Contents

Chapter 1

Introduction to Parrot

Welcome to Parrot

This document provides a gentle introduction to the Parrot virtual machine for anyone considering writing code for Parrot by hand, writing a compiler that targets Parrot, getting involved with Parrot development or simply wondering what on earth Parrot is.

What is Parrot?

Virtual Machines

Parrot is a virtual machine. To understand what a virtual machine is, consider what happens when you write a program in a language such as Perl, then run it with the applicable interpreter (in the case of Perl, the perl executable). First, the program you have written in a high level language is turned into simple instructions, for example *fetch the value of the variable named x*, *add 2 to this value*, *store this value in the variable named y*, etc. A single line of code in a high level language may be converted into tens of these simple instructions. This stage is called *compilation*.

The second stage involves executing these simple instructions. Some languages (for example, C) are often compiled to instructions that are understood by the CPU and as such can be executed by the hardware. Other languages, such as Perl, Python and Java, are usually compiled to CPU-independent instructions. A *virtual machine* (sometimes known as an *interpreter*) is required to execute those instructions.

While the central role of a virtual machine is to efficiently execute instructions, it also performs a number of other functions. One of these is to abstract away the details of the hardware and operating system that a program is running on. Once a program has been compiled to run on a virtual machine, it will run on any platform that the VM has been implemented on. VMs may also provide security by allowing more fine-grained limitations to be placed on a program, memory management functionality and support for high level language features (such as objects, data structures, types, subroutines, etc).

Design goals

Parrot is designed with the needs of dynamically typed languages (such as Perl and Python) in mind, and should be able to run programs written in these languages more efficiently than VMs developed with static languages in mind (JVM, .NET). Parrot is also designed to provide interoperability between languages that compile to it. In theory, you will be able to write a class in Perl, subclass it in Python and then instantiate and use that subclass in a Tcl program.

Historically, Parrot started out as the runtime for Perl 6. Unlike Perl 5, the Perl 6 compiler and runtime (VM) are to be much more clearly separated. The name *Parrot* was chosen after the 2001 April Fool's Joke which had Perl and Python collaborating on the next version of their languages. The name reflects the intention to build a VM to run not just Perl 6, but also many other languages.

Parrot concepts and jargon

Instruction formats

Parrot can currently accept instructions to execute in four forms. PIR (Parrot Intermediate Representation) is designed to be written by people and generated by compilers. It hides away some low-level details, such as the way parameters are passed to functions. PASM (Parrot Assembly) is a level below PIR - it is still human readable/writable and can be generated by a compiler, but the author has to take care of details such as calling conventions and register allocation. PAST (Parrot Abstract Syntax Tree) enables Parrot to accept an abstract syntax tree style input - useful for those writing compilers.

All of the above forms of input are automatically converted inside Parrot to PBC (Parrot Bytecode). This is much like machine code, but understood by the Parrot interpreter. It is not intended to be human-readable or human-writable, but unlike the other forms execution can start immediately, without the need for an assembly phase. Parrot bytecode is platform independent.

The instruction set

The Parrot instruction set includes arithmetic and logical operators, compare and branch/jump (for implementing loops, if...then constructs, etc), finding and storing global and lexical variables, working with classes and objects, calling subroutines and methods along with their parameters, I/O, threads and more.

Registers and fundamental data types

The Parrot VM is register based. This means that, like a hardware CPU, it has a number of fast-access units of storage called registers. There are 4 types of register in Parrot: integers (I), numbers (N), strings (S) and PMCs (P). There are N of each of these, named I0,I1,..N0.., etc. Integer registers are the same size as a word on the machine Parrot is running on and number registers also map to a native floating point type. The amount of registers needed is determined per subroutine at compile-time.

PMCs

PMC stands for Polymorphic Container. PMCs represent any complex data structure or type, including aggregate data types (arrays, hash tables, etc). A PMC can implement its own behavior for arithmetic, logical and string operations performed on it, allowing for language-specific behavior to be introduced. PMCs can be built in to the Parrot executable or dynamically loaded when they are needed.

Garbage Collection

Parrot provides garbage collection, meaning that Parrot programs do not need to free memory explicitly; it will be freed when it is no longer in use (that is, no longer referenced) whenever the garbage collector runs.

Obtaining, building and testing Parrot

Where to get Parrot

See http://www.parrot.org/download for several ways to get a recent version of parrot.

