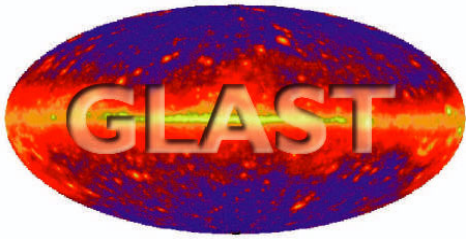


# *LAT Flight Software*

# XLX Programmers' Guide

Type: Programmers' Guide
Version: V2-0-1
Author: J. Swain
Created: 26 April 2004
Updated: 27 April 2004
Printed: 22 September 2004

This document provides an introduction to XLX, a package the wraps the EXPAT XML parsing library.

# Contents

# 0 Introduction

## 0.0 Intended Audience

This document is intended to provide a brief introduction to the XLX package for programmers who wish to build their own XML parser on top of XLX and those who wish to develop the mechanics of the underlying parser. Any poor soul tasked with modifying the auto-generated code produced by the *reg_parser* application should read the *reg_parser manual.*

## 0.1 References

*XLX Automatically Generated Documentation*,
http://www.slac.stanford.edu/exp/glast/flight/doxygen/Doxyidx.htm.

*GLAST Acronym List.*

## 0.2 Request for Comments

Please post corrections or questions to the SNITZ release announcement for the relevant XLX version. Failing this, email jswain@slac.stanford.edu

## 0.3 Overview

XLX provides a wrapper around the EXPAT XML parsing library. In particular XLX provides a *state* structure that keeps track of the parser's progress through the XML hierarchy, allowing the behavior of the parser to be modified depending on the previous tags encountered. XLX also provides structures that describe the tag and character elements that can be parsed. These structures match the string identifying the element to callbacks that perform element specific actions. Using these tools reduces the creation of a new parser, in its simplest form, to the declaration of a chain of tag/character element tags, the associate callbacks and a simple *main()* function to invoke the parser.

This document describes how to create a new XML parser and then goes on to detail some of the internals of XLX.

# 1 Creating a new XML parser

XLX uses a simple stack inside the `state` structure to keep track of progress through the XML hierarchy. Upon encountering a known start tag in the XML file EXPAT invokes a callback that pushes the corresponding `tag` structure onto the stack and calls the `enter` function pointed to by that structure. If the `tag` is a character element then the corresponding `descriptor` structure is located, pushed onto the `stack`, and the `character` function pointed to by that structure is called. Should any of these function pointers be `NULL`, XLX will call the default specified in the `state` structure, assuming that the default pointer is non-`NULL`.

The simplest XML parser can thus be built on top of XLX by defining a set of tags, descriptors and associated callbacks, then writing a simple `main()` to create the parser and feed in the XML filename. By way of an example a simplified version of the *latc_parser* is shown below.

**0.** A function, `setAddr`, with the signature of an `enter` function is declared. This function sets the correct coordinate of the address structure at each start `tag` (where applicable).

**1.** One of several `tags` declared in this code snippet.

**2.** The `tag` structure is derived from another structure, `named`, as is the `descriptor` structure (see below). This ensures that all `tags` and `descriptors` start with a text string ID and type information (the enumeration `XLX_TAG` in this case). The `named` structure within the `tag` is defined within the inner set of brackets.

**3.** For some applications it is useful to be able to have recursive element, that is, an element contains another instance of itself. In this code snippet all the `tags` are defined to be non-recursive (the alternative uses the flag `XLX_RECURSIVE`).

**4.** Pointer to the base of an array containing pointers to all the `named` structures (`tags` and `descriptors`) that this `tag` can contain. `temChild` is not shown in this snippet, but will look very similar to `latChild` (line 10).

**5.** Pointer to the `enter` function for this `tag`, `setAddr`.

**6.** There is no `exit` function pointer for this `tag`. If a default has been defined then it will be used.

***7.*** An array of pointers to `named` structures that can be contained within the `lat tag`.

**8.** Pointer to the previously described `TEM tag`.

**9.** The array is terminated with a `NULL` pointer.

**10.** The root element of XML documents parsed by this parser is a proto-tag called `doc`, which contains the base elements (in this case a single element, `lat`). The parser starts with `doc` on the stack and never exits from it.

```
 0  void setAddr(state* xState, const char** attribute);

 1  static tag TEM = {
 2    {"TEM", XLX_TAG},
 3    XLX_NON_RECURSIVE,
 4    TEM_child,
 5    setAddr,
 6    NULL
    };

 7  static named* latChild[] = {
      (named*)(&TAM),
      (named*)(&ROI),
      (named*)(&TIE),
      (named*)(&SCH),
      (named*)(&AEM),
 8    (named*)(&TEM),
 9    NULL
    };

    static tag lat = {
      {"LAT", XLX_TAG},
      XLX_NON_RECURSIVE,
      latChild,
      NULL,
      NULL
    };

    static named* docChild[] = {
      (named*)(&lat),
      NULL
    };

10  static tag doc = {
      {"", XLX_TAG},
      XLX_NON_RECURSIVE,
      docChild,
      NULL,
      NULL
    };
```

**11.** This particular descriptor is for an element “engine_0”, which the auto-generated[1] structure name tells us is part of the TAM.

