FieldExtract

EXTRACT REGISTER INFORMATION
FROM MICROCONTROLLER DOCUMENTATION

BY
MICHAEL J. MYRTER
B.S., Boston University, 2002

PROJECT

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science,
University of Alaska Fairbanks, 2011

TABLE OF CONTENTS

TABLE OF FIGURES

1. Introduction

By constructing two applications and one file format specification, this project delivers significant automation to the creation of software development tools for microcontrollers. FieldExtract gathers register bit field information from processor documentation. The application translates bit field information from manufacturer reference manuals to a standardized, easily consumable file format, FieldWarehouse. The second project application, FieldToHeader, uses the FieldWarehouse data to produce header files useful for programmatic access to register bit fields.

Rationale

Typically, creating high-level software for microcontrollers requires programmatic access to chip registers. Often, the programmer will use header files to represent and access register bit fields. The header file author finds all register information in the chip's documentation, distributed by the chip manufacturer.

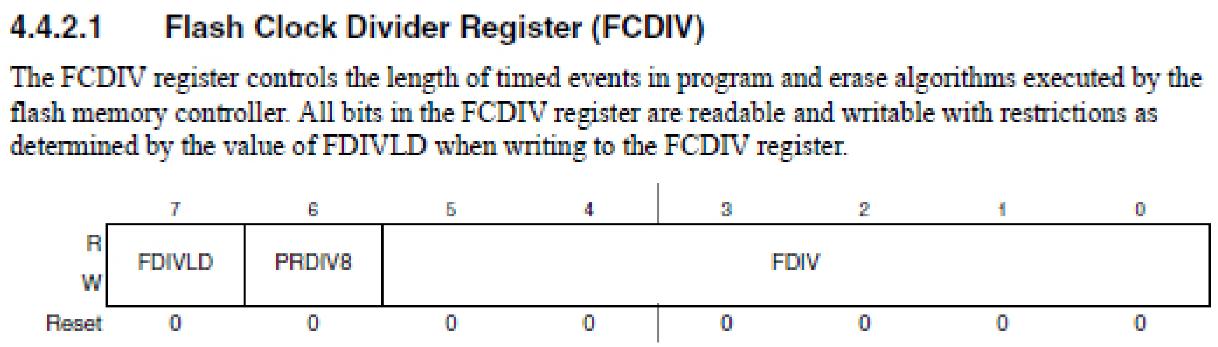
Authoring processor header files shares a common challenge with authoring any software tool that provides access to microcontroller registers, such as a debugger's register viewing functionality. The common challenge lies in the transfer of register bit field information from chip documentation to proposed software tool. A common viewpoint [1] is that this transfer is accomplished easily by manual extraction and typing. However, for complex devices with large register sets, transformation from documentation to software tool can be tedious and error prone. For example, the Freescale PowerPC 5554 Reference Manual [2] contains 1775 registers in 1194 pages. While manual extraction of several dozen registers is a sensible technique, it is not desirable to extract over one thousand registers by hand. Such is the case especially if the software tool is required to support a large number of microcontrollers, each potentially containing a unique peripheral register set. Further, given the continual release of new microcontrollers, the tedious transformation from document to tool may be frequent.

The project reduces the effort to transform chip documentation to development tool for register access. Areas of related work include information extraction and automated tool generation for embedded software. The primary focus is the extraction of bit field information from chip documentation. Using one target architecture, the Freescale ColdFire V1 family, FieldExtract lends significant automation to the extraction of register bit field information from device documentation. The secondary project focus is the file format FieldWarehouse and the application FieldToHeader, which transforms bit field information to a high-level language header file. Finally, the project shows how automated extraction of bit field information is useful to reduce the effort to create a variety of development tools for microcontrollers. A survey of previous work shows that

FieldExtract fills a gap in attempts to automate the creation of tools for embedded software.

Target Information

Within a typical microcontroller user manual, two common representations of register bit field information include the bit field diagram and the bit field description table. Figure 1 depicts a field diagram for a ColdFire V1 device, and Figure 2 is its corresponding field description table [3].

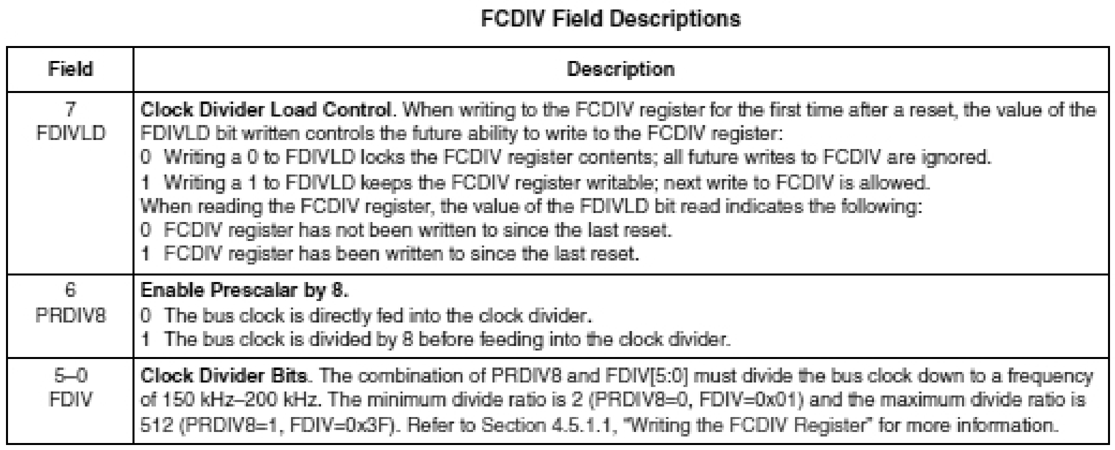


4.4.2.1 Flash Clock Divider Register (FCDIV)

The FCDIV register controls the length of timed events in program and erase algorithms executed by the flash memory controller. All bits in the FCDIV register are readable and writable with restrictions as determined by the value of FDIVLD when writing to the FCDIV register.

	7	6	5	4	3	2	1	0
R W	FDIVLD	PRDIV8	FDIV					
Reset	0	0	0	0	0	0	0	0

Figure 1, Example PDF Field Diagram



FCDIV Field Descriptions

Field	Description
7 FDIVLD	**Clock Divider Load Control.** When writing to the FCDIV register for the first time after a reset, the value of the FDIVLD bit written controls the future ability to write to the FCDIV register: 0 Writing a 0 to FDIVLD locks the FCDIV register contents; all future writes to FCDIV are ignored. 1 Writing a 1 to FDIVLD keeps the FCDIV register writable; next write to FCDIV is allowed. When reading the FCDIV register, the value of the FDIVLD bit read indicates the following: 0 FCDIV register has not been written to since the last reset. 1 FCDIV register has been written to since the last reset.
6 PRDIV8	**Enable Prescalar by 8.** 0 The bus clock is directly fed into the clock divider. 1 The bus clock is divided by 8 before feeding into the clock divider.
5–0 FDIV	**Clock Divider Bits.** The combination of PRDIV8 and FDIV[5:0] must divide the bus clock down to a frequency of 150 kHz–200 kHz. The minimum divide ratio is 2 (PRDIV8=0, FDIV=0x01) and the maximum divide ratio is 512 (PRDIV8=1, FDIV=0x3F). Refer to Section 4.5.1.1, "Writing the FCDIV Register" for more information.

Figure 2, Example PDF Field Description Table

For this project's target documents, the field diagram and field description table contain the majority of information that describes a register's bit fields. The project focuses on the automated extraction of the following information:

- Module or peripheral name
- Register name

- Register abbreviation
- Register size
- Register's defining document section
- Bit field abbreviation
- Bit field index
- Bit field size
- ASCII representation of bit field description

After FieldExtract assembles the above data in FieldWarehouse format, FieldToHeader consumes the data to produce a header file that is useful to access bit fields within source code. With the exception of the module or peripheral name, FieldExtract finds all of the information in the field description tables. However, this project will suggest a method to expand the extraction to include the field diagrams.

Preliminary Challenges

There are three major challenges to automating the extraction of register bit field information.

First, the general problem of automated table extraction is difficult and has been an active field of research since the 1990's [4]. The types of challenges to extraction vary with the target document format. For example, the Adobe PDF file format with its underlying PostScript language is a common format for microcontroller documentation. Table extraction from PDF is challenging [5] because the target document may not tag the table in PostScript. This leaves the automated consumer to determine both the existence of the table, its logical format, and its meaning solely from numeric page drawing commands and text.

The second major obstacle to automated extraction is that no standard format for microcontroller documentation exists. The table model is constrained only by likely conventions, such as the use of tables to display bit field descriptions. FieldExtract must distinguish between register information and unrelated information in the document. Relevant data, such as register endianness, may appear outside the register description in the documentation.

The final challenge is constructing algorithms to consume PostScript data for a large, complex microcontroller reference manual. PostScript is a page description language that is ostensibly human readable. However, for this project's target documents, the PostScript code is highly verbose and numeric. Such code makes it difficult to construct an extraction algorithm that relies on table extraction techniques like table cell position analysis.

2. FieldExtract Method

FieldExtract uses a variety of methods to gather bit field information from documentation, including target domain knowledge and techniques from the information extraction and table recognition and literature.

Within this project's initial target documents, the Freescale ColdFire V1 microcontroller reference manuals [6], all required bit field information exists in the documents' field diagrams and field description tables. Tables are used to describe bit fields in many chip reference manuals offered both by Freescale and other manufacturers, such as Texas Instruments [7]. Throughout the target documents, table structures are so similar as to make possible the task of automated extraction of bit field information. Indeed, the use of tables in chip reference manuals from other Freescale architectures closely mimics [2] that of the target architecture, ColdFire V1. This project uses the commonalities and conventions of table usage to construct an automated bit field extractor for the target data with the hope that the extraction can be extended to other target architectures and manufacturers. Thus, FieldExtract uses a narrowly defined extraction based on domain knowledge.