Building Parrot

The first step to building Parrot is to run the *Configure.pl* program, which looks at your platform and decides how Parrot should be built. This is done by typing:

```
perl Configure.pl
```

Once this is complete, run the `make` program `Configure.pl` prompts you with. When this completes, you will have a working `parrot` executable.

Please report any problems that you encounter while building Parrot so the developers can fix them. You can do this by creating a login and opening a new ticket at https://trac.parrot.org. Please include the *myconfig* file that was generated as part of the build process and any errors that you observed.

The Parrot test suite

Parrot has an extensive regression test suite. This can be run by typing:

```
make test
```

Substituting make for the name of the make program on your platform. The output will look something like this:

```
C:\Perl\bin\perl.exe t\harness --gc-debug
  t\library\*.t  t\op\*.t  t\pmc\*.t  t\run\*.t  t\native_pbc\*.t
  imcc\t\*\*.t  t\dynpmc\*.t  t\p6rules\*.t t\src\*.t t\perl\*.t
t\library\dumper...............ok
t\library\getopt_long..........ok
...
All tests successful, 4 test and 71 subtests skipped.
Files=163, Tests=2719, 192 wallclock secs ( 0.00 cusr +  0.00 csys =  0.00 CPU)
```

It is possible that a number of tests may fail. If this is a small number, then it is probably little to worry about, especially if you have the latest Parrot sources from the SVN repository. However, please do not let this discourage you from reporting test failures, using the same method as described for reporting build problems.

Some simple Parrot programs

Hello world!

Create a file called *hello.pir* that contains the following code.

```
.sub main
    say "Hello world!"
.end
```

Then run it by typing:

```
parrot hello.pir
```

As expected, this will display the text `Hello world!` on the console, followed by a new line.

Let's take the program apart. `.sub main` states that the instructions that follow make up a subroutine named `main`, until a `.end` is encountered. The second line contains the `print` instruction. In this case, we are calling the variant of the instruction that accepts a constant string. The assembler takes care of deciding which variant of the instruction to use for us.

Using registers

We can modify hello.pir to first store the string `Hello world!` in a register and then use that register with the print instruction.

```
.sub main
    $S0 = "Hello world!"
    say $S0
.end
```

PIR does not allow us to set a register directly. We need to prefix the register name with `$` when referring to a register. The compiler will map $S0 to one of the available string registers, for example S0, and set the value. This example also uses the syntactic sugar provided by the `=` operator. `=` is simply a more readable way of using the `set` opcode.

To make PIR even more readable, named registers can be used. These are later mapped to real numbered registers.

```
.sub main
    .local string hello
    hello = "Hello world!"
    say hello
.end
```

The `.local` directive indicates that the named register is only needed inside the current subroutine (that is, between `.sub` and `.end`). Following `.local` is a type. This can be `int` (for I registers), `float` (for N registers), `string` (for S registers), `pmc` (for P registers) or the name of a PMC type.

PIR vs. PASM

PASM does not handle register allocation or provide support for named registers. It also does not have the `.sub` and `.end` directives, instead replacing them with a label at the start of the instructions.

Summing squares

This example introduces some more instructions and PIR syntax. Lines starting with a `#` are comments.

```
.sub main
    # State the number of squares to sum.
    .local int maxnum
    maxnum = 10

    # We'll use some named registers. Note that we can declare many
    # registers of the same type on one line.
    .local int i, total, temp
    total = 0

    # Loop to do the sum.
    i = 1
loop:
    temp = i * i
    total += temp
    inc i
    if i <= maxnum goto loop

    # Output result.
    print "The sum of the first "
    print maxnum
    print " squares is "
    print total
    print ".\n"
.end
```

PIR provides a bit of syntactic sugar that makes it look more high level than assembly. For example:

```
.local pmc temp, i
temp = i * i
```

Is just another way of writing the more assembly-ish:

```
.local pmc temp, i
mul temp, i, i
```

And:

```
.local pmc i, maxnum
if i <= maxnum goto loop
# ...
loop:
```

Is the same as:

```
.local pmc i, maxnum
le i, maxnum, loop
# ...
loop:
```

And:

```
.local pmc temp, total
total += temp
```

Is the same as:

```
.local pmc  temp, total
add total, temp
```

As a rule, whenever a Parrot instruction modifies the contents of a register, that will be the first register when writing the instruction in assembly form.

