Kalpa Publications in Computing
Volume 20, 2024, Pages 35–46
Proceedings of 2024 Concurrent Processes Architectures and Embedded Systems Hybrid Virtual Conference

Varied timing, OCCAM modeling, and hardware-software equivalence in a worked IoT example

Lawrence J. Dickson[1]

Space Sciences Corporation
Lemitar, New Mexico
`larry@spacesciencescorp.com`

Abstract

CSP-based OCCAM modeling promises to bring great clarity to Internet of Things (IoT) designs that have been made vague by abstract tools. This requires an absoluteness in timing and data access that is more characteristic of the Transputer than of modern systems. However, it turns out some standard C and *n*x library tools can provide it, if human timing suffices.

IoT design is strongly dependent on use cases, and we had the fortune of a very definite project use case that drove the small OCCAM-modeled system described in the Reference. Key tools in the design include Unix sockets, ssh, and the kernel-based call select(). This paper will complete that system, begun in the referenced Fringe, and relate it to the general issues to which this technique can be expanded. Its occam modeling will be defended based on timing knowledge, and its hardware-software equivalence (HSE) will be proven.

This development must be read in conjunction with the Reference, which see. It develops the Reference to allow three or more independent racing systems. It explains the applicability of the revised approach to general IoT.

Reference

Lawrence J. Dickson: Race-Condition-Robust Hardware-Software Equivalence in *n*x. IEEE-COPA 2023 Fringe, September 25-27, 2023. RCRHSE-easychair.pdf

This [8] can be found in the github [9], along with old and current code.

Acknowledgements

The code and proofs described here and in the Reference were the work of the author, needed for a specific project. The applicability of the techniques here developed was made apparent by discussions with others in the COPA community, including Jeremy Martin, Lindsay Quarrie, and Rob Fryer. I also thank the referees for their valuable suggestions.

1 Motivation

In the Internet of Things (IoT), code needs to run the first time. The common habit of poking around a menu to find something that works is not available if there is no menu, even no screen. This run also needs to give exactly the right results, the first time. Otherwise some expensive machinery may gnash and cry out.

On the other hand, some aspects of IoT may make things easier than typical high-resolution user appeals. Generally, final outcome is expected promptly in human time, measured in seconds (even minutes), and the data that controls this outcome may be measured in bytes instead of megabytes. And when large files have to be passed, powerful processors (plural) may be available to do the necessary processing on them.

All these factors point away from centralized, abstracted programming of the multilayered type characteristic of modern handhelds like Android. At the same time, they point toward distributed design with strict communication channels such as was pioneered by Communicating Sequential Processes (CSP) [7], the computer language OCCAM [1], and the classic Transputer [2] hardware. A question arises whether such design is accessible using modern tools and hardware. The answer is affirmative if Hardware-Software Equivalence (HSE), an OCCAM/Transputer characteristic ([1] page 71), can be achieved.

2 Use Case

There is one fact about IoT which I didn't mention above, and which led me to an answer to that question. This is that IoT design is strongly oriented to real use cases (e.g. [5]). I was guided by a real use case to design a simple client/server system in C that achieves HSE. Using it, I got true synchronicity at key points of an independently clocked system involving three major processors.

The project used a quad-camera system to survey a toolbox drawer and detect whether tools therein are all there, undamaged and correctly positioned. The three major processors are two Raspberry Pis, Camera (C) and Rosebrock (R), and an AI-capable tablet (T). Though both Raspberry Pis, C and R are different, being optimized for different purposes, C for camera input and R for stitching code named after Arthur Rosebrock, author of PyImageSearch [4], a well-maintained library of computervision and related programs, largely in Python. The Tablet uses AI to interpret the stitched picture against a template, to decide whether tools are undamaged and in their correct slots in the drawer.

The HSE tool, partially completed in the Reference (the project was underway at that time), uses scriptable program calls in *n*x, including Linux and the tablet-capable Termux. Its server is trigonal.c and associated clients are trigcsr.c, trigrhub.c, and trigtsr.c. Of these, trigrhub.c is newly published here. These use Unix sockets [6], and achieve HSE through a simple ssh call that causes two Unix sockets, one on each hardware device, to behave the same as a single socket on the software of one device. Provable and prompt synchronization is achieved by using select(), the C library call that adjudicates file descriptor races. The only assumption is that the server and client processes are each sequential in the code that is triggered by a select() win, so that each select() win acts as a true interrupt, not interleaved with the execution of any other select() win on the same device.

