Z4205

ZPORT – An I/O Port Language for the Zilog Z8 Microcontroller

Abstract
A small language called ZPORT has been written to translate General I/O port requests for the Zilog Z8 microcontroller into C code. A formal definition of the language is provided here. This paper shows how to use the Free Software Foundation compiler tools flex and bison (also known as Lex and Yacc) to build the compiler from the language definition. The Z8! Encore evaluation board and ZDS II (Developer Studio) were used to help develop this language.

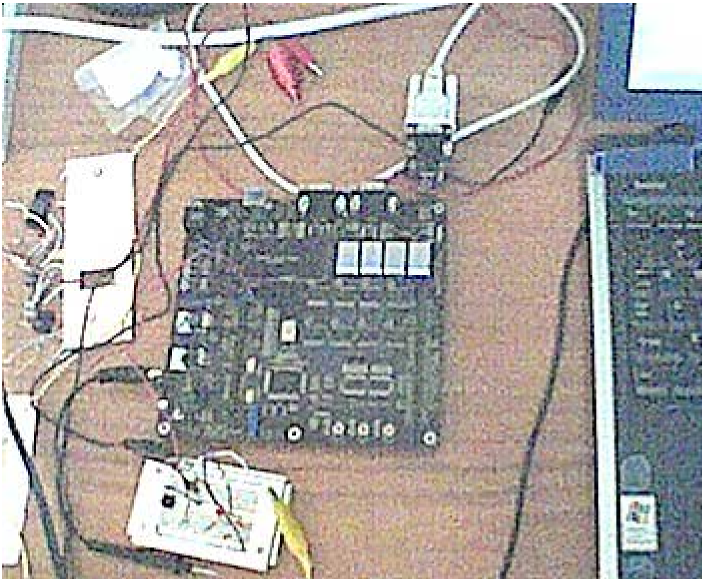

Figure 6. The Z8 Encore system used to develop ZPORT.

Anatomy of the lexical analysis file, port.l

```
%option noyywrap

%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
%}

line            [A-Ha-h][0-7]
lparen          [(]
rparen          [)]
lbrace          [{]
rbrace          [}]
comma           [,]
bang            [!]
or              "||"
```

We don't want to start using another input file when we're done with 'port.def' so we'll say 'noyywrap'.

Any include files used by the user code section below should be stated here.

A port line is defined by the port letter ('a' through 'h') and a number (0 through 7). A valid port line is 'a3' and 'A3' (the same line). An invalid port line is 'i3' or 'a8'.

The token name.

A left brace token is just the '{' symbol.

An 'or' token is 2 vertical bars next to each other.

Separates the definitions section (above) and the rules section (below).

```
and                  "&&"

%%
{line}        {yylval.string = strdup(yytext); return LINE;}
outputs       {return OUTPUTS;}
inputs        {return INPUTS;}
{lbrace}      {return LBRACE;}
{rbrace}      {return RBRACE;}
{lparen}      {return LPAREN;}
{rparen}      {return RPAREN;}
{comma}       {return COMMA;}
{bang}        {yylval.string = strdup(yytext); return BANG;}
{or}          {return OR;}
{and}         {return AND;}

%%
/*
extern int yylex();

int main( argc, argv )
int argc;
char *argv[];
     {
      yyin = fopen( "\\dev\\parser\\port1\\port.def", "r" );
     return yylex();
     }
*/
```

This refers to the definition of 'line' in the previous section.

The lexical analyzer returns terminal symbols to the syntactical analyzer. As the name implies, a terminal symbol is the end of the line where the syntactical analyzer needs to either shift (push on the syntax stack) or reduce (use a grammar rule to replace terminal and non-terminal characters with a non-terminal) the syntax.

Separates the rules section (above) and the user defined code section (below).

When you use this file as input to 'flex.exe' it produces the file 'lex.yy.c' which contains the function 'yylex()' but not 'main()' – you need to supply that.

Change the path to match the location of your input file.

Remove this and the matching comment above to test the lexical portion of your language. Normally, the syntactic analyzer (produced by 'bison.exe') will use this as input and you can just comment out this portion of the code.

Anatomy of the syntactic analysis file, port.y

```
%{
#include <stdio.h>
#include <malloc.h>
#include <string.h>
#include "compiler.h"

void yyerror (char* s);
int  yylex(void);
void pushInputLine(char* line);
void pushOutputLine(char* line);
void printBoilerplate(void);

%}

%union {
     int number;
```

The C declarations used by the user code section at the end of this file are put here.

