CASE WESTERN RESERVE UNIVERSITY
Case School of Engineering
Department of Electrical Engineering and Computer Science
EECS 337 Systems Programming (Compiler Design)
Fall 2013
Project 140 points + 10 extra credit
Due: December 5, 2013 @ 2:30 PM

Introduction

A compiler and linker translate a high level programming language into an executable image for a target processor (text to binary). Another operation of a compiler is to translate one programming language into another programming language (text to text). This is one type of a cross compiler.

For this project you design and implement an ANSI C compiler and target a PIC 16F1827 processor. You input ANSI C source and output PIC 16F1827 assembler code. You take the compiler from Assignment 3 and implement the declarations, functions and update the code generator. You can update the code generator provided or you can use your code generator from Assignment 12. The PIC assembler code shall compile and run under MPLAB. The execution of the PIC instructions shall correctly implement the C source programs. For extra credit you update the code generator to handle the mustang.c file and target a PIC16F1827 development board. The mustang.c program implements the right side of the turning signal for the Ford Mustang.

Requirements

The name of the compiler shall be ansi_c and shall run as a console application under Linux or Unix. It shall input the three C source programs (../hw0[1|2]/Code*.c[pp]) and output a correct PIC assembler program (Code*.asm) file. You shall run the project_test.sh script file (./project_test.sh &> project_test.txt) and the output file shall be printed out as part of the project report.

Each students shall design and implement a version of the C to PIC cross compiler. At the end of the project, each student shall compress the project code (tar) and submit the file to blackboard. Please follow the submission instructions to make grading the assignment easy on the professor. Answer the questions at the end of the project. The student shall turn in the project document on the last day of class. When you print the source code, please print multiple pages on each sheet.

- Cover page
- Introduction
- Answer questions
- project_test.txt
- Code*.asm
- [codegen.c] // If you wrote your own version, please include the source

Review

In the previous assignment the calculator program translated the input source code into three address codes and then into PIC assembler code. The scanner reads and converts the input into tokens and attributes. The scanner installed the identifiers and constants into the symbol table and returned the index value. The parser uses the index to find the symbol table entry. The parser updates or uses the symbol table depending on a declaration or statement production. In a declaration the type and identifier are inserted into the symbol table to create a variable. In an assignment statement an identifier can be on both sides (left = right) of the production.

The output of the parser was a QUAD with an operator and up to three operands. The linked list was then passed to the code generator. The code generator translated the operator and operands into multiple PIC assembler instructions (TUPLE). The final instructions were optimized and generated (prefix, code, postfix) using an absolute address PIC memory model.

The project target machine is the PIC 16F1827 architecture. Therefore an optimized version of a compiler can be designed. For example the immediate values for the PIC are char or integer and can range from 0 to 255. It is then possible to put static checking in the scanner to audit the source code values before passing the values to the parser. The scanner packages the input values into dynamic memory and returns the values to the parser. It leaves the parser the job of installing the identifiers into the symbol table. A C program also has levels and functions. The level information must be part of the symbol table and used in the code generation. The PIC does not allow two symbols with the same names so the code generator must handle this. The output could be in physical addresses only (0x20 to 0x7f) and the symbols sections would not need to be generated (symbol EQU address). The address only function has been included in the source code and is turned on and off with the address flag on the command line.

```
Usage: ansi_c [[+|-]echo] [[+|-]debug] [[+|-]yydebug] [[+|-]symbol]
       [[+|-]address] [+test] [filename] [...]
```

The echo flag repeats the input characters to the console window. The debug flag prints the TUPLE linked list at the top of the parser and again after post processing. The yydebug flag puts the yacc parser into verbose mode. The symbol flag prints the symbol table and the free symbol table. The address flag generates the output using only physical addresses. The test flag prints the example code generator functions. Any other input is treated as an input file.

