard Processing Module.....	25
95	9.2.1	GetReaders Message.....	25
96	9.2.1.1	Request	25
97	9.2.1.2	Normal Response	25
98	9.2.1.3	Error Response	25
99	9.2.2	RunCommand Message	26
100	9.2.2.1	Request	26
101	9.2.2.2	Normal Response	26
102	9.2.2.3	Error Response	26
103	10	Application Interface – Messaging/Transport Bindings	27
104	10.1	XML-RPC/HTTP MTB	27
105	10.1.1	Why XML-RPC?.....	28
106	10.1.2	Why HTTP 1.1?.....	28
107	10.1.3	Control Channel Request.....	28
108	10.1.4	Control Channel Response	29
109	10.1.5	Core Message Specification	29
110	10.1.5.1	GetSavantID Message.....	29
111	10.1.5.1.1	Request Example	29
112	10.1.5.1.2	Normal Response Example	30
113	10.1.5.1.3	Error Response Example	30
114	10.1.5.2	GetCapabilities Message.....	30
115	10.1.5.2.1	Request Example	30
116	10.1.5.2.2	Normal Response Example	31
117	10.1.5.2.3	Error Response Example	31

118	10.1.5.3 Shutdown Message	31
119	10.1.5.3.1 Request Example	31
120	10.1.5.3.2 Normal Response Example	32
121	10.1.5.3.3 Error Response Example	32
122	10.1.5.4 ResetAll Message.....	32
123	10.1.5.4.1 Request Example	32
124	10.1.5.4.2 Normal Response Example	33
125	10.1.5.4.3 Error Response Example	33
126	10.1.6 ReaderProxy Message Specification	33
127	10.1.6.1 GetReaders.....	33
128	10.1.6.1.1 Request Example	33
129	10.1.6.1.2 Normal Response Example	34
130	10.1.6.1.3 Error Response Example	34
131	10.1.6.2 RunCommand	34
132	10.1.6.2.1 Request Example	34
133	10.1.6.2.2 Normal Response Example	35
134	10.1.6.2.3 Error Response Example	35
135	10.2 SOAP-RPC/HTTP MTB	36
136	10.2.1 Why SOAP?	36
137	10.2.2 Why SOAP RPC?.....	36
138	10.2.3 Why HTTP 1.1?.....	36
139	10.2.4 Savant and Enterprise Applications as a SOAP node.....	37
140	10.2.5 AI Messages	37
141	10.2.6 SOAP Implementation Constraints	38
142	10.2.7 Target Object URI Specification	38
143	10.2.8 AI Schema Architecture	40
144	10.2.9 Core Message Specification	42
145	10.2.9.1 GetSavantID Message.....	42
146	10.2.9.1.1 Request	42
147	10.2.9.1.2 Normal Response.....	42
148	10.2.9.1.3 Error Response	42
149	10.2.9.2 GetCapabilities Message.....	43

150	10.2.9.2.1	Request	43
151	10.2.9.2.2	Normal Response.....	43
152	10.2.9.2.3	Error Response	44
153	10.2.9.3	Shutdown Message	44
154	10.2.9.3.1	Request	44
155	10.2.9.3.2	Normal Response.....	44
156	10.2.9.3.3	Error Response	45
157	10.2.9.4	ResetAll Message.....	45
158	10.2.9.4.1	Request	45
159	10.2.9.4.2	Normal Response.....	45
160	10.2.9.4.3	Error Response	46
161	10.2.10	ReaderProxy Message Specification	46
162	10.2.10.1	GetReaders Message.....	46
163	10.2.10.1.1	Request	46
164	10.2.10.1.2	Normal Response.....	47
165	10.2.10.1.3	Error Response	47
166	10.2.10.2	RunCommand Message	48
167	10.2.10.2.1	Request	48
168	10.2.10.2.2	Normal Response.....	48
169	10.2.10.2.3	Error Response	48
170	10.2.11	Message Format Constraints	49
171	10.2.12	SOAP Terminology	50
172	11	Abbreviations.....	50
173	12	References.....	51
174	13	APPENDIX A: SOAP RPC Schemas.....	52
175	13.1	Core.xsd.....	52
176	13.2	ReaderProxy.xsd	54
177	13.3	Error.xsd.....	56
178	14	APPENDIX B: SOAP RPC Examples	57
179	14.1	Control Channel Request – GetSavantID.....	57
180	14.2	Control Channel Response - GetSavantIDResponse	57
181	15	Appendix C Future Standard Processing Modules	58

182

183

183 **1 Background Information**

184 This document draws from the previous work at the Auto-ID Center, and we recognize
185 the contribution of the following individuals: Sanjay Sarma (MIT), Prasad Putta (Oat
186 Systems), Jim Waldrop (MIT), Dan Engels (MIT), Sridhar Ramachandran (Oat Systems),
187 Gabriel Nasser (Oat Systems).

188 The following papers capture the contributions of these individuals:

- 189 ▪ Engels, D., Foley, J., Waldrop, J., Sarma, S. and Brock, D., "The Networked
190 Physical World: An Automated Identification Architecture" Proceedings of the
191 IEEE/ACM International Conference on Computer Aided Design (ICCAD01), 76-77,
192 2001.

- 193 ▪ The Savant Technical Manual, Version 0.1 (alpha)
194 <http://www.autoidcenter.org/publishedresearch/MIT-AUTOID-TM-003.pdf>

195 **2 Introduction**

196 This document defines Version 1.0 of the Savant. Savant is software that sits between
197 tag readers and enterprise applications, providing a variety of computational functions on
198 behalf of applications.

- 199 ➤ Throughout, questions or areas where revision is needed are indicated by
200 "OpenIssue" paragraphs, like this one.

201 *Elsewhere, comments that seek to provide further explanation of the specification are*
202 *indicated by paragraphs in Italics with a shaded background, like this one. Such*
203 *comments are non-normative, meaning that they are not to be considered authoritative*
204 *insofar as defining the functioning of Savant.*

205 *Some paragraphs of this kind are labeled "Roadmap"; these explain how the authors*
206 *expect the Savant specification to evolve in future versions.*

207 **3 Terminology**

208 The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT",
209 "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in
210 this document are to be interpreted as described in RFC 2119 [[RFC2119](#)]. When these
211 words are used as defined in RFC 2119, they appear in all caps; otherwise, they have
212 their ordinary meanings.

213 **4 EPC System Network Architecture**

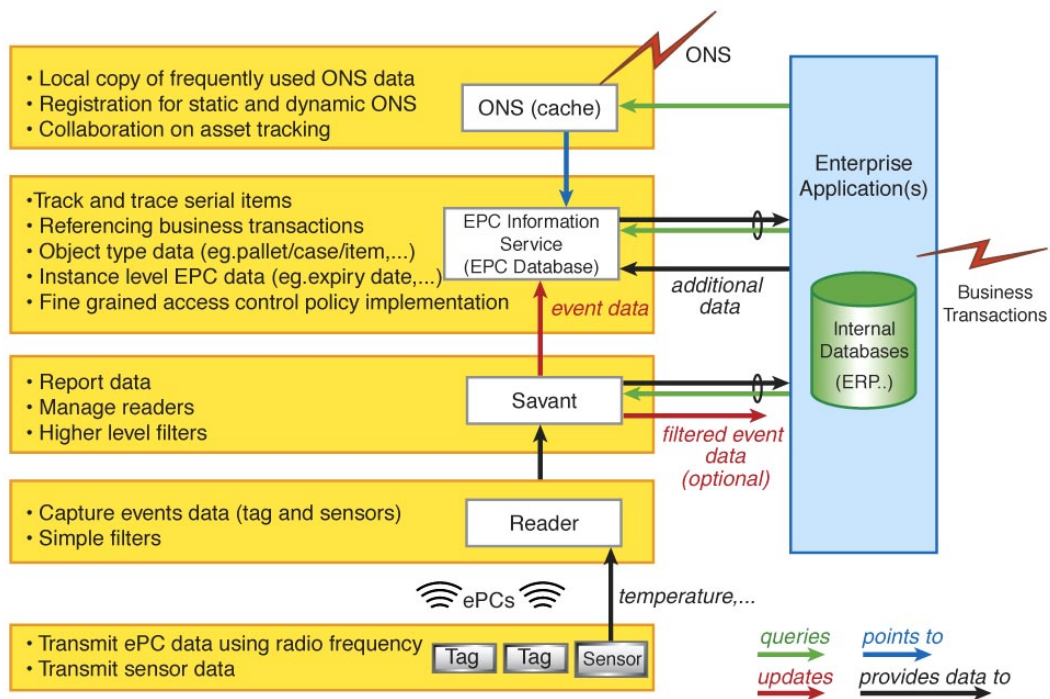
214 Radio Frequency Identification is a technology used to identify, track and locate assets.
215 The vision that drives the developments at the Auto-ID Center is the universal unique
216 identification of individual items. The unique number, called EPC (electronic product
217 code) will be encoded in an inexpensive Radio Frequency Identification (RFID) tag. The

218 EPC Network will also capture and make available (via Internet and for authorized
 219 requests) other information that pertains to a given item to authorized requestors.

220 4.1 EPC Network Software Architecture Components

221 The EPC Network Architecture as in Fig. 1 shows the high-level components of the EPC
 222 Network.

EPC Network Architecture-inside the Enterprise



223

224 **Figure 1: EPC Network Architecture: Components and Layers**

225 These functional components from the figures above are described in the sections below.

226 4.1.1 Readers

227 Readers are devices responsible for detecting when tags enter their read range. They may
 228 also be capable of interrogating other sensors coupled to tags or embedded within tags.

229 The Auto-ID Reader Protocol Specification 1.0 defines a standard protocol by which
 230 Readers communicate with Savants and other hosts. The Savant also has an “adapter”
 231 provision to interface to older readers that do not implement the Auto-ID Reader
 232 Protocol.

233 4.1.2 Savant

234 Savant is “middleware” software designed to process the streams of tag or sensor data
 235 (event data) coming from of one or more reader devices. Savant performs filtering,

236 aggregation, and counting of tag data, reducing the volume of data prior to sending to
237 Enterprise Applications. The Auto-ID Savant Specification 1.0 defines the working of
238 Savant, and the interface to Enterprise Applications.

239 **4.1.3 EPC Information Service**

240 The EPC Information Service makes EPC Network related data available in PML format
241 to requesting services. Data available through the EPC Information Service may include
242 tag read data collected from Savant (for example, to assist with object tracking and
243 tracing at serial number granularity); instance-level data such as date of manufacture,
244 expiry date, and so on; and object class-level data such as product catalog information.
245 In responding to requests, the EPC Information Service draws upon a variety of data
246 sources that exist within an enterprise, translating that data into PML format. When the
247 EPC data is distributed across the supply chain, an industry may create an EPC Access
248 Registry that will act as a repository for EPC Information Service interface descriptions.
249 The Auto-ID EPC Information Service Specification 1.0 defines the protocol for
250 accessing the EPC Information Service.

251 **4.1.4 ONS – Object Name Service**

252 The Object Name Service provides a global lookup service to translate an EPC into one
253 or more Internet Uniform Reference Locators (URLs) where further information on the
254 object may be found. These URLs often identify an EPC Information Service, though
255 ONS may also be used to associate EPCs with web sites and other Internet resources
256 relevant to an object.

257 ONS provides both static and dynamic services. Static ONS typically provides URLs for
258 information maintained by an object's manufacturer. Dynamic ONS services record a
259 sequence of custodians as an object moves through a supply chain.

260 ONS is built using the same technology as DNS, the Domain Name Service of the
261 Internet. The Auto-ID Object Name Service Specification 1.0 defines the working of
262 ONS and its interface to applications.

263 **4.1.5 ONS local cache**

264 The local ONS cache is used to reduce the need to query the global Object Name Service
265 for each object which is seen, since frequently-asked / recently-asked values can be
266 stored in the local cache, which acts as the first port of call for ONS type queries. The
267 local cache may also manage lookup of private internal EPCs for asset tracking. Coupled
268 with the local cache will be registration functions for registering EPCs with the global
269 ONS system and with a dynamic ONS system for private tracking and collaboration
270 within the supply chain seen by each unique object.

271 **4.2 EPC Network Data Standards**

272 The operation of the components of the EPC Network is subject to data standards that
273 specify the syntax and semantics of data exchanged among components.

274 **4.2.1 Electronic Product Code (EPC)**

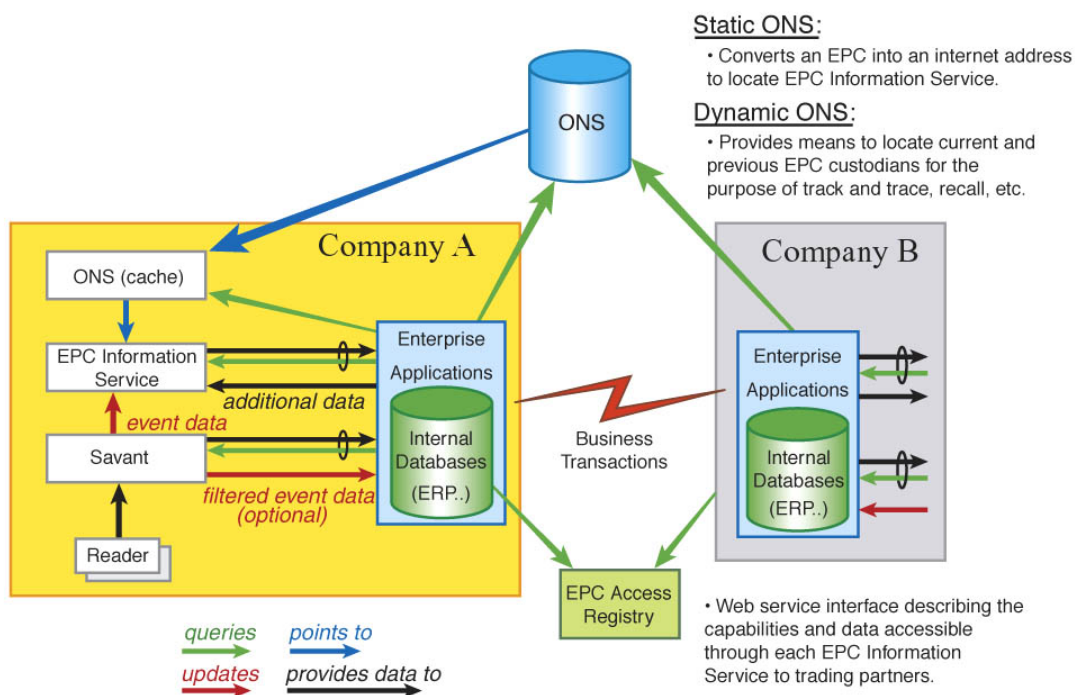
275 The Electronic Product Code is the fundamental identifier for a physical object. The
 276 Auto-ID Electronic Product Code Data Specification 1.0 defines the abstract content of
 277 the Electronic Product Code, and its concrete realization in the form of RFID tags,
 278 Internet URIs, and other representations.

