Diploma Thesis

XML-centered Information Handling in Technical Documentation

Michael Ebner

Graz, June 2005

Assessor: o. Univ.-Prof. Dr. Dr. h.c. mult. Hermann Maurer
Institute for Information Systems and Computer Media
Graz, University of Technology

Supervisors: Dipl.-Ing. (FH) Ingo Schreiber
Munich, Infineon Technologies AG

Univ.-Ass. Dipl.-Ing. Dr.techn. Harald Krottmaier
Institute for Information Systems and Computer Media
Graz, University of Technology

Diplomarbeit

XML zentriertes information handling in der Technischen Dokumentation

Michael Ebner

Graz, Juni 2005

Begutachter: o. Univ.-Prof. Dr. Dr. h.c. mult. Hermann Maurer
Institut für Informationssysteme und Computer Medien
Graz, Technische Universität

Betreuer: Dipl.-Ing. (FH) Ingo Schreiber
München, Infineon Technologies AG

Univ.-Ass. Dipl.-Ing. Dr.techn. Harald Krottmaier
Institut für Informationssysteme und Computer Medien
Graz, Technische Universität

Abstract

This diploma thesis has been developed at Infineon Technologies AG in Munich, Germany. The aim was to examine the existing systems used for technical documentation and to build a showcase emphasizing the work with documents in an XML-based environment. Implementing the idea of single sourcing and combining it with the possibilities XML provides, it shows that it is now possible to take advantage of the benefits that evolve from these two technologies. All this has been realized in a web application that supports various forms of online XML editing and also provides server-side validation and transformation. In addition, a rendering engine has been included that is able to turn XML data into printable PDF documents.

Abstract in German

Diese Diplomarbeit wurde bei der Infineon Technologies AG in München entwickelt. Das Ziel war es, die dort zur technischen Dokumentation verwendeten Systeme zu untersuchen und aufbauend auf den Ergebnissen einen Showcase zu entwickeln, der die technische Dokumentation in einer XML basierten Umgebung unterstützt. Es wurde die Idee des sogenannten 'Single Sourcing' mit den Möglichkeiten die XML bietet kombiniert um die Vorteile dieser beiden Technologien zu nutzen. Der Showcase wurde in einer Web Applikation umgesetzt, die verschiedene Möglichkeiten der online XML Editierung bietet und ebenfalls eine serverseitige Validierung und Transformation unterstützt. Desweiteren wurde ein Programm eingebunden, das es ermöglicht XML Daten in ein PDF Dokument zu konvertieren.

Acknowledgements

This diploma thesis was conducted at Infineon Technologies AG in Munich, the scientific support was done by the Institute for Information Systems and Computer Media at Graz University of Technology. Grateful acknowledgements are made to the many people whose help in preparing this thesis was invaluable. In particular, my technical supervisor, Ingo Schreiber and my scientific supervisor, Harald Krottmaier. I would also like to thank my parents for their assistance and for making my studies possible, and my girlfriend Lisa because her encouragement was a great source of inspiration throughout my studies.

Table of Contents

Chapter 1

Introduction

In various articles published on the Internet as well as in IT and business journals, the *eXtensible Markup Language* (XML) has been declared to be the standard that will revolutionize the Internet and electronic business. The interest in getting a detailed view on this new format, the technologies associated with it, and its use in electronic business, led me to chose this as the subject of my thesis.

A project launched at Infineon Technologies AG is working on the conception and implementation of an XML-based technical documentation and its publication via multiple channels. Being part of the company's activities in the area of technical communication, this interesting project has offered to me the possibilities to gain experience in implementing XML technologies and to get to know the impacts of their use on a company's technical documentation processes. The chief objective of this thesis was the implementation of a showcase, providing an XML-based editing environment for technical documentation. Along with the implementation of this system, practical experiences in using XML technologies should be gathered.

My proceedings at work started with examining a well-defined goal for this thesis, because technical documentation provides a huge area of possibilities. After analyzing the current situation I decided to develop a XML enabled showcase for technical documentation. I became acquainted with Adobe FrameMaker and how this software is used today at Infineon Technologies. Then I started to work out the advantages and disadvantages of the current use of FrameMaker in combination with its XML capabilities. Afterwards I started to design and implement a showcase that enables technical documentation on a pure XML basis. The Infineon internal process of finding an appropriate *Enterprise Content Management System* (ECMS) is not completed yet. Therefore the showcases' editing environment uses a file system based solution, but can be easily connected to an XML-based Content Management System in the future.

The chapters and their content have been organized according to the steps that were taken during the work. Chapter 2 contains an introduction to XML as well as the explanations on other technologies used during the work on my thesis. Chapter 3 gives an overview of the situation I met when I started my work at Infineon Technologies. Chapter 4 contains a look-out to what is possible with an XML-based technical documentation. Additionally a general insight to what open standards - of which XML is one representative - may be used for in the future will be given. Chapter 5 explains what the showcase I implemented is capable of. Chapter 6 gives a summary and a conclusion on the treated subjects.

Chapter 2

Used Technologies

This chapter contains a short introduction to all technologies I used during the development of my Diploma Thesis. If the reader is already familiar with XML and its related technologies the next four sections can be skipped and it should be continued with section 2.5.

2.1 XML

The *eXtensible Markup Language* (XML) is a *World Wide Web Consortium* (W3C) standard for document markup [1]. It is said to be the descendant of *Hypertext Markup Language* (HTML), but in fact it is a descendant and a simplification to the much more complex *Standard Generalized Markup Language* (SGML), which was also a recommendation of W3C back in 1986. [2]

XML defines a generic syntax used to mark up data with simple, human-readable tags. It provides a standard format for computer documents that is flexible enough to be customized for areas as diverse as electronic data interchange, vector graphics, object serialization to remote procedure calls and even more.

XML is a meta-markup language for text documents. Data in XML is included as strings of text. The data is surrounded by text markup that describes the data. XML's basic unit of data and markup is called an *element*. The XML specification defines the exact syntax this markup must follow: how elements are delimited by tags, what a tag must look like, what names are acceptable for elements, where attributes are placed and so forth. The markup in an XML document looks a lot like the markup in an HTML document, but there are some crucial differences. Most important, XML is a *meta-markup language*. That means it does not have a fixed set of tags and elements. Any attempt to create a finite set of such tags is doomed to failure. Instead, XML allows developers and writers to invent elements they need. As a reminder, the X in XML stands for eXtensible. Extensible means that the language can be extended and adapted to meet many different needs. For instance: A mathematician would be able to describe any formula in XML. Chemists can use elements that describe molecules, atoms, reactions, and other items encountered in chemistry and technical writers can use the meta language for structuring their documents.

XML fulfills the following definitions of a meta-markup language:

- Flexible Data Structure: The language is not based on a restricted set of tags and elements. It can be extended to satisfy the needs of many different purposes and it can easily be converted to many different formats.
- Platform Independence: Data defined with XML can be used, edited and processed on any platform, without taking care of system specific characteristics.
- Textorientation: Due to the platform independent aspect, a storage as binary data is not possible. Therefore the information is stored in plain text and the markup is distinguished through special characters.

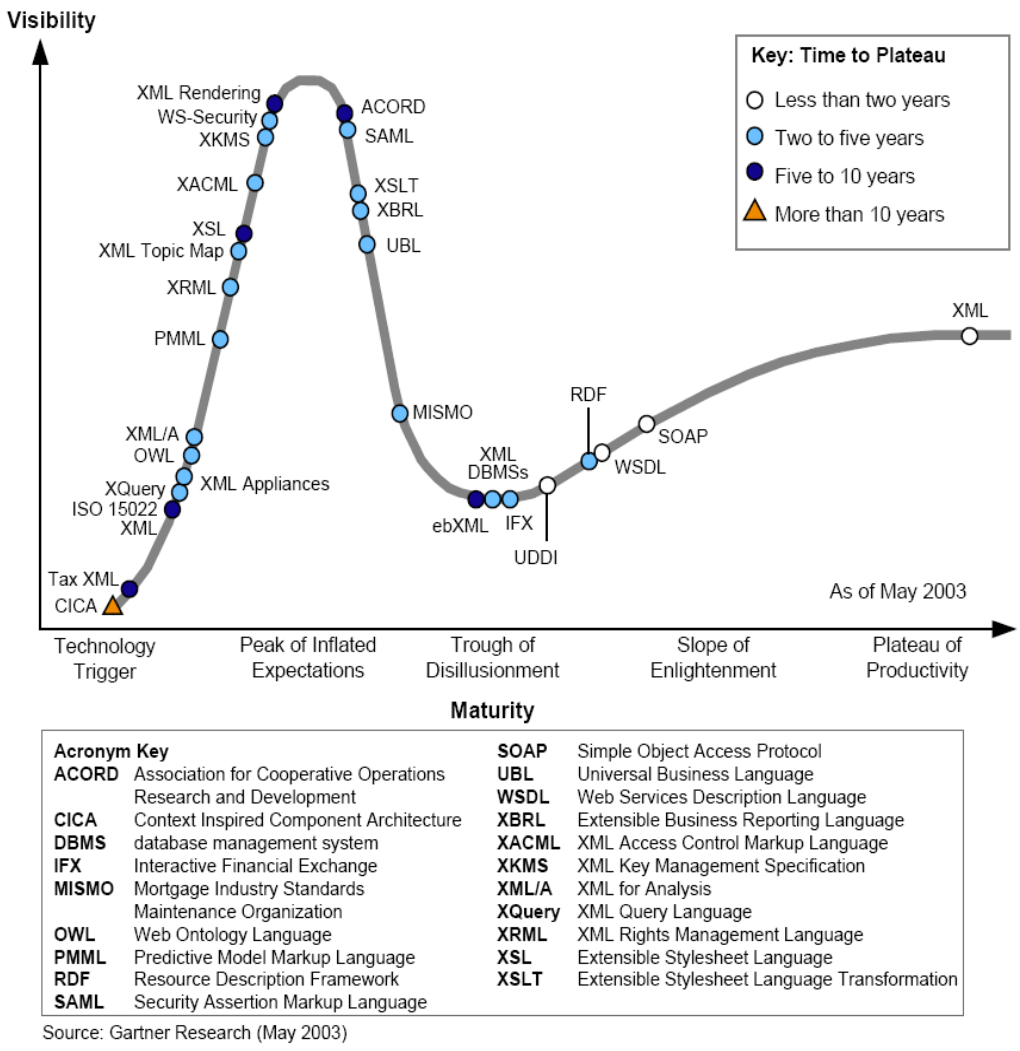
Although XML is flexible in the elements it allows to be defined, it is quite strict in many other aspects. It provides a grammar for XML documents that defines where tags may be placed, what they must look like, which element names are legal, how attributes are attached to elements, and so forth. This grammar is specific enough to allow the development of XML parsers that can read any XML document. Documents that satisfy this grammar are said to be well-formed. Documents that are not well-formed are not allowed. Similar to a C program that contains syntax errors, XML processors will reject documents that contain well-formedness errors. [4]

The markup permitted in an XML document can also be restricted with the use of a so-called *schema*, which allows not only to check for well-formedness, but also if a document is valid against that *schema*. This issue will be covered in detail in section 2.2.

To get a first impression of what a XML document could look like, a simple example is listed below. These few lines loosely describe the structure of a book. Nonetheless this is a well-formed document and XML parsers can read and understand it. In addition this sample document will also be the basis for all examples that will be treated throughout the next sections.

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

<Book>

<Title> 16-Bit CMOS Microcontroller </Title>
<Author> Johann Beispiel </Author>
<PublishingDate> 2002/05/20 </PublishingDate>

<Chapter ID = "c100" >
      <ChapterTitle> Introduction </ChapterTitle>
             <Paragraph ID = "p100" >
                    The 16-Bit CMOS Microcontroller is...
             </Paragraph>
</Chapter>

</Book>
```

Concluding it can be said that, as the development of XML started in 1996, it has been a long way until today. Nevertheless XML has survived the hype cycle - shown in 2.1 - as predicted by the Gartner Research Group [3]. However, only the future can tell if other XML related technologies will also become as popular and sophisticated as XML is today. Combining all these technologies and their usage, I think we are standing at the start of the 'Slope of Enlightenment' today when using XML for technical documentation.



Acronym Key			
ACORD	Association for Cooperative Operations Research and Development	**SOAP**	Simple Object Access Protocol
CICA	Context Inspired Component Architecture	**UBL**	Universal Business Language
DBMS	database management system	**WSDL**	Web Services Description Language
IFX	Interactive Financial Exchange	**XBRL**	Extensible Business Reporting Language
MISMO	Mortgage Industry Standards Maintenance Organization	**XACML**	XML Access Control Markup Language
OWL	Web Ontology Language	**XKMS**	XML Key Management Specification
PMML	Predictive Model Markup Language	**XML/A**	XML for Analysis
RDF	Resource Description Framework	**XQuery**	XML Query Language
SAML	Security Assertion Markup Language	**XRML**	XML Rights Management Language
		XSL	Extensible Stylesheet Language
		XSLT	Extensible Stylesheet Language Transformation

Figure 2.1: Hype Cycle for XML Technologies, 2003

2.2 Structure of XML documents

In the following two sections a short introduction of Document Type Definitions and Schemata will be given. Some of the differences, advantages and disadvantages of these two technologies will be covered.

2.2.1 Document Type Definition

Document Type Definition (DTD) was part of the XML 1.0 recommendation. A DTD is written in a formal syntax that explains precisely which elements appear where in the document and what the elements' content and attributes are. [4]

With the help of XML, different ”sets of tags” (= markup languages) can be defined which are adapted to their intended fields of application. XML documents can also be created without the definition of a document structure. But DTDs are useful as they

- serve to unify the structure of different kinds of documents used in a company, e.g. user manuals or data sheets issued by various business groups.
- define a so called ”shared context”. Such a shared context is a formal description of the rules a metadata must follow and serves as a contract between the document sender and the document receiver. The sender agrees that the document conforms to the shared context. The document receiver agrees to interpret the document according to the shared context. [5]

Unfortunately a DTD does not provide the following information about an XML document:

- What the root element of the document is.
- How many instances of an element are allowed in a document.
- What the data inside an element looks like, for example: is it a string, a date, a integer or real value, or something completely different.

A DTD that defines the structure for the XML document that was introduced in section 2.1 would look like the listing below. As you can see elements and attribute lists are described in a formal syntax. The first line declares a 'Book' element and states that it must contain exactly one 'Title','Author' and 'PublishingDate' child element followed by one or more 'Chapter' elements.

```
<! ELEMENT Book (Title , Author , PublishingDate , Chapter+)>

<! ELEMENT Title ( #PCDATA )>
<! ELEMENT Author ( #PCDATA )>
<! ELEMENT PublishingDate ( #PCDATA )>
<! ELEMENT Chapter (ChapterTitle , Paragraph+)>
      <! ATTLIST Chapter ID ID  #REQUIRED >

<! ELEMENT ChapterTitle ( #PCDATA )>
<! ELEMENT Paragraph ( #PCDATA )>
      <! ATTLIST Paragraph ID ID  #REQUIRED >
```

The nesting of elements in a DTD can be constrained by adding special characters to the elements' name. To indicate how many of that elements are expected at that position the following syntax is used:

? means zero or one.

* means zero or more.

+ means one or more.

If none of these characters is added the element is required exactly once. The simplest content specification is one that says an element may only contain parsed character data, but may not contain any child elements of any type. In this case the content specification consists of the keyword '#PCDATA' inside parentheses.

With the 'ATTLIST' keyword one or more attributes can be defined for an element. In this case only the IDs for 'Chapter' and 'Paragraph' were defined and set as '#REQUIRED' so they cannot be omitted.

2.2.2 XML Schema

The *World Wide Web Consortium* (W3C) began work on XML Schema [6] in 1998, and the first version became official in May 2001. The intent was to create a schema language that is more expressive than DTDs, supports namespaces, and unlike DTDs, uses XML syntax. [7]

Schema provides the following features in addition to the advantages already mentioned above at section 2.2.1, they are:

- Simple and complex data types.
- Type derivation and inheritance.

- Element occurrence constraints.
- Namespace-aware element and attribute declaration.
- Control over the uniqueness of values in an instance.
- Data type declaration for elements and attributes.

The listing below points out the differences between XML Schema and Document Type Definition. It shows the Schema for the same example that has been used in section 2.1. At first glance it can be seen that the definition is far more complex than the one used in a DTD. For a complete explanation of each of these tags please refer to [6].

```
<? xml version = "1.0" ?>
<xs:schema xmlns:xs = "http://www.w3.org/2001/XMLSchema" >
<xs:element name = "Book" >
 <xs:complexType>
 <xs:sequence>
     <xs:element name = "Title"
                   minOccurs = "1" maxOccurs = "1" />
     <xs:element name = "Author"
                   minOccurs = "1" maxOccurs = "unbounded" />
     <xs:element name = "PublishingDate"
                   minOccurs = "1" type = "xs:date" />

     <xs:element name = "Chapter" >
     <xs:complexType>
     <xs:sequence>
     <xs:element name = "ChapterTitle"
                   minOccurs = "1" maxOccurs = "1" />
     <xs:element name = "Paragraph"
                   minOccurs = "1" maxOccurs = "unbounded" >
     <xs:complexType>
       <xs:simpleContent>
           <xs:extension base = "xs:string" >
               <xs:attribute ref = "ID" type = "xs:ID" />
           </xs:extension>
       </xs:simpleContent>
     </xs:complexType>
     </xs:element>
</xs:sequence>
       <xs:attribute ref="ID" type="xs:ID" />
</xs:complexType>
</xs:element>

</xs:sequence>
</xs:complexType>
</xs:element>
</xs:schema>
```

The only disadvantage of XML Schema is that it can get quite complex and hard to maintain if your amount of different elements, attributes and their data types is very large.

2.2.3 RELAX NG

The *Regular Language description for XML, New Generation* (RELAX NG) [8] emerged from:

- RELAX Core [9] developed by Murata Makoto and
- *Tree Regular Expressions for XML* (TREX) [10] developed by James Clark.

The work on these two developments listed above has been discontinued. In the year 2001 they were unified and further on developed under control of the *Organization for the Advancement of Structured Information Standards* (OASIS). The goal was to create a competitive alternative to the XML Schema Language.

RELAX NG has been built from a scientific basis that deals with regular tree grammars. XML documents can be seen as context-free grammar for strings, but in fact a XML document is the string representation of a tree structure and therefore tree grammars are ideal for representing a XML document structure.

XML Schema can only describe a subset of regular tree grammars, so called single-type grammars. DTDs are even more limited, they can only describe a subset of single-type grammars, so called local tree grammars. Therefore one of the main advantages of RELAX NG is that it can describe every regular tree grammar.

One possible example [11] to illustrate the theory mentioned above and to show the differences between DTD, XML Schema and RELAX NG could look like this: one element named ’doc’ that holds two elements with the name ’elem’. The first one contains some text the second one should be an empty element.

```
<doc>
   <elem> Hello, World! </elem>
   <elem />
</doc>
```

This grammar cannot be described with a DTD because of the fact that an element has to be declared with a unique data type and cannot be declared in a content related way. So either

```
<! ELEMENT elem ( #PCDATA ) >
```

or

```
<! ELEMENT elem EMPTY >
```

has to be chosen. Also the parent element ’doc’ does not allow a differentiation:

```
<! ELEMENT doc ( elem, elem ) >
```

Even XML Schema would not be able to describe that grammar in a correct way when presumed that the number of occurrences of ’elem’ should not be restricted:

```
<xs:element name = "doc" >
   <xs:complexType>
      <xs:sequence>
         <xs:element  name = "elem"  maxOccurs = "unbounded"
                      type = "xs:string" />
         <xs:element name = "elem" maxOccurs = "unbounded" >
            <xs:complexType>
               <xs:complexContent>
                  <xs:restriction base = "xs:anyType" />
               </xs:complexContent>
            </xs:complexType>
            </xs:element>
        </xs:sequence>
    </xs:complexType>
</xs:element>
```

The validator would work correct for all the elements containing text but reading the first empty element it would behave in a greedy way and treat the empty element as if it were a text element. Due to the fact that the empty element does not contain any text it does not match with the first declaration of 'elem' and the validator would abort.

On the contrary RELAX NG would be able to solve that problem without much complexity. The declaration of the elements would look like this:

```
<element name = "doc" >
    <oneOrMore>
        <element name = "elem" >
            <text />
        </element>
    </oneOrMore>
    <oneOrMore>
        <element name = "elem" >
            <empty />
        </element>
    </oneOrMore>
</element>
```

The semantics of RELAX NG are very straightforward, in this respect, they are a natural extension of DTD semantics. [12] What a RELAX NG schema describes is patterns that consist of quantifications, orderings, and alternations. In addition, RELAX NG introduces a pattern for unordered collection, which neither DTDs nor XML Schemas support. Moreover, RELAX NG treats elements and attributes in an almost uniform manner. Element and attribute uniformity corresponds much better with the conceptual space of XML than does the rigid separation in both DTDs and XML Schemas.

The quantifications available to RELAX NG are identical to those in DTDs. Any pattern may be conditioned as <oneOrMore>, <zeroOrMore>, or <optional>. These correspond to the DTD quantifiers +, *, and ? that are also used in regular expressions. In fact the RELAX NG compact syntax uses the very same quantifiers that are used in DTDs. These very general quantifiers make it more difficult to state specific cardinality constraints, as with the XML Schema 'minOccurs' and 'maxOccurs' attributes can be done. However, it is possible and much easier to work around these limits using named patterns than it is to do so in DTDs.

Ordering multiple patterns is just a matter of listing the patterns in an order. But a sequence of patterns at the same level can be given different semantics using the <choice>, <group>, or <interleave> elements. The <group> tag is used in just the same way that parentheses are in DTDs. By itself, a <group> element does not mean anything, but when used inside <choice> or <interleave> elements, a group acts as one pattern rather than several. The <choice> element expresses simple alternation between contained patterns. The <interleave> element provides the possibility to mix patterns while obeying the cardinality of each contained pattern.

For the sake of completeness the RELAX NG grammar of the example used in section 2.1 is listed below. It shows that RELAX NG can get quite long-winded, even for such a small example, due to the fact that it uses very few attributes.