Essentially, the field diagram is a table structure with information possibly residing just outside of the table boundary, and the field description table is a simple table structure. Therefore, it is appropriate to look to the information extraction literature for techniques to locate and extract tables. Zanibbi et al [4] provide a comprehensive survey of table extraction projects through 2003. Using their classification of techniques, FieldExtract locates and extracts bit field information using the following techniques:

- Primitive structure recognition
    - Whitespace separation
    - Internal cell topology
    - Table captions
    - Entry index character structures
- Observational recognition using domain knowledge
    - Document parameter encoding
    - Logical table structure and syntax
- Data transformation
    - HTML translation
    - Tree transformation
    - Syntactic string matching
    - Data segmentation by data clustering

The following chapter will map these techniques to the project's implementation. FieldExtract's general approach is to use a limited set of target documents to construct an automated table extractor. Within the scope of the project, the extractor is useful for gathering register information for one microcontroller architecture and serves as a base for expansion to additional target architectures. This bottom-up approach contrasts with a top-down, general purpose table extraction technique. The primary reason behind using a bottom-up approach is to minimize the complexity of constructing a table extractor. Using the observation that tables are commonly used to display register information, a simple construction process for a table extractor will allow for rapid development of extractors for a large number of target architectures. If the target data is predictable and constrained, a complex, general-purpose table extractor may not be required. Indeed, according to Zanibbi, "one trend appears to be that table recognizers using detailed, narrowly defined models to recover tables in well-characterized document sets appear to perform their intended tasks best" [4]. Although very effective, top-down techniques are available [8], within the target domain, a simple bottom-up approach may lead to faster deployment and easier expansion of support to additional architectures. Regardless of the usefulness of FieldExtract for additional architectures, this project focuses on one and shows that FieldExtract is effective for automated bit field information extraction.

Project Context

Figure 3 depicts the context of the project with scope lines in grey.

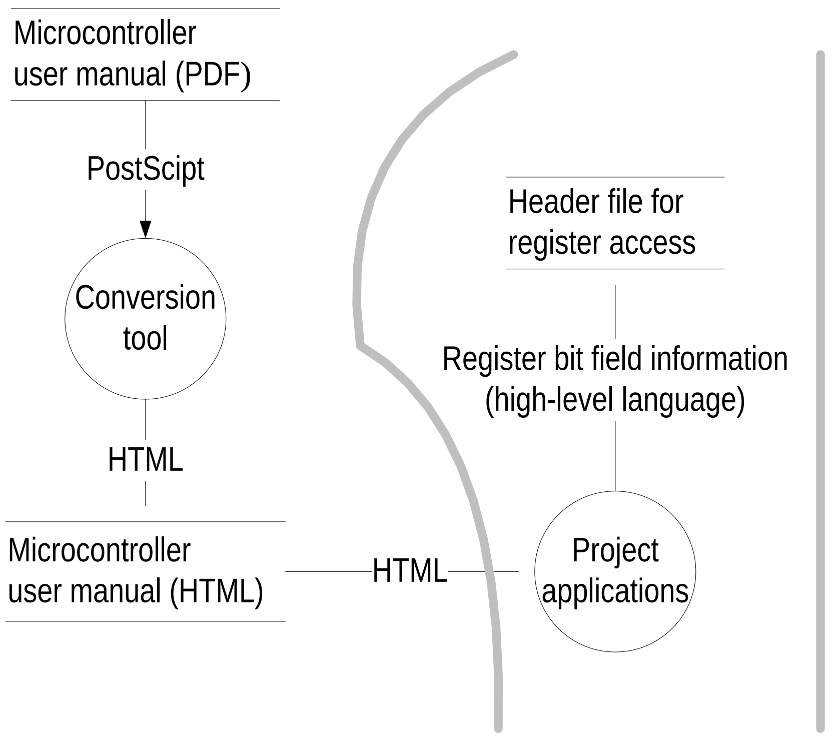

Microcontroller user manual (PDF)
PostScipt
Conversion tool
HTML
Microcontroller user manual (HTML)
HTML
Header file for register access
Register bit field information (high-level language)
Project applications

Figure 3, Project Context

Following the flow of data through this project from source document to target header file, the path begins with the microcontroller reference manual. This is commonly published in Adobe PDF format. Because of the challenges of consuming the underlying PostScript code, the project sought a data transformation that would both simplify the consumption of data and order the data in a manner that is useful for table extraction. HTML is an obvious choice for this role. The markup language is both simple to consume and table oriented. Further, a variety of tools exist for conversion between PDF and HTML. This project's scope begins with the HTML representation of the PDF document. By freeing the project from the complex task of general-purpose table extraction from PDF, the project focuses on table extraction in a simpler format using syntactic and semantic analysis. Finally, the scope lines permit future support of target documents in formats other than PDF using the intermediate HTML transformation.

3. Implementation

Project Elements

Figure 4 shows the data flow through the elements of this project with context lines in grey.

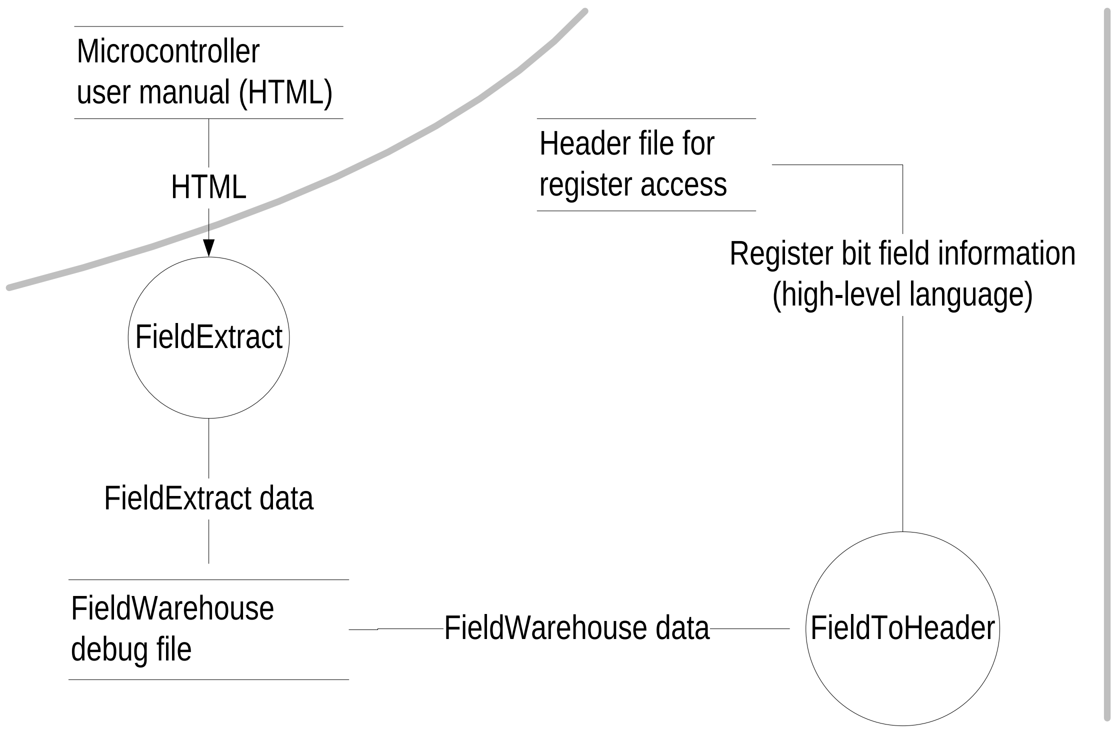

Microcontroller user manual (HTML)
HTML
FieldExtract
FieldExtract data
FieldWarehouse debug file
FieldWarehouse data
FieldToHeader
Register bit field information (high-level language)
Header file for register access

Figure 4, Data Flow

There are three major elements to the project:

- FieldWarehouse is an architecture independent, expandable, human readable file format that describes register bit field information. The Appendix contains the specification for this format.

- FieldToHeader converts a FieldWarehouse file to a high-level header file useful for programmatic access to register bit fields. For this project, FieldToHeader produces a C family header file suitable for the target architecture, ColdFire V1.

- FieldExtract is the primary focus of this project. The application extracts bit field information from an HTML representation of a microcontroller user manual. For this project, FieldExtract supports the ColdFire V1 documents only. Further, as discussed below, the only supported HTML translation is that of Adobe Acrobat X [9].

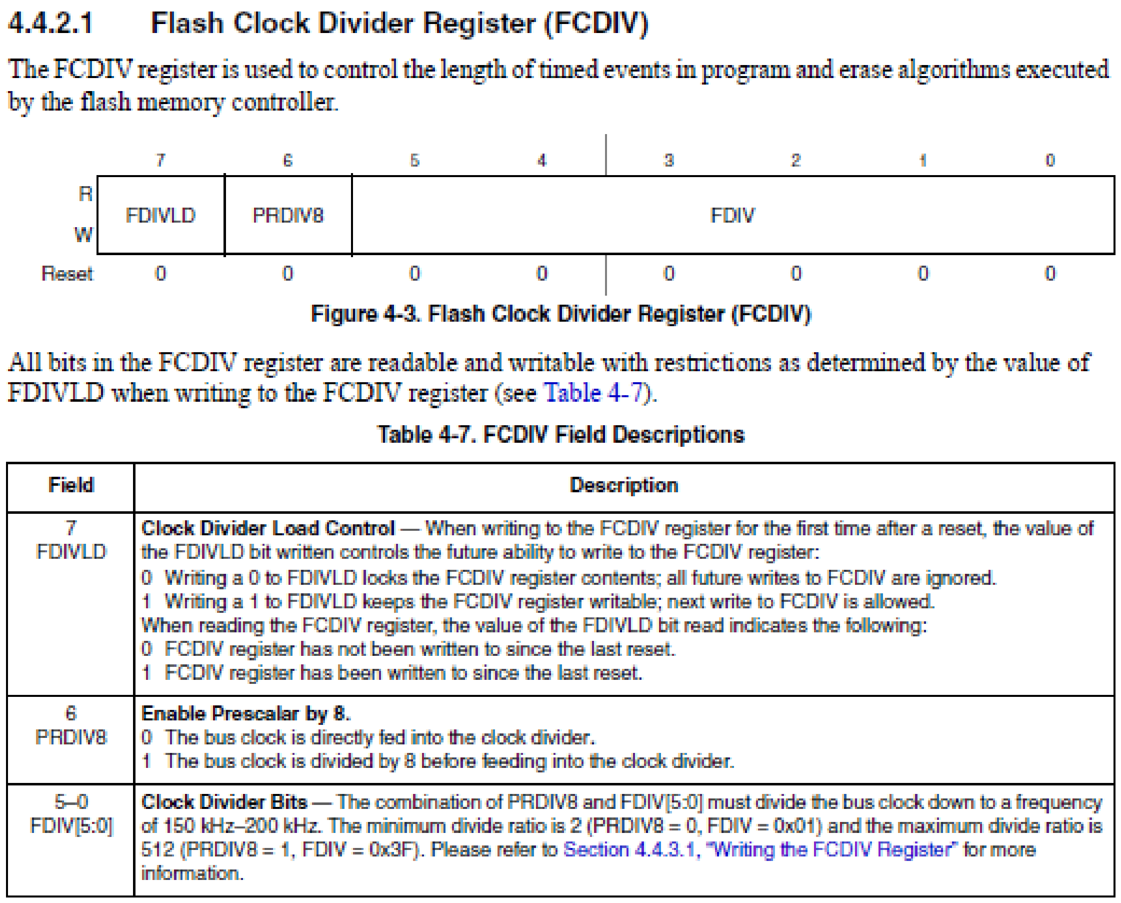
Zanibbi outlines three roles in the table extraction process: parser, segmenter, and classifier [4]. All roles utilize the table extraction techniques listed previously. Segmenters determine the existence and scope of table structures in the target data. For example, "Here is a table, and here is where it ends." Based on a table model, classifiers assign structure and relation types to target data. For example, "Here is a string that indicates a bit field index for ColdFire V1." Using segmenters and classifiers, parsers define inputs for analysis and produce parsing graphs of the target data.

