Advanced C Programming

Advanced C programming

Agenda

- AWK
- Autotools
- Kbuild
- RSS,PSS,Maps
- UIO

Introduction

AWK is a pattern-directed scanning and processing language. An AWK program consists of a set of actions to be taken against streams of textual data. AWK extensively uses regular expressions. It is a standard feature of most Unix-like operating systems.

AWK was created at Bell Labs in the 1977. Its name is derived from the family names of its authors – Alfred Aho, Peter Weinberger, and Brian Kernighan.

There are two major implementations of AWK. The traditional Unix AWK and the newer GAWK. GAWK is the GNU Project's implementation of the AWK programming language. GAWK has several extensions to the original AWK.

The awk command programming language requires no compiling, and allows the user to use variables, numeric functions, string functions, and logical operators.

Awk is a utility that enables a programmer to write tiny but effective programs in the form of statements that define text patterns that are to be searched for in each line of a document and the action that is to be taken when a match is found within a line.

Awk is mostly used for pattern scanning and processing. It searches one or more files to see if they contain lines that matches with the specified patterns and then performs the associated actions.

Introduction

WHAT CAN WE DO WITH AWK?

1. AWK Operations:

(a) Scans a file line by line

(b) Splits each input line into fields

(c) Compares input line/fields to pattern

(d) Performs action(s) on matched lines

2. Useful For:

(a) Transform data files

(b) Produce formatted reports

3. Programming Constructs:

(a) Format output lines

(b) Arithmetic and string operations

(c) Conditionals and loops

AWK Syntax

# awk options program file

Awk can take the following options:

-F fs : To specify a file separator.

-f file: To specify a file that contains awk script.

-v var=value: To declare a variable.

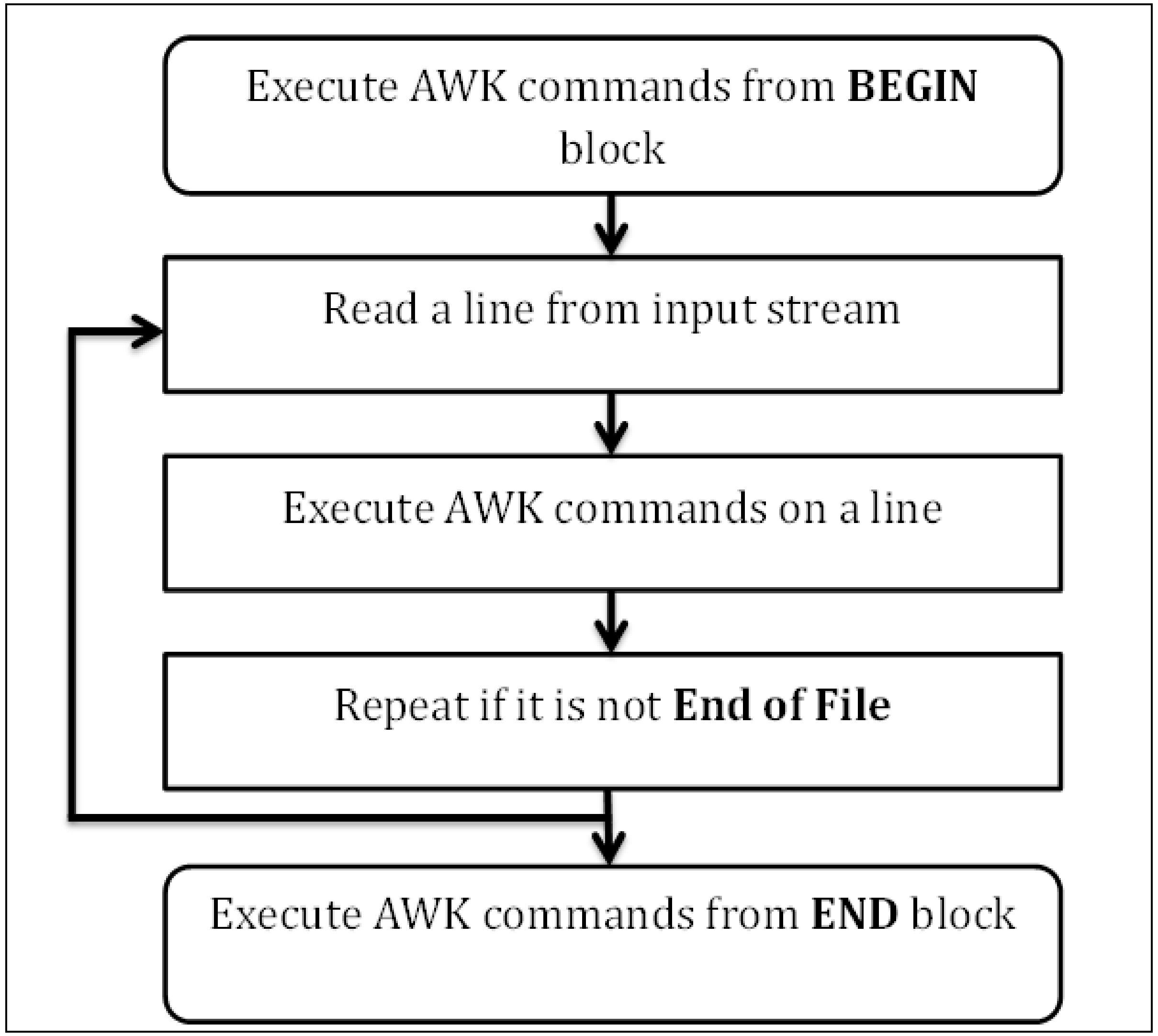
The structure of an AWK program has the following form:

pattern { action }

Example: awk '{print}' employee.txt

AWK Workflow

AWK follows a simple workflow − Read, Execute, and Repeat.


Execute AWK commands from BEGIN block
Read a line from input stream
Execute AWK commands on a line
Repeat if it is not End of File
Execute AWK commands from END block

AWK Workflow

Read

AWK reads a line from the input stream (file, pipe, or stdin) and stores it in memory.

Execute

All AWK commands are applied sequentially on the input. By default AWK execute commands on every line. We can restrict this by providing patterns.

Repeat

This process repeats until the file reaches its end.

BEGIN Block

BEGIN block

The syntax of the BEGIN block is as follows:

Syntax

BEGIN {awk-commands}

The BEGIN block gets executed at program start-up. It executes only once. This is good place to initialize variables. BEGIN is an AWK keyword and hence it must be in upper-case. This block is optional.

BODY Block

Body Block

The syntax of the body block is as follows:

Syntax

```
/pattern/ {awk-commands}
```

The body block applies AWK commands on every input line. By default, AWK executes commands on every line. We can restrict this by providing patterns. Note that there are no keywords for the Body block.

END Block

END Block

The syntax of the END block is as follows:

Syntax

END {awk-commands}

The END block executes at the end of the program. END is an AWK keyword and hence it must be in upper-case. This block is optional.

BEGIN Example

```
# awk 'BEGIN{printf "Name\tRole\tDepartment\tSalary\n"} {print}' employee.txt
```

-f option

```
# awk -f command.awk marks.txt
```

-v option

```
# awk -f command.awk marks.txt
```

-F option

Sometimes the separator in some files is not space nor tab but something else. You can specify it using –F option:

```
# awk -F: '{print $1}' /etc/passwd
```

AWK Variables

With awk, you can process text files. Awk assigns some variables for each data field found:

$0 for the whole line.

$1 for the first field.

$2 for the second field.

$n for the nth field.

The whitespace character like space or tab is the default separator between fields in awk.

# awk '{print $1}’ employee.txt

Multiple commands

To run multiple commands, separate them with a semicolon like this:

```
# echo "Hello Tom" | awk '{$2="Adam"; print $0}'
```

Builtin Variables

Awk’s built-in variables include the field variables:

NR: NR command keeps a current count of the number of input records. Remember that records are usually lines. Awk command performs the pattern/action statements once for each record in a file.

NF: NF command keeps a count of the number of fields within the current input record.

FS: FS command contains the field separator character which is used to divide fields on the input line. The default is “white space”, meaning space and tab characters. FS can be reassigned to another character (typically in BEGIN) to change the field separator.