```
<? xml version = "1.0" ?>
<grammar xmlns = "http://relaxng.org/ns/structure/1.0" >

<start> <ref name = "Book" /> </start>

<define name = "Book" >
   <element name = "Book" >
              <ref name = "Title" />
              <ref name = "Author" />
              <ref name = "PublishingDate" />
              <oneOrMore>
                 <ref name = "Chapter" />
              </oneOrMore>
   </element>
</define>


<define name = "Title" >
   <element name = "Title" >
              <text />
</element>
</define>
```

```
<define name = "Author" >
   <element name = "Author" >
            <text />
</element>
</define>

<define name = "PublishingDate" >
   <element name = "PublishingDate" >
            <text />
</element>
</define>

<define name = "Chapter" >
   <element name = "Chapter" >
            <attribute name = "ID" />
            <element name = "ChapterTitle" >
                        <text />
            </element>
            <oneOrMore>
                        <ref name = "Paragraph" />
            </oneOrMore>
            <text />
</element>
</define>

<define name = "Paragraph" >
   <element name = "Paragraph" >
            <attribute name = "ID" />
            <text />
</element>
</define>

</grammar>
```

RELAX NG is based on the generic concept of patterns. Patterns are similar to XPath node sets, a collection of nodes with an internal structure. The difference between patterns and the other approaches may seem subtle, but a DTD or XML Schema element definition tries to give a description of the element itself. When RELAX NG defines the same element, a pattern is defined that is checked against elements in the instance document to see if they match, much as if it were a regular expression being used to match text [13]. The difference seems to be tiny on the surface, but the pattern approach gives far more flexibility to write, maintain, and combine schemas.

2.3 XSLT

It turns out to be that the *eXtensible Stylesheet Language Transformation* (XSLT) is one of the most important standards, beside XML. Due to the fact that XML is often used as a data interchange format, it is important to have a universal, efficient method at hand, that is capable of converting XML documents. With XSLTs it is very easy to transform XML documents between different structural models but it is also possible to generate other formats like LaTeX or plain text out of XML.

XSLT is a programming language that has been developed especially for transformation of XML documents and because of that, it is the better choice than general purpose languages like C or Java in most cases. For some developers XSLT will seem rather different to normal programming languages because it is based on the concept of functional programming[1]. Good XSLT programming will need a rethinking on design and programming methodology, therefore it is an important aspect to cope with the ideas and consequences of functional programming.

Transformation, in this context, means that an amount of information is transformed into another form of data representation, considering a specific set of rules. If it is not intended to transform the whole amount of information, a preselection has to be done before. With the help of XPath a subset of nodes can be selected from a document, therefore XSLT has a strong relationship with XPath. For more information on XPath please refer to the specification at W3C [14].

The output of an XSL transformation is described with an XSLT stylesheet. Such a stylesheet is a well-formed XML document that uses special XSLT keywords and instructions. These instructions are interpreted and executed with an XSLT processor.

The big difference to Cascading Stylesheets is that XSL transformations do more than just adding formatting properties. XSLTs use XPath to access elements and subtrees of the source document, but they never change the source. Instead the output of the transformation is directed to a destination file. The destination document can be XML formatted,

[1]http://en.wikipedia.org/wiki/Functional_programming

but does not have to.

In most cases, after a transformation some of the information is lost, but it is also possible that after an XSL transformation information is added to the output. Therefore transformations are not reversible in general.

An XSLT processor needs at least two documents:

1. the **source document**, which has to be well-formed XML at least
2. and a **stylesheet**, which inherits the transformation instructions

Afterwards the output stream contains the transformed document. Figure 2.2 pictures the workflow of an XSL transformation. Here only some possible output formats are displayed and some binary formats like PDF also require an additional rendering step after the XSL transformation. However, a complete description of possible output formats and how they are generated will be covered in section 4.3.

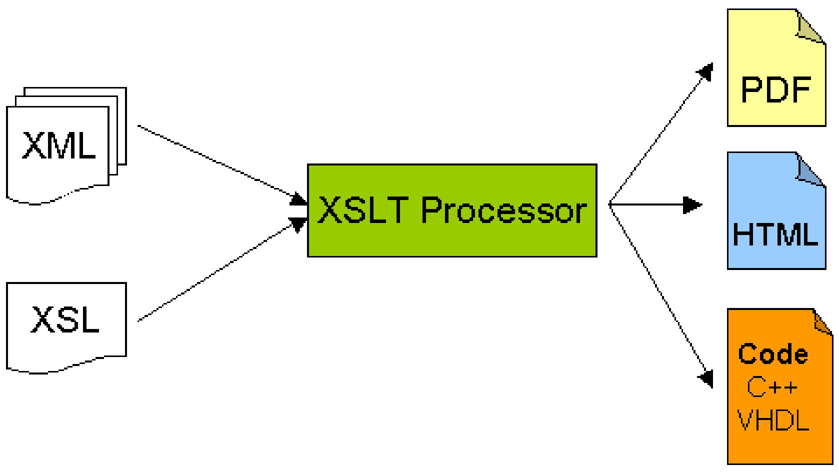


Figure 2.2: The workflow of an XSL transformation

2.3.1 XSLT 1.0 vs. XSLT 2.0

In November 1999 XSLT version 1.0 became a W3C recommendation. As is true with the first versions of many languages, it did not become clear which extensions to the language would prove to be the most important until there had been some real-world experience with it. [15]

XSLT in its version 1.0 has some major limitations. Therefore five new features in XSLT 2.0 will be described here that will help to overcome those limitations: [16]

Grouping: It is a common operation when transforming XML documents in XSLT, to group data. Especially to manage the tabular data from database tables and to publish them on the Web. XSLT 1.0 did not include built-in support for grouping. Many grouping problems certainly can be solved using various techniques, but such solutions tend to be rather complex and verbose. One of XSLT 2.0's requirements was that it must simplify grouping. As a simple example below shows, it is well on its way to meeting that goal.

```
<cities>
   <city name = "milan"    country = "italy"      pop = "5" />
   <city name = "paris"    country = "france"     pop = "7" />
   <city name = "munich"   country = "germany"    pop = "4" />
   <city name = "lyon"     country = "france"     pop = "2" />
   <city name = "venice"   country = "italy"      pop = "1" />
</cities>
```

The goal is to group the cities by the country they are in. To achieve this goal XSLT 1.0 needs a rather complicated work-around that is very error-prone. The code with XSLT 2.0 would like this:

```
<xsl:for-each-group select ="cities/city"
                    group-by ="@country" >
<row>
   <col><xsl:value-of select ="@country" /> </col>
   <col><xsl:value-of select ="current-group()/@name"
                      separator =",  " /> </col>
   <col><xsl:value-of select ="sum(current-group()/@pop)" />
   </col>
</row>
</xsl:for-each-group>
```

In the above example, <xsl:for-each-group> initializes the ”current group” as part of the XPath evaluation context. The current group is simply a sequence. Once the group is set up by using the ’group-by’ attribute, it can thereafter be referred to the current group using the ’current-group()’ function. This completely eliminates the redundancy that was present in the XSLT 1.0 solution.

Note also the ’separator’ attribute on <xsl:value-of>. The mere presence of this attribute instructs the processor to output not just the string value of the first member of the sequence, but the string values of all members of the sequence, in sequence order. The value of the separator attribute is an optional string that is used as a delimiter between each string in the output. For the sake of backward compatibility with XSLT 1.0, only the sequence’s first member’s string value is output when the separator attribute is not present.

Finally, <xsl:for-each-group> is able to solve different kinds of grouping problems depending on which one of the three attributes is chosen:

- group-by: which has been used in the example above
- group-adjacent: which enables grouping based on adjacency of nodes in document order
- group-starting-with: which groups by patterns of elements in a sequence

Multiple output documents: The second XSLT 2.0 feature that simplifies the writing of XSL Stylesheets is its support for creating multiple XSLT outputs. This feature allows the definition of multiple XSLT transformation outputs in one XSL stylesheet. In many real-world applications, the need to create multiple outputs from one XML document is given. For example, when creating an HTML report there could also be several *Scalable Vector Graphics* (SVG) , *Cascading Style Sheets* (CSS) or metadata files created for the report.

Since XSLT 1.0 does not have the capability to define multiple outputs in one XSL stylesheet, there have to be separate XSLT transformations applied for each one. Although it is possible to write one XSL stylesheet that relies on XSLT parameters to indicate which output to create, the XSL stylesheet needs to execute multiple times and can be quite complicated because of the need to manage these parameters.

To simplify these kind of operations XSLT 1.0 processors provide various extensions to solve the problem. However, these extensions in XSL Stylesheets prevent them from being portable across processors. In XSLT 2.0, a new <xsl:result-document> element can be used to define multiple outputs. Generally, two steps are required when defining multiple outputs:

- Setting up the output formats
- Specifying the output file names.

Because each output might have a different format there have to be named output formats defined, using <xsl:output> at the top of the XSL stylesheet. Afterwards each <xsl:result-document> element generates the output format named by its attribute 'format'.

```
<xsl:output method ="xml" indent ="yes" name ="xml-format" />
```

It is used later by <xsl:result-document> to create one XML document. In addition to specifying the output format, you can specify expressions defining the output file names. In the example, the output files are named using a number by calling the 'position()' function:

```
<xsl:result-document
    href ="../output/result_{position()}.xml"
    format ="xml-format" />
```

Temporary Trees: Another new construct introduced in XSLT 2.0 are Temporary Trees. Instead of representing the intermediate XSL transformation results and XSL variables as strings, as in XSLT 1.0, the intermediate results and XSL variables, constructed by <xsl:variable>, <xsl:param>, or <xsl:with-param> elements, are stored as a set of document nodes, called temporary trees.

With temporary trees it is possible to evaluate the content of a variable or a parameter using the XPath expressions, and modularize the XSL processing. This approach offers a lot of flexibility when applying templates or extracting data from XSL variables or parameters. It also improves the possibility to break up complex transformations into several modules and apply iterative processing on the XML documents

Data type bindings: Another simplifying feature is data type binding, which is especially useful when using different types of data, such as dates, durations, numbers and other XML schema data types. In XSLT 1.0, data operations are limited to string, number, and boolean processing. In XSLT 2.0, a stylesheet can use the 44 built-in XML Schema data types and it is possible to construct functions associated with those data types. Data types can be specified for all the variables or parameters defined by <xsl:variable>, <xsl:param> and <xsl:with-param> using the 'as' attribute. The 'as' attribute tells the XSLT processor to check the data types of the variables or parameters, and convert them to the specified data types. This early runtime check prevents further processing of bad data and the unexpected results that may be difficult to debug.

To use these data types, it is necessary to include the XML Schema namespace URI along with the other namespace declarations in the start-tag of <xsl:stylesheet>. A small example is listed below to show the use of this functionality:

```
<xsl:stylesheet version = "2.0"
     xmlns:xs = "http://www.w3.org/2001/XMLSchema"
     xmlns:xsl = "http://www.w3.org/1999/XSL/Transform" >

