

**progargs**

|                      |
|----------------------|
| <b>COLLABORATORS</b> |
|----------------------|

|               |                            |                |                  |
|---------------|----------------------------|----------------|------------------|
|               | <i>TITLE :</i><br>progargs |                |                  |
| <i>ACTION</i> | <i>NAME</i>                | <i>DATE</i>    | <i>SIGNATURE</i> |
| WRITTEN BY    |                            | March 28, 2025 |                  |

|                         |
|-------------------------|
| <b>REVISION HISTORY</b> |
|-------------------------|

|        |      |             |      |
|--------|------|-------------|------|
| NUMBER | DATE | DESCRIPTION | NAME |
|        |      |             |      |

# Contents

|          |  |          |
|----------|--|----------|
| <b>1</b> | <b>progargs</b>                              | <b>1</b> |
| 1.1      | ProgArgs.doc . . . . .                       | 1        |
| 1.2      | progargs.library/AddProgArgsA . . . . .      | 1        |
| 1.3      | progargs.library/AllocProgArgsA . . . . .    | 3        |
| 1.4      | progargs.library/ExecProgArgsA . . . . .     | 6        |
| 1.5      | progargs.library/FreeProgArgs . . . . .      | 12       |
| 1.6      | progargs.library/GetProgArgsA . . . . .      | 13       |
| 1.7      | progargs.library/PA_AddTagArgsA . . . . .    | 14       |
| 1.8      | progargs.library/PA_AllocPooled . . . . .    | 15       |
| 1.9      | progargs.library/PA_AllocTagArgsA . . . . .  | 15       |
| 1.10     | progargs.library/PA_AllocVecPooled . . . . . | 17       |
| 1.11     | progargs.library/PA_CreatePool . . . . .     | 17       |
| 1.12     | progargs.library/PA_DeletePool . . . . .     | 18       |
| 1.13     | progargs.library/PA_FreePooled . . . . .     | 19       |
| 1.14     | progargs.library/PA_FreeTagArgs . . . . .    | 19       |
| 1.15     | progargs.library/PA_FreeVecPooled . . . . .  | 20       |
| 1.16     | progargs.library/PA_GetArgEntry . . . . .    | 21       |
| 1.17     | progargs.library/PA_NextArgEntry . . . . .   | 22       |
| 1.18     | progargs.library/PA_RemTagArgsA . . . . .    | 23       |
| 1.19     | progargs.library/RemProgArgsA . . . . .      | 23       |

---

# Chapter 1

## progargs

### 1.1 ProgArgs.doc

```
AddProgArgsA()  
AllocProgArgsA()  
ExecProgArgsA()  
FreeProgArgs()  
GetProgArgsA()  
PA_AddTagArgsA()  
PA_AllocPooled()  
PA_AllocTagArgsA()  
PA_AllocVecPooled()  
PA_CreatePool()  
PA_DeletePool()  
PA_FreePooled()  
PA_FreeTagArgs()  
PA_FreeVecPooled()  
PA_GetArgEntry()  
PA_NextArgEntry()  
PA_RemTagArgsA()  
RemProgArgsA()
```

### 1.2 progargs.library/AddProgArgsA

11 Dec 1994 20:01:43

#### NAME

AddProgArgsA -- add/modify values in an argument set.  
AddProgArgs -- varargs stub for AddProgArgsA().

#### SYNOPSIS

```
result = AddProgArgsA( progArgs, argTags )  
D0                      A0          A1
```

```
BOOL AddProgArgsA( struct ProgArgs *, struct TagItem * )
```

```
result = AddProgArgsA( progArgs, arg1Tag, ... )
```

```
BOOL AddProgArgsA( struct ProgArgs *, Tag, ... )
```

---

## FUNCTION

This allows a program to modify the individual values in an argument set. Note that these are the values initially supplied as the entries to `AllocProgArgsA()`; this function does NOT allow a program to add completely new arguments.

The exact behavior of this function can be controlled by various control tags supplied in the argument tag list. This includes whether to interpret the supplied tag values in a `ReadArgs()` style [integers are pointers to integers, multi arguments are an array of string pointers], and whether to copy or just refer to the data.

These control tags can occur multiple times in the argument array, and effect only the interpretation of the tags which follow them.

## INPUTS

`progArgs` - ( `struct ProgArgs *` ) Argument set to modify.  
`argTags` - ( `struct TagItem *` ) Array of a { `ArgID`, `Value` } tags.

## RESULT

`result` - ( `BOOL` ) `TRUE` if all okay; `FALSE` if there was a problem.

## TAGS

`PAC_RDArgsMode` -- ( `BOOL` )  
 Argument parameters are in `ReadArgs()` style.  
 Default: `FALSE`.

If `TRUE`, the arguments following this tag will be parsed as `ReadArgs()` arguments [i.e., a `/N` is a pointer to a `LONG` rather than an actual `LONG`.] If `FALSE`, the standard Workbench style will be used.