279 **4.2.2 Physical Markup Language (PML)**

280 The Physical Mark-Up Language (PML) is a collection of common, standardized
 281 XML vocabularies to represent and distribute information related to EPC Network
 282 enabled objects. The PML standardizes the content of messages exchanged within the
 283 EPC network. It is, therefore, part of the Auto-ID Center’s effort to develop standardized
 284 interfaces and protocols for the communication with and within the Auto-ID
 285 infrastructure. The core part of the physical mark-up-language (PML Core) provides a
 286 standardized format for the exchange of the data captured by the sensors in the Auto-ID
 287 infrastructure, e.g. RFID readers. The Auto-ID PML Core specification 1.0 defines the
 288 syntax and semantics of PML Core.

289 **4.3 EPC Network Architecture – across Enterprises**

EPC Network Architecture-across Enterprises



290

291

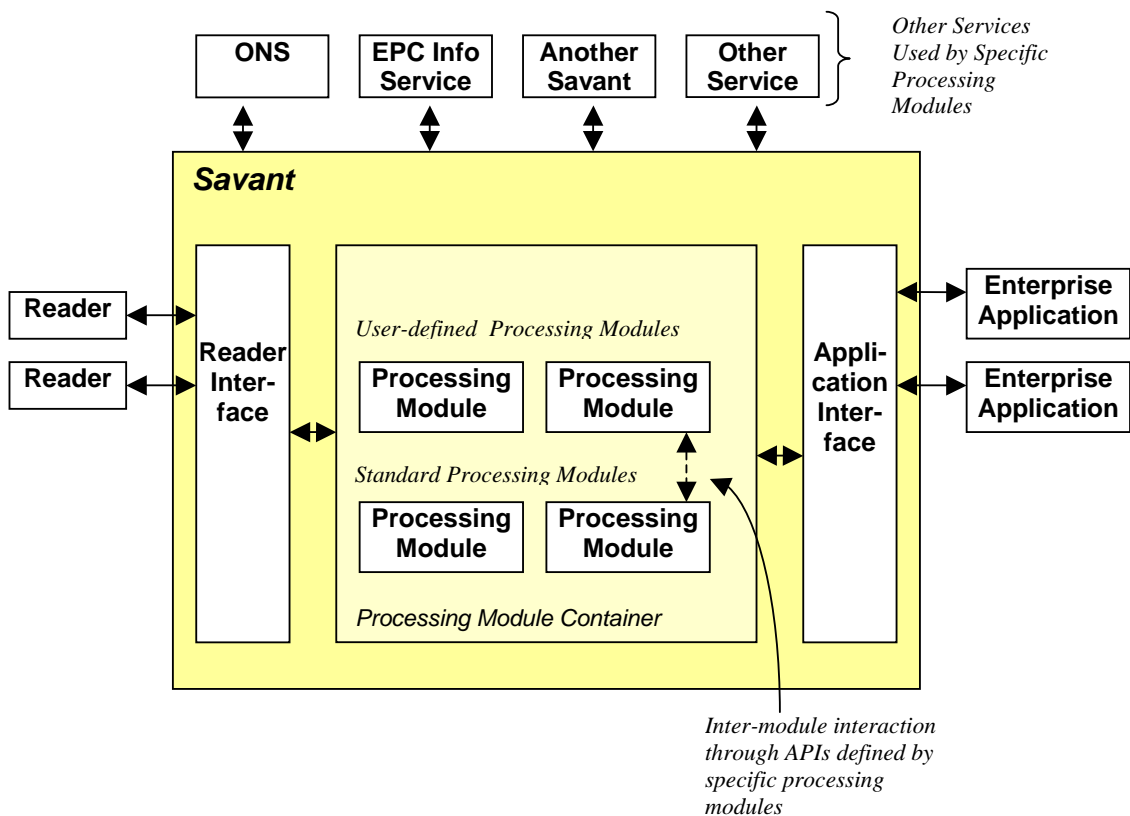
292

Figure 2: EPC Network Architecture: across Enterprises

293 **5 Savant Overview**

294 The Savant is a software system that sits between tag readers and enterprise applications.
 295 It is intended to address the unique computational requirements presented by EPC
 296 applications. Many of the unique challenges arise from the vastly larger quantity of fine-
 297 grained data that originates from RF tag readers, as compared to the granularity of data
 298 that traditional enterprise applications are accustomed to. Hence, quite a bit of
 299 processing performed by Savant is concerned with data reduction operations such as
 300 filtering, aggregation, and counting. Other challenges arise from specific features of the
 301 EPC architecture, including the ONS and PML Service components.

302 Specific requirements for EPC processing vary greatly from application to application.
 303 Moreover, EPC is in its infancy, and as it matures there will be a great deal of innovation
 304 and change in what applications do. Therefore, the emphasis in the specification of
 305 Savant is on extensibility rather than specific processing features. The Savant is defined
 306 in terms of "Processing Modules", or "Services" each of which provides a specific set of
 307 features, and which may be combined by the user to meet the needs of his or her
 308 application. The modular structure is designed to promote innovation by independent
 309 groups of people, avoiding the creation of a single monolithic specification that attempts
 310 to satisfy all needs for everybody.



311
 312 The Savant itself is a container for Processing Modules. Processing Modules interact
 313 with the outside world through two interfaces defined in this specification. The Reader
 314 Interface provides the connection to tag readers, principally RFID readers. The bulk of

315 the details of this interface are specified elsewhere, in the Auto-ID Reader Protocol 1.0
316 specification [\[ReaderProtocol1.0\]](#), though Savant also permits connections to readers via
317 other protocols.

318 The Application Interface provides the connection to external applications, often existing
319 enterprise “back end” applications but possibly to new EPC-specific applications and
320 even to other Savants. The Application Interface is defined by a protocol that is fully
321 specified in this document. The Application Interface is specified in terms of command
322 sets, each command set being defined by a Processing Module. The Application
323 Interface thus serves as a common conduit between Savant processing modules and
324 external applications. (Processing Modules may, if necessary, communicate with pre-
325 existing external services using those services’ native protocols.) The Application
326 Interface is specified using a layered approach similar to that employed in
327 [\[ReaderProtocol1.0\]](#), in which one layer defines the commands and their abstract syntax,
328 and a lower layer specifies a binding to a particular syntax and protocol. Several
329 bindings may be defined.

330 Besides the two external interfaces defined by Savant (Reader Interface and Application
331 Interface), Processing Modules may interact with each other through APIs that they
332 define themselves. Processing Modules may also interact with other external services via
333 interfaces exposed by those services. A special case of such interaction is one Savant
334 interacting with another. This specification, however, does not define how Processing
335 Modules gain access to such external services.

336 *Roadmap (non-normative): It is expected that a future version of this specification will*
337 *specify how processing modules access particular external services, especially EPC*
338 *Information Service, ONS, and other Savant instances.*

339 Processing Modules are either defined by Auto-ID standards or defined by users and
340 other third parties. Those Processing Modules defined by Auto-ID standards are called
341 Standard Processing Modules. Every implementation of Savant must provide an
342 implementation for every Standard Processing Module. Some Standard Processing
343 Modules are further required to be present in every deployed instance of Savant; these are
344 called REQUIRED Standard Processing Modules. Others may be included or omitted by
345 the user in a given deployed instance; these are called OPTIONAL Standard Processing
346 Modules.

347 In the Savant Specification 1.0, there are only two Standard Processing Modules defined.
348 The first is a REQUIRED Standard Processing Module called `autoid.core`. This
349 Standard Processing Module provides a minimal set of Application Interface commands
350 that allow applications to learn what other Processing Modules are available and also to
351 get basic information about what readers are connected. The second is a REQUIRED
352 Standard Processing Module called `autoid.readerproxy`. This Standard
353 Processing Module provides a means for applications to issue commands directly to
354 readers through the Application Interface.

355 *Roadmap (non-normative): As `autoid.core` and `autoid.readerproxy` are the*
356 *only Standard Processing Modules defined in the Savant Specification 1.0, in every*
357 *deployment of Savant 1.0 the majority of functionality will necessarily be provided by*

358 *user-defined Processing Modules. Future versions of the Savant Specification will define*
359 *additional Standard Processing Modules, including for example modules for ONS access,*
360 *EPC Information Service access, and modules that provide standardized counting,*
361 *filtering, and aggregation operations. With these and other Processing Modules, it will*
362 *be possible for users to deploy Savant-based applications without writing additional*
363 *Processing Module software. This is not the case, however, in Version 1.0.*

364 *The Reference Implementation for Savant 1.0 provides Processing Modules beyond what*
365 *is specified by the Savant Specification 1.0. In particular, it provides an Event*
366 *Management System (EMS), a Task Management System (TMS), a Real-time In-memory*
367 *Event Database (RIED), a means for defining plug-in software for EMS and TMS, and a*
368 *configuration system for specifying the interconnection and configuration of plug-ins*
369 *within EMS and TMS. These features, while available in the Reference Implementation,*
370 *are not governed by the Savant Specification 1.0 and are therefore not necessarily*
371 *available in implementations of Savant other than the Reference Implementation.*
372 *Documentation of these features is available elsewhere.*

373 The remainder of this document provides detailed specifications of the components of
374 Savant. Throughout the document, the term “implementation of Savant” is used to refer
375 to a software implementation that conforms to this specification. A “deployed instance of
376 Savant” refers to an implementation of Savant running on a host computer system.

377 Section 6 defines the Reader Interface. Section 7 defines Processing Modules. Section
378 8 defines the Application Interface. Section 9 defines Standard Processing Modules.
379 Section 10 defines the Application Interface Message Transport Bindings

380 **6 Reader Interface**

381 This section specifies the interaction between Savant and reader devices, principally
382 RFID tag readers. All such devices are collectively referred to as “Readers” throughout
383 this section.

384 **6.1 Support for the Auto-ID Reader Protocol 1.0**

385 **RI.6.1.1** An implementation of Savant MUST support interaction with Readers using
386 the Auto-ID Reader Protocol 1.0 [[ReaderProtocol1.0](#)].

387 **RI.6.1.2** An implementation of Savant MUST support at least one of the
388 Messaging/Transport Bindings specified by [[ReaderProtocol1.0](#)].

389 **RI.6.1.3** An implementation of Savant SHOULD support as many
390 Messaging/Transport Bindings specified by [[ReaderProtocol1.0](#)] as is practical.

391 **RI.6.1.4** An implementation of Savant MUST provide a means for Processing
392 Modules to invoke all commands specified by [[ReaderProtocol1.0](#)], including commands
393 defined by [[ReaderProtocol1.0](#)] as RECOMMENDED or OPTIONAL.

394 *Explanation (non-normative): While not every Reader will support every*
395 *RECOMMENDED or OPTIONAL command, Savant must not prevent Processing*
396 *Modules from using such commands in cases where a Reader does support them.)*

397 **6.2 Support for other Reader protocols**

398 **RI.6.2.1** An implementation of Savant MAY support interaction with Readers using
399 protocols other than [\[ReaderProtocol1.0\]](#).

400 *Explanation (non-normative): This permits Savant to interact with legacy readers.*

401 **6.3 Support for Vendor-specific Reader Protocol Extensions**

402 **RI.6.3.1** An implementation of Savant MUST provide a means for Savant Modules to
403 invoke vendor-specific extensions to [\[ReaderProtocol1.0\]](#).

404 ➤ The previous requirement will likely need refinement once the vendor extension
405 mechanism in [\[ReaderProtocol1.0\]](#) is defined.

406 **6.4 Number of Readers**

407 **RI.6.4.1** An implementation of Savant MUST permit interaction with at least one
408 Reader.

409 **RI.6.4.2** Implementations of Savant MAY limit the number of Readers with which
410 simultaneous interaction is permitted. Implementations SHOULD avoid placing arbitrary
411 limits on the number of Readers, and instead allow the number of readers to be limited
412 only by the amount of available memory within Savant or by the underlying operating
413 system's limit on I/O connections.

414 **7 Processing Modules**

415 This section specifies how Processing Modules are defined and used within Savant.

416 **7.1 Definition**

417 **PM.7.1.1** An implementation of Savant contains one or more Processing Modules.
418 An implementation of Savant MAY provide a means to create application-specific
419 Processing Modules.

420 *Roadmap (non-normative): Version 1.0 of the Savant Specification does not specify*
421 *anything about what mechanism is provided for defining extensions. For example, there*
422 *is no specification of any API for defining extensions. The Savant Reference*
423 *Implementation for Version 1.0 does provide particular extension mechanisms, including*
424 *the Adapter, Filter, Queue, and Logger interfaces of the Event Management System*
425 *(EMS), and the Task interface of the Task Management System (TMS). These*
426 *mechanisms are specific features of the Reference Implementation, and are not part of*
427 *the Savant Specification 1.0. They are documented elsewhere.*

428 *It is expected that a future version of the Savant Specification will specify a particular*
429 *mechanism for defining Processing Modules which may or may not be the same as what*
430 *is provided in the Savant Reference Implementation for Version 1.0.*

431 **7.2 Access to Standard External Interfaces**

432 **PM.7.2.1** An implementation of Savant MUST provide a means for Processing
433 Modules to interact with Readers.

434 (See Support for the Auto-ID Reader Protocol 1.0.)

435 **PM.7.2.2** An implementation of Savant MUST provide a means for Processing
436 Modules to receive and respond to commands via the Control Channel of the Application
437 Interface.

438 **PM.7.2.3** An implementation of Savant MUST provide a means for Processing
439 Modules to send asynchronous notifications via the Notification Channel Application
440 Interface.

441 *Roadmap (non-normative): Version 1.0 of the Savant Specification does not specify*
442 *specific APIs through which Processing Modules access the Reader Interface and*
443 *Application Interface. It is expected that specification of such APIs will be a part of a*
444 *future version of the Savant Specification. Those future specifications may or may not be*
445 *compatible with equivalent mechanisms provided by the Savant Reference*
446 *Implementation for Version 1.0. Moreover, the details of asynchronous messaging are*
447 *not described in Version 1.0 and are expected in a later version.*

448 **7.3 Interaction among Processing Modules**

449 **PM.7.3.1** An implementation of Savant MAY permit Processing Modules to expose
450 APIs that are accessible to other Processing Modules in the same deployed instance of
451 Savant. These APIs MAY include operations that are not available through the
452 Application Interface.

453 *Roadmap (non-normative): Version 1.0 of the Savant Specification does not specify*
454 *how such internal APIs are defined or how other Processing Modules discover and use*
455 *them. It is expected that future versions of this specification will specify these aspects.*

456 **7.4 Interaction with Other External Services**

457 **PM.7.4.1** An implementation of Savant MAY permit Processing Modules to
458 establish connections and interact with other external services, via interfaces other than
459 the Reader Interface or the Application Interface.