       <xsl:variable name = "num" select = "Order/Item/@number"
 as = "xs:integer" />
```

User-defined functions: XSLT 2.0 introduces the ability for users to define their own functions which can then be used in XPath expressions. This is an extremely powerful mechanism that should prove to be very useful. Stylesheet functions, as they are called, are defined using the <xsl:function> element. This element has one required attribute called 'name'. It contains any number of <xsl:param> elements, followed by zero or more <xsl:variable> elements, followed by exactly one <xsl:result> element. This restricted content model may sound limiting, but the true power lies in the use of XPath 2.0 to define the result in the select attribute of the <xsl:result> element. As XPath 2.0 includes the ability to do conditional expressions (if...then) and iterative expressions (for...return).

In general it has to be considered that XSLT 2.0 goes hand in hand with XPath 2.0. The two languages are specified separately and have separate requirements documents only because XPath 2.0 is also meant to be used in contexts other than XSLT, such as XQuery 1.0. But for the purposes of XSLT users, the two are linked together. It is not possible to use XPath 2.0 with XSLT 1.0 or XPath 1.0 with XSLT 2.0.

2.4 XSL-FO

This section covers *XSL Formatting Objects* (XSL-FO). It is a complete XML application for describing the precise layout of text on a page. It has elements that represent pages, blocks of text on the pages, graphics, horizontal rules, and more. Most of the time, however, XSL-FO is not written directly. Instead, an XSLT stylesheet that transforms your document's native markup into XSL-FO is written. The application rendering the document reads the XSL-FO and displays it to the user. Since no major browser currently supports direct rendering of XSL-FO documents, there is normally a third step in which another processor transforms the XSL-FO into a third format, such as PDF.

An XSL-FO document describes the layout as a series of nested boxes or areas that are placed on one or more pages. These boxes contain text or occasionally other items, such as an image or a horizontal rule. There are two kinds of boxes that are created by particular elements in the formatting objects document:

- Block areas
- Inline areas

For the most part, the rendering engine decides exactly where to place the boxes and how big to make them, based on their contents. However, properties for these boxes that adjust both their relative and absolute position, spacing, and size on a page can be specified. Most of the time the individual boxes don't overlap. But they can be forced to do so by setting the properties absolute-position, left, top, right, and top on the boxes.

Considered by itself, each box has a content area in which its content, generally text but possibly an image or a rule, is placed. This content area is surrounded by a padding area of blank space. An optional border can surround the padding. The size of the area is the combined size of the border, padding, and content. The box may also have a margin that adds blank space outside the box's area, as shown in figure 2.3.

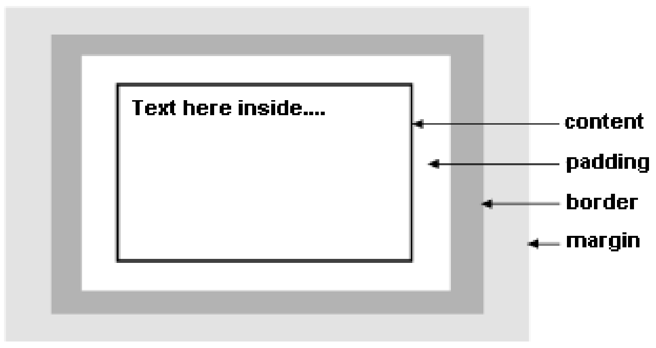


Figure 2.3: Parts of an XSL-FO area

The elements in the XSL-FO document do not map in a one-to-one fashion to the boxes on the page. Instead, the XSL-FO document contains a slightly more abstract representation of the document. The formatting software uses the XSL-FO elements to decide which boxes to create and where to place them. In the process it will split the large blocks, which the XSL-FO document describes with block elements, into smaller line and glyph areas. It may also split single block areas that the XSL-FO document describes into multiple block areas if a page break is required in the middle of a large block, although XSL-FO does let the user prevent these breaks if necessary.

The formatter also generates the correct number of pages for the content that is found. In short, the XSL-FO document contains hints and instructions that the formatter uses to decide what items to place where on which pages. More information on XSL-FO processors can be found in section 4.3.4.

Concluding it can be said that with Formatting Objects and a sophisticated XSL-FO processor really good looking, printable documents can be generated, that can take it on with the output of any ”What You See Is What You Get” (WYSIWYG) program.

2.5 ASP.NET and C#

ASP.NET is more than the next version of *Active Server Pages* (ASP) [17]. Because it has evolved from ASP, ASP.NET looks very similar to its predecessor – but only at first sight. Some items look very familiar, and they may remind of ASP. But concepts like Web Forms, Web Services, or Server Controls gives ASP.NET the power to build real Web Applications.

It provides a unified Web development model that includes the services necessary for developers to build enterprise-class Web applications. While ASP.NET is largely syntax compatible with ASP, it also provides a new programming model and infrastructure for more scalable and stable applications that help provide greater protection. You can feel free to augment your existing ASP applications by incrementally adding ASP.NET functionality to them.

ASP.NET is a compiled, .NET-based environment; you can create applications in any .NET compatible language, including Visual Basic .NET, C# (pronounced "C sharp"), and JScript .NET. Additionally, the entire .NET Framework is available to any ASP.NET application. Developers can easily access the benefits of these technologies. [18]

John Kopp describes the .NET technology in his C# introduction [19] as follows:

> The two main elements of .NET that represent the widest shift from previous Windows based environments are the *common language runtime* (CLR)

> and the .NET framework class library. With older languages, such as C++, source code, code that programmers write, is compiled or turned into a machine specific module usually referred to as an executable or a binary. This module will run only on a particular operating system. A Windows program will not run directly on Unix, for instance. C# is compiled into an intermediate form, called the *Microsoft Intermediate Language* (MSIL). The MSIL is run within the common language runtime. The common language runtime, in turn, converts the MSIL into commands or code that will run on a particular operating system. Readers familiar with Java will, no doubt, recognize the resemblance to Java byte code and the Java Virtual Machine. In addition to this degree of platform and operating system independence, the CLR provides another significant benefit.
>
> Programs running within the CLR are contained. They do not directly interface to the operating system and are limited to the address space provided by the CLR. This prevents the accidental and sometimes deliberate invasion of the address space used by the operating system or by other programs. In addition to increasing program and operating system stability, this also increases security. It's harder for malicious code to reach places to damage.
>
> The CLR also provides for garbage collection (automatic freeing of memory after object use) and support for metadata used to describe and control class use. The .NET framework class library provides a set of classes that provide essential functionality for applications built within the .NET environment. Web functionality, XML support, database support, threading and distributed computing support is provided by the .NET framework class library.

Another great advantage is the separation of semantics and syntax for an entire family of programming languages. The semantics of a programming language are its underlying sense, its meaning. This includes the data types it allows, its rules for type inheritance and the types of constructs it allows. These control how programs and applications are built and how solutions are formed.

Language syntax is the form and style of a programming language. This includes issues such as where brackets are needed, use of space and how constructs such as loops or control structures are formed. By separating semantics and syntax, .NET allows an entire family of programming languages to share the same library functions, the same web functionality and the same database access methodology. Therefore, all .NET family languages are built on top of the CLR. The two most prominent .NET languages are C# and Visual Basic.NET. They share the same semantics, but C# has a syntax derived from C and shared with C++ and Java, while Visual Basic has an alternative look and style.

C# is a new language designed by Microsoft to combine the power of C/C++ and the productivity of Visual Basic. Initial language specifications also reveal obvious similarities to Java, including syntax, strong web integration and automatic memory management. So, if you have programmed in either C++ or Java, starting with C# should be fairly straightforward. [20]

Chapter 3

The Present

This diploma thesis has been written at the IT line for Business Transformations (IT BT) and its RDS (Research & Development Solutions) department at the Infineon Technologies AG in Munich. Infineon is a large enterprise in the international semiconductor industry that designs, develops, manufactures and markets a broad range of semiconductors and complete system solutions targeted at selected industries.

Their products serve applications in the wireless and wireline communications, automotive, industrial, computer, security and chip card markets. Their product portfolio consists of both memory and logic products and includes digital, mixed-signal and analogue integrated circuits (ICs) as well as discrete semiconductor products and system solutions.

3.1 The Role of IT BT

IT BT represents the IT line that executes those projects that are proposed as demands from the different Business Units and fulfill the project approval process. RDS is the department that attends projects and applications belonging to the Research and Development area. This thesis has been written at the RDS department in the area of Technical Communication and therefore I will give a short introduction about the situation that I met when I was starting my diploma thesis and elaborate on how technical documentation is done at Infineon throughout the various Business Units.

3.2 The current Process in Technical Documentation

This chapter will provide an overview on Infineons technical documentation in general as well as an insight into the current documentation process.

The current documentation processes are optimized for the production of print media. The technical authors and document managers take care of authoring, editing, quality management, layout and publishing. At Infineon Technologies the standard tool for writing technical documents at the moment is Adobe FrameMaker (FM) in the current version 7.1. Besides FrameMaker other Microsoft Office Products like Word or Excel are also in use. At present, the documentation is being created using different applications and is stored in various proprietary formats, for instance as FrameMaker or Microsoft Office files. Those files are stored in different systems and databases. Nearly all documents are managed and versioned manually using a directory structure on file servers or local hard disks. As a result, there are sometimes several documents bearing the same name at various places.

Management of reusable text parts is mostly done manually, using functionalities of the word processing software to support it if possible. This means, for instance, that if a new document version is released and parts of it are being used in other documents, the author has to remember this and effect the changes. If the documents are published in other data formats, he also has to redo the data conversion processes as well. Documents have to be collected manually in order to publish them on the Internet. The conversion to the different data formats for electronic publishing has to be initiated by hand, too.

There are several problems with Adobe FrameMaker when it is used for technical documentation. The first problem is a common one, that FrameMaker shares with Word or Excel, and that is the file format that is used to store the data in. FrameMaker uses a proprietary file format that even differs from FM version to version and there are certain incompatibilities known among FrameMaker version 5 and 6 to 7.

However the greatest disadvantage of such an proprietary format is that it cannot or can only hardly be processed by any other software. But the need to extract certain data out of an technical document exists and should not be underestimated. That is one reason why open formats like XML are gaining more and more importance in the area of Technical Communication.

The whole process of technical documentation is described in figure 3.1, which is not very detailed but should give a good overview. The area labeled ’A’ shows the creation and modification process and ’B’ describes the publication process.

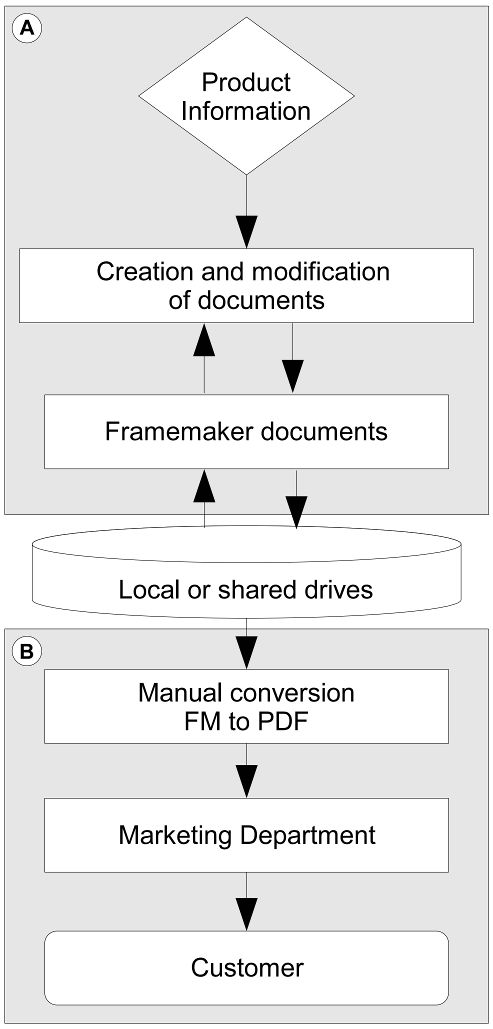


Figure 3.1: Internal documentation process

To give a short overview what document types are treated here I want to give a short list of types that are currently in use, on the technical documentation process described above, at Infineon Technologies.

- **The Product Requirement Document (PRD)**
  is based upon a first product idea. It also contains information on all relevant items

needed to evaluate a business opportunity. This includes: a product summary, information on the market, a system concept, technical concept, a financial plan, customer involvement, as well as information on the software and development tools.

- **The Product Specification**
  is being created based upon the information from the PRD. This specification describes the entire system, lists all relevant standards and test procedures.

- **The Design Specification**
  implements the information provided by the product specification. This specification defines the function of a chip for example, its architecture and the divisioning between hardware and firmware features.

- **The Device Description and Qualification Report**
  The Device Description contains some information about the product, the production process, its thermal characteristics and the amounts and types of packages it will be sold in. Depending on the Business Unit, Qualification Reports are either also part of the Device Description document or issued separately.

- **The Product Brief**
  is a marketing oriented document describing a product idea. It is distributed to selected potential customers and is used to evaluate the interest in this future product.

- **The Data Sheet**
  is created based on data from the PRD and the specifications mentioned above. Depending on the Business Unit, either all information related to the technical characteristics and functionalities is being described in a so-called Data Book or in a Data Sheet combined with a separate User Manual. For less complex products, only a Data Sheet is being used.

- **The Application Notes**
  are created by the Application Engineers based on the information from the Design Specification documents. Application Notes contain information on possible uses of the products.

- **Errata Sheets**
  contain corrections of Data Sheet errors.

- **User Manual**
  is a document for novice users that explains how to use or operate a product. Usually organized topically or by task.

3.2.1 FrameMaker in structured mode

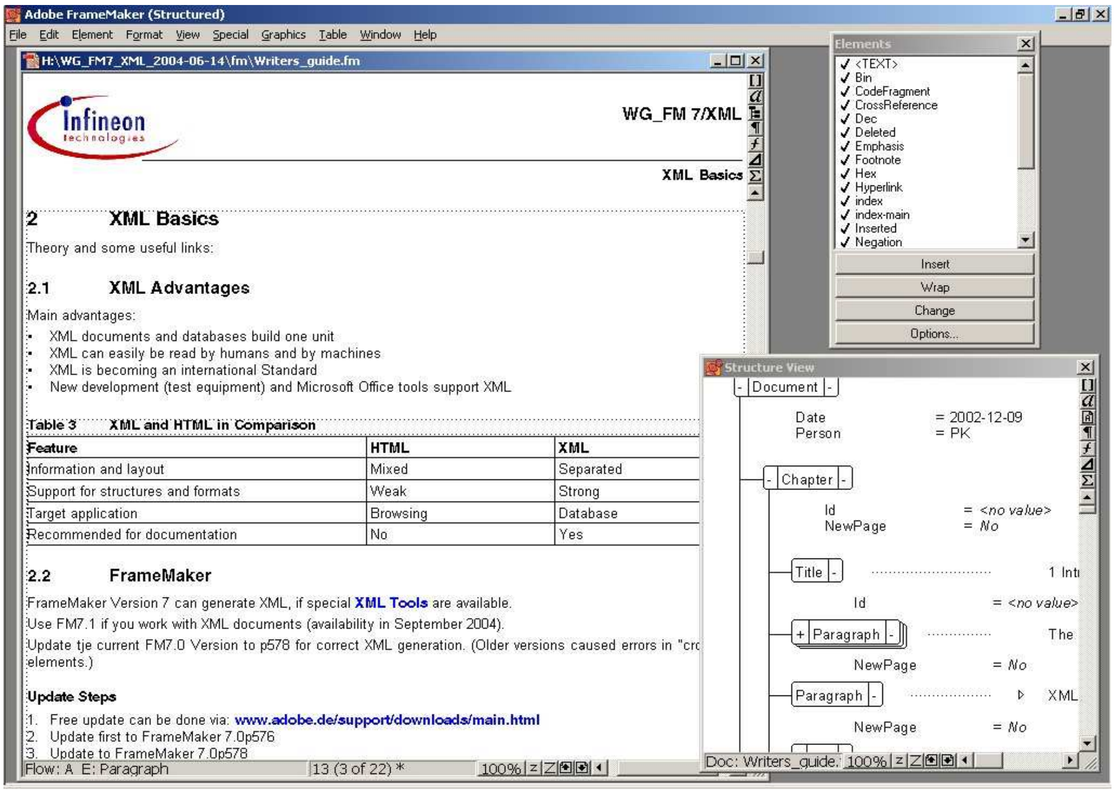
As described above, technical documentation at Infineon is done with Adobe FrameMaker mainly. FrameMaker is a professional level desktop publishing package, it provides a large feature set for publishing to both paper and electronic formats. FrameMaker's formatting engine allows users to do complex text formatting and page layout. With the version of FrameMaker 7.1 a so called ”structured mode” has been introduced. This means that FrameMaker allows the user to define structural elements like chapters, paragraphs, figures, tables or any other kind of element that describes pieces of information.

In order to use this feature a lot of work has to be done before, you have to set up so called ”Application settings”. These settings include a document called *Element Definition Document* (EDD) and Read-Write rules. An EDD is similar to a DTD, but unfortunately compromises the idea of separating content and layout, and adds formatting information to the elements structure. Another drawback of an EDD is that it is an Adobe proprietary format and therefore can only be accessed and edited when you open it with Adobe FrameMaker.

The major advantage of working with FrameMaker in structured mode is that it allows you to export your FM documents as XML documents. In order to do such an XML export Read-Write rules are needed to specify how element names are translated to tag names in XML. Furthermore you need an EDD which ensures that elements are structured in a way they are allowed to, similar to a DTD.

The screenshot in figure 3.2 shows how Adobe FrameMaker looks like in structured mode. On the left side of the figure you can see the WYSIWYG window of the current document. In front is the structured view that represents the current document in a treeform and on the upper window all elements are listed that can be inserted at the current cursor position.

If FrameMaker is used as an authoring tool for a large group of people there are certainly some difficulties that could be avoided using a pure XML authoring environment. First of all, if a group of authors should write their documents using the same set of tags, FM application settings including an EDD and Read-Write rules would have to be distributed to all of them by hand. That means that an EDD cannot be referenced as simple as a DTD by providing an *Universal Resource Identifier* (URI). Furthermore a DTD can be extended easily and does not have to be reversioned, as long as certain constraints are considered. On the contrary, if changes are made or something is extended in an EDD, the Read-Write rules have to be updated and the whole FM application settings have to be redistributed among all authors.



Infineon technologies

WG_FM 7/XML

XML Basics

2 XML Basics

Theory and some useful links:

2.1 XML Advantages

Main advantages:

- XML documents and databases build one unit
- XML can easily be read by humans and by machines
- XML is becoming an international Standard
- New development (test equipment) and Microsoft Office tools support XML

Table 3 XML and HTML in Comparison

Feature	HTML	XML
Information and layout	Mixed	Separated
Support for structures and formats	Weak	Strong
Target application	Browsing	Database
Recommended for documentation	No	Yes

2.2 FrameMaker

FrameMaker Version 7 can generate XML, if special **XML Tools** are available.

Use FM7.1 if you work with XML documents (availability in September 2004).

Update tje current FM7.0 Version to p578 for correct XML generation. (Older versions caused errors in "crc elements.)

Update Steps

1. Free update can be done via: **www.adobe.de/support/downloads/main.html**
2. Update first to FrameMaker 7.0p576
3. Update to FrameMaker 7.0p578

Date	=	2002-12-09
Person	=	PK
Id	=	*<no value>*
NewPage	=	*No*
Id	=	*<no value>*
NewPage	=	*No*
NewPage	=	*No*

Figure 3.2: FrameMaker in structured mode

3.2.2 Use of FrameMaker's XML export

The possibility to export XML from a FrameMaker document is currently used to apply certain automated steps onto a document. These automation would be too difficult to be applied on FM documents directly, with the help of a scripting language or through the FrameMaker API. Therefore the documents are exported as XML and afterwards a few XSLTs are used to perform tasks like:

- generating so called 'special chapters', which include the content of a normal chapter in compressed form at the end of the document
- generating SVGs which for example draw a pin assignment, out of tabular data

After the transformation the XML documents are imported into FrameMaker to continue the work inside the FrameMaker environment. Figure 3.3 points out the workflow described above and should show that this is rather a workaround than something suitable for productive use.

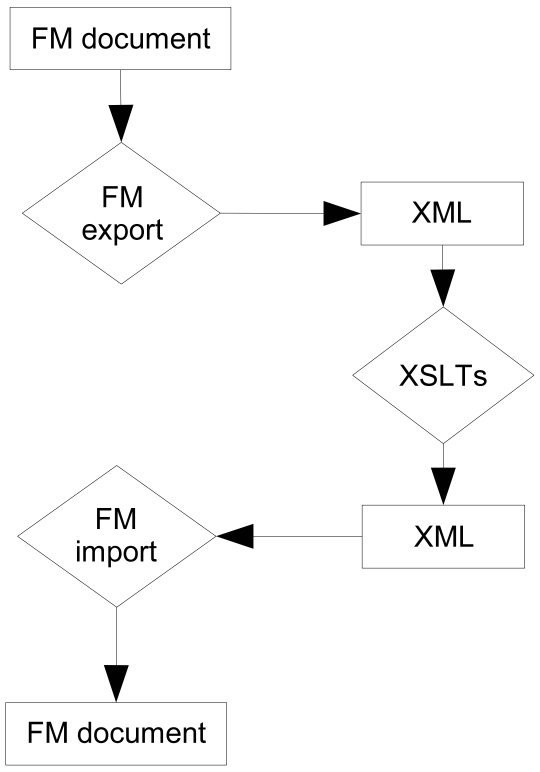


Figure 3.3: The use of the FrameMaker XML import and export

3.3 Conclusion

Concluding it can be said that the tools described above and how they are used in technical documentation today, is something that evolved over the years. It is almost impossible to replace such a system from one day to another. However a prediction can be made that the use of Adobe FrameMaker in structured mode, together with an EDD, Read-Write rules and various XSLTs will become far to complex to be handled efficiently over the time. And therefore I believe that it is an reasonable effort to take a closer look on possible alternatives.

Chapter 4

The Future

This chapter is going to point out how Technical Documentation could be done in the future, on the basis of a XML-based environment. Changes brought about with the introduction of XML and the advantages and disadvantages of this new technology will be shown. Most of the information has been derived from discussions with authors, document managers and team members.

4.1 XML-based Technical Documentation

The explosion of content and the need to manage, mine and obtain information in a quick and efficient way is nowadays more important than ever before. Therefore the advantages of an XML-based solution for technical documentation are described in the next sections.

4.1.1 Standardization

Standardization in information representation and transfer is crucial. XML is a platform- and application independent, and vendor-neutral mechanism. XML relies on other technologies, in particular: SGML for syntax, URIs for name identifiers and Unicode for character encoding, which are all standards. This high grade of standardization provides a solid basis to use this technology in the field of technical documentation.

4.1.2 Manageability

The advantage of data being independent of any particular platform, application or vendor is that it can be transformed to produce different types of outputs for different media devices as for instance: paper, CD-ROM or a web browser without the need to modify the original content.

When modifications are required, only the original version of the content needs to be edited before republishing to the various target media. This leads to efficiency and easy maintainability, without the inherent problems of version control and the effort required in making modifications in medium-specific document versions. This allows authors to concentrate on authoring rather than formatting and thus to be more productive.

4.1.3 Longevity

Proprietary binary formats only lasts as along as the systems which support it. XML data exists as plain text. This gives data a longer life span with future readability and reuse of data. Even if a system becomes obsolete, the data will live on and will remain accessible in the long term. This is a very valuable attribute of the XML technology, consider the support for a product like Adobe FrameMaker would stop, the process described in section 3.2 would get in serious trouble.

4.1.4 Extensibility

XML provides a standard framework to create markup vocabularies that are business-oriented and can easily be used to build a meta language for any kind of technical documentation. Due to its complexity this was much more difficult with its predecessor *Standard Generalized Markup Language* (SGML).

4.1.5 Development

XML is simpler than SGML, and easier to implement. This has the following practical advantages:

1. One of the prime advantages of XML is that generic programs, such as parsers, can be developed that can be used with any XML vocabulary. Even though the programs themselves could be platform-dependent, the XML documents themselves are plain text, and therefore platform-independent.

2. A programmer can use tools from different vendors to process the same data. With several choices of XML processors already available, it also frees the programmer from having to write parsers from scratch. This allows the programmer to concentrate on other aspects of the application.

3. XML is usually referred to as ”portable data” in the sense that its parsing is ”application independent” and one XML parser can read every possible XML document.

4. The time and effort used in set-up of the programming environment (libraries, modules, software) can be used for different XML vocabularies.

5. The skills attained in a project on one specific vocabulary is often transferable to others. For example, once having learned how to write XSLT style sheets for transformations, this skill can be reused with any other XML vocabulary.

4.1.6 Decoupling Structure and Presentation

The vocabularies based on XML syntax can keep the structure of the content separate from its presentation. This has long-term advantages, particular towards document maintenance and automated processing. This is a major improvement over HTML which allowed the mixture of structure and presentation, both implicitly and explicitly. And especially in Technical Documentation this separation of structure and layout is a major benefit, for example in generating different output formats from one content source. This will be covered in detail in section 4.3.

4.1.7 Human-to-Machine and Machine-to-Machine Interfaces

XML provides both machine-to-machine and human-to-machine interface due to its markup characteristics following a strict syntax. XML is a data representation that has the characteristics of a document. This document could be a file, a record in a relational database or a stream of bytes arriving at a network socket. The concept is very powerful because it implies that the same information can be processed (data view) or can be presented (document view) in the same application at the same time. Even though it is expected that large documents will be processed by machines, they are still human-interpretable. [21]

4.1.8 Business-to-Business Communication

XML simplifies business-to-business communication, particularly for the following reasons:

1. The only thing that is to be mutually agreed upon is the XML vocabulary that will be used to represent data.

2. Neither company has to know how the other's back-end systems (platforms, operating systems, programming languages) are organized, which does not put any extra technical burden while keeping the privacy. All that is required is that each company develop the mapping to transform XML documents into the internal format used by the back-end systems.

3. XML-based solutions are scalable: If there is an addition of another partner, there is no need by the host company to interact with the systems of the new company. All that is required is that they follow the protocol which is represented by the XML vocabulary.

4.1.8.1 ebXML

One of the technically mature and wide spread standards for B2B communication on XML basis is *Electronic Business using eXtensible Markup Language* (ebXML). The following information is derived from the ebXML homepage.[1].

ebXML is a modular suite of specifications that enables enterprises of any size and in any geographical location to conduct business over the Internet. Using ebXML, companies now have a standard method to exchange business messages, conduct trading relationships, communicate data in common terms and define and register business processes.

ebXML provides :

- The only globally developed open XML-based standard built on a rich heritage of electronic business experience.
- Creates a single global electronic market that enables all parties irrespective of size to engage in Internet based electronic business.
- Enables parties to complement and expand electronic business to new and existing trading partners.
- Facilitates convergence of current and emerging XML efforts.

ebXML delivers the value by

- Developing technical specifications for the open ebXML infrastructure.
- Creating the technical specifications with the world's best experts.
- Collaborating with other initiatives and standard development organizations.
- Building on the experience and strengths of existing *Electronic Data Interchange* (EDI) knowledge.
- Enlisting industry leaders to participate and adopt ebXML infrastructure.
- Realizing the commitment by ebXML participants to implement the ebXML technical specifications.

[1] `www.ebxml.org`

ebXML was started in 1999 as an initiative of OASIS and the United Nations agency CEFACT. The original project envisioned and delivered five layers of substantive data specification, including XML standards for:

- Business processes
- Core data components
- Collaboration protocol agreements
- Messaging
- Registries and repositories

4.1.8.2 An Electronic Business Scenario

The following example should help to illustrate the key benefits of ebXML, as Benoît Marchal described it in his article [22].

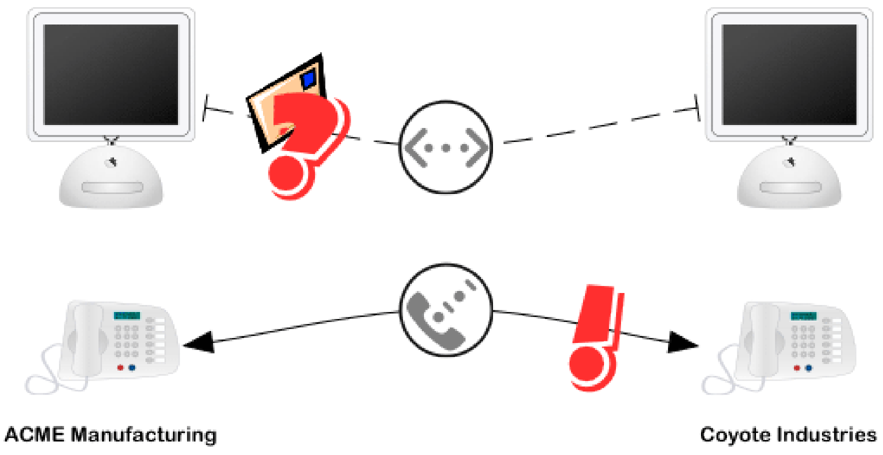


Figure 4.1: An Electronic Business Scenario without ebXML according to [22]

Figure 4.1 is an electronic business scenario without ebXML. ACME Manufacturing and one of its largest supplier, Coyote Industries, want to implement an electronic relationship.

The goal is to streamline ACME procurement, reducing costs and increasing competitiveness. Assume that business issues have been dealt with: ACME wants to buy from Coyote, they have agreed on terms and payments. ACME and Coyote add an XML connector to their business management system (such as SAP, Sage or QuickBooks). That sounds easy, but in fact it is rather difficult.

There are many technical issues to be dealt with: which XML schema to use, what security to put in place, how to chain electronic messages (e.g. does Coyote invoice ACME for every order or does it issue a monthly invoice), what communication protocols to adopt. There are also many parameters to get right: the address of the server, time-outs, and more.

Currently setting up an XML connector is a lengthy and costly business. It requires many back and forth meetings/phone calls between trained professionals and, generally speaking, it is not very efficient.

Figure 4.2 is the ebXML equivalent of figure 4.1. The only difference, but a significant one, is that the slow and costly process of setting up the connector has been automated.

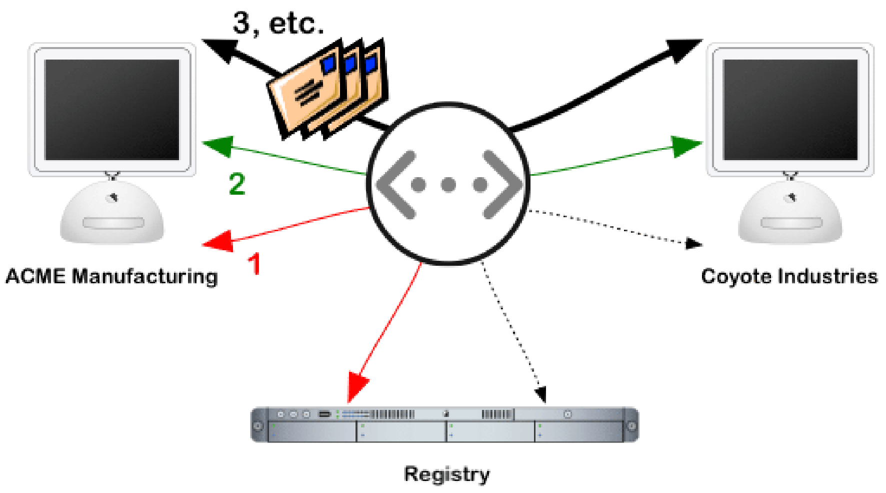


Figure 4.2: An Electronic Business Scenario with ebXML according to [22]

It starts (step 1) with ACME looking up Coyote information in a registry. A registry is an open directory where companies list a very detailed description of their electronic capabilities.

In ebXML lingo, the description is known as a *Collaboration Protocol Profile* (CPP). The CPP contains the answer to the questions discussed in the previous section: which XML schemas Coyote recognizes, the security mechanisms it has put in place, the address of its server and much more.

In a second step, ACME matches Coyote description with its own to find commonalities. In the process it creates a *Collaborative Partner Agreement* (CPA) that describes how to establish the electronic relationship. In essence, the CPA is the answer to all the questions put forward in the first case.

What does the CPA have to include? In most cases, the choices are simple and can be automated. For example, if Coyote supports HTTPS or PGP but ACME supports S/MIME and PGP, the only practical option is PGP. Some issues might not be so clean cut and would be reviewed by the operator installing the software. ACME sends the CPA to Coyote for validation. If Coyote accepts it, they are in business.

Step 3 and beyond is to exploit the business relationship. The dotted line between Coyote and the registry in figure 4.2 indicates that Coyote will update its CPP as it adds or removes capabilities. E.g. if Coyote adds S/MIME support, it needs to say so in its CPP.

The fundamental difference between the two scenarios is that, with ebXML, the costly configuration is now automatic, which means cheaper and faster.
Obviously there is another difference between the scenarios with and without ebXML. If you can establish electronic relationships in minutes instead of weeks, then you can afford to build many more of those.

To get a better understanding on how the interaction with the help of ebXML works, a closer look on the Collaboration Protocol Profile (CPP) is taken.

The CPP is a formal description file that lists what an organization can do in terms of ebXML operations. Formal is the operative word because it means that software can decode it. More specifically, the CPP contains:

- the organization's name and contact information;
- the transport and security protocols it recognizes;
- the messaging protocols it recognizes;

- the business processes (and therefore ultimately the XML schemas) that the organization recognizes;
- all kind of technical information such as URLs, time-out, certificates, and more that are necessary to setup an ebXML relationship.

Since the CPP describes all the capabilities of an organization, it tends to be a fairly long document (20 pages or more). The CPP is written in XML and follows very strict conventions.

The XML snippet below is a tiny excerpt from Coyote's CPP. This excerpt deals with the transport protocols that an organization supports. It contains two tp:Transport elements to indicate that the organization accepts messages either via *Hyper Text Transfer Protocol* (HTTP) or via *Simple Mail Transfer Protocol* (SMTP).

```
<tp:Transport tp:transportId="http_connection">
<tp:TransportReceiver>
<tp:TransportProtocol tp:version="1.1">HTTP</tp:TransportProtocol>

<tp:Endpoint tp:uri="https://www.coyote.com/ebxml"
             tp:type="allPurpose"/>
<tp:TransportServerSecurity>
<tp:TransportSecurityProtocol tp:version="3.0">SSL
   </tp:TransportSecurityProtocol>
<tp:ServerCertificateRef tp:certId="server_cert"/>
<tp:ClientSecurityDetailsRef tp:securityId="tp_security"/>