The result of this design is predictable behavior in passing of message data, and (using the compount timing anchor) true and prompt synchronicity. The HSE of this code is so strong that both the software and hardware versions use the same binaries for both server and clients. In this they resemble Transputer bytecode.

The critical race is between a client write on one process (or device) and the corresponding client read on another process (or device). The server offers a one-deep buffer, resembling an atom of Short Message Service (SMS [3]) in telephony, and adjudicates the race using select(). In the case of a client endpoint (only ever communicating with one other client) as reader, the race is between server accepting that client read and server receiving the communication partner client's data write. In the case of a client hub (multiple client communication partners) as reader, the race is between that client's write of an identifier of its partner and the partner client's data write.

The result for a client-to-client communication is buffering in the case the writing client wins, and instant pass-through at write time in the case the reading client wins. The key structure is a timing anchor (see Reference), in which one client communicates a trigger to a second, and the second, after an action, communicates an acknowledge to the first. The action is expected to be relatively time-consuming, and is typically an scp pull of some multi-megabyte data file or files.

See Figure 1 for how this works in practice.

2.1 Discussion of Figure 1

Notice that the communications between Camera (C) and Rosebrock (R), like those between R and Tablet (T), consist of arrows to the right (Message) followed by wide arrow to the right (Data Query), and arrow to the left (Acknowledge). These together form a "timing anchor" (see Reference for more detail). The Message has two possible race outcomes: read wins (direct arrow to read with wait), or write wins (arrow to server buffer, followed by arrow to read with no wait). Due to the time spent with Data Query, the Acknowledge is always read wins, and thus synchronizes the two devices.

Figure 2 (excerpted from Figure 1) shows timing anchors of both race types. Because read always wins the Acknowledge, the bottom arrow is always the synchronization point.

The occam pseudocode for a single timing anchor (from C to R) is as follows, with C code to left and R code to right, and declarations not shown:

```
PAR                                SEQ
  mcr ! Message                      mcr ? Message
  -- Data must pre-exist PAR         -- Data fetch command constructed
  qcr ! Data                         qcr ? Data
  arc ? Acknowledge                  arc ! Acknowledge
```

It seems odd that I represent these by a PAR on one side and a SEQ on the other, but the reason is that I impose the rule that *pseudocode must imply information transmission timing knowledge or lack of knowledge.* The left side returns from Message transmission instantly (due to server buffering if there is no receiver read pending), but C does not know its Message is transmitted to R until the Acknowledge arrives. Similarly, since the Data Query is implemented by an scp pull (a call in R), C does not know that has happened until the Acknowledge has arrived. The PAR expresses these uncertainties, and the OCCAM rules thus insist that all information sent (Message and Data) must pre-exist the PAR and remain untouched by C during the lifetime of the PAR.

By contrast, the SEQ is appropriate for the role of R. As soon as its read of Message is complete, it knows that data is transmitted. It can use the data received in Message to construct its Data Query (e.g. the names and directory locations of the big files of Data to be pulled by its scp call). After that is done, and its capture of Data is complete, it sends its Acknowledge.

