



Micro XML SAX Parser User's Guide

This guide provides the information you need to use Fusion software after it has been ported to your hardware/software platform.

Software Version:	1.0
Part Number/Version:	???
Release Date:	???

NetSilicon Softworks Group

810 Lawrence Dr., Suite #200

Newbury Park, Ca. 91320-6616

USA: (800) 541-9508

Outside the USA: 1-805-499-7722

IMPORTANT

No title to or ownership of the software described in this document or any of its parts, including patents, copyrights and trade secrets, is transferred to customers. NetSilicon makes no representations or warranties regarding the contents of this document. Information in this document is subject to change without notice and does not represent a commitment on the part of NetSilicon. This manual is protected by United States Copyright Law, and may not be copied, reproduced, transmitted or distributed, in whole or part, without the express prior written permission of NetSilicon.

COPYRIGHT NOTICE

© 2000 NetSilicon, Inc.

Printed in the United States of America. All rights reserved.

TRADEMARKS

NetSilicon and the NetSilicon logo are trademarks of NetSilicon, Inc.

All other brand and product names are trademarks, service marks, registered trademarks, or registered service marks of their respective companies.

IMPORTANT NOTICE

NetSilicon reserves the right to make changes to its products without notice, and advises its customers to obtain the latest version of relevant information to verify, before placing orders, that the information being relied on is current.

CHAPTER 1: OVERVIEW

The Micro XML SAX Parser, is a small foot print SAX (Simple API for XML) parser (as defined at <http://www.megginson.com/SAX/index.html>) written in the C language. It will parse XML 1.0 documents (see <http://www.w3.org/TR/2000/REC-xml-20001006>). It is not a validating parser and does not read external DTD. The parser does check for well formed ness and returns a fatal error if the document is not well formed. It conforms to the SAX2 standard, as much as is possible given that SAX is defined for Java and the parser is written in C. Where possible the parser makes tradeoffs to minimize memory usage. The parser uses the SAX paradigm of callbacks for XML elements and other constructs.

CHAPTER 2: INSTALLATION

CONFIGURATION

XML is defined to use the Unicode standard (see <http://www.unicode.org/>) for characters. However the most natural Unicode encoding UTF-16, uses sixteen bits per character. This makes each character take twice as much memory as ASCII. Therefore the stream is twice as large if it is represented as UTF-16. Because memory foot print is an issue for embedded devices the parser can be configured to either process the input stream as ASCII or UTF-16 characters. The advantage of ASCII is that the XML document will be half the size, the downside is that you can only have ASCII (read English) characters. By default the parser expects an ASCII XML document. If you compile the source files with the preprocessor define **NS_USE_UNICODE** then the parser will process a Unicode XML document.

Many documents are defined as a charset encoding of UTF-8. UTF-8 is a variable length encoding of Unicode where ASCII characters are left as ASCII characters but non ASCII characters are stored in two, through six bytes. UTF-8 has the advantage that if the document has only ASCII characters then the size of the document will be the same as the ASCII file, but it can also contain other language characters although those may be encoded in more than two bytes. The effect is that if the document has a lot of non-ASCII characters then the document will be larger than the equivalent UTF-16. The parser with **NS_USE_UNICODE** UTF-8 or UTF-16 documents. However if the document is in UTF-8 format then the document will be converted to UTF-16 and it will have to dynamically allocate memory for the UTF-16 buffer. The parser also includes a convenience function to convert a UTF-8 char to a Unicode character (`utf8CharToUnicode`). See the Miscellaneous API section for more details.

PORTING

The parser is designed to be portable and to that end it does not depend upon any operating system, C runtime or compiler includes. However there are some very rare cases where the parser uses dynamic memory. For dynamic memory allocation by default the parser uses `malloc`.

However any dynamic memory allocation system can be used. The file `x_malloc.c` is a wrapper for `malloc` and can be replaced by any alternative.

DEPLOYING THE PARSER

The code and data (text and data) space for the parser is about 13K. It does not require any block storage space (bss).

The parser requires dynamic memory allocation for the following cases:

1. Parameter Entities defined in a DTD
2. Entity references defined in a DTD
3. Attribute defaults defined in a DTD
4. Attribute values with entity or character references whether predefined or not.
5. Attribute values containing new lines that require character compression. That is where the end of line consists of the carriage return line feed (0x0d, 0x0a) , which needs to be converted to just a line feed (0x0a). See sections 3.3.3 and 2.11 of the XML 1.0 specification for more information.
6. Processing a UTF-8 document in `NS_USE_UNICODE` build a buffer is allocated for the UTF-16 buffer.