Instructions
From a console window, make a directory on your computer in your EECS337 directory under your Case ID and call it project.
mkdir ~/EECS337/caseid/project/ ; where caseid is YOUR Case ID, in lower case

Change directory to the project directory.
cd ~/EECS337/caseid/project/

Download a copy of the file: project_caseid.tar to the project directory from http://blackboard.case.edu/ in the EECS337 homework assignment area. To untar the tar file type the command:
tar xvf project_caseid.tar

The following files will be created in the current working directory.
Makefile
project_test.sh
main.c
scan.l
gram.y
yystype.h
tokens.h
audit.c
mustang.c
test1.c
test2.c
test3.c
test3.txt
tuple.c
symbols.c
codegen.c

The original C scanner (scan.l) and grammar files (gram.y) from assignment 03 have been updated to include a main program (main.c), an include file (yystype.h) a Makefile and extra source files (audit.c symbols.c tuples.c codegen.c tokens.h). **The scan.l and gram.y files are the only files you need to edit to complete the first part of the project. Then update or rewrite the codegen.c file to complete the project.** A number of other files have been included to perform the scanner, parser, symbol table and code generator functions. You need to update the scanner file to return tokens and attributes to the parser and insert the parser functions into the yacc file to use the symbol table and generate the PIC assembler code.

The ansi_c compiler from Assignment 03 has a scanner (scan.l) and parser (gram.y) to handle the C programming language. The scanner has no actions for the regular expressions and the parser has no actions. You update the scanner and the parser using the files included with the project. To build and test the initial ansi_c compiler type the commands: **make clean** and **make**

./ansi_c +debug +symbol +yydebug test1.c

Implementation Details
Implement the source code to perform the cross compiler operations. Use the files provided or write your own. You can reuse the source code from any of the assignments. An outline of the step to complete the project is shown below.

- Update the scanner to include passing attributes to the parser.
- Update the parser to implement declarations for the symbol table (create and find).
- Update the parser to translate functions into PIC assembler code.
- Update or rewrite the PIC code generator to support all the PIC16F1827 instructions.
- [Extra] Update the code generator to support the embedded development board.

Part 1: Scanner Update
Open the scan.l file and find the regular expression for IDENTIFIER that returns the check_type() function. Change the action to the action shown below. The scanner functions are provided in the audit.c file.

```
{ count(); yylval.tuple = identifier( yytext, yyleng); return(check_type()); }
```

Find the hexadecimal regular expression that returns a CONSTANT. Change the hexadecimal action to the action shown below.

```
{ count(); yylval.tuple = constant_hex( yytext, yyleng); return(CONSTANT); }
```

Repeat this process for the **octal, decimal, char, float** and **string_literal** actions replacing the function word "hex" with each of the names. Then change **all** the other regular expressions that return a token to assign that token to the attribute variable (yylval.token). (first and last regular expressions are shown below)

```
"auto"          { count(); yylval.token = AUTO; return(AUTO); }
…
"?"             { count(); yylval.token = '?'; return('?'); }
```

Save the scan.l file.

Open the gram.y file. Find the declaration_specifiers production and the same function name in the audit.c file. In the audit.c file in the comment above the function, copy and paste the parser functions to each of the four productions in the gram.y file and be sure to include the action inside curly brackets inside the production. Save the gram.y file and close the audit.c file. The audit (static checking) allows INT, CHAR and VOID types.

Part 2: Parser Declarations Update
For the symbol table each set of curly brackets (block or function body) increments the level count. At the end of each block the symbol table is saved onto the free symbol table list and the level is decremented. The free symbol table information is needed by the code generator for generating the symbol table. In this case when the level is 0 then it is a global variable. Above level 0 is a block statement or function body and the symbol can be re-declared. Therefore the code generator uses the level information in the symbol name generation. The symbol name is appended with the underscore character '_' and the level number during code generation. For example the code fragment below has one global variable and three stack variables:

```
int i = 0;
main()
{
  int i = 1;
  {
    int i = 2;
    {
     int i = 3;
    }
  }
}
```

The identifier is looked up in the symbol table to find the physical address and level information. This is printed for each symbol. If the symbol is greater than level 0 the code generator generates the level number appended to the symbol separated by the underscore character '_'. Else if the symbol or label is only one character long then the underscore '_' symbol is also appended to the symbol name.