Following the data flow from the source document, the initial role consists of the HTML data transformation. The project uses Acrobat X to segment and classify the source document contents based on a document table model. In a sense, every group of similar microcontroller documents, grouped by microcontroller architecture, is an individual table model. Each model uses variations on bit field representations. The project relies on the PDF to HTML translator's segmentation and classification of elements related to the field description tables.

The next role in the project data flow is that of parser. Using the segmentation and classification from the HTML transformation, FieldExtract parses the HTML into a tree structure based on the World Wide Web consortium's Document Object Model (DOM) [10]. By traversing the tree that represents the HTML, FieldExtract further classifies and segments the data that relate to and represent the field description tables. FieldExtract analyzes the tree data, and the table extraction is complete. FieldExtract produces the FieldWarehouse file, easily consumable by FieldToHeader.

Example Data Flow

The following is an example of the document translation and data flow from PDF user manual to the FieldToHeader output file. Figure 5 shows the field diagram and field description tables from the original PDF document [3].



4.4.2.1 Flash Clock Divider Register (FCDIV)

The FCDIV register is used to control the length of timed events in program and erase algorithms executed by the flash memory controller.

	7	6	5	4	3	2	1	0
R / W	FDIVLD	PRDIV8	FDIV					
Reset	0	0	0	0	0	0	0	0

Figure 4-3. Flash Clock Divider Register (FCDIV)

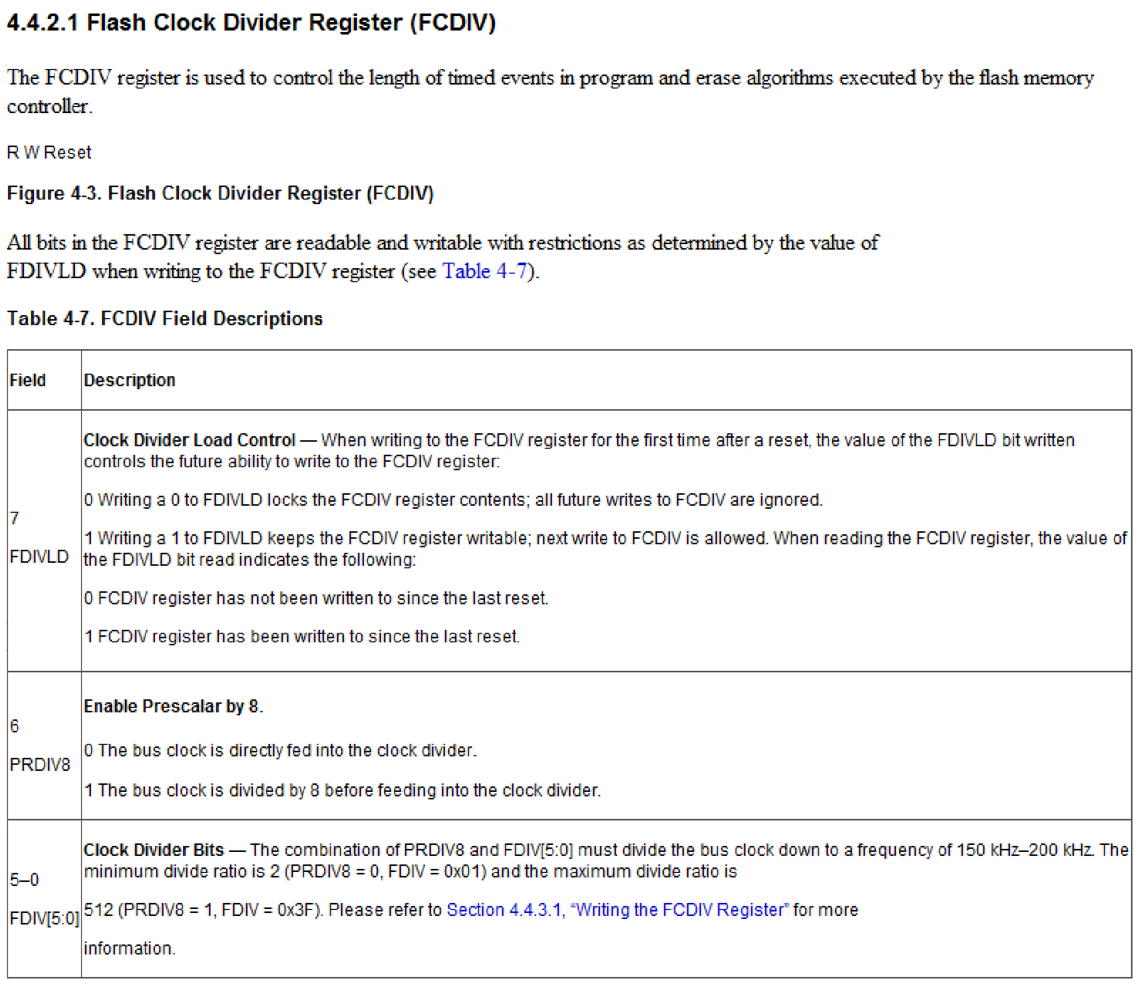
All bits in the FCDIV register are readable and writable with restrictions as determined by the value of FDIVLD when writing to the FCDIV register (see Table 4-7).

Table 4-7. FCDIV Field Descriptions

Field	Description
7 FDIVLD	**Clock Divider Load Control** — When writing to the FCDIV register for the first time after a reset, the value of the FDIVLD bit written controls the future ability to write to the FCDIV register: 0 Writing a 0 to FDIVLD locks the FCDIV register contents; all future writes to FCDIV are ignored. 1 Writing a 1 to FDIVLD keeps the FCDIV register writable; next write to FCDIV is allowed. When reading the FCDIV register, the value of the FDIVLD bit read indicates the following: 0 FCDIV register has not been written to since the last reset. 1 FCDIV register has been written to since the last reset.
6 PRDIV8	**Enable Prescalar by 8.** 0 The bus clock is directly fed into the clock divider. 1 The bus clock is divided by 8 before feeding into the clock divider.
5–0 FDIV[5:0]	**Clock Divider Bits** — The combination of PRDIV8 and FDIV[5:0] must divide the bus clock down to a frequency of 150 kHz–200 kHz. The minimum divide ratio is 2 (PRDIV8 = 0, FDIV = 0x01) and the maximum divide ratio is 512 (PRDIV8 = 1, FDIV = 0x3F). Please refer to Section 4.4.3.1, "Writing the FCDIV Register" for more information.

Figure 5, PDF Example Document

Figure 6 shows the document after translation from Acrobat X. Note that the field diagram is not shown. Although the field description table has been faithfully reproduced as an HTML table, the field diagram is converted to a bitmap image. Image analysis is beyond the scope of the project, and the image is discarded.



4.4.2.1 Flash Clock Divider Register (FCDIV)

The FCDIV register is used to control the length of timed events in program and erase algorithms executed by the flash memory controller.

R W Reset

Figure 4-3. Flash Clock Divider Register (FCDIV)

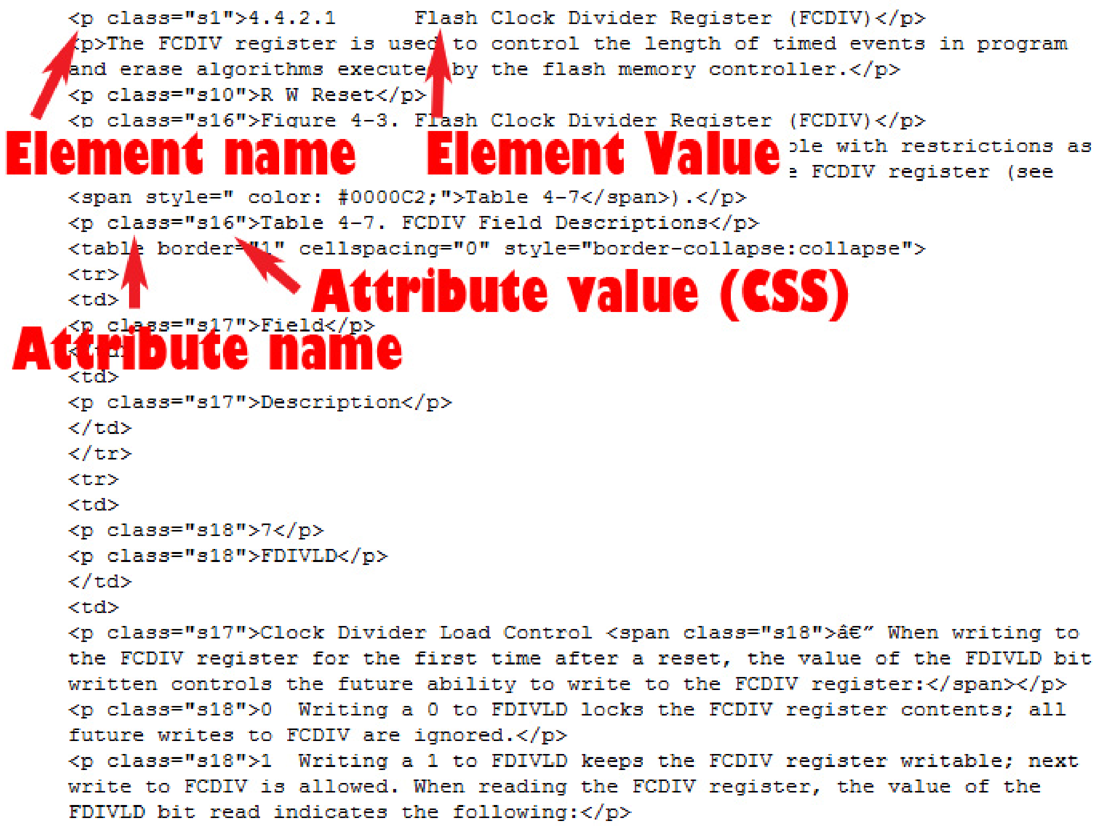
All bits in the FCDIV register are readable and writable with restrictions as determined by the value of FDIVLD when writing to the FCDIV register (see Table 4-7).