460 **PM.7.4.2** As a special case of **PM.7.4.1**, an implementation of Savant SHOULD
461 provide access to ONS and EPC Information Services.

462 **PM.7.4.3** As a special case of **PM.7.4.1**, An implementation of Savant MAY permit
463 Processing Modules to establish connections with other instances of Savant as a client of
464 those instances' Application Interfaces.

465 *Explanation (non-normative): This requirement permits Savant-to-Savant interaction,*
466 *enabling a hierarchical mode of deployment sometimes referred to as an "internal*
467 *Savant."*

468 **7.5 Registration of Capabilities**

469 **PM.7.5.1** Processing Modules MUST register their capabilities with the
470 `autoid.core` Processing Module (Section **Fehler! Verweisquelle konnte nicht**
471 **gefunden werden.**) at initialization time. For Version 1 of the Savant Specification the
472 Processing Modules will be required to register their names only with the
473 `autoid.core` Processing Module. Refer to section 9.1.2 for further clarification.

474 **7.6 Support for Standard Processing Modules**

475 **PM.7.6.1** An implementation of Savant MUST always include all Standard
476 Processing Modules defined in Section 9 that are specified as REQUIRED.

477 **7.7 Processing Module Naming Conventions**

478 **PM.7.7.1** Every Processing Module in a given deployed instance of Savant MUST
479 have a unique name.

480 **PM.7.7.2** Processing Module names MUST have the form
481 *component.component.component...*, where ‘.’ is the Unicode “period” character
482 (002E) and each *component* is a non-empty string of Unicode characters, excluding ‘.’.
483 When determining whether two Processing Module names are equal, Savant MUST treat
484 names as case-sensitive.

485 **PM.7.7.3** Processing Module names whose leftmost component is `autoid` or
486 `autoidx` are reserved for Standard Processing Modules (REQUIRED Standard
487 Processing Modules and OPTIONAL Standard Processing Modules, respectively). User-
488 defined and third-party-defined Processing Modules MUST NOT be given names
489 beginning with `autoid` or `autoidx`.

490 **PM.7.7.4** User-defined and third-party-defined Processing Modules SHOULD be
491 given names according to the following convention, which is designed so that Processing
492 Modules defined independently by different organizations will always be given distinct
493 names even without prior coordination. Let *a.b...y.z* be an Internet domain name
494 owned by an organization that authors a Processing Module. Then the organization
495 SHOULD assign a name to the Processing Module of the form *z.y...b.a.XXX*, where
496 *XXX* denotes one or more components chosen by the organization so as not to conflict
497 with the name of any other Processing Module defined by that same organization.

498 *Example: If Sample Software Inc, who owns the domain name `sample.com`, wishes to*
499 *define a processing module for an inventory application, it would choose a name such as*
500 *`com.sample.inventory`.*

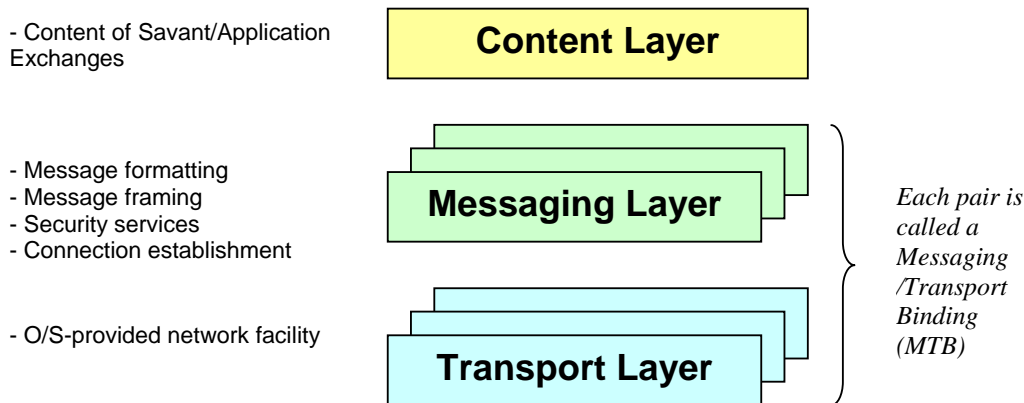
501 *Comment (non-normative): this naming convention is patterned after the naming*
502 *conventions defined by the Java Language Specification [tJLS2] to name Java packages.*
503 *Processing Module names, however, have no defined relationship to Java package*
504 *names, and indeed no implementation of Savant is required to use Java for any purpose.*

505 8 Application Interface – Structure

506 This section specifies the interaction between Savant and applications.

507 8.1 Layered Design

508 The Application Interface is specified in three distinct layers, as illustrated below.



509

510 The layers are:

511 *Content Layer* This layer specifies the abstract content of messages exchanged between
512 the Savant and Applications. This layer is the heart of the Application Interface,
513 defining the operations that are available to applications and what they mean.

514 *Messaging Layer* This layer specifies how abstract messages defined in the Content
515 Layer are encoded (serialized), framed, transformed, and carried on a specific
516 network transport. Security services, if any, are supplied by this layer. (Examples of
517 security services include authentication, authorization, message confidentiality, and
518 message integrity.) The Messaging Layer specifies how an underlying network
519 connection is established, any initialization messages required to establish
520 synchronization or to initialize security services, and any processing such as
521 encryption that is performed on each message.

522 *Transport Layer* This layer corresponds to the networking facilities provided by the
523 operating system or equivalent, and is specified elsewhere.

524 An instance of Savant MAY provide multiple alternative implementations of the
525 Messaging Layer. Each such implementation is called a Messaging/Transport Binding
526 (MTB). Different MTBs provide for different kinds of transport, e.g., TCP/IP versus
527 Bluetooth versus serial line, and for different kinds of messaging protocols, e.g., SOAP
528 versus straight XML versus MQSeries. Different MTBs may also provide different kinds
529 of security services.

530 Several standard MTBs are defined in this specification. Others may be defined and
531 specified in separate, future specifications. Section 27 defines the standard MTBs and
532 specifies which MTBs an implementation of Savant MUST support.

533 Regardless of what MTB is used, an implementation of Savant MUST permit multiple
534 simultaneous, independent connections via the Application Interface. Processing
535 Modules MUST assume that there are concurrent active connections, and implement any
536 locking or other operations to insure correct operation in the face of concurrency.

537 **8.2 Message Channels**

538 The interface between the Content Layer and the Messaging Layer is defined in terms of
539 message channels, each representing an independent communication between a deployed
540 instance of Savant and one Enterprise Application. Because a given instance of Savant
541 may be engaged in multiple, simultaneous, independent connections with Enterprise
542 Applications via the Application Interface, in general there may be many message
543 channels active at once.

544 Message channels are of two kinds:

545 *Control Channel* A control channel carries requests issued by an Enterprise Application
546 to the Savant, and responses to these requests from the Savant to the Application. All
547 messages exchanged on a control channel follow this request/response pattern.

548 *Notification Channel* A notification channel carries messages issued asynchronously by
549 the Savant to an Enterprise Application. Messages on a notification channel only
550 flow in this direction. The notification channel is primarily used to support a mode of
551 operation in which the Savant asynchronously delivers information derived from
552 asynchronous tag reads or other events, without the Enterprise Application needing to
553 poll.

554 When several channels are active, there is no requirement that all active channels use the
555 same MTB. For example, a deployed instance of Savant may have one active Control
556 Channel using an XML-over-HTTP style MTB, one active Notification Channel that uses
557 MQSeries-based MTB, and another active Notification channel that uses an XML-over-
558 HTTP style MTB similar to the Control Channel's MTB but initiated in the opposite
559 direction. This flexibility is required because (a) control channel traffic and notification
560 channel traffic have different patterns, and so have natural affinity to different kinds of
561 underlying protocols; and (b) different Enterprise Applications may impose differing
562 requirements on what protocols are used.

563 In some cases, a Processing Module may wish to send notifications to the same
564 Enterprise Application instance that is issuing it commands. Some MTBs may provide
565 both a Control Channel and an accompanying Notification Channel (a "back channel" of
566 sorts), which it multiplexes over the same underlying Transport Layer connection.

567 *Comment (non-normative): while the Reader Protocol [\[ReaderProtocol1.0\]](#) has*
568 *corresponding notions of layering, MTBs, and Control vs Notification Channels, the*
569 *details are somewhat different. This is because the Reader Protocol has a restriction that*
570 *a Reader is in communication with at most one Host at any one time; hence, there is at*
571 *most one Control Channel and one Notification Channel, and furthermore the two*
572 *channels can always be bundled together. The Application Interface of Savant, in*
573 *contrast, supports multiple simultaneous connections to Enterprise Applications, and so*
574 *the workings of channels require greater generality.*

575 **8.3 Command Structure**

576 Active Control Channels carry request/response traffic between Enterprise Applications
577 and Savant. This traffic is processed in the following manner:

578 An Enterprise Application sends a request message to Savant on a previously established
579 Control Channel. The request message includes a Processing Module name, a
580 command name known to that Processing Module, and any arguments for that
581 command. The specific communication between the Processing Modules is left to the
582 implementer of the Savant.

583 The Savant dispatches the received command name and arguments to the named
584 Processing Module.

585 The Processing Module performs some function according to the received command
586 name and arguments, eventually returning a result (which could be a normal result or
587 an error result).

588 The Savant sends the result back to the Enterprise Application in a response message
589 carried on the same Control Channel that received the original request.

590 If in the first step the request message specifies a Processing Module that is not present in
591 the deployed instance of Savant, the Savant **MUST** respond with a `NoSuchModule`
592 message (defined in Section 8.5.1). If in the third step the Processing Module does not
593 recognize the command name, the Savant **MUST** respond with a `NoSuchCommand`
594 message. Other error conditions such as invalid arguments to a command **MUST** be
595 handled by the individual Processing Module, which **SHOULD** respond with a suitable
596 error response. The error response **SHOULD** follow the model of Error Code and Error
597 Message.

598 **8.4 Control Channel Request Message**

599 This section defines the general format of a Control Channel request message. A Control
600 Channel request message consists of a Processing Module name, the name of a command
601 known to that processing module, and one or more arguments for that command. In
602 specifications of Processing Modules, each command is documented by specifying an
603 ordered list of arguments, with their data types. This constitutes an abstract syntax for
604 the command. A given MTB then defines how this information (Processing Module
605 name, command name, and arguments) is encoded into a concrete message payload.
606 Each MTB **MUST** define specifically how the command set is structured and encoded.

607 **8.5 Control Channel Response Message**

608 This section defines the general format of a Control Channel response message. The
609 details of how the Response Message is structured and encoded **MUST** reside in each
610 MTB section.

611 **8.5.1 noSuchModule Response Message**

612 This section defines the noSuchModule response message generated in response to a
613 request message that names a Processing Module not present in the deployed instance of
614 Savant. The syntax of the Response is specific to each MTB and will be defined therein.

615 **8.5.2 noSuchCommand Response Message**

616 This section defines the noSuchCommand response message generated in response to a
617 request message that names a command not recognized by the specified Processing
618 Module. The syntax of the Response is specific to each MTB and will be defined therein.

619 **8.5.3 Error Response Message**

620 This section defines the general format of error responses generated by Processing
621 Modules. All Standard Processing Modules MUST return errors in a error code and error
622 message format. Error codes starting from 1 to 10000 are reserved for the Standard
623 Processing Modules. Vendor produced Processing Modules MUST NOT use error codes
624 within that range. This version of the Savant Specification does not include a delineated
625 set of standard error code and messages. It is expected that future versions will.

626 **8.6 Notification Channel Message**

627 This section defines the general format of Notification Channel messages. The details of
628 these messages are defined in corresponding sections of each MTB.

629 **9 Standard Processing Modules**

630 This section defines Standard Processing Modules for Savant. Standard Processing
631 Modules have behavior and interfaces that is fully specified within this specification, and
632 are always available in every implementation of Savant (see Support for Standard
633 Processing Modules). Standard Processing Modules designated as REQUIRED must
634 always be present in every deployed instance of Savant. Standard Processing Modules
635 designated as OPTIONAL may be excluded from a particular deployment at the user's
636 option. When an OPTIONAL Standard Processing Module is included, it must comply
637 with all normative specifications that define that Standard Processing Module. The
638 Reference Implementation for Savant will include an implementation of all Standard
639 Processing Modules, whether REQUIRED or OPTIONAL.

640 *Roadmap (non-normative): In the Savant Specification 1.0, there are two REQUIRED*
641 *Standard Processing Module (autoid.core and autoid.readerproxy) and no*
642 *OPTIONAL Standard Processing Modules. It is expected that future versions of the*
643 *Savant Specification will have many more Standard Processing Modules defined.*
644 *Indeed, defining new Standard Processing Modules is the primary way that the Savant*
645 *Specification will be enlarged to include new functionality. This modular structure not*
646 *only keeps the specification easier to understand, but also permits independent*
647 *development of new functionality by independent groups of people.*

648 *Examples of Standard Processing Modules expected to be defined in the near future*
 649 *include: a module to provide ONS access, a module to provide EPC Information Service*
 650 *access, and a module to provide Application Level Events (ALE) functionality. The Event*
 651 *Management System (EMS), Task Management System (TMS), and Real-time In-memory*
 652 *Event Database (RIED) of the Reference Implementation for Savant 1.0 may also become*
 653 *one or more Standard Processing Modules. While most Standard Processing Modules*
 654 *are self-contained and provide a specific package of functionality, EMS and TMS would*
 655 *be examples of Processing Modules that are themselves extensible via the interior plug-in*
 656 *points they provide.*

657 **9.1 The autoid.core Standard Processing Module**

658 The autoid.core Standard Processing Module provides a minimal set of Application
 659 Interface commands that allow applications to learn identity information from a Savant,
 660 what other Processing Modules are available and also some basic Savant management
 661 functions. It is a minimum set that every implementation of Savant must support. The
 662 autoid.core Standard Processing Module is a REQUIRED Standard Processing
 663 Module.

664 **9.1.1 GetSavantID Message**

665 An external application sends an autoid.core.GetSavantID message to interrogate a
 666 Savant for its unique numeric identifier (in all probability, an EPC code). Compliant
 667 systems MUST implement this command.

668 **9.1.1.1 Request**

GetSavantID Request		
Field	Type	Description
None		This function takes no parameters.

669 **9.1.1.2 Normal Response**

GetSavantID Normal Response		
Field	Type	Description
SavantID	String	A unique identification string for the Savant. The string could be an EPC code.

670 **9.1.1.3 Error Response**

671 The error response is returned if the Savant is unable to return a valid EPC code.

GetSavantID Error Response		
Field	Type	Description
ErrorCode	int	Error code.

GetSavantID Error Response		
Field	Type	Description
ErrorString	String	Description of the error.