RS: RS command stores the current record separator character. Since, by default, an input line is the input record, the default record separator character is a newline.

Builtin Variables

OFS: OFS command stores the output field separator, which separates the fields when Awk prints them. The default is a blank space. Whenever print has several parameters separated with commas, it will print the value of OFS in between each parameter.

ORS: ORS command stores the output record separator, which separates the output lines when Awk prints them. The default is a newline character. print automatically outputs the contents of ORS at the end of whatever it is given to print.

;display line number along with entire line

```
#  awk '{print NR,$0}' employee.txt
```

;display first and last string

```
# awk '{print $1,$NF}' employee.txt
```

Builtin Variables

OFS: OFS command stores the output field separator, which separates the fields when Awk prints them. The default is a blank space. Whenever print has several parameters separated with commas, it will print the value of OFS in between each parameter.

ORS: ORS command stores the output record separator, which separates the output lines when Awk prints them. The default is a newline character. print automatically outputs the contents of ORS at the end of whatever it is given to print.

;display line number along with entire line

```
#  awk '{print NR,$0}' employee.txt
```

;display first and last string

```
# awk '{print $1,$NF}' employee.txt
```

Builtin Variables

;Display lines between 3 to 6

```
# awk 'NR==3, NR==6 {print NR,$0}' employee.txt
```

Builtin Variables

ARGC

It implies the number of arguments provided at the command line.

```
# awk 'BEGIN {print "Arguments =", ARGC}' One Two Three Four
```

```
# awk 'BEGIN {
   for (i = 0; i < ARGC - 1; ++i) {
      printf "ARGV[%d] = %s\n", i, ARGV[i]
   }
}' one two three four
```

Builtin Variables

CONVFMT

It represents the conversion format for numbers. Its default value is %.6g.

```
# awk 'BEGIN { print "Conversion Format =", CONVFMT }’
```

ENVIRON

It is an associative array of environment variables.

```
# awk 'BEGIN { print ENVIRON["USER"] }'
```

Builtin Variables

FILENAME

It represents the current file name.

```
# awk 'END {print FILENAME}’ employee.txt
```

FS

It represents the (input) field separator and its default value is space. You can also change this by using -F command line option.

```
# awk 'BEGIN {print "FS = " FS}' | cat -vte
```

Builtin Variables

NF

It represents the number of fields in the current record. For instance, the following example prints only those lines that contain more than two fields.

```
# echo -e "One Two\nOne Two Three\nOne Two Three Four" | awk 'NF > 2’
```

NR

It represents the number of the current record. For instance, the following example prints the record if the current record number is less than three.

```
# echo -e "One Two\nOne Two Three\nOne Two Three Four" | awk 'NR < 3’
```

Builtin Variables

FNR

It is similar to NR, but relative to the current file. It is useful when AWK is operating on multiple files. Value of FNR resets with new file.

OFMT

It represents the output format number and its default value is %.6g.

```
# awk 'BEGIN {print "OFMT = " OFMT}’
```

OFS

It represents the output field separator and its default value is space.

```
# awk 'BEGIN {print "OFS = " OFS}' | cat -vte
```

Builtin Variables

ORS

It represents the output record separator and its default value is newline.

```
# awk 'BEGIN {print "ORS = " ORS}' | cat -vte
```

RLENGTH

It represents the length of the string matched by match function. AWK's match function searches for a given string in the input-string.

```
# awk 'BEGIN { if (match("One Two Three", "re")) { print RLENGTH } }’
```

Builtin Variables

RS

It represents (input) record separator and its default value is newline.

```
# awk 'BEGIN {print "RS = " RS}' | cat -vte
```

RSTART

It represents the first position in the string matched by match function.

```
# awk 'BEGIN { if (match("One Two Three", "Thre")) { print RSTART } }’
```

Builtin Variables

SUBSEP

It represents the separator character for array subscripts and its default value is \034.

```
# awk 'BEGIN { print "SUBSEP = " SUBSEP }' | cat -vte
```

$0

It represents the entire input record.

```
# awk '{print $0}' marks.txt
```

Builtin Variables

$n

It represents the nth field in the current record where the fields are separated by FS.

```
# awk '{print $3 "\t" $4}' marks.txt
```

AWK Operators

AWK also provides a large set of operator:

- ❑Arithmetic Operators
- ❑Increment and Decrement Operators
- ❑Assignment Operators
- ❑Relational Operators
- ❑Logical Operators
- ❑Ternary Operator
- ❑Unary Operators
- ❑Exponential Operators
- ❑String Concatenation Operator
- ❑Array Membership Operator
- ❑Regular Expression Operators

Arthematic Operators

Addition

It is represented by plus (+) symbol which adds two or more numbers.

```
# awk 'BEGIN { a = 50; b = 20; print "(a + b) = ", (a + b) }’
```

Subtraction

It is represented by minus (-) symbol which subtracts two or more numbers.

```
# awk 'BEGIN { a = 50; b = 20; print "(a - b) = ", (a - b) }'
```

Arthematic Operators

Multiplication

It is represented by asterisk (*) symbol which multiplies two or more numbers.

```
# awk 'BEGIN { a = 50; b = 20; print "(a * b) = ", (a * b) }’
```

Division

It is represented by slash (/) symbol which divides two or more numbers.

```
# awk 'BEGIN { a = 50; b = 20; print "(a / b) = ", (a / b) }’
```

Arthematic Operators

Modulus

It is represented by percent (%) symbol which finds the Modulus division of two or more numbers.

```
# awk 'BEGIN { a = 50; b = 20; print "(a % b) = ", (a % b) }'
```

Exponent Operators

There are two formats of exponential operators:

Exponential Format 1

It is an exponential operator that raises the value of an operand.

```
# awk 'BEGIN { a = 10; a = a ^ 2; print "a =", a }'
```

Exponential Format 2

It is an exponential operator that raises the value of an operand.

```
# awk 'BEGIN { a = 10; a = a**2; print "a =", a }'
```

Increment and Decrement Operators

AWK supports the following increment and decrement operators:

Pre-Increment

It is represented by ++. It increments the value of an operand by 1. This operator first increments the value of the operand, then returns the incremented value.

```
# awk 'BEGIN { a = 10; b = ++a; printf "a = %d, b = %d\n", a, b }’
```

Pre-Decrement

It is represented by --. It decrements the value of an operand by 1. This operator first decrements the value of the operand, then returns the decremented value.

```
# awk 'BEGIN { a = 10; b = --a; printf "a = %d, b = %d\n", a, b }'
```

Increment and Decrement Operators

Post-Increment

It is represented by ++. It increments the value of an operand by 1. This operator first returns the value of the operand, then it increments its value.

```
# awk 'BEGIN { a = 10; b = a++; printf "a = %d, b = %d\n", a, b }’
```

Post-Decrement

It is represented by --. It decrements the value of an operand by 1. This operator first returns the value of the operand, then it decrements its value.

```
# awk 'BEGIN { a = 10; b = a--; printf "a = %d, b = %d\n", a, b }'
```

Assignment Operators

AWK supports the following assignment operators:

Simple Assignment

It is represented by =.

```
# awk 'BEGIN { name = "Jerry"; print "My name is", name }'
```

Shorthand Addition

It is represented by +=.

```
# awk 'BEGIN { cnt = 10; cnt += 10; print "Counter =", cnt }'
```

Assignment Operators

Shorthand Subtraction

It is represented by -=.

```
# awk 'BEGIN { cnt = 100; cnt -= 10; print "Counter =", cnt }'
```

Shorthand Multiplication

It is represented by *=.

```
#  awk 'BEGIN { cnt = 10; cnt *= 10; print "Counter =", cnt }'
```

Assignment Operators

Shorthand Division

It is represented by /=.

```
# awk 'BEGIN { cnt = 100; cnt /= 5; print "Counter =", cnt }'
```

Shorthand Modulo

It is represented by %=.

```
# awk 'BEGIN { cnt = 100; cnt %= 8; print "Counter =", cnt }'
```

Assignment Operators

Shorthand Exponential

It is represented by ^=.

```
# awk 'BEGIN { cnt = 2; cnt ^= 4; print "Counter =", cnt }'
```

