

Non-preemptive Occam in DSP real-time system

The parallel programming language occam has successively been used to program a real-time system for a Texas TMS320C32 DSP. 28000 lines of machine-generated ANSI C have been generated from the occam sources in a mixed language system, together with 17000 lines of hand-written ANSI C. The freely available SPoC (Southampton Portable occam Compiler) has been used to generate the C sources. The compiler comes bundled with a non-preemptive scheduler. Descheduling points have been inserted by SPoC at all occam constructs where a process may change external state (mainly communication). This application shows that a true real-time system may be built with this strategy, it also has the potential to implement harder real-time responsiveness than we required. Observe that in occam, semaphores and monitors are non-existent, this functionality is semantically defined in the overlying CSP definition. SPoC may be ported to any system that has a C compiler - Autronica ported SPoC to the DSP. The application we have developed is a new Autronica radar-based fluid level gauge, branded GL-100.

In the occam language every statement is considered to be a process. The two statements $x := 1$ and $y := 2$ enclosed in a PAR start two processes that will run, stop and then be joined with the parent process. Through the freeware SPoC occam to ANSI-C translator, this is the process granularity that Autronica, a Norwegian company, has had available in the development of a new real-time embedded system based on Texas Instrument's TMS320C32 DSP. The process scheduler and run-time system come bundled with SPoC, and non-preemptive scheduling is supported.

ARCHITECTURE

The new Autronica GL-100 radar-based fluid level gauge now being delivered to the world's oil tankers replaces the 1984 vintage GL-90 installed aboard some 330 ships, totaling 5700 tanks, plus 1000 processing units to sub-contractor for land-based tanks.

The main external data producer is a radar antenna looking down into tanks, with an X-band FMCW signal being heterodyned to If for delivery to our board. See fig.1. The computation/communication ratio of GL-100 is rather high. Computation is done by the signal processing routines, which synthesize reflection diagrams from the sensor's measurement- and reference signals, and find the reflection of choice. The equivalent of several 2K samples FFT's has to be computed for every produced value. Communication in this context also covers data-acquisition. Samples are input via DMA, this steals only a fraction of the bus bandwidth. We serve an end-of-DMA interrupt several times per second. Communication with a local area network is done over dual-port memory (where a Lon-controller is sitting at the other side) according to a "ping-pong" protocol [1]. This communication is polled and can handle 5-10 packets per second. It includes inputs for configuration and ship's tilting, and output of results. Overhead for internal bookkeeping, like serving the timer interrupt every 1 ms (for occam TIMERS) and

process switching and communication, also loads the processor lightly. The final result is an instrument that delivers the fluid level of oil-tanks with a 2-mm accuracy.

COMMUNICATING STATE-MACHINES

Like any other real-time system our system may be viewed as a set of communicating state-machines or processes. Our state-machines communicate over unidirectional point to point channels according to defined protocols. Communication is synchronous, blocking and unbuffered - saying the same thing with three phrases. Ripping up a channel and inserting a separate buffer process must exclusively program buffering. Our state-machines are controlled by a simple scheduler that always runs the longest awaiting