The symbols are used in assignment operations (left = right) and converted into “MOVWF symbol” or “MOVF symbol, W” PIC instructions. Constants (right only) are converted into “MOVLW value” PIC instructions. The code generator output for the symbol table and code fragment for the example is:

```
;symbol table:
i_ EQU 0x20
;symbol table free:
i_1 EQU 0x21
i_2 EQU 0x22
i_3 EQU 0x23

mloop:
; here begins the main program
     movlw 0x00
     movwf i_
     movf  PORTA,w
     call  main
     goto  mloop
     return
main:
     movlw 0x01
     movwf i_1
     movlw 0x02
     movwf i_2
     movlw 0x03
     movwf i_3
     return    ; if main does not have a return
```

Open the gram.y file and change the `%start file` to `code` and add the production below.

```
code : file
     {
#ifdef    YYDEBUG
          if( IS_FLAGS_DEBUG( data.flags))
          {
               printf( "Debug: yacc tuples\n");
               print_tuple_list( $1.tuple);
          }
#endif
          code_generator_pic16f1827( $1.tuple);
     }
     ;
```

In the gram.y file find the compound_statement production and replace the compound_statement production with the code below. Also open the symbols.c file. Find the same function names in the symbols.c file and notice in the comments above the functions are the function calls being used in the code pasted below.

```
left_bracket
     : '{'
     {
          symbol_left_bracket();
     }
     ;

right_bracket
     : '}'
     {
          symbol_right_bracket();
     }
     ;

compound_statement   : left_bracket right_bracket
     {
          $$.tuple = 0;
     }
     | left_bracket statement_list right_bracket
     {
          $$.tuple = $2.tuple;
     }
     | left_bracket declaration_list right_bracket
     {
          $$.tuple = $2.tuple;
     }
     | left_bracket declaration_list statement_list right_bracket
     {
          $$.tuple = tuple_tail_to_head( $2.tuple, $3.tuple);
     }
     ;
```

In gram.y find the declaration production. Change the two productions to the code below.

```
declaration
     : declaration_specifiers ';'
     {
          $$.tuple = 0;
     }
     | declaration_specifiers init_declarator_list ';'
     {
          $$.tuple = symbol_declaration( $1.token, $2.tuple);
          $$.tuple = tuple_declaration( $1.token, $$.tuple);
     }
     ;
```

In gram.y find the init_declarator_list production. Insert the action to the production with the comma.

```
$$.tuple = symbol_init_declarator_list( $1.tuple, $3.tuple);
```

In gram.y find the init_declarator production. Insert the action to the production with the equal sign.

```
$$.tuple = symbol_init_declarator( $1.tuple, $3.tuple);
```