The following argument types are currently recognized

| Template        | <code>ReadArgs()</code>          | Workbench  |
|-----------------|----------------------------------|--|
| default         | <code>UBYTE *</code>             | <code>UBYTE *</code>   |
| <code>/N</code> | <code>LONG *</code>              | <code>LONG</code>  |
| <code>/M</code> | <code>UBYTE **, NULL term</code> | <code>struct MultiArg *, term with</code><br><code>NULL ma_String</code> |

`PAC_CopyMode` -- ( `BOOL` )  
 Copy supplied arguments.  
 Default: `FALSE`.

If `TRUE`, the arguments following this tag will be copied into the argument list, rather than just passed by reference. If an argument you are supplying [for example, a string from a string gadget] may not be valid for the existence of this argument, you **MUST** tell the library to make a copy of it.

## EXAMPLE

## NOTES

```

.

BUGS
.

SEE ALSO
    AllocProgArgsA(), RemProgArgsA(), libraries/progargs.h

```

## 1.3 progargs.library/AllocProgArgsA

11 Dec 1994 19:57:41

### NAME

```

AllocProgArgsA -- allocate a new argument set.
AllocProgArgs -- varargs stub for AllocProgArgsA().

```

### SYNOPSIS

```

progArgs = AllocProgArgsA( entries, cmdTags )
D0                      A0      A1

struct ProgArgs * AllocProgArgsA( struct ArgEntry *,
                                   struct TagItem * )

progArgs = AllocProgArgsA( entries, cmdlTag, ... )

struct ProgArgs * AllocProgArgsA( struct ArgEntry *, Tag, ... )

```

### FUNCTION

This fuction allocates a new argument set, based on the argument templates supplied by 'entries'. The cmdTags are then sent to ExecProgArgs() for any initial operations which should be performed.

Argument entries are basically an extensions to the DOS ReadArgs() templates; each entry contains a similar template for that -single- argument, along with additional data such as a default value and description. For example:

```

#define FILES_ARG (TAG_ARGENTRY+1)

struct MultiArg DefFiles[] = {
    { "#?", NULL, NULL },
    { NULL }
};

struct ArgEntry myEntries[] = {
    { FILES_ARG, "FILES/A/M", "Files to process",
      AEFLAG_DEFAULT, &DefFiles[0], NULL },
    { TAG_END }
};

```

Creates an argument set with one argument - FILES - which must always be present and has multiple values. You can later refer to this argument through AddProgArgsA(), RemProgArgsA() and GetProgArgsA() through the ID FILES\_ARG which was assigned to it. If no explicit value exists for the argument, GetProgArgsA() will return the default of a single file "#?".

In addition to the array of ArgEntry structures which describe your argument set, you may optionally supply a TagList of commands to initially execute; if any of these commands fail, the entire allocation will abort and return NULL. See ExecProgArgsA() for more details on what commands are available.

Argument ID numbers MUST be in the range from TAG\_USER+0x10000 to TAG\_USER+0x1FFFF; other ID numbers are reserved for control tags and future extensions.

Default values for arguments are supplied by setting AEFLAG\_DEFAULT in ArgEntry.ae\_Flags, and providing the appropriate type [based on the template] in ae\_Default. See below for a description of what these types are.

Additional description of what an argument is can be supplied by pointing ae\_Description to a string of text; this text will be written as a comment to a text file or icon for any undefined arguments.

For upwards compatibility, templates can only contain the uppercase letters 'A' to 'Z', '=', and '/'. Keyword synonyms can be defined using '=' as with ReadArgs(), i.e. "FILES=FL/A/M" would result in 'FL' being equivalent to 'FILES'. When writing arguments out to disk, the first keyword will always be used.

In order to simplify the interface, Workbench ToolTypes and CLI ReadArgs() values are massaged into a standard set of data types. The complete set of switches which an argument template may have and the data types supported are:

```
/A: Always
    CLI and tooltype: This argument must be supplied, or a failure.
    This is not currently supported very well for ToolTypes and
    text files.
/K: Keyword
    CLI: The keyword must be supplied before its value.
    Tooltype: No effect.
/F: Finish line
    CLI: The argument blindly takes the rest of the line.
    Tooltype: No effect.
/N: Numeric
    CLI and tooltype: The result is a signed long rather than a
    pointer to a string. Inputs can be a simple decimal string, a
    hexadecimal string of the form $nnn or 0xnnn, or a binary string
    of the form %nnn. Setting the flag AEFLAG_HEXNUM will cause the
    number to be written out as a hexadecimal string rather than the
    default decimal format.
/S: Switch
    CLI: Set to TRUE if keyword is supplied on command line.
    Tooltype: Set to TRUE if value is 'yes' 'true' '1' etc.
               Set to FALSE if value is 'no' 'false' '1' etc.
/T: Toggle
    CLI: Toggles boolean value.
    Tooltype: Set to TRUE if value is 'yes' 'true' '1' etc.
               Set to FALSE if value is 'no' 'false' '1' etc.
```

---

/M: Multiple strings  
 CLI: Result is a pointer to array of MultiArg structures, terminated by a NULL ma\_String. ma\_Directory is the program's current directory.  
 Tooltype: Result is a pointer to an array of MultiArg structures, filled in from the workbench arguments.  
 Writing multi-args to tooltypes or a text file is not currently supported.