Shorthand Exponential

It is represented by **=.

```
# awk 'BEGIN { cnt = 2; cnt **= 4; print "Counter =", cnt }'
```

Assignment Operators

Shorthand Exponential

It is represented by ^=.

```
# awk 'BEGIN { cnt = 2; cnt ^= 4; print "Counter =", cnt }'
```

Shorthand Exponential

It is represented by **=.

```
# awk 'BEGIN { cnt = 2; cnt **= 4; print "Counter =", cnt }'
```

Relationship Operators

AWK supports the following relational operators:

Equal to

It is represented by ==. It returns true if both operands are equal, otherwise it returns false.

```
# awk 'BEGIN { a = 10; b = 10; if (a == b) print "a == b" }'
```

Not Equal to

It is represented by !=. It returns true if both operands are unequal, otherwise it returns false.

```
#  awk 'BEGIN { a = 10; b = 20; if (a != b) print "a != b" }'
```

Relationship Operators

Less Than

It is represented by <. It returns true if the left-side operand is less than the right-side operand; otherwise it returns false.

```
# awk 'BEGIN { a = 10; b = 20; if (a < b) print "a  < b" }'
```

Less Than or Equal to

It is represented by <=. It returns true if the left-side operand is less than or equal to the right-side operand; otherwise it returns false.

```
#  awk 'BEGIN { a = 10; b = 10; if (a <= b) print "a <= b" }'
```

Relationship Operators

Greater Than

It is represented by >. It returns true if the left-side operand is greater than the right-side operand, otherwise it returns false.

```
# awk 'BEGIN { a = 10; b = 20; if (b > a ) print "b > a" }'
```

Greater Than or Equal to

It is represented by >=. It returns true if the left-side operand is greater than or equal to the right-side operand; otherwise it returns false.

```
b >= a
```

Logical Operators

AWK supports the following logical operators:

Logical AND

It is represented by &&. Its syntax is as follows:

Syntax

expr1 && expr2

It evaluates to true if both expr1 and expr2 evaluate to true; otherwise it returns false. expr2 is evaluated if and only if expr1 evaluates to true.

```
# awk 'BEGIN {
   num = 5; if (num >= 0 && num <= 7) printf "%d is in octal format\n", num
}'
```

Logical Operators

Logical OR

It is represented by ||. The syntax of Logical OR is:

Syntax

expr1 || expr2

It evaluates to true if either expr1 or expr2 evaluates to true; otherwise it returns false. expr2 is evaluated if and only if expr1 evaluates to false.

```
# awk 'BEGIN {
   ch = "\n"; if (ch == " " || ch == "\t" || ch == "\n")
   print "Current character is whitespace."
}'
```

Logical Operators

Logical NOT

It is represented by exclamation mark (!). The Syntax of logical NOT is:

Syntax

! expr1

It returns the logical compliment of expr1. If expr1 evaluates to true, it returns 0; otherwise it returns 1.

```
# awk 'BEGIN { name = ""; if (! length(name)) print "name is empty string." }'
```

Logical Operators

Logical NOT

It is represented by exclamation mark (!). The Syntax of logical NOT is:

Syntax

! expr1

It returns the logical compliment of expr1. If expr1 evaluates to true, it returns 0; otherwise it returns 1.

```
# awk 'BEGIN { name = ""; if (! length(name)) print "name is empty string." }'
```

Ternary Operators

We can easily implement a condition expression using ternary operator.

Syntax

condition expression ? statement1 : statement2

When the condition expression returns true, statement1 gets executed; otherwise statement2 is executed.

```
# awk 'BEGIN { a = 10; b = 20; (a > b) ? max = a : max = b; print "Max =", max}'
```

Uniary Operators

AWK supports the following unary operators:

Unary Plus

It is represented by +. It multiplies a single operand by +1.

```
# awk 'BEGIN { a = -10; a = +a; print "a =", a }'
```

Unary Minus

It is represented by -. It multiplies a single operand by -1.

```
# awk 'BEGIN { a = -10; a = -a; print "a =", a }'
```

String concatenation Operators

Space is a string concatenation operator that merges two strings.

```
# awk 'BEGIN { str1 = "Hello, "; str2 = "World"; str3 = str1 str2; print str3 }'
```

Array membership Operators

It is represented by **in**. It is used while accessing array elements.

```
# awk 'BEGIN {
   arr[0] = 1; arr[1] = 2; arr[2] = 3; for (i in arr) printf "arr[%d] = %d\n", i, arr[i]
}'
```

Regular Expression Operators

There two forms of regular expressions operators.

Match

It is represented as ~. It looks for a field that contains the match string.

```
# awk '$0 ~ "manager"' employee.txt
```

Not Match

It is represented as !~. It looks for a field that does not contain the match string.

```
# awk '$0 !~ "manager"' employee.txt
```

print

Each print statement makes at least one line of output. However, it isn’t limited to only one line. If an item value is a string containing a newline, the newline is output along with the rest of the string. A single print statement can make any number of lines this way.

```
# awk '{print "Welcome to awk command tutorial "}’
```

```
# awk 'BEGIN { print "line one\nline two\nline three" }’
```

```
# awk '{ print $1, $2 }’ employee.txt
```

```
# awk '{ print $1 $2 }’ employee.txt
```

```
# awk ‘/manager/ {print $0}’ employee.txt
```

print

```
# awk '{ print $2, $2 }' employee.txt ;out of order printing
```

```
#awk 'BEGIN{cnt=0} /manager/ {cnt++} END{print cnt}' employee.txt
```

Length greater than 25

```
#awk 'length($0) > 25' employee.txt
```

printf

For more precise control over the output format than what is provided by print, use printf. With printf you can specify the width to use for each item, as well as various formatting choices for numbers (such as what output base to use, whether to print an exponent, whether to print a sign, and how many digits to print after the decimal point).

```
# awk '{ printf "%-10s %s\n", $1, $2 }' employee.txt
```

```
# awk 'BEGIN { print "Name      Role"
          print "----      ------" }
         { printf "%-10s %s\n", $1, $2 }' employee.txt
```

AWK Regular Expression

AWK is very powerful and efficient in handling regular expressions. A number of complex tasks can be solved with simple regular expressions. Any command-line expert knows the power of regular expressions.

Dot

It matches any single character except the end of line character. For instance, the following example matches fin, fun, fan etc.

```
# echo -e "cat\nbat\nfun\nfin\nfan" | awk '/f.n/'
```

AWK Regular Expression

Start of line

It matches the start of line. For instance, the following example prints all the lines that start with pattern The.

```
# echo -e "This\nThat\nThere\nTheir\nthese" | awk '/^The/'
```

End of line

It matches the end of line. For instance, the following example prints the lines that end with the letter n.

```
# echo -e "knife\nknow\nfun\nfin\nfan\nnine" | awk '/n$/'
```

AWK Regular Expression

Match character set

It is used to match only one out of several characters. For instance, the following example matches pattern Call and Tall but not Ball.

```
# echo -e "Call\nTall\nBall" | awk '/[CT]all/'
```

Exclusive set

In exclusive set, the carat negates the set of characters in the square brackets. For instance, the following example prints only Ball.

```
# echo -e "Call\nTall\nBall" | awk '/[^CT]all/'
```

AWK Regular Expression

Alteration

A vertical bar allows regular expressions to be logically ORed. For instance, the following example prints Ball and Call.

```
# echo -e "Call\nTall\nBall\nSmall\nShall" | awk '/Call|Ball/'
```

Zero or One Occurrence

It matches zero or one occurrence of the preceding character. For instance, the following example matches Colour as well as Color. We have made u as an optional character by using ?.

```
# echo -e "Colour\nColor" | awk '/Colou?r/'
```

AWK Regular Expression

Zero or More Occurrence

It matches zero or more occurrences of the preceding character. For instance, the following example matches ca, cat, catt, and so on.

```
# echo -e "ca\ncat\ncatt" | awk '/cat*/'
```

One or More Occurrence

It matches one or more occurrence of the preceding character. For instance below example matches one or more occurrences of the 2.

```
# echo -e "111\n22\n123\n234\n456\n222"  | awk '/2+/'
```

AWK Regular Expression

Grouping

Parentheses () are used for grouping and the character | is used for alternatives. For instance, the following regular expression matches the lines containing either Apple Juice or Apple Cake.

```
# echo -e "Apple Juice\nApple Pie\nApple Tart\nApple Cake" | awk
   '/Apple (Juice|Cake)/'
```

AWK Arrays

AWK has associative arrays and one of the best thing about it is – the indexes need not to be continuous set of number; you can use either string or number as an array index. Also, there is no need to declare the size of an array in advance – arrays can expand/shrink at runtime.