</tp:TransportServerSecurity>
</tp:TransportReceiver>
</tp:Transport>
<tp:Transport tp:transportId="smtp_connection">
<tp:TransportReceiver>
<tp:TransportProtocol>SMTP</tp:TransportProtocol>
<tp:Endpoint tp:uri="mailto:ebxml@coyote.com"
             tp:type="allPurpose"/>
</tp:TransportReceiver>
</tp:Transport>
```

The transport elements specify the protocol in use (tp:TransportProtocol), the address of the server (tp:Endpoint) and optionally security information (tp:TransportServerSecurity). In the example above, the HTTP connection is secured through SSL, the SMTP connection is not.

Every organization has a CPP. Coyote has one but its business partner, ACME Manufacturing also has one. To establish the electronic relationship between the two, it suffices to look up the commonalities in the two CPPs. Software can very easily establish the match.

Suppose ACME's CPP states that ACME supports HTTP, then the software finds a match and it can configure itself for HTTP. Since the CPP contains the address of the server and the security parameter, the configuration requires no human intervention. For simplicity, the above example has concentrated on transport protocols because that is where you need the fewest parameters. Bear in mind that the CPP lists every possible option, including which XML schemas to use (through the business process). A complete CPP is a very long document.

Still one question remains: it all sounds great but what if the ACME's CPP is incompatible with Coyote. Say ACME supports FTP only. As we have seen, Coyote does not support FTP. Should that happen, it means that two parties cannot establish an electronic relationship. At least one of the two must upgrade its system to make it compatible with the other. Unfortunately software is not capable of generating relationships out of nowhere. This has always been a major problem long before ebXML and will remain so even after ebXML.

4.1.9 Modular DTD

As we have seen in the previous sections, XML in technical documentation provides some great advantages. But there are also some dangers you can easily run into, if you haven't planned your actions very well. If you plan to use just a single DTD to describe your XML vocabulary, be aware of the fact that this DTD can become quite unhandy the more elements you define. That is what happened with the DTD respectively EDD as described in section 3.2, the size of the EDD increased very fast over the time. And today the EDD used at Infineon Technologies for technical documentation is so huge that it gets hard to maintain and work with it at all.

Besides that, one fact has to be considered: FrameMaker is able to generate a DTD out of an EDD, but there is one major drawback: FrameMaker is able to import a modular DTD but is not able to export a modular DTD, as shown in figure 4.3. To be able to cope with that DTD / EDD in the future, it is necessary to split

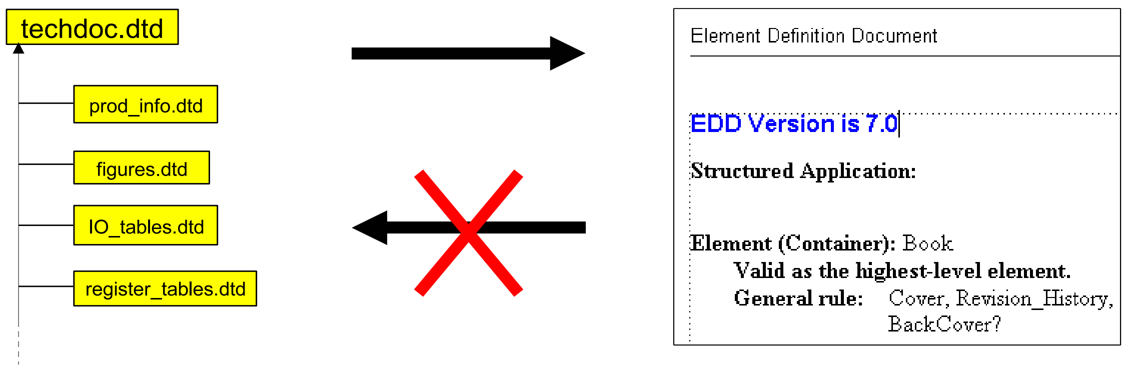

techdoc.dtd
prod_info.dtd
figures.dtd
IO_tables.dtd
register_tables.dtd
Element Definition Document
EDD Version is 7.0
Structured Application:
Element (Container): Book
Valid as the highest-level element.
General rule: Cover, Revision_History, BackCover?

Figure 4.3: FrameMaker is not capable of writing modular DTDs

it up into smaller pieces. For the current situation at Infineon Technologies there are two possible alternatives to achieve that.

The first one is a distributed DTD. Which means, every Business Unit (BU) has its own set of DTDs that fit their needs. But they all agree upon one slim, global DTD for technical documentation that consists only of a small set of elements, that is necessary for document publishing. Figure 4.4 pictures this approach. In this case Business Unit specific information, like registers or I/O table data is converted with XSL transformations to normal tabular data before publishing the document.

Advantages of distributed DTDs:

- Each Business Unit has its own XML vocabulary.
- Only the most necessary elements are listed in the global DTD.
- Changes inside the XML vocabulary of a BU only concern the XSLTs of this Business Unit.

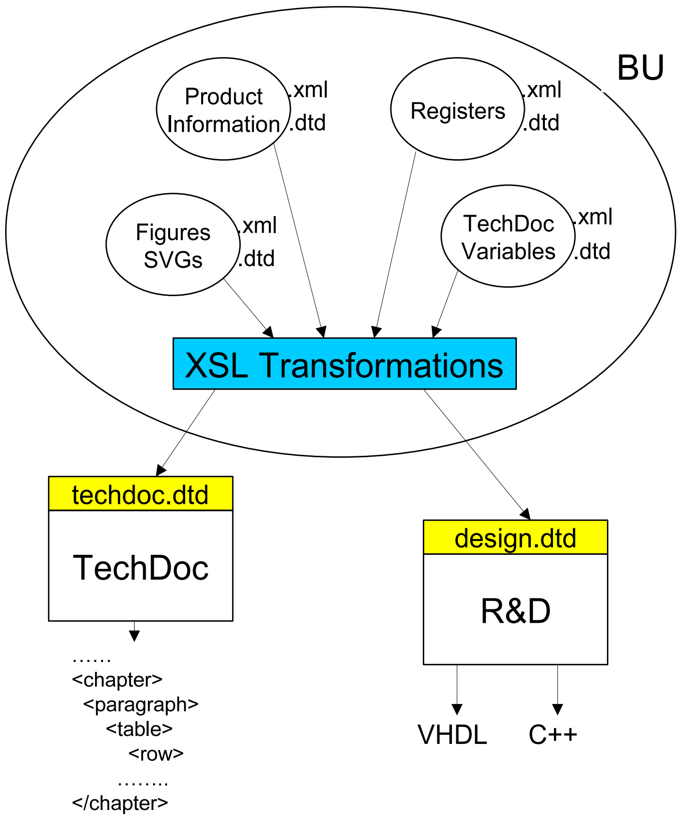


Figure 4.4: Approach of a distributed DTD

Disadvantages:

- Many XSLTs have to be maintained.
- If changes have to be made inside the global DTD all BUs must be informed and have to agree to these changes.
- Exchange of data between BUs is more difficult due to the fact that they use different XML vocabularies.

The second alternative uses nested DTDs. In figure 4.5 a global Infineon Technical Documentation DTD is build up from smaller DTD components.

Advantages of nested DTDs:

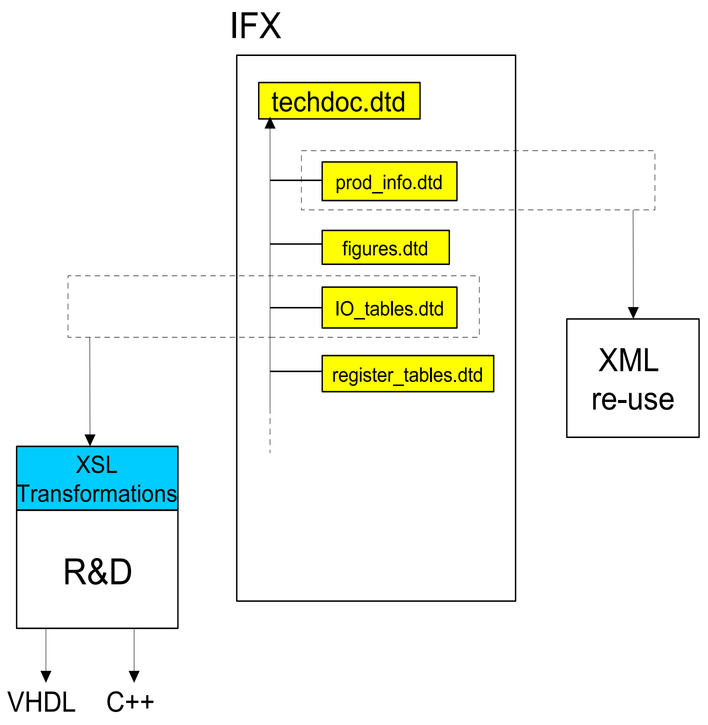


Figure 4.5: Approach of a nested DTD

- The DTD can be referenced by URI from all participants.
- XML reuse between BUs is possible.
- Single smaller DTDs can also be referenced.
- Easily extendable by adding new DTD components.

Disadvantages:

- Heavy impact on XSLTs when changes to the DTD are made.
- Double use of element names only becomes visible by validating against the global DTD.
- Versioning of global DTD is necessary.

A global decision on which of these two alternatives is best can hardly be made. Considering all the pros and cons, it depends on the commitments all parties concerned are willing to make, to choose the right approach.

4.2 Single Sourcing

Over the last years one term has caused a sensation among technical writers: ”Single Sourcing”. The reason is the enormous amount of text and image material that builds up in documentation throughout all companies. So what’s the big deal about single sourcing? Kurt Ament provides a good answer to this question in his book [23]:

> ”Single sourcing is a documentation method that enables you to reuse the information that you develop. You build modular information, then assemble that information into different formats. Reusing information saves you time and money because it eliminates duplicate work.
>
> Single sourcing also increases the usability of your documentation. By developing modular information that is usable in any format, you raise your documentation standards. Usability standards that are ’nice to have’ in traditional documents become ’must haves’ in modular documents. In effect, single sourcing forces you to do the right thing for your users.”

However, it is very important that single sourcing has to be seen as a methodology. It is not a technology that can simply be used on technical documents. Kurt Ament points this out:

> ”Single sourcing is a methodology, not a technology. Although the software tools associated with single sourcing are complex, it is modular writing, not technology, that ultimately determines the success of single sourcing projects. To ensure success, develop local, project-based standards for modular writing. Base your standards on what actually works in your own projects.”

The key features to do successful single sourcing in technical documentation are examined throughout the next sections.

4.2.1 Conditions

When using the single sourcing method, concentrate on content and not on format. Therefore content-based documentation is very different from format-based documentation, which is written only for a particular document format and is designed to be read in a predefined sequence.

So there are several things that have to be considered:

Format-based content is not reusable

In traditional documentation individual documents are hand-crafted for a particular output format only, for example:

- Printed manual
- Online help
- Website

But the content is always tied to the format and cannot be reused easily. Reuse of this type of documents always requires extensive rewriting.

Content-based content is reusable

To take advantage of the re-usability separation of content from format has to be done when developing information. To achieve this a technology like XML is the ideal solution, furthermore it enables the development information on an element level rather than a document level:

- Document level (linear)
- Element level (modular)

Linear writing is not reusable

Linear writing assumes a given reading sequence. It is intended to be read from beginning to end. For instance in linear writing pieces of text like ”later in this chapter" or ”earlier in this section" would be found very often. Obviously these references would not make much sense in an online help system.

All these sequence references would have to be removed and have to be rewritten to use the information in a modular way. Therefore linear documentation is almost impossible to reuse.

Modular writing is reusable

Modular writing is non-sequential, stand alone content modules that make sense on their own have to be written. The idea is to ’chunk’ information based on the type of information being presented. For example, use descriptive text to explain what a product is and use procedures to explain how to operate the product and flowcharts to illustrate product processes.

To assemble those chunks to a document use cross-references, to link each step to another. When chunking and linking information in this way, a development of content, that can easily be assembled into different documents is possible.

4.2.2 Granularity of Structure

Granularity of a documents’ structure is an important point to successful single sourcing, although there cannot be a universally valid answer to the question what the best granularity for your document structure is. That has to be defined for each case individually. If a document structure is to fine and uses too much elements, it is easily possible to get lost in the amount of different elements and when writing content it comes out to spending more time with tagging a single piece of information than actually writing content.

On the other hand, if a document is structured to roughly, the ability to reuse pieces of information is lost and together with that the advantage of single sourcing. It sometimes is a bit tricky, but it is important to keep the balance when planning how to structure technical documents.

4.2.3 Reasons for Single Sourcing

If planning and implementation is done properly, single sourcing can save time and money. If once single sourcing is established it enables the development of content and the reuse in many different ways. When reusing the same content in different formats for different audiences and purposes, reduction of duplicate work is achieved.

For example, when developing a printed manual and an online help system for the same product, both documents will contain very similar information, only the formats differ. If development of these two documents is done separately, the same work is done twice. But if content modules are used instead, there is no need to do the work for both formats separately, this cuts down the workload.

When using single sourcing the right way, it does more than just reducing workload, it also increases flexibility. Imagine a project that at the beginning targets the development of an online help system for a specific product. After half of the work is done, the possibility to provide a printable version of the online help system should also be guaranteed. This new requirement would be hard to fulfill if the help system would have implemented in HTML, for instance. A system would have been developed where content and layout is mixed together. Due to the advantages of single sourcing the development of this online help system has already been implemented with XML and with the use of XSLTs it is provided to the web. And because of the clear separation of content and formatting it is now possible to provide a printable version of the help system with little more effort.

This simple example shows that single sourcing also increases flexibility.

4.2.4 Improved Document Assembly

Single sourcing used in an XML environment provides another advantage. The document assembly does not have to be done by hand anymore, XSLT gives the user a powerful technology at hand that is able to combine modular content chunks to a single output file.

Only an 'assembly file' is needed anymore where your content chunks are referenced in. And if a XSLT is used on that file, the output would be a properly formatted document, including the dereferenced content. Figure 4.6 illustrates this workflow.

4.2.5 Evaluating Single Sourcing

While single sourcing can help companies reduce costs and improve quality, implementing it requires research and planning. The costs required to purchase, implement, and maintain the software to support the single sourcing are low compared to the effort that has to be invested in teaching the technical writers. Because first of all they have to learn the methodology of the single sourcing approach only

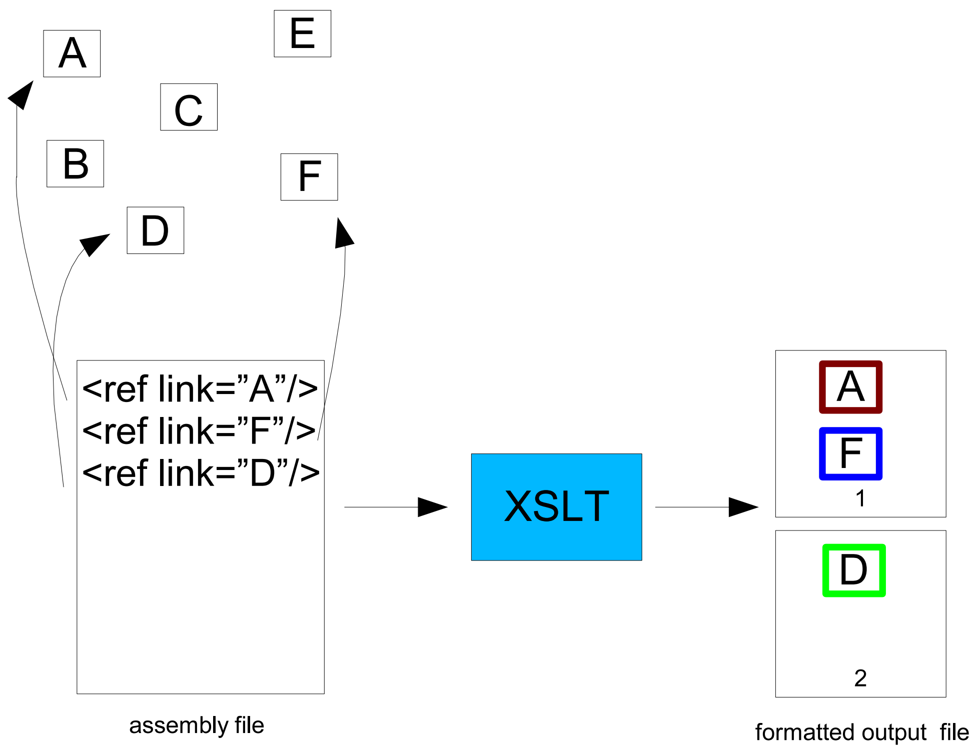


Figure 4.6: Example of an assembly file

after that, the software can support their work.

The concept of single sourcing promises reduced cost and increased quality. By writing, editing, reviewing, and localizing the information once and reusing it many times in different formats and forms, production and maintenance costs drop significantly. The quality of deliverables increases because the information content can be tailored to each format. [24]

Single sourcing can be a challenging transition but may not be the correct solution for every situation. For a company like Infineon Technologies that manufactures so many products and to each product a significant number of technical documents has to be provided, in my opinion it is an absolute must. Although the advantages outweigh the drawbacks, a survey made by Cherryleaf Ltd. [25] in April 2003 shows in figure 4.7 that single sourcing is not that widely-used yet.

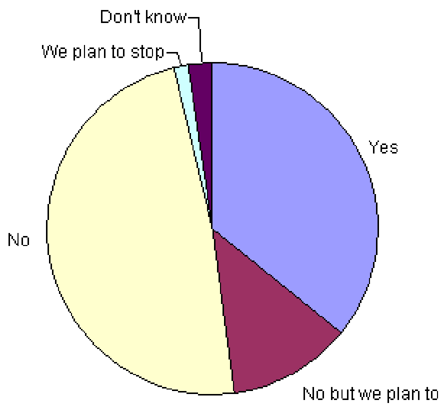


Figure 4.7: Survey results - Use of single sourcing solutions

Response	**Percentage**
Yes - We use a single sourcing authoring solution	35.9
No, but we plan to in the next 12 months	11.9
No	48.5
Yes, but we plan to stop doing so in the next 12 months	1.5
Don't know	2.2

4.3 Transformation to different Output Formats

The markup in a typical XML document describes the document's structure, but it should not describe the document's layout. It says how the document is organized but not how it looks. Although XML documents are plain text, and someone could read them in native form, it is much more commonly an XML document is rendered into some other format before being presented to a human audience. One of the key ideas of markup languages in general, and XML in particular, is that the input format need not be the same as the output format. The input markup language is designed for the convenience of the writer. The output language is designed for the convenience of the reader.

Of course this requires a means of transforming the input format into the output format. Most XML documents undergo some kind of transformation before being presented to the reader. The transformation may be to a different XML vocabulary like XHTML or XSL-FO, or it may be to a non-XML format like PostScript.

XML's major transformation language is the *eXtensible Stylesheet Language Transformations* (XSLT). However, XSLT is not the only transformation language you can use with your XML documents. Other stylesheet languages such as the *Document Style Semantics and Specification Language* (DSSSL) [26] are also available. So are a variety of proprietary tools like OmniMark[2]. Most of these have particular strengths and weaknesses for particular kinds of documents.

Custom programs written in a variety of programming languages, such as Java, C++, Perl, and Python, can also be used to transform documents. This is sometimes useful when you need something more than a simple transformation, for instance, interpreting certain elements as database queries and actually inserting the results of those queries into the output document. However, the biggest factor when choosing which tool to use is simply which language and syntax you're most comfortable with.

There are many different choices for the output format from a transformation. A PostScript file can be printed on paper, overhead transparencies, slides. A PDF document can be viewed in all these ways and shown on the screen as well. Simple HTML has the advantages of being very broadly accessible across platforms and being very easy to generate via XSLT from source XML documents. Generating a PDF or a PostScript file normally requires an additional conversion step in which special software converts some custom XML output format like XSL-FO to what you actually want.

An alternative to a transformation-based presentation is to provide a descriptive stylesheet that simply states how each element in the original document should be formatted. This is the realm of *Cascading Style Sheets* (CSS). This works particularly well for narrative documents where all that is needed is a list of the fonts, styles, sizes, and so on to apply to the content of each element. The key is that

[2] http://www.omnimark.com/

when all markup is stripped from the document, what remains is more or less a plain-text version of what you want to see. No reordering or rearrangement is necessary.

This approach works not very well for data-oriented documents where the raw content may be nothing more than an undifferentiated mass of numbers, dates, or other information that is hard to understand without the context and annotations provided by the markup. However, in this case a combination of the two approaches works well. First a transformation can produce a new document containing rearranged and annotated information. Then a CSS stylesheet can apply style rules to the elements in this transformed document.

4.3.1 HTML

The *HyperText Markup Language* (HTML) derives from the same family of markup languages as XML does. That means that the transformation from XML data to HTML is a direct one, there are no rendering engines or compilers needed. So with the help of XSLTs you can easily convert your XML documents to HTML. Figure 4.8 pictures this workflow.

The main advantage of HTML is that with the quick growth of the *World Wide Web* (WWW), it is available almost everywhere and on any possible architecture or operating system. Therefore, with the help of XSLT, you have the possibility to provide your latest information to the web directly out of your XML data basis. With the further development of browser technology in the near future, even the step of XSL transformation will no longer be necessary, because the browsers will be able to present XML information directly (of course with the support of style information it will always look better).

Another convenience of the direct transformation from XML to HTML is the fact that you can use it as some sort of preview for the information you are creating. As mentioned in section 4.3, XML structured information brings an advantage to the writer, but it is not that easy to read. As a writer you have to re-read your writing from time to time, and to do this, an HTML preview really is a great ease.

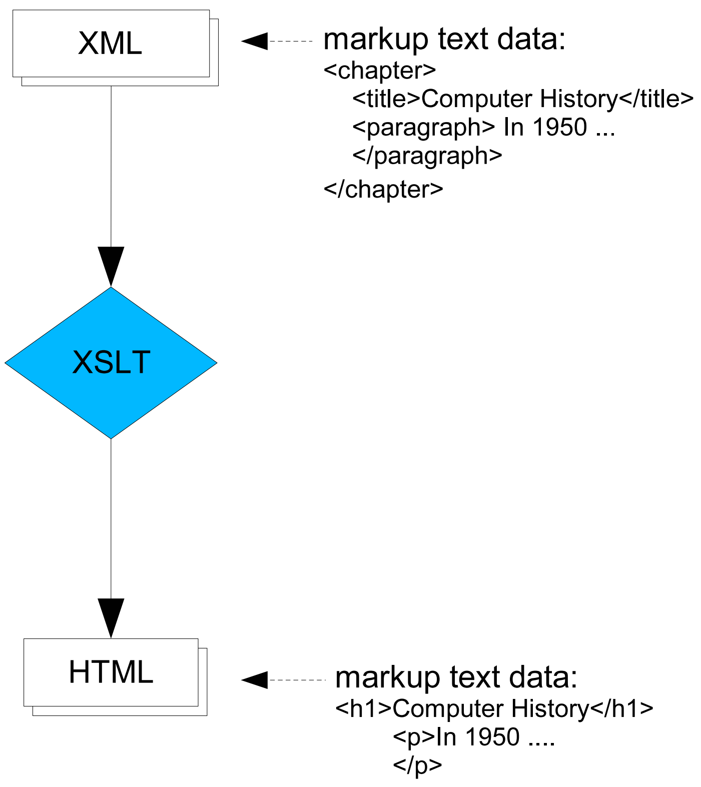


Figure 4.8: Workflow to convert XML to HTML

4.3.2 Windows Help file

HTML Help is the help format introduced by Microsoft with Windows 98. With this format, help files are built from HTML text. The most important disadvantages of HTML Help are that it requires Microsoft Internet Explorer 4 or later, plus the HTML Help ActiveX library. On Windows 95 or NT4, using HTML Help is not recommended. These operating systems do not support HTML Help out of the box, and Windows 95 machines are probably too slow to work with HTML Help comfortably. But HTML Help is also the help system used by Microsoft's .NET framework. [27]

One reason to use HTML Help is that some customers will expect to receive documentation in help file format of Windows, especially when developing software for Windows platforms. However it would also be possible to document any hardware related information (e.g. chip specifications) with these help files.

HTML Help is not a plain HTML format, instead it is Windows' Compiled HTML Help (CHM) format. Therefore you need a special program to turn you HTML pages into '.chm' files, this is the *HTML Help Compiler* (HHC) that comes with Microsoft HTML Help 1.4 SDK[3] Figure 4.9 shows the workflow to turn XML data to CHM format.

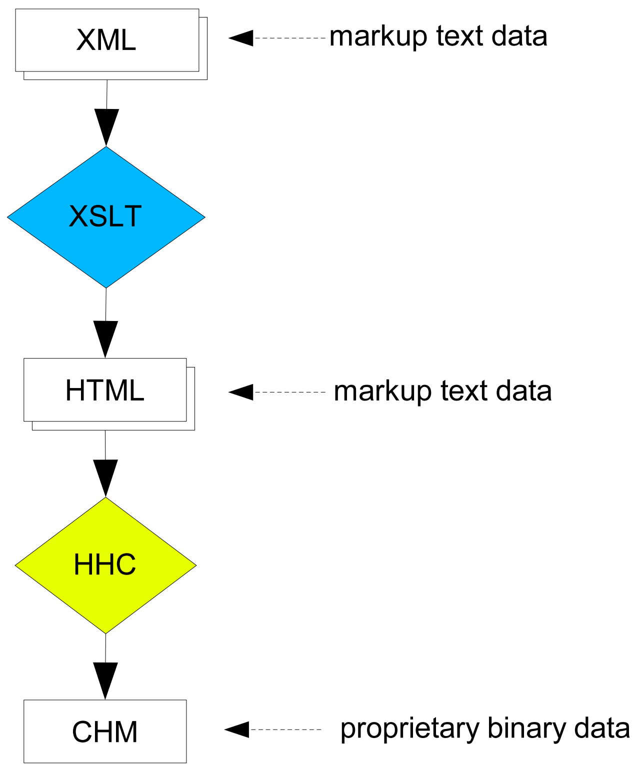
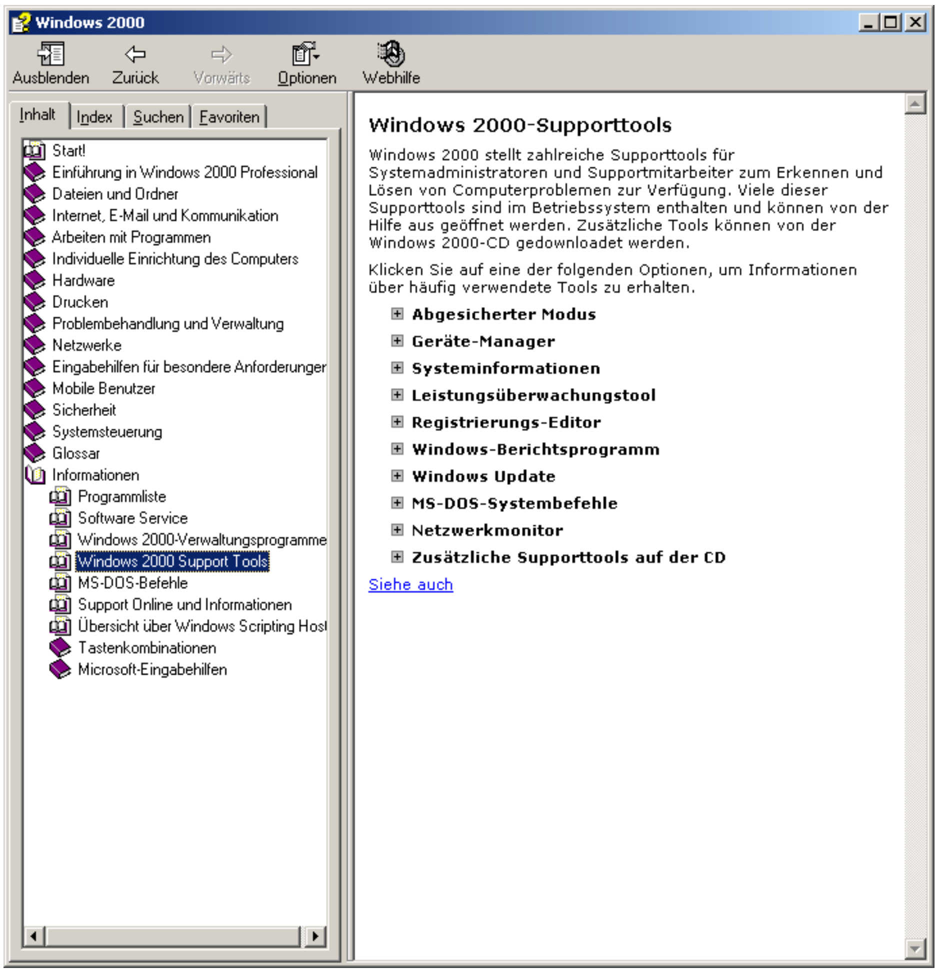


Figure 4.9: Workflow to convert XML to CHM

[3]http://msdn.microsoft.com/library/default.asp?url=/library/en-us/htmlhelp/html/vsconHH1Start.asp

Nevertheless XML can be easily transformed to HTML and the step to compile HTML files to CHM can be automated. And because of the fact that Microsoft Windows and its help files are widely-used, the possibility to transform technical documentation to CHM should always be considered. To give a small impression figure 4.10 shows how HTML Help looks like.


Windows 2000
Ausblenden
Zurück
Vorwärts
Optionen
Webhilfe
Inhalt
Index
Suchen
Favoriten
Start!
Einführung in Windows 2000 Professional
Dateien und Ordner
Internet, E-Mail und Kommunikation
Arbeiten mit Programmen
Individuelle Einrichtung des Computers
Hardware
Drucken
Problembehandlung und Verwaltung
Netzwerke
Eingabehilfen für besondere Anforderunger
Mobile Benutzer
Sicherheit
Systemsteuerung
Glossar
Informationen
Programmliste
Software Service
Windows 2000-Verwaltungsprogramme
Windows 2000 Support Tools
MS-DOS-Befehle
Support Online und Informationen
Übersicht über Windows Scripting Host
Tastenkombinationen
Microsoft-Eingabehilfen

Windows 2000-Supporttools

Windows 2000 stellt zahlreiche Supporttools für Systemadministratoren und Supportmitarbeiter zum Erkennen und Lösen von Computerproblemen zur Verfügung. Viele dieser Supporttools sind im Betriebssystem enthalten und können von der Hilfe aus geöffnet werden. Zusätzliche Tools können von der Windows 2000-CD gedownloadet werden.

Klicken Sie auf eine der folgenden Optionen, um Informationen über häufig verwendete Tools zu erhalten.

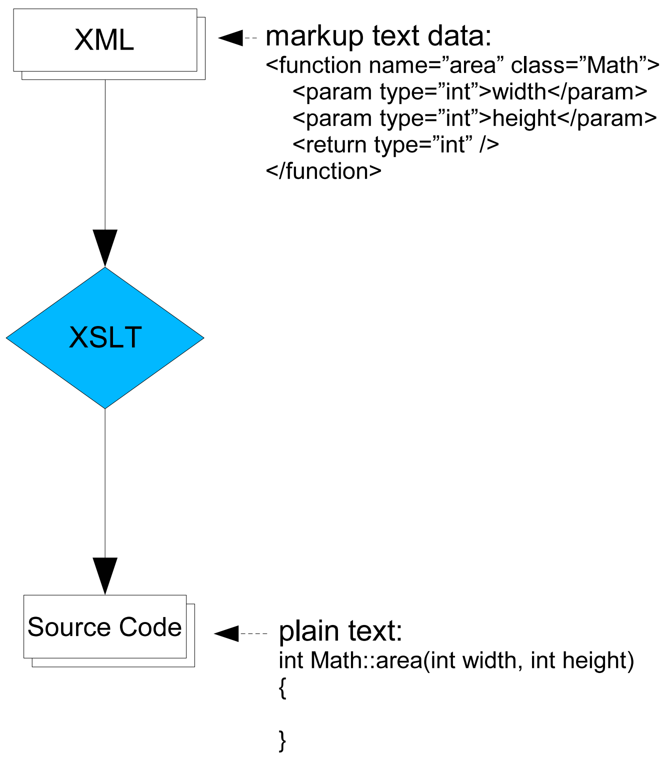
- **Abgesicherter Modus**
- **Geräte-Manager**
- **Systeminformationen**
- **Leistungsüberwachungstool**
- **Registrierungs-Editor**
- **Windows-Berichtsprogramm**
- **Windows Update**
- **MS-DOS-Systembefehle**
- **Netzwerkmonitor**
- **Zusätzliche Supporttools auf der CD**

Siehe auch

Figure 4.10: Windows HTML Help

4.3.3 Source Code

A special case of transforming XML data to plain text would be to generate source code out of an XML document. Of course it will hardly be possible to develop the whole source code of a project from XML. But for example, if specifications for a software are written in XML, these could easily be extended to build header files or function bodies, as shown in figure 4.11. In a large software project a lot of manual work could be automated in this way.



```
<function name="area" class="Math">
   <param type="int">width</param>
   <param type="int">height</param>
   <return type="int" />
</function>
```

```
int Math::area(int width, int height)
{

}
```

Figure 4.11: Workflow to generate Source Code from XML

4.3.4 PDF Rendering

One of the most interesting points is to bring XML structured documents into a nice looking, readable format. To achieve that a technology described in section 2.4 is used, the XSL Formatting Objects. This XML vocabulary represents a page driven layout of your XML data. In the future XSL-FO may be the common language for expressing the output format of XML documents across all kinds of software and hardware, like web browsers or word processors either on a personal computer, printer or even cell phone.

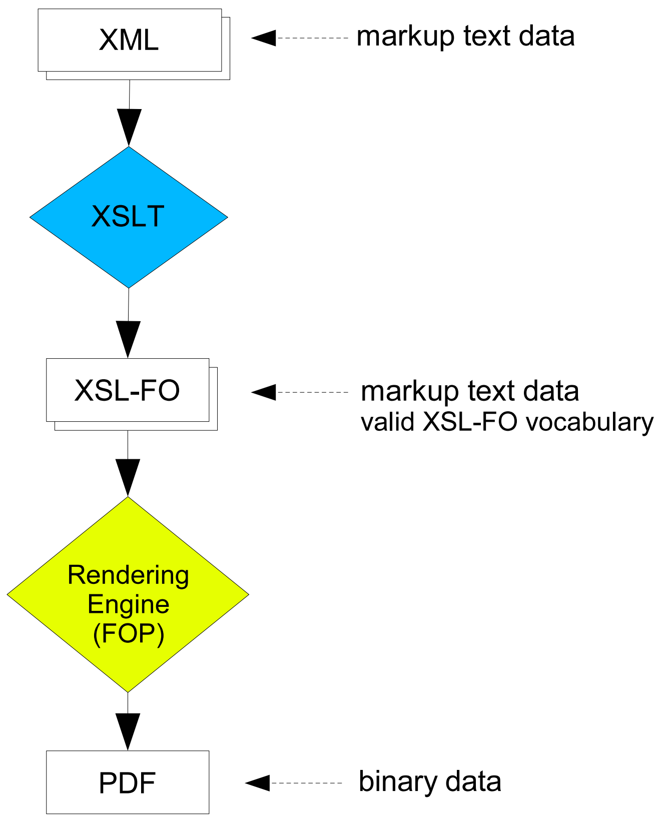


Figure 4.12: Workflow to generate PDF from XML

The first step in this workflow is to transform XML structured data to an XSL-FO valid vocabulary. Therefore an XSLT has to be written that converts the XML vocabulary of the technical document to Formatting Objects and that allows to define the layout for that content to look exactly the way it is specified with the XSLT. The second step in this process is to convert the formatting objects document into some other format that can be viewed onscreen or on paper. This requires running a formatting program, a so called *Formatting Objects Processor* (FOP) or more generally: a rendering engine. This workflow is pictured in figure 4.12.

4.3.4.1 Rendering Engines

XSL-FO engines unfortunately either come as open source programs or feature rich products for purchase, but not both at the same time. There are several programs listed in table 4.1 that provide XSL-FO rendering, some of them comply very well with the XSL-FO W3C Recommendation, others do not. They also differ in the set of features they provide. Therefore an evaluation of available FO processors is recommended, based on the initial situation and the requirements the engine has to fulfill. In the showcase provided with this thesis, an evaluation version of the Apoc XSL-FO processor from Chive Software has been used, due to the fact that it supports a good .NET integration.

4.3.4.2 Limitations of XSL-FO

Although XSL-FO is able to satisfy almost all needs for publishing technical documentation, there are certain limitations that cannot be overcome with the current Formatting Objects vocabulary. In fact some rendering engine implementations like the Epic Editor from Arbortext provide extensions to remove some of the boundaries of XSL-FO.

Eliot Kimber of ISOGEN offers a detailed paper about the drawbacks of using XSL-FO for technical documentation, some of the most important limitations are listed here, for the full list please refer to his paper [28].

The following requirements are currently not fulfilled in XSL-FO:

- Multiple, independent, multi-page flowed areas within a single page body (for example two independent articles on the same page that continue to different pages). There are no known proprietary extensions for satisfying this requirement.

- Text that flows around arbitrary curved areas (but text flowing around rectangular areas is possible using side floats). There are no extensions that

Vendor	**Product**	**Remarks**
3B2 www.3b2.com	3B2-FO	available on multiple operating systems
Antenna House www.antennahouse.com	XSL Formatter	conforms to XSL-FO v1.0 W3C Recommendation
Apache project xml.apache.org	FOP	open source, java based
Arbortext www.arbortext.com	Epic Editor	generates PDF and PostScript
Chive Products www.chive.com	Apoc XSL-FO	.NET integration
IBM www.alphaworks.ibm.com/tech/xfc	XSL Formatting Objects Composer	partial implementation of XSL-FO v1.0 W3C Recommendation
RenderX www.renderx.com	XEP Rendering Engine	java based, supports a subset of SVG
XyEnterprise www.xyenterprise.com	XPP	available on multiple operating systems

Table 4.1: Several available FO rendering engines

satisfy this requirement.

- Page-location sensitive inclusion or exclusion of content. For example, there is no direct way to condition the text of a cross reference based on whether or not the target of the reference occurs on the same page as the reference itself. There are no extensions that satisfy this requirement.

- Any other presentation tuning semantics that require feedback from the pagination step to the initial FO generation step. But note that this feedback could be provided in implementation-specific ways, enabling a multi-pass process in which second and subsequent FO generation instances would have access to information about what page a given input node occurred on in the previous pagination instance. None of the current products provide this feature.

- Dynamic, page-sensitive information in figure captions, table captions, and table headers (although this can be faked using floats or tricks with page headers or footers in some cases). Epic Publisher provides an extension for before floats that can be used to satisfy this requirement.

- Composition of back-of-the-book indexes such that lists of page number citations are reduced to unique page numbers, as well as the automatic creation of page ranges.

- Rotation of flowed text at angles other than multiples of 90 degrees. The FO specification only provides for rotation in increments of 90 degrees. There are no extensions for other rotation options. However, embedded SVG can be used with FOP and XSL Formatter to create a wide range of text effects.

- Creation of PDF bookmarks, links, and annotations. While PDF is only a defacto standard, it is almost universally used for online delivery of print-quality documents. PDF includes a number of useful features for online display, including navigation bookmarks and hyperlinks. The FO specification does not provide any direct facilities from which these PDF artifacts could be derived. However, all the FO implementations provide extensions for creating bookmarks, links, and other PDF specific, online artifacts.

Chapter 5

Showcase Portal

One of the main targets of this thesis was the development of a showcase that demonstrates how XML-based technical documentation could be realized in the near future. The features of server-side transformations and PDF generation were also a main part of this work, always considering the aspects of single sourcing.

5.1 Design

After collecting all requirements for this showcase the design was set up. Due to the fact that many authors should work in a single sourcing environment that should enable them to re-use and assemble their documents out of XML components, the decision to implement the whole showcase as a web application was made. Figure 5.1 shows the units that this showcase is built of.

The validation part is used to validate XML documents against DTDs or XML Schema, if none of these is declared inside the file it is checked at least for well-formedness. DTDs and Schemata are stored inside the portal, as well as the transformation files. But they should not be edited or extended by the authors themselves. Instead some sort of management level should ensure the consistency of the XML documents inside the working environment. This management should also adapt the XSL files if something is changed inside the XML vocabulary that is currently used.

The transformations are used in many different ways and provide very versatile functionality, they can be used to publish technical documents or generate source

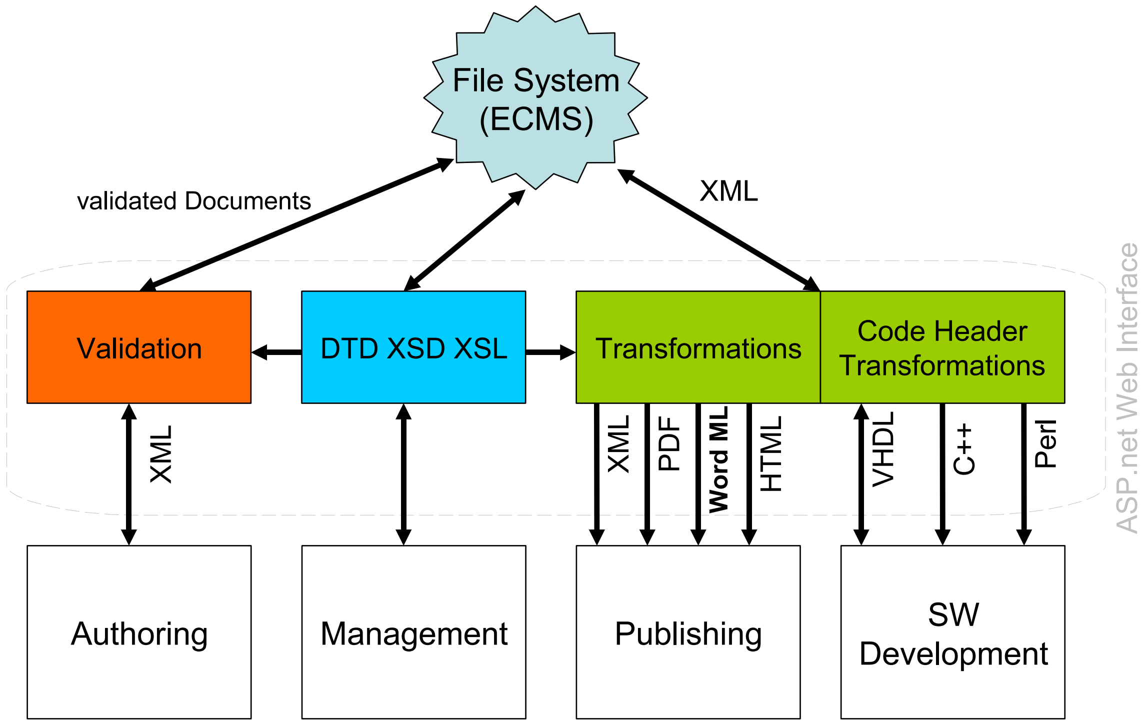


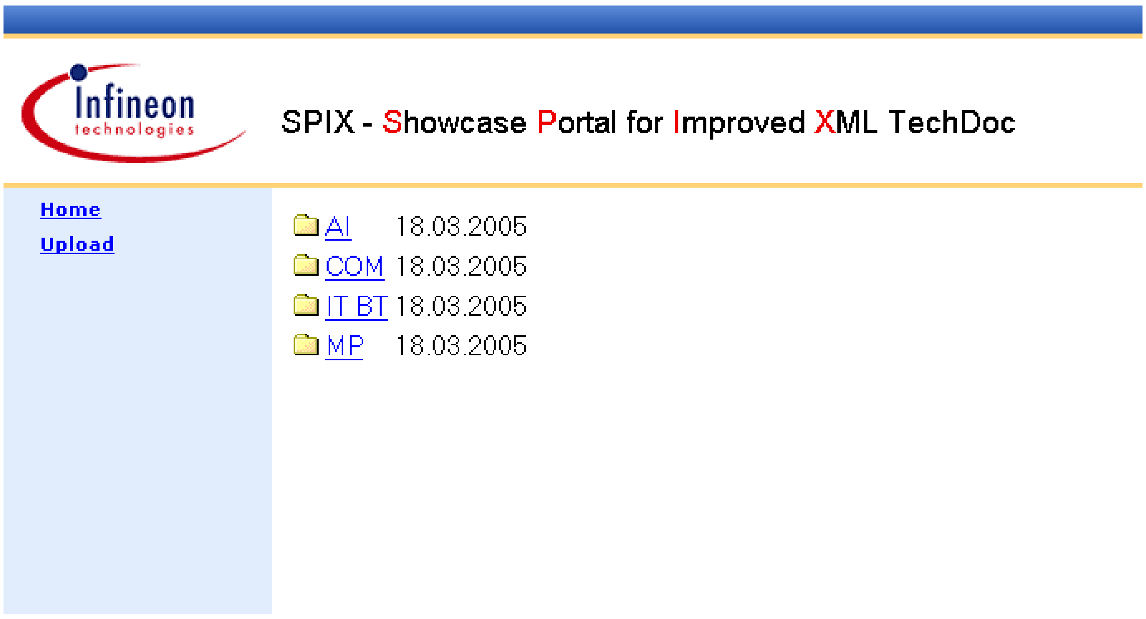
Figure 5.1: Structure of the showcase

code for different language types, but they are also used for previewing written content in a quick and easy way.

5.2 Functional Overview

This section gives a short overview of how the information inside the portal should be structured to achieve good results. First of all a high-level hierarchy is a good starting point, this is mainly done to categorize the content inside the portal in a certain manner. In this example - shown in figure 5.2 - the division on the root level has been done according to the distribution of Infineons' Business Units (BU). Although this division does not prevent the use of content modules across different BUs.

It is an advantage to keep a common working environment one level below the root. As pictured in figure 5.3 the directory structure on this level supports a clean



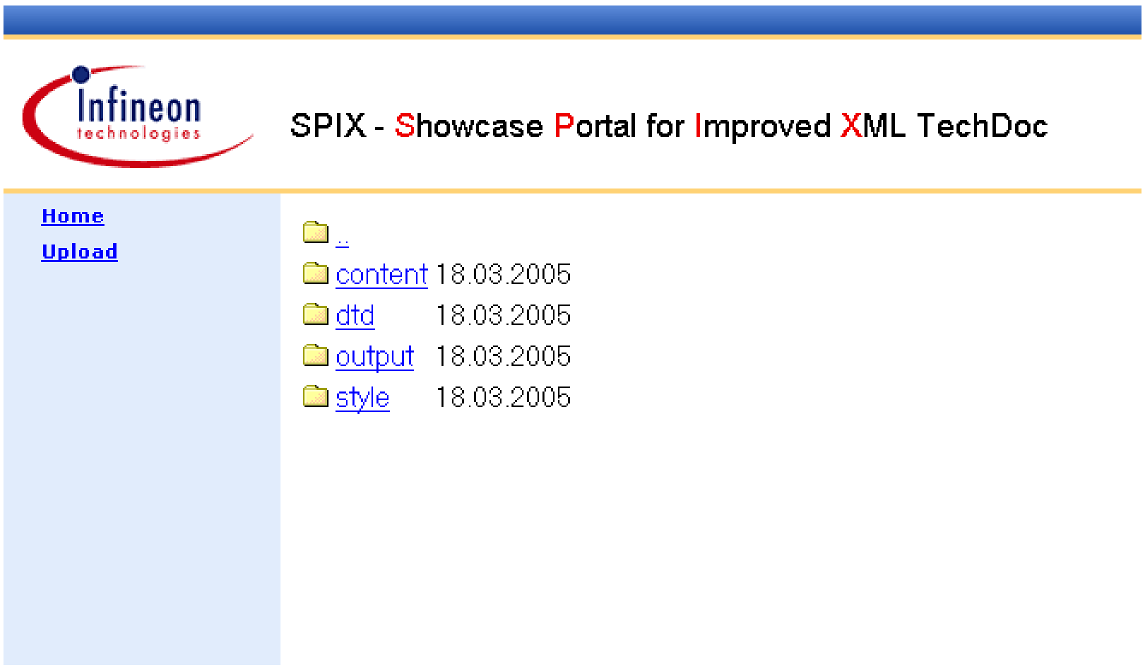
AI	18.03.2005
COM	18.03.2005
IT BT	18.03.2005
MP	18.03.2005

Figure 5.2: Division according to Business Units

XML-based environment. It is recommended to use such a directory configuration for efficient work with this showcase.

Directories described in detail:

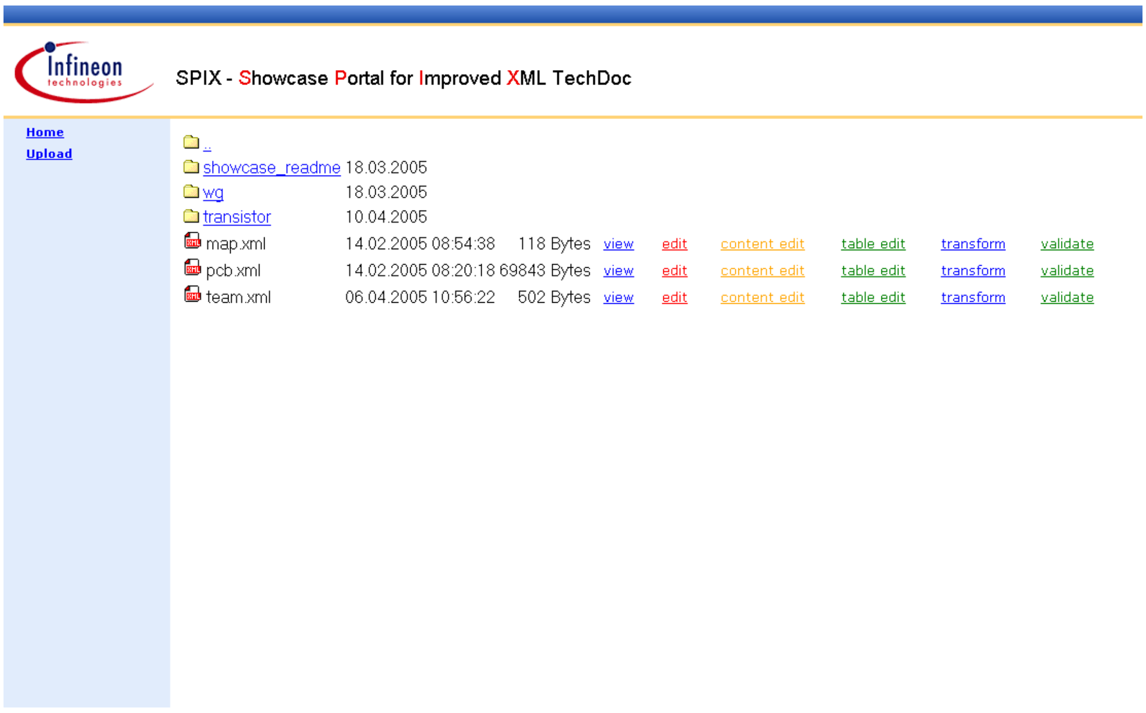
- **content**
  contains only pure XML content modules
- **dtd**
  contains all structural information describing the XML modules, files in here can either be referenced by relative path or by URI
- **output**
  contains all files that are generated as a result of XSL transformations
- **style**
  contains all transformation files (XSLTs) together with Cascading Style Sheets (CSS)


SPIX - Showcase Portal for Improved XML TechDoc

```
..
content 18.03.2005
dtd     18.03.2005
output  18.03.2005
style   18.03.2005
```

Figure 5.3: Directory structure for a XML-based environment

The main work will be done inside the content directory, this is also where most of the functionality on XML files is carried out. Figure 5.4 points this out. Each of these feature will be treated throughout the next sections.

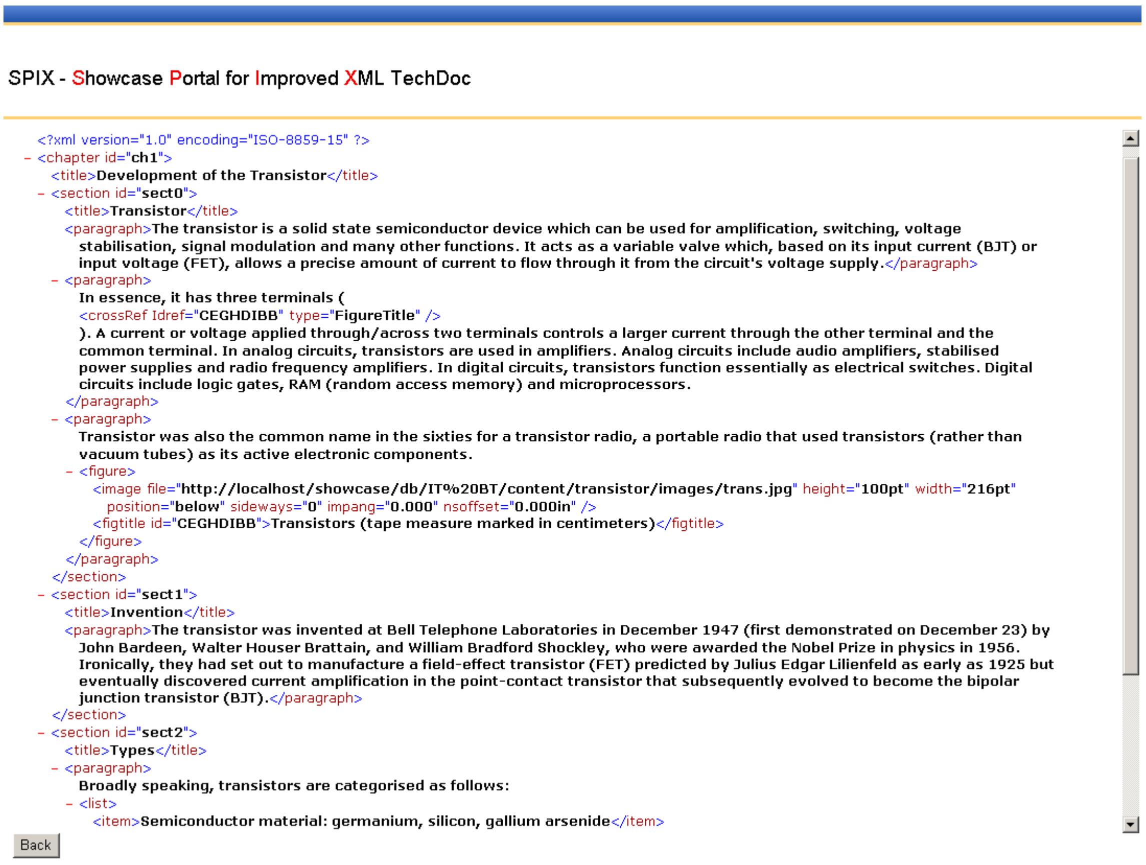

SPIX - Showcase Portal for Improved XML TechDoc

Home
Upload

..								
showcase_readme	18.03.2005							
wg	18.03.2005							
transistor	10.04.2005							
map.xml	14.02.2005 08:54:38	118 Bytes	view	edit	content edit	table edit	transform	validate
pcb.xml	14.02.2005 08:20:18	69843 Bytes	view	edit	content edit	table edit	transform	validate
team.xml	06.04.2005 10:56:22	502 Bytes	view	edit	content edit	table edit	transform	validate

Figure 5.4: Features on XML files

For opening up a XML file in view mode, which supports syntax highlighting, use the 'view' button beside that file. The XML file will then be displayed as shown in figure 5.5.



