Fast and Safe “C” Strings

For Better “C” Programming Development

Version 1.1

Programmers Guide

Open System Command and Retrieval (OSCAR).

Fast and Safe “C” Strings *“Programmers Guide"*.

Second Edition (July, 1999)
Third Edition – First Release **(February, 2007)**

This is a major revision of, and obseletes, the
First Edition of the Fast and Safe “C” String
"Programmers Guide" November 1992

Postal Address:

16/38 Kings Park Road,
West Perth,
Western Australia,
Australia 6005.

Telephone **(61)-8-9324 1119, Mobile (61) 401 054 155**
Email **oscarptyltd@ozemail.com.au**

Preface

Portability of languages, operating systems, system utilities and applications programs is a highly desirable goal. While coding in Assembler is a must for maximum efficiency, high level languages definitely give the desired portability, *as long as you have an efficient and effective compiler and good string handling facilities.*

"C" is being used more and more to provide this portability between machines, but it lacks the ability to use many of the string instructions provided by IBM 370 and Intel 808xx chips effectively - thus leaving *millions* of machines running up to 25 (repeat *twenty-five*) times slower than they ought to be running for string copies and compares. And string handling is required for most applications.

There are two main versions of these routines provided in this package:

- A generic set of macros that should work on any ANSI compatible C compiler. These give speed improvements of about 5 fold over the usual:

  while (*dest++=*src++);

- A set of routines that should be usable with any C compiler that has the capability to generate ASM code and call in a macro assembler. Tests show a speed improvement of about 20 fold on an 8086 machine, and probably similar on a 370 type mainframe.

- The ability to generate optimal assembler code when used with Borland C or C++ Builder and Tasm.

Both sets of routines automatically check that the receiving string is long enough to hold that data and truncate if necessary, and blank fill on the right if you are copying a short string to a longer FIXED length string.

The routines in this package:

- Speed copying strings by a factor of up to 25 times through the use Assembler Macros. Generic "C" macros speed Copying about 5 times.
- Speed comparing strings by a factor of up to 10 times through the use of generic "C" Macros.
- Add a generic **CPY** function for copying one string to another (fixed and varying length strings).
- Automatically truncate long strings when they are copied to short strings so that storage following the shorter strings is not accidentally overwritten.
- Automatically pads longer strings with **blanks** when a short string is copied to a longer fixed length string.
- Add a generic **CMP** function for comparing fixed and varying length strings with each other, and optionally checking that the longer strings have blanks on the end thus providing a true string comparison.
- Provide an easier method of defining and using **EXTERNAL** variables.

CONTENTS

Increasing the Speed and Safety of C Strings

Many languages such as PL/I and PASCAL keep the current length of VARYING strings at the front of the string, instead of C's method of using a Binary 0 at the end of each string. Thus when you copy a string to another string (or compare one string with another), C must either find out how much data to copy or compare, or it must look for the binary 0 as it is COPYing.

Many machines such as the IBM PC with Intel 80xxx chips or equivalent, and IBM mainframes like the 370 series of computers have instructions (or groups of instructions) for moving and comparing data quickly. For example:

8086 REP MOVSW and REP MOVSB

370 MVC, MVCL, CLC etc..

Z80 LDIR

To better utilize the above instructions, these routines were developed.

In addition, "C" string handling is usually performed with string functions - and functions with parameters are expensive to call and error prone (for example, there is nothing in "C" to stop a long string being copied to a short string, and destroying data that follows the short string).

While we consider these routines to be a temporary solution until C compiler manufacturers add inbuilt string support to C, and have it incorporated into Standard C, the routines offer greatly increased speed and safety over the normal STR routines currently provided.

Other Problems with C Strings

In addition to the Efficiency problem mentioned above, "C" has no checking when strings are copied. It is very easy to copy too much data from one place to another, and overwrite areas of storage accidentally, thus resulting in faulty programs with potentially disastrous results.

PL/I, PASCAL and the IBM 370 Assembler automatically check if too much data is being moved, and truncates a string copy if necessary.

PL/I

PL/I does a lot of work when you assign one string to another. First of all, it checks that the *receiving string* is long enough to hold the data.

Secondly, for **Fixed Length Strings** PL/I pads the data with blanks or truncates the string as necessary. For **Varying Length Strings**, no padding applies.

PL/I is faster because it can simply load the length of the string from the first word or byte if the string is Varying Length, or directly use the length of Fixed Length strings. It doesn't waste time searching for the binary zero.