Its syntax is as follows:

Syntax

array_name[index] = value

Where array_name is the name of array, index is the array index, and value is any value assigning to the element of the array.

AWK Arrays

Creating Array

To gain more insight on array, let us create and access the elements of an array.

```
# awk 'BEGIN {
   fruits["mango"] = "yellow";
   fruits["orange"] = "orange"
   print fruits["orange"] "\n" fruits["mango"]
}'
```

AWK Arrays

Deleting Array Elements

For insertion, we used assignment operator. Similarly, we can use delete statement to remove an element from the array. The syntax of delete statement is as follows:

Syntax

delete array_name[index]

The following example deletes the element orange. Hence the command does not show any output.

```
# awk 'BEGIN {
   fruits["mango"] = "yellow";
   fruits["orange"] = "orange";
   delete fruits["orange"];
   print fruits["orange"]
}'
```

AWK Arrays

Multi-Dimensional arrays

AWK only supports one-dimensional arrays. But you can easily simulate a multi-dimensional array using the one-dimensional array itself.

Syntax

array["0,0"] = 100

Though we gave 0,0 as index, these are not two indexes. In reality, it is just one index with the string 0,0.

AWK Arrays

```
# awk 'BEGIN {
   array["0,0"] = 100;
   array["0,1"] = 200;
   array["0,2"] = 300;
   array["1,0"] = 400;
   array["1,1"] = 500;
   array["1,2"] = 600;

   # print array elements
   print "array[0,0] = " array["0,0"];
   print "array[0,1] = " array["0,1"];
   print "array[0,2] = " array["0,2"];
   print "array[1,0] = " array["1,0"];
   print "array[1,1] = " array["1,1"];
   print "array[1,2] = " array["1,2"];
}'
```

AWK Control Flow

AWK provides conditional statements to control the flow of a program.

If statement

It simply tests the condition and performs certain actions depending upon the condition. Given below is the syntax of if statement:

Syntax

```
if (condition)
   action
```

AWK Control Flow

We can also use a pair of curly braces as given below to execute multiple actions:

Syntax

```
if (condition) {
  action-1
  action-1
  .
  .
  action-n
}
```

For instance, the following example checks whether a number is even or not:

```
# awk 'BEGIN {num = 10; if (num % 2 == 0) printf "%d is even number.\n", num }'
```

AWK Control Flow

If Else Statement

In if-else syntax, we can provide a list of actions to be performed when a condition becomes false.

The syntax of if-else statement is as follows:

Syntax

```
if (condition)
  action-1
else
  action-2
```

In the above syntax, action-1 is performed when the condition evaluates to true and action-2 is performed when the condition evaluates to false. For instance, the following example checks whether a number is even or not:

```
# awk 'BEGIN {
  num = 11; if (num % 2 == 0) printf "%d is even number.\n", num;
    else printf "%d is odd number.\n", num
}'
```

AWK Control Flow

If-Else-If Ladder

We can easily create an if-else-if ladder by using multiple if-else statements.

```
# awk 'BEGIN {
   a = 30;

   if (a==10)
   print "a = 10";
   else if (a == 20)
   print "a = 20";
   else if (a == 30)
   print "a = 30";
}'
```

AWK Loops

Loops are used to execute a set of actions in a repeated manner. The loop execution continues as long as the loop condition is true.

For Loop

The syntax of for loop is:

Syntax

```
for (initialization; condition; increment/decrement)
   action
```

Initially, the for statement performs initialization action, then it checks the condition. If the condition is true, it executes actions, thereafter it performs increment or decrement operation. The loop execution continues as long as the condition is true. For instance, the following example prints 1 to 5 using for loop:

```
# awk 'BEGIN { for (i = 1; i <= 5; ++i) print i }'
```

AWK Loops

While Loop

The while loop keeps executing the action until a particular logical condition evaluates to true. Here is the syntax of while loop:

Syntax

```
while (condition)
   action
```

AWK first checks the condition; if the condition is true, it executes the action. This process repeats as long as the loop condition evaluates to true. For instance, the following example prints 1 to 5 using while loop.

```
# awk 'BEGIN {i = 1; while (i < 6) { print i; ++i } }'
```

AWK Loops

Do-While Loop

The do-while loop is similar to the while loop, except that the test condition is evaluated at the end of the loop. Here is the syntax of do-whileloop:

Syntax

```
do
   action
while (condition)
```

In a do-while loop, the action statement gets executed at least once even when the condition statement evaluates to false. For instance, the following example prints 1 to 5 numbers using do-while loop.

```
# awk 'BEGIN {i = 1; do { print i; ++i } while (i < 6) }'
```

AWK Loops

Break Statement

As its name suggests, it is used to end the loop execution. Here is an example which ends the loop when the sum becomes greater than 50.

```
# awk 'BEGIN {
   sum = 0; for (i = 0; i < 20; ++i) {
      sum += i; if (sum > 50) break; else print "Sum =", sum
   }
}'
```

AWK Loops

Continue Statement

The continue statement is used inside a loop to skip to the next iteration of the loop. It is useful when you wish to skip the processing of some data inside the loop. For instance, the following example uses continue statement to print the even numbers between 1 to 20.

```
# awk 'BEGIN {
   for (i = 1; i <= 20; ++i) {
      if (i % 2 == 0) print i ; else continue
   }
}'
```

AWK Loops

Exit Statement

It is used to stop the execution of the script. It accepts an integer as an argument which is the exit status code for AWK process. If no argument is supplied, exit returns status zero. Here is an example that stops the execution when the sum becomes greater than 50.

```
# awk 'BEGIN {
   sum = 0; for (i = 0; i < 20; ++i) {
      sum += i; if (sum > 50) exit(10); else print "Sum =", sum
   }
}'
```

Let us check the return status of the script.

```
# echo $?
```

User define Functions

AWK allows us to define our own functions. A large program can be divided into functions and each function can be written/tested independently. It provides re-usability of code.

Syntax

```
function function_name(argument1, argument2, ...) {
  function body
}
```

In this syntax, the **function_name** is the name of the user-defined function. Function name should begin with a letter and the rest of the characters can be any combination of numbers, alphabetic characters, or underscore. AWK's reserve words cannot be used as function names.

Functions can accept multiple arguments separated by comma. Arguments are not mandatory. You can also create a user-defined function without any argument.

function body consists of one or more AWK statements.

User define Functions

function.awk

```
# Returns minimum number
function find_min(num1, num2){
   if (num1 < num2)
   return num1
   return num2
}
# Returns maximum number
function find_max(num1, num2){
   if (num1 > num2)
   return num1
   return num2
}
# Main function
function main(num1, num2){
   # Find minimum number
   result = find_min(10, 20)
   print "Minimum =", result

   # Find maximum number
   result = find_max(10, 20)
   print "Maximum =", result
}
# Script execution starts here
BEGIN {
   main(10, 20)
}
```

Arithematic Functions

AWK has the following built-in arithmetic functions:

atan2(y, x)

It returns the arctangent of (y/x) in radians.

```
# awk 'BEGIN {
   PI = 3.14159265
   x = -10
   y = 10
   result = atan2 (y,x) * 180 / PI;

   printf "The arc tangent for (x=%f, y=%f) is %f degrees\n", x, y, result
}'
```

Arithematic Functions

cos(expr)

This function returns the cosine of expr, which is expressed in radians.

```
# awk 'BEGIN {
   PI = 3.14159265
   param = 60
   result = cos(param * PI / 180.0);

   printf "The cosine of %f degrees is %f.\n", param, result
}'
```

Arithematic Functions

exp(expr)