Table 4-7. FCDIV Field Descriptions

Field	Description
7 FDIVLD	**Clock Divider Load Control** — When writing to the FCDIV register for the first time after a reset, the value of the FDIVLD bit written controls the future ability to write to the FCDIV register: 0 Writing a 0 to FDIVLD locks the FCDIV register contents; all future writes to FCDIV are ignored. 1 Writing a 1 to FDIVLD keeps the FCDIV register writable; next write to FCDIV is allowed. When reading the FCDIV register, the value of the FDIVLD bit read indicates the following: 0 FCDIV register has not been written to since the last reset. 1 FCDIV register has been written to since the last reset.
6 PRDIV8	**Enable Prescalar by 8.** 0 The bus clock is directly fed into the clock divider. 1 The bus clock is divided by 8 before feeding into the clock divider.
5–0 FDIV[5:0]	**Clock Divider Bits** — The combination of PRDIV8 and FDIV[5:0] must divide the bus clock down to a frequency of 150 kHz–200 kHz. The minimum divide ratio is 2 (PRDIV8 = 0, FDIV = 0x01) and the maximum divide ratio is 512 (PRDIV8 = 1, FDIV = 0x3F). Please refer to Section 4.4.3.1, "Writing the FCDIV Register" for more information.

Figure 6, Example HTML Transformation

Figure 7 shows the HTML source for the beginning of this document section. The words in red refer to DOM and XML terminology. The element name is the common paragraph tag, and its value is the text for the name of the register. The attribute name "class" indicates the use of an HTML cascading style sheet (CSS), and the CSS name is "s16". The CSS is a reusable class, defined at the beginning of the document, which refers to a set of presentation details for the enclosed element value. FieldExtract uses the CSS information to detect target information, described below.

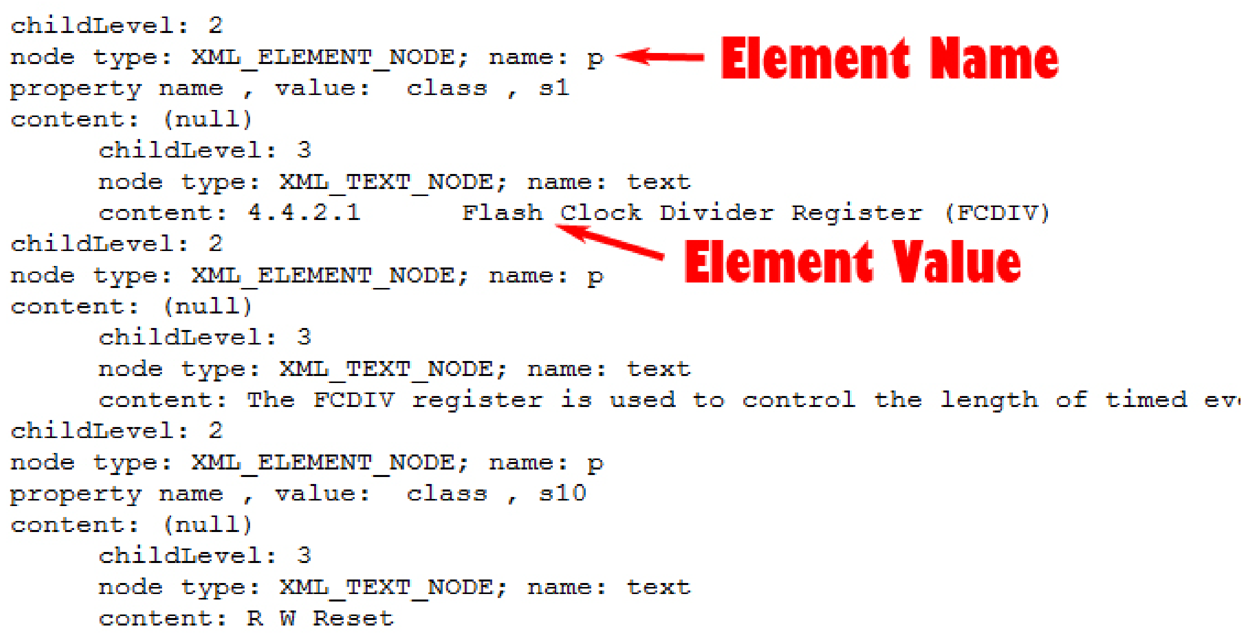


```
<p class="s1">4.4.2.1      Flash Clock Divider Register (FCDIV)</p>
<p>The FCDIV register is used to control the length of timed events in program
and erase algorithms executed by the flash memory controller.</p>
<p class="s10">R W Reset</p>
<p class="s16">Figure 4-3. Flash Clock Divider Register (FCDIV)</p>
                                                  ble with restrictions as
                                                  e FCDIV register (see
<span style=" color: #0000C2;">Table 4-7</span>).</p>
<p class="s16">Table 4-7. FCDIV Field Descriptions</p>
<table border="1" cellspacing="0" style="border-collapse:collapse">
<tr>
<td>
<p class="s17">Field</p>

<td>
<p class="s17">Description</p>
</td>
</tr>
<tr>
<td>
<p class="s18">7</p>
<p class="s18">FDIVLD</p>
</td>
<td>
<p class="s17">Clock Divider Load Control <span class="s18">â€" When writing to
the FCDIV register for the first time after a reset, the value of the FDIVLD bit
written controls the future ability to write to the FCDIV register:</span></p>
<p class="s18">0  Writing a 0 to FDIVLD locks the FCDIV register contents; all
future writes to FCDIV are ignored.</p>
<p class="s18">1  Writing a 1 to FDIVLD keeps the FCDIV register writable; next
write to FCDIV is allowed. When reading the FCDIV register, the value of the
FDIVLD bit read indicates the following:</p>
```

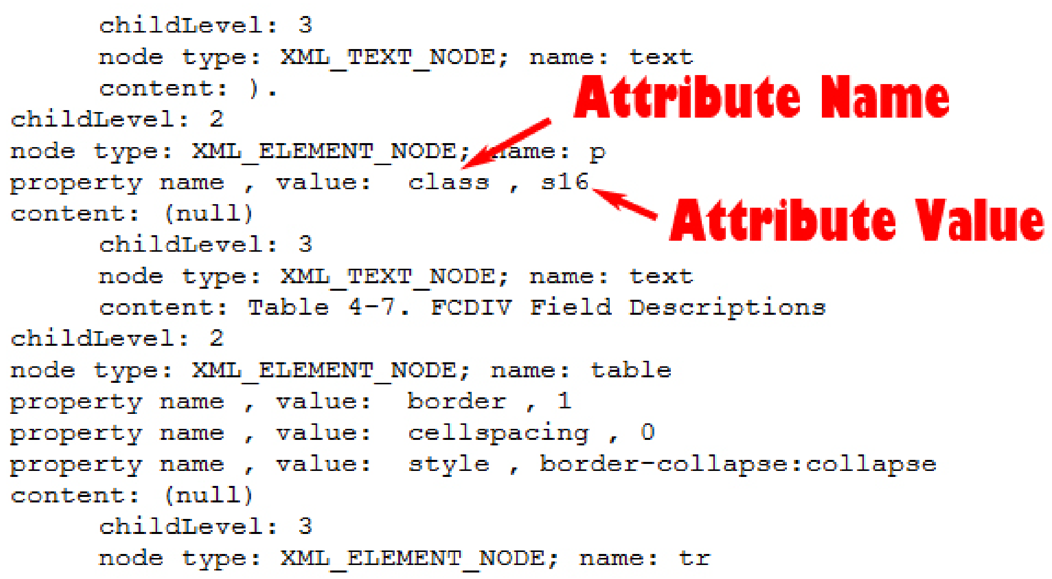
Figure 7, Example HTML Source

Figures 8 and 9 show the DOM tree output from FieldExtract, which is available to the user for debugging. The element name, element value, attribute name, and attribute value from above are shown.



```
childLevel: 2
node type: XML_ELEMENT_NODE; name: p
property name , value:  class , s1
content: (null)
        childLevel: 3
        node type: XML_TEXT_NODE; name: text
        content: 4.4.2.1      Flash Clock Divider Register (FCDIV)
childLevel: 2
node type: XML_ELEMENT_NODE; name: p
content: (null)
        childLevel: 3
        node type: XML_TEXT_NODE; name: text
        content: The FCDIV register is used to control the length of timed ev
childLevel: 2
node type: XML_ELEMENT_NODE; name: p
property name , value:  class , s10
content: (null)
        childLevel: 3
        node type: XML_TEXT_NODE; name: text
        content: R W Reset
```