672 **9.1.2 GetCapabilities Message**

673 An external application sends an `autoid.core.GetCapabilities` message to
 674 interrogate a Savant for a list of all the configured processing modules. Compliant
 675 systems MUST implement this command.

676 **9.1.2.1 Request**

GetCapabilities Request		
Field	Type	Description
None		This function takes no parameters.

677 **9.1.2.2 Normal Response**

GetCapabilities Normal Response		
Field	Type	Description
Capabilities	String array	A string array representation of the processing modules names configured in the Savant. At a minimum the array will have the required standard processing modules.

678 *The get capabilities will not be required to return the commands that a particular*
 679 *processing module supports, in this version. However, in future versions the return value*
 680 *will be a structure that contains the module name, command name and version.*

681 **9.1.2.3 Error Response**

682 The error response is returned if the Savant is unable to return a valid array.

GetCapabilities Error Response		
Field	Type	Description
ErrorCode	int	Error code.
ErrorString	String	Description of the error.

683 **9.1.3 Shutdown Message**

684 An external application sends a `autoid.core.Shutdown` message to inform the Savant to
 685 notify all subsystems to shutdown and then the core processing module will also
 686 shutdown. Compliant systems MUST implement this command.

687 **9.1.3.1 Request**

Shutdown Request		
Field	Type	Description
None		This function takes no parameters.

688 **9.1.3.2 Normal Response**

Shutdown Normal Response		
Field	Type	Description
Shutdown	boolean	True if all subsystems were notified false otherwise

689 **9.1.3.3 Error Response**

690 The error response is returned if the Savant is unable to inform subsystems to shutdown.

Shutdown Error Response		
Field	Type	Description
ErrorCode	int	Error code.
ErrorString	String	Description of the error.

691 **9.1.4 ResetAll Message**

692 An external application sends a `autoid.core.ResetAll` message to inform the Savant to
 693 notify all subsystems to reset to their respective last known configuration and then the
 694 core processing module will also reset. Similar to a warm boot process. Compliant
 695 systems MUST implement this command.

696 **9.1.4.1 Request**

ResetAll Request		
Field	Type	Description
None		This function takes no parameters.

697 **9.1.4.2 Normal Response**

ResetAll Normal Response		
Field	Type	Description
ResetAll	boolean	True if all subsystems were notified false otherwise

698 **9.1.4.3 Error Response**

699 The error response is returned if the Savant is unable to inform subsystems to reset.

ResetAll Error Response		
Field	Type	Description
ErrorCode	int	Error code.
ErrorMessage	String	Description of the error.

700

701 **9.2 The `autoid.readerproxy` Standard Processing Module**

702 The `autoid.readerproxy` Standard Processing Module provides a minimal set of
 703 Application Interface commands that allow external applications to inquire what readers
 704 are connected and configured to interact with the Savant and provide a standard
 705 mechanism for passing commands directly to the individual readers. It is a minimum set
 706 that every implementation of Savant MUST support. The `autoid.readerproxy`
 707 Standard Processing Module is a REQUIRED Standard Processing Module.

708 **9.2.1 GetReaders Message**

709 An external application sends an `autoid.readerproxy.GetReaders` message to
 710 inquire the Savant for all readers it is configured to communicate. The Savant MUST
 711 return all readers that are configured through the ReaderProtocolv1.1 specification. A
 712 Savant MAY return readers that are communicating with the Savant through a non-
 713 standard means. Compliant systems MUST implement this command.

714 **9.2.1.1 Request**

GetReaders Request		
Field	Type	Description
None		This function takes no parameters.

715 **9.2.1.2 Normal Response**

GetReaders Normal Response		
Field	Type	Description
Readers	String array	String array of the Reader ID's the Savant is configured to communicate with.

716 **9.2.1.3 Error Response**

717 The error response is returned if the Savant is unable to return a string array.

GetReaders Error Response		
Field	Type	Description

GetReaders Error Response		
Field	Type	Description
ErrorCode	int	Error code.
ErrorString	String	Description of the error.

718

719 9.2.2 RunCommand Message

720 An external application sends a autoid.readerproxy.RunCommand message to the Savant
 721 as a standard pass through mechanism for applications to take advantage of all commands
 722 any RFID reader may expose. Compliant systems MUST implement this command.

723 9.2.2.1 Request

RunCommand Request		
Field	Type	Description
ReaderID	String	The Reader ID where the command is destined
Command	String	The name of the command.
Arguments	String	The set of arguments needed to execute the command on the reader. That might include a comma separated variable length string with a UTF8 null character termination. The detailed structure of the arguments is defined in the MTB and Reader Protocol 1.0 Specification.

724 9.2.2.2 Normal Response

RunCommand Normal Response		
Field	Type	Description
RunCommand	String	The output from the reader based on execution of the command. The response may be empty indicating there was not a response from the reader.

725 9.2.2.3 Error Response

726 The error response is returned if the Savant cannot communicate with the reader.

RunCommand Error Response		
Field	Type	Description
ErrorCode	int	Error code.
ErrorString	String	Description of the error.

727 **10 Application Interface – Messaging/Transport Bindings**

728 This section defines Messaging/Transport Bindings (MTBs) for the Application Interface.

729 **AIB.10.1.1** An implementation of Savant MUST support one of the Application
 730 Interface MTBs defined in this section. As there is in total only two MTB’s defined in
 731 Version 1.0 of the specification, this requirement is equivalent to saying that an
 732 implementation of Savant MUST support either the XML-RPC/HTTP defined in section
 733 10.1 or SOAP-RPC/HTTP MTB defined in Section 10.1.

734 **10.1 XML-RPC/HTTP MTB**

735 This MTB carries XML-RPC messages over TCP using the HTTP 1.1 protocol
 736 [RFC2616]. A goal of this MTB is to permit enterprise applications an open and simple
 737 means for network based XML data exchanges.

738 Because HTTP is a strict request/response protocol, the same connection cannot be used
 739 for both the Control Channel and the Notification Channel. In this MTB, one HTTP
 740 connection (initiated by the Application) is used to carry the Control Channel, while
 741 another HTTP connection (initiated by the Savant) is used when needed to carry the
 742 Notification Channel in an asynchronous manner. Following is a list of possible tags and
 743 their associated data types in the body XML. Refer to the [\[XML-RPC\]](#) spec for more
 744 detail.

745

Tag	Type	Example(s)
<int>	Four byte signed integer i.e., an integer in the range $-2^{31} \leq x < 2^{31}$	-42
<boolean>	0 (false) 1 (true)	1
<string>	String	Hello world
<double>	double-precision signed floating point number as defined by IEEE 754, represented in decimal. An optional exponent may be included, prefixed by the letter ‘E’ or ‘e’	-12.214 -12.214e42 42e+7 .23 64. 28
<dateTime.iso8601>	date/time in ISO 8601 format, including a timezone	1998-07-17T14:08:55-05:00
<base64>	Base64-encoded binary	eW91IGNhbid0IHJlYWQgdGhpcyE=
<struct>	A value can also be of type <struct>.A <struct>	

	contains <member>s and each <member> contains a <name> and a <value>.	
<array>	A value can also be of type <array>. An <array> contains a single <data> element, which can contain any number of <value>s.	

746

747 **10.1.1 Why XML-RPC?**

748 XML-RPC defines a very simple and easy to implement mechanism for issuing
749 procedure calls over a network to processing modules. The first version of the Savant
750 specification is very basic with the expectation that subsequent versions would become
751 more feature rich, so XML-RPC maps conveniently to the current definitions and
752 invocations of Standard Processing Module requirements. It does not define actual
753 mappings to any particular programming language and the representation is entirely
754 platform independent, which was another goal of version 1. Also, the Savant reference
755 implementation of the AI can be effectively implemented using the XML-RPC
756 mechanism.

757 **10.1.2 Why HTTP 1.1?**

758 The current version of the XML-RPC specification is mapped on top of an HTTP post
759 request and since [HTTP 1.1] is ubiquitous and easier to implement, it is the protocol of
760 choice for use in this MTB. Also the inherent request/response characteristics of HTTP
761 provide an efficient solution for implementing the request/response pattern followed by
762 the Control Channel.

763 **10.1.3 Control Channel Request**

764 An XML-RPC message is an HTTP-POST request. The body of the request is in XML. A
765 procedure executes on the server and the value it returns is also formatted in XML.

766 Procedure parameters can be scalars, numbers, strings, dates, etc.; and can also be
767 complex record and list structures. The control channel port number SHOULD be
768 configurable during installation. The default port MUST be port 8080. The request
769 format looks as follows:

770

771 POST /Savant/ApplicationInterface HTTP/1.1

772 User-Agent: [Some Enterprise Application]

773 Host: mycompany.example.org

774 Content-Type: text/xml

775 Content-length: 181

776
777 <?xml version="1.0"?>
778 <methodCall>
779 <methodName>autoid.core.GetSavantID</methodName>
780 </methodCall>
781

782 The value of the XML-RPC `methodName` element is formed by concatenating the
783 Processing Module name, a dot character, and the name of an operation defined by that
784 processing module.

785 **10.1.4 Control Channel Response**

786 The XML-RPC response is similar to the request in that it follows the HTTP 1.1 response
787 pattern for the heading with an XML body as the content. An example is as follows:

788
789 HTTP/1.1 200 OK
790 Connection: close
791 Content-Length: 158
792 Content-Type: text/xml
793 Date: Fri, 17 Jul 1998 19:55:08 GMT
794 Server: [Some Enterprise Application]
795
796 <?xml version="1.0"?>
797 <methodResponse>
798 <params>
799 <param>
800 <value>
801 <string> urn:epc:1:2.24.400</string>
802 </value>
803 </param>
804 </params>
805 </methodResponse>
806

807 **10.1.5 Core Message Specification**

808 The AI messages supported by the `autoid.core` Standard Processing Module are
809 detailed in the following sections.

810 **10.1.5.1 GetSavantID Message**

811 ***10.1.5.1.1 Request Example***

812 <?xml version="1.0"?>

```
813     <methodCall>
814         <methodName>autoid.core.GetSavantID</methodName>
815     </methodCall>
```

816 **10.1.5.1.2 Normal Response Example**

```
817     <?xml version="1.0"?>
818     <methodResponse>
819         <params>
820             <param>
821                 <value>
822                     <string>urn:epc:1:2.24.400</string>
823                 </value>
824             </param>
825         </params>
826     </methodResponse>
```

827 **10.1.5.1.3 Error Response Example**

```
828     <?xml version="1.0"?>
829     <methodResponse>
830         <fault>
831             <value>
832                 <struct>
833                     <member>
834                         <name>ErrorCode</name>
835                         <value><int>1</int></value>
836                     </member>
837                     <member>
838                         <name>ErrorString</name>
839                         <value><string>SAVANT_AI_ERROR_1.</string></value>
840                     </member>
841                 </struct>
842             </value>
843         </fault>
844     </methodResponse>
```

845 **10.1.5.2 GetCapabilities Message**

846 **10.1.5.2.1 Request Example**

```
847     <?xml version="1.0"?>
848     <methodCall>
849         <methodName>autoid.core.GetCapabilities</methodName>
850     </methodCall>
```

851 **10.1.5.2.2 Normal Response Example**

```
852 <?xml version="1.0"?>
853 <methodResponse>
854   <params>
855     <param>
856       urn:epc:1:2.24.400
857       <value>
858         <array>
859           <data>
860             <string>autoid.core</string>
861             <string>autoid.readerproxy</string>
862             <string>org.example.foo</string>
863           </data>
864         </array>
865       </value>
866     </param>
867   </params>
868 </methodResponse>
```

869 **10.1.5.2.3 Error Response Example**

```
870 <?xml version="1.0"?>
871 <methodResponse>
872   <fault>
873     <value>
874       <struct>
875         <member>
876           <name>ErrorCode</name>
877           <value><int>2</int></value>
878         </member>
879         <member>
880           <name>ErrorString</name>
881           <value><string>SAVANT_AI_ERROR_2.</string></value>
882         </member>
883       </struct>
884     </value>
885   </fault>
886 </methodResponse>
```

887 **10.1.5.3 Shutdown Message**

888 **10.1.5.3.1 Request Example**

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

```
890     <methodCall>
891         <methodName>autoid.core.Shutdown</methodName>
892     </methodCall>
```

893 **10.1.5.3.2 Normal Response Example**

```
894     <?xml version="1.0"?>
895     <methodResponse>
896         <params>
897             <param>
898                 <value>
899                     <boolean>0</boolean>
900                 </value>
901             </param>
902         </params>
903     </methodResponse>
```

904 **10.1.5.3.3 Error Response Example**

```
905     <?xml version="1.0"?>
906     <methodResponse>
907         <fault>
908             <value>
909                 <struct>
910                     <member>
911                         <name>ErrorCode</name>
912                         <value><int>3</int></value>
913                     </member>
914                     <member>
915                         <name>ErrorString</name>
916                         <value><string>SAVANT_AI_ERROR_3.</string></value>
917                     </member>
918                 </struct>
919             </value>
920         </fault>
921     </methodResponse>
```

922 **10.1.5.4 ResetAll Message**

923 **10.1.5.4.1 Request Example**

```
924     <?xml version="1.0"?>
925     <methodCall>
926         <methodName>autoid.core.ResetAll</methodName>
927     </methodCall>
```


928 **10.1.5.4.2 Normal Response Example**

```
929     <?xml version="1.0"?>
930     <methodResponse>
931         <params>
932             <param>
933                 <value>
934                     <boolean>0</boolean>
935                 </value>
936             </param>
937         </params>
938     </methodResponse>
```

939 **10.1.5.4.3 Error Response Example**

```
940     <?xml version="1.0"?>
941     <methodResponse>
942         <fault>
943             <value>
944                 <struct>
945                     <member>
946                         <name>ErrorCode</name>
947                         <value><int>4</int></value>
948                     </member>
949                     <member>
950                         <name>ErrorString</name>
951                         <value><string>SAVANT_AI_ERROR_4.</string></value>
952                     </member>
953                 </struct>
954             </value>
955         </fault>
956     </methodResponse>
```

957 **10.1.6 ReaderProxy Message Specification**

958 The AI messages supported by the `autoid.readerproxy` Standard Processing
959 Module are detailed in the following sections.