This function is used to find the exponential value of a variable.

```
# awk 'BEGIN {
   param = 5
   result = exp(param);

   printf "The exponential value of %f is %f.\n", param, result
}'
```

Arithematic Functions

int(expr)

This function truncates the expr to an integer value.

```
# awk 'BEGIN {
   param = 5.12345
   result = int(param)

   print "Truncated value =", result
}'
```

Arithematic Functions

log(expr)

This function calculates the natural logarithm of a variable.

```
# awk 'BEGIN {
   param = 5.5
   result = log (param)

   printf "log(%f) = %f\n", param, result
}'
```

Arithematic Functions

rand

This function returns a random number N, between 0 and 1, such that 0 <= N < 1. For instance, the following example generates three random numbers

```
# awk 'BEGIN {
   print "Random num1 =" , rand()
   print "Random num2 =" , rand()
   print "Random num3 =" , rand()
}'
```

Arithematic Functions

sin(expr)

This function returns the sine of expr, which is expressed in radians.

```
# awk 'BEGIN {
   PI = 3.14159265
   param = 30.0
   result = sin(param * PI /180)

   printf "The sine of %f degrees is %f.\n", param, result
}'
```

Arithematic Functions

sqrt(expr)

This function returns the square root of expr.

```
# awk 'BEGIN {
   param = 1024.0
   result = sqrt(param)

   printf "sqrt(%f) = %f\n", param, result
}'
```

Arithematic Functions

srand([expr])

This function generates a random number using seed value. It uses expr as the new seed for the random number generator. In the absence of expr, it uses the time of day as the seed value.

```
# awk 'BEGIN {
   param = 10

   printf "srand() = %d\n", srand()
   printf "srand(%d) = %d\n", param, srand(param)
}'
```

String Functions

AWK has the following built-in String functions:

gsub(regex, sub, string)

gsub stands for global substitution. It replaces every occurrence of regex with the given string (sub). The third parameter is optional. If it is omitted, then $0 is used.

```
# awk 'BEGIN {
   str = "Hello, World"
   print "String before replacement = " str

   gsub("World", "Jerry", str)
   print "String after replacement = " str
}'
```

String Functions

index(str, sub)

It checks whether sub is a substring of str or not. On success, it returns the position where sub starts; otherwise it returns 0. The first character of str is at position 1.

```
# awk 'BEGIN {
   str = "One Two Three"
   subs = "Two"
   ret = index(str, subs)

   printf "Substring \"%s\" found at %d location.\n", subs, ret
}'
```

String Functions

length(str)

It returns the length of a string.

```
# awk 'BEGIN {
   str = "Hello, World !!!"
   print "Length = ", length(str)
}'
```

String Functions

match(str, regex)

It returns the index of the first longest match of regex in string str. It returns 0 if no match found.

```
# awk 'BEGIN {
   str = "One Two Three"
   subs = "Two"
   ret = match(str, subs)

   printf "Substring \"%s\" found at %d location.\n", subs, ret
}'
```

String Functions

split(str, arr, regex)

This function splits the string str into fields by regular expression regex and the fields are loaded into the array arr. If regex is omitted, then FS is used.

```
# awk 'BEGIN {
   str = "One,Two,Three,Four"
   split(str, arr, ",")
   print "Array contains following values"

   for (i in arr) {
      print arr[i]
   }
}'
```

String Functions

strtonum(str)

This function examines str and return its numeric value. If str begins with a leading 0, it is treated as an octal number. If str begins with a leading 0x or 0X, it is taken as a hexadecimal number. Otherwise, assume it is a decimal number.

```
# awk 'BEGIN {
   print "Decimal num = " strtonum("123")
   print "Octal num = " strtonum("0123")
   print "Hexadecimal num = " strtonum("0x123")
}'
```

String Functions

sub(regex, sub, string)

This function performs a single substitution. It replaces the first occurrence of the regex pattern with the given string (sub). The third parameter is optional. If it is omitted, $0 is used.

```
# awk 'BEGIN {
   str = "Hello, World"
   print "String before replacement = " str

   sub("World", "Jerry", str)
   print "String after replacement = " str
}'
```

String Functions

substr(str, start, l)

This function returns the substring of string str, starting at index start of length l. If length is omitted, the suffix of str starting at index start is returned.

```
# awk 'BEGIN {
   str = "Hello, World !!!"
   subs = substr(str, 1, 5)

   print "Substring = " subs
}'
```

String Functions

tolower(str)

This function returns a copy of string str with all upper-case characters converted to lower-case.

```
# awk 'BEGIN {
   str = "HELLO, WORLD !!!"
   print "Lowercase string = " tolower(str)
}'
```

toupper(str)

This function returns a copy of string str with all lower-case characters converted to upper case.

```
# awk 'BEGIN {
   str = "hello, world !!!"
   print "Uppercase string = " toupper(str)
}'
```

Redirection

Redirection Operator

The syntax of the redirection operator is:

Syntax

print DATA > output-file

It writes the data into the output-file. If the output-file does not exist, then it creates one. When this type of redirection is used, the output-file is erased before the first output is written to it. Subsequent write operations to the same output-file do not erase the output-file, but append to it. For instance, the following example writes Hello, World !!! to the file.

```
# awk 'BEGIN { print "Hello, World !!!" > "/tmp/message.txt" }'
```

Redirection

Append Operator

The syntax of append operator is as follows:

Syntax

print DATA >> output-file

It appends the data into the output-file. If the output-file does not exist, then it creates one. When this type of redirection is used, new contents are appended at the end of file. For instance, the following example appends Hello, World !!! to the file.

```
# awk 'BEGIN { print "Hello, World !!!" >> "/tmp/message.txt" }'
```

Auto tools

- Yes, the *autotools* are old
- Yes, they have their pain points
- Yes, people hate them
- Due to this, people tend to roll-their-own, and roll-their-own build systems tend to be even worse than the *autotools*
- But
  - They bring a number of very useful benefits
  - They are not that complicated when you take the time to get back to the basics