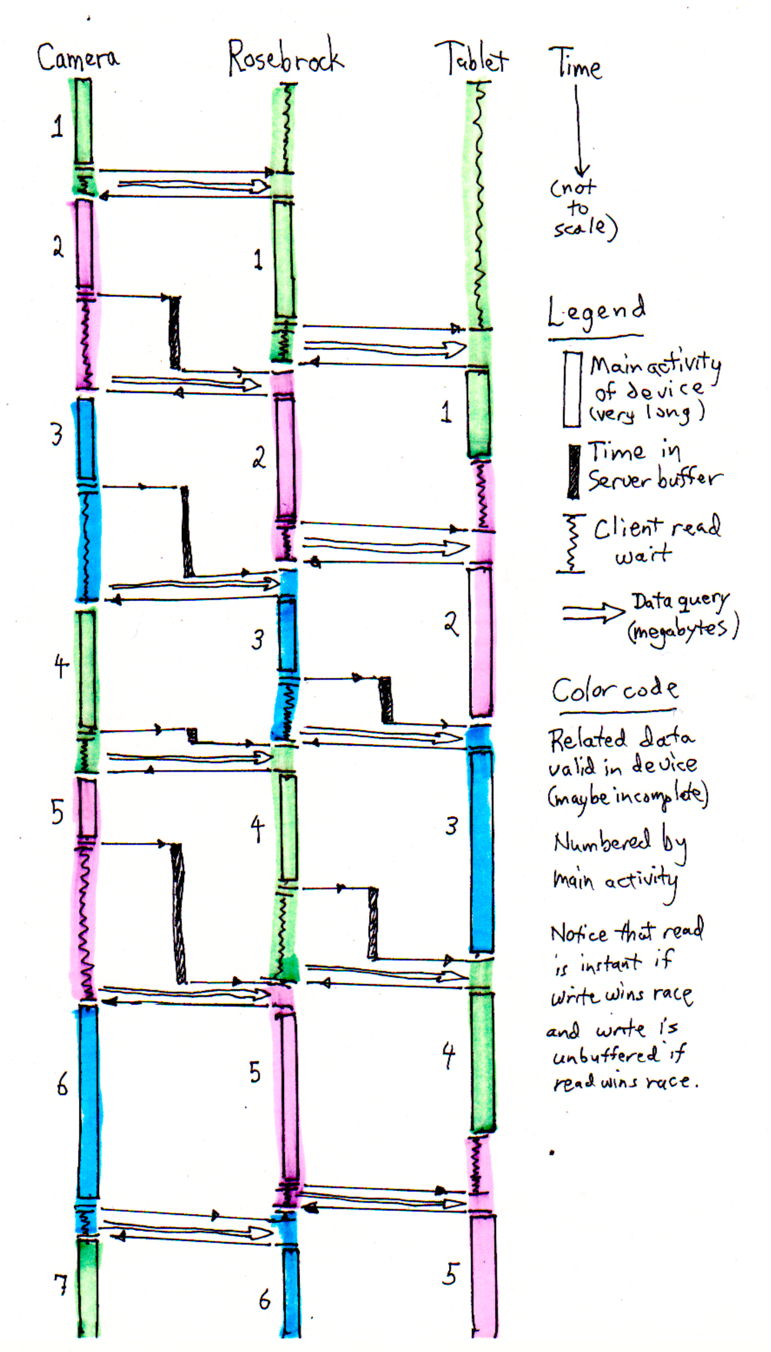
Camera
Rosebrock
Tablet
Time
(not to scale)
Legend
Main activity of device (very long)
Time in Server buffer
Client read wait
Data query (megabytes)
Color code
Related data valid in device (maybe incomplete)
Numbered by main activity
Notice that read is instant if write wins race and write is unbuffered if read wins race.

Figure 1: Three parallel devices controlled by timing anchors

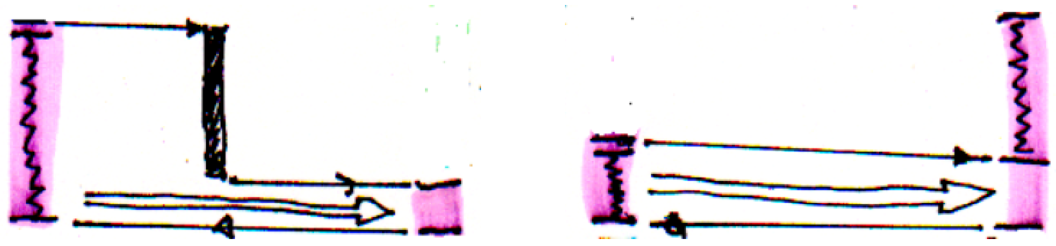

Figure 2: Timing anchor, both race types shown

The table that follows shows the client calls actually used (see Reference, noting the innovation of trigrhub, described below, since R is a hub):

Table 1: A timing anchor from C to R

process C script	process R script
`trigcsr '!CRthename'`	`trigrhub '?C'`
	`action`
`trigcsr`	`trigrhub '!RCA'`

The action is an scp pull. The argument to the first trigrhub tells which communication partner is to write to this hub this time. The second trigcsr call is deceptively placed, as C executes this read call immediately after the first call to trigcsr is completed, which because of the server buffering is essentially instantaneous. This means the second trigcsr wins the race of the Acknowledge and waits for the Acknowledge from R, thus synchronizing the two devices.