```
  <?xml version="1.0" encoding="ISO-8859-15" ?>
- <chapter id="ch1">
    <title>Development of the Transistor</title>
  - <section id="sect0">
      <title>Transistor</title>
      <paragraph>The transistor is a solid state semiconductor device which can be used for amplification, switching, voltage
        stabilisation, signal modulation and many other functions. It acts as a variable valve which, based on its input current (BJT) or
        input voltage (FET), allows a precise amount of current to flow through it from the circuit's voltage supply.</paragraph>
    - <paragraph>
        In essence, it has three terminals (
        <crossRef Idref="CEGHDIBB" type="FigureTitle" />
        ). A current or voltage applied through/across two terminals controls a larger current through the other terminal and the
        common terminal. In analog circuits, transistors are used in amplifiers. Analog circuits include audio amplifiers, stabilised
        power supplies and radio frequency amplifiers. In digital circuits, transistors function essentially as electrical switches. Digital
        circuits include logic gates, RAM (random access memory) and microprocessors.
      </paragraph>
    - <paragraph>
        Transistor was also the common name in the sixties for a transistor radio, a portable radio that used transistors (rather than
        vacuum tubes) as its active electronic components.
      - <figure>
          <image file="http://localhost/showcase/db/IT%20BT/content/transistor/images/trans.jpg" height="100pt" width="216pt"
            position="below" sideways="0" impang="0.000" nsoffset="0.000in" />
          <figtitle id="CEGHDIBB">Transistors (tape measure marked in centimeters)</figtitle>
        </figure>
      </paragraph>
    </section>
  - <section id="sect1">
      <title>Invention</title>
      <paragraph>The transistor was invented at Bell Telephone Laboratories in December 1947 (first demonstrated on December 23) by
        John Bardeen, Walter Houser Brattain, and William Bradford Shockley, who were awarded the Nobel Prize in physics in 1956.
        Ironically, they had set out to manufacture a field-effect transistor (FET) predicted by Julius Edgar Lilienfeld as early as 1925 but
        eventually discovered current amplification in the point-contact transistor that subsequently evolved to become the bipolar
        junction transistor (BJT).</paragraph>
    </section>
  - <section id="sect2">
      <title>Types</title>
    - <paragraph>
        Broadly speaking, transistors are categorised as follows:
      - <list>
          <item>Semiconductor material: germanium, silicon, gallium arsenide</item>
```

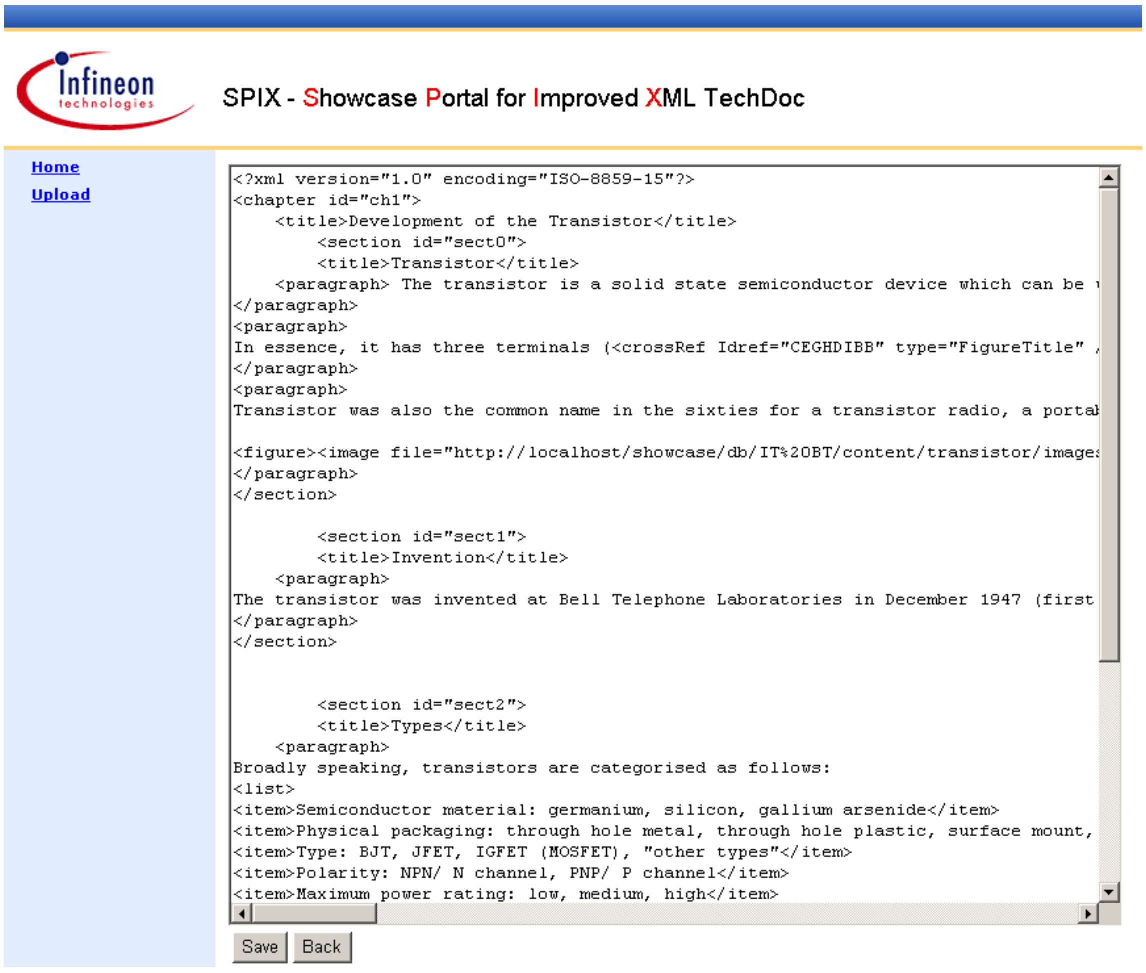
Figure 5.5: Viewing mode of a XML file

5.3 Online Editing

To enable XML-based technical documentation on a web application the possibility to edit content online has to be provided. This sounds easy but in fact it is not, because of the limited capabilities web browsers offer to enter information. The first idea was to provide a editing mode with syntax highlighting for XML. Some research on this topic has been made throughout the design phase of this showcase. Online XML editing is not that common yet, therefore only a small number of tools supporting such a functionality is available. The decision was made to try a java-script based open source program called ”helene”[1].

[1] http://helene.muze.nl/

But after a short time of testing it turned out that the performance with large XML files was too bad to work with that tool efficiently. This was the reason to switch back to a normal text-box based editing mode without syntax highlighting, as shown in figure 5.6.

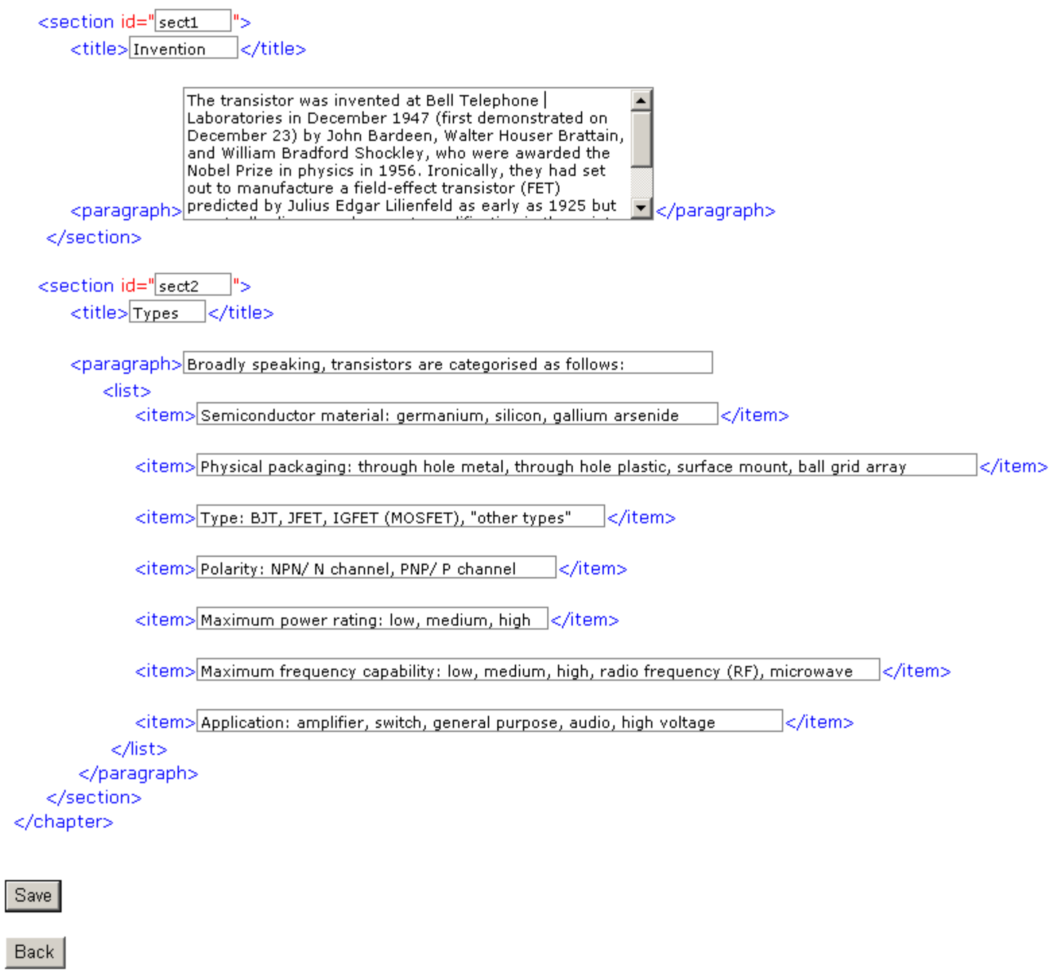


```
<?xml version="1.0" encoding="ISO-8859-15"?>
<chapter id="ch1">
    <title>Development of the Transistor</title>
        <section id="sect0">
        <title>Transistor</title>
    <paragraph> The transistor is a solid state semiconductor device which can be
</paragraph>
<paragraph>
In essence, it has three terminals (<crossRef Idref="CEGHDIBB" type="FigureTitle"
</paragraph>
<paragraph>
Transistor was also the common name in the sixties for a transistor radio, a portal

<figure><image file="http://localhost/showcase/db/IT%20BT/content/transistor/image
</paragraph>
</section>

        <section id="sect1">
        <title>Invention</title>
    <paragraph>
The transistor was invented at Bell Telephone Laboratories in December 1947 (first
</paragraph>
</section>


        <section id="sect2">
        <title>Types</title>
    <paragraph>
Broadly speaking, transistors are categorised as follows:
<list>
<item>Semiconductor material: germanium, silicon, gallium arsenide</item>
<item>Physical packaging: through hole metal, through hole plastic, surface mount,
<item>Type: BJT, JFET, IGFET (MOSFET), "other types"</item>
<item>Polarity: NPN/ N channel, PNP/ P channel</item>
<item>Maximum power rating: low, medium, high</item>
```

Figure 5.6: Online editing mode

5.4 Content Editing

To provide other forms of XML editing, the implementation of a so-called content editing mode was another goal. This specific type of editing mode has two major advantages. First of all it removes the annoying task of programming web forms over and over again for each application, only to allow the user to enter information that is afterwards stored in a database or transformed into XML again. The second advantage of content editing mode - that figure 5.7 shows an example of - is that the user can enter data directly into a XML document, without being able to modify or even destroy the structure of the document. Only the content itself is editable.



```
  <section id="sect1">
      <title>Invention</title>

                The transistor was invented at Bell Telephone
                Laboratories in December 1947 (first demonstrated on
                December 23) by John Bardeen, Walter Houser Brattain,
                and William Bradford Shockley, who were awarded the
                Nobel Prize in physics in 1956. Ironically, they had set
                out to manufacture a field-effect transistor (FET)
      <paragraph>predicted by Julius Edgar Lilienfeld as early as 1925 but</paragraph>
   </section>

  <section id="sect2">
      <title>Types</title>

      <paragraph>Broadly speaking, transistors are categorised as follows:
          <list>
              <item>Semiconductor material: germanium, silicon, gallium arsenide</item>

              <item>Physical packaging: through hole metal, through hole plastic, surface mount, ball grid array</item>

              <item>Type: BJT, JFET, IGFET (MOSFET), "other types"</item>

              <item>Polarity: NPN/ N channel, PNP/ P channel</item>

              <item>Maximum power rating: low, medium, high</item>

              <item>Maximum frequency capability: low, medium, high, radio frequency (RF), microwave</item>

              <item>Application: amplifier, switch, general purpose, audio, high voltage</item>
           </list>
       </paragraph>
   </section>
</chapter>
```

Figure 5.7: Content editing mode

The user is able to call this function on any XML file stored inside the portal. Afterwards the file is parsed and a web form is created that displays the markup of the XML file in a syntax highlighted manner with correct indention. Only the values of attributes and the elements' content is replaced with text-boxes, containing the particular data. With this technique it is not important if the user has any knowledge about the structural information of a XML file. The user is now able to enter or edit content inside a XML document and does not have to take care of any markup data. And by pressing the save button, the changed data is stored directly into the XML file, guaranteeing that the structural information is preserved.

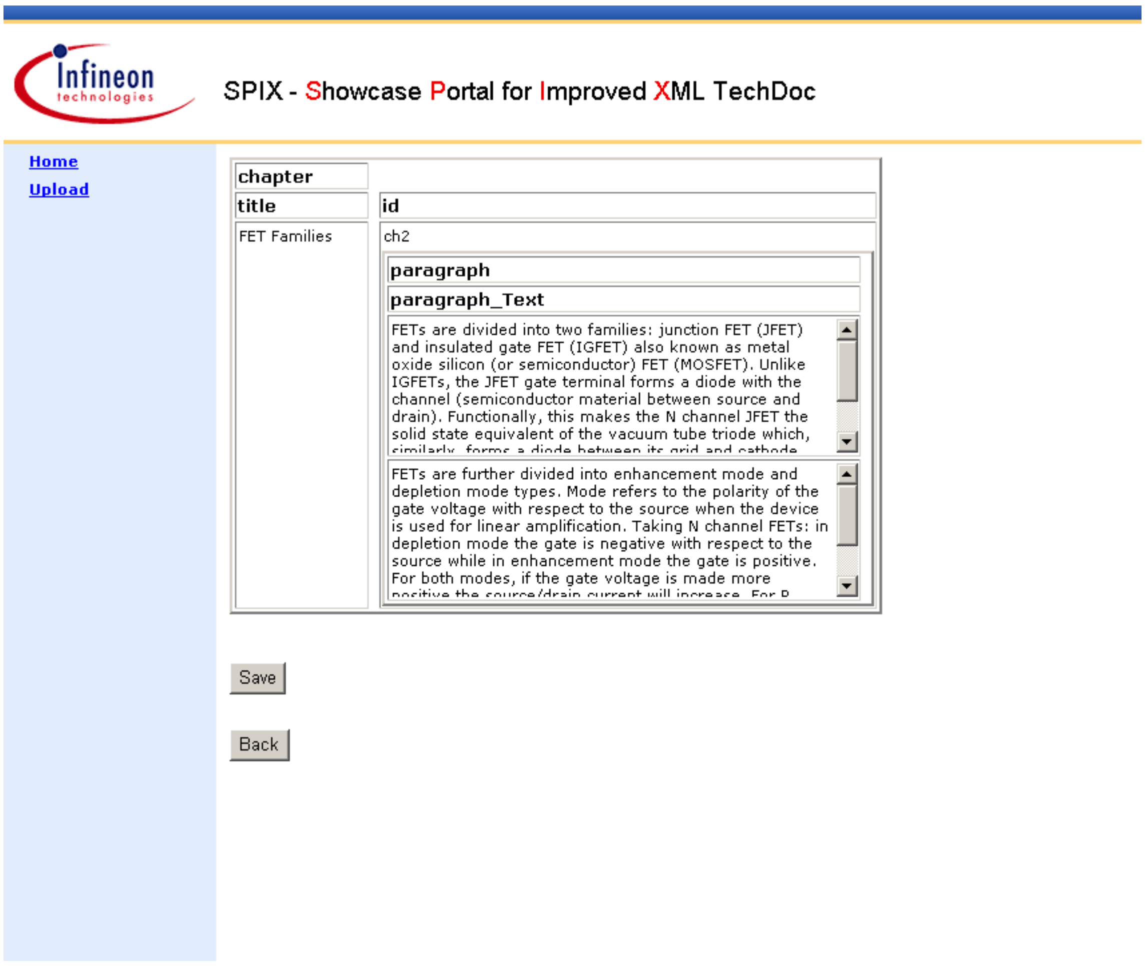
One thing has to be taken in consideration: the author of the XML document has to take care of the semantic meaning of the markup very carefully. Due to the fact the user enters information into this document in content editing mode, only the markup tags will give a clue which information has to be entered into a specific field.

5.5 Tabular Editing

Tabular editing gives user and author another method to enter or modify the content of a XML document. Tabular editing is based on the technology of treemaps. The term treemap describes the notion of turning a tree into a planar space-filling map [29]. These treemaps are able to represent a tree structured information (this applies to any well-formed XML document) as nested areas.

When calling the 'tabular editing' function on a XML file inside the portal, this file is parsed and the XML structure is stored internally. The storage takes place inside the servers memory and the structure of the file is represented as linked tables. Afterwards a procedure is called on these linked tables that runs recursively through them and builds a web form consisting of nested HTML tables.

As described in section 5.4 attribute and element values are also editable, figure 5.8 pictures that. Again, when saving this web form the information is stored directly back into the XML file. Due to some limitations in the ASP.NET XML support this editing mode cannot cope with XML documents that uses same elements in different depths of a tree structure.


SPIX - Showcase Portal for Improved XML TechDoc

chapter	
title	id
FET Families	ch2

paragraph

paragraph_Text

FETs are divided into two families: junction FET (JFET) and insulated gate FET (IGFET) also known as metal oxide silicon (or semiconductor) FET (MOSFET). Unlike IGFETs, the JFET gate terminal forms a diode with the channel (semiconductor material between source and drain). Functionally, this makes the N channel JFET the solid state equivalent of the vacuum tube triode which, similarly, forms a diode between its grid and cathode

FETs are further divided into enhancement mode and depletion mode types. Mode refers to the polarity of the gate voltage with respect to the source when the device is used for linear amplification. Taking N channel FETs: in depletion mode the gate is negative with respect to the source while in enhancement mode the gate is positive. For both modes, if the gate voltage is made more positive the source/drain current will increase. For P

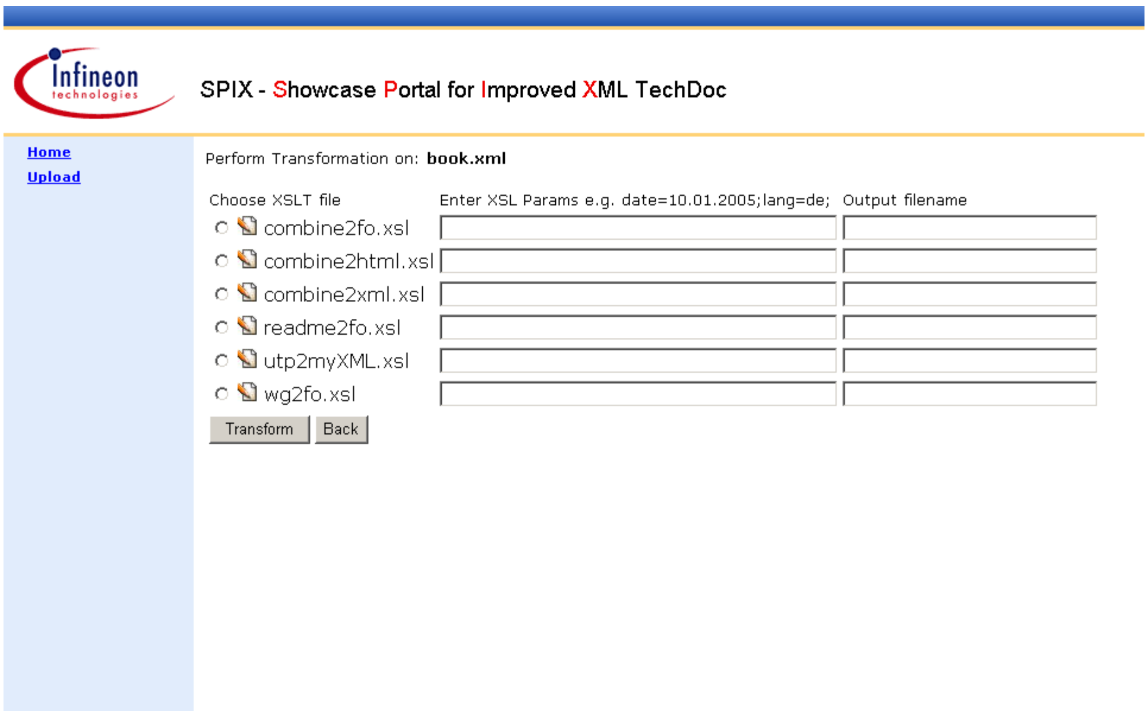
Figure 5.8: Tabular editing mode

5.6 XSL Transformations

Another feature of this showcase is that it provides server-side XSL transformations. Virtually any kind of transformation can be used on XML files. As stated in section 4.3, many different output types of these transformations are possible, the result can be either a XML formatted document or something completely different.

This showcase also supports stylesheet parameters to alter the output of transformations in different ways. One of the possibilities of stylesheet parameters will be shown in section 5.7.1. However there are many other different things that can be achieved with these parameters like: conditional text inside a document, any kind of sort criteria can be applied to the document or actual dates or names can be passed into the document. With the help of XSL transformations the user is also able to generate tables of content and dissolve cross references. A full listing of a transformation providing this functionality can be found in appendix B.

A transformation can be applied the way it is shown in figure 5.9. First of all the XSLT that should be used, has to be selected with a radio button. Afterwards the stylesheet parameters and the output filename can be determined. If no output filename is specified then the filename of the input file is used.



Perform Transformation on: **book.xml**

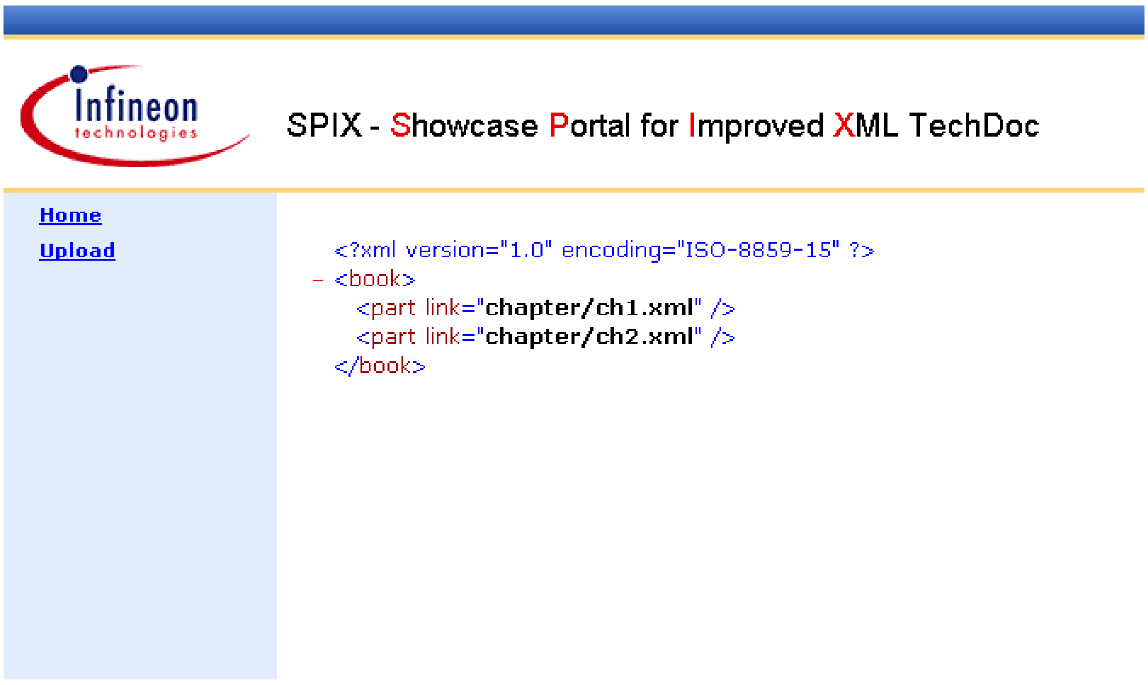
Choose XSLT file	Enter XSL Params e.g. date=10.01.2005;lang=de;	Output filename
○ combine2fo.xsl		
○ combine2html.xsl		
○ combine2xml.xsl		
○ readme2fo.xsl		
○ utp2myXML.xsl		
○ wg2fo.xsl		

Figure 5.9: Applying a transformation on a XML file

For the server-side transformations the XSL processor MSXML, that is integrated into the .Net environment, has been used. Any concerns on the performance of server-side transformations could not be confirmed. The performance mainly depends on the amount of data that has to be transferred between client and server.

5.7 PDF Generation

PDF Generation is a two step process, as already stated in section 4.3.4. During the first step a XSL transformation is applied to convert the XML document to a XSL-FO compliant vocabulary. In this example 'book.xml' displayed in figure 5.10, is used as an assembly file. What an assembly file should look like and how it works, can be read in section 4.2.4. When applying the XSL transformation on 'book.xml', it combines two XML components, in this case chapter I and chapter II, to a complete XSL-FO book.