Save the gram.y. To build and test the symbol table version type the commands:
make clean
make
./ansi_c +debug +symbol test1.c

```
; automatic code generation for PIC16F1827
; EECS337 Compiler Design
; by: caseid, date: Fall 2013
; for PIC16F1827 processor
; CPU configuration
     list      p=16f1827      ; list directive to define processor
     #include   <p16f1827.inc> ; processor specific variable definitions
     __CONFIG _CONFIG1, _FOSC_INTOSC & _WDTE_OFF & _PWRTE_OFF & _MCLRE_ON &
_CP_OFF & _CPD_OFF & _BOREN_OFF & _CLKOUTEN_ON & _IESO_OFF & _FCMEN_OFF
     __CONFIG _CONFIG2, _WRT_OFF & _PLLEN_OFF & _STVREN_OFF & _BORV_19 &
_LVP_OFF
;symbol table:
c_ EQU 0x20
;symbol table free:
; power-up or reset vector
     org     0x0000            ; processor reset vector
     pagesel init
     goto init ; skip interrupt vector space (reserved)
; interrupt vector
     org  0x04
inter:               ; on interrupt PC set to 0x04 and automatically clears GIE
     retfie          ; return from interrupt and automatically sets GIE
; beginning of program code
     org  0x08
init:
; On reset all ports are inputs.
     movlb1    ; switch to bank 1 memory
```

```
;     clrf TRISA      ; set PORTA to all outputs
      clrf TRISB      ; set PORTB to all outputs
      movlb 0    ; switch to bank 0 memory
mloop:
; here begins the main program
      movlw 0x00
      movwf c_
      goto mloop
      return     ; if main does not have a return
; only standard library function
printf:
      movwf PORTB; output w to standard output (stdout)
      return
      end        ; end of program code
symbol table:
next: 0x0 token: CHAR value: 0 address: 0x20 mask: 0x0 level: 0 length: 2
buffer: c
symbol table free:
```

To test using an initialization value in the declaration then type the command:
./ansi_c +debug +symbol test2.c

The output will be the same as test1.c except for the one statement below.

```
       movlw  0x02
```

Once you have this much working then you are ready to add the rest of the functions to the parser. Save a copy of your gram.y file and close the audit.c and symbols.c files.

Part 3: Parser Functions Update
For a function body identifier a label “function:” is generated for the function name. Any call to the function is generated as a “CALL function” PIC instruction. The end of the function must have a return (RETURN, RETLW) instruction. This static checking is handled in the first stage of the code generator (code_post_process). The code generator is already hooked into the parser when we inserted the new %start production (code).

Open the gram.y and the tuple.c files. In gram.y find the primary_expr production. Insert the matching actions into the productions and include curly brackets around the actions.

```
              $$.tuple = tuple_primary_expr_identifier( $1.tuple);
              $$.tuple = tuple_primary_expr_constant( $1.tuple);
              $$.tuple = tuple_primary_expr_string_literal( $1.tuple);
              $$ = $2;      /* C allows struct to struct copy */
```

In gram.y find the postfix_expr production. Insert the actions for the productions with the ‘(‘ ‘)’ characters.

```
      | postfix_expr '(' ')'
      {
          $$.tuple = tuple_postfix_expr( $1.tuple, 0);
      }
      | postfix_expr '(' argument_expr_list ')'
      {
          $$.tuple = tuple_postfix_expr( $1.tuple, $3.tuple);
      }
```

In gram.y find the argument_expr_list production. Insert the tail to head function to link the production linked lists for the ‘,’ character.

```
              $$.tuple = tuple_tail_to_head( $1.tuple, $3.tuple);
```

In gram.y find the additive_expr production. Insert the action for the production with the ‘+’ character.

```
              $$.tuple = tuple_additive_expr( I_ADD, $1.tuple, $3.tuple);
```

In gram.y find the assignment_expr production. Insert the action for the production with the assignment_operator.

```
              $$.tuple = tuple_assignment_expr( $1.tuple, $2.token, $3.tuple);
```

In gram.y find the declarator production. Insert the action for the production with the pointer non-terminal symbol.

```
                              $$ = $2;
```

In gram.y find the declarator2 production. Insert the first action for the '(' declarator ')' production the second action for two with parameter_*_list non-terminal symbols.

```
              $$ = $2;
              $$.tuple = tuple_tail_to_head( $1.tuple, $3.tuple);
```

In gram.y find the parameter_list production. Insert the first action for the first production and the second action for second production.

```
              $$.tuple = tuple_parameter_list( $1.tuple, 0);
              $$.tuple = tuple_parameter_list( $1.tuple, $3.tuple);
```

In gram.y find the parameter_declaration production. Insert the action for the first production.

```
              $$ = $2;
```