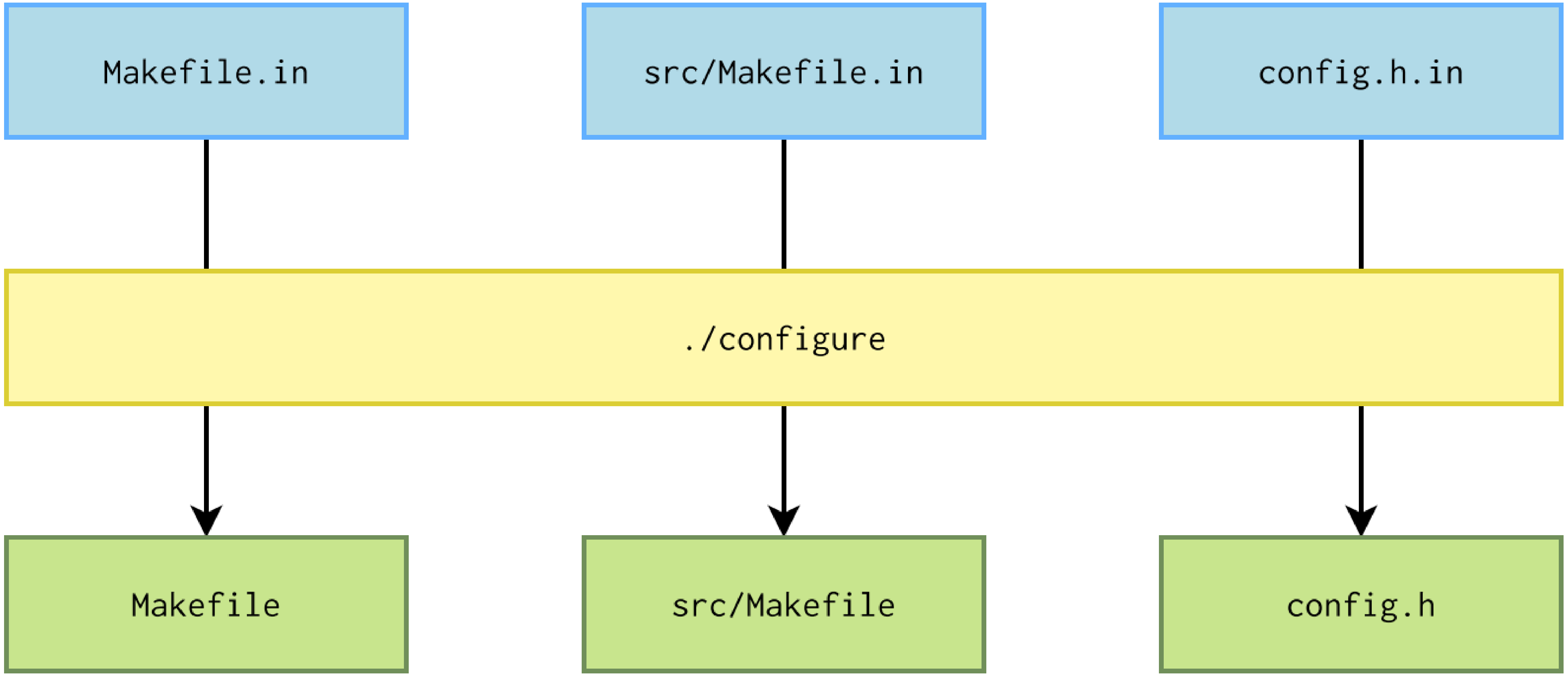
Auto tools

- Standardized build procedure and behavior: users know how to build things that use the *autotools*
  - Good for human users, but also for build systems
- Proper handling for diverted installation
  - I.e. build with `prefix=/usr`, but divert the installation to another directory. Needed for cross-compilation.
- Built-in support for out-of-tree build
- Built-in handling of dependencies on header files
- Support for cross-compilation aspects
- Somewhat esoteric, but standardized languages used
  - Learn once, use for many projects
  - New contributors are more likely to know the *autotools* than your own custom thing
- Of course, there are alternatives, *CMake* being the most interesting and widely used.

Auto tools

- The basic steps to build an *autotools* based software component are:
  1. **Configuration**
     `./configure`
     Will look at the available build environment, verify required dependencies, generate `Makefile`s and a `config.h`
  2. **Compilation**
     `make`
     Actually builds the software component, using the generated Makefiles.
  3. **Installation**
     `make install`
     Installs what has been built.

Auto tools


Makefile.in
src/Makefile.in
config.h.in
./configure
Makefile
src/Makefile
config.h

Auto tools

- `all`, builds everything. The default target.
- `install`, installs everything that should be installed.
- `install-strip`, same as `install`, but then strips debugging symbols
- `uninstall`
- `clean`, remove what was built
- `distclean`, same as `clean`, but also removes the generated *autotools* files
- `check`, run the test suite
- `installcheck`, check the installation
- `dist`, create a tarball

Auto tools

- `prefix`, defaults to */usr/local*
  - `exec-prefix`, defaults to `prefix`
    - `bindir`, for programs, defaults to `exec-prefix/`*bin*
    - `libdir`, for libraries, defaults to `exec-prefix/`*lib*
- `includedir`, for headers, defaults to `prefix/`*include*
- `datarootdir`, defaults to `prefix/`*share*
  - `datadir`, defaults to `datarootdir`
  - `mandir`, for man pages, defaults to `datarootdir/`*man*
  - `infodir`, for info documents, defaults to `datarootdir/`*info*
- `sysconfdir`, for configuration files, defaults to `prefix/`*etc*
- `--<option>` available for each of them
  - E.g: `./configure --prefix=~/sys/`

Auto tools

- `CC`, C compiler command
- `CFLAGS`, C compiler flags
- `CXX`, C++ compiler command
- `CXXFLAGS`, C++ compiler flags
- `LDFLAGS`, linker flags
- `CPPFLAGS`, C/C++ preprocessor flags
- and many more, see `./configure --help`
- E.g: `./configure CC=arm-linux-gcc`

Auto tools

- *autotools* identify three **system types**:
  - **build**, which is the system where the build takes place
  - **host**, which is the system where the execution of the compiled code will take place
  - **target**, which is the system for which the program will generate code. This is only used for compilers, assemblers, linkers, etc.
- Corresponding `--build`, `--host` and `--target` *configure* options.
  - They are all automatically *guessed* to the current machine by default
  - `--build`, generally does not need to be changed
  - `--host`, must be overridden to do cross-compilation
  - `--target`, needs to be overridden if needed (to generate a cross-compiler, for example)
- Arguments to these options are *configuration names*, also called *system tuples*

Auto tools Cross compilation

- By default, *autotools* will guess the **host** machine as being the current machine
- To cross-compile, it must be overridden by passing the `--host` option with the appropriate *configuration name*
- By default, *autotools* will try to use the cross-compilation tools that use the *configuration name* as their prefix.
- If not, the variables `CC`, `CXX`, `LD`, `AR`, etc. can be used to point to the cross-compilation tools.

Auto tools Out of Tree build

- *autotools* support out of tree compilation by default
- Consists in doing the build in a directory separate from the source directory
- Allows to:
  - Build different configurations without having to rebuild from scratch each time.
  - Not clutter the source directory with build related files
- To use out of tree compilation, simply run the configure script from another empty directory
  - This directory will become the build directory

Auto tools install

- By default, `make install` installs to the directories given in `--prefix` and related options.
- In some situations, it is useful to *divert* the installation to another directory
  - Cross-compilation, where the build machine is not the machine where applications will be executed.
  - Packaging, where the installation needs to be done in a temporary directory.
- Achieved using the `DESTDIR` variable.