**12.** As with a `tag` the `descriptor` is derived from the common base class `named`. The type in this case is `XLX_DES` (rather than `XLX_TAG`).

**13.** No specific `character` function is defined, so the default for the parser will be used.

```
11  static descriptor TAM_register_engine_0 = {
12    {"engine_0", XLX_DES},
13    NULL}};
```

[1] The full versions of these tags and descriptors are auto-generated by an XLX-XML parser, reg_parser.

```
14  void makeMap (state* xState, const char* element, int len);
15  void makeTree(state* xState, const char* element, int len);

    int main(int argc, char** argv)
    {
      void*      fio  = NULL;
      LATCtree*  tree = NULL;
      LATCmap*   map  = LATC_newMap();
      state      lState;
      unsigned   i;

16    initState(&lState,
17              INITIAL_STACK_LIMIT, &doc,
18              NULL, NULL, makeMap,
19              0);
     for(i = 1; i < (argc - 1); ++i)
       parse(argv[i], &lState);

     tree = LATC_newByMap(map);
     initState(&lState,
               INITIAL_STACK_LIMIT, &doc,
               NULL, NULL, makeTree,
               0);

     for(i = 1; i < (argc - 1); ++i)
       parse(argv[i], &lState);

     fio = calloc(LATC_sizeForFIO(), 1);

     LATC_writeFile(tree, fio, argv[argc-1]);
     free(fio);
     free(tree);
     free(map);

     return 0;
    }
```

**14.** The *latc_parser* takes two passes to read an XML configuration file. The first pass identifies the components that have a configuration in the file. This information is used to create a densely packed configuration tree that can be populated during the second pass. A densely packed configuration tree is required for the LATC file writing functions.

**15.** The second `character` function populates the configuration tree.

**16.** The XLX `state` structure is initialized by this function call.

**17.** The `INITIAL_STACK_LIMIT` parameter can be used to ensure that the `stack` is as large as the anticipated depth of the XML hierarchy. The `stack` grows (exponentially) each time a call to `push` a new item onto the `stack` would exceed the limit. As mentioned earlier, the `stack` is seeded with a pointer to `doc tag`.

**18.** For the *latc_parser*, the default `enter` and `exit` function pointers are left `NULL`. For the first pass, all `tags` used the `character` function `makeMap`, for the second pass (line 64) `makeTree` is used.

**19.** In this case the verbosity level of 0 is specified (silent). Level 1 prints out each start and end element encountered.

The rest of the `main()` function is concerned with handling LATC specific data and will not be discussed here.

# 2 Inside the XLX parser

## 2.0 Tags

Each XML parser built on XLX has a vocabulary of XML elements. Inside the XLX parser these elements are represented by structures derived from the `tag` (for non-character elements) and `descriptor` (for character elements) structures. These structures associate the text string identifying the element with one or more functions defined by the author of the parser.

In fact the identifying string is held inside a base structure, `named`, from which both the `tag` and `descriptor` structure are derived. In addition to the ID string, the named structure contains a crude form of runtime type information, an enumeration, that allows the parser to cast pointers to `named` objects down to the correct type.

```
typedef enum{

  XLX_TAG  = 0,
  XLX_DES  = 1,
  N_XTYPES = 2

} xlxType;

typedef struct {

  const char* name; // String  identifying XML element
  xlxType     xType;// Identifies the type of the derived class

} named;
```

```
typedef void(*pCharacter)(state*, const char*, int);

typedef struct {

  named      id;        // Base class
  pCharacter character; // Unique action for parsing this tag

} descriptor;
```

Of the two types of element currently defined in XLX, the descriptor is the simplest. It simply associates a string with a pointer to a function that parser XML character elements. The underlying EXPAT library provides a pointer to the string and the number of characters it

contains, along with a user-defined pointer. The XLX parser casts the `void*` user pointer to a `state` pointer, and passes the arguments through to the parsing function.

The hierarchy of XML elements is embodied in the `tag` structure.

```
    typedef enum{

      XLX_NON_RECURSIVE  = 0,
      XLX_RECURSIVE      = 1

    } xlxSelf;

    typedef struct{

      named         id;
20    xlxSelf       self;
21    named**       child;
22    pEnter        enter;
      pExit         exit;

    } tag;
```

**20.** Sometimes it can be useful for an XML tag to contain another instance of itself, i.e. be recursive. If this is permitted then `self` should be set to `XLX_RECURSIVE`.

**21.** Each (non-character) element has a set of elements that it can contain. This information is captured as a `NULL` terminated list of pointers to `named` objects.

**22.** A pair of pointers to functions are used to trigger action upon entering and exiting a particular element. Note that the XLX framework is responsible for ensuring that a XML start or end tag is acceptable (the string from a start tag must match the name of an item in the `child` list, the string from an end tag must match the string in `id`).

## 2.0.0 The Proto-Tag

The XLX parser needs an entry point to the chain of `tags` and `descriptors`. The proto-tag provides this entry point. The parser is initialize inside this tag (as though the parser had just encountered an XML start tag with the corresponding ID) and never leaves (parsing stops when the parser exits a child of the proto-tag). Drawing on *latc_parser* as an example, the code snippet below shows the definition of the proto-tag (called `doc` in this case) with no string identifier.

```
static tag lat = {

  {"LAT", XLX_TAG},
  XLX_NON_RECURSIVE,
  latChild,
  NULL,
  NULL

};

static named* docChild[] = {

  (named*)(&lat),
  NULL

};

static tag doc = {

  {"", XLX_TAG},
  XLX_NON_RECURSIVE,
  docChild,
  NULL,
  NULL

};
```

## 2.1 State

The core of the XLX parser is the `state` structure. This structure contains any information that is shared across multiple `tags` or `descriptors`, as well as tracking progress through the chain of named structures (described in the previous section). It is anticipated that most XML parsers built on top of XLX will derive their own state structure from this base class.

```
typedef void(*pCharacter)(state*, const char*, int);

typedef void(*pEnter)(state*, const char**);

typedef void(*pExit)(state*);

typedef struct {

  stack      st;        // Stack of element ptrs
  pEnter     enter;     // Default start action
  pExit      exit;      // Default end action
  pCharacter character; // Default character element parsing
  int        verbose;   // Verbosity level of the parser

} state;
```

Default actions for entering and exiting a tag, or parsing a character element, are defined by setting the function pointers `enter`, `exit` and `character` in the state structure. The only other centrally maintained information is the verbosity level of the parser, which controls the amount of information printed to the screen while the parser is working.

```
void initState    (state*       this,
                   unsigned     stackLimit,
                   void*        stackSeed,
                   pEnter       enter,
                   pExit        exit,
                   pCharacter   character,
                   int          verbose);
```

The `state` structure is initialized via the `initState` function call. The `enter`, `exit`, `character` and `verbose` arguments to the function call are used to set the values of the elements of the structure, while `stackLimit` and `stackSeed` are passed into the `stack` initialization routine, which will be described in the next section.

```
void start        (void*        data,
                   const char*  element,
                   const char** attribute);

void end          (void*        data,
                   const char*  element);

void character    (void*        data,
                   const char*  element,
                   int          len);
```

The glue that joins the EXPAT library to XLX is provided by the three call-backs registered with EXPAT before parsing begins. The functions `start`, `end` and `character` are called when EXPAT encounters a start tag, end tag or character element respectively. The function `start` searches current `tag`'s list of children for the one corresponding to the start tag string. When the child is located it's pointer is pushed onto the `stack` and, if it is a `tag`, rather than a `descriptor`, the `enter` function is called. In the event that `enter` is `NULL`, the default `enter` function is called, provided one has been defined.

The function `end` verifies that the end tag string matches the name of the object on the top of the `stack` and then pops the object off. If the object is a `tag`, rather than a `descriptor`, the `exit` function is called. In the event that `exit` is `NULL`, the default `exit` function is called, provided one has been defined.

The function `character` parser first verifies that the object on the top of the `stack` is a `descriptor`, and then calls the object's `character` function. In the event that `character` is `NULL`, the default `character` function is called, provided that one has been defined.

## 2.2 Stack

The `state` structure contains a `stack` structure to keep track of the progress of the parser through the chain of `tags` and `descriptors`. The XLX `stack` is very basic. At initialization a chunk of memory is allocated, the size of which is defined by the `limit` argument to the `initStack` function call. This memory is treated as an array of `void*` pointers, called `base`. A call to `push` a pointer onto the `stack` sets the element at position `used` and increments `used`. If the array is already full (`used == limit`) then a new block of memory, twice the size of the original block, is allocated, the contents transferred from old to new and the new one deleted. The new addition is then `pushed` into the array.

A call to `pop` does the reverse of `push`; `used` is decremented and the last array element (now indexed by `used`) is returned. The function `peek` is essentially a non-destructive `pop`. The last array element (indexed by `used-1`) is returned but `used` is not altered.

```
typedef void(*pDelete)(void*);

typedef struct{

  unsigned limit; //Maximum number of objects the stack can hold
  unsigned used;  // Number of objects currently on the stack
  void**   base;  // Pointer to the base of the stack (1st item)

} stack;

unsigned    initStack   (stack*, unsigned limit);

void        endStack    (stack*);

void        destroyStack(stack*, pDelete);

unsigned    push        (stack*, void*  );

void*       pop         (stack*);

void*       peek        (stack*);
```

Once the `stack` is no longer required the resources it uses can be freed. The function `endStack` simply frees the memory used for the array of pointers and set the other `stack` elements to zero. The function `destroyStack` takes as an argument a pointer to a function that will be called once for each of the `used` items on the stack. Once this has been done it simply calls `endStack`.