In gram.y find the declaration_list, statement_list, file and function_body productions. Insert the action for the two non-terminal productions.

```
              $$.tuple = tuple_tail_to_head( $1.tuple, $2.tuple);
```

In gram.y find the jump_statement production. Replace the productions for RETURN.

```
       | RETURN ';'
       {
              $$.tuple = new_tuple( I_RETURN, 0, 0, MASK_INSTR, 0, 0);
       }
       | RETURN expr ';'
       {
              $$.tuple = tuple_jump_statement( $2.tuple);
       }
```

In gram.y find the function_definition production. Insert the first action for the first productions and the second for the second production.

```
           $$.tuple = tuple_function_definition( 0, $1.tuple, $2.tuple);
           $$.tuple = tuple_function_definition( $1.token, $2.tuple, $3.tuple);
```

Save the gram.y. To build the final version type the commands: **make clean** and **make**
./ansi_c +test3.c > test3.asm

Open the test3.txt file to check results or use the diff command:
diff test3.asm test3.txt

Part 4: Code Generator Update
Update or rewrite the code generator to support all the PIC16F1827 instructions. The current version supports the PIC16F84A processor. Use your work from Assignment 12 or extract the information from the PIC documentation. Consider using the values and names from the p16f1827.inc file. For example the current code generator does not support using the operand STATUS,Z and uses the physical address instead. Update the test function to show an example of each new instruction added. Test using the command:
./ansi_c +test
Part 5: Output Generation
The compiler shall output the target source code to a file with the same input file name, except change the file extension from .c to .asm. Copy the C sample program coded from assignment 01 and 02 to your project directory. Your PIC assembler files shall run correctly under MPLAB. Then run the ansi_c compiler to create the output files.
cp ../hw01/Code_1_6_1.c .
cp ../hw01/Code_1_6_2.c .
cp ../hw02/Code_1_6_4.cpp .

./ansi_c Code_1_6_1.c > Code_1_6_1.asm
./ansi_c Code_1_6_2.c > Code_1_6_2.asm
./ansi_c Code_1_6_4.cpp > Code_1_6_4.asm

Part 6: Submit Results
When project code is completed execute the project script file "./project_test.sh". The script file shall remove the old object files, make all source programs and run the tests. To redirect the standard error (stderr) and standard output (stdout) to a file use the following command.
./project_test.sh &> project_test.txt

Submit your project source code and final output files. Create a tar file from your ~/EECS337/ directory **using the following commands** and using your case id.
make clean
rm -f *~ .*~ #* .#* *% .*%
cd ~/EECS337
tar cvf project_*caseid*.tar *caseid*/project/*

This command will create a project_caseid.tar file using the files from your caseid/project/ directory. Notice it is **important** to include the directory path as part of the tar file create. This allows for each students source code to be recreated under a directory using your caseid. This tar ball shall contain the final version of your ansi_c compiler for the EECS337 Compilers class. Submit your tar file (project_caseid.tar) to blackboard under the Project section. Submit it with the comments **project** with your **CaseID** on the title line.

Write a project report. Include a cover page, introduction, answer the project questions, print your project_test.txt file, your Code*.asm files, and your codegen.c file. Turn in the report on the last day of class.

Part 7: Extra Credit (10 points)
Update the code generator to handle the mustang.c file and target the PIC16F1827 development board. The mustang.c program implements the right side of the turning signal for the Ford Mustang. Instead of three blinking lights this example uses four to match the four LEDS on the development board. You will want to change the CONFIGURE parameters to use the 10 MHZ crystal, the analog input on RA0, the switch on RA5 and the four LEDS on RB0, RB1, RB2 and RB3. See the information in the Appendix for ordering a PIC development board [optional].