2.2 Hardware-software equivalence

In the above discussions I have called C, R, and T ”devices”, thus referring to the hardware option. I have not specified where the server runs. However, as detailed in the Reference, any or all of the clients may be software processes running on the same system as the server, so that the total count of actual devices may be anything from 1 to 4, including the server.

The server runs as a daemon (i.e. it is started before any script using the clients, as seen in Table 1, and it does not close until after any such script is done). The communication channels are all soft (Unix) sockets. The case where a hardware link is used converts two such sockets, one on each of the communicating systems, to a single link using a command like this:

```
ssh -nNT -L /tmp/9Lcamera.socket:/tmp/9Lcamera.socket pi@192.168.10.3
```

This also runs as a daemon on the client system, referencing the real IP address of the server. Notice in this case both soft sockets have the same name. As a consequence, the binary of both server and client is unchanged whether hardware or software implementation is used - just like classic Transputer binaries! (See the for loop near the top of the next section to see how to generate the HSE binaries.)

3 Hub-Upgrade: Adding a Hub to the Reference

The code is available for inspection and easy compilation at https://github.com/SpaceSciencesCorp/Trigonal/releases/tag/v1.0.0

```
for i in trigonal trigcsr trigrhub trigtsr ; do
  gcc -c $i.c
  gcc -o $i $i.o
done
```

For practical usefulness, some clients in this system must be hubs (able to talk to more than one other client). This was missing in the code demonstrated in September 2023 at IEEE-COPA 2023. My efforts since then have been aimed at modifying the server to deal with hubs.

One addition was key: the hub read has to identify its communication partner. Given the timing anchor design, this is always possible, and it proved to suffice. The pending Boolean and associated one-deep buffer were both replicated by the number of communication partners, thus permitting unique responsiveness in the case of the early write. And a new variable with a unique nonzero value for each communication partner allows cooperation with the late write, using the fact that each client call set (including the hub client) is required to be part of a purely sequential script.

In the code actually written, the client call sets are endpoints C and T, and hub R which communicates with the two endpoints.

The modifications in the code of the Reference are contained in an Appendix to this paper.

3.1 Setup

See the Reference.

3.2 Run

See the Reference for general instructions. Changes will be noted here.

In C2 run:

```
./trigcsr '!CRABCD'
```

And in R2 run (using the new hub client pointing back to C):

```
./trigrhub '?C'
```

The message will pass through and appear in R2. It does not matter which of the clients is triggered first. Thus, you can generate an acknowledge as follows:

In C2 run:

```
./trigcsr
```

(this is an endpoint read as in the Reference.)

And in R2 run:

```
./trigrhub '!RCA'
```

(the hub write is the same as an endpoint write.)

The two characters immediately after the exclamation point are important, as the program reads them and interprets them as message source and target. Similarly, in the hub read, the character immediately after the question mark is important, as the program reads it and interprets it as message source.

To shut down, hit terminal C1 with a Ctrl-C to kill the `ssh -nNT` and follow that in terminal C1 with

```
rm /tmp/9Lcamera.socket
```

Watch terminal R1. It will probably say 2 connections left. Finish the others from terminal R2 with

```
./trigtsr DOWN
./trigrhub DOWN
```

If it didn't say 2 connections left, you will also need

```
./trigcsr DOWN
```

and ./trigonal will quit.

3.3 Restrictions and scripting

The techniques developed here make use of the behavior of *n*x sockets, which establish a bidirectional communication (called a *socket link*) between *n*x processes. Here we solely use "Unix sockets" (soft sockets) and let the ssh tunnel take care of any hardware connection. Thus, the processes may be on the same hardware (one socket) or on different hardware (two sockets connected by one ssh tunnel). The behavior is equivalent.

Each socket link is driven solely by the client programs `trig?sr` for an endpoint, or `trig?hub` for a hub, called on each process in a single sequential script or equivalent. The calls are grouped in *timing anchors*, two calls each side per anchor, with no interleaving, according to the scheme shown above in Table 1 (subsection 2.1).

Here `thename` is some ASCII information useful, for instance, in building filenames, and `action` is some appropriate shared action, such as an scp pull of a few megabytes of data from C to R. The calls to `trig?sr` without a parameter, or `trig?hub` with a parameter starting with a question mark, are *client reads,* while those with a parameter starting with an exclamation point are *client writes.* Because of the sequential restriction, the horizontal pairs in the table always communicate with each other. See the Reference for further discussion and a graphic.

3.4 Robust communication

See the Reference.

3.5 Multiple timing anchor sequence troubles

This is the problem solved in this Hub-Upgrade.

The distinction here is between a *client endpoint,* which can only ever communicate with one other client, and a *client hub,* which can communicate with two or more other clients. The old code dealt only with client endpoints, and was thus restricted to two clients at a time with no overlap.

In the old code, a client (i.e., Rosebrock) that could read from more than one other client had no way of knowing which to take data from. Since in order to set up a timing anchor, the programmer had to know, the solution was for the programmer to explicitly communicate this knowledge in the call. This applies to read only, hence the use of the CSP/occam question mark as the initial was appropriate.

The modification in trigonal.c expects this extra step in the hub read case, and doubles up some variables to allow both sides to overlap. A new variable, source_Rosebrock, lets the mated writing client know to check if it is truly late with a preceding hub read asking for its side. Any early hub read will block until the correct mating write comes along, even if other activity is happening before then.

4 Usage example

This section gives a pseudocoded example of a practical use of this system.

4.1 Crude pseudocode

Here the timings are only approximate, as the three processes, whether soft (bash terminal) or hard, have independent timing. (In real usage, each process is hard, the first two being Raspberry Pis and the third an Android tablet with Termux running.) `CR(n)` is the n-th timing anchor between C (or Camera) process and R (or Rosebrock) process. `RT(n)` is the n-th timing anchor between R process and T (or Tablet) process. The sequential main program that runs on C is `Cameras(n)` while R has `Stitch(n)` and T has `AI(n)`. Each program runs on the order of a minute, while each timing anchor requires on the order of a second.

The symbol `{ K || L || M }` refers to programs K, L, and M running on C, R, and T approximately simultaneously. Here a `-` in place of a program means nothing is happening on that process.

```
{ Cameras(1) || - || - }
CR(1)
{ Cameras{2} || Stitch(1) || - }
RT(1)
CR(2)
{ Cameras(3) || Stitch(2) || AI(1) }
RT(2)
CR(3)
{ Cameras(4) || Stitch(3) || AI(2) }
 . . .
{ Cameras(N) || Stitch(N-1) || AI(N-2) }
RT(N-1)
CR(N)
{ - || Stitch(N) || AI(N-1) }
RT(N)
{ - || - || AI(N) }
```

4.2 Partial scripts

In this subsection, I will partially explicate the above pseudocode as bash scripting, one (sequential) script running on each process. It isn't quite accurate, as I merely put a note on the action in the `Pull` lines, which would actually require some sed scripting and the construction of an scp pull of the file shown. What is shown below is only up through the line `{ Cameras(3) || Stitch(2) || AI(1) }`.

C process

```
Cameras 1
./trigcsr '!CRphotos1.zip'
./trigcsr
Cameras 2
./trigcsr '!CRphotos2.zip'
./trigcsr
Cameras 3
```

R process

```
./trigrhub '?C'
Pull photos1.zip
./trigrhub '!RCA'
Stitch 1
./trigrhub '!RTpicture1.png'
./trigrhub '?T'
./trigrhub '?C'
Pull photos2.zip
./trigrhub '!RCA'
Stitch 2
```

T process

```
./trigtsr
Pull picture1.png
./trigtsr '!TRA'
AI 1
```

5 Conclusion and future work

The timing anchor developed here uses classic, well-proven tools in the *n*x world, such as select(), sockets, and ssh/scp, to perform three tasks between a source device/process and a destination device/process, when the source concludes with producing significant data for the destination to work on: (a) inform the destination when the source's data is stable, (b) insure this data remains stable while being transmitted to a ready destination, and (c) note the time synchronously on both devices/processes when the destination has the data. It does this without any assumptions about shared dates. clocks, or timing.