The instruction call stack requirements can vary if there is a DTD which has parameter or entity references since processing of these can require recursion. But approximately 1K bytes seems reasonable.

CHAPTER 3: APPLICATION PROGRAM INTERFACE

To use the SAX parser a file will need to include the file `xmlsaxapi.h`. This file contains all the function prototypes and data definitions necessary to use the SAX parser.

Basic Types

The parser uses some basic types for processing the data.

- `X_CHAR` Defined to be a `char` by default, and to an unsigned short (for 16 bit UTF-16 character) if `NS_USE_UNICODE` is selected.
- `String` Callbacks return character strings as a `String` type. `String` is a structure of a pointer to the first character (`X_CHAR *`) and a pointer to the last character in a string (`X_CHAR *`). It is not, therefore `NULL` terminated. `String` is defined this way so that the input XML document can be reused, and therefore save space. In practice not every `String` is from the input XML document and therefore it is not safe to assume that data will be valid outside the callback.

It is defined as an abstraction so that the implementation may be changed later on. Another alternative implementation would have added `NULL` characters to the input XML document (permanently destroying it) that would have allowed reuse and a simple `NULL` terminated (`X_CHAR *`) implementation. We're interested in feed back about the current implementation of `String`.

GENERAL NOTES

The first parameter to every callback routine is a `void *userData`. The parameter is the `userData` passed in from the application to the `SAXParser`, for the benefit of the user. It can be `NULL` if not needed. In some cases this will provide the benefit of a "this" pointer.

Each callback returns an `int`. The callback should return 0 for success. Returning anything other than zero will cause `SAXParser` to terminate with a fatal error (it will return `XML_ERROR`). The last parameter passed to `SAXParser` will contain a structure containing information about the error. The `error` field (of the error parameter) will be set to `PARSER_USER_INITIATED` and `userErrorCode` set to the value returned from the callback.

API FUNCTIONS

SAXParser is the main parsing function. It will call the specified callback functions when significant XML constructs are encountered. The SAX parser is invoked with the following function:

```
enum X_Status SAXParser(X_CHAR *stream,
                        void *userData,
                        ContentHandler *contentHandler,
                        LexicalHandler *lexicalHandler,
                        ErrorHandler *errorHandler,
                        struct ParserErrorEventType *error);
```

Parameters:

- **stream:** A NULL terminated string of bytes (not Unicode), or unsigned short (Unicode), the application should use X_CHAR which is defined appropriately for Unicode selection. Note the whole document must be passed
- **userData:** A pointer to data that is passed down through to the callbacks for the users purpose. It provides a “this pointer” capability.
- **contentHandler:** A structure containing pointer to functions that are callbacks for various events. It must not be NULL but any callback in it can be NULL if the user is not interested in notification of the event. Maps to a similar SAX object.
- **lexicalHandler:** Similar to contentHandler and having similar characteristics. Maps to a similar SAX object.
- **errorHandler:** Similar to contentHandler and having similar characteristics. Maps to a similar SAX object.

Return value:

- XML_OK
- XML_ERROR

PARAMETER TYPE DETAILS

ContentHandler – is a structure of pointers to functions as defined in SAX2. Each is discussed in detail below. The SAX2 specification, see resources above, may be helpful. ContentHandler is defined in *ContentHandler.h*.

- **int (*characters)(void *userData, String *ch);**

This function is called for characters inside an element (or CDATA). According to the SAX2 specification the characters in the element can be spread over more than one call to (*characters). In the parser the only time that characters will be spread over one call is if there is entity replacement or end of line compression.

-
- `int (*endDocument)(void *userData);`

This function is called when the end of the document is encountered.

- `int (*endElement)(void *userData, String *namespaceURI, String *localName, String *qName);`

This function is called when an end element is encountered. The namespaceURI is the namespace prefix converted to a URI. If the namespace isn't defined then the parameter will contain the prefix and the errorHandler function is called with a non fatal error. If the element doesn't have a name space then namespaceURI will point to a String where the first item points to X_NULL. The localName parameter contains the local part of the element name. The qName parameter contains the raw element name.