```
  <?xml version="1.0" encoding="ISO-8859-15" ?>
- <book>
    <part link="chapter/ch1.xml" />
    <part link="chapter/ch2.xml" />
  </book>
```

Figure 5.10: Example of an assembly file

In the second step this XSL-FO file has to be rendered into a PDF. Figure 5.11 shows a 'gen.pdf' button next to the 'book.fo' file that has been generated as a result of the XSL transformation. By clicking on that button the processing engine is started and the file is rendered into a PDF document.

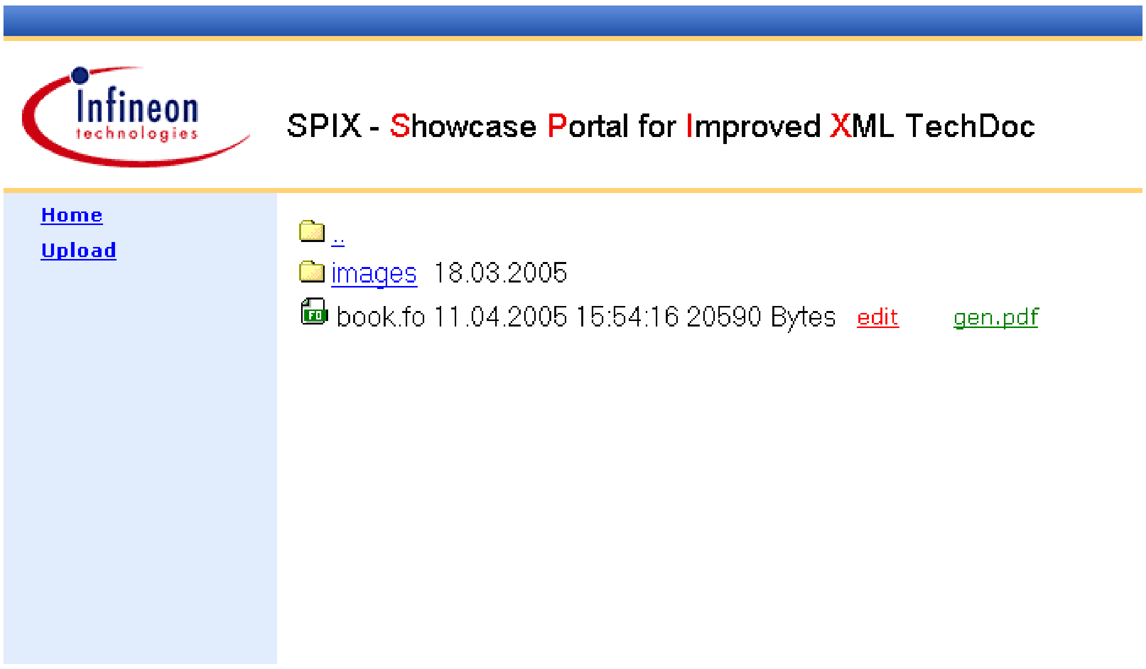

Infineon technologies
SPIX - Showcase Portal for Improved XML TechDoc
Home
Upload
..
images 18.03.2005
book.fo 11.04.2005 15:54:16 20590 Bytes edit gen.pdf

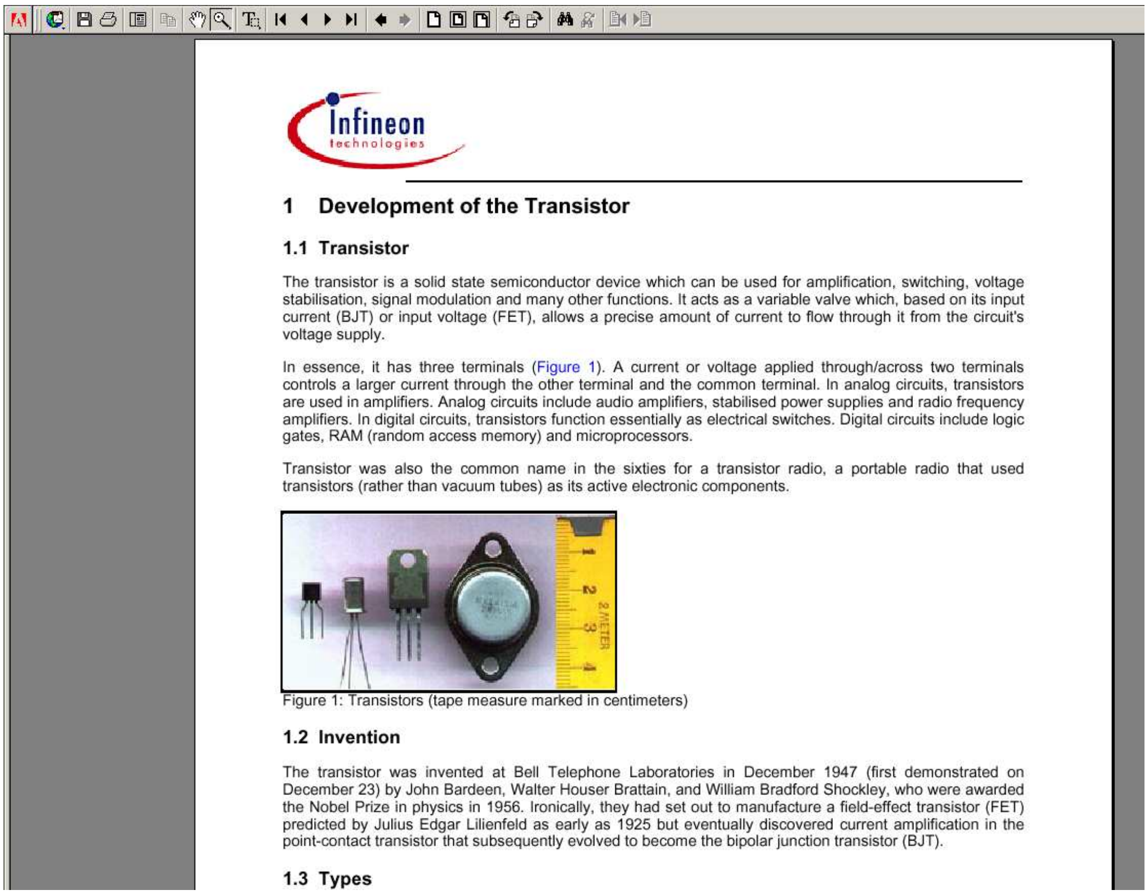
Figure 5.11: Rendering a PDF from a FO file

After the rendering process is done, the final PDF looks like as it is shown in figure 5.12. To illustrate this example it is filled with sample data about the history of transistors. Nevertheless the book contains a table of contents now and all the cross references were dissolved. However this should not distract from the fact that many different document types, like those listed in section 3.2 can be generated and published with the help of this workflow.

5.7.1 Linking to Source Documents

To enable the process of reviewing documents, without sending them back and forth per mail, a feature has been implemented to generate a sort of preview PDF document. It looks exactly the same way as the PDF document that is generated for publishing, but in addition it provides ’edit source’ links on the right hand side of the document border. When clicking on one of these links the source document that provides the content for this part of the PDF document is opened right away for editing in the showcase portal.

This feature really supports the idea of single sourcing, because with one click



1 Development of the Transistor

1.1 Transistor

The transistor is a solid state semiconductor device which can be used for amplification, switching, voltage stabilisation, signal modulation and many other functions. It acts as a variable valve which, based on its input current (BJT) or input voltage (FET), allows a precise amount of current to flow through it from the circuit's voltage supply.

In essence, it has three terminals (Figure 1). A current or voltage applied through/across two terminals controls a larger current through the other terminal and the common terminal. In analog circuits, transistors are used in amplifiers. Analog circuits include audio amplifiers, stabilised power supplies and radio frequency amplifiers. In digital circuits, transistors function essentially as electrical switches. Digital circuits include logic gates, RAM (random access memory) and microprocessors.

Transistor was also the common name in the sixties for a transistor radio, a portable radio that used transistors (rather than vacuum tubes) as its active electronic components.

Figure 1: Transistors (tape measure marked in centimeters)

1.2 Invention

The transistor was invented at Bell Telephone Laboratories in December 1947 (first demonstrated on December 23) by John Bardeen, Walter Houser Brattain, and William Bradford Shockley, who were awarded the Nobel Prize in physics in 1956. Ironically, they had set out to manufacture a field-effect transistor (FET) predicted by Julius Edgar Lilienfeld as early as 1925 but eventually discovered current amplification in the point-contact transistor that subsequently evolved to become the bipolar junction transistor (BJT).

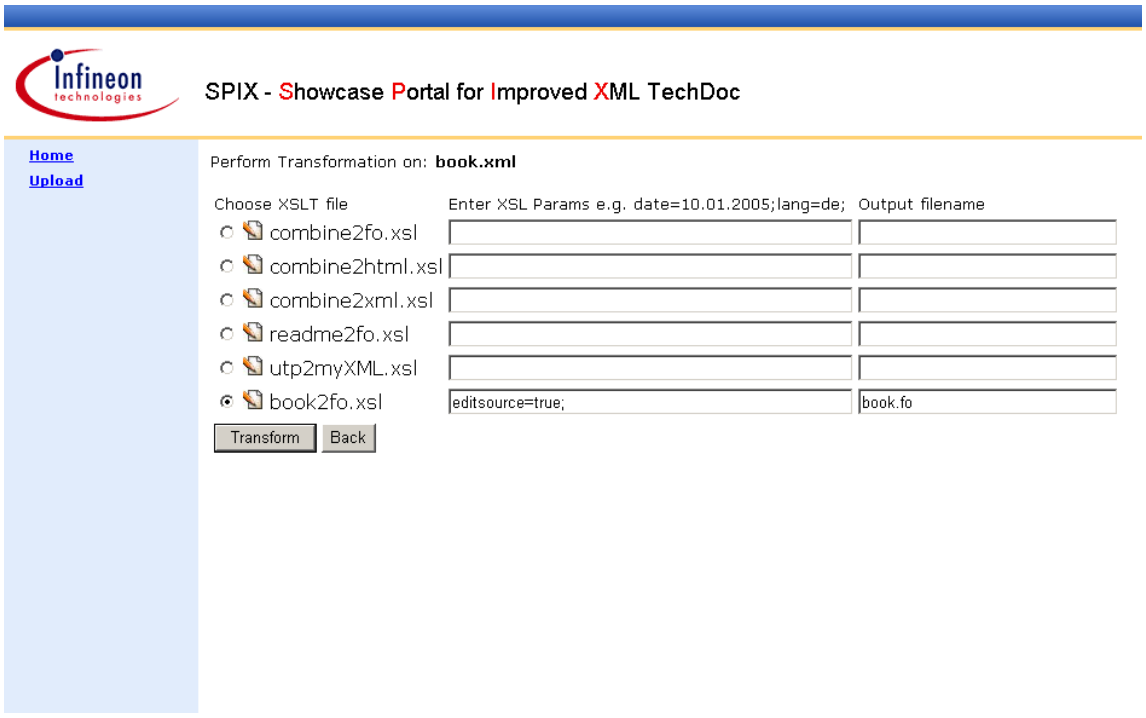
1.3 Types

Figure 5.12: Final 'book.pdf' generated completely out of XML data

in a document it is possible to jump directly to the source that is behind that document and by changing that source, all documents that derive from that source also inherit the changes (after the transformation to the desired output format is reprocessed).

To activate the 'edit source' links, the stylesheet parameter 'editsource' has to be set to 'true' before the XSL transformation is started. How this must look like is pictured in figure 5.13.

The result, after the transformation is done and the rendering is completed, can be seen in figure 5.14. Now there is a 'edit source' link beside each chapter heading, by clicking on one of those links the according XML component is opened in



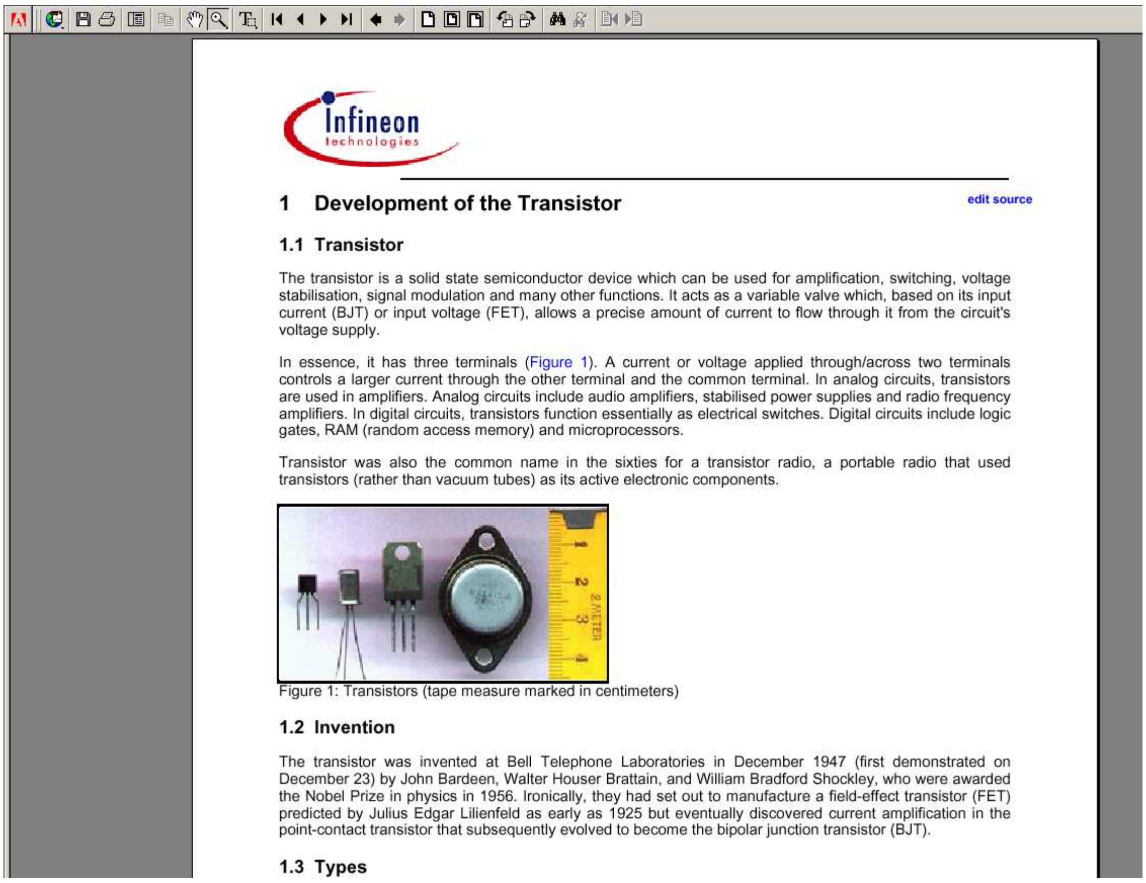
SPIX - Showcase Portal for Improved XML TechDoc

Perform Transformation on: **book.xml**

Choose XSLT file	Enter XSL Params e.g. date=10.01.2005;lang=de;	Output filename
○ combine2fo.xsl		
○ combine2html.xsl		
○ combine2xml.xsl		
○ readme2fo.xsl		
○ utp2myXML.xsl		
● book2fo.xsl	editsource=true;	book.fo

Figure 5.13: Activate the 'edit source' functionality

the online editing mode inside the portal. With a single click, it is now possible to jump to the source that lies behind the PDF document.



1 Development of the Transistor

1.1 Transistor

The transistor is a solid state semiconductor device which can be used for amplification, switching, voltage stabilisation, signal modulation and many other functions. It acts as a variable valve which, based on its input current (BJT) or input voltage (FET), allows a precise amount of current to flow through it from the circuit's voltage supply.

In essence, it has three terminals (Figure 1). A current or voltage applied through/across two terminals controls a larger current through the other terminal and the common terminal. In analog circuits, transistors are used in amplifiers. Analog circuits include audio amplifiers, stabilised power supplies and radio frequency amplifiers. In digital circuits, transistors function essentially as electrical switches. Digital circuits include logic gates, RAM (random access memory) and microprocessors.

Transistor was also the common name in the sixties for a transistor radio, a portable radio that used transistors (rather than vacuum tubes) as its active electronic components.

Figure 1: Transistors (tape measure marked in centimeters)

1.2 Invention

The transistor was invented at Bell Telephone Laboratories in December 1947 (first demonstrated on December 23) by John Bardeen, Walter Houser Brattain, and William Bradford Shockley, who were awarded the Nobel Prize in physics in 1956. Ironically, they had set out to manufacture a field-effect transistor (FET) predicted by Julius Edgar Lilienfeld as early as 1925 but eventually discovered current amplification in the point-contact transistor that subsequently evolved to become the bipolar junction transistor (BJT).

1.3 Types

Figure 5.14: An ’editable’ PDF document

Chapter 6

Conclusions

The final chapter gives an overview of the issues included in this document and covers some impressions the author gained during his work on this thesis.

6.1 General Aspects

The introduction of XML can have a very positive effect on a company's documentation process. However, as with any application, XML does not by itself offer a key for a successful technical documentation. ”It is only a standard on which solutions can be built. The basis of these solutions will only be realized when, along with leveraging XML technology in its different forms, the challenges currently being faced are taken into consideration.” [30]

One of the main advantages is the separation of content, structure and layout, this enables a clear assignment to the corresponding authoring, maintenance and publishing persons in charge. An easier content creation process in association with the automated layout and conversion into multiple formats is leading to a faster process and additionally can save costs. Furthermore, using XML to semantically markup documents guarantees the reuse of the information stored inside.

The discussion with authors and other responsibles working in technical documentation has been proven to be the most important basis for gathering information and requirements. Only after analyzing the requirements and considering the current problems of technical documentation the showcase could be built in a way that it offers new, improved possibilities in technical documentation.

XSLT and XSL-FO turned out to be very effective technologies for use in technical documentation. XSL transformations provide good flexibility and full control over the conversion of XML documents to other formats. Formatting Objects already is a good starting point for layout driven formatting and when some of the limitations are overcome with further development, it will become even more important.

The decision to implement this showcase as a web application was not an easy one, there were some difficulties that could not be overcome, like the syntax highlighting in editing mode. However in the long run the advantages will outweigh the disadvantages and although this showcase is far away from being a competitive product, it fulfilled its needs and gave a good impression of how technical documentation could be done in the near future.

6.2 Personal Impressions

Working in a company that is one of the 'global players' in semiconductor industries and has more than 30.000 employees world wide is indeed a very interesting experience. It gave me a number of great new impressions and opened my view on some sort of problems that will probably never be encountered in a smaller company.

The people in the department I was working at were really very supportive and they always encouraged me in my work. Additionally they gave me good and honest feedback for a number of smaller presentations I made throughout the whole time at Infineon Technologies. Concluding I can say that I have learned much more than just plain technical knowledge during the writing of this thesis.

Appendix A

XML Documents

The XML source files of the examples used in chapter 5 are listed here:

book.xml - is an example of an assembly file that combines two chapters to a complete book

```
 <? xml version = "1.0" encoding = "ISO-8859-15" ?>

<book>
       <part link = "chapter/ch1.xml" />
       <part link = "chapter/ch2.xml" />
</book>
```

ch1.xml - an example file for XML structured content, including chapter, section, list and cross reference elements

```
<? xml version = "1.0" encoding = "ISO-8859-15" ?>
<chapter id = "ch1" >
<title> Development of the Transistor </title>
<section id = "sect0" >
<title> Transistor </title>
<paragraph> The transistor is a solid state semiconductor device
which can be used for amplification, switching, voltage stabilisation,
signal modulation and many other functions.  It acts as a variable
valve which, based on its input current (BJT) or input voltage
```

```
(FET), allows a precise amount of current to flow through it
from the circuit's voltage supply.
</paragraph>
<paragraph>
In essence, it has three terminals (<crossRef Idref="CEGHDIBB"
 type="FigureTitle" />).  A current or voltage applied through/across
two terminals controls a larger current through the other terminal
and the common terminal.  In analog circuits, transistors are
used in amplifiers.  Analog circuits include audio amplifiers,
stabilised power supplies and radio frequency amplifiers.  In
digital circuits, transistors function essentially as electrical
switches.  Digital circuits include logic gates, RAM (random
access memory) and microprocessors.
</paragraph>
<paragraph>
Transistor was also the common name in the sixties for a transistor
radio, a portable radio that used transistors (rather than vacuum
tubes) as its active electronic components.