By default, an argument is interpreted as a pointer to a NULL-terminated string.

Extended template switches:  
 NOT IMPLEMENTED YET!

^A: Program arguments  
 This marks a /M type as the main program arguments -- ie, the multiple arguments collected by the command line and the arguments passed by Workbench.  
 ^I: This argument is parsed for input only  
 ^O: This argument is parsed for output only  
 These two flags control the read/write (respectively) behavior of an argument. If neither or both are supplied, the argument is read/write. Otherwise it is either read- or write-only.  
 ^C: CLI argument  
 ^T: Text file argument  
 ^W: Workbench tooltype  
 These control in which context an argument is valid. If none if them are supplied, it is valid in all. Otherwise, it is only valid in the ones supplied. For example, to have an argument be parsed when reading the CLI command line or a text file but not in Workbench tool types, use 'Key^C^T'.

#### INPUTS

entries - ( struct ArgEntry \* ) Argument templates.  
 cmdTags - ( struct TagItem \* ) Initial operations.

#### RESULT

progArgs - ( struct ProgArgs \* ) A new argument set or NULL.

#### TAGS

#### EXAMPLE

A typical call to allocate an argument set and get its default values, based on the above "FROM" ArgEntry, is:

```
struct ProgArgs * myArgs;

myArgs = AllocProgArgs(&myEntries[0],
    PA_WBStartup,      _WBenchMsg, /* Hand in WB startup */
    RPA_WBArguments,   FILES_ARG,  /* Parse WB startup */
    PA_IgnoreError,    TRUE,        /* Don't fail on next err */
    RPA_ProgIcon,      !_WBenchMsg,/* If no WB startup, read */
                                /* directly from icon */
    PA_IgnoreError,    FALSE,       /* Allow fail on next err */
```



```

        RPA_CmdInput,      TRUE,      /* Parse command line */
        TAG_END);

    if(!myArgs) exit(20);

NOTES
    .

BUGS
    .

SEE ALSO
    ExecProgArgsA(), FreeProgArgs(), AddProgArgsA(), RemProgArgsA()

```

## 1.4 progargs.library/ExecProgArgsA

11 Dec 1994 19:39:07

### NAME

ExecProgArgsA -- perform operations on a group of arguments.  
 ExecProgArgs -- varargs stub for ExecProgArgsA().

### SYNOPSIS

```

    result = ExecProgArgsA( progArgs, cmdTags )
    D0                      A0          A1

```

```

    BOOL ExecProgArgsA( struct ProgArgs *, struct TagItem * )

```

```

    result = ExecProgArgsA( progArgs, cmdlTag, ... )

```

```

    BOOL ExecProgArgsA( struct ProgArgs *, Tag, ... )

```

### FUNCTION

Performs a set of operations on a previously allocated program arguments set. This includes reading and writing CLI, Workbench and textfile arguments. Just what operations are performed -- whether they read or write arguments, etc -- depends on the supplied command tags.

Operations are available for reading arguments from the command line, Workbench, icons, and text files; there are operations for writing arguments to icons and text files.

The library provides a number of small features to fully support ToolTypes. Strings longer than about 100 characters will be split into multiple tool types; the first with the regular "keyword=str" form, and the following entries with the form "keyword.Xn=str", where 'n' is the sequential number of this part of the argument. The library will then automatically join all of these entries together when reading the tool types back in. This essentially allows strings to be of any arbitrary length.

In addition, non-printable characters within the string are escaped with the convention \xnn, where 'nn' is the hex value of the character, and the escape character itself is represented as '\'.

The tags are sequentially parsed in the order that they appear. This means that any attribute tag, such as PA\_IgnoreError, will apply only to the operations which follow it, and NOT any of the tags prior to where it is encountered in the tag list. Multiple attribute tags may occur in the tag list; later ones will override the values from earlier ones. The order that tags are processed in is the same as that used by the utility.library/NextTagItem() function.

#### INPUTS

progArgs - ( struct ProgArgs \* ) Previously allocated argument set.  
cmdTags - ( struct TagItem \* ) Operations to perform.

#### RESULT

result - ( BOOL ) TRUE if there were no errors, else FALSE.

#### TAGS

Global control tags:  
~~~~~

PA\_CLISTartup -- ( UBYTE \* )  
Supply the program's CLI arguments.  
Default: NULL

This tag supplies the command line given by the CLI. If NULL, the default of GetCmdStr() is used; otherwise, it will be set to the string this tag supplies.

PA\_WBStartup -- ( struct WBStartup \* )  
Supply the program's Workbench startup message.  
Default: NULL

This tag supplies the Workbench startup message given to the program, which will be used to determine where the program's .info file is for reading ToolTypes and Workbench arguments. This tag essentially sets what will be later used for PROGDIR: and the program's name from the startup message. If NULL, the defaults of the actual lock on PROGDIR: and the program name retrieved from DOS will be used.

PA\_SetDirectory -- ( BPTR ) Lock  
Set directory to read/write files from.  
Default: PADIR\_PROGRAM

When this tag is encountered, the directory which later filenames are relative to is set to ti\_Data. Initially, this directory is the directory which the program exists in, aka PROGDIR:

A few magic values are defined for this tag; these are used to select standard directories:

PADIR\_PROGRAM -- Program directory, PROGDIR:  
PADIR\_CURRENT -- Process's current directory  
PADIR\_TEMP -- T:  
PADIR\_ENV -- ENV:

```
PADIR_ENVARC  -- ENV *and* ENVARC:
                ENV: is first written, then the resulting
                file is copied directly to ENVARC:
PADIR_PREFS   -- PREFS:
PADIR_PRESETS -- PREFS:Presets/
                The "Presets" directory will be created if
                it does not already exist.
```

Note that changing to T, ENV, ENVARC, PREFS or PRESETS may fail if a lock can not be obtained on that directory. If this happens [and PA\_IgnoreError is TRUE], the library will fall back to PADIR\_CURRENT.

Changes in this tag's value are not remembered between calls to ExecProgArgsA() and AllocProgArgsA().

```
PA_EscapeChar -- ( UBYTE )
Set escape character for strings.
Default: '\'
```

Sets the character used to escape non-printable codes when reading and writing text files and ToolTypes. The escape sequence is formatted as ExNN, where E is the escape character, x is the literal char x, and NN is the two digit hex value of the desired character.

Setting the escape character to 0 inhibits all escape substitution and parsing.

```
PA_IgnoreError -- ( BOOL )
Set handling of error conditions.
Default: FALSE
```

If TRUE, commands which cause errors will be skipped, rather than terminating the execution of all the following commands. This is useful for reading a text file which may not exist -- if this tag is not set to TRUE, then the failed read attempt will abort without executing any of the command after it.

Changes in this tag's value are not remembered between calls to ExecProgArgsA() and AllocProgArgsA().

```
PA_CreateIcon -- ( BOOL )
Set creation of program icon if doesn't exit.
Default: TRUE
```

If TRUE, when writing out tooltypes, the library will create an icon for the program if one doesn't already exist. Otherwise, if no icon already exists an error will be returned. This essentially determines whether GetDiskObject() or GetDiskObjectNew() is called when reading the old tooltypes.

This only effects the writing of icons; reading a non-existent icon will cause an error no matter how this flag is set.

---

Changes in this tag's value are not remembered between calls to ExecProgArgsA() and AllocProgArgsA().

PA\_CreateDirs -- ( BOOL )

Set automatic creation of file directories.

Default: FALSE

If TRUE, when writing out a textfile or an explicitly name diskobject [WPA\_TextFileName/WPA\_DiskObjectName], the library will create any subdirectories in the filename path if they don't already exist. This is useful when writing a file to ENV: which is in a subdirectory, i.e. ENV:myProgram/myPrefs.tt". If 'myProgram' doesn't already exist in ENV: and this tag is TRUE, it will be created for you. This works for any number of sub-directories in the path which need to be created.

Changes in this tag's value are not remembered between calls to ExecProgArgsA() and AllocProgArgsA().

PA\_AllComments -- ( BOOL )

Set writing of all arguments as comments.

Default: FALSE

If TRUE, when writing arguments only argument comments and no values are written. This can be used to write the "default" values out to an icon or file -- all of the arguments have no value, so the program's default values are used when they are next read in.

Changes in this tag's value are not remembered between calls to ExecProgArgsA() and AllocProgArgsA().

PA\_ForceWrite -- ( BOOL )

Set writing of all arguments in the set.

Default: TRUE

If TRUE, all entries in the argument will be written, no matter what they will overwrite. If FALSE, an argument will not be written if its current value is undefined, but it is defined in the file being overwritten.

Changes in this tag's value are not remembered between calls to ExecProgArgsA() and AllocProgArgsA().

PA\_TransHook -- ( struct Hook \* )

Set hook to localize strings.

Default: NULL

This tag supplies the address of a hook through which the library will translate various strings. This is primarily intended for localization support -- the supplied hook should in some way look up the string in its locale catalog.

---

The Hook should be a standard OS hook structure. The library will call this hook with a 'PA\_StrTransMsg' sent as the message and either the calling ProgArgs structure or a value specified with the PA\_TransObject (below) as the object. The callee should be sure to check the stm\_Message is STM\_STRING and ignore all other messages. The value returned by the hook must ALWAYS be either a pointer to a valid string or NULL.

Note that if this tag is supplied, ae\_Description should be a string ID code rather than a literal string.

The library currently only sends translation messages for ArgEntry.ae\_Description, and the translation is performed at the time that the description be being referenced, NOT during initialization in AllocProgArgsA().

PA\_TransObject -- ( void \* )

Set parameter to pass as 'object' to PA\_TransHook.

Default: Address of this ProgArgs.

This tag supplies an alternate 'object' to send to the above translation hook. This may, for example, be a pointer to a block containing all the program's strings which will be searched.

Argument reading tags:

~~~~~

RPA\_CmdInput -- ( BOOL )

Do ReadArgs() style parsing of command input.

If TRUE, the program's next command line will be parsed. This is the typical way to get the program's CLI input - the first occurrence of this tag will retrieve the program's command line. Successive use of this tag will then wait for the user to input another command string. Normal DOS interaction occurs for each parse, i.e. prompting for input when the user supplies '?'.  
  
If running under Workbench, no action occurs.

RPA\_CmdLine -- ( BOOL )

Do ReadArgs() style parsing of command line.

If TRUE, the program's command line (ie, GetCmdStr() or the custom command line supplied by PA\_CLIStartup) will be parsed. NO user interaction occurs, so this is generally not the way you should initially parse the command line.

RPA\_CmdString -- ( UBYTE \* )

Do ReadArgs() style parsing of string.

This tag supplies a string which will be parsed through the standard DOS ReadArgs() call. No prompts are generated for additional information. As with ReadArgs(), this string should end with a \n.

RPA\_WBArguments -- ( Tag )

Extract arguments from program's Workbench startup message.

This tag uses the Workbench startup message previously supplied with PA\_WBStartup, and places them into the given argument tag. The value put in the argument is an array of MultiArg structures, representing the Workbench arguments. This means that the argument they are put in should be of type /M.

RPA\_ProgIcon -- ( BOOL )

Extract arguments from program's icon.

This tag, if TRUE, will cause the program's .info file to be examined for relevant tool types. This is much like PPA\_WBStartup except that no Workbench startup arguments are parsed, and this can also be used by programs started through the CLI. It is typically used by a program to extract any tool types from its icon to set up defaults before parsing the command line.

Note that this only opens an icon file with the program's name; you must be sure to be in PADIR\_PROGRAM to get your program's actual icon file.

RPA\_DiskObjectName -- ( BOOL )

Extract arguments from an arbitrary DiskObject.

This tag supplies the name of an icon from which to extract its tool types. The name is relative to the current directory that has been selected, and should not include the trailing info'.

RPA\_TextFileName -- ( BOOL )

Extract arguments from a text file.

This tag supplies the name of a file from which to read the program's arguments. This file is formatted much like a tooltypes array, one argument per line.

Argument writing tags:

~~~~~

WPA\_ProgIcon -- ( BOOL )

Write tooltypes out to program icon.

When encountered the program arguments are written out to the program's .info file in the tool type array. This works for both CLI and Workbench started programs.

Note that this only opens an icon file with the program's name; you must be sure to be in PADIR\_PROGRAM to write your program's actual icon file.

WPA\_DiskObjectName -- ( UBYTE \* )

Name of file to write disk object to.

Supplies the name of the file to write the DiskObject out as, without the trailing '.info'. This can be used to write out somewhere besides your program's .info file.

WPA\_TextFileName -- ( UBYTE \* )

Name of file to write arguments out to as text.

Supplies the name of a file to write the arguments out to, as a standard ASCII file.

#### EXAMPLE

```
/* Read arguments from the file "ENV:myAppPrefs" */

result = ExecProgArgs(myArgs,
    PA_SetDirectory,    PADIR_ENV,        /* Look in ENV: */
    PA_IgnoreError,    TRUE,              /* Just use defaults */
                                          /* if it doesn't exist. */
    RPA_TextFileName,   "myAppPrefs",    /* Read the file */
    TAG_END);

if(!result) exit(20);

/* Write file back to ENV: and ENVARC: */

result = ExecProgArgs(myArgs,
    PA_SetDirectory,    PADIR_ENVARC,    /* Write ENV: & ENVARC: */
    WPA_TextFileName,   "myAppPrefs",    /* Write the files */
    TAG_END);

if(!result) exit(20);
```

#### NOTES

.

#### BUGS

.

#### SEE ALSO

AllocProgArgsA()

## 1.5 progargs.library/FreeProgArgs

11 Dec 1994 19:56:14

#### NAME

FreeProgArgs -- deallocate a previously allocated argument set.

#### SYNOPSIS

```
FreeProgArgs( progArgs )
            A0
```

```
void FreeProgArgs( struct ProgArgs * )
```

## FUNCTION

Deallocates all memory associated with an argument set previously allocated with AllocProgArgs().

## INPUTS

progArgs - ( struct ProgArgs \* ) The argument set to free.

## RESULT

nothing.

## TAGS

## EXAMPLE

## NOTES

.

## BUGS

.

## SEE ALSO

AllocProgArgs()

## 1.6 progargs.library/GetProgArgsA

22 Dec 1994 04:52:32

## NAME

GetProgArgsA -- get values from an argument set.  
GetProgArgs -- varargs stub for GetProgArgsA().

## SYNOPSIS

```
result = GetProgArgsA( progArgs, argTags )
D0                                A0      A1
```