As is usual in assembly languages, loops and selection are implemented in terms of conditional branch statements and labels, as shown above. Assembly programming is one place where using goto is not bad form!

Recursively computing factorial

In this example we define a factorial function and recursively call it to compute factorial.

```
.sub factorial
    # Get input parameter.
    .param int n

    # return (n > 1 ? n * factorial(n - 1) : 1)
    .local int result

    if n > 1 goto recurse
    result = 1
    goto return

recurse:
    $I0 = n - 1
    result = factorial($I0)
    result *= n

return:
    .return (result)
.end


.sub main :main
    .local int f, i

    # We'll do factorial 0 to 10.
    i = 0
loop:
    f = factorial(i)

    print "Factorial of "
    print i
    print " is "
    print f
    print ".\n"

    inc i
    if i <= 10 goto loop
.end
```

The first line, `.param int n`, specifies that this subroutine takes one integer parameter and that we'd like to refer to the register it was passed in by the name `n` for the rest of the sub.

Much of what follows has been seen in previous examples, apart from the line reading:

```
.local int result
result = factorial($I0)
```

The last line of PIR actually represents a few lines of PASM. The assembler builds a PMC that describes the signature, including which register the arguments are held in. A similar process happens for providing the registers that the return values should be placed in. Finally, the `factorial` sub is invoked.

Right before the `.end` of the `factorial` sub, a `.return` directive is used to specify that the value held in the register named `result` is to be copied to the register that the caller is expecting the return value in.

The call to `factorial` in main works in just the same was as the recursive call to `factorial` within the sub `factorial` itself. The only remaining bit of new syntax is the `:main`, written after `.sub main`. By default, PIR assumes that execution begins with the first sub in the file. This behavior can be changed by marking the sub to start in with `:main`.

Compiling to PBC

To compile PIR to bytecode, use the `-o` flag and specify an output file with the extension *.pbc*.

```
parrot -o factorial.pbc factorial.pir
```

Where next?

Documentation

What documentation you read next depends upon what you are looking to do with Parrot. The opcodes reference and built-in PMCs reference are useful to dip into for pretty much everyone. If you intend to write or compile to PIR then there are a number of documents about PIR that are worth a read. For compiler writers, the Compiler FAQ is essential reading. If you want to get involved with Parrot development, the PDDs (Parrot Design Documents) contain some details of the internals of Parrot; a few other documents fill in the gaps. One way of helping Parrot development is to write tests, and there is a document entitled *Testing Parrot* that will help with this.

The Parrot Mailing List

Much Parrot development and discussion takes place on the parrot-dev mailing list. You can subscribe by filling out the form at http://lists.parrot.org/mailman/listinfo/parrot-dev or read the NNTP archive at http://groups.google.com/group/parrot-dev/.

IRC

The Parrot IRC channel is hosted on irc.parrot.org and is named `#parrot`. Alternative IRC servers are at irc.pobox.com and irc.rhizomatic.net.

Chapter 2

Overview

The Parrot Interpreter

This document is an introduction to the structure of and the concepts used by the Parrot shared bytecode compiler/interpreter system. We will primarily concern ourselves with the interpreter, since this is the target platform for which all compiler frontends should compile their code.

The Software CPU

Like all interpreter systems of its kind, the Parrot interpreter is a virtual machine; this is another way of saying that it is a software CPU. However, unlike other VMs, the Parrot interpreter is designed to more closely mirror hardware CPUs.

For instance, the Parrot VM will have a register architecture, rather than a stack architecture. It will also have extremely low-level operations, more similar to Java's than the medium-level ops of Perl and Python and the like.

The reasoning for this decision is primarily that by resembling the underlying hardware to some extent, it's possible to compile down Parrot bytecode to efficient native machine language.

Moreover, many programs in high-level languages consist of nested function and method calls, sometimes with lexical variables to hold intermediate results. Under non-JIT settings, a stack-based VM will be popping and then pushing the same operands many times, while a register-based VM will simply allocate the right amount of registers and operate on them, which can significantly reduce the amount of operations and CPU time.

To be more specific about the software CPU, it will contain a large number of registers. The current design provides for four groups of N registers; each group will hold a different data type: integers, floating-point numbers, strings, and PMCs. (Polymorphic Containers, detailed below.)

Registers will be stored in register frames, which can be pushed and popped onto the register stack. For instance, a subroutine or a block might need its own register frame.

The Operations

The Parrot interpreter has a large number of very low level instructions, and it is expected that high-level languages will compile down to a medium-level language before outputting pure Parrot machine code.

Operations will be represented by several bytes of Parrot machine code; the first `INTVAL` will specify the operation number, and the remaining arguments will be operator-specific. Operations will usually be targeted at a specific data type and register type; so, for instance, the `dec_i_c` takes two `INTVAL`s as arguments, and decrements contents of the integer register designated by the first `INTVAL` by the value in the second `INTVAL`. Naturally, operations which act on `FLOATVAL` registers will use `FLOATVAL`s for constants; however, since the first argument is almost always a register **number** rather than actual data, even operations on string and PMC registers will take an `INTVAL` as the first argument.

As in Perl, Parrot ops will return the pointer to the next operation in the bytecode stream. Although ops will have a predetermined number and size of arguments, it's cheaper to have the individual ops skip over their arguments returning the next operation, rather than looking up in a table the number of bytes to skip over for a given opcode.

There will be global and private opcode tables; that is to say, an area of the bytecode can define a set of custom operations that it will use. These areas will roughly map to the subroutines of the original source; each precompiled module will have its own opcode table.

For a closer look at Parrot ops, see *docs/pdds/pdd06_pasm.pod*.

PMCs

PMCs are roughly equivalent to the `SV`, `AV` and `HV` (and more complex types) defined in Perl 5, and almost exactly equivalent to `PythonObject` types in Python. They are a completely abstracted data type; they may be string, integer, code or anything else. As we will see shortly, they can be expected to behave in certain ways when instructed to perform certain operations - such as incrementing by one, converting their value to an integer, and so on.

The fact of their abstraction allows us to treat PMCs as, roughly speaking, a standard API for dealing with data. If we're executing Perl code, we can manufacture PMCs that behave like Perl scalars, and the operations we perform on them will do Perlish things; if we execute Python code, we can manufacture PMCs with Python operations, and the same underlying bytecode will now perform Pythonic activities.

For documentation on the specific PMCs that ship with Parrot, see the *docs/pmc* directory.

Vtables

The way we achieve this abstraction is to assign to each PMC a set of function pointers that determine how it ought to behave when asked to do various things. In a sense, you can regard a PMC as an object in an abstract virtual class; the PMC needs a set of methods to be defined in order to respond to method calls. These sets of methods are called **vtables**.

A vtable is, more strictly speaking, a structure which expects to be filled with function pointers. The PMC contains a pointer to the vtable structure which implements its behavior. Hence, when we ask a PMC for its length, we're essentially calling the `length` method on the PMC; this is implemented by looking up the `length` slot in the vtable that the PMC points to, and calling the resulting function pointer with the PMC as argument: essentially,

```
(pmc->vtable->length)(pmc);
```

If our PMC is a string and has a vtable which implements Perl-like string operations, this will return the length of the string. If, on the other hand, the PMC is an array, we might get back the number of elements in the array. (If that's what we want it to do.)

Similarly, if we call the increment operator on a Perl string, we should get the next string in alphabetic sequence; if we call it on a Python value, we may well get an error to the effect that Python doesn't have an increment operator suggesting a bug in the compiler front-end. Or it might use a "super-compatible Python vtable" doing the right thing anyway to allow sharing data between Python programs and other languages more easily.

At any rate, the point is that vtables allow us to separate out the basic operations common to all programming languages - addition, length, concatenation, and so on - from the specific behavior demanded by individual languages. Perl 6 will be Perl by passing Parrot a set of Perlish vtables; Parrot will equally be able to run Python, Tcl, Ruby or whatever by linking in a set of vtables which implement the behaviors of values in those languages. Combining this with the custom opcode tables mentioned above, you should be able to see how Parrot is essentially a language independent base for building runtimes for bytecompiled languages.

One interesting thing about vtables is that you can construct them dynamically. You can find out more about vtables in *docs/vtables.pod.*

String Handling

Parrot provides a programmer-friendly view of strings. The Parrot string handling subsection handles all the work of memory allocation, expansion, and so on behind the scenes. It also deals with some of the encoding headaches that can plague Unicode-aware languages.