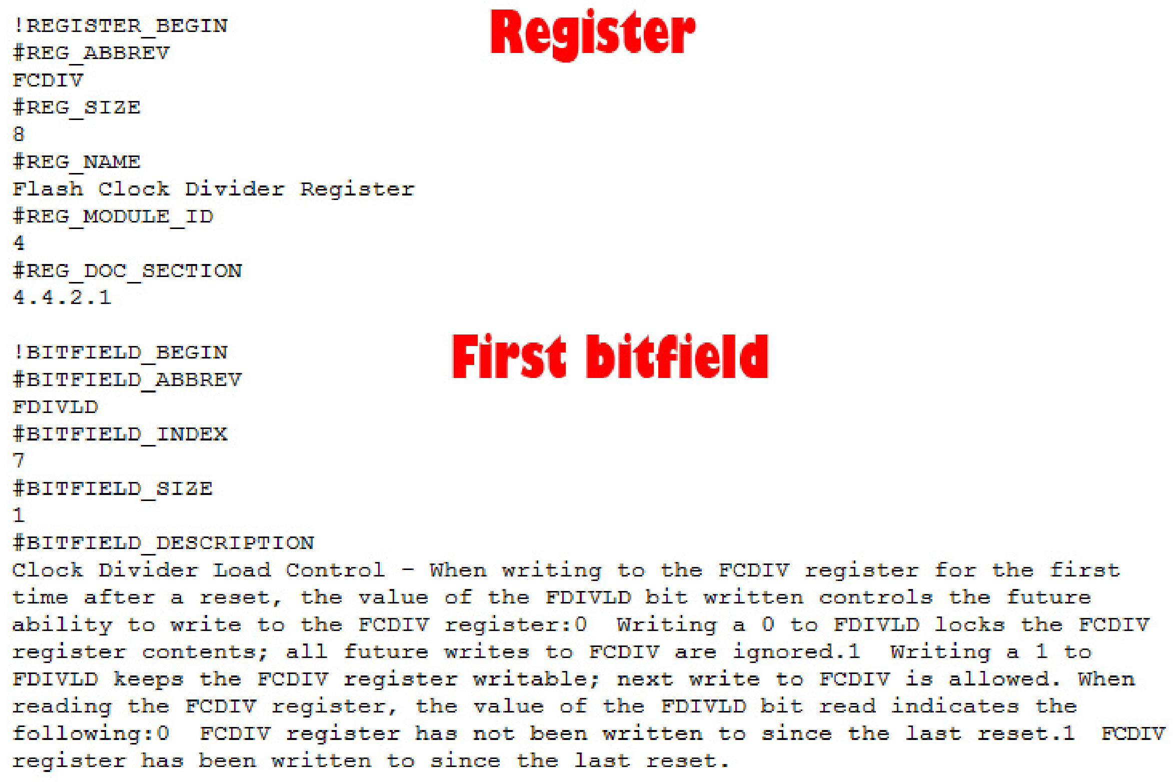
Figure 8, Example DOM Elements



```
        childLevel: 3
        node type: XML_TEXT_NODE; name: text
        content: ).
childLevel: 2
node type: XML_ELEMENT_NODE; name: p
property name , value:  class , s16
content: (null)
        childLevel: 3
        node type: XML_TEXT_NODE; name: text
        content: Table 4-7. FCDIV Field Descriptions
childLevel: 2
node type: XML_ELEMENT_NODE; name: table
property name , value:  border , 1
property name , value:  cellspacing , 0
property name , value:  style , border-collapse:collapse
content: (null)
        childLevel: 3
        node type: XML_ELEMENT_NODE; name: tr
```

Figure 9, Example DOM Attributes

Figure 10 shows the FieldWarehouse output from FieldExtract. FieldWarehouse is described below and detailed in the Appendix.

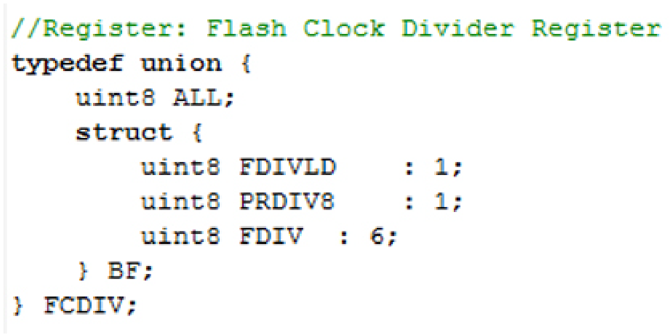


```
!REGISTER_BEGIN
#REG_ABBREV
FCDIV
#REG_SIZE
8
#REG_NAME
Flash Clock Divider Register
#REG_MODULE_ID
4
#REG_DOC_SECTION
4.4.2.1

!BITFIELD_BEGIN
#BITFIELD_ABBREV
FDIVLD
#BITFIELD_INDEX
7
#BITFIELD_SIZE
1
#BITFIELD_DESCRIPTION
Clock Divider Load Control - When writing to the FCDIV register for the first
time after a reset, the value of the FDIVLD bit written controls the future
ability to write to the FCDIV register:0  Writing a 0 to FDIVLD locks the FCDIV
register contents; all future writes to FCDIV are ignored.1  Writing a 1 to
FDIVLD keeps the FCDIV register writable; next write to FCDIV is allowed. When
reading the FCDIV register, the value of the FDIVLD bit read indicates the
following:0  FCDIV register has not been written to since the last reset.1  FCDIV
register has been written to since the last reset.
```

Figure 10, Example FieldWarehouse Output

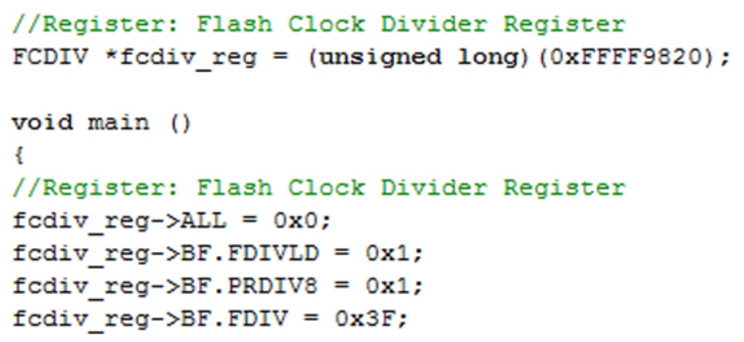
Figure 11 shows the output from FieldToHeader for this register.



```
//Register: Flash Clock Divider Register
typedef union {
    uint8 ALL;
    struct {
        uint8 FDIVLD    : 1;
        uint8 PRDIV8    : 1;
        uint8 FDIV  : 6;
    } BF;
} FCDIV;
```

Figure 11, Example FieldToHeader Output

Figure 12 shows the use of this header file in a ColdFire application's source code.



```
//Register: Flash Clock Divider Register
FCDIV *fcdiv_reg = (unsigned long)(0xFFFF9820);

void main ()
{
//Register: Flash Clock Divider Register
fcdiv_reg->ALL = 0x0;
fcdiv_reg->BF.FDIVLD = 0x1;
fcdiv_reg->BF.PRDIV8 = 0x1;
fcdiv_reg->BF.FDIV = 0x3F;
```

Figure 12, Example ColdFire Application

Extraction Techniques

Using Zanibbis extraction classifications, the following are examples of FieldExtract's techniques to gather bit field information from field description tables. Generalizations regarding the data are subject to an acceptable number of extraction failures caused by an anomaly in the data or a limitation in the FieldExtract algorithms. Results and difficulties are discussed in subsequent sections.

- Table Captions - Table captions in the target documents tended to appear in the same location with respect to the table. Bit field description table captions generally adhered to a consistent string pattern.

- Internal Cell Topology and Table Entry Index Structure - Bit field indexes generally adhered to a predictable pattern with respect to text location and character usage.

- Logical Table Structure and Syntax - Most bit field description tables in the target documents adhered to a simple two column table with rows representing a single bit field.

- Whitespace Separation - FieldExtract depends on whitespace and newline characters in a variety of ways. For example, to distinguish a bit field index from its bit field abbreviation, the target document places these on separate lines within a table cell.

- Document Parameter Encoding Using Domain Knowledge - The use of cascading style sheets to consistently indicate a target datum was crucial to the success of the project. For example, register and chapter document titles always use the same small set of CSS names, reflecting the consistent use of font. The CSS names are the domain specific document parameters.

- Syntactic String Matching - String matching is another crucial FieldExtract technique. For example, register document section titles can be identified using a small number of string search keys.

- Data Segmentation by Clustering - Although extensive character-level clustering analysis was not required for field description tables, it would be useful to further analyze the ASCII description entry of each bit field. This entry may be lengthy and may utilize nested table structures. For example, a sub-table may indicate the meaning of possible bit field values. Additionally, field diagrams may include clusters of characters bordering the exterior of the table, such as bit field reset values.

The following is a high-level list of the most important FieldExtract tasks and their relation to the extraction techniques above:

- Locate module, locate register - These tasks utilize string matching, CSS parameters, and the DOM tree transformation to detect document section titles that correspond to the beginning of module and register sections.

    - FieldExtract gathers module names, register names, and register abbreviations from these titles.
    - The relative position of target data within the DOM tree provides structure to simplify searching and analysis. For example, the parent of a register name is always an HTML paragraph tag whose CSS name is consistent within one document.

- Locate field description table, extract bit field information - These tasks utilize string matching on table captions and cell contents, consistent cell topology and index structure using whitespace separation, and the DOM tree transformation.

    - String matching is useful on cell content to extract bit indexes and abbreviations from a consistent pattern within the cells.
    - The DOM tree provides structured traversals of the table cells, where rows are children of the HTML table tag. Columns are children of the rows, and columns within a row are siblings.

Finally, semantic information is useful in situations like determining the size of registers and the location of unimplemented bit fields. In our target documents, the register and bit field sizes are not stated explicitly in the field description tables. Generally, these sizes are expressed graphically in the field diagram. FieldExtract infers register size by the highest bit field index in the field description table. For another example, unimplemented bit fields may be expressed as a separate entry in the bit field description table or may be omitted. FieldExtract detects the latter case by looking for missing bit field indexes within the table.

Challenges

The most serious implementation challenge in this project was finding a sufficiently accurate tool to transform PDF data to the HTML table structures. Although numerous tools are available, the quality of these tools varies widely with respect to table extraction. Some tools produce errant output, and some produce images of tables instead of HTML tables. Further, tools have varying sensitivity to source document errors. Acrobat X was chosen as the transformation tool because it faithfully transformed all of the field description tables from PDF to HTML table structures.

One drawback to the Acrobat X conversion is its transformation of the field diagram table into a bitmap image instead of an HTML table for the target documents. This eliminated the automated extraction of bit field reset values and bit field read/write restrictions. Ultimately, this information is not required for the final product, the high-level header file representing the bit fields. By augmenting the transformation with another tool, it would be possible to extract the field diagram information. For example, the open source application pdftotext [11] faithfully reproduces the field diagrams in ASCII for the target documents. This transformation eliminates the use of CSS and DOM tree information for extraction, but this is not significant for extracting bit field reset values and read/write capabilities.

Another related, serious implementation challenge was the lack of a field description table for registers that included only the field diagram. This was the greatest source of failure in the results of the project. However, the number of cases was sufficiently small and did not seriously threaten the usefulness of FieldExtract. Further, recovery of the field diagram using pdftotext would eliminate these failures.