```
BOOL GetProgArgsA( struct ProgArgs *, struct TagItem * )
```

```
result = GetProgArgsA( progArgs, arg1Tag, ... )
```

```
BOOL GetProgArgsA( struct ProgArgs *, Tag, ... )
```

## FUNCTION

Extracts the requested arguments from the given argument set. The tag list should be a standard array of { TagValue, PutAddress } tag pairs, where TagValue is an argument ID being requested and PutAddress is the address of a ULONG in memory in which to place the value. If an explicit value for the argument is not found in the argument list, the argument's default value (if any) is returned.

If any of the requested arguments have no explicit or default value an result of FALSE will be returned; if all values are extracted okay, TRUE is returned.

## INPUTS



progArgs - ( struct ProgArgs \* ) Argument set to query.  
 argTags - ( struct TagItem \* ) Arguments to retrieve.

#### RESULT

result - ( BOOL ) TRUE if all is okay; FALSE if any argument was not found.

#### TAGS

#### EXAMPLE

#### NOTES

.

#### BUGS

.

#### SEE ALSO

AllocProgArgsA(), AddProgArgsA()

## 1.7 progargs.library/PA\_AddTagArgsA

11 Dec 1994 20:21:21

#### NAME

PA\_AddTagArgsA -- add tags to an argument array.  
 PA\_AddTagArgs -- varargs stub for PA\_AddTagArgsA().

#### SYNOPSIS

```
result = PA_AddTagArgsA( base, args )
D0                                A0    A1
```

```
BOOL PA_AddTagArgsA( struct TagArgs *, struct TagItem * )
```

```
result = PA_AddTagArgsA( base, arglType, ... )
```

```
BOOL PA_AddTagArgsA( struct TagArgs *, Tag, ... )
```

#### FUNCTION

Adds the given tag values to the argument array. If these tags did not exist, new ones are created in the array. If they did exist, their values are changed to the new value in ti\_Data.

#### INPUTS

base - ( struct TagArgs \* ) Argument array to modify.  
 args - ( struct TagItem \* ) Argument values to set.

#### RESULT

result - ( BOOL ) TRUE if all is okay; FALSE if failure (ie memory).

#### TAGS

#### EXAMPLE

---

## NOTES

.

## BUGS

.

## SEE ALSO

PA\_AllocTagArgsA(), PA\_RemTagArgsA()

## 1.8 progargs.library/PA\_AllocPooled

11 Dec 1994 19:03:14

## NAME

PA\_AllocPooled -- allocate a memory block from a pool.

## SYNOPSIS

```
memory = PA_AllocPooled( poolHeader, memSize )
D0                      A0          D0
```

```
APTR PA_AllocPooled( APTR, unsigned long )
```

## FUNCTION

Allocates a new memory block of the given size from a memory pool.

## INPUTS

poolHeader - ( APTR ) Previously created pool.  
memSize - ( unsigned long ) Amount of memory to allocate.

## RESULT

memory - ( APTR ) The new block of memory or NULL.

## TAGS

## EXAMPLE

## NOTES

This function is simply an interface to the amiga.lib pool functions which the library uses internally. See AllocPooled() for details.

## BUGS

.

## SEE ALSO

AllocPooled(), LibAllocPooled(),  
PA\_CreatePool(), PA\_DeletePool(), PA\_FreePooled()

## 1.9 progargs.library/PA\_AllocTagArgsA

11 Dec 1994 20:26:59

## NAME

PA\_AllocTagArgsA -- create a brand new argument array.  
PA\_AllocTagArgs -- varargs stub for PA\_AllocTagArgsA().

## SYNOPSIS

```
base = PA_AllocTagArgsA( controlTags)
D0                                A0

struct TagArgs * PA_AllocTagArgsA( struct TagItem * )

base = PA_AllocTagArgsA( controll1Tag, ... )

struct TagArgs * PA_AllocTagArgsA( Tag, ... )
```

## FUNCTION

Creates a new 'TagArgs' structure. This is a dynamic form of a utility.library tag list; there are functions which can be used to dynamically add, modify and remove tags from the array.

The tag list created by an argument array has two important properties:

1. Only one instance of any unique tag ID can exist in the array. This means that calling PA\_AddTagArgsA() for a tag ID which already exists in the array will simply modify its value, rather than adding a new tag.
2. The tag list is unordered. A program can not depend on the tag IDs it puts into an argument array to result in them being in any particular order when it later looks at the resulting tag list.

## INPUTS

controlTags - ( struct TagItem \* ) Control overall behavior;  
currently ignored.

## RESULT

base - ( struct TagArgs \* ) The argument array base structure.

## TAGS

## EXAMPLE

## NOTES

.

## BUGS

.

## SEE ALSO

PA\_FreeTagArgs(), PA\_AddTagArgsA(), PA\_RemTagArgsA()

## 1.10 progargs.library/PA\_AllocVecPooled

11 Dec 1994 19:16:35

### NAME

PA\_AllocVecPooled -- allocate memory from pool and remember size.

### SYNOPSIS

```
memory = PA_AllocVecPooled( poolHeader, memSize )
D0                      A0          D0
```

```
APTR PA_AllocVecPooled( APTR, ULONG )
```

### FUNCTION

Allocates a memory block from a pool, and remembers the size of the block. This function is identical to PA\_AllocPooled(), except the size of the block is remembered for when it is later freed. PA\_FreeVecPooled() must be used to free the block.

### INPUTS

```
poolHeader - ( APTR ) The pool to allocate in.
memSize    - ( ULONG ) Size of the block to allocate.
```

### RESULT

```
memory - ( APTR ) A new block of memory, or NULL.
```

### TAGS

### EXAMPLE

### NOTES

.

### BUGS

.

### SEE ALSO

```
PA_FreeVecPooled(), PA_CreatePool(), PA_DeletePool()
```

## 1.11 progargs.library/PA\_CreatePool

11 Dec 1994 19:09:14

### NAME

PA\_CreatePool -- create a new memory pool.

### SYNOPSIS

```
pool = PA_CreatePool( memFlags, puddleSize, threshSize )
D0                      D0          D1          D2
```

```
APTR PA_CreatePool( unsigned long, unsigned long, unsigned long )
```

### FUNCTION

Creates a new memory pool, which can then have Alloc/Free operations

performed on it.

#### INPUTS

memFlags - ( unsigned long ) Memory type of this pool.  
puddleSize - ( unsigned long ) Bytes in each puddle.  
threshSize - ( unsigned long ) Block size to go out of puddle.

#### RESULT

pool - ( APTR ) A newly created pool for use, or NULL.

#### TAGS

#### EXAMPLE

#### NOTES

This function is simply an interface to the amiga.lib pool functions which the library uses internally. See CreatePool() for details.

#### BUGS

.

#### SEE ALSO

CreatePool(), LibCreatePool(),  
PA\_AllocPooled(), PA\_DeletePool(), PA\_FreePooled()

## 1.12 progargs.library/PA\_DeletePool

11 Dec 1994 19:12:06

#### NAME

PA\_DeletePool -- deallocate a previously created and used pool.

#### SYNOPSIS

PA\_DeletePool( poolHeader )  
A0

void PA\_DeletePool( APTR )

#### FUNCTION

Frees all memory associated with the given pool.

#### INPUTS

poolHeader - ( APTR ) A previously created pool.

#### RESULT

nothing.

#### TAGS

#### EXAMPLE

#### NOTES

---

This function is simply an interface to the amiga.lib pool functions which the library uses internally. See DeletePool() for details.

**BUGS**

.

**SEE ALSO**

DeletePool(), LibDeletePool(),  
PA\_AllocPooled(), PA\_CreatePool(), PA\_FreePooled()

## 1.13 progargs.library/PA\_FreePooled

11 Dec 1994 19:14:10

**NAME**

PA\_FreePooled -- free a memory block in a pool.

**SYNOPSIS**

PA\_FreePooled( poolHeader, memory, memSize )  
D0                    A0                    A1                    D0

void PA\_FreePooled( APTR, APTR, unsigned long )

**FUNCTION**

Frees a memory block previously allocated in the given pool.

**INPUTS**

poolHeader - ( APTR ) Pool memory block is in.  
memory     - ( APTR ) The memory being freed.  
memSize    - ( unsigned long ) Size of the block.

**RESULT**

nothing.

**TAGS****EXAMPLE****NOTES**

This function is simply an interface to the amiga.lib pool functions which the library uses internally. See FreePooled() for details.

**BUGS**

.

**SEE ALSO**

FreePooled(), LibFreePooled(),  
PA\_AllocPooled(), PA\_CreatePool(), PA\_DeletePool()

## 1.14 progargs.library/PA\_FreeTagArgs

11 Dec 1994 20:25:57

## NAME

PA\_FreeTagArgs -- free a previously allocated argument array.

## SYNOPSIS

PA\_FreeTagArgs( base)  
A0

void PA\_FreeTagArgs( struct TagArgs \* )

## FUNCTION

Frees all memory used by the given argument array.

## INPUTS

base - ( struct TagArgs \* ) The argument array to free.

## RESULT

nothing.

## TAGS

## EXAMPLE

## NOTES

.

## BUGS

.

## SEE ALSO

PA\_AllocTagArgsA()

## 1.15 progargs.library/PA\_FreeVecPooled

11 Dec 1994 19:31:37

## NAME

PA\_FreeVecPooled -- free a pool memory block, extracting block size.

## SYNOPSIS

PA\_FreeVecPooled( poolHeader, memory )  
A0 A1

void PA\_FreeVecPooled( APTR, APTR )

## FUNCTION

Frees the given memory block from the pool. The memory block size given to PA\_AllocVecPooled() is the size of the freed memory block

## INPUTS

poolHeader - ( APTR ) The pool the block is in.  
memory - ( APTR ) The memory block to free.

RESULT  
nothing.

TAGS

EXAMPLE

NOTES  
.

BUGS  
.

SEE ALSO  
PA\_AllocVecPooled(), PA\_CreatePool(), PA\_DeletePool()

## 1.16 progargs.library/PA\_GetArgEntry

22 Dec 1994 16:12:04

NAME  
PA\_GetArgEntry -- find ArgEntry structure for an argument ID.

SYNOPSIS  
argEntry = PA\_GetArgEntry( progArgs, entryID )  
D0 A0 D0  
  
struct ArgEntry \* PA\_GetArgEntry( struct ProgArgs \*, Tag )

FUNCTION  
Returns the ArgEntry structure associated with the given argument ID. This is the same structure which was originally passed in by the program to describe this ID.

INPUTS  
progArgs - ( struct ProgArgs \* ) Argument set to look in.  
entryID - ( Tag ) Argument ID to get entry for.

RESULT  
argEntry - ( struct ArgEntry \* ) ArgEntry describing the ID or NULL.

TAGS

EXAMPLE

NOTES  
.

BUGS  
.

SEE ALSO

---