This is done primarily by a similar vtable system to that used by PMCs; each encoding will specify functions such as the maximum number of bytes to allocate for a character, the length of a string in characters, the offset of a given character in a string, and so on. They will, of course, provide a transcoding function either to the other encodings or just to Unicode for use as a pivot.

The string handling API is explained in *docs/strings.pod.*

Bytecode format

We have already explained the format of the main stream of bytecode; operations will be followed by arguments packed in such a format as the individual operations require. This makes up the third section of a Parrot bytecode file; frozen representations of Parrot programs have the following structure.

Firstly, a magic number is presented to identify the bytecode file as Parrot code. Next comes the fixup segment, which contains pointers to global variable storage and other memory locations required by the main opcode segment. On disk, the actual pointers will be zeroed out, and the bytecode loader will replace them by the memory addresses allocated by the running instance of the interpreter.

Similarly, the next segment defines all string and PMC constants used in the code. The loader will reconstruct these constants, fixing references to the constants in the opcode segment with the addresses of the newly reconstructed data.

As we know, the opcode segment is next. This is optionally followed by a code segment for debugging purposes, which contains a munged form of the original program file.

The bytecode format is fully documented in *docs/parrotbyte.pod.*

Chapter 3

Submitting bug reports and patches

ABSTRACT

How to submit bug reports, patches and new files to Parrot.

How To Submit A Bug Report

If you encounter an error while working with Parrot and don't understand what is causing it, create a bug report using the *parrotbug* utility. The simplest way to use it is to run

```
% ./parrotbug
```

in the distribution's root directory, and follow the prompts.

However, if you do know how to fix the problem you encountered, then think about submitting a patch, or (see below) getting commit privileges.

How To Create A Patch

Try to keep your patches specific to a single change, and ensure that your change does not break any tests. Do this by running `make test`. If there is no test for the fixed bug, please provide one.

In the following examples, *parrot* contains the Parrot distribution, and *workingdir* contains *parrot*. The name *workingdir* is just a placeholder for whatever the distribution's parent directory is called on your machine.

```
workingdir
    |
    +--> parrot
            |
            +--> LICENSE
            |
            +--> src
```

```
        |
        +--> tools
        |
        +--> ...
```

`svn`

If you are working with a checked out copy of parrot then please generate your patch with `svn diff`.

```
cd parrot
svn status
svn diff > my_contribution.patch
```

Single `diff`

If you are working from a released distribution of Parrot and the change you wish to make affects only one or two files, then you can supply a `diff` for each file. The `diff` should be created in *parrot*. Please be sure to create a unified diff, with `diff -u`.

```
cd parrot
diff -u docs/submissions.pod docs/submissions.new > submissions.patch
```

Win32 users will probably need to specify `-ub`.

Recursive `diff`

If the change is more wide-ranging, then create an identical copy of *parrot* in *workingdir* and rename it *parrot.new*. Modify *parrot.new* and run a recursive `diff` on the two directories to create your patch. The `diff` should be created in *workingdir*.

```
cd workingdir
diff -ur --exclude='.svn' parrot parrot.new > docs.patch
```

Mac OS X users should also specify `--exclude=.DS_Store`.

`CREDITS`

Each and every patch is an important contribution to Parrot and it's important that these efforts are recognized. To that end, the *CREDITS* file contains an informal list of contributors and their contributions made to Parrot. Patch submitters are encouraged to include a new or updated entry for themselves in *CREDITS* as part of their patch.

The format for entries in *CREDITS* is defined at the top of the file.

How To Submit A Patch

1. Go to Parrot's ticket tracking system at https://trac.parrot.org/parrot/. Log in, or create an account if you don't have one yet.
2. If there is already a ticket for the bug or feature that your patch relates to, just attach the patch directly to the ticket.
3. Otherwise select "New Ticket" at the top of the site. https://trac.parrot.org/parrot/newticket

4. Give a clear and concise Summary. You do **NOT** need to prefix the Summary with a `[PATCH]` identifier. Instead, in the lower-right corner of the *newticket* page, select status `new` in the *Patch status* drop-down box.
5. The Description should contain an explanation of the purpose of the patch, and a list of all files affected with summary of the changes made in each file. Optionally, the output of the `diffstat(1)` utility when run on your patch(s) may be included at the bottom of the message body.
6. Set the Type of the ticket to "patch". Set other relevant drop-down menus, such as Version (the version of Parrot where you encountered the problem), Platform, or Severity. As mentioned above, select status `new` in the *Patch status* drop-down box.
7. Check the box for "I have files to attach to this ticket". Double-check that you've actually done this, because it's easy to forget.