Document sections representing more than one register presented another challenge. In some cases, multiple registers shared all bit field information. In other cases, one register document section included multiple field description tables. FieldExtract is able to analyze the first field description table and warn of the additional tables, but it does not extract the additional tables.

The remaining extraction problems involved miscellaneous issues like typographical errors, single anomalies in the table structures, multiple CSS values for document section names, and missing table captions. If the number of problems were sufficient to merit changing the extraction algorithms, improving FieldExtract's algorithms could solve some of these issues.

Additional Implementation Details

libxml2

This open source collection of tools [12] for HTML and XML implements the Document Object Model. FieldExtract relies on this library to form the HTML parse tree and to provide the API to traverse the DOM tree and analyze its content.

Error detection and correction

FieldExtract can detect a variety of errors and anomalies. The user is warned if a module contains no registers or a register contains no bit fields. As noted above, inconsistent indexes are reported, and unimplemented bit fields are inserted where appropriate. UTF8 exclusive characters are converted to appropriate ASCII characters, such as converting m-dash and n-dash characters to hyphens.

Testing and debug output

This project does not seek to eliminate the need for direct inspection of FieldExtract output. The first reason is that general table extraction is not perfect, and no extraction method is likely to emerge that is perfect even within the narrow domain of microcontroller user manuals. The second reason that direct inspection is necessary is that information like bit field indexes and sizes must be extracted at a very high level, if not perfect level, of accuracy.

FieldExtract produces two items that are useful for direct inspection of output, regression testing, and debugging of extraction algorithms. FieldExtract's extensive debug output details the extraction of a document, which is useful both for tracing the cause of extraction algorithm failures as well as for viewing warnings regarding a particular extraction. Secondly, FieldWarehouse provides a succinct, easily readable format to compare the source document with the bit field extractions.

FieldWarehouse

The FieldWarehouse file format is an expandable, architecture independent representation of register and bit field information. FieldWarehouse is ASCII, line delimited, and designed for ease of reading and modification. The intention of the format is to facilitate a simple, well-defined, easily consumable file to provide bit field information to automated tools generators, such as FieldToHeader or a microcontroller debugger. The draft format in the Appendix is sufficient for this project only but could be expanded easily.

FieldToHeader

Transforming a FieldWarehouse file to a high-level language header file, FieldToHeader is an example of a variety of automated tools that could consume FieldWarehouse data. FieldToHeader produces C-style structs but could be expanded to produce a variety of header styles, languages, and compiler-specific output.

4. Experimental Analysis

The following definitions apply to assessing the results of this project:

- Register Entry - a contiguous document chunk that defines a single target register or multiple target registers. One register entry may define multiple registers.
- Accurate Register Identification – occurs when FieldExtract correctly identifies the following for a given register entry: module name, register name, register abbreviation, register size, register's defining document section (chapter/section number), bit field abbreviation, bit field index within register, bit field size, and ASCII representation of every bit field description.
- Inaccurate Register Identification – occurs when FieldExtract skips or incorrectly interprets one or more above items for a genuine register entry.
- False negative - inaccurate register or bit field identification; skipped or mangled information.
- False positive – occurs when FieldExtract identifies a non-genuine register or bit field entry as a genuine entry.

The following definitions are commonly used in the information extraction literature and provide a comparison with previous work [4]:

- Precision = correctly identified / (correctly identified + false positives)
- Recall = correctly identified / (correctly identified + false negatives)
- F-measure = (2 * recall * precision) / (recall + precision)

Based on the following results, FieldExtract compares favorably with past work in table extraction. FieldExtract achieves an F-measure of 88.4%, and past works' F-measures range from the low 80% to the mid 90% [8].

- Total number of pages: 4739
- Total register entries: 1550
- False positive: 13
- False negative: 310
- Precision: 98.9%
- Recall: 80.0%
- F-measure: 88.4%

Figure 13 illustrates precision, recall, and F-measure for the complete ColdFire V1 document set. The very high precision values and satisfactory recall values suggest that FieldExtract could be more aggressive in its identification of registers. Although a potentially higher false positive does not guarantee a lower false negative, the 18.9% disparity between precision and recall suggests an algorithmic change should be considered.

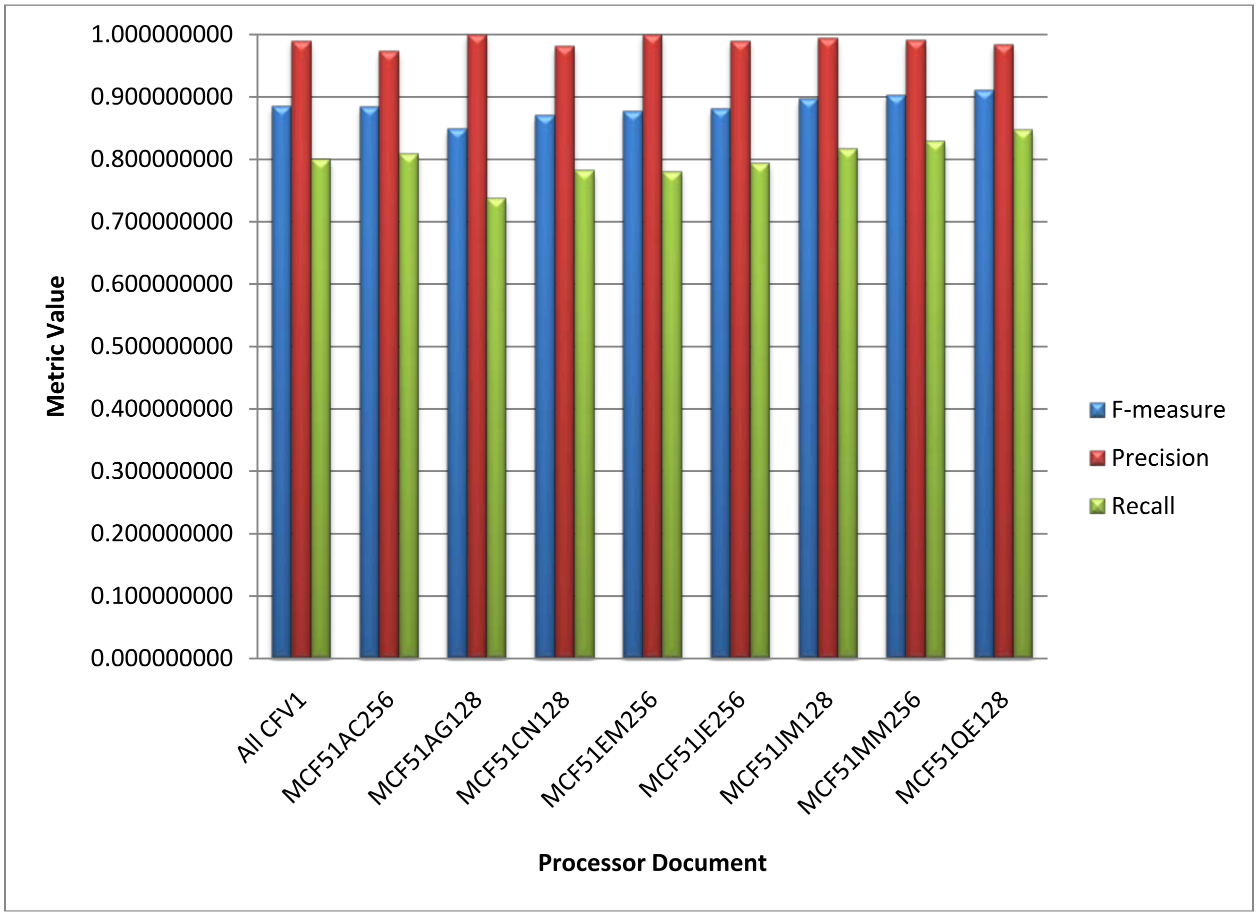


Figure 13, F-measure, Precision, and Recall for FieldExtract

5. Past Work, Future Work, and Conclusion

Past Work

This project relates to previous work in table extraction and automated tool creation. Embedded software development is an area with many opportunities for bringing automation to tool creation, such as debuggers, simulators, and hardware driver development. One challenge to automation is the difficulty in establishing an easily consumable format that describes the pertinent information from the microprocessor. A universal format is not necessary, but tool developers need well defined, easily consumable formats. There have been numerous attempts at providing development tools with a standard format for representing registers, such as the HAIL project [1]. However, none have emerged as a de facto standard. What has been missing from previous efforts is the automation of the extraction of the underlying bit field information from the microprocessor manual. Indeed, it is not uncommon to encounter the attitude that register information "can be easily translated from the device documentation" [1]. Whereas this might be true of a simple register format within one document, when the goal is to support a large and growing number of complex devices, translation is tedious and error prone when performed manually.

According to one classification [13], table extraction is a subset of information retrieval, information extraction, and information integration. Retrieval locates documents containing target data for extraction. Information integration aims to gather individual datum from heterogeneous sources to build a consistent view of the information.

Zanibbi et al provide a comprehensive survey of table extraction techniques through 2003 [4]. Table recognition models are either static or adaptive based on run time parameters or input data. Categories of advanced models include grammars for tables of contents, grammars for table lines, regular expressions applied to cell contents, parameter setting based on machine learning or human interaction, and knowledge encoded graphs applied to cell contents. The survey observes that extraction parameters constrain the underlying table model, and that the constraints apply to assumptions about the table locations, structures, and contents. This is the inherent weakness of a top-down, general purpose approach to table extraction. If the target domain lies outside of the assumptions, the particular complex, top-down approach may be useless.

Table extraction literature begins in the mid-1990's with plain text extractions. An early approach [14] recognized the importance of cell content in table identification by using natural language processing techniques from linguistics. The technique segmented content using plumb lines to separate clusters of text. Other important contributions of this study include the application of syntax and semantics to table structures and the use of domain specific knowledge of the table content. This work applied to legacy construction documents.

HTML table extraction begins in the early 2000's. One of the first works applied to data mining on the web [15]. The challenge is that an HTML table tag may not indicate a table but may be used to facilitate page display. The work used simple heuristics to distinguish genuine tables, for example, a genuine table must have had at least two cells. The number of links, figures and forms must not have exceeded a threshold value. Content analysis compared string similarities within table rows and columns.

Several machine learning based approaches to table extraction have emerged. One of the first approaches [16] used approximately 1400 HTML documents for ground truth and training. The training process calculated 16 features based on table layout, content, and word grouping. Another machine learning approach used feature calculations based on sparse text line detection [17]. A sparse text line was defined as having met two conditions. First, pairs of consecutive words must be spaced outside of a threshold, and second, the length of the line is much shorter than a threshold.

