Application Note: AN10066

Using safe pointers for string processing

This application note is a short how-to on programming/using the xTIMEcomposer tools. It shows using safe pointers for string processing.

Required tools and libraries

This application note is based on the following components:

- xTIMEcomposer Tools - Version 14.0.0

Required hardware

Programming how-tos are generally not specific to any particular hardware and can usually run on all XMOS devices. See the contents of the note for full details.

1 Using safe pointers for string processing

When using the multicore extensions to C, pointers are safe. This means that extra checks are inserted to avoid common memory access errors. It also means that pointers sometimes need to be annotated to indicate their use.

A couple of standard string processing functions are presented here to show the use of safe pointers in xC.

2 strlen

The following function implements a version of the standard `strlen` function that gets the length of a zero-terminated string:

```
int my_strlen_1(const char *str)
{
  int n = 0;
  while (*str != 0) {
    str++;
    n++;
  }
  return n;
}
```

Here there is no difference to standard C. However, the implementation implements bounds checking. This means that if a string is passed in that is not zero terminated, the program will trap instead of reading invalid memory and returing a nonsense value:

```
char str[3] = "abc";  // oh-oh, not zero-terminated
int len = my_strlen_1(str); // this will trap since the loop will
                            // read past the three bytes allocated to str
```

Since the trap occurs here at the point of error it is much easier to debug than a later follow-on subtle program error.

Although the above definition of `strlen` is workable, it could be better. In xC, arrays and pointers are not the same - although they can be implictly converted. In particular:

- Array parameters can only access the elements in front of them (you can subtract from pointers to access behind).
- Arrays cannot be null.

If your function satisfies the properties of being an array, then it is more efficient and safer to use an array argument. So the function becomes:

```
int my_strlen_2(const char str[])
{
  int n = 0;
  const char *p = str;
  while (*p != 0) {
    p++;
    n++;
  }
  return n;
}
```

In this case you get a more efficient implementation since the bounds checking does not need to worry about elements earlier in memory than the pointer `str`. You also get extra safety checks. The argument cannot be null so the following code will trap:

```
char *str = null;
int len = my_strlen_2(str);  // this will trap at the point of the call
```

This will trap at the point of the function call (not within the `my_strlen_2` function). Having the trap as early as possible in execution greatly helps you debug where the cause of the error is.

3 strchr

The `strchr` function returns a pointer to the first occurence of a character within a string. The C prototype for the function is:

```
char * strchr(char * str, int c);
```

However, if you try and prototype a function like this in xC you will get the error:

```
error: pointer return type must be marked movable, alias or unsafe
```

To understand this error, you need to understand that safe pointers have three types in xC: *restricted*, *aliasing* and *movable*. By default, local pointers are aliasing - so you can have more than one pointer pointing to the same object. Function parameters default to *restricted* - so they cannot alias each other. Return values to functions cannot be restricted so they must be explicitly marked as aliasing or movable.

If a pointer return value is marked as aliasing, it can alias any of the aliasing pointer parameters to the function (or any global objects). In this case both the incoming pointer and the return value need to be marked as aliasing. Then the function can be written in the obvious way:

```
char * alias my_strchr(char * alias str, int c) {
  char *p = str;
  while (*p != 0 && *p != c) {
    p++;
  }
  if (*p == 0)
    return null;
  else
    return p;
}
```

By keeping track of aliasing pointers, the compiler can check for program errors involving parallel race conditions and dangling pointers.