960 **10.1.6.1 GetReaders**

961 **10.1.6.1.1 Request Example**

```
962     <?xml version="1.0"?>
963     <methodCall>
964         <methodName>autoid.readerproxy.GetReaders</methodName>
965     </methodCall>
```

966 **10.1.6.1.2 Normal Response Example**

```
967     <?xml version="1.0"?>
968     <methodResponse>
969         <params>
970             <param>
971                 <value>
972                     <array>
973                         <data>
974                             <value><string>urn:epc:1:2.24.401</string></value>
975                             <value><string>urn:epc:1:2.24.402</string></value>
976                             <value><string>urn:epc:1:2.24.403</string></value>
977                             <value><string>urn:epc:1:2.24.404</string></value>
978                         </data>
979                     </array>
980                 </value>
981             </param>
982         </params>
983     </methodResponse>
```

984 **10.1.6.1.3 Error Response Example**

```
985     <?xml version="1.0"?>
986     <methodResponse>
987         <fault>
988             <value>
989                 <struct>
990                     <member>
991                         <name>ErrorCode</name>
992                         <value><int>5</int></value>
993                     </member>
994                     <member>
995                         <name>ErrorString</name>
996                         <value><string>SAVANT_AI_ERROR_5</string></value>
997                     </member>
998                 </struct>
999             </value>
1000         </fault>
1001     </methodResponse>
```

1002 **10.1.6.2 RunCommand**

1003 **10.1.6.2.1 Request Example**

```
1004     <?xml version="1.0"?>
```

```

1005 <methodCall>
1006     <methodName>autoid.readerproxy.RunCommand</methodName>
1007     <params>
1008         <param>
1009             <struct>
1010                 <member>
1011                     <name>ReaderID</name>
1012                     <value><string>urn:epc:1.2.24.401</string></value>
1013                 </member>
1014                 <member>
1015                     <name>Command</name>
1016                     <value><string>GetTagList</string></value>
1017                 </member>
1018             </struct>
1019         </param>
1020     </params>
1021 </methodCall>

```

1022 **10.1.6.2.2 Normal Response Example**

```

1023 <?xml version="1.0"?>
1024 <methodResponse>
1025     <params>
1026         <param>
1027             <value>
1028                 <array>
1029                     <data>
1030                         <value><string>urn:epc:1:2.30.333</string></value>
1031                         <value><string>urn:epc:1:2.30.334</string></value>
1032                         <value><string>urn:epc:1:2.30.335</string></value>
1033                         <value><string>urn:epc:1:2.30.336</string></value>
1034                     </data>
1035                 </array>
1036             </value>
1037         </param>
1038     </params>
1039 </methodResponse>

```

1040 **10.1.6.2.3 Error Response Example**

```

1041 <?xml version="1.0"?>
1042 <methodResponse>
1043     <fault>
1044         <value>

```

```

1045     <struct>
1046         <member>
1047             <name>ErrorCode</name>
1048             <value><int>6</int></value>
1049         </member>
1050         <member>
1051             <name>ErrorString</name>
1052             <value><string>SAVANT_AI_ERROR_6.</string></value>
1053         </member>
1054     </struct>
1055 </value>
1056 </fault>
1057 </methodResponse>

```

1058 **10.2 SOAP-RPC/HTTP MTB**

1059 This MTB carries SOAP RPC messages over TCP using the [HTTP 1.1] protocol. The
1060 goal of this MTB is to permit enterprise applications an open and simple means for
1061 SOAP RPC based XML data exchanges.

1062 All parts of this specification are normative, with the exception of examples and sections
1063 explicitly marked as "Non-Normative" or "Informative"

1064 **10.2.1 Why SOAP?**

1065 SOAP is standard message exchange paradigm for exchanging structured and typed
1066 information between peers in a decentralized and distributed environment. It provides a
1067 framework by which application specific data can be conveyed in an extensible manner.
1068 More information on SOAP can be found in the references section.

1069 **10.2.2 Why SOAP RPC?**

1070 SOAP defines a uniform representation for RPC requests and responses and facilitates the
1071 exchange of messages that map conveniently to definitions and invocations of method
1072 and procedure calls. It does not define actual mappings to any particular programming
1073 language and the representation is entirely platform independent. The Savant AI can be
1074 effectively implemented using the SOAP RPC mechanism.

1075 **10.2.3 Why HTTP 1.1?**

1076 SOAP RPC can be implemented independent of any particular transport protocol binding,
1077 but since [HTTP 1.1] is ubiquitous and easier to implement, it is the protocol of choice
1078 for use in this MTB. Also the inherent request/response characteristics of HTTP provide
1079 an efficient solution for implementing the request/response pattern followed by the
1080 Control Channel.

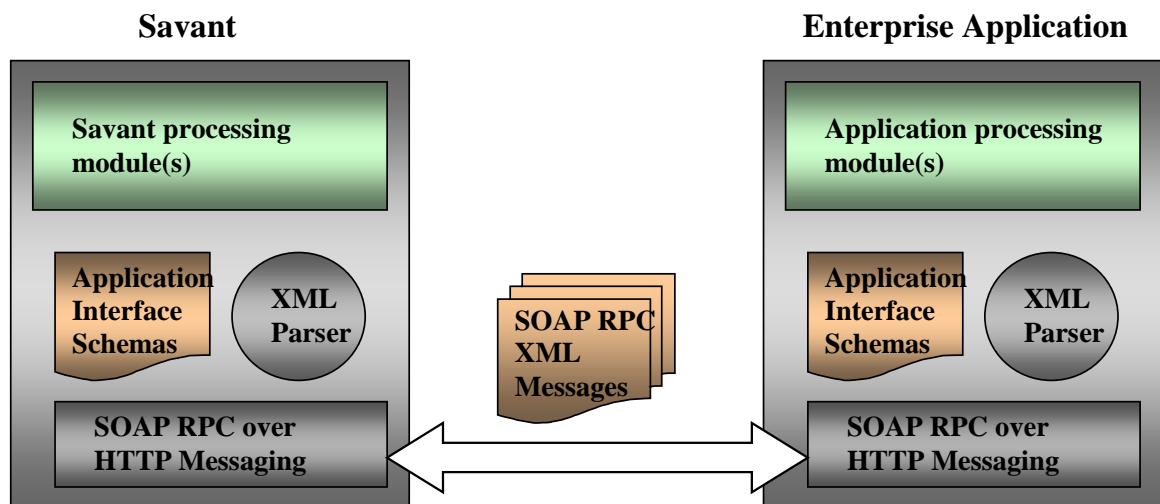
1081 Because HTTP is a strict request/response protocol, the same connection cannot be used
1082 for both the Control Channel and the Notification Channel. In this MTB, one HTTP

1083 connection (initiated by the Application) is used to carry the Control Channel, while
1084 another HTTP connection (initiated by the Savant) is used when needed to carry the
1085 Notification Channel in an asynchronous manner.

1086 **10.2.4 Savant and Enterprise Applications as a SOAP node**

1087 The interaction between Savant and enterprise applications will be implemented as SOAP
1088 RPC exchanges. As such the Savant and the corresponding application in the exchange
1089 will behave as SOAP nodes and need to expose a SOAP RPC interface using the HTTP
1090 protocol binding as described in [SOAP Part 2]. How the SOAP module is integrated
1091 with the Savant and the enterprise application is irrelevant from the MTB standpoint and
1092 what really matters is that they can expose a SOAP interface for external message
1093 exchange between them. For the remainder of this document we use the term SOAP node
1094 as a generalized reference for a Savant and enterprise application that implements this
1095 MTB.

Savant Application Interface SOAP RPC Messaging



1096
1097

1098 **10.2.5 AI Messages**

1099 This MTB prescribes SOAP RPC formatted XML messages as the information content of
1100 the Control and Notification Channel messages, exchanged between the SOAP nodes.
1101 For Control Channel Messages the enterprise application is the SOAP sender and Savant
1102 is the SOAP receiver and for Notification channel messages it is vice-versa.

1103
1104 The SOAP message when received by the SOAP receiver must be dispatched to the
1105 appropriate processing module and the appropriate standard interface must be invoked.
1106 The SOAP message needs to have enough information for the SOAP receiver to be able

1107 to do this. If the processed SOAP message is a Control Channel request then the
1108 corresponding response also needs to be returned to the SOAP sender. Thus, standard
1109 RPC parameters need to be defined for the SOAP receiver to be able to communicate
1110 with a Standard Processing Module. Section 9 of this specification declares Standard
1111 Processing Modules and defines the messages supported by them. This MTB provides
1112 SOAP RPC representations of the message parameters that must be used to communicate
1113 with the Standard Processing Modules. These SOAP messages containing the standard
1114 RPC parameters are called AI messages.

1115 The following sections provide the actual implementation details of this particular MTB.
1116 Essentially, it provides the rules for SOAP message formatting and RPC construction for
1117 the AI messages. For the remainder of this document we will use the term ‘AI message’
1118 as a specific reference for SOAP Control and Notification channel XML Messages
1119 formatted in SOAP RPC and exchanged between two SOAP nodes.

1120 **10.2.6 SOAP Implementation Constraints**

1121 As explained in the sections above Savant and the enterprise applications behave as
1122 SOAP nodes in the message exchange. This section identifies specific SOAP features
1123 which are required to be implemented by a SOAP node. It is assumed that the reader has
1124 read and understood [SOAP Part 1] and [SOAP Part 2] specification documents and
1125 understands RPC based messaging. The SOAP version 1.2 specification has many
1126 features that are optional and are not required to be implemented. Implementation of this
1127 MTB requires a SOAP implementation to implement some of these optional features. The
1128 following SOAP features MUST be implemented by a SOAP implementation to be
1129 compliant to this MTB.

1130

- 1131 1. The SOAP implementation MUST be conformant of the SOAP version 1.2
1132 specifications as defined in section 1.2 “Conformance” of [SOAP Part 1].
- 1133 2. The SOAP implementation MUST implement the SOAP encoding as described in
1134 section 3 “SOAP Encoding” of [SOAP Part 2] which is also identified by the URI
1135 “<http://www.w3.org/2003/05/soap-encoding>”.
- 1136 3. The SOAP implementation MUST implement the SOAP RPC representation as
1137 described in section 4 “SOAP RPC Representation” of [SOAP Part 2] which is
1138 also identified by the URI “<http://www.w3.org/2003/05/soap-rpc>”.
- 1139 4. The SOAP implementation MUST implement the SOAP HTTP binding as
1140 described in section 7 “SOAP HTTP Binding” of [SOAP Part 2] and identified by
1141 the URI “<http://www.w3.org/2003/05/soap/bindings/HTTP>”.

1142 **10.2.7 Target Object URI Specification**

1143 This section specifies a standard mechanism for addressing the Standard Processing
1144 Modules on a given SOAP node. For this particular MTB the Savant server is a SOAP
1145 node that exposes a SOAP RPC service interface over an HTTP binding. The Standard
1146 Processing Modules as declared in section 9 can thus be identified as web resources on
1147 the Savant server. These modules expose well-defined service interfaces that can be

1148 invoked to retrieve state information or to modify some state that it owns. The SOAP
1149 RPC client calls MUST be directed to these modules on the Savant server for processing
1150 via SOAP RPC services.

1151 In adherence of the SOAP HTTP binding as specified in section 7 ‘SOAP HTTP
1152 Binding’ of [SOAP Part 2] and following the best practices for SOAP usage over HTTP
1153 as outlined in section 4.1.3 ‘Web Architecture Compatible usage of SOAP’ of [SOAP
1154 Part 0], the Standard Processing Modules on the Savant server can be identified as web
1155 resources. These web resources will be the targets of the SOAP RPC client call. The
1156 Standard Processing Modules as web resources on the Savant server can thus be exposed
1157 via a SOAP RPC service, as the target object of the SOAP RPC client call. In other words
1158 a Standard Processing Module can be understood to be a SOAP RPC service on the
1159 Savant server. Every service on a given server is identifiable by a URI commonly called
1160 as ‘target object URI’ and is called so as the URI is the target object of the client call.
1161 Standard URNs can be used for the target object URI. Below, we define standard URNs
1162 called as a ‘Service URN’ that MUST be used as the target object URI for the service
1163 corresponding to a Standard Processing Module on the Savant server.

Standard Processing Module Name	Service URN
autoid.core	urn:autoid:specification:interchange:Savant:core:xml:service:1
autoid.readerproxy	urn:autoid:specification:interchange:Savant:readerproxy:xml:service:1

1164

1165 The service URN above has been constructed following the namespace design guidelines
1166 as outlined in the “PML Namespace Design Guidelines” section of [PML Core]. The
1167 structure of the namespace specific scheme (NSS) part for the ‘autoid’ namespace ID
1168 (NID) as outlined in this section has a ‘*specification-subclass*’ and a ‘*specification-ID*’
1169 hierarchy. The service URNs above have been formulated by identifying ‘Savant’ as the
1170 value for the ‘*specification-subclass*’ hierarchy and by mechanically mapping the name
1171 of the Standard Processing Module as the value for the ‘*specification-ID*’ hierarchy. This
1172 approach MUST be followed for constructing a service URN corresponding to a Standard
1173 Processing Module.

1174 For Non-standard Processing Modules, the authority defining the module MAY use the
1175 same approach as followed above for defining the service URN. This SHOULD be in a
1176 namespace authoritative of the Non-standard Processing Module owner.

1177 The service endpoint URL is the location of the SOAP RPC service as identified by the
1178 target object URI, mutually agreed upon between the SOAP sender and the SOAP
1179 receiver. The URL used for the SOAP RPC client call is commonly called as ‘action
1180 URL’. The action URL for the SOAP RPC client call then will be the concatenation of
1181 the service endpoint URL and the target object URI. For the underlying HTTP request the
1182 target object URI will be a part of the request URI formatted to invoke the SOAP RPC
1183 service. The SOAP RPC client calls must then be dispatched to the Standard Processing
1184 Modules on the Savant server as identified by these service URNs.

1185 An example of the above is:

1186

1187 POST /urn:autoid:specification:interchange:Savant:core:xml:service:1 HTTP/1.1

1188 Host: mycompany.example.org

1189 Content-Type: application/soap+xml; charset="utf-8"

1190 Content-Length: nnnn

1191

1192 <?xml version='1.0'?>

1193 <env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope" >

1194 <env:Body>

1195

1196 <!-- Standard SOAP RPC's for Savant AI embedded here -->

1197

1198 </env:Body>

1199 </env:Envelope>

1200 10.2.8 AI Schema Architecture

1201 As described previously, the AI messages are SOAP RPC messages. In accordance with
1202 the SOAP specification, the RPC parameters are embedded in the ‘Body’ element of the
1203 SOAP Envelope. SOAP version 1.2 allows us to strongly type these RPC parameters by
1204 providing the capability to define them in XML Schema syntax. Therefore we have
1205 defined the RPC parameters for all the AI messages in XML Schema syntax. AI
1206 messages are categorized by its Standard Processing Module, which processes them. As
1207 such we provide one XML Schema corresponding to every Standard Processing Module
1208 defined in the Savant specification. Each schema contains the definition of the SOAP
1209 RPC parameters corresponding to the AI messages that can be processed by the module.
1210 The schemas follow the PML design methodology as outlined in the sections “PML
1211 Design Methodology Overview” and “PML Namespace Design Guidelines” sections of
1212 [PML Core]. The AI schemas can be found in the Appendix section of this document.