This eliminates the need for shared dates or clocks among multiple systems, and by using the timing anchor acknowledge to send a date/time, can realign those that have become unaligned. It tackles a big problem in IoT, which gets worse as the number of devices increases: date skew, sometimes upon reboot, and clock drift (these can result in systems seizing up due to future dates on files, for example). If all devices have nearly identical timing, occasional synchronization using timing anchors can assure nearly identical dates/times on all devices.

Future development will allow more complex networks than the current one-hub, one-server, two-endpoint arrangement. It will be vastly eased by the hardware-software equivalence in this system, which will make HSE possible in wider fronts in IoT, allowing cleanly designed small devices/processes to perform individual, reliable, provable tasks in synchrony. It permits clean analogy with classic CSP/occam/Transputer systems that pulled far beyond their weight class as far back as the 1990s.

Bibliography

[1] INMOS Ltd. *occam 2 Reference Manual.* Prentice Hall, 1988. http://www.transputer.net/obooks/obooks.asp.

[2] INMOS Ltd. *Transputer Instruction Set, a compiler writer's guide.* Prentice Hall, 1988. http://www.transputer.net/iset/iset.asp.

[3] Wikipedia. *SMS.* Wikimedia Foundation Inc, 2024. https://en.wikipedia.org/wiki/SMS.

[4] Adrian Rosebrock. *Pyimagesearch AI & Computer Vision Programming.* PyImageSearch, 2024. https://pyimagesearch.com.

[5] U. Franke, H. Fritz, A. Kuehnle, and J. Schick. Transputers on the Road. In *Transputer Applications and Systems, Proceedings of World Transputer Congress 1993*, volume 1, pages 1–17. IOS Press, Amsterdam, The Netherlands, 1993.

[6] Neil Matthew and Richard Stones. *Beginning Linux Programming.* Wrox Press, Birmingham, 1997. ISBN: 1-874416-68-0.

[7] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, London, 1985. ISBN: 0-131-53271-5.

[8] Lawrence J. Dickson. Race-Condition-Robust Hardware-Software Equivalence in *n*x. In *IEEE-COPA 2023 Fringe, September 25-27, 2023*, RCRHSE-easychair.pdf

[9] Lawrence J. Dickson. Trigonal code history, Space Sciences Corporation, 2024. https://github.com/SpaceSciencesCorp/Trigonal/releases/tag/v1.0.0

Appendix

The Appendix consists of the code diff. Only two diffs are required: that of digonal.c (the old trigonal.c) with the current trigonal.c, and that of trigrsr.c (the Rosebrock endpoint client) with trigrhub.c (the Rosebrock hub client).

For more information, and the old code, see the Reference.

trigrsr.c trigrhub.c diff

```
--- trigrsr.c   2023-04-21 08:34:53.000000000 -0700
+++ trigrhub.c  2023-10-06 14:15:26.000000000 -0700
@@ -7,9 +7,9 @@
 #include <unistd.h>
 #include "uscocrlf.h"
 /* Client:
- * Connects immediately
- * If argc>1 it sends argv[1] contents to server
- * Otherwise it awaits buffer from server and outputs it to stdout
+ * Connects immediately, expects argc > 1 (HUB)
+ * It sends argv[1] contents to server
+ * If argv[1][0]=='?' it awaits buffer from server and outputs it to stdout
  * Then closes the connection
  */
 int
@@ -46,7 +46,7 @@
         fprintf(stderr, "connect returned %d\n", ret);
     }
     /*}}} */
-    /*{{{ Send argv[1] or receive and output to stdout*/
+    /*{{{ Send argv[1]; sometimes receive and output to stdout*/
     /* Send if argc>1. */
     if (argc > 1) {
         ret = write(data_socket, argv[1], strlen(argv[1]) + 1);
@@ -57,20 +57,24 @@
             fprintf(stderr, "write returned %d, content %s\n", ret, argv[1]);
         }
         sleep(1);
-    } else {
         /* Receive result. */
-        ret = read(data_socket, buffer, sizeof(buffer));
-        if (ret == -1) {
-            perror("read");
-            exit(EXIT_FAILURE);
-        } else {
-            /* Ensure buffer is 0-terminated. */
-               buffer[sizeof(buffer) - 1] = 0;
-               fprintf(stderr,
-                 "read returned %d, buffer size with trailing null %d, content %s\n",
-                 ret, strlen(buffer)+1, buffer);
-               printf(buffer);
+           if (argv[1][0]=='?') {
+               ret = read(data_socket, buffer, sizeof(buffer));
+               if (ret == -1) {
+                   perror("read");
+                   exit(EXIT_FAILURE);
+               } else {
+                   /* Ensure buffer is 0-terminated. */
+                   buffer[sizeof(buffer) - 1] = 0;
+                   fprintf(stderr,
+                     "read returned %d, buffer size with trailing null %d, content %s\n",
+                     ret, strlen(buffer)+1, buffer);
+                   printf(buffer);
+               }
            }
+       } else {
+           fprintf(stderr, "REQUIRES ARGUMENT\n");
+           exit(EXIT_FAILURE);
        }
        /*}}}  */
        /* Close socket. */
```