```
AllocProgArgsA(), PA_NextArgEntry()
```

## 1.17 progargs.library/PA\_NextArgEntry

22 Dec 1994 16:15:55

### NAME

PA\_NextArgEntry -- get next sequential ArgEntry in an argument set.

### SYNOPSIS

```
nextEntry = PA_NextArgEntry( progArgs, curEntry )
D0                      A0          A1
```

```
struct ArgEntry * PA_NextArgEntry( struct ProgArgs *,
                                   struct ArgEntry * )
```

### FUNCTION

Returns the ArgEntry which is sequentially after the given one. If the given ArgEntry is NULL, the first entry in the argument set is returned. If the current entry is the last in the argument set, a NULL is returned. This function can be used to loop over all of the arguments defined in an argument set.

### INPUTS

```
progArgs - ( struct ProgArgs * ) Argument set to look in.
curEntry - ( struct ArgEntry * ) Current entry in the set.
```

### RESULT

```
nextEntry - ( struct ArgEntry * ) Entry sequentially after the
given current one.
```

### TAGS

### EXAMPLE

```
/* Show all argument templates in an argument set */

void ShowArguments(struct ProgArgs * pa)
{
    struct ArgEntry* ae;

    ae = NULL;
    while( (ae=PA_NextArgEntry(pa,ae)) ) {

        printf("%s\n",ae->ae_Template);

    }
}
```

### NOTES

.

### BUGS

.

SEE ALSO  
 AllocProgArgsA(), PA\_GetArgEntry()

## 1.18 progargs.library/PA\_RemTagArgsA

11 Dec 1994 20:23:29

### NAME

PA\_RemTagArgsA -- remove tags from an argument array.  
 PA\_RemTagArgs -- varargs stub for PA\_RemTagArgsA().

### SYNOPSIS

```
result = PA_RemTagArgsA( base, args )
D0                      A0      A1

BOOL PA_RemTagArgsA( struct TagArgs *, struct TagItem * )

result = PA_RemTagArgsA( base, arg1Type, ... )

BOOL PA_RemTagArgsA( struct TagArgs *, Tag, ... )
```

### FUNCTION

Removes the given tag values from the argument array. They will no longer exist in this argument array's tag list.

The tag ti\_Data fields of the argument array MUST contain the magic values REMARG\_REMOVE or REMARG\_IGNORE to control whether or not this argument will actually be removed; any other value is illegal.

### INPUTS

base - ( struct TagArgs \* ) Argument array to modify.  
 args - ( struct TagItem \* ) Arguments to remove.

### RESULT

result - ( BOOL ) TRUE if all is okay, else FALSE.

### TAGS

### EXAMPLE

### NOTES

.

### BUGS

.

### SEE ALSO

PA\_AddTagArgsA(), PA\_AllocTagArgsA()

## 1.19 progargs.library/RemProgArgsA

11 Dec 1994 20:12:26

## NAME

RemProgArgsA -- remove values from an argument set.  
RemProgArgs -- varargs stub for RemProgArgsA().

## SYNOPSIS

```
result = RemProgArgsA( progArgs, argTags )
D0                      A0          A1
```

```
BOOL RemProgArgsA( struct ProgArgs *, struct TagItem * )
```

```
result = RemProgArgsA( progArgs, arg1Tag, ... )
```

```
BOOL RemProgArgsA( struct ProgArgs *, Tag, ... )
```

## FUNCTION

This allows a program to remove the individual values in an argument set. Note that these are the values initially supplied as the entries to AllocProgArgsA(); this function does NOT allow a program to add completely new arguments. The act of 'removing' a value simply means that the associated ArgEntry no longer has a value; a search for it in the argument tag list will result in it not being found, and if the arguments are written out to disk, this one will not be supplied. [Or will only be supplied as a comment field, if the argument has a description associated with it.]

The tag ti\_Data fields of the argument array MUST contain the magic values REMARG\_REMOVE or REMARG\_IGNORE to control whether or not this argument will actually be removed; any other value is illegal.

## INPUTS

progArgs - ( struct ProgArgs \* ) Argument set to modify.  
argTags - ( struct TagItem \* ) Arguments to remove.

## RESULT

result - ( BOOL ) TRUE if all is okay; FALSE if there was a problem.

## TAGS

## EXAMPLE

## NOTES

.

## BUGS

.

## SEE ALSO

AllocProgArgs(), AddProgArgs()