- `int (*endPrefixMapping)(void *userData, String *prefix);`

This function is called for an end element that terminates an element that defined a name space. The prefix being passed is the subject name space. It will be a pointer to nulls if it is the default name space.

- `int (*ignoreableWhitespace)(void *userData, String *ch);`

This function is not called. It is not clear from the SAX2 specification under what conditions this function should be called.

- `int (*processingInstruction)(void *userData, String *target, String *data);`

The function is called when a processing instruction is encountered. The parameter target is the processing instruction target and data is everything else.

- `int (*skippedEntity)(void *userData, String *name);`

This is not called. It may be implemented in a later version.

- `int (*startDocument)(void *userData);`

The function is called at the start of parsing an XML document.

- `int (*startElement)(void *userData, String *namespaceURI, String *localName, String *qName, Attributes *atts);`

The function is called when a start element is encountered. The namespaceURI is the namespace prefix converted to a URI. If the namespace isn't defined then the parameter will contain the prefix and the errorHandler function is called with a non fatal error. If the element doesn't have a name space then namespaceURI will point to a String where the first item points to X_NULL. The localName parameter contains the local part of the element name. The qName parameter contains the raw element name. The atts parameter contains the attributes defined in the element.

- `int (*startPrefixMapping)(void *userData, String *prefix, String *uri);`

The function is called each time a name space is defined in an attribute of a start element. The prefix is the namespace prefix. It is a pointer to nulls if it is the default name space. The uri parameter is the URI of the name space.

LexicalHandler – is a structure of pointers to functions as defined in SAX2. Each is discussed in detail below. The SAX2 specification, see resources above, may be helpful. *LexicalHandler* is defined in *LexicalHandler.h*.

- `int (*comment)(void *userData, String *characters);`
The function is called when a user encounters a comment. The characters parameter contains the characters of the comment.
- `int (*endCDATA)(void *userData);`
The function indicates the end of a CDATA section. The characters in the CDATA are reported in the characters callback of the ContentHandler structure.
- `int (*endDTD)(void *userData);`
The function indicates the end of the DTD.
- `int (*startCDATA)(void *userData);`
The function indicates the beginning of a CDATA section. The characters in the CDATA are reported in the characters callback of the ContentHandle structure.
- `int (*startDTD)(void *userData, String *name,
String *publicId,
String *systemId);`

The function is called when the start of the DTD is encountered. The name parameters is the first token after the DTD token and the publicID and systemID are the values of the PUBLIC and SYSTEM names respectively.

ErrorHandler – is a structure of pointers to function as defined in SAX2. Each is discussed in detail below. The SAX2 specification, see resources above, may be helpful. *ErrorHandler* is defined in *ErrorHandler.h*.

- `void (*errorHandler)(void *userData,
struct ParserErrorEventType
*error);`
The function is called when a name space not found error is encountered. The error parameter is defined below.
- `void (*fatalErrorHandler)(void *userData, struct
ParserErrorEventType *error);`
This function is called for every kind of parser error except name space not found errors. The error parameter is defined below.

-
- `void (*warningHandler)(void *userData, struct ParserErrorEventType *error);`

This function is not called in the current version of the parser.

Attributes – this structure contains the attributes defined in a start element. It is not very similar to the SAX2 standard in form, but provides similar functionality.

- `int numberOfAttributes;`
The number of attributes defined in the start element.
- `Attribute *attribute;`
An array of attributes and values.

Attribute – the structure contains the an attribute name value pair. Attribute is defined in *Attributes.h*.

- `String qualifiedName;`
The is the attribute name as it appears in the start tag.
- `String prefix;`
This is the attribute name space prefix URI. If the name space prefix could not be resolved then it would have the unresolved prefix and the errorHandler function will be called and prefix will contain the prefix short name.
- `String localPart;`
This is the local part of the attribute name.
- `String value;`
The is the attribute value after all entity replacements and defaults have been made.

ParserErrorEventType – the structure provides details of the error similar to those found in SAXParserException. ParserErrorEventType is defined in *ParserError.h*.

- `enum ParserError error;`
This is the type of error the parser encountered from an enumerated list of types. See blow for list.
- `int characterNumber;`
This is the character number in the line of the document where the error occurred.
- `int lineNumber;`
This is the line number within the document where the error occurred.
- `int userErrorCode;`
If the error field is set to `PARSER_USER_INITIATED` then this field will have the value returned from the user callback function.