Build the ansi_c program and fix any errors using the commands:
make clean
make

Test using the extra credit test file (mustang.c) using the command below:
./ansi_c mustang.c > mustang.asm

Edit the project_test.sh file and remove the comment to test the mustang.c file. When the extra credit code is complete and working then execute the homework script file "./project_test.sh" using the command below.
./project_test.sh &> project_test.txt

Simulate your mustang.asm file using the PIC simulator or download to the development board. Print out this version of the project_test.txt file. Put your name and date on it and turn it in instead of the output file generated above.

Grading by the Professor
1) Project Report – cover page, introduction, project_test.txt, .asm, .c, answer questions
2) Source code tar file – naming convention followed, extracts correctly
3) Building the code – any compiler warnings or errors, project.txt
   a. Declarations
   b. Functions
   c. Code Generator
4) Running the code – compiler runs and generates PIC assembler, Code*.asm
   a. Declarations
   b. Functions
   c. Code Generator
5) Project questions and review of source code
6) Extra credit – mustang.asm

Your final directory structure for the ansi_c compiler should be as below and may include other source files you added to your project code:

```
EECS337/caseid/project/Code_1_6_1.asm
EECS337/caseid/project/Code_1_6_1.c
EECS337/caseid/project/Code_1_6_2.asm
EECS337/caseid/project/Code_1_6_2.c
EECS337/caseid/project/Code_1_6_4.asm
EECS337/caseid/project/Code_1_6_4.cpp
EECS337/caseid/project/Makefile
EECS337/caseid/project/ansi_c
EECS337/caseid/project/audit.c
EECS337/caseid/project/codegen.c
EECS337/caseid/project/gram.y
EECS337/caseid/project/main.c
EECS337/caseid/project/project_caseid.tar
EECS337/caseid/project/project_test.sh
EECS337/caseid/project/project_test.txt
EECS337/caseid/project/scan.l
EECS337/caseid/project/symbols.c
EECS337/caseid/project/mustang.asm        // extra credit
EECS337/caseid/project/mustang.c
EECS337/caseid/project/test1.c
EECS337/caseid/project/test2.c
EECS337/caseid/project/test3.c
EECS337/caseid/project/test3.txt
EECS337/caseid/project/tokens.h
EECS337/caseid/project/tuple.c
EECS337/caseid/project/yystype.h
```

Part 8: Project Questions

1) The book claims using the same size data structure or allocating and freeing memory in a predictable manner (LIFO, FIFO) helps minimize memory fragmentation. So, if the ansi_c compiler uses only one data structure (TUPLE) for the parser and symbol table, does the heap memory still get fragmented? Explain your answer.

2) What are two ways to generate the symbol table information for the PIC16F1827 processor?

3) What are the new PIC16F1827 instructions not found in the PIC16F84A processor?

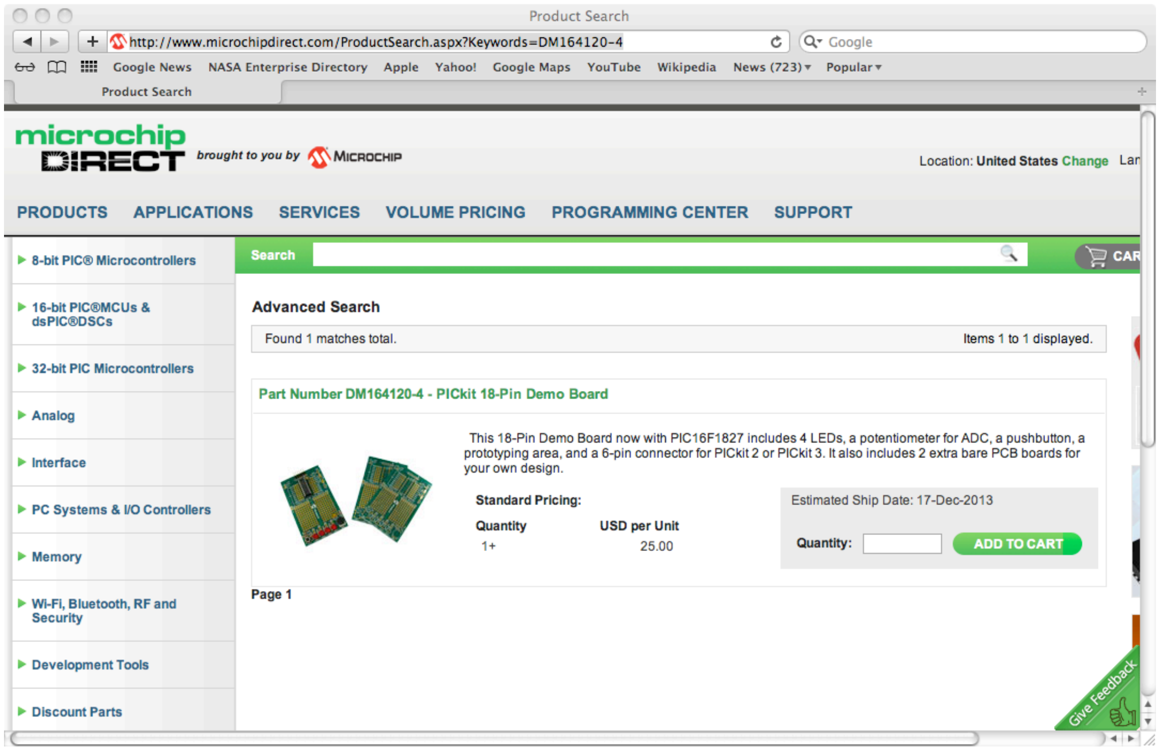
4) What are the PIC instructions to initialize and read the analog input from RA0?

5) What are the PIC configuration parameters to use the 10 MHZ crystal on the development board?

Appendix [Optional] Development Board and USB Programmer

You can order a development board ($25) and programmer ($44.95) from the Microchips on-line store.

http://www.microchipdirect.com/ProductSearch.aspx?Keywords=DM164120-4


Product Search
http://www.microchipdirect.com/ProductSearch.aspx?Keywords=DM164120-4
Google News
NASA Enterprise Directory
Apple
Yahoo!
Google Maps
YouTube
Wikipedia
News (723)
Popular
microchip DIRECT
brought to you by MICROCHIP
Location: United States Change
PRODUCTS
APPLICATIONS
SERVICES
VOLUME PRICING
PROGRAMMING CENTER
SUPPORT
8-bit PIC® Microcontrollers
16-bit PIC®MCUs & dsPIC®DSCs
32-bit PIC Microcontrollers
Analog
Interface
PC Systems & I/O Controllers
Memory
Wi-Fi, Bluetooth, RF and Security
Development Tools
Discount Parts
Search
CART
ADD TO CART
Give Feedback

Advanced Search

Found 1 matches total. Items 1 to 1 displayed.

Part Number DM164120-4 - PICkit 18-Pin Demo Board

This 18-Pin Demo Board now with PIC16F1827 includes 4 LEDs, a potentiometer for ADC, a pushbutton, a prototyping area, and a 6-pin connector for PICkit 2 or PICkit 3. It also includes 2 extra bare PCB boards for your own design.

Standard Pricing:

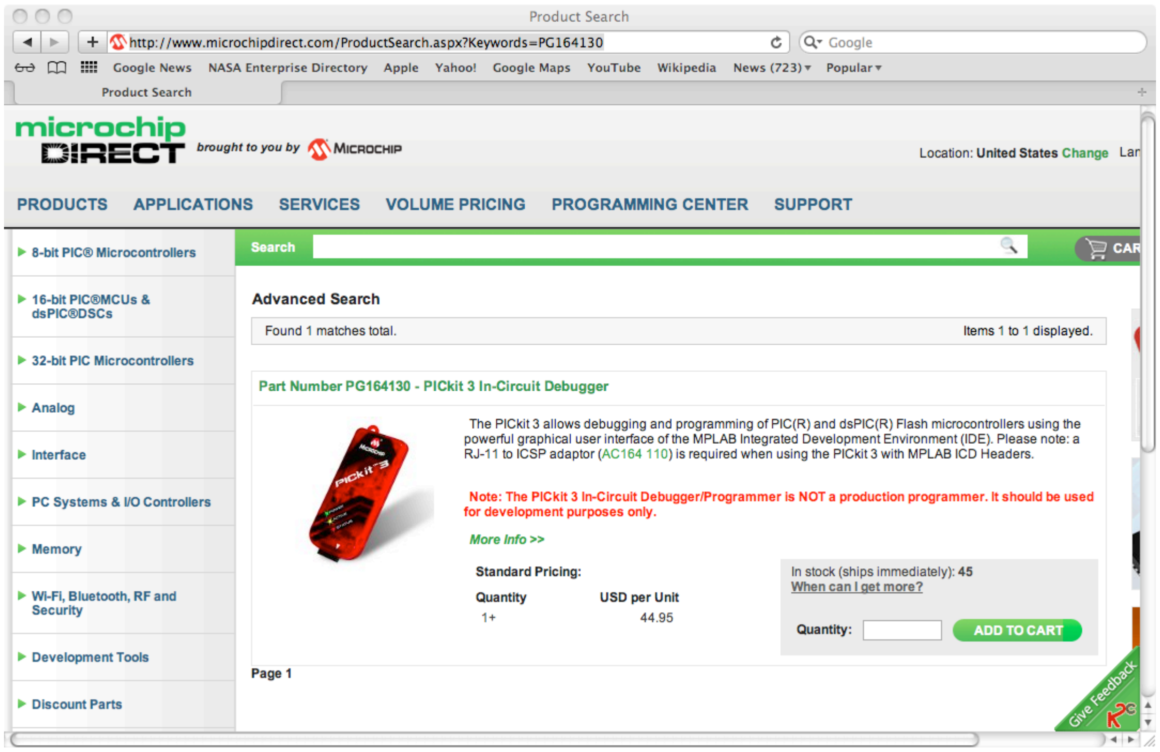
Quantity	USD per Unit
1+	25.00

Estimated Ship Date: 17-Dec-2013

Quantity:

Page 1

http://www.microchipdirect.com/ProductSearch.aspx?Keywords=PG164130


Product Search
http://www.microchipdirect.com/ProductSearch.aspx?Keywords=PG164130
Google News
NASA Enterprise Directory
Apple
Yahoo!
Google Maps
YouTube
Wikipedia
News (723)
Popular
microchip DIRECT
brought to you by MICROCHIP
Location: United States Change
PRODUCTS
APPLICATIONS
SERVICES
VOLUME PRICING
PROGRAMMING CENTER
SUPPORT
8-bit PIC® Microcontrollers
16-bit PIC®MCUs & dsPIC®DSCs
32-bit PIC Microcontrollers
Analog
Interface
PC Systems & I/O Controllers
Memory
Wi-Fi, Bluetooth, RF and Security
Development Tools
Discount Parts
Search
CART
ADD TO CART
Give Feedback

Advanced Search

Found 1 matches total. Items 1 to 1 displayed.

Part Number PG164130 - PICkit 3 In-Circuit Debugger

The PICkit 3 allows debugging and programming of PIC(R) and dsPIC(R) Flash microcontrollers using the powerful graphical user interface of the MPLAB Integrated Development Environment (IDE). Please note: a RJ-11 to ICSP adaptor (AC164 110) is required when using the PICkit 3 with MPLAB ICD Headers.

Note: The PICkit 3 In-Circuit Debugger/Programmer is NOT a production programmer. It should be used for development purposes only.

More Info >>

Standard Pricing:

Quantity	USD per Unit
1+	44.95

In stock (ships immediately): 45
When can I get more?

Quantity:

Page 1