digonal.c trigonal.c diff

```
--- digonal.c   2023-04-21 08:55:16.000000000 -0700
+++ trigonal.c  2023-10-07 12:04:29.000000000 -0700
@@ -16,11 +16,13 @@
        int connection_camera, connection_Rosebrock, connection_tablet;
        int data_camera, data_Rosebrock, data_tablet, fd_size;
        int current_camera, current_Rosebrock, current_tablet;
-       int pending_camera, pending_Rosebrock, pending_tablet;
+       int pending_camera, pending_CR, pending_TR, pending_tablet;
+       int source_Rosebrock;
        int nconnecs, result_camera, result_Rosebrock, result_tablet, retval;
        char buffer[BUFFER_SIZE];
        char todo_camera[BUFFER_SIZE];
-       char todo_Rosebrock[BUFFER_SIZE];
+       char todo_CR[BUFFER_SIZE];
+       char todo_TR[BUFFER_SIZE];
        char todo_tablet[BUFFER_SIZE];
        /*{{{  Create connection socket, bind, listen: CAMERA*/
        /* Create local socket. */
@@ -142,7 +144,8 @@
        current_camera = connection_camera;
        current_Rosebrock = connection_Rosebrock;
        current_tablet = connection_tablet;
-       pending_camera = pending_Rosebrock = pending_tablet = 0;
+       pending_camera = pending_CR = pending_TR = pending_tablet = 0;
+       source_Rosebrock = 0; /* initialized to OFF */
        valid_camera = 1;
        valid_Rosebrock = 1;
        valid_tablet = 1;
@@ -216,7 +219,8 @@
                    /* Handle commands. */
                    if (!strncmp(buffer, "DOWN", strlen(buffer))) {
                        down_flag = 1;
-                   } else if (current_Rosebrock == data_Rosebrock) {
+                   } else if ((current_Rosebrock == data_Rosebrock) &&
+                       (source_Rosebrock == (int)'C')) {
                        /*{{{  Send result.*/
                        ret = write(data_Rosebrock, buffer, strlen(buffer)+1);
                        if (ret == -1) {
@@ -227,13 +231,14 @@
                        }
                        /* Close socket. */
                        close(data_Rosebrock);
+                       source_Rosebrock = 0;
                        current_Rosebrock = connection_Rosebrock;
                        /*}}}  */
                    } else {
-                       strcpy(todo_Rosebrock, buffer);
-                       fprintf(stderr, "camera to Rosebrock pending content %s\n",
-                         todo_Rosebrock);
-                       pending_Rosebrock = 1;
+                       strcpy(todo_CR, buffer);
+                       fprintf(stderr, "camera to CR pending content %s\n",
+                         todo_CR);
+                       pending_CR = 1;
                    }
                }
                if (detect_end) {
@@ -270,23 +275,10 @@
                        fprintf(stderr, "accept returned %d\n", data_Rosebrock);
                        current_Rosebrock = data_Rosebrock;
                        if (data_Rosebrock >= fd_size) fd_size = data_Rosebrock + 1;
-                       if (pending_Rosebrock) {
-                           /*{{{  Send result.*/
-                        ret = write(data_Rosebrock, todo_Rosebrock, strlen(todo_Rosebrock)+1);
-                        if (ret == -1) {
-                            perror("pending␣Rosebrock␣write");
-                            exit(EXIT_FAILURE);
-                        } else {
-                            fprintf(stderr, "pending␣Rosebrock␣write␣returned␣%d\n", ret);
-                        }
-                        pending_Rosebrock = 0;
-                        detect_end = 1;
-                        /*}}}  */
-                    }
                 }
                 /*}}}  */
             } else { /* data_Rosebrock */
-                /*{{{  Wait for next data packet, until DOWN.*/
+                /*{{{  Wait for next data packet, until DOWN. Default detect_end.*/