Parser Error

- **PARSER_NO_ERROR** No error occurred, this value will never be passed to an error function.
- **PARSER_NO_ATTRIBUTE** Not an error, it is used for internal purposes. This value will never be passed to an error function.
- **PARSER_NO_HEAP** An attempt to allocation dynamic memory (malloc) failed.
- **PARSER_NAME_WITH_EXCESS_COLONS** A name that was expected to be a qualified name, that is, of the form “prefix:localpart” had extra colons.
- **PARSER_NO_LOCAL_PART** A name that was expected to a qualified name of the form “prefix:localpart” had the form “prefix:”.
- **PARSER_UNEXPECTED_END_OF_STREAM** The string terminator was encountered in the middle of an element.
- **PARSER_INVALID_NAME** Various errors where a name did not have the correct form, for example the first name of character was not valid.
- **PARSER_ATTRIBUTE_GRAMMER_ERROR** The attribute for is not correct, for example the equals sign is missing or the quotes are not symmetric.
- **PARSER_ATTRIBUTE_OVERFLOW** The parser is “hard coded” to allow a maximum of MAX_ATTRIBUTES (attributes.h). An element with more than MAX_ATTRIBUTES will generate this error.
- **PARSER_NAMESPACE_OVERFLOW** The parser is hard coded to allow a maximum of MAX_NAMESPACE_LEVELS (namespacetypes.h) of active name spaces. More than MAX_NAMESPACE_LEVELS will generate this error.
- **PARSER_INVALID_ENTITY** The character or parameter reference or entity reference is not valid. For example a character reference may contain a character that is not a digit or hex digit or the reference might not have a valid name.
- **PARSER_CHARACTER_ENTITY_OVERFLOW** The character reference defines a value that won’t fit in the character. If for example the parser was configured for ASCII and it encountered a reference of “&256;” it would generate this error.
- **PARSER_UNKNOWN_ENTITY** An entity reference or parameter entity is referenced but not defined.
- **PARSER_INVALID_CHARACTER** A general error usually indicating markup in the wrong place.
- **PARSER_UNDECLARED_NAMESPACE** A name space prefix is referenced that hasn’t been defined.

-
- **PARSER_INTERNAL_ERROR** An internal error, this really shouldn't ever happen.
 - **PARSER_ILLEGAL_MARKUP** A '<' was encountered in the wrong place.
 - **PARSER_USER_INITIATED** The user returned a non zero value from a callback function.

CHAPTER 4: MISCELLANEOUS FUNCTIONS

The String type is not the native way to express character data in C, therefore it can make programming in the callbacks a little more difficult. To address this problem two helper functions are provided.

```
int StringPtrCompareCString(String *s1, X_CHAR *s2);
```

The function will compare an XML Parser String with a C string in the same way strcmp compares two C strings.

For example to see if the element just encountered was a “pet” element the following code could be used.

```
int HandleStartElement(void *userData, String
*namespaceURI,
                        String *localName,
                        String *qName, Attributes *atts) {
    if (StringPtrCompareCString(localName, "pet") == 0)
        printf("Got a pet\n");
    return 0;
}
```

```
X_CHAR *StringPtrTo_X_CHAR_Ptr(String *s);
```

This functions provide the equivalent of a strdup function. Memory is allocated with malloc and therefore must be free-ed.

```
int utf8CharToUnicode(char **inPtrPtr);
```

This function converts a UTF-8 char to a Unicode character. It will return 0xffff upon error. It also advances the input UTF-8 pointer. Because of the way Unicode is defined a UTF-8 encoded character can have a value beyond 65535 (16 bits worth). The parser assumes a UTF-16 character, therefore the programmer needs to verify that all characters will fit into a 16 bit character type. The application programmer will need to perform any surrogate processing for characters beyond 65535.

When converting to Unicode 16 bit characters it is difficult to know in advance how large a buffer will be required. The bounds are that it will be no larger than twice the size of the UTF-8 stream and it may be smaller than the UTF-8 stream if the UTF-8 stream contains mostly greater than two byte sequences. The following fragment demonstrates use of the function.

```
unsigned char *utf8Buffer = ... /* A NULL terminated
character
                                string, assumed as input. */
```

```
unsigned char *tempUTF8;
X_CHAR *unicodeBuffer;
X_CHAR *tempUnicode;
/* Allocate space for the destination Unicode string with
   string termination.
   Note: This size is the upper bound potentially wasting
   memory.
*/
unicodeBuffer = x_malloc((strlen(utf8Buffer) + 1) * 2);
if (unicodeBuffer == X_NULL) {
    /* Do some error processing. */
    return error;
}
tempUTF8 = utf8Buffer;
tempUnicode = unicodeBuffer;
while (*tempUTF8 != '\0') {
    unsigned int value = utfCharToUnicode(&tempUTF8);
    if (value > 0xffff) {
        /* Overflow, the character won't fit in
           16 bits, do some special processing. */
        return error;
    }
    else if (value == 0xffff) {
        /* An illegal encoding, do some error
           processing. */
        return error;
    }
    else {
        /* A good character. */
        *tempUnicode = value;
        tempUnicode++;
    }
}
*tempUnicode = X_STRING_TERMINATOR;
```

CHAPTER 5:

EXAMPLE

For this example consider the following simple document.

```
<?xml version='1.0'?>
<pets>
  <cat color='brown'/>
  <rabbit color='black'>fluffy
</rabbit>
</pets>
```

The following is an example program that prints out the start and end elements of the preceding XML document.

```
#include <stdio.h>
#include "xmlsaxapi.h"

/* Convenience print function. */
void PrintString(String *s) {
    X_CHAR *ptr;
    for(ptr = s->first; ptr <= s->last; ptr++)
        putchar(*ptr);
}

/* Start element callback */
int HandleStartElement(void *userData, String
*namespaceURI, String *localName,
                        String *qName, Attributes *atts) {
    int i;
    printf("Start element: ");
    PrintString(qName);
    putchar('\n');
    for(i=0; i<atts->numberOfAttributes; i++) {
        printf(" Attribute: Name = ");
        PrintString(&atts->attribute[i].qualifiedName);
        printf(" Value = ");
        PrintString(&atts->attribute[i].value);
        putchar('\n');
    }
    return 0;
}

/* End element callback */
int HandleEndElement(void *userData, String *namespaceURI,
String *localName,
                        String *qName) {
    printf("Start element: ");
    PrintString(qName);
    putchar('\n');
    return 0;
}
```

```

/* Error callbacks */
void HandleError(void *userData, struct
ParserErrorEventType *error) {
    printf("Error: %d line %d char %d\r\n",
        error->error, error->lineNumber, error-
>characterNumber);
}

void HandleFatalError(void *userData, struct
ParserErrorEventType *error) {
    printf("Fatal Error: %d line %d char %d\r\n",
        error->error, error->lineNumber, error-
>characterNumber);
}

void HandleWarning(void *userData, struct
ParserErrorEventType *error) {
    printf("Warning: %d line %d char %d\r\n",
        error->error, error->lineNumber, error-
>characterNumber);
}

/* For details see ContentHandler.h */
ContentHandler contentHandler = {
    X_NULL,
    X_NULL,
    HandleEndElement,
    X_NULL,
    X_NULL,
    X_NULL,
    X_NULL,
    X_NULL,
    X_NULL,
    HandleStartElement,
    X_NULL
};

/* For details see LexicalHandler.h */
LexicalHandler lexicalHandler = {
    X_NULL,
    X_NULL,
    X_NULL,
    X_NULL,
    X_NULL
};

/* For details see ErrorHandler.h */
ErrorHandler errorHandler = {
    HandleError,
    HandleFatalError,
    HandleWarning
};

int main(int argc, char *argv[]) {
    X_CHAR *xmlDocument = "<?xml version='1.0'?>"
        "<pets>"
        "  <cat color='brown'/>"
        "  <rabbit color='black'>fluffy"

```

```
        " </rabbit>"
        "</pets>";
    struct ParserErrorEventType error;
    if (SAXParser(xmlDocument, NULL, &contentHandler,
&lexicalHandler,
        &errorHandler, &error) == XML_ERROR)
        fprintf(stdout, "Parser returned with an error\n");
    return 0;
}
```

The program produces the following output.

```
Start element: pets
Start element: cat
  Attribute: Name = color Value = brown
Start element: cat
Start element: rabbit
  Attribute: Name = color Value = black
Start element: rabbit
Start element: pets
```

It is left as an exercise to add printing of the parsed character data.