1213 The AI message specification consists of the following 3 AI schemas

- 1214 1. **Core.xsd**: Contains definitions of messages supported by the “autoid.core”
1215 Standard Processing Module.
- 1216 2. **ReaderProxy.xsd**: Contains definitions of messages supported by the
1217 “autoid.readerproxy” Standard Processing Module.
- 1218 3. **Error.xsd**: Contains type definitions of reusable Savant domain types that are
1219 reused by types from previous 2 schemas.

1220 For this specification, the XML schema documents listed below are normative for the
1221 following namespaces:

Namespace	File Name
urn:autoid:specification:interchange:Savant:core:xml:soap-rpc:1	Core.xsd

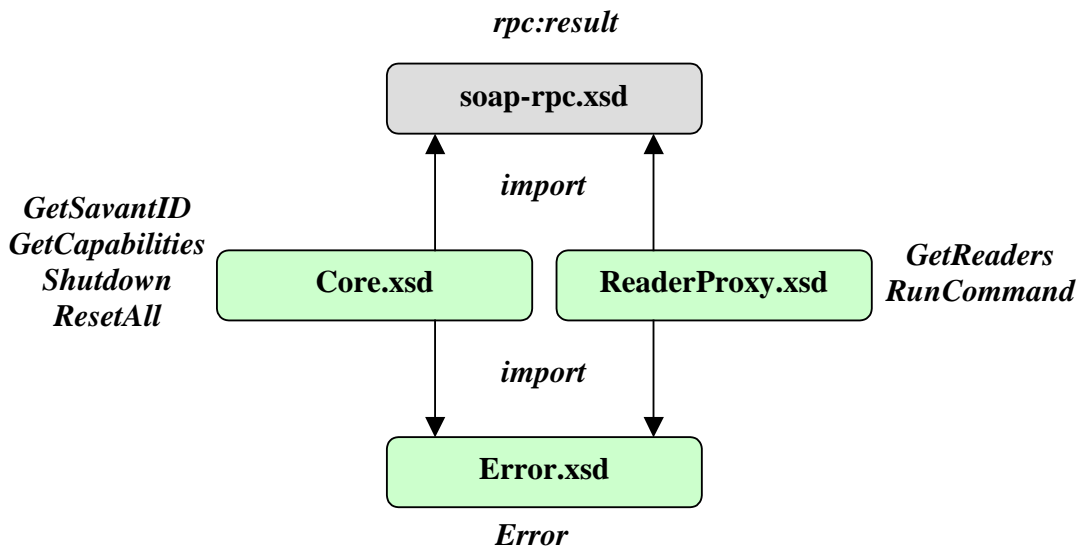
urn:autoid:specification:interchange:Savant:readerproxy:xml:soap-rpc:1	ReaderProxy.xsd
urn:autoid:specification:domain:Savant>Error:xml:soap-rpc:1	Error.xsd

1222

1223 The location of the normative XML schema documents from SOAP version 1.2
 1224 specifications is specified in section 1.1 “Notational Conventions” of both [SOAP Part 1]
 1225 and [SOAP Part 2]. Local copies of these schema documents have been provided for
 1226 reference. The following SOAP schemas have been referenced from the AI schemas.

1. **soap-rpc.xsd**: Contains definitions of SOAP RPC parameters. The SOAP RPC “result” element is referenced in the AI schemas.

1229 The schema package for this specification additionally includes the ‘soap-envelope.xsd’
 1230 and ‘soap-encoding.xsd’ XML schema documents from [SOAP Part 1] and [SOAP Part
 1231 2] respectively. Though the AI schemas do not require them, they MAY be used for local
 1232 validation of a SOAP message.



1233
 1234

1235 The AI schemas listed above are normative for the Standard Processing Modules defined
 1236 in this specification and MUST be implemented by a compliant Savant. For Non-standard
 1237 Processing Modules, the authority defining the module MAY use a similar architecture
 1238 and design as followed above for the schemas that define the AI messages supported by
 1239 the module in consideration. The target namespace URIs for the schemas SHOULD be
 1240 defined in a namespace authoritative of the Non-standard Processing Module owner.

1241 The following sections provide the SOAP RPC parameters for each AI message
 1242 categorized by its Standard Processing Module as defined in section 9.

1243 10.2.9 Core Message Specification

1244 The AI messages supported by the `autoid.core` Standard Processing Module are
1245 defined in the 'Core.xsd' schema. The following sections define the message parameters
1246 for each message.

1247 10.2.9.1 GetSavantID Message

1248 10.2.9.1.1 Request

1249 Functional Specification

1250 For the functional specification for this message refer to section 9.1.1.1

1251 XML Message Specification

1252 The 'GetSavantID' request message is implemented as the *GetSavantID* message
1253 element.

1254 Example

```
1255 <?xml version="1.0" encoding="UTF-8"?>  
1256 <core:GetSavantID xmlns:core="urn:autoid:specification:interchange:Savant:core:xml:soap-rpc:1"  
1257 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
1258 xsi:schemaLocation="urn:autoid:specification:interchange:Savant:core:xml:soap-rpc:1  
1259 ../SchemaFiles/Interchange/Core.xsd"/>
```

1260 10.2.9.1.2 Normal Response

1261 Functional Specification

1262 For the functional specification for this message refer to section 9.1.1.2

1263 XML Message Specification

1264 The 'GetSavantID' normal response message is implemented as the
1265 *GetSavantIDResponse* message element.

1266 Example

```
1267 <?xml version="1.0" encoding="UTF-8"?>  
1268 <core:GetSavantIDResponse xmlns:core="urn:autoid:specification:interchange:Savant:core:xml:soap-rpc:1"  
1269 xmlns:rpc="http://www.w3.org/2003/05/soap-rpc" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
1270 xsi:schemaLocation="urn:autoid:specification:interchange:Savant:core:xml:soap-rpc:1  
1271 ../SchemaFiles/Interchange/Core.xsd">  
1272 <rpc:result>core:SavantID</rpc:result>  
1273 <core:SavantID>urn:epc:1:2.24.400</core:SavantID>  
1274 </core:GetSavantIDResponse>
```

1275 10.2.9.1.3 Error Response

1276 Functional Specification

1277 For the functional specification for this message refer to section 9.1.1.3

1278 XML Message Specification

1279 The 'GetSavantID' error response message is implemented as the *GetSavantIDError*
1280 message element.

1281 **Example**

```
1282 <?xml version="1.0" encoding="UTF-8"?>  
1283 <core:GetSavantIDError xmlns:core="urn:autoid:specification:interchange:Savant:core:xml:soap-rpc:1"  
1284 xmlns:err="urn:autoid:specification:domain:Savant:Error:xml:soap-rpc:1"  
1285 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
1286 xsi:schemaLocation="urn:autoid:specification:interchange:Savant:core:xml:soap-rpc:1  
1287 ../SchemaFiles/Interchange/Core.xsd">  
1288 <err:ErrorCode>1</err:ErrorCode>  
1289 <err:ErrorString>GET_SAVANT_ID_ERROR</err:ErrorString>  
1290 </core:GetSavantIDError>  
1291
```

1292 **10.2.9.2 GetCapabilities Message**

1293 **10.2.9.2.1 Request**

1294 **Functional Specification**

1295 For the functional specification for this message refer to section 9.1.2.1

1296 **XML Message Specification**

1297 The 'GetCapabilities' request message is implemented as the *GetCapabilities* message
1298 element.

1299 **Example**

```
1300 <?xml version="1.0" encoding="UTF-8"?>  
1301 <core:GetCapabilities xmlns:core="urn:autoid:specification:interchange:Savant:core:xml:soap-rpc:1"  
1302 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
1303 xsi:schemaLocation="urn:autoid:specification:interchange:Savant:core:xml:soap-rpc:1  
1304 ../SchemaFiles/Interchange/Core.xsd"/>  
1305
```

1306 **10.2.9.2.2 Normal Response**

1307 **Functional Specification**

1308 For the functional specification for this message refer to section 9.1.2.2

1309 **XML Message Specification**

1310 The 'GetCapabilities' normal response message is implemented as the
1311 *GetCapabilitiesResponse* message element.

1312 **Example**

```
1313 <?xml version="1.0" encoding="UTF-8"?>  
1314 <core:GetCapabilitiesResponse xmlns:core="urn:autoid:specification:interchange:Savant:core:xml:soap-rpc:1"  
1315 xmlns:rpc="http://www.w3.org/2003/05/soap-rpc" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
1316 xsi:schemaLocation="urn:autoid:specification:interchange:Savant:core:xml:soap-rpc:1  
1317 ../SchemaFiles/Interchange/Core.xsd">  
1318 <rpc:result>core:Capabilities</rpc:result>  
1319 <core:Capabilities>autoid.core</core:Capabilities>  
1320 <core:Capabilities>autoid.readerproxy</core:Capabilities>  
1321 <core:Capabilities>stuff.myreader.example.org</core:Capabilities>
```

1322 `</core:GetCapabilitiesResponse>`
1323

1324 **10.2.9.2.3 Error Response**

1325 **Functional Specification**

1326 For the functional specification for this message refer to section 9.1.2.3

1327 **XML Message Specification**

1328 The ‘GetCapabilities’ error response message is implemented as the
1329 **GetCapabilitiesError** message element.

1330 **Example**

```
1331 <?xml version="1.0" encoding="UTF-8"?>
1332 <core:GetCapabilitiesError xmlns:core="urn:autoid:specification:interchange:Savant:core:xml:soap-rpc:1"
1333 xmlns:err="urn:autoid:specification:domain:Savant:Error:xml:soap-rpc:1"
1334 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
1335 xsi:schemaLocation="urn:autoid:specification:interchange:Savant:core:xml:soap-rpc:1
1336 ../SchemaFiles/Interchange/Core.xsd">
1337   <err:ErrorCode>1</err:ErrorCode>
1338   <err:ErrorString>GET_CAPABILITIES_ERROR</err:ErrorString>
1339 </core:GetCapabilitiesError>
1340
```

1341 **10.2.9.3 Shutdown Message**

1342 **10.2.9.3.1 Request**

1343 **Functional Specification**

1344 For the functional specification for this message refer to section 9.1.3.1

1345 **XML Message Specification**

1346 The ‘Shutdown’ request message is implemented as the **Shutdown** message element.

1347 **Example**

```
1348 <?xml version="1.0" encoding="UTF-8"?>
1349 <core:Shutdown xmlns:core="urn:autoid:specification:interchange:Savant:core:xml:soap-rpc:1"
1350 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
1351 xsi:schemaLocation="urn:autoid:specification:interchange:Savant:core:xml:soap-rpc:1
1352 ../SchemaFiles/Interchange/Core.xsd"/>
1353
```

1354 **10.2.9.3.2 Normal Response**

1355 **Functional Specification**

1356 For the functional specification for this message refer to section 9.1.3.2

1357 **XML Message Specification**

1358 The ‘Shutdown’ normal response message is implemented as the **ShutdownResponse**
1359 message element.

1360 **Example**

```
1361 <?xml version="1.0" encoding="UTF-8"?>
1362 <core:ShutdownResponse xmlns:core="urn:autoid:specification:interchange:Savant:core:xml:soap-rpc:1"
1363 xmlns:rpc="http://www.w3.org/2003/05/soap-rpc" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
1364 xsi:schemaLocation="urn:autoid:specification:interchange:Savant:core:xml:soap-rpc:1
1365 ../SchemaFiles/Interchange/Core.xsd">
1366   <rpc:result>core:Shutdown</rpc:result>
1367   <core:Shutdown>true</core:Shutdown>
1368 </core:ShutdownResponse>
1369
```

1370 **10.2.9.3.3 Error Response**

1371 **Functional Specification**

1372 For the functional specification for this message refer to section 9.1.3.3

1373 **XML Message Specification**

1374 The ‘Shutdown’ error response message is implemented as the *ShutdownError* message
1375 element.

1376 **Example**

```
1377 <?xml version="1.0" encoding="UTF-8"?>
1378 <core:ShutdownError xmlns:core="urn:autoid:specification:interchange:Savant:core:xml:soap-rpc:1"
1379 xmlns:err="urn:autoid:specification:domain:Savant:Error:xml:soap-rpc:1"
1380 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
1381 xsi:schemaLocation="urn:autoid:specification:interchange:Savant:core:xml:soap-rpc:1
1382 ../SchemaFiles/Interchange/Core.xsd">
1383   <err:ErrorCode>1</err:ErrorCode>
1384   <err:ErrorString>SHUTDOWN_ERROR</err:ErrorString>
1385 </core:ShutdownError>
1386
```

1387 **10.2.9.4 ResetAll Message**

1388 **10.2.9.4.1 Request**

1389 **Functional Specification**

1390 For the functional specification for this message refer to section 9.1.4.1.

1391 **XML Message Specification**

1392 The ‘ResetAll’ request message is implemented as the *ResetAll* message element.

1393 **Example**

```
1394 <?xml version="1.0" encoding="UTF-8"?>
1395 <core:ResetAll xmlns:core="urn:autoid:specification:interchange:Savant:core:xml:soap-rpc:1"
1396 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
1397 xsi:schemaLocation="urn:autoid:specification:interchange:Savant:core:xml:soap-rpc:1
1398 ../SchemaFiles/Interchange/Core.xsd"/>
1399
```

1400 **10.2.9.4.2 Normal Response**

1401 **Functional Specification**

1402 For the functional specification for this message refer to section 9.1.4.2

1403 XML Message Specification

1404 The 'ResetAll' normal response message is implemented as the *ResetAllResponse*
1405 message element.

1406 Example

```
1407 <?xml version="1.0" encoding="UTF-8"?>
1408 <core:ResetAllResponse xmlns:core="urn:autoid:specification:interchange:Savant:core:xml:soap-rpc:1"
1409 xmlns:rpc="http://www.w3.org/2003/05/soap-rpc" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
1410 xsi:schemaLocation="urn:autoid:specification:interchange:Savant:core:xml:soap-rpc:1
1411 ../SchemaFiles/Interchange/Core.xsd">
1412   <rpc:result>core:SavantID</rpc:result>
1413   <core:ResetAll>true</core:ResetAll>
1414 </core:ResetAllResponse>
1415
```

1416 10.2.9.4.3 Error Response

1417 Functional Specification

1418 For the functional specification for this message refer to section 9.1.4.3

1419 XML Message Specification

1420 The 'ResetAll' error response message is implemented as the *ResetAllError* message
1421 element.

1422 Example

```
1423 <?xml version="1.0" encoding="UTF-8"?>
1424 <core:ResetAllError xmlns:core="urn:autoid:specification:interchange:Savant:core:xml:soap-rpc:1"
1425 xmlns:err="urn:autoid:specification:domain:Savant:Error:xml:soap-rpc:1"
1426 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
1427 xsi:schemaLocation="urn:autoid:specification:interchange:Savant:core:xml:soap-rpc:1
1428 ../SchemaFiles/Interchange/Core.xsd">
1429   <err:ErrorCode>1</err:ErrorCode>
1430   <err:ErrorString>RESET_ALL_ERROR</err:ErrorString>
1431 </core:ResetAllError>
```

1432 10.2.10 ReaderProxy Message Specification

1433 The AI messages supported by the `autoid.readerproxy` Standard Processing
1434 Module are defined in the 'ReaderProxy.xsd' schema. The following sections define the
1435 message parameters for each message.

1436 10.2.10.1 GetReaders Message

1437 10.2.10.1.1 Request

1438 Functional Specification

1439 For the functional specification for this message refer to section 9.2.1.1

1440 XML Message Specification

1441 The 'GetReaders' request message is implemented as the *GetReaders* message element.

1442 **Example**

```
1443 <?xml version="1.0" encoding="UTF-8"?>
1444 <rp:GetReaders xmlns:rp="urn:autoid:specification:interchange:Savant:readerproxy:xml:soap-rpc:1"
1445 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
1446 xsi:schemaLocation="urn:autoid:specification:interchange:Savant:readerproxy:xml:soap-rpc:1
1447 ../SchemaFiles/Interchange/ReaderProxy.xsd"/>
1448
```

1449 **10.2.10.1.2 Normal Response**

1450 **Functional Specification**

1451 For the functional specification for this message refer to section 9.2.1.2

1452 **XML Message Specification**

1453 The 'GetReaders' normal response message is implemented as the *GetReadersResponse*
1454 message element.

1455 **Example**

```
1456 <?xml version="1.0" encoding="UTF-8"?>
1457 <rp:GetReadersResponse xmlns:rp="urn:autoid:specification:interchange:Savant:readerproxy:xml:soap-rpc:1"
1458 xmlns:rpc="http://www.w3.org/2003/05/soap-rpc" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
1459 xsi:schemaLocation="urn:autoid:specification:interchange:Savant:readerproxy:xml:soap-rpc:1
1460 ../SchemaFiles/Interchange/ReaderProxy.xsd">
1461   <rpc:result>rp:Readers</rpc:result>
1462   <rp:Readers>urn:epc:1:2.24.500</rp:Readers>
1463   <rp:Readers>urn:epc:1:2.24.501</rp:Readers>
1464   <rp:Readers>urn:epc:1:2.24.502</rp:Readers>
1465 </rp:GetReadersResponse>
1466
```

1467 **10.2.10.1.3 Error Response**

1468 **Functional Specification**

1469 For the functional specification for this message refer to section 9.2.1.3

1470 **XML Message Specification**

1471 The 'GetReaders' error response message is implemented as the *GetReadersError*
1472 message element.

1473 **Example**

```
1474 <?xml version="1.0" encoding="UTF-8"?>
1475 <rp:GetReadersError xmlns:rp="urn:autoid:specification:interchange:Savant:readerproxy:xml:soap-rpc:1"
1476 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
1477 xmlns:err="urn:autoid:specification:domain:Savant:Error:xml:soap-rpc:1"
1478 xsi:schemaLocation="urn:autoid:specification:interchange:Savant:readerproxy:xml:soap-rpc:1
1479 ../SchemaFiles/Interchange/ReaderProxy.xsd">
1480   <err:ErrorCode>1</err:ErrorCode>
1481   <err:ErrorString>GET_READERS_ERROR</err:ErrorString>
1482 </rp:GetReadersError>
1483
```

1484 **10.2.10.2 RunCommand Message**

1485 **10.2.10.2.1 Request**

1486 **Functional Specification**

1487 For the functional specification for this message refer to section 9.2.2.1

1488 **XML Message Specification**

1489 The 'RunCommand' request message is implemented as the **RunCommand** message
1490 element.

1491 **Example**

```
1492 <?xml version="1.0" encoding="UTF-8"?>
1493 <rp:RunCommand xmlns:rp="urn:autoid:specification:interchange:Savant:readerproxy:xml:soap-rpc:1"
1494 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
1495 xsi:schemaLocation="urn:autoid:specification:interchange:Savant:readerproxy:xml:soap-rpc:1
1496 ../SchemaFiles/Interchange/ReaderProxy.xsd">
1497   <rp:ReaderID>urn:epc:1:2.24.500</rp:ReaderID>
1498   <rp:Command>READER_COMMAND</rp:Command>
1499   <rp:Arguments>ARG1,ARG2,ARG3</rp:Arguments>
1500 </rp:RunCommand>
1501
```

1502 **10.2.10.2.2 Normal Response**

1503 **Functional Specification**

1504 For the functional specification for this message refer to section 9.2.2.2

1505 **XML Message Specification**

1506 The 'RunCommand' normal response message is implemented as the
1507 **RunCommandResponse** message element.

1508 **Example**

```
1509 <?xml version="1.0" encoding="UTF-8"?>
1510 <rp:RunCommandResponse xmlns:rp="urn:autoid:specification:interchange:Savant:readerproxy:xml:soap-rpc:1"
1511 xmlns:rpc="http://www.w3.org/2003/05/soap-rpc" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
1512 xsi:schemaLocation="urn:autoid:specification:interchange:Savant:readerproxy:xml:soap-rpc:1
1513 ../SchemaFiles/Interchange/ReaderProxy.xsd">
1514   <rpc:result>rp:RunCommand</rpc:result>
1515   <rp:RunCommand>SUCCESS</rp:RunCommand>
1516 </rp:RunCommandResponse>
1517
```

1518 **10.2.10.2.3 Error Response**

1519 **Functional Specification**

1520 For the functional specification for this message refer to section 9.2.2.3

1521 **XML Message Specification**

1522 The 'RunCommand' error response message is implemented as the **RunCommandError**
1523 message element.

1524 **Example**

```
1525 <?xml version="1.0" encoding="UTF-8"?>
1526 <rp:RunCommandError xmlns:rp="urn:autoid:specification:interchange:Savant:readerproxy.xml:soap-rpc:1"
1527 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
1528 xmlns:err="urn:autoid:specification:domain:Savant:Error.xml:soap-rpc:1"
1529 xsi:schemaLocation="urn:autoid:specification:interchange:Savant:readerproxy.xml:soap-rpc:1
1530 ../SchemaFiles/Interchange/ReaderProxy.xsd">
1531   <err:ErrorCode>1</err:ErrorCode>
1532   <err:ErrorString>RUN_COMMAND_ERROR</err:ErrorString>
1533 </rp:RunCommandError>
```

1534 **10.2.11 Message Format Constraints**

1535 This section provides specific message formatting constraints for the AI messages
1536 described in the previous sections:

- 1537 1. The value of the HTTP Content-type header MUST be chosen as
1538 "application/soap+xml".
- 1539 2. Following the recommendations outlined in section 4.1.3 “Web Architecture
1540 Compatible SOAP Usage” of [SOAP Part 0], SOAP RPC message exchanges for
1541 AI messages MUST be implemented using the SOAP Request-Response message
1542 exchange pattern as specified in section 6.2 of [SOAP Part 2]. The HTTP POST
1543 method MUST be used for the request with the AI message in the request body. In
1544 other words the following SOAP features MUST be used with the [HTTP 1.1]
1545 binding exchanging AI messages:
 - 1546 a. The SOAP Message Exchange Pattern (MEP) feature with the property
1547 name as identified by the URI
1548 “http://www.w3.org/2003/05/soap/bindingFramework/ExchangeContext/E
1549 xchangePatternName” having the property value
1550 "http://www.w3.org/2003/05/soap/mep/request-response/" for the SOAP
1551 Request-Response MEP
 - 1552 b. The SOAP Web Method feature with the property name as identified by
1553 the URI “http://www.w3.org/2003/05/soap/features/web-method/” and
1554 having the property value “POST” for the HTTP POST method
- 1555 3. The default SOAP encoding style MUST be used to encode all messages. As such
1556 the value of the SOAP ‘encodingStyle’ attribute in the SOAP message MUST be
1557 set to the following URI
1558 ‘http://www.w3.org/2003/05/soap-encoding’
- 1559 4. In accordance with section 4.2.2 “RPC Response” of [SOAP Part 2] the value of
1560 the SOAP RPC ‘result’ element of the RPC Response message MUST be the
1561 XML Qualified Name of the “return” parameter. This rule MUST be applied to all
1562 Normal Response messages.

1563 Error Response messages as specified in the previous section are application error
1564 messages generated by Standard Processing Modules. All Error Response messages
1565 MUST be formatted as a SOAP fault message and included in the optional SOAP ‘detail’
1566 element as specified under section 5.4 “SOAP Fault” of [SOAP Part 1].
1567

1568

1569 **10.2.12 SOAP Terminology**

1570

1571 **SOAP**

1572 The formal set of conventions governing the format and processing rules of a
1573 SOAP message. These conventions include the interactions among SOAP nodes
1574 generating and accepting SOAP messages for the purpose of exchanging
1575 information along a SOAP message path.

1576 **SOAP feature**

1577 An extension of the SOAP messaging framework. Examples of features include
1578 "reliability", "security", "correlation", "routing", and "Message Exchange
1579 Patterns" (MEPs).

1580 **SOAP node**

1581 The embodiment of the processing logic necessary to transmit, receive, process
1582 and/or relay a SOAP message, according to the set of conventions defined by this
1583 recommendation. A SOAP node is responsible for enforcing the rules that govern
1584 the exchange of SOAP messages. It accesses the services provided by the
1585 underlying protocols through one or more SOAP bindings.

1586 **SOAP sender**

1587 A SOAP node that transmits a SOAP message.

1588 **SOAP receiver**

1589 A SOAP node that accepts a SOAP message.

1590 **SOAP message**

1591 The basic unit of communication between SOAP nodes.

1592 **SOAP message exchange pattern (MEP)**

1593 A template for the exchange of SOAP messages between SOAP nodes enabled by
1594 one or more underlying SOAP protocol bindings. A SOAP MEP is an example of
1595 a SOAP feature

1596 **SOAP binding**

1597 The formal set of rules for carrying a SOAP message within or on top of another
1598 protocol (underlying protocol) for the purpose of exchange. Examples of SOAP
1599 bindings include carrying a SOAP message within an HTTP entity-body, or over a
1600 TCP stream.

1601 **11 Abbreviations**

1602 This section is for clarification of terms used throughout the document.

AI	AI
HTTP	Hyper Text Transfer Protocol
MTB	Message Transport Binding
MEP	Message Exchange Pattern
PML	Physical Markup Language
RPC	Remote Procedure Call
SOAP	Simple Object Access Protocol
TCP	Transmission Control Protocol
URI	Uniform Resource Identifier
URL	Uniform Resource Location
URN	Uniform Resource Name
XML	eXtensible Markup Language

1603

1604 **12 References**

1605

1606 [SOAP Part 1] W3C Recommendation "[SOAP 1.2 part 1: Messaging Framework](http://www.w3.org/TR/2003/REC-soap12-part1-20030624/)",
 1607 Martin Gudgin, Marc Hadley, Jean-Jacques Moreau, Henrik Frystyk Nielsen, 24 June
 1608 2003 (See <http://www.w3.org/TR/2003/REC-soap12-part1-20030624/>.)

1609 [SOAP Part 2] W3C Recommendation "[SOAP 1.2 part 2: Adjuncts](http://www.w3.org/TR/2003/REC-soap12-part2-20030624/)", Martin Gudgin,
 1610 Marc Hadley, Jean-Jacques Moreau, Henrik Frystyk Nielsen, 24 June 2003 (See
 1611 <http://www.w3.org/TR/2003/REC-soap12-part2-20030624/>.)

1612 [HTTP 1.1] IETF "[RFC 2616: Hypertext Transfer Protocol -- HTTP/1.1](http://www.ietf.org/rfc/rfc2616.txt)", R. Fielding, J.
 1613 Gettys, J. C. Mogul, H. Frystyk, T. Berners-Lee, January 1997. (See
 1614 <http://www.ietf.org/rfc/rfc2616.txt>.)

1615 [PML Core] Auto-ID "PML Core Specification, Version 1.0", Christian Floerkemeier,
 1616 Dipan Anarkat, Ted Osinski, Mark Harrison, September 2003.

1617 [RFC2119] S. Bradner, "Key words for use in RFCs to Indicate Requirement Levels,"
 1618 IETF RFC 2119, <http://www.ietf.org/rfc/rfc2119.txt>.

1619 [ReaderProtocol1.0] Auto-ID "Reader Protocol Specification 1.0," Auto-ID Working
 1620 Draft, <http://develop.autoidcenter.org/TR/reader-protocol-1.0.doc>.

1621 [JLS2] B. Joy, J. Gosling, G. Steele, and G. Bracha, "The Java Language Specification
 1622 (2nd Edition)," Addison-Wesley, 2000,
 1623 http://java.sun.com/docs/books/jls/second_edition/html/.

