UCDebug in action

Nicholas Cutler

Anyone following RISC OS shows recently may have been intrigued by the talks about UCDebug, an entirely new development from an equally unexpected source, the University of Cantabria in Spain. This application was introduced by Bernard Boase in Archive 24:6. Having had an opportunity to try this software out for myself, this article will therefore attempt to show what UCDebug can do.

Firstly, UCDebug is a debugger, a tool used by programmers to help find errors (bugs) in their programs. Typically, a debugger will allow you to run a program in a controlled environment, to stop execution at arbitary points, and to examine data in variables and memory. To facilitate this, processor architectures often have a special breakpoint instruction, and extra registers allowing the debugger to maintain its own state entirely separate from that of the faulty program being executed. Some degree of memory protection is also necessary to prevent a faulty program from affecting other tasks, or the operating system itself.

Debuggers naturally fall into two types, machine level and source level. UCdebug falls into the first category as it allows you to step through a program by individual assembly language instructions, and to examine the contents of the processor's registers. Source level debuggers have additional symbolic capabilities allowing you to work more naturally with programs in a high-level language. The obvious example of the latter being DDT, the debugger included with the Desktop Development Environment, the programming tools sold by RISC OS Open.

It is also fair to say that UCDebug was designed to be an educational tool to help students gain some experience with assembly language programming. The low memory requirements and relative simplicity of RISC OS, plus the availability of economical hardware in the form of the Raspberry Pi, makes it ideal for this task. It does, however, mean that UCDebug has a few limitations which will make it less useful to software developers. For this reason, I have conceived this article not as a software review, but rather as an introduction to debuggers and their capabilities. I also encountered some difficulties in using UCDebug, and more especially in preparing programs for debugging, so my other purpose here is to communicate what I learnt about that process.

Before considering using UCDebug, you should be aware that it is very limited in the range of hardware that it supports. Initial attempts with the Pandaboard which I now use as my main machine failed: the debugger would load, but crashed as soon as I tried to execute the program. Similar behaviour was observed on a Raspberry Pi 4, although it did work on the older Pi 3B+.

To sensibly use a debugger, you will first need to prepare a program for debugging. This is where another limitation of UCDebug becomes apparent: it is a machine level debugger, and consequently your program will need to have been written entirely in assembly language. This is also likely to considerably restrict the use of UCDebug as most assembly language programmers will have access to alternative tools. Although I did try some simple programs in the C programming language in the hope of working with the machine code output by the compiler, I was unable to obtain an executable program which the debugger could accept. Consequently some familiarity with assembly language will be beneficial to follow this article.

Another limitation is that it will only work

with programs in the Linux-like ELF format, rather than the native format. This will mean that you will need to use the GNU assember *as*, and the linker *ld* to prepare programs. Those, like me, who may be more used to the *objasm* assembler can still write programs in that format and assemble them with the alternative *asasm* assembler. If you prefer, I have included a Perl script to convert assember files to the GNU format. This can be invoked using a command like:

```
perl convasm/pl s.asmfile > s.gnusource
```

The very old version of Perl included with the RISC OS Open development tools is quite adequate for this task.

If you do prepare a source file in the GNU format, it is worth remembering that *as* doesn't understand the ENTRY directive; instead the entry point of your program should be the label _start, which should also be defined as a global symbol.

```
  .global _start
_start:
   @ your code follows ...
```

Another difference to be aware of is that *as* will not accept the ADR pseudo-instruction to generate an address in a register when it refers to a different area. If you need to use a region of memory for data storage then you will have to do something like:

```
  .data
mydata:
  @ literal data ...

  .text
  LDR r0,datalabel
datalabel:
  .word mydata
```

Once an assembly language file has been prepared using either method, it will need to be assembled using a command like:

```
as -mapcs-32 -mcpu=cortex-a53 gnusource.s -o gnusource.o
```

The cpu parameter can be changed to suit the machine you are using. If your program uses any floating point instructions then additionally include the option `-mfpu=vfpv2` on the command line. Because UCdebug only works at the machine level, it is not necessary to include debugging information with the program, and the -g option can be omitted.