The syntactic types of terminal and non-terminals can be one of these types: a number, string or pNode. If you omit this union then every terminal and non-terminal is considered a number. We define pNode in 'compiler.h' as follows:

```
typedef struct thisNode {
        operandType operand;   // &&, ||
        char   idValue[4];     // Only used when operand = enumId
        struct thisNode* pLeft;
        struct thisNode* pRight;
} node;
```

```
    char* string;
    node* pNode;
}

%token INPUTS OUTPUTS LBRACE RBRACE COMMA BANG LPAREN RPAREN
%token <string> LINE BANG
%token <int>   AND OR
%type  <string> line_level
%type  <string> line_sequence line_level_sequence
%type  <string> pattern_action
%type  <pNode>  expr term action

%start program
```

This tells bison that every program in this language consists of the non-terminal syntactic value ‘program’. Note that ‘program’ is defined in the next section for grammar rules.

```
%% /* Grammar rules and actions */
```

The grammar rules for the language you’re writing start in this section. This is arguably the most important part of the entire process. A badly written grammar will cause endless problems.

```
program:        input output pattern_action_list
;
```

This production rule says that a ‘program’ is composed of ‘input’ followed by ‘ouput’ and then by a ‘pattern_action_list’. These are non-terminal symbols (lower case by convention) so we need to look to further rules to see what an ‘input’ etc. is really made of.

```
input:  INPUTS LBRACE line_sequence RBRACE
;

output: OUTPUTS LBRACE line_level_sequence RBRACE        {printf("\nwhile (TRUE) {");}
;
```

The code within the braces is executed when this production rule is reached. Without any code a default rule {$$ = $1;} is executed. This default rule says that the symbol on the left-hand side of the colon is set to the first symbol on the right-hand side.

```
line_sequence:          LINE line_sequence_remainder        {pushInputLine($1);}
;

line_sequence_remainder:    /* empty */
                            |
                            COMMA line_sequence
;

line_level_sequence:  line_level line_level_sequence_remainder    {pushOutputLine($1);}
;

line_level_sequence_remainder:      /* empty */
```

A ‘line_level_sequence_remainder’ can be empty …

… or it can be a ‘,’ followed by a ‘line_level_sequence’.

```
;
```

A production rule composed of terminal symbols from the lexical analyzer. Note that the second rule concatenates the ‘!’ with the port line so you can have lines like ‘!a3’ as well as ‘h2’.

```
line_level:     LINE
                |
                BANG LINE      {$$ = $1; $$ = strcat($$, $2);}
;

pattern_action_list: /* empty */
                |
                pattern_action_list pattern_action
;
```

Notice how a ‘pattern_action_list’ is defined in terms of itself. This is a recursive definition and in particular, it’s left-recursive since ‘pattern_action_list’ is on the left-hand side. This uses less stack space and is the recommended style in ‘bison’.

```
pattern_action: expr LBRACE action RBRACE  {doPatternAction($1, $3);}
;
```

When we reach this production rule we’ve got a complete ‘pattern {action}’ sequence and call ‘doPatternAction()’ to output the expression and it’s action.

```
action: action COMMA line_level      {$$ = addNodeOperatorAction($1, $3);}
    |
    line_level        {$$ = addNodeActionId($1);}
```

```
;

expr: expr AND term   {$$ = addNodeOperator(AND, $1, $3);}
          |
          expr OR term {$$ = addNodeOperator(OR, $1, $3);}
          |
          term
;

term: LPAREN expr RPAREN     {$$ = $2;}
        |
        line_level        {$$ = addNodeId($1);}
;

%% /* Additional C code */

#include "lex.yy.c"

void yyerror (char* s)
{
        printf (" %s\n", s);
}

main ()
{
        /* Initialize IO boilerplate code. */
        FILE* stream = fopen("\\dev\\parser\\port1\\boilerplate.c", "r" );
        if (stream == NULL) {
                printf("Error opening boilerplate code.\n");
        } else {
                #define BUF_LEN (256)
                char buffer[BUF_LEN];
                while (fgets(buffer, BUF_LEN, stream) != NULL) {
                        printf("%s", buffer);
                }
                fclose(stream);
        }

        // To turn on debugging, make sure the next line is uncommented and
        //  turn on the -t (also use -v -l) options in bison.exe.
        // yydebug = 1;
        yyin = fopen("\\dev\\parser\\port1\\port.def", "r" );
        yyparse ();
        // The closing braces for 'main()' and 'while (TRUE)'.
        printf("\n} /* while */");
        printf("\n} /* main */\n");
}
```

All the code below this %% marker is supplied by you, the language author.

This is the output file generated by 'flex.exe' run on the 'port.l' input file. Rather than copy the entire file here we just use this C mechanism to include the file for compilation.

Along with 'main()' you need to supply 'yyerror()'. This function is called by the compiler if there is an error parsing the input language.

This file contains all the Zilog specific code to setup I/O lines and then read or set the values of those lines.

User defined code omitted here for brevity. See the source code for the full listing.

How all the parts fit together to build a compiler.