                 /* Wait for next data packet. */
                 ret = read(data_Rosebrock, buffer, sizeof(buffer));
                 if (ret == -1) {
@@ -309,6 +301,45 @@
                 /* Handle commands. */
                 if (!strncmp(buffer, "DOWN", strlen(buffer))) {
                     down_flag = 1;
+                } else if (buffer[0] == '?') { /* client read */
+                    detect_end = 0; /* Default for read early */
+                    fprintf(stderr, "Rosebrock␣client␣read␣source␣%c\n", buffer[1]);
+                    if (buffer[1] == 'C') { /* Source is C */
+                        source_Rosebrock = (int)'C';
+                        if (pending_CR) {
+                            /*{{{  Send result.*/
+                            ret = write(data_Rosebrock, todo_CR, strlen(todo_CR)+1);
+                            if (ret == -1) {
+                                perror("pending␣CR␣write");
+                                exit(EXIT_FAILURE);
+                            } else {
+                                fprintf(stderr, "pending␣CR␣write␣returned␣%d\n", ret);
+                            }
+                            pending_CR = 0;
+                            source_Rosebrock = 0;
+                            detect_end = 1;
+                            /*}}}  */
+                        }
+                    } else if (buffer[1] == 'T') {
+                        source_Rosebrock = (int)'T';
+                        if (pending_TR) {
+                            /*{{{  Send result.*/
+                            ret = write(data_Rosebrock, todo_TR, strlen(todo_TR)+1);
+                            if (ret == -1) {
+                                perror("pending␣TR␣write");
+                                exit(EXIT_FAILURE);
+                            } else {
+                                fprintf(stderr, "pending␣TR␣write␣returned␣%d\n", ret);
+                            }
+                            pending_TR = 0;
+                            source_Rosebrock = 0;
+                            detect_end = 1;
+                            /*}}}  */
+                        }
+                    } else {
+                        fprintf(stderr, "Rosebrock␣client␣read␣source␣ILLEGAL.\n");
+                        exit(EXIT_FAILURE);
+                    }
                 } else if (buffer[2] == 'C') {
                     if (current_camera == data_camera) {
                         /*{{{  Send result.*/
@@ -424,7 +455,8 @@
                 /* Handle commands. */
                 if (!strncmp(buffer, "DOWN", strlen(buffer))) {
                     down_flag = 1;
-                } else if (current_Rosebrock == data_Rosebrock) {
+                } else if ((current_Rosebrock == data_Rosebrock) &&
+                    (source_Rosebrock == (int)'T')) {
                     /*{{{  Send result.*/
                     ret = write(data_Rosebrock, buffer, strlen(buffer)+1);
                     if (ret == -1) {
@@ -435,13 +467,14 @@
                     }
                     /* Close socket. */
                     close(data_Rosebrock);
+                    source_Rosebrock = 0;
                     current_Rosebrock = connection_Rosebrock;
                     /*}}}  */
                 } else {
-                    strcpy(todo_Rosebrock, buffer);
-                    fprintf(stderr, "tablet␣to␣Rosebrock␣pending␣content␣%s\n",
-                      todo_Rosebrock);
-                    pending_Rosebrock = 1;
+                    strcpy(todo_TR, buffer);
+                    fprintf(stderr, "tablet␣to␣TR␣pending␣content␣%s\n",
+                      todo_TR);
+                    pending_TR = 1;
                 }
             }
             if (detect_end) {
```