<figure><image file="http://localhost/showcase/db/IT
%20BT/content/transistor/images/trans.jpg"  height = "100pt"
width="216pt" position="below" sideways="0" impang="0.000"
 nsoffset="0.000in" /><figtitle id="CEGHDIBB">Transistors
(tape measure marked in centimeters)</figtitle></figure>
</paragraph>
</section>

<section id="sect1">
<title>Invention</title>
<paragraph>
The transistor was invented at Bell Telephone Laboratories in
December 1947 (first demonstrated on December 23) by John Bardeen,
Walter Houser Brattain, and William Bradford Shockley, who were
awarded the Nobel Prize in physics in 1956.  Ironically, they
had set out to manufacture a field-effect transistor (FET) predicted
by Julius Edgar Lilienfeld as early as 1925 but eventually discovered
current amplification in the point-contact transistor that subsequently
evolved to become the bipolar junction transistor (BJT).
</paragraph>
```

```
</section>


<section id="sect2">
<title>Types</title>
<paragraph>
Broadly speaking, transistors are categorised as follows:
<list>
<item>Semiconductor material: germanium, silicon, gallium arsenide
</item>
<item>Physical packaging: through hole metal, through hole
plastic, surface mount, ball grid array</item>
<item>Type: BJT, JFET, IGFET (MOSFET), "other types"</item>

<item>Polarity: NPN/ N channel, PNP/ P channel</item>
<item>Maximum power rating: low, medium, high</item>
<item>Maximum frequency capability: low, medium, high, radio
frequency (RF), microwave</item>
<item>Application: amplifier, switch, general purpose, audio,
high voltage</item>
</list>

</paragraph>
</section>
</chapter>
```

ch2.xml - second chapter

```
<?xml version="1.0" encoding="ISO-8859-15"?>
<chapter id="ch2">
<title>FET Families</title>

<paragraph>
FETs are divided into two families: junction FET (JFET) and
insulated gate FET (IGFET) also known as metal oxide silicon
(or semiconductor) FET (MOSFET). Unlike IGFETs, the JFET gate
terminal forms a diode with the channel (semiconductor material
```

between source and drain). Functionally, this makes the N channel JFET the solid state equivalent of the vacuum tube triode which, similarly, forms a diode between its grid and cathode. Also, both devices operate in the ’depletion mode’, they both have a high input impedance, and they both conduct current under the control of an input voltage.
</paragraph>

<paragraph>
FETs are further divided into enhancement mode and depletion mode types. Mode refers to the polarity of the gate voltage with respect to the source when the device is used for linear amplification. Taking N channel FETs: in depletion mode the gate is negative with respect to the source while in enhancement mode the gate is positive. For both modes, if the gate voltage is made more positive the source/drain current will increase. For P channel devices the polarities are reversed. Most IGFETs are enhancement mode types and nearly all JFETS are depletion mode types.
</paragraph>

</chapter>

Appendix B

XSL Transfomation

With this stylesheet transformation it is possible to combine XML modules to a single file, when they are referenced as it is stated above in the 'book.xml'. Furthermore this stylesheet generates a table of contents and dissolves cross references of the input XML files. The result is a nicely formatted XSL-FO based book.

book2fo.xsl

```
<? xml version = "1.0" encoding = "ISO-8859-15" ?>
<xsl:stylesheet version = "1.0"
xmlns:xsl = "http://www.w3.org/1999/XSL/Transform" xmlns:fo =
"http://www.w3.org/1999/XSL/Format" >
<xsl:output omit-xml-declaration = "no" method = "xml" encoding
= "UTF-8" indent = "yes" />
```

Here the stylesheet parameter that is used for the 'editable PDF' functionality is initialised. The testing wether it should be enabled or not is done below.

```
<!- ###### Stylesheet Parameters ####################### ->
<xsl:param name = "editsource" />
<!- ######################################## ->

<!- ###### Includes ######################## ->
<xsl:include href = "wg/page-header-footer.xsl" />
<!- ######################################## ->
```

The ’serverPath’ variable is used to find the source document on the server if the ’editable PDF’ functionality is enabled. It would also be possible to pass this information in as a stylesheet parameter for more flexibility.

```
<!- ###### Global Variables ##################### ->
<xsl:variable name="serverPath"
select = "'http://localhost/showcase/editxml.aspx?file=db/it
bt/content/transistor/'" />
<!- ################################# ->

<xsl:template match="/" >

<fo:root>
<!- The <fo:layout-master-set> element contains one or more
page templates ->
<fo:layout-master-set>
<!- Each <fo:simple-page-master> element contains a single page
template.
Each template must have a unique name (master-name):  ->
<fo:simple-page-master master-name="A4" page-width="210mm"
 page-height="297mm" margin-top="1cm" margin-bottom="1cm"
margin-left="2cm" margin-right="2cm" >
<fo:region-body margin-top="2.4cm" margin-bottom="2cm" />
<fo:region-before region-name = "xsl-region-before" extent =
"2.4cm" display-align="before" />
<fo:region-after region-name="xsl-region-after"    extent =
"2cm" />
<fo:region-start    extent="1.5cm" />
<fo:region-end extent="1.5cm" />
</fo:simple-page-master>
</fo:layout-master-set>
<!- One or more <fo:page-sequence> elements describe page contents.

The master-reference attribute refers to the simple-page-master
template
with the same name:  ->
```

The code below generates the table of contents. Chapter and section numbering is done by counting the specific elements. Additionally internal links are included to allow the user to jump to each part of the book directly from the table of contents. The destination of each link is specified by the unique ID each chapter or section carries.

```
<!- ###### TOC ################################## ->
<fo:page-sequence master-reference = "A4" >
<fo:flow flow-name = "xsl-region-body" >
<xsl:for-each select = "book/part" >
<!- ######### Do the Chapter numbering ######## ->
<xsl:variable name = "chapternumber" >
<xsl:number count = "part" format = "1" />
</xsl:variable>
<xsl:for-each select = "document(@link)/chapter" >

<fo:block text-align = "right" >
<fo:inline keep-with-next.within-line = "always" >
<xsl:value-of select = "$chapternumber" />
&#x00a0;&#x00a0;&#x00a0;&#x00a0; &#x00a0;&#x00a0;&#x00a0;&#x00a0;

<fo:basic-link internal-destination = "{@id}" color = "blue" >

<xsl:value-of select = "title" />
</fo:basic-link>
</fo:inline>
<fo:inline keep-together.within-line = "always" >
<fo:leader leader-pattern = "dots" leader-pattern-width = "4pt"
 leader-alignment = "reference-area" keep-with-next.within-line
= "always" />

<fo:page-number-citation ref-id = "{@id}" />

</fo:inline>
</fo:block>
```

```
<!- ######### Do the Section numbering ######## ->

<xsl:for-each select = "section" >
<xsl:variable name = "sectionnumber" >
<xsl:number count = "section" format = "1" />
</xsl:variable>

<fo:block text-align = "right" >
<fo:inline keep-with-next.within-line = "always" >
<xsl:value-of  select = "$chapternumber" /> .  <xsl:value-of
select = "$sectionnumber" /> &#x00a0;&#x00a0;&#x00a0;&#x00a0;
&#x00a0;&#x00a0;&#x00a0;&#x00a0;
<fo:basic-link internal-destination = "{@id}" color = "blue" >

<xsl:value-of select = "title" />
</fo:basic-link>
</fo:inline>
<fo:inline keep-together.within-line = "always" >
<fo:leader leader-pattern = "dots" leader-pattern-width = "4pt"
 leader-alignment = "reference-area" keep-with-next.within-line
= "always" />

<fo:page-number-citation ref-id = "{@id}" />

</fo:inline>
</fo:block>
</xsl:for-each>

</xsl:for-each>
</xsl:for-each>
</fo:flow>
</fo:page-sequence>
```

The code below copes with the content of each module specified inside the assembly file.

```
<!- ############################################ ->
<!- ########## Content ########################## ->
<!- ############################################ ->
<xsl:for-each select = "book/part" >
<xsl:variable name = "chapternumber" >
<xsl:number count = "part" format = "1" />
</xsl:variable>
<xsl:variable name = "filePath" select = "@link" />
```

This ’xsl:for-each’ dissolves every chapter that is included inside the assembly file and opens the required document to process the content stored inside.

```
<xsl:for-each select = "document(@link)/chapter" >

<fo:page-sequence master-reference = "A4" >

<xsl:call-template name = "draw-page-header" />
<xsl:call-template name = "draw-page-footer" />
<fo:flow flow-name = "xsl-region-body" >


<!- ############### chapter header ############### ->
<fo:block id = "{@id}" >
<fo:block font-family = "Helvetica" >
<fo:block font-size = "14pt" font-weight = "bold" >
<fo:table space-before = "5pt" >
<fo:table-column column-width = "16cm" />
<fo:table-column column-width = "2cm" />
<fo:table-body>
<fo:table-row>
<fo:table-cell>
<fo:block>
<xsl:value-of select = "$chapternumber" /> &#x00a0;&#x00a0;&#x00a0;&#x00a0;
<xsl:value-of select = "title" />
```

```
</fo:block>
</fo:table-cell>
<fo:table-cell>
```

Here the actual test for an 'editable PDF' is done. If the 'editsource' parameter is set to 'true' then an external link is included that points to the XML source document that lies behind this part of the PDF.

```
<!- Test if edtiable is enabled or not ->
<xsl:if test = "$editsource='true'" >
<fo:block font-size = "8pt" >
<fo:basic-link   text-align = "right" color = "blue" external-destination
= "{$serverPath}{$filePath}" > edit source </fo:basic-link>
</fo:block>
</xsl:if>
</fo:table-cell>
</fo:table-row>
</fo:table-body>
</fo:table>
</fo:block>
</fo:block>
</fo:block>

<xsl:apply-templates>
<xsl:with-param name = "chapNo" select = "$chapternumber" />
</xsl:apply-templates>

</fo:flow>
</fo:page-sequence>
</xsl:for-each>
</xsl:for-each>
</fo:root>
</xsl:template>



<!- #### Section ##################################   ->
```

```
<xsl:template match = "section" >
<xsl:param name = "chapNo" />

<!- ############### Section header ############### ->
<fo:block id = "{@id}" >
<fo:block font-family = "Helvetica" >
<fo:block font-size = "12pt" font-weight = "bold" >
<fo:block keep-with-next.within-column = "always" space-before.optimum
= "12pt" space-before.minimum = "12pt * 0.8" space-before.maximum
= "12pt * 1.2" >
<xsl:value-of select = "$chapNo" /> . <xsl:number count = "*//section"
 format = "1" /> &#x00a0;&#x00a0; <xsl:value-of select = "title"
/>
</fo:block>
</fo:block>
</fo:block>
</fo:block>

<xsl:apply-templates />
</xsl:template>

<!- #### Section Label   #################################   ->

<xsl:template match = "section" mode = "label" >
<xsl:value-of select = "title" />
</xsl:template>


<!- #### paragraph #################################   ->
<xsl:template match = "paragraph" >

<fo:block font-family = "Helvetica" font-size = "10pt" text-align
= "justify" space-before.optimum = "1em" space-before.minimum
= "0.8em" space-before.maximum = "1.2em" >
<xsl:apply-templates />
</fo:block>
```

```
</xsl:template>
```

In this part of the code cross references are dissolved and a internal link pointing to the target of the reference is inserted.

```
<!- ####   CrossRef ##################################   ->
<xsl:template match = "crossRef" >
<xsl:variable name = "thisref" >
<xsl:value-of select = "@Idref" />
</xsl:variable>

<fo:basic-link internal-destination = "{@Idref}" color = "blue"
>
<xsl:apply-templates select = "//*[@id=$thisref]" mode = "label"
/>
</fo:basic-link>

<xsl:apply-templates />
</xsl:template>


<!- #### Lists   #################### ->
<xsl:template match = "list" >

<fo:list-block  padding-after = "12pt"  space-after.minimum =
"12pt" space-after = "12pt" >

<xsl:for-each select = "item" >
<fo:list-item>
<fo:list-item-label end-indent = "label-end()" >
<fo:block font-family = "Helvetica" font-size = "12pt" > &#x2022;
</fo:block>
</fo:list-item-label>
<fo:list-item-body start-indent = "body-start()" >
<fo:block font-family = "Helvetica" font-size = "10pt" >
<xsl:apply-templates />
```

```
</fo:block>
</fo:list-item-body>
</fo:list-item>
</xsl:for-each>

</fo:list-block>

</xsl:template>


<!- #### Table printing   #################### ->
<xsl:template match = "table" >

<fo:table border = "yes" padding = "2mm" border-width = "0.3mm"
 border-style = "none" table-layout = "auto" width = "100%" >

<!- ## No.  of Columns   ### ->
<xsl:for-each select = "thead/row/cell" >
<xsl:if test = "position()!=last()" >
<fo:table-column column-width = "{@width}" border-right-width
= "0.3mm" border-right-style = "solid" />
</xsl:if>
<xsl:if test = "position()=last()" >
<fo:table-column column-width = "{@width}" />
</xsl:if>
</xsl:for-each>

<xsl:apply-templates />
</fo:table>

</xsl:template>

<xsl:template match = "thead" >
<fo:table-header font-weight = "900" >
<xsl:apply-templates />
</fo:table-header>
</xsl:template>
```

```
<xsl:template match = "tbody" >
<fo:table-body wrap-option = "no-wrap" >
<xsl:apply-templates />
</fo:table-body>
</xsl:template>

<xsl:template match = "row" >
<fo:table-row>
<xsl:apply-templates />
</fo:table-row>
</xsl:template>

<xsl:template match = "cell" >
<fo:table-cell padding-left = "3pt" padding-right = "3pt" padding-bottom
= "3pt" padding-top = "3pt" border-top-width = "0.3mm" border-top-style
= "solid" border-bottom-width = "0.3mm" border-bottom-style =
"solid" >
<fo:block padding-left = "3mm" ><xsl:apply-templates />
</fo:block>
</fo:table-cell>
</xsl:template>
<!- ##################################### ->
```

Here the reference to external figures is dissolved and they are included into the PDF document with the 'fo:external-graphic' command. The table around the figure is only used to get a nice looking border.

```
<!- #### figure include #################### ->
<xsl:template match = "figure" >

<fo:table   space-before = "10pt" table-layout = "fixed"
width = "{./image/@width}" height = "{./image/@height}" border-width
= "1.3pt" border-color = "black" border-style = "solid" >
<fo:table-column column-number = "1"
column-width = "proportional-column-width(1)" />
```

```
<fo:table-body>
<fo:table-row>
<fo:table-cell>
<fo:block>
<fo:external-graphic src="{./image/@file}" content-width =
"{./image/@width}" content-height="{./image/@height}">
</fo:external-graphic></fo:block>
</fo:table-cell>
</fo:table-row>
</fo:table-body>
</fo:table>


<xsl:apply-templates/>
</xsl:template>
```

Here the consecutively figure numbering is done and inserted below the actual figure together with the figure title.

```
<!- ##### figure title near figure ####### ->
<xsl:template match="figtitle">

<fo:block id="{@id}">
<fo:inline>
Figure <xsl:number count="*//figtitle" format="1" />:
</fo:inline>
<fo:inline>
<xsl:apply-templates/>
</fo:inline>
</fo:block>

</xsl:template>
```

Cross references to figures are dissolved in this part of the code. Therefore the appropriate figure number has to be found and afterwards inserted into the PDF.

```
<!- ##### figure title as CrossRef Label ####### ->
<xsl:template match="figtitle" mode="label">Figure  <xsl:number
 count="*//figtitle" format="1" />
</xsl:template>



<xsl:template match="externalref">

<fo:basic-link external-destination="{@link}" color="blue"
>
<xsl:apply-templates/>
</fo:basic-link>
</xsl:template>

</xsl:stylesheet>
```

page-header-footer.xsl - contains the header and footer information used for XSL-FO files

```
<?xml version="1.0" encoding="ISO-8859-15"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform" xmlns:fo=
"http://www.w3.org/1999/XSL/Format">
<xsl:output omit-xml-declaration="no" method="xml" encoding
="UTF-8" indent="yes"/>


<!- #### Draw the Page header ##################################   ->

<xsl:template name="draw-page-header">

<fo:static-content flow-name="xsl-region-before">
<fo:block text-align="start">
<fo:external-graphic src="http://localhost/showcase/db/IT
%20BT/content/wg/images/logo_infineon.bmp" width="122px" height
="57px"/>
```

```
</fo:block>
<fo:block font-family = "Helvetica" margin-left = "123px" >
<fo:table table-layout = "fixed" width = "80%" >
<fo:table-column column-number = "1" column-width = "80pt" />
<fo:table-column column-number = "2" column-width = "400pt" />

<fo:table-body>
<fo:table-row height = "4pt" >
<fo:table-cell text-align = "left" display-align = "before" >
<fo:block>
<fo:block />
</fo:block>
</fo:table-cell>
<fo:table-cell text-align = "center" display-align = "before"
 border-bottom-width = "0.5pt" border-bottom-style = "solid"
border-bottom-color = "black" >
<fo:block>
<fo:block />
</fo:block>
</fo:table-cell>
</fo:table-row>
</fo:table-body>
</fo:table>
</fo:block>
</fo:static-content>

</xsl:template>

<!- #### Draw the Page Footer #################################   ->

<xsl:template name = "draw-page-footer" >

<fo:static-content flow-name = "xsl-region-after" >
<fo:block font-family = "Helvetica" font-size = "10pt" >
<fo:table table-layout = "fixed" width = "100%" border-top-width
= "0.5pt" border-top-style = "solid" border-top-color = "black"
>
<fo:table-column column-number = "1"
column-width = "proportional-column-width(1)" />
```

```
<fo:table-column column-number="2"
column-width="proportional-column-width(1)"/>
<fo:table-column column-number="3"
column-width="proportional-column-width(1)"/>
<fo:table-body>
<fo:table-row height="12pt">
<fo:table-cell text-align="left" display-align="after">
<fo:block>
<fo:block/>
</fo:block>
</fo:table-cell>
<fo:table-cell text-align="center" display-align="after">

<fo:block>
<fo:block><fo:page-number/></fo:block>
</fo:block>
</fo:table-cell>
<fo:table-cell text-align="right" display-align="after">
<fo:block>
<fo:block/>
</fo:block>
</fo:table-cell>
</fo:table-row>
</fo:table-body>
</fo:table>
</fo:block>
</fo:static-content>

</xsl:template>

</xsl:stylesheet>
```

List of Figures

Bibliography

[1] T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. M.
eXtensible Markup Language (XML) 1.0. Technical report, World Wide Web Consortium (W3C), October 2000.
www.w3.org/TR/REC-xml (2005/05/28)

[2] SGML international standard - iso8879, 1986.

[3] R Knox, C Abrams, W Andrews, T Friedman, K Harris, A Linden, D Logan, M Knox and R Wagner
Hype Cycle for XML Technologies, May 2003

[4] E. Rusty Harold, W. Scott Means
XML in a Nutshell 3rd Edition, September 2004.

[5] Dick Kevin
XML A Managers Guide
Addison-Wesley, USA 2000, p. 13ff

[6] W3C XML Schema
www.w3.org/XML/Schema (2005/05/28)

[7] Priscilla Walmsley
XML Schema: An Overview, Januar 2002
www.informit.com/articles/article.asp?p=25002 (2005/05/28)

[8] RELAX NG
www.relaxng.org/ (2005/05/28)

[9] RELAX Core
www.xml.gr.jp/relax/ (2005/07/26)

[10] TREX
www.thaiopensource.com/trex/ (2005/07/26)

[11] Christoph Lange
XML-Dokumenttypen mit Schema-Sprachen beschreiben, May 2002
cip.uni-trier.de/lange/semxml/semarb.html (2005/07/26)

[12] David Mertz
XML Matters: Kicking back with RELAX NG, February 2003
www-128.ibm.com/developerworks/xml/library/x-matters25.html
(2005/07/26)

[13] Eric van der Vlist
RELAX NG, December 2003
books.xmlschemata.org/relaxng/page2.html (2005/08/05)

[14] W3C Recommendation
XML Path Language (XPath), November 1999
www.w3.org/TR/xpath (2005/05/28)

[15] Evan Lenz
What's New in XSLT 2.0, April 2002
www.xml.com/pub/a/2002/04/10/xslt2.html (2005/07/26)

[16] Jinyu Wang
Five XSLT 2.0 Features that Simplify XML Document Transformations, 2005
www.oracle.com/technology/pub/articles/wang_xslt.html (2005/07/26)

[17] Sudhir Mangla
Beginners Introduction to ASP.NET, August 2003.
www.devhood.com/tutorials/tutorial_details.aspx?tutorial_id=768
(2005/05/28)

[18] MSDN Library
Introduction to ASP.NET, ©2005 Microsoft Corporation.
msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconIntroductionToASP.asp (2005/05/28)

[19] John Kopp
C# Tutorial - Introduction
cplus.about.com/od/learnin2/l/aa020804a.htm (2005/05/28)

[20] Joey Mingrone
An Introduction to C#
devcentral.iticentral.com/articles/CSharp/intro_Csharp/default.php
(2005/05/28)

[21] Pankaj Kamthan and Hsueh-Ieng Pai
Perspectives of XML in E-Commerce, December 2000
parallel.ru/docs/Internet/IRT/articles/js215/index.htm (2005/05/28)

[22] Benoît Marchal
ebXML: An Electronic Business Scenario, July 2003
www.developer.com/xml/print.php/2234201 (2005/05/28)

[23] Kurt Ament
Single Sourcing: Building Modular Documentation
Noyes Publications, September 2002

[24] Rebecca Sukach, Robert Kennedy, and Marie Devine
Implementing Single Sourcing in Your Organization
www.stc.org/confproceed/2002/PDFs/STC49-00032.pdf (2005/05/28)

[25] Cherryleaf Ltd.
Cherryleaf survey results - Use of single sourcing solutions, April 2003
www.cherryleaf.com/singlesourcessurvey.htm (2005/05/28)

[26] James Clark
Document Style Semantics and Specification Language
www.jclark.com/dsssl/ (2005/05/28)

[27] Jan Goyvaerts
How to Create HTML Help Files with HelpScribble, April 2004
www.helpscribble.com/htmlhelp.html (2005/05/28)

[28] Kimber W. Eliot
Using XSL Formatting Objects for Production-Quality Document Printing, December 2002

www.idealliance.org/papers/xml02/dx_xml02/papers/06-05-01/06-05-01.pdf (2005/05/28)

[29] Ben Shneiderman
Treemaps for space-constrained visualization of hierarchies, December 1998
www.cs.umd.edu/hcil/treemap-history/index.shtml (2005/05/28)

[30] C.Y. Hsieh
File Design for Information Systems
pluto.ksi.edu/ cyh/cis501/ch8.html (2005/05/28)