The code generated by IBM's PL/I is *inline* code to move or compare the data for fixed length strings.

The following is the result of the small program that comes with this package. It shows the relative times for various methods of copying strings.

Copying a 66 byte string 300000 times

Type of Copy	Elap Time	Loop Overhead	Real Copy	Ratio
INCRCPY (Static Pointers)	136.00	2.00	134.00	1.00
INCRCPY (Register Pointers)	67.00	2.00	65.00	2.06
STRCPY	31.00	2.00	29.00	4.62
MEMCPY	12.00	2.00	10.00	13.40
ASM	7.00	2.00	5.00	**26.80**
		NOTE	=====	===>^

/*Using the Routines*

To use the routines, you use special macros to define your variables. The macros set up **ENUM**s and various locations so that the **CPY, CMP** and other macros can generate efficient code for their respective functions.

There are three main types of definitions:

- "C" string definitions
- PL/I definitions
- PASCAL definitions.

The definitions are all similar, but force the **CPY** and **CMP** macros to take slightly different actions. For example, to define a PASCAL style string, you code:

var (name,charfixed,30,"Initial Value",static);

This describes a PASCAL varying string with a maximum of 30 characters, initialized to "Initial Value". To define a PL/I fixed string, the format is similar:

dcl (name,charfixed,30,"Initial Value",auto);

Differences between the PL/I, PASCAL and "C" Definitions

Although the definitions of the strings look similar, the **CPY** and **CMP** macros perform slightly different actions depending on the type of variable being used.

When a string is defined as a "C" string, the **STRLEN** function is used to find the length of the string *every time*. This means that there is no advantage in using this type of string with these routines - however it does allow you to import "C" variables easily.

PASCAL strings and PL/I strings are very similar: the length of the string is maintained by these routines. For fixed length strings, the current length is held at the end of the string; for varying length strings, the current length of the string is held at the beginning of the string, either as a word or byte depending on the definition.

For *long* PL/I varying length strings, a word or integer is used to hold the length. For PASCAL or *short* PL/I strings, a single byte is used thus limiting the maximum length to 255 bytes.

Note: There is a very important difference between PL/I and PASCAL strings. When comparing PL/I strings *with any other type of string*, the shorter string is padded with blanks so that a true compare of strings can be made rather than the invalid method of using the difference between the lengths of the strings if the strings compare equal to that point.

A Partial Fix

Based on the timings in the table above, there is no doubt that "C" needs proper string handling incorporated in the language design. Simply extending the syntax to allow 'quoted strings' as well as "quoted strings" would fix most of the problems, and allow compiler writers to generate superb code.

Until this is done, these "C" macros and supporting routines will give most of the speed benefits one can obtain with Assembler.

Note: These routines are not quite as they could be due to severe limitations in the "C" macro processor. For example, the macros provided would be *much* more powerful if it was possible to test the value of a defined variable in a macro, and generate code based on the result. It seems that full solutions must wait for a "C" compiler that does the job more efficiently, or for a more flexible macro processor.

An Overview of How the Routines Function

When you use the macros supplied to define variables, the macros set up **ENUM**s and other locations that can be used by the **CPY** and other macros. For example, an **ENUM** called **VSTR_TYPE** tell the macros if the string is Fixed, Varying, PL/I, Pascal or a C string.

Normally, in an integrated development environment, these macros will call *memcpy* and *memcmp* to perform the required actions. When your program is executing correctly, you can use the INLINE define to trigger the Assembler pass which will generate optimal code for the moves and compares.

NOTE:

Depending on the optimization provided in your compiler, the generic C macros that are designed to be used in an integrated environment will generate optimal code. For example, MSC 5, MSC 6, Borland and Zortech will generate better code than QuickC because they optimize out the "dead code" generated by some of the macros.

Purpose

The Declare or DCL statement defines and initializes variables. It follows this form:

dcl (*name,attribute,length,initial-value,class***) ;**

where

name is the name of the variable being declared.

and *attributes* are:

charvar specifies a *varying* length character string. The **pligstr** *length* specifies the largest size that the string can contain - copying longer strings to the variable results in the string being truncated.

Varying length strings are similar to "C" strings - they take on the length of last string copied to them.

charfixed specifies a *fixed* length string. The *length* specifies the *current and maximum* length of the string. Fixed length strings are always fixed in length. If you copy a short string or literal to a fixed length string, the string is padded with blanks on the right hand side.

Fixed length strings are similar to **CHAR** strings in PASCAL - they are always a constant length.

bin specifies a Binary or Integer Variable. The *lengths* specify the size of the variable in *bits*. The length specifies the type of "C" integer as shown below:

7	specifies a *signed* char.
8	specifies an *unsigned* char.
15	specifies a *signed* integer.
16	specifies an *unsigned* integer.
31	specifies a *signed* long integer.
32	specifies an *unsigned* long integer.

float specifies a Floating Point Variable.

and

initial-val specifies the value that is to be given to the variable.

and *class* is described in Appendix A.

PL/I Style Variables

When you define a variable to be a PL/I variable, you define to the **CPY, CMP** and other macros that the variable is a special style.

The **DCL** macros use **ENUM**s to communicate to the **CPY** macro that special actions may be required when a variable is copied to a PL/I style variable. Fixed length strings are either padded with blanks or truncated when another variable is copied to it.

Notes

1. Binary and Float variables behave in exactly the same manner as the equivalent "C" variables - you can assign numbers directly to them, use them in **PRINTF** statements and other normal functions.
2. Strings are handled quite differently from "C" strings. The macros define the maximum lengths of the strings, and contain a special length integer that contains the current length of the string, resulting in improved performance when copying, comparing etc.. one string to another.
3. For strings, the name you specify is defined as a *const* pointer to another variable containing the actual string. This does two things: it ensures that you cannot accidentally overwrite the variable, and also allows the macro to define the string in a special format depending on the type of string (fixed length, or long or short varying strings).
4. *Note:* There is a very important difference between PL/I and PASCAL strings. When comparing PL/I strings *with any other type of string*, the shorter string is padded with blanks so that a true compare of strings can be made rather than the PASCAL method of using the difference between the lengths of the strings if the strings compare equal to that point.

Examples

1. **dcl (name,charfixed,20," ",static);**

   The declaration above describes a variable called *name. Name* is a fixed length string with a maximum length of 20 characters. The variable is defined as a **static** variable.

2. **dcl (addr,charvar,80," ",ext);**

   The declaration above describes a variable called *addr. Addr* is a varying length string with a maximum length of 80 characters. Because this is a varying length string, the *current* length of the string is held in the first word.

   The variable is given the **ext** (external) class, and can be referenced in another module that also declares the variable **addr** with the **extern** class.

3. **dcl (j,bin,16,0,auto);**

   The declaration above describes a variable called *j. J* is a binary fixed integer of 16 bits in length (an unsigned int). Coding 15 instead of 16 would make *j* a signed integer.

The CAT, CATCHAR and CATLIT Functions

Purpose

The **CAT** function concatenates one variable with another. It follows this form:

> **cat** (*dest,src)*
>
> **catchar** *(dest,'char');*
>
> **catlit** (*dest,"literal"*)
>
> where *dest* and *src* are variables defined with the **DCL, VAR** or **CSTR** macros.

Declaring Variables

Before using the generic **CAT** function, you must use the appropriate declaration macro to define the variable.

Notes

1. The **CAT** and **CATLIT** macro appends the source variable to the destination variable.
2. The *dest* string cannot be a Fixed String. By definition, fixed length strings are automatically padded with blanks if a short string is copied to a shorter string, or truncated if a long string is copied to a short fixed length string. In any case, the string is already at maximum length, and thus concatenation will have no effect.

Examples

1. **dcl (name,charvar,20,"John Smith",static);**

   catlit(name,",");

   The declaration above describes a variable called *name. Name* is a varying length string with a maximum length of 20 characters. The **catlit** adds a "," to the string *name*.

The CLEAR Function

Purpose

The **CLEAR** function sets a variable to 0, a null string, or all blanks if it is a fixed length string. It follows this form:

> **clear** (*variable)*
>
> where *variable* is a variable defined with the **DCL, VAR** or **CSTR** macros.

Declaring Variables

Before using the generic **CLEAR** function, you must use the appropriate declaration macro to define the variable.

Notes

1. The **CLEAR** macro sets binary numbers to 0, varying length strings to a null ("") string, and fixed length strings to blanks.
2. Due to limitations in the C macro processor, it is impossible to set fixed lengths to the correct number of blanks when the string is defined, unless the *"initial-value"* has the correct number of blanks for padding.
3. However, when a fixed length is first referenced, it is initialized correctly, that is padded with blanks if necessary.
4. You may use the **CLEAR** macro to initialize the strings correctly, or copy the string to itself.

