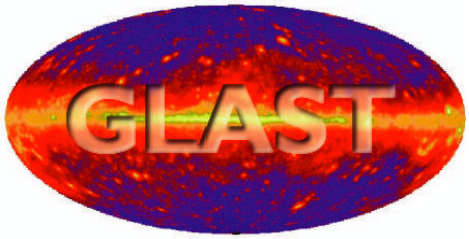

LAT Flight Software

reg_parser Manual

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 a brief overview of the reg_parser XML parser.

Contents

0 Introduction

0.0 Intended Audience

This document is intended to be a brief introduction to the reg_parser XML parsing application. This application auto-generates source code for LATC package, so only LATC developers should ever need to consult this document.

0.1 References

XLX Programmer's Guide, James Swain.

LATC Developer's Guide. James Swain.

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

reg_parser 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

reg_parser is an XML parsing application built on top of the XLX library. The application has one purpose: the automatic generation of source code for the LATC package based on the contents of one or more LAT register description XML files. The LRD files detail the components of the LAT, their hierarchy (which component contains which components, how many multiples of each can there be), the width of their registers, the number of registers and the widths and offsets of the fields within each register. The reg_parser application reads the LRD files and generates C source code declaring structures that describe the registers for the storage and manipulation of LATC configuration information. It also generates code declaring the tags and descriptors of another XML parser, latc_parser, that is used to create LAT configuration files from XML.

1 Invoking reg_parser

At the time of writing reg_parser should be invoked from the cmt directory of the LATC version being built.

```
    reg_parser lrd_gem.xml lrd_aem.xml tem.xml
```

The output files will be written to the LATC and src directories.

It is assumed that the xml files reside in the lrd directory of the LATC version being built.

1.0 CMT

At some point in the future reg_parser will be called by CMT as part of the LATC build process.

2 Inside reg_parser

2.0 Vocabulary

The reg_parser vocabulary is defined in XLX/src/reg_tags_p.h.

The vocabulary reg_parser is very limited.

1. There are only three non-trivial tags plus the proto-tag lrd and the document tag glat.
2. Each of the tags is an instance of the XLX base structure with no additional elements.
3. None of the tags has more than one child.
4. Only the `cpt` tag can be recursive.

Note, however, that there are five different start and end functions for the three tags. Much of the complexity of reg_parse is wrapped up inside these functions.

```
static named* fldChild[] = {NULL};

static tag fld = {
  {"field", XLX_TAG},
  XLX_NON_RECURSIVE,
  fldChild,
  startField,
  NULL,
};

static named* regChild[] = {(named*)(&fld), NULL};

static tag reg = {
  {"register", XLX_TAG},
  XLX_NON_RECURSIVE,
  regChild,
  startRegister,
  endRegister
};

static named* cptChild[] = {(named*)(&reg), NULL};

static tag cpt = {
  {"component", XLX_TAG},
  XLX_RECURSIVE,
  cptChild,
  startComponent,
  endComponent
};

static named* glatChild[] = {(named*)(&cpt), NULL};

static tag glat = {
  {"GLAT", XLX_TAG},
  0,
  glatChild,
  NULL,
  NULL
};

static named* lrdChild[] = {(named*)(&glat), NULL};

static tag lrd = {
  {"", XLX_TAG},
  0,
  lrdChild,
  NULL,
  NULL
};
```

2.1 State

The reg_parser uses a state structure derived from the XLX state structure.

```
 0  #define REG_NAME_LEN 40

 1  #define CPT_TIME_LEN 8

    typedef struct {

 2    state    pState;

 3    stack    cptDetail;

 4    L_head   cpt;
 5    L_head   addr;
 6    L_head   ndef;
 7    L_head   base_tags;

 8    char     time   [CPT_TIME_LEN];
 9    char     regName[REG_NAME_LEN];
      unsigned regId;
      unsigned nFld;


10    unsigned short addrRng[4];
      unsigned short index;

11    FILE*    structs;
      FILE*    enums;
      FILE*    tags;

12    RegStateFn* fns;

    } RegState;
```

0. For simplicity the maximum number of characters a component name can have is limit to 40.

1. The lifetime of a component is either STATIC or DYNAMIC, so eight characters is a sufficient length for this string.

2. To derive one structure from another in C, the base structure must be the first element of the structure.

3. In LAT register description XML files components can contain other components (TEM contains CCC, TCC, TIC). For each one of these components a cptDetail structure is created and pushed onto a stack.

4. While cptDetail is used to track the hierarchy of the components an absolute list of the component name strings, in the order they are encountered, is also need. The linked list cpt provides this.

5. For each component there is an accompanying four-element vector that describes the range of the elements in a valid LATCaddr for the component. This linked list contains strings that can be printed into a C source file to declare these ranges.

6. There is a total multiplicity for each component (essentially the multiple of the elements of the address range).

7. Some subset of the components are base tags, they are not contained by any other component and will appear on the list of grandchildren of the proto-tag. The linked list base_tags contains the name strings of these components.

Since register objects do not contain other register objects, only the last register tag encountered supplies state information, so this is just stored in the state structure.

8. When inside a register element the lifetime of the fields will either be STATIC or DYNAMIC.

9. The register name is arbitrarily limited to 40 characters.

10. The address range for a give component is derived from the multiplicity of that component and the multiplicities of all the components it is contained by. Thus there are 16 TEMs containing 4 CCCs so the address range of the CCC component is (16, 4, 0, 0). The addRng array is the accumulator of these multiplicities, while index tracks the depth through the XML hierarchy, which is equivalent to the index of the next addrRng element to be set.

11. Some of the auto-generated files are written to continuously during the parsing process; the contents of such files depends only on an individual tag and the state of the parser. These files are opened when the state is initialized and closed when the state is ended.

12. The filenames to be used for the auto-generated source code are passed into the initState function elements of a structure.

```
typedef struct {

  const char* hPath;   //!< Interface path //

  const char* type;    //!< latcType filename //
  const char* inc0;    //!< File giving TIME defs //

  const char* cPath;   //!< Implementation path //

  const char* enums;  //!< Enumerations //
  const char* desc;   //!< description //
  const char* tags;   //!< XML tags //
  const char* dtd;    //!< DTD //
  const char* tree;   //!< Definition of the LATCtree strcuture //
  const char* numb;   //!< File creating number array //
  const char* inc1;   //!< File giving LATCtree typedef //

  const char* liox;   //!< Association of LAT fns to types //
  const char* addr;   //!< Range of acceptable addresses //

  const char* inc2;   //!< File giving LatcDescriptor defn //
  const char* inc3;   //!< File giving LatcTag defn //

} RegStateFn;
```

2.2 Heavy Lifting

Use the force, read the source: reg_state.c, reg_cptTag.c, reg_regTag.c, reg_fldTag.c.