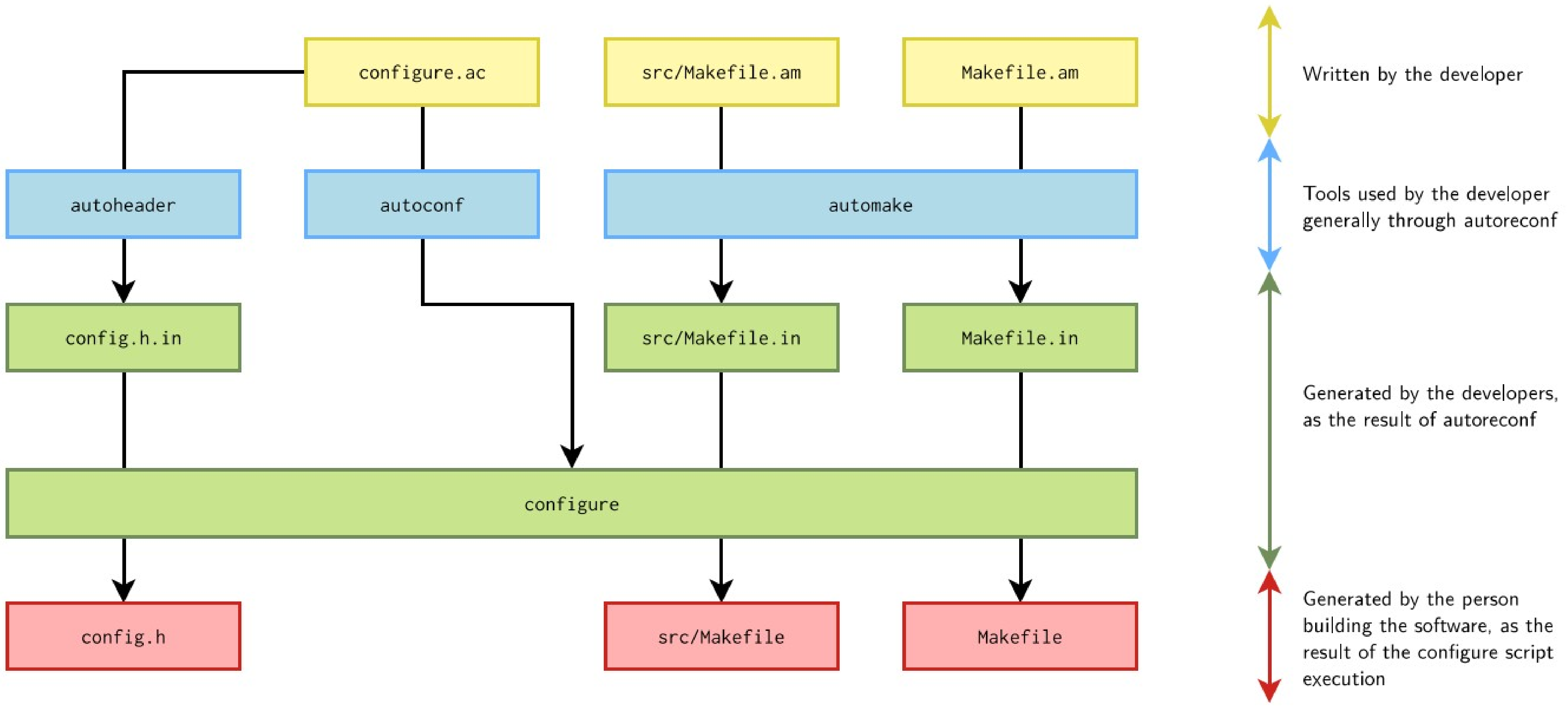
Auto tools Analyzing issues

- `autoconf` keeps a log of all the tests it runs in a file called `config.log`
- Very useful for analysis of `autoconf` issues
- It contains several sections: *Platform*, *Core tests*, *Running config.status*, *Cache variables*, *Output variables*, *confdefs.h*
- The end of the *Core tests* section is usually the most interesting part
  - This is where you would get more details about the reason of the *configure* script failure
- At the beginning of `config.log` you can also see the `./configure` line that was used, with all options and environment variables.

Auto tools

- The `configure` script is a shell script generated from `configure.ac` by a program called `autoconf`
  - `configure.ac` used to be named `configure.in` but this name is now deprecated
  - Written in shell script, augmented with numerous *m4* macros
- The `Makefile.in` are generated from `Makefile.am` files by a program called `automake`
  - Uses special `make` variables that are expanded in standard `make` constructs
- Some auxilliary tools like `autoheader` or `aclocal` are also used
  - `autoheader` is responsible for generating the *configuration header* template, `config.h.in`
- Generated files (`configure`, `Makefile.in`, `Makefile`) should not be modified.
  - Reading them is also very difficult. Read the real source instead!

Auto tools


configure.ac
src/Makefile.am
Makefile.am
autoheader
autoconf
automake
config.h.in
src/Makefile.in
Makefile.in
configure
config.h
src/Makefile
Makefile
Written by the developer
Tools used by the developer generally through autoreconf
Generated by the developers, as the result of autoreconf
Generated by the person building the software, as the result of the configure script execution

kbuild

kbuild was originially developed to be used with the Linux Kernel. It implements a powerfull configuration and build infrastructure that allowes you to build a project with only the sources, that are really needed to fit your configuration.

But Kbuild can not only be used along with the Linux kernel. Lots of other projects use the Kbuild system as well. A few famos examples:

buildroot -- https://buildroot.org/

zephyr -- https://www.zephyrproject.org/

U-Boot -- http://www.denx.de/wiki/U-Boot

The Kbuild is a makefile framework for writing simple makefiles for complex and configurable tasks that you can easily use for your own purposes.

kbuild

The top Makefile reads the .config file, which comes from the application configuration process.

The top Makefile is responsible for building building the application It builds this goal by recursively descending into the subdirectories of the application source tree. The list of subdirectories which are visited depends upon the application configuration. The top Makefile can include several other Makefiles that supply function specific information to the top Makefile.

Each subdirectory has a kbuild Makefile which carries out the commands passed down from above. The kbuild Makefile uses information from the .config file to construct various file lists used by kbuild to build any built-in targets.

scripts/Makefile.* contains all the definitions/rules etc. that are used to build the application based on the kbuild makefiles.

Maps,SMAPS

Each row in /proc/$PID/maps describes a region of contiguous virtual memory in a process or thread. Each row has the following fields:

```
address           perms offset  dev   inode   pathname
08048000-08056000 r-xp 00000000 03:0c 64593   /usr/sbin/gpm
```

- **address -** This is the starting and ending address of the region in the process's address space
- **permissions -** This describes how pages in the region can be accessed. There are four different permissions: read, write, execute, and shared. If read/write/execute are disabled, a - will appear instead of the r/w/x. If a region is not shared, it is private, so a p will appear instead of an s. If the process attempts to access memory in a way that is not permitted, a segmentation fault is generated. Permissions can be changed using the mprotect system call.

r = read w = write x = execute s = shared p = private (copy on write)

Maps,SMAPS

offset - If the region was mapped from a file (using mmap), this is the offset in the file where the mapping begins. If the memory was not mapped from a file, it's just 0.

device - If the region was mapped from a file, this is the major and minor device number (in hex) where the file lives.

inode - If the region was mapped from a file, this is the file number.

pathname - If the region was mapped from a file, this is the name of the file. This field is blank for anonymous mapped regions. There are also special regions with names like [heap], [stack], or [vdso]. [vdso] stands for virtual dynamic shared object. It's used by system calls to switch to kernel mode.

RSS,PSS

RSS is the Resident Set Size and is used to show how much memory is allocated to that process and is in RAM. It does not include memory that is swapped out. It does include memory from shared libraries as long as the pages from those libraries are actually in memory. It does include all stack and heap memory.

VSZ is the Virtual Memory Size. It includes all memory that the process can access, including memory that is swapped out, memory that is allocated, but not used, and memory that is from shared libraries.

RSS,PSS

So if process A has a 500K binary and is linked to 2500K of shared libraries, has 200K of stack/heap allocations of which 100K is actually in memory (rest is swapped or unused), and it has only actually loaded 1000K of the shared libraries and 400K of its own binary then:

RSS: 400K + 1000K + 100K = 1500K

VSZ: 500K + 2500K + 200K = 3200K

Since part of the memory is shared, many processes may use it, so if you add up all of the RSS values you can easily end up with more space than your system has.

The memory that is allocated also may not be in RSS until it is actually used by the program. So if your program allocated a bunch of memory up front, then uses it over time, you could see RSS going up and VSZ staying the same.

RSS,PSS

There is also **PSS (proportional set size).** This is a newer measure which tracks the shared memory as a proportion used by the current process. So if there were two processes using the same shared library from before:

PSS: 400K + (1000K/2) + 100K = 400K + 500K + 100K = 1000K

Threads all share the same address space, so the RSS, VSZ and PSS for each thread is identical to all of the other threads in the process. Use ps or top to view this information in linux/unix.

UIO Driver

For many types of devices, creating a Linux kernel driver is overkill. All that is really needed is some way to handle an interrupt and provide access to the memory space of the device. The logic of controlling the device does not necessarily have to be within the kernel, as the device does not need to take advantage of any of other resources that the kernel provides. One such common class of devices that are like this are for industrial I/O cards.

To address this situation, the userspace I/O system (UIO) was designed. For typical industrial I/O cards, only a very small kernel module is needed. The main part of the driver will run in user space. This simplifies development and reduces the risk of serious bugs within a kernel module.

UIO Driver

Devices that are already handled well by other kernel subsystems (like networking or serial or USB) are no candidates for an UIO driver. Hardware that is ideally suited for an UIO driver fulfills all of the following:

- The device has memory that can be mapped. The device can be controlled completely by writing to this memory.
- The device usually generates interrupts.
- The device does not fit into one of the standard kernel subsystems.

UIO Driver

Devices that are already handled well by other kernel subsystems (like networking or serial or USB) are no candidates for an UIO driver. Hardware that is ideally suited for an UIO driver fulfills all of the following:

- The device has memory that can be mapped. The device can be controlled completely by writing to this memory.
- The device usually generates interrupts.
- The device does not fit into one of the standard kernel subsystems.

UIO Driver Advantages

If you use UIO for your card’s driver, here’s what you get:

- only one small kernel module to write and maintain.
- develop the main part of your driver in user space, with all the tools and libraries you’re used to.
- bugs in your driver won’t crash the kernel.
- updates of your driver can take place without recompiling the kernel.

UIO Driver Working

Each UIO device is accessed through a device file and several sysfs attribute files. The device file will be called /dev/uio0 for the first device, and /dev/uio1, /dev/uio2 and so on for subsequent devices.

/dev/uioX is used to access the address space of the card. Just use mmap() to access registers or RAM locations of your card.

Interrupts are handled by reading from /dev/uioX. A blocking read() from /dev/uioX will return as soon as an interrupt occurs. You can also use select() on /dev/uioX to wait for an interrupt. The integer value read from /dev/uioX represents the total interrupt count. You can use this number to figure out if you missed some interrupts.

UIO Driver Working

For some hardware that has more than one interrupt source internally, but not separate IRQ mask and status registers, there might be situations where userspace cannot determine what the interrupt source was if the kernel handler disables them by writing to the chip's IRQ register. In such a case, the kernel has to disable the IRQ completely to leave the chip's register untouched. Now the userspace part can determine the cause of the interrupt, but it cannot re-enable interrupts. Another cornercase is chips where re-enabling interrupts is a read-modify-write operation to a combined IRQ status/acknowledge register. This would be racy if a new interrupt occurred simultaneously.

To address these problems, UIO also implements a write() function. It is normally not used and can be ignored for hardware that has only a single interrupt source or has separate IRQ mask and status registers. If you need it, however, a write to /dev/uioX will call the irqcontrol() function implemented by the driver. You have to write a 32-bit value that is usually either 0 or 1 to disable or enable interrupts. If a driver does not implement irqcontrol(), write() will return with -ENOSYS.

UIO Driver Working

To handle interrupts properly, your custom kernel module can provide its own interrupt handler. It will automatically be called by the built-in handler.

For cards that don’t generate interrupts but need to be polled, there is the possibility to set up a timer that triggers the interrupt handler at configurable time intervals. This interrupt simulation is done by calling uio_event_notify() from the timer’s event handler.

Each driver provides attributes that are used to read or write variables. These attributes are accessible through sysfs files. A custom kernel driver module can add its own attributes to the device owned by the uio driver, but not added to the UIO device itself at this time. This might change in the future if it would be found to be useful.

UIO Driver Attributes

The following standard attributes are provided by the UIO framework:

- name: The name of your device. It is recommended to use the name of your kernel module for this.
- version: A version string defined by your driver. This allows the user space part of your driver to deal with different versions of the kernel module.
- event: The total number of interrupts handled by the driver since the last time the device node was read.

These attributes appear under the /sys/class/uio/uioX directory. Please note that this directory might be a symlink, and not a real directory. Any userspace code that accesses it must be able to handle this.

UIO Driver Attributes

The following standard attributes are provided by the UIO framework:

- name: The name of your device. It is recommended to use the name of your kernel module for this.
- version: A version string defined by your driver. This allows the user space part of your driver to deal with different versions of the kernel module.
- event: The total number of interrupts handled by the driver since the last time the device node was read.

These attributes appear under the /sys/class/uio/uioX directory. Please note that this directory might be a symlink, and not a real directory. Any userspace code that accesses it must be able to handle this.

UIO Driver Memory mapping

Each UIO device can make one or more memory regions available for memory mapping. This is necessary because some industrial I/O cards require access to more than one PCI memory region in a driver.

Each mapping has its own directory in sysfs, the first mapping appears as /sys/class/uio/uioX/maps/map0/. Subsequent mappings create directories map1/, map2/, and so on. These directories will only appear if the size of the mapping is not 0.

UIO Driver Memory mapping

Each mapX/ directory contains four read-only files that show attributes of the memory:

- **name:** A string identifier for this mapping. This is optional, the string can be empty. Drivers can set this to make it easier for userspace to find the correct mapping.
- **addr:** The address of memory that can be mapped.
- **size:** The size, in bytes, of the memory pointed to by addr.
- **offset:** The offset, in bytes, that has to be added to the pointer returned by mmap() to get to the actual device memory. This is important if the device's memory is not page aligned. Remember that pointers returned by mmap() are always page aligned, so it is good style to always add this offset.

UIO Driver Memory mapping

Each mapX/ directory contains four read-only files that show attributes of the memory:

- **name:** A string identifier for this mapping. This is optional, the string can be empty. Drivers can set this to make it easier for userspace to find the correct mapping.
- **addr:** The address of memory that can be mapped.
- **size:** The size, in bytes, of the memory pointed to by addr.
- **offset:** The offset, in bytes, that has to be added to the pointer returned by mmap() to get to the actual device memory. This is important if the device's memory is not page aligned. Remember that pointers returned by mmap() are always page aligned, so it is good style to always add this offset.

UIO Driver PORT IO

The new directory /sys/class/uio/uioX/portio/ is added. It only exists if the driver wants to pass information about one or more port regions to userspace. If that is the case, subdirectories named port0, port1, and so on, will appear underneath /sys/class/uio/uioX/portio/.

Each portX/ directory contains four read-only files that show name, start, size, and type of the port region:

- **name:** A string identifier for this port region. The string is optional and can be empty. Drivers can set it to make it easier for userspace to find a certain port region.
- **start:** The first port of this region.
- **size:** The number of ports in this region.
- **porttype:** A string describing the type of port.

UIO Driver PORT IO

The new directory /sys/class/uio/uioX/portio/ is added. It only exists if the driver wants to pass information about one or more port regions to userspace. If that is the case, subdirectories named port0, port1, and so on, will appear underneath /sys/class/uio/uioX/portio/.

Each portX/ directory contains four read-only files that show name, start, size, and type of the port region:

- **name:** A string identifier for this port region. The string is optional and can be empty. Drivers can set it to make it easier for userspace to find a certain port region.
- **start:** The first port of this region.
- **size:** The number of ports in this region.
- **porttype:** A string describing the type of port.

UIO Driver Userspace

Once you have a working kernel module for your hardware, you can write the userspace part of your driver. You don’t need any special libraries, your driver can be written in any reasonable language, you can use floating point numbers and so on. In short, you can use all the tools and libraries you’d normally use for writing a userspace application.

UIO Driver Information

Information about all UIO devices is available in sysfs. The first thing you should do in your driver is check name and version to make sure your talking to the right device and that its kernel driver has the version you expect.

You should also make sure that the memory mapping you need exists and has the size you expect.

There is a tool called lsuio that lists UIO devices and their attributes. It is available here:

http://www.osadl.org/projects/downloads/UIO/user/

With lsuio you can quickly check if your kernel module is loaded and which attributes it exports. Have a look at the manpage for details.

The source code of lsuio can serve as an example for getting information about an UIO device. The file uio_helper.c contains a lot of functions you could use in your userspace driver code.

UIO Driver mmap

After you made sure you’ve got the right device with the memory mappings you need, all you have to do is to call mmap() to map the device’s memory to userspace.

The parameter offset of the mmap() call has a special meaning for UIO devices.

UIO Driver mmap

After you made sure you’ve got the right device with the memory mappings you need, all you have to do is to call mmap() to map the device’s memory to userspace.

The parameter offset of the mmap() call has a special meaning for UIO devices.

UIO Driver Interrupt

After you successfully mapped your devices memory, you can access it like an ordinary array. Usually, you will perform some initialization. After that, your hardware starts working and will generate an interrupt as soon as it’s finished, has some data available, or needs your attention because an error occurred.

/dev/uioX is a read-only file. A read() will always block until an interrupt occurs. There is only one legal value for the count parameter of read(), and that is the size of a signed 32 bit integer (4). Any other value for count causes read() to fail. The signed 32 bit integer read is the interrupt count of your device. If the value is one more than the value you read the last time, everything is OK. If the difference is greater than one, you missed interrupts.

You can also use select() on /dev/uioX.

UIO Driver code

```
#include  <stdlib.h>
#include  <stdio.h>
#include  <unistd.h>
#include  <fcntl.h>

int main()
{
        int fd;
        unsigned long nint;
        if ((fd = open("/dev/uio0", O_RDONLY)) < 0) {
        perror("Failed to open /dev/uio0\n");
        exit(EXIT_FAILURE);
        }
        fprintf(stderr, "Started uio test driver.\n");
        while (read(fd, &nint, sizeof(nint)) >= 0)
        fprintf(stderr, "Interrupts: %ld\n", nint);
        exit(EXIT_SUCCESS);
}
```

Thank you