Examples

1. **dcl (name,charvar,20,"John Smith",static);**

   clear(name);

   The declaration above describes a variable called *name. Name* is a varying length string with a maximum length of 20 characters. The **clear** sets the string to *null*, and sets the current length to 0.

The CMP, CMPCHAR and CMPLIT Instructions

Purpose

The **CMP, CMPCHAR** and **CMPLIT** function compares one variable to another. It follows this form:

> **cmp** (*lhs_variable,rhs_variable)*
>
> **cmpchar** *(dest,'char');*
>
> **cmplit** (*lhs_variable,"literal-string")*
>
> where *lhs_variable* is a variable defined with the **DCL, VAR** or **CSTR** macros.

Declaring Variables

Before using the generic **CMP** instructions, you must use the appropriate declaration macro to define the variable.

The declaration macros such as **DCL** use **ENUM**s to communicate to the **CMP** macro that special actions may be required when a variable is copied to a PL/I, PASCAL or "C" style variable. Fixed length strings are effectively padded with blanks when another variable is compared to it.

Although the definitions of the strings look similar, the **CMP** and **CMPLIT** macros perform slightly different actions depending on the type of variable being used.

When a string is defined as a "C" string, the **STRLEN** function is used to find the length of the string *every time*. This means that there is no advantage in using this type of string - although it does allow you to import "C" variables easily.

PASCAL strings and PL/I strings are similar: the length of the string is known to the routines. For fixed length strings, the current length is held at the end of the string; for varying length strings, the current length of the string is held at the beginning of the string, either as a word or byte depending on the definition.

For *long* PL/I varying length strings, a word or integer is used to hold the length. For short PL/I or PASCAL strings, a single byte is used thus limiting the maximum length to 255 bytes.

Notes

1. The **CMP** macro compares the source variable to the destination variable.
2. Strings are handled quite differently from "C" strings. The macros define the maximum lengths of the strings, and contain a special length integer that contains the current length of the string, resulting in improved performance when copying, comparing etc. one string to another.
3. *Note:* There is a very important difference between PL/I and PASCAL strings. When comparing PL/I strings with any other type of string, the shorter string is padded with blanks so that a true compare of strings can be made rather than the PASCAL method of using the difference between the lengths of the strings if the strings compare equal to that point.

Examples

1. **dcl (lang,charfixed,20,"C",static);**

   if (cmplit(lang,"C")==0) ...

   The declaration above describes a variable called *lang*. *Lang* is a fixed length string with a maximum length of 20 characters. The **cmplit** compares the value in the string *lang* to a literal "C", and if it is equal, executes the next instruction.

2. **dcl (addr,charvar,80," ");**

   The declaration above describes a variable called *addr*. *Addr* is a varying length string with a maximum length of 80 characters. Because this is a varying length string, the *current* length of the string is held in the first word.

The CPY, CPYCHAR and CPYLIT Instructions

Purpose

The **CPY** instruction copies one variable to another. It follows this form:

> **cpy** (*destination,source*) ;
>
> **cpychar** *(dest,'char');*
>
> **cpylit** (*destination,"literal") ;*
>
> where *destination* and *source* are variables defined with the **DCL, VAR** or **CSTR** macros.

Declaring Variables

Before using the generic **CPY** instructions, you must use the appropriate declaration macro to define the variable.

The declaration macros such as **DCL** use **ENUM**s to communicate to the **CPY** macro that special actions may be required when a variable is copied to a PL/I, PASCAL or "C" style variable. Fixed length strings are either padded with blanks or truncated when another variable is copied to it.

Although the definitions of the strings look similar, the **CPY** and **CMP** macros perform slightly different actions depending on the type of variable being used.

When a string is defined as a "C" string, the **STRLEN** function is used to find the length of the string *every time*. This means that there is no advantage in using this type of string - although it does allow you to import "C" variables easily.

PASCAL strings and PL/I strings are similar: the length of the string is known to the routines. For fixed length strings, the current length is held at the end of the string; for varying length strings, the current length of the string is held at the beginning of the string, either as a word or byte depending on the definition.

For *long* PL/I or PASCAL varying length strings, a word or integer is used to hold the length, otherwise a single byte is used, thus limiting the maximum length to 255 bytes.

Notes

1. The **CPY** macro copies the source variable to the destination variable.

   If the source and destination are numbers (Binary or Float), the rules applying to any conversion are the same as for your "C" compiler, and truncation or padding of numbers may occur.

2. Strings are handled quite differently from "C" strings. The macros define the maximum lengths of the strings, and contain a special length integer that contains the current length of the string, resulting in improved performance when copying, comparing etc. one string to another.

Examples

1. **dcl (name,charfixed,20," ");**

   The declaration above describes a variable called *name. Name* is a fixed length string with a maximum length of 20 characters.

2. **dcl (addr,charvar,80," ");**

   The declaration above describes a variable called *addr. Addr* is a varying length string with a maximum length of 80 characters. Because this is a varying length string, the *current* length of the string is held in the first word.

3. **dcl (j,bin,16,0);**

   The declaration above describes a variable called *j*. *J* is a binary fixed integer of 16 bits in length (an unsigned int). Coding 15 instead of 16 would make *j* a signed integer.

Appendix A - Storage Classes

The following storage classes may be specified in the "cstr", "dcl" and "var" storage definitions.

auto specifies that the variable is to be allocated as a local variable - it disappears when the function ends.

Auto variables can only be declared in functions.

ext specifies that the variable is to be allocated as an external variable. External variables can be referenced by different modules or functions in your program.

External variables can only be declared outside functions.

extern specifies that the variable has been defined in another module as an *ext* variable.

Extern variables can be declared anywhere in your program.

Note: The initial value is ignored. You can code the value as "", but it must be coded.

static specifies that the variable is to allocated as a local variable but that the variable will retain its value between function call.

Static variables can be declared anywhere in your program.

Using Fast ASM Macros

"C" Compilers that have intrinsic functions for *memcpy, memset* and *memcmp* automatically generate quick code when used with these functions. For example, copying a 30 character fixed string to another will generate a single 370 **MVC** instruction.

Some compilers (such as Borland's C and the SAS 370 C) allow ASM instructions to be placed in the generated code. When used with the appropriate set of macro instructions, these compilers can produce highly efficient, optimal code for strings.

Additionally, the C Macros in this package come with equivalent Assembler Macros which will generate optimal code for the type of copy or compare being performed. To use this facility, simply set the variable "inline" or "INLINE" before #including the safe, fast C macros.

For example:

BCC -Dinline ...

or **#define inline**

then

#include <dclstart.h>

You can define the macro in Borland's IDE, and still use the Integrated Developement System.

When the **INLINE** option is used, the Assembler will be called to generate the optimal code.

Appendix C - Use of External Variables

Compared with PL/I or Turbo PASCAL, specifying global initialized variables in C is extremely difficult and error prone. For example in PL/I it is a simple matter of declaring a variable with the External (or ext) attribute in all the modules that need to use that variable, as follows:

PL/I Globals

Module 1

```
dcl max_files bin fixed init(100) ext;    /* Define and Initialize Variable */
```

Module 2

```
dcl max_files bin fixed init(100) ext;    /* Define and Initialize Variable */
```

Note that the same definition can be used in multiple modules - the link editor will combine external variables correctly.

C Globals

In C, the following must be coded:

Module 1

```
int max_files= 100;                /* Define and Initialize Variable */
```

Module 2

```
extern int max_files;              /* CANNOT specify Init Value */
```

In other words, you define and initialize the variable in one module, and reference the definition with the extern in the module(s) that need to use it.

If you specify an initial value in module 2, a link editor error occurs - in other words you must have two ".h" or include files, one with initial values, and the other specifying the keyword *extern* - without the initial value.

Simplified External Variables

To overcome this problem, when you use the ***dcl*** or similar macros to define your external data storage areas, simply code the following before including the data definition file:

```
#define ext extern
```

This will replace all the **ext** definitions with **extern** and remove all the initial values.

For example:

Module 1

```
dcl(max_files,bin31,100,ext); /* Define and Initialize Variable */
```

Module 2

```
#define ext extern

dcl(max_files,bin31,100,ext); /* Define and Initialize Variable */
```

Note: Some compilers will not replace the **ext** with **extern** correctly. If you compiler will not do this replacement correctly, simply edit your include file, and replace all **ext)** with **extern)**, and save it as a different file. Then, apart from the change from **ext)** to **extern)**, the files are identical and may be changed easily.

Readers Comment Form

This manual is part of the Oscar library that serves as a reference source for Managers, Systems Analysts, Programmers and Operators. This form may be used to communicate your views about this publication.

OSCAR may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use any information you supply.

Possible topics for comments are:

Clarity Accuracy Completeness Organization Legibility

If you wish a reply, give your name and address or Email address

_____

_____

_____

_____

_____

Number of latest Newsletter associated with this publication: _____

Please send your comments to:

Email oscarptyltd@ozemail.com.au