   DO NOT paste the patch file content into the Description.
8. Click the "Create ticket" button. On the next page attach your patch file(s).

Applying Patches

You may wish to apply a patch submitted by someone else before the patch is incorporated into SVN.

For single `diff` patches or `svn` patches, copy the patch file to *parrot*, and run:

```
cd parrot
patch -p0 < some.patch
```

For recursive `diff` patches, copy the patch file to *workingdir*, and run:

```
cd workingdir
patch -p0 < some.patch
```

In order to be on the safe side run 'make test' before actually committing the changes.

Configuration of files to ignore

Sometimes new files will be created in the configuration and build process of Parrot. These files should not show up when checking the distribution with

```
svn status
```

or

```
perl tools/dev/manicheck.pl
```

The list of these ignore files can be set up with:

```
svn propedit svn:ignore <PATH>
```

In order to keep the two different checks synchronized, the MANIFEST and MANIFEST.SKIP file should be regenerated with:

```
perl tools/dev/mk_manifest_and_skip.pl
```

How To Submit Something New

If you have a new feature to add to Parrot, such as a new test.

1. Add your new file path(s), relative to *parrot*, to the file MANIFEST. Create a patch for the MANIFEST file according to the instructions in **How To Submit A Patch**.
2. If you have a new test script ending in `.t`, some mailers may become confused and consider it an application/x-troff. One way around this (for *nix users) is to diff the file against /dev/null like this:
   ```
   cd parrot
   diff -u /dev/null newfile.t > newfile.patch
   ```
3. Go to Parrot's ticket tracking system at https://trac.parrot.org/parrot/. Log in, or create an account if you don't have one yet.
4. Select "New Ticket" https://trac.parrot.org/parrot/newticket.
5. Give a clear and concise Summary.

   Prefix it with a `[NEW]` identifier.
6. The Description should contain an explanation of the purpose of the feature you are adding. Optionally, include the output of the `diffstat(1)` utility when run on your patch(es).
7. Set the Type of the ticket to "patch". Set other relevant drop-down menus, such as Version, Platform, or Severity.
8. Check the box for "I have files to attach to this ticket"

   Double-check that you've actually done this, because it's easy to forget.

   DO NOT paste the content of the new file or files into the body of the message.
9. Click the "Create ticket" button. On the next page attach the patch for MANIFEST and your new file(s).

What Happens Next?

If you created a new ticket for the submission, you will be taken to the page for the new ticket and can check on the progress of your submission there. This identifier should be used in all correspondence concerning the submission.

Everyone on Trac sees the submission and can comment on it. A developer with SVN commit authority can commit it to SVN once it is clear that it is the right thing to do.

However developers with SVN commit authority may not commit your changes immediately if they are large or complex, as we need time for peer review.

A list of active tickets can be found here: http://trac.parrot.org/parrot/report/1

A list of all the unresolved patches is at: http://trac.parrot.org/parrot/report/15

Patches for the Parrot website

The http://www.parrot.org website is hosted in a Drupal CMS. Submit changes through the usual ticket interface in Trac.

Getting Commit Privileges

If you are interested in getting commit privileges to Parrot, here is the procedure:

1. Submit several high quality patches (and have them committed) via the process described in this document. This process may take weeks or months.
2. Obtain a Trac account at https://trac.parrot.org/parrot
3. Submit a Parrot Contributor License Agreement; this document signifies that you have the authority to license your work to Parrot Foundation for inclusion in their projects. You may need to discuss this with your employer if you contribute to Parrot on work time or with work resources, or depending on your employment agreement.

   http://www.parrot.org/files/parrot_cla.pdf
4. Request commit access via the `parrot-dev` mailing list, or via IRC (#parrot on irc.parrot.org). The existing committers will discuss your request in the next couple of weeks.

   If approved, a metacommitter will update the permissions to allow you to commit to Parrot; see `RESPONSIBLE_PARTIES` for the current list. Welcome aboard!

Thanks for your help!