DOM trees and cascading style sheets have been used in a more sophisticated extractor [13] that relied on displaying the DOM tree in a web browser. The visualized DOM tree was used for a bottom-up, expanding box search that attempted to identify genuine tables from the visualized DOM tree. Another HTML focused table extraction sought to construct a search engine to support a universal meta-data format [18]. The lack of a universal meta-data format, untagged documents, and limitations with web ranking schemes are cited as examples of the challenges of creating a search engine for tables on the web and in digital libraries.

Finally, work in table extraction from PDF documents is limited. The first work appears in 2007 [5], which identified the primary challenge of PDF table extraction as being the lack of table identification within the PostScript. The project claims that the tagging of tables is seldom used in their PDF data. The extraction technique relies on font information and position using an increasing bounding box. Tables are grouped into four categories, ruled, unruled, horizontally and vertically ruled. In order to support these four categories, the algorithm introduces limitations. One key limitation is that only 25% of the table cells may span multiple columns. The authors mention that the imperfect solution is the result of possessing no domain specific information. The limitation is interesting because it directly applies to the bit field diagram, which may contain more than 25% of cells that span more than one column. Thus, the limitation of this general purpose technique possibly eliminates the entire domain of this project, microcontroller register documentation.

The Hardware Access Interface Language (HAIL) [1] is an excellent example of past work in automated tool generation for embedded systems. HAIL is a high-level, domain specific language that assists with authoring hardware device drivers. The specification for the language is divided into four categories. Register map descriptions are similar in function to the FieldWarehouse format. Second, the address space description is an abstraction that separates logical addressing from physical addressing in an execution environment and allows for the sharing and reuse of memory maps between devices. Third, device instantiation is a sequence oriented code that describes the tasks required to initialize a hardware device. Finally, the invariant specification accepts bit field access restrictions, such as read-only, to produce C++ code that throws exceptions when an access restriction is violated at run time. Together, the four categories allow the user to create a HAIL specification that the HAIL compiler accepts to produce a set of source files for the development of hardware device drivers.

Another effort relates to automating the creation of compilers for embedded systems [18]. The project accomplishes automated compiler back end generation based on a human generated architecture description. The architecture description language is a concise specification for embedded systems development tools. The specification is divided into two parts, the instruction set and all other target information, including pipeline stages, cache, and register information.

Future Work

Two improvements to FieldExtract would increase the accuracy of output with the target documents. Support for multiple bit field tables within one document register entry would allow processing of multiple registers, which are currently skipped. Secondly, FieldExtract's input could be augmented using the pdftotext representation of the bit field diagrams, providing bit field reset values and read/write restrictions. Additionally, FieldExtract could gather information from outside of the field diagrams and field description tables. Although tables are the basic construct for displaying register information, extractable information lies outside of tables as well. For example, a register's document section typically contains a narrative that describes the operation of the register. This text could be extracted and placed in the FieldWarehouse file's REG_DESCRIPTION field. Additional improvements to FieldExtract include easing the creation of new extraction algorithms for additional target architectures. Support for regular expressions and a convenient framework for per-document CSS parameters would be helpful to expand FieldExtract to many architectures.

FieldToHeader could be expanded to support multiple styles of header files, such as get/set C macros, as well as multiple languages like C++ and Java using classes. FieldWarehouse can be expanded within the specification, but a more radical evolution is possible. Given the past work in register information representation, FieldWarehouse could evolve into an application that converts between multiple register file formats.

This project is the beginning of a larger effort to partially automate the conversion of microprocessor documentation to software development tool for embedded systems.

Past work has shown the need for a well-defined, easily consumable representation of register bit fields and other microprocessor information. Once the information is in a standard format, the possibilities for automated tool creation are many. Figure 14 depicts several possibilities for project expansion.

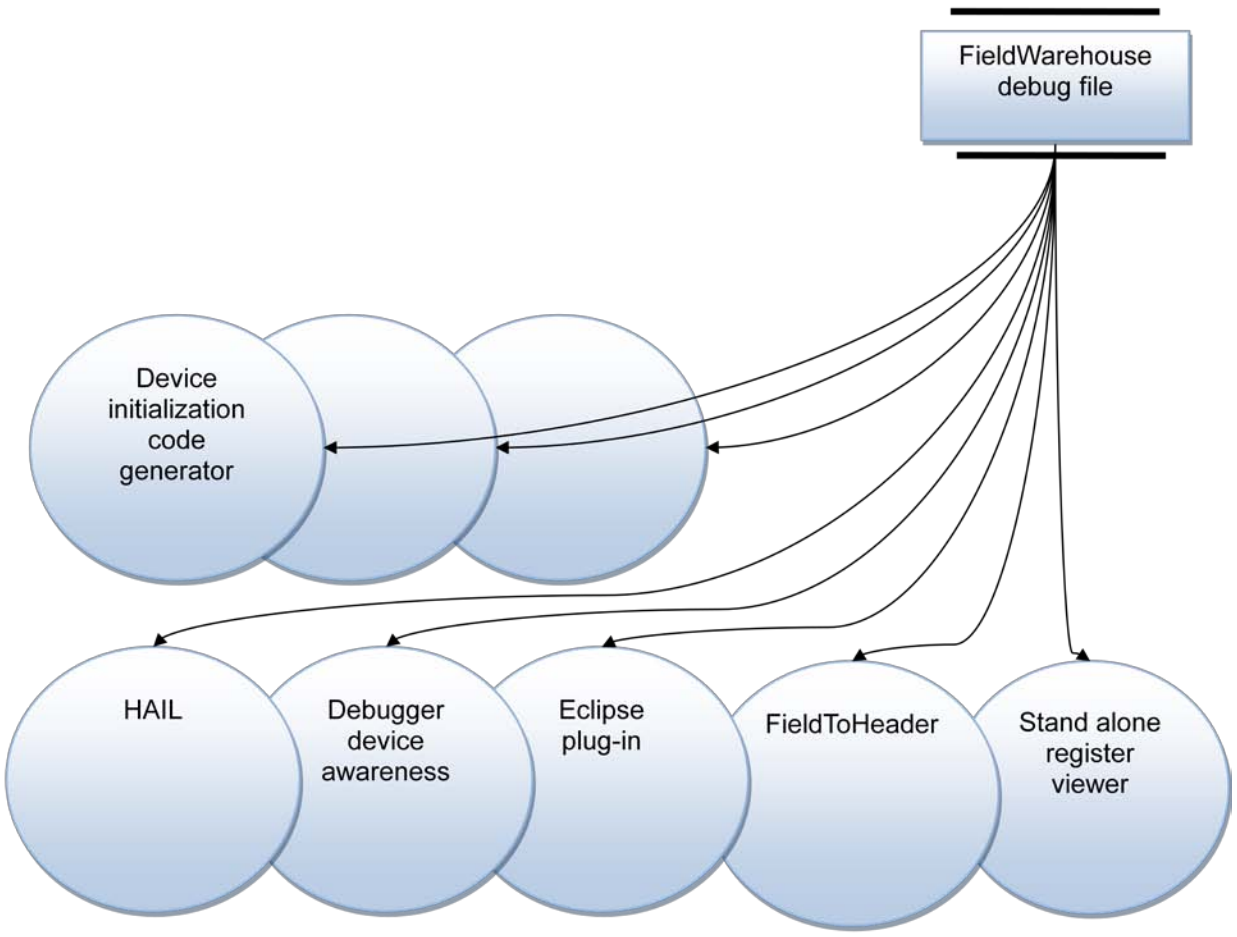

FieldWarehouse debug file
Device initialization code generator
HAIL
Debugger device awareness
Eclipse plug-in
FieldToHeader
Stand alone register viewer

Figure 14, Project Expansion

Conclusion

Within the target document set, this project achieves its goal of bringing meaningful automation to the extraction of bit field information from microcontroller user manuals. Using a bottom-up, domain specific approach to table extraction, FieldExtract's

F-measure of 88.4% compares favorably with past work in table extraction. The project successfully produced header files for every CFV1 target document with an acceptable level of accuracy. This effort ties into past work in automated tool creation for embedded systems, such as the HAIL project. Future work is possible by partially automating the creation of a variety of embedded systems software tools like simulators, compilers, and debuggers.

6. Appendix: FieldWarehouse Specification

FieldWarehouse Bit Field Format, Version 0.1, April 2011

Summary

The FieldWarehouse file format is an expandable, architecture independent representation of register and bit field information. FieldWarehouse is ASCII, line delimited, and designed for ease of human reading and modification. FieldWarehouse is a simple key/typed value scheme grouped into sections. The intention of the format is to provide a simple, well-defined, easily consumable file to present bit field information to automated tools generators.

Sections

A section begins with an exclamation point and one of the section indicators below. The following sections apply to this version.

!NOTE_BEGIN
!NOTE_END
A note section begins and ends as above. It is the only section with matched indicators. The note applies to the closest previous section, such as a register or bit field. A note section may not appear first in the file.

!FIELDWAREHOUSE_HEADER_BEGIN
This must appear on the first line of the file. The header section continues until the next section.

!MODULE_BEGIN
!REGISTER_BEGIN
!BITFIELD_BEGIN
These sections define a module, register, or bit field. Each applies to the closest previous section. The section continues until the next section. A bit field must not appear before the first register section, and a register must not appear before the first module section.

!ADDRESS_BEGIN
!EXTENDED_INFO_BEGIN
These sections are not defined and are included for future use.

Keys

Keys appear in sections and apply to the nearest previous section. Keys begin with a pound sign and appear immediately before the key value(s). Unless noted, keys possess one value.

Values

Values appear in sections and apply to the nearest previous key. The value(s) appear(s) immediately after the key. Unless noted, keys possess one value.

Value Types

The FieldWarehouse types dictate the value's format.

TYPE_STRING
The value is any series of ASCII characters that extends to the newline character.

TYPE_U_INT
TYPE_S_INT
The value is an unsigned or signed integer. The value may be a base ten number with no prefix, a hexadecimal number with a **0x** prefix, or a binary number with a **b** prefix.

TYPE_ENUM

    val1
    val2
    ...
    val*n*

The value is item from the set {val1, val2, ..., val*n*}. The enumeration's set is defined in the key's definition in this document. The value must be a single entry from the enumeration set. Multiple values are not permitted unless noted.