- 1624 [XML-RPC] Dave Winer, "The XML-RPC Specification" UserLand Software, 2003
 1625 <http://www.xmlrpc.com/spec>
 1626 [RFC2616] R. Fielding et al, "Hypertext Transfer Protocol -- HTTP/1.1," IETF RFC
 1627 2616, <http://www.ietf.org/rfc/rfc2616.txt>
 1628 [SOAP Part 0] W3C Proposed Recommendation "SOAP Version 1.2 part 0: Primer",
 1629 Nilo Mitra, 24 June 2003, <http://www.w3.org/TR/2003/REC-soap12-part0-20030624>

1630 13 APPENDIX A: SOAP RPC Schemas

1631 13.1 Core.xsd

```

1632 <?xml version="1.0"?>
1633 <schema targetNamespace="urn:autoid:specification:interchange:Savant:core:xml:soap-rpc:1"
1634 xmlns="http://www.w3.org/2001/XMLSchema" xmlns:autoid="http://www.autoidcenter.org/2003/xml"
1635 xmlns:core="urn:autoid:specification:interchange:Savant:core:xml:soap-rpc:1"
1636 xmlns:err="urn:autoid:specification:domain:Savant:Error:xml:soap-rpc:1"
1637 xmlns:rpc="http://www.w3.org/2003/05/soap-rpc" elementFormDefault="qualified"
1638 attributeFormDefault="unqualified" version="1.0">
1639   <annotation>
1640     <documentation>
1641       <autoid:copyright>Copyright ©2003 Auto-ID Center, All Rights Reserved.</autoid:copyright>
1642       <autoid:disclaimer>Auto-ID Center, its members, officers, directors, employees, or agents shall not be
1643 liable for any injury, loss, damages, financial or otherwise, arising from, related to, or caused by the use of this
1644 document. The use of said document shall constitute your express consent to the foregoing
1645 exculpation.</autoid:disclaimer>
1646       <autoid:program>Auto-ID version 1.0</autoid:program>
1647       <autoid:purpose>Savant Specification version 1.0</autoid:purpose>
1648     </documentation>
1649   </annotation>
1650   <import namespace="urn:autoid:specification:domain:Savant:Error:xml:soap-rpc:1"
1651 schemaLocation="../Domain/Savant/Error.xsd"/>
1652   <import namespace="http://www.w3.org/2003/05/soap-rpc" schemaLocation="../External/soap-rpc.xsd"/>
1653   <element name="GetSavantID" type="core:GetSavantIDType"/>
1654   <complexType name="GetSavantIDType">
1655     <annotation>
1656       <documentation>
1657         <autoid:definition>GetSavantID request message. Interrogate a Savant for its unique numeric
1658 identifier</autoid:definition>
1659       </documentation>
1660     </annotation>
1661     <sequence/>
1662   </complexType>
1663   <element name="GetSavantIDResponse" type="core:GetSavantIDResponseType"/>
1664   <complexType name="GetSavantIDResponseType">
1665     <annotation>
1666       <documentation>
1667         <autoid:definition>GetSavantID response message</autoid:definition>
1668       </documentation>
1669     </annotation>
1670     <sequence>
1671       <element ref="rpc:result">
1672         <annotation>
1673           <documentation>
1674             <autoid:definition>SOAP RPC result element used to identify the return value of an SOAP
1675 RPC. MUST contain the XML QName of the 'SavantID' element</autoid:definition>
1676           </documentation>
1677         </annotation>
1678       </element>
1679       <element name="SavantID" type="string">

```

```

1680         <annotation>
1681             <documentation>
1682                 <autoid:definition>A unique identification string for the Savant.</autoid:definition>
1683             </documentation>
1684         </annotation>
1685     </element>
1686 </sequence>
1687 </complexType>
1688 <element name="GetSavantIDError" type="err:ErrorType"/>
1689 <element name="GetCapabilities" type="core:GetCapabilitiesType"/>
1690 <complexType name="GetCapabilitiesType">
1691     <annotation>
1692         <documentation>
1693             <autoid:definition>GetCapabilities request message. Interrogate a Savant for a list of all the
1694 configured processing modules</autoid:definition>
1695         </documentation>
1696     </annotation>
1697 </sequence>
1698 </complexType>
1699 <element name="GetCapabilitiesResponse" type="core:GetCapabilitiesResponseType"/>
1700 <complexType name="GetCapabilitiesResponseType">
1701     <annotation>
1702         <documentation>
1703             <autoid:definition>GetCapabilities response message</autoid:definition>
1704         </documentation>
1705     </annotation>
1706 <sequence>
1707     <element ref="rpc:result">
1708         <annotation>
1709             <documentation>
1710                 <autoid:definition>SOAP RPC result element used to identify the return value of an SOAP
1711 RPC. MUST contain the XML QName of the 'Capabilities' element</autoid:definition>
1712             </documentation>
1713         </annotation>
1714     </element>
1715     <element name="Capabilities" type="string" minOccurs="1" maxOccurs="unbounded">
1716         <annotation>
1717             <documentation>
1718                 <autoid:definition>Names of Processing modules configured in the
1719 Savant</autoid:definition>
1720             </documentation>
1721         </annotation>
1722     </element>
1723 </sequence>
1724 </complexType>
1725 <element name="GetCapabilitiesError" type="err:ErrorType"/>
1726 <element name="Shutdown" type="core:ShutdownType"/>
1727 <complexType name="ShutdownType">
1728     <annotation>
1729         <documentation>
1730             <autoid:definition>Shutdown request message. Notify all subsystems to shutdown and then the
1731 core processing module will also shutdown</autoid:definition>
1732         </documentation>
1733     </annotation>
1734 </sequence>
1735 </complexType>
1736 <element name="ShutdownResponse" type="core:ShutdownResponseType"/>
1737 <complexType name="ShutdownResponseType">
1738     <annotation>
1739         <documentation>
1740             <autoid:definition>Shutdown response message</autoid:definition>
1741         </documentation>
1742     </annotation>
1743 </sequence>
1744     <element ref="rpc:result">

```

```

1745         <annotation>
1746             <documentation>
1747                 <autoid:definition>SOAP RPC result element used to identify the return value of a SOAP
1748 RPC. MUST contain the XML QName of the 'Shutdown' element</autoid:definition>
1749             </documentation>
1750         </annotation>
1751     </element>
1752     <element name="Shutdown" type="boolean">
1753         <annotation>
1754             <documentation>
1755                 <autoid:definition>True if all subsystems were notified false otherwise</autoid:definition>
1756             </documentation>
1757         </annotation>
1758     </element>
1759 </sequence>
1760 </complexType>
1761 <element name="ShutdownError" type="err:ErrorType"/>
1762 <element name="ResetAll" type="core:ResetAllType"/>
1763 <complexType name="ResetAllType">
1764     <annotation>
1765         <documentation>
1766             <autoid:definition>ResetAll request message. Inform the Savant to notify all subsystems to reset to
1767 their respective last known configuration and then the core processing module will also reset</autoid:definition>
1768         </documentation>
1769     </annotation>
1770 </sequence>
1771 </complexType>
1772 <element name="ResetAllResponse" type="core:ResetAllResponseType"/>
1773 <complexType name="ResetAllResponseType">
1774     <annotation>
1775         <documentation>
1776             <autoid:definition>ResetAll response message</autoid:definition>
1777         </documentation>
1778     </annotation>
1779 </sequence>
1780     <element ref="rpc:result">
1781         <annotation>
1782             <documentation>
1783                 <autoid:definition>SOAP RPC result element used to identify the return value of a SOAP
1784 RPC. MUST contain the XML QName of the 'ResetAll' element</autoid:definition>
1785             </documentation>
1786         </annotation>
1787     </element>
1788     <element name="ResetAll" type="boolean">
1789         <annotation>
1790             <documentation>
1791                 <autoid:definition>True if all subsystems were notified false otherwise</autoid:definition>
1792             </documentation>
1793         </annotation>
1794     </element>
1795 </sequence>
1796 </complexType>
1797 <element name="ResetAllError" type="err:ErrorType"/>
1798 </schema>
1799

```

1800 13.2 ReaderProxy.xsd

```

1801 <?xml version="1.0"?>
1802 <schema targetNamespace="urn:autoid:specification:interchange:Savant:readerproxy.xml:soap-rpc:1"
1803 xmlns="http://www.w3.org/2001/XMLSchema" xmlns:autoid="http://www.autoidcenter.org/2003/xml"
1804 xmlns:rpx="urn:autoid:specification:interchange:Savant:readerproxy.xml:soap-rpc:1"
1805 xmlns:err="urn:autoid:specification:domain:Savant:Error.xml:soap-rpc:1"

```

```

1806 xmlns:rpc="http://www.w3.org/2003/05/soap-rpc" elementFormDefault="qualified"
1807 attributeFormDefault="unqualified" version="1.0">
1808   <annotation>
1809     <documentation>
1810       <autoid:copyright>Copyright ©2003 Auto-ID Center, All Rights Reserved.</autoid:copyright>
1811       <autoid:disclaimer>Auto-ID Center, its members, officers, directors, employees, or agents shall not be
1812 liable for any injury, loss, damages, financial or otherwise, arising from, related to, or caused by the use of this
1813 document. The use of said document shall constitute your express consent to the foregoing
1814 exculpation.</autoid:disclaimer>
1815       <autoid:program>Auto-ID version 1.0</autoid:program>
1816       <autoid:purpose>Savant Specification version 1.0</autoid:purpose>
1817     </documentation>
1818   </annotation>
1819   <import namespace="http://www.w3.org/2003/05/soap-rpc" schemaLocation="../External/soap-rpc.xsd"/>
1820   <import namespace="urn:autoid:specification:domain:Savant:Error:xml:soap-rpc:1"
1821 schemaLocation="../Domain/Savant/Error.xsd"/>
1822   <element name="GetReaders" type="rpx:GetReadersType"/>
1823   <complexType name="GetReadersType">
1824     <annotation>
1825       <documentation>
1826         <autoid:definition>GetReaders request message. Inquire the Savant for all readers it is configured
1827 to communicate</autoid:definition>
1828       </documentation>
1829     </annotation>
1830     <sequence/>
1831   </complexType>
1832   <element name="GetReadersResponse" type="rpx:GetReadersResponseType"/>
1833   <complexType name="GetReadersResponseType">
1834     <annotation>
1835       <documentation>
1836         <autoid:definition>GetReaders response message</autoid:definition>
1837       </documentation>
1838     </annotation>
1839     <sequence>
1840       <element ref="rpc:result">
1841         <annotation>
1842           <documentation>
1843             <autoid:definition>SOAP RPC result element used to identify the return value of an SOAP
1844 RPC. MUST contain the XML QName of the 'Readers' element</autoid:definition>
1845           </documentation>
1846         </annotation>
1847       </element>
1848       <element name="Readers" type="string" minOccurs="0" maxOccurs="unbounded">
1849         <annotation>
1850           <documentation>
1851             <autoid:definition>A unique identification string for the Reader</autoid:definition>
1852           </documentation>
1853         </annotation>
1854       </element>
1855     </sequence>
1856   </complexType>
1857   <element name="GetReadersError" type="err:ErrorType"/>
1858   <element name="RunCommand" type="rpx:RunCommandType"/>
1859   <complexType name="RunCommandType">
1860     <annotation>
1861       <documentation>
1862         <autoid:definition>GetCapabilities request message. RunCommand message to the Savant as a
1863 standard pass through mechanism for applications to take advantage of all commands any RFID reader may
1864 expose</autoid:definition>
1865       </documentation>
1866     </annotation>
1867     <sequence>
1868       <element name="ReaderID" type="string">
1869         <annotation>
1870           <documentation>

```

```

1871         <autoid:definition>The Reader ID where the command is destined</autoid:definition>
1872     </documentation>
1873 </annotation>
1874 </element>
1875 <element name="Command" type="string">
1876     <annotation>
1877         <documentation>
1878             <autoid:definition>The name of the command.</autoid:definition>
1879         </documentation>
1880     </annotation>
1881 </element>
1882 <element name="Arguments" type="string">
1883     <annotation>
1884         <documentation>
1885             <autoid:definition>The set of arguments needed to execute the command on the reader.
1886 </autoid:definition>
1887         </documentation>
1888     </annotation>
1889 </element>
1890 </sequence>
1891 </complexType>
1892 <element name="RunCommandResponse" type="rpx:RunCommandResponseType"/>
1893 <complexType name="RunCommandResponseType">
1894     <annotation>
1895         <documentation>
1896             <autoid:definition>RunCommand response message</autoid:definition>
1897         </documentation>
1898     </annotation>
1899 <sequence>
1900     <element ref="rpc:result">
1901         <annotation>
1902             <documentation>
1903                 <autoid:definition>SOAP RPC result element used to identify the return value of an SOAP
1904 RPC. MUST contain the XML QName of the 'RunCommand' element</autoid:definition>
1905             </documentation>
1906         </annotation>
1907     </element>
1908     <element name="RunCommand" type="string" minOccurs="0">
1909         <annotation>
1910             <documentation>
1911                 <autoid:definition>The output from the reader based on execution of the command.
1912 </autoid:definition>
1913             </documentation>
1914         </annotation>
1915     </element>
1916 </sequence>
1917 </complexType>
1918 <element name="RunCommandError" type="err:ErrorType"/>
1919 </schema>

```

1920 13.3 Error.xsd

```

1921 <?xml version="1.0"?>
1922 <schema targetNamespace="urn:autoid:specification:domain:Savant:Error:xml:soap-rpc:1"
1923 xmlns="http://www.w3.org/2001/XMLSchema" xmlns:autoid="http://www.autoidcenter.org/2003/xml"
1924 xmlns:err="urn:autoid:specification:domain:Savant:Error:xml:soap-rpc:1" elementFormDefault="qualified"
1925 attributeFormDefault="unqualified" version="1.0">
1926     <annotation>
1927         <documentation>
1928             <autoid:copyright>Copyright ©2003 Auto-ID Center, All Rights Reserved.</autoid:copyright>
1929             <autoid:disclaimer>Auto-ID Center, its members, officers, directors, employees, or agents shall not be
1930 liable for any injury, loss, damages, financial or otherwise, arising from, related to, or caused by the use of this
1931 document. The use of said document shall constitute your express consent to the foregoing
1932 exculpation.</autoid:disclaimer>
1933             <autoid:program>Auto-ID version 1.0</autoid:program>

```



```

1934         <autoid:purpose>Savant Specification version 1.0</autoid:purpose>
1935     </documentation>
1936 </annotation>
1937 <complexType name="ErrorType">
1938     <annotation>
1939         <documentation>
1940             <autoid:definition>Error object for Savant AI messages</autoid:definition>
1941         </documentation>
1942     </annotation>
1943     <sequence>
1944         <element name="ErrorCode" type="integer">
1945             <annotation>
1946                 <documentation>
1947                     <autoid:definition>A numeric code for the error</autoid:definition>
1948                 </documentation>
1949             </annotation>
1950         </element>
1951         <element name="ErrorString" type="string">
1952             <annotation>
1953                 <documentation>
1954                     <autoid:definition>Description of the error</autoid:definition>
1955                 </documentation>
1956             </annotation>
1957         </element>
1958     </sequence>
1959 </complexType>
1960 </schema>
1961

```

1962 14 APPENDIX B: SOAP RPC Examples

1963 14.1 Control Channel Request – GetSavantID

1964

1965 POST /urn:autoid:specification:interchange:Savant:core:xml:service:1 HTTP/1.1

1966 Host: mycompany.example.org

1967 Content-Type: application/soap+xml; charset="utf-8"

1968 Content-Length: nnnn

1969

1970 <?xml version='1.0'?>

1971 <env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope" >

1972 <env:Body>

1973 <core:GetSavantID env:encodingStyle="http://www.w3.org/2003/05/soap-encoding"

1974 xmlns:core="urn:autoid:specification:interchange:Savant:core:xml:soap-rpc:1"/>

1975 </env:Body>

1976 </env:Envelope>

1977 14.2 Control Channel Response - GetSavantIDResponse

1978

1979 HTTP/1.1 200 OK

1980 Connection: close

1981 Content-Type: application/soap+xml; charset="utf-8"

1982 Content-Length: nnnn

1983 Date: Fri, 17 Jul 2003 19:55:08 GMT
1984
1985 <?xml version='1.0'?>
1986 <env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope" >
1987 <env:Body>
1988 <core:GetSavantIDResponse env:encodingStyle="http://www.w3.org/2003/05/soap-encoding"
1989 xmlns:core="urn:autoid:specification:interchange:Savant:core:xml:soap-rpc:1"
1990 xmlns:rpc="http://www.w3.org/2003/05/soap-rpc">
1991 <rpc:result>core:SavantID</rpc:result>
1992 <core:SavantID>urn:epc:1:2.24.400</core:SavantID>
1993 </core:GetSavantIDResponse>
1994 </env:Body>
1995 </env:Envelope>

1996 **15 Appendix C Future Standard Processing Modules**
1997 **FPM 1.1** Time Synchronization module