Finally, the program must be 'linked' to create a finished executable version. Even when there is only one source file, the linking stage is necessary to update any absolute addresses. Another restriction of UCDebug is that it requires programs to start at the address of &18088, instead of the more usual &8000. To achieve this, link your program with a command like:

```
ld -Ttext=18088 o.gnusource -o progname
```

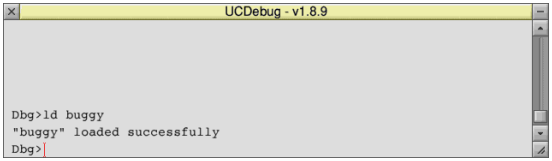
Having prepared a program, it is now possible to load it into the debugger. To begin with it is easiest to use the ready prepared program buggy for this purpose. This deliberately faulty program uses the sieve method to find all the prime numbers less than 1024. The remainder of this article will use UCDebug to find the errors in this program, but curious readers will find the source of the corrected program in s.sieve.

Firstly, load the debugger by double clicking on it in the usual way. Please note that UCDebug was itself built with *gcc*, and will need the !Sharedlibs directory to be installed. Also, before running it for the first time, you may need set the filetype of the !Runimage file to &E1F, and remove the suffix from the name.

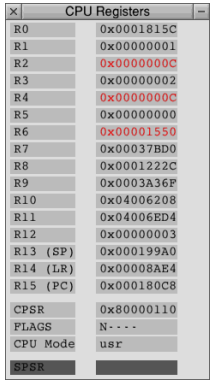
Once started, UCDebug will place an icon on the iconbar, and open four windows, namely: a console where you can interact with the debugger, a disassembly of your program, the

contents of the processor's registers, and the data in memory. We will start with the first of these windows:

This is where you can enter commands to control the debugger, and where you will see messages confirming the result. To begin with locate the example program "buggy", and load it into the debugger with the command `ld buggy`. If all goes well, confirmation will be given as in the illustration below:

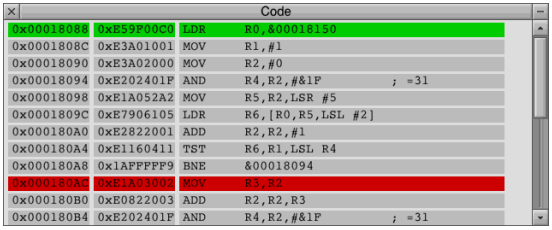


Having loaded your program, UCDebug does not run it immediately, but rather stops at the first instruction (highlighted in green in the disassembly window). For a full description of this example program refer to the comments in the source file. However, the first loop scans for the next number in the list which has not been marked. This will be the next smallest prime number, and the loop will finish with this stored in R2. So set a breakpoint at &180AC with the command `br 180AC`. As confirmation that this has worked the relevant instruction in the disassembly will be highlighted in red. It is also possible to set breakpoints by clicking the middle button of the mouse over the desired location. However, I found that the mouse pointer freezes once the debugger has started running your program. If you find this to be the case, then you will need to resort entirely to typed commands.



Having set the breakpoint, type `go`, and the debugger will now run your program as far as that point. Look in the window showing the contents of the registers and check the value in R2. This now shows the value 1 (highlighted in red because the register value has been changed by the program). In fact we were expecing it to be set to the first prime number, namely 2. The solution to this is either to start with number 1 already marked, or to start the loop counter from 1 rather than 0. However, it is not necessary to correct this immediately, as the debugger instead allows us to change the value in the register and continue with the next part of the program. To do this use the command `reg R2=2` to change the contents of R2. When changing the contents of registers or memory, you can enter values in decimal, or in hex by prefixing them with 0x. However, all memory addresses are assumed to be in hex, so that the prefix is not necessary when specifying locations for breakpoints.

The next stage of the program proceeds to mark all those numbers which are multiples of the specified prime. To do this, the program takes a copy of the value of R2 and repeatedly adds this to generate all multiples. Using a single bit to mark a number as composite, a pattern is built up in memory and R6. To check this next part of the program set a breakpoint at &180C8 (using `br 180C8`), just after R6 is saved to memory. Now type go a few times and watch the pattern build up in memory and in R6. After a few iterations the value of R6 is &150, representing the pattern 1 0101 0000 in



binary. The program is correctly building up the expected pattern of alternate bits representing all the multiples of two, but it started with the fifth bit, not the fourth. This is because the program has been counting up from one, whereas the computer counts from zero.

To correct this we can use the debugger to alter the values in memory and R2. Issue the command `set l8l5C=0xA8` to change memory, and `reg R2=7` to update the register.

Having changed this, there are hopefully no more errors within this loop, so we now want to allow this part of the program to complete without stopping on each iteration. To do this we clear the second breakpoint at &180C8 with the command `br l80C8` (note that the br command simply toggles a breakpoint between on and off). Then set a further breakpoint after the loop, on the instruction `add r2,r3,#l`, with the command `br l80D4`. Now start the program running again by typing `go`. The loop should then complete and all multiples of two should have been marked in memory. We can use the memory window to check this, and locations from &1815C should show a pattern of A's in the hexadecimal display.

At this point we can be reasonably confident that there are no further errors in the second loop, `sieve_n`. Continuing the execution of the program again, it will jump back to the beginning to find the next prime number. The debugger will, once again, stop at our first breakpoint at &180AC, where we can see that the value of R2 is, once again incorrect having missed out the prime number 3. Change this register to the correct value with `reg R2=3`.

The next step will be to confirm the error in the `sieve_n` loop, so we will use the debugger to execute just one machine instruction at a time. Type the command tr a couple of times, so that the debugger reaches the first line of the loop, at the statement `and r4,r2,#3l`. Having been copied into R3, the value of R2 should itself have been be reduced by one. The simplest way to do this is to insert an extra instruction `sub r2,r2,#l` immediately after the move statement, although a more satisfactory solution is to increment R2 conditionally in the findprime loop. For now, we will use the debugger to change R2 so that it is ready for the first iteration of the loop with the command `reg R2=5` (the first multiple of 3, minus 1 because the computer counts from zero). Check in the register display window that this has taken effect, but that R3 is still equal to 3. Now, start the programming running again and allow the second loop to finish with the debugger stopped at &180D4. In the memory window we should notice that the pattern in memory has changed now that all multiples of 3 have been marked:

Data

0x00018154	00000000	00000000	AEBAEBA8	EBAEBAEB	.....
0x00018164	BAEBAEBA	AEBAEBAE	EBAEBAEB	BAEBAEBA	.....
0x00018174	AEBAEBAE	EBAEBAEB	BAEBAEBA	AEBAEBAE	.....
0x00018184	EBAEBAEB	BAEBAEBA	AEBAEBAE	EBAEBAEB	.....
0x00018194	BAEBAEBA	AEBAEBAE	EBAEBAEB	BAEBAEBA	.....
0x000181A4	AEBAEBAE	EBAEBAEB	BAEBAEBA	AEBAEBAE	.....
0x000181B4	EBAEBAEB	BAEBAEBA	AEBAEBAE	EBAEBAEB	.....
0x000181C4	BAEBAEBA	AEBAEBAE	EBAEBAEB	BAEBAEBA	.....
0x000181D4	AEBAEBAE	EBAEBAEB	00000F41	61656100	.....A....aea
0x000181E4	01006962	00000005	79732E00	6261746D	bi.....symtab
0x000181F4	74732E00	62617472	68732E00	74727473	..strtab..shstrt
0x00018204	2E006261	74786574	61642E00	2E006174	ab..text..data..

At this point the program will again jump back to the beginning to find the next prime number, 5. Rather than letting this happen, we will single step for a few instructions and watch the value in R2. Issuing the `tr` command a few times, or alternatively `gt 18094` to execute up until the start of the `findprime` loop will let us see what happens. The value of R2 should be updated before returning to the first loop so that the next

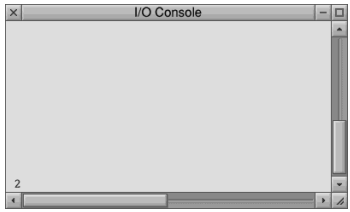
number in sequence is checked. Looking at the dissassembly of the program, and the register display, notice that R2 is set to R3 plus 1. If we were counting from 1 then this would be fine. However, because the computer counts from zero (we have already had to modify R2 to take this into account), the addition is not necessary. Instead all that is needed is to restore R2 to its starting value. The necessary change is to replace the add instruction with `mov r2,r3`.

If you have been following this article on your computer, you will have located the first three bugs in the faulty program. At this point it would be a good idea to close UCDebug, and make the necessary corrections to the source file. Edit the file s.buggy in your favourite text editor, using the correct version s.sieve as a guide to avoid introducing any further errors.

With these corrections made to the program, if you try running it you will discover that there is a further error in the output routine. So, once again, follow the instructions above to prepare the file for debugging, start UCDebug and load the program.

This time we are confident that any further errors are in the final loop which prints out the prime numbers. Scroll the code window with the mouse, or type `code 180C8` so the the relevant part of the program is visible. We will now start by setting a breakpoint at `printprime` (using `br 180E4`), on the instruction `mov r2,#0`. Start the program, so that it runs up until this point. Checking the display in the memory window will confirm that all of the composite numbers have been marked, and only the primes are clear. You can check this yourself by writing out the bit pattern of the first word. The only bits which remain clear (equal to zero) are those representing prime numbers.

The output routine begins with a short loop which scans the list for the unmarked numbers. When it finds a clear bit, the loop finishes with the value in R2. The next section then sets up registers and calls SWI &D6 (OS_ConvertCardinal2) to convert the number to a string for display. The next instruction calls SWI &02 (OS_Write0) to write this string on the output. It is worth going forwards to this point to verify that the expected string has been placed in memory. To save setting a breakpoint, use the command `gt 18120`. If you step forward by one instruction from here this string will be written to the output, and the debugger will open another window showing this:

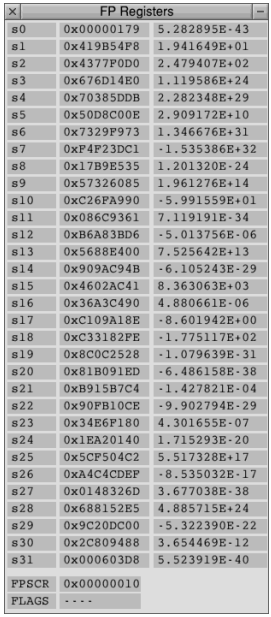


The next few instructions output a carriage-return and line-feed before restoring registers and looping back to identify the next prime number. Go back to this point with the command `gt 180E8` and check the register display. Notice that although R2 looks correct, R1 still contains the address of the output buffer, which will cause problems later. Change it back to 1 with `reg R1=1`. Now run the program up until the instruction `mov r3,r0` at &18108. The value in R2 shows that the program has correctly identified the second prime number. Failing to restore the value of R1 after the output routine would have prevented it from finding and printing the remaining primes. To correct this we can insert an instruction `mov r1,#1` just before the final unconditional branch.

This completes the debugging session, so you can now edit the file s.buggy once more to include this additional instruction. If you now assemble this source file and link it, you

should have a working program.

For those who wish to experiment further, an additional example in assembly language has been supplied with this article. The program s.factor attempts to solve the opposite problem and find the factors of a composite number. Although there are no deliberate mistakes in this code, you can nevertheless use the debugger to see how it works. It also uses a few VFP instructions to find the square root of a number. The ability to use VFP instructions and registers is one area where UCDebug currently goes further than DDT. If you try this, remember to open the VFP registers window from the iconbar menu of UCDebug first.



This article has introduced the concept of debugging tools, and used UCDebug to show the capabilities of a simple machine-level debugger. Following the worked example of finding the errors in the Sieve program has demonstrated how useful such tools can be, especially when working in assembly language. It is arguably fair to say that UCDebug fulfils its intended role as a teaching aid well, but as a practical tool for developers it has significant limitations. Anyone programming in assembly language on a regular basis will probably have access to better tools. Similarly, the inability to work with output from high-level languages and the native AIF format binaries will significantly limit its use among those developing software for RISC OS. However, the occasional programmer may benefit from an accessible introduction to debugging tools, and assembly language programming.

Nicholas Cutler *ncc25@srcf.net*