TYPE_BOOL

    TRUE
    FALSE

The value is an enumeration of true and false. The value must be a single entry from the enumeration set.

TYPE_NONE
The value is not defined.

TYPE_EXTENDED
This type is not defined and included for future use.

Predefined Values

The following values are predefined and may be used only where indicated.

VALUE_NULL
The value does not exist.

ENDIANNESS_BIG
ENDIANNESS_LITTLE
ENDIANNESS_BI_DEFAULT_BIG
ENDIANNESS_BI_DEFAULT_LITTLE
The value defines the processor endianness.

Section FIELDWAREHOUSE_HEADER

This section must appear first and begins with !FIELDWAREHOUSE_HEADER_BEGIN. There is exactly one FieldWarehouse header. The following key/value pairs apply to this section. Multiple keys or values are not permitted unless noted. A key/value pair that is not marked "required" is optional.

#FIELDWAREHOUSE_VERSION
TYPE_STRING
FieldWarehouse version; required

#PROCESSOR_NAME
TYPE_STRING
processor name; required

```
#PROCESSOR_ENDIANNESS
        TYPE_ENUM
                ENDIANNESS_BIG
                ENDIANNESS_LITTLE
                ENDIANNESS_BI_DEFAULT_BIG
                ENDIANNESS_BI_DEFAULT_LITTLE
processor endianness; optional

#PROCESSOR_MANUFACTURER
        TYPE_STRING
processor manufacturer; optional

#PROCESSOR_ARCHITECTURE
        TYPE_STRING
processor architecture; optional
```

#PROCESSOR_REF_DOC
This key possesses exactly three values:
    document id, TYPE_U_INT
    document name, TYPE_STRING
    document version, TYPE_STRING
processor reference document info; optional

#PROCESSOR_DESCRIPTION
    TYPE_STRING
processor description; optional; many keys allowed; each key possesses exactly one value

#PROCESSOR_EXTENDED
    TYPE_NONE
not defined; for future use

Section MODULE

This section begins with !MODULE_BEGIN. The following key/value pairs apply to this section. Multiple keys or values are not permitted unless noted. A key/value pair that is not marked "required" is optional.

#MODULE_ABBREV
    TYPE_STRING
module abbreviation; required

#MODULE_NAME
    TYPE_STRING
module name; optional

#MODULE_ID
    TYPE_U_INT
module id; optional

#MODULE_DESCRIPTION
    TYPE_STRING
module description; optional; many keys allowed; each key possesses exactly one value

#MODULE_EXTENDED
    TYPE_NONE
undefined; for future use

Section REGISTER

This section begins with !REGISTER_BEGIN. The following key/value pairs apply to this section. Multiple keys or values are not permitted unless noted. A key/value pair that is not marked "required" is optional.

#REG_ABBREV
    TYPE_STRING
register abbreviation; required

#REG_SIZE
    TYPE_U_INT
register size; required

#REG_NAME
    TYPE_STRING
register name; optional

#REG_DOC_ID
    TYPE_U_INT
register document ID; optional

#REG_DOC_SECTION
    TYPE_STRING
register document section; optional

#REG_ID
    TYPE_U_INT
register ID; optional

#REG_ADDRESS_STRING
    TYPE_STRING
register address string; optional

#REG_ADDRESS_U_INT
    TYPE_U_INT
register address integer; optional

#REG_ADDRESS_EXTENDED
    TYPE_NONE
undefined; for future use

#REG_MODULE_ID
    TYPE_U_INT
module ID; optional

#REG_DESCRIPTION
    TYPE_STRING
description; optional; many keys allowed; each key possesses exactly one value

#REGISTER_EXTENDED
    TYPE_NONE
undefined; for future use

Section BITFIELD

This section begins with !BITFIELD_BEGIN. The following key/value pairs apply to this section. Multiple keys or values are not permitted unless noted. A key/value pair that is not marked "required" is optional.

#BITFIELD_ABBREV
    TYPE_STRING
bit field abbreviation; required

#BITFIELD_INDEX
    TYPE_U_INT
bit field index; required

#BITFIELD_SIZE
    TYPE_U_INT
bit field size; required

#BITFIELD_NAME
    TYPE_STRING
bit field name; optional

#BITFIELD_DESCRIPTION
    TYPE_STRING
bit field description; optional; many keys allowed; each key possesses exactly one value

#REG_ID:
    TYPE_U_INT
register ID; optional

#BITFIELD_ID
    TYPE_U_INT
bit field ID; optional

#BITFIELD_READABLE
    TYPE_BOOL
bit field readable; optional

#BITFIELD_WRITABLE
  TYPE_BOOL
bit field writable; optional

#BITFIELD_RESET_VAL
  TYPE_U_INT
bit field reset value; optional

#BITFIELD_VALUE
This key possesses exactly two values
  bit field value, TYPE_U_INT
  bit field description, TYPE_STRING
possible bit field values; optional; many keys allowed; each posses two values

#BITFIELD_EXTENDED
  TYPE_NONE
undefined; for future use

Section NOTE

This section is a free form section that begins with !NOTE_BEGIN and ends with !NOTE_END. The section contains any number of line delimited strings. No keys may be used within the section. The contents of the note apply to the nearest previous section.

Sections ADDRESS and EXTENDED_INFO

These sections begin with !ADDRESS_BEGIN and !EXTENDED_INFO_BEGIN respectively. These sections are undefined. These sections are included for future use.

7. References

[1] J. Sun et al, “HAIL: a language for easy and correct device access,” in *Proceedings of the 5th ACM International Conference on Embedded Software*. New York, NY: ACM, 2005. [Online]. Available: http://doi.acm.org/10.1145/1086228.1086230. [Accessed April 1, 2011].

[2] Freescale Semiconductor, *MPC5553/54 Microcontroller Reference Manual, Rev. 4.0*. Chandler, AZ: Freescale Semiconductor, April 2007.

[3] Freescale Semiconductor, *MCF51QE128 Reference Manual, Rev. 3*. Chandler, AZ: Freescale Semiconductor, Sept. 2007.

[4] R. Zanibbi et al, “A Survey of Table Recognition Models, Observations, Transformations, and Inferences,” in *International Journal on Document Analysis and Recognition*, vol. 7, 2004, pp. 1-16. [Online]. Available: http://dx.doi.org/10.1007/s10032-004-0120-9. [Accessed April 1, 2011].

[5] T. Hassan and R. Baumgartner, “Table Recognition and Understanding from PDF Files,” in *Proceedings of the Ninth International Conference on Document Analysis and Recognition*, Vol. 2. Washington, DC: IEEE Computer Society, 2007, pp. 1143-1147. [Online]. Available: http://portal.acm.org/citation.cfm?id=1304833. [Accessed April 1, 2011].

[6] V1 MCU. [Website]. Available: http://www.freescale.com/webapp/sps/site/taxonomy.jsp?tid=mcHp&code=68KCFV1. [Accessed April 1, 2011].

[7] Texas Instruments, *TMS320x281x Serial Peripheral Interface Reference Guide, Rev. February 2009*. Dallas, TX: Texas Instruments, February 2009. [Online]. Available: http://focus.ti.com/lit/ug/spru059e/spru059e.pdf. [Accessed April 1, 2011].

[8] Y. Wang and J. Hu, “A machine learning based approach for table detection on the web,” in *Proceedings of the 11th international conference on World Wide Web.* New York, NY: ACM, 2002, pp. 242-250. [Online]. Available: http://doi.acm.org/10.1145/511446.511478. [Accessed April 1, 2011].

[9] Adobe Systems, Inc., *Acrobat X Pro*. [Software]. Available: http://www.adobe.com/products/acrobatpro.html. [Accessed February 14, 2011].

[10] W3C Document Object Model (DOM). [Website]. Available: http://www.w3.org/DOM/. [Accessed April 1, 2011].

[11] *Xpdf* including *pdftotext.* [Software]. Available: http://www.foolabs.com/xpdf/. [Accessed February 14, 2011].

[12] *libxml2*. [Software]. Available: http://xmlsoft.org/index.html. [Accessed February 14, 2011].

[13] W. Gatterbauer and P. Bohunsky, " Table extraction using spatial reasoning on the CSS2 visual box model," in *Proceedings of the 21st National Conference on Artificial Intelligence*, v. 2. AAAI Press, 2006, pp. 1313-1318. [Online]. Available: http://www.aaai.org/Papers/AAAI/2006/AAAI06-206.pdf. [Accessed April 1, 2011].

[14] S. Douglas et al, "Using Natural Language Processing for Identifying and Interpreting Tables in Plain Text," in *Proceedings of Fourth Annual Symposium on Document Analysis and Information Retrieval*, 1995, pp. 535-545. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.12.2053&rep=rep1&type=pdf. [Accessed April 1, 2011].

[15] H. Chen et al, "Mining tables from large scale HTML texts," in *Proceedings of the 18th Conference on Computational Linguistics*, v. 1. Stroudsburg, PA: Association for Computational Linguistics, 2000, pp. 166-172. [Online]. Available: http://dx.doi.org/10.3115/990820.990845. [Accessed April 1, 2011].

[16] Y. Wang and J. Hu, "A machine learning based approach for table detection on the web," in *Proceedings of the 11th international conference on World Wide Web.* New York, NY: ACM, 2002, pp. 242-250. [Online]. Available: http://doi.acm.org/10.1145/511446.511478. [Accessed April 1, 2011].

[17] Y. Liu et al, "Identifying table boundaries in digital documents via sparse line detection," in *Proceeding of the 17th ACM Conference on Information and Knowledge Management*. New York, NY: ACM, 2008, pp. 1311-1320. [Online]. Available: http://doi.acm.org/10.1145/1458082.1458255. [Accessed April 1, 2011].

[18] Y. Liuet al, "TableSeer: Automatic Table Metadata Extraction and Searching in Digital Libraries," in *Proceedings of the 7th ACM/IEEE-CS joint conference on Digital Libraries*. New York, NY: ACM, 2007, pp. 91-100. [Online]. Available: http://doi.acm.org/10.1145/1255175.1255193. [Accessed April 1, 2011].

[19] S. Farfeleder et al, "Effective Compiler Generation by Architecture Description," in *Proceedings of the 2006 ACM SIGPLAN/SIGBED Conference on Language, Compilers, and Tool Support for Embedded Systems.* New York, NY: ACM, 2007, pp. 145-152. [Online]. Available: http://doi.acm.org/10.1145/1134650.1134671. [Accessed April 1, 2011].