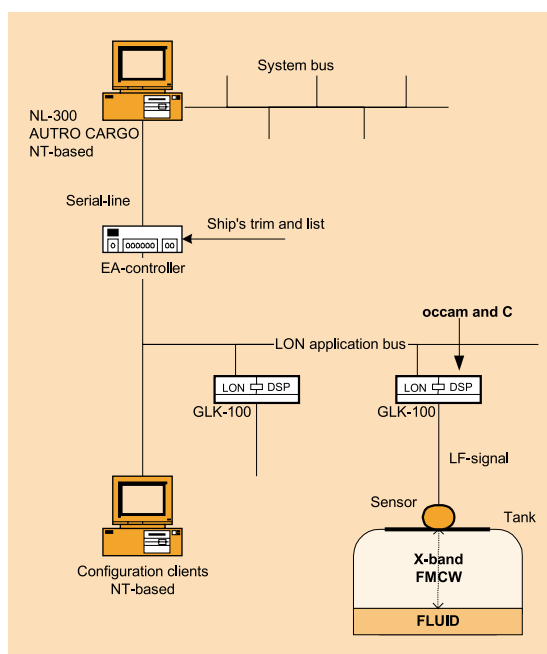


Figure 1. Simplified system block diagram

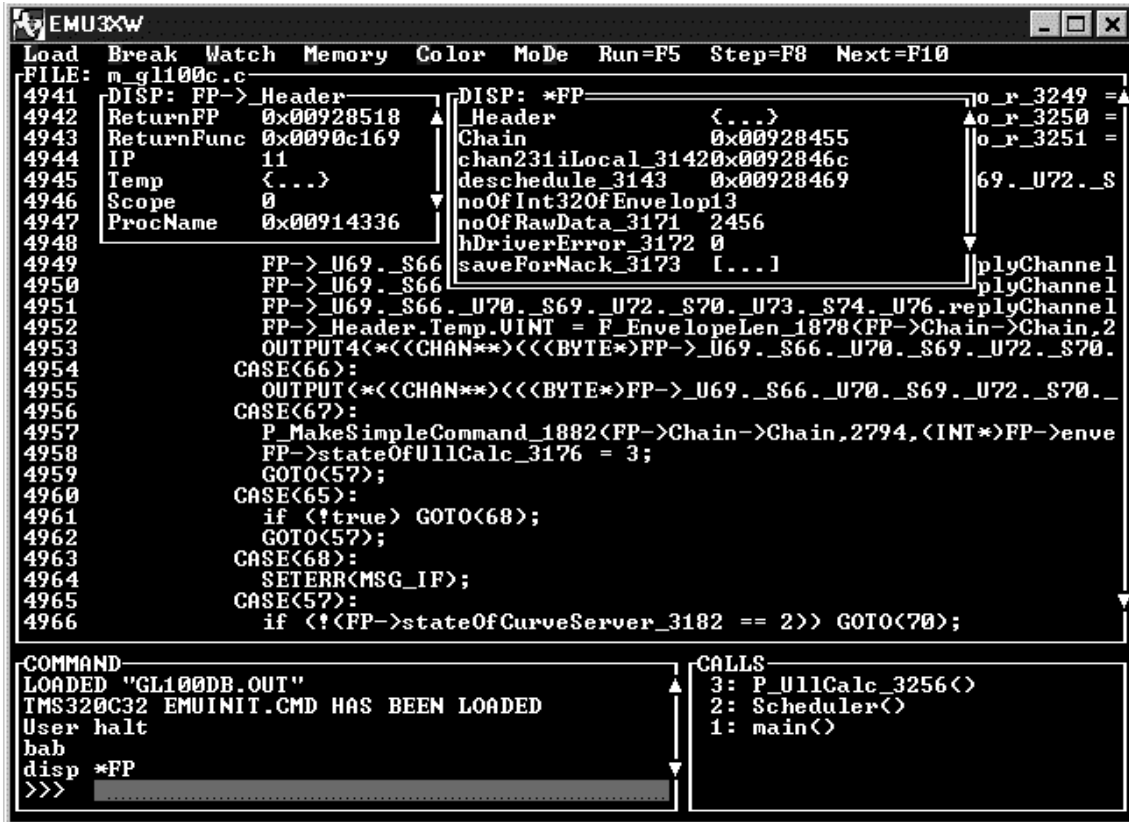


Figure 2. Texas Emulator window displaying variables of process "UllCalc".

process with the highest priority, or idles waiting for an interrupt if all processes are blocked. This is done in an orderly fashion, the only place where a process is scheduled is at a call in a single line in the scheduler. The only place a process is willingly returned from is in the scheduler's following line. If a process needs to communicate with an other process, this is done by initialising the proper common data structures and return to the scheduler to let it decide what to do next. The same happens when a process needs to create new processes: the scheduler is always returned to when proper data structures have been set. Mutual exclusion of common resources is guaranteed through this non-preemptive scheduler because we don't yield in random places.

STATE-MACHINES ARE MACHINE-GENERATED

Both constructions of the state-machines and yielding are out of our control. SPoC generates them, with our 60-80 occam processes as input. Occam is a simple language where processes and channels are first-rate citizens. As such their behaviour is closely guarded: processes can dynamically communicate only over shared channels, and a sender and receiver must agree on the protocol. The language is strongly typed, and both channel and variable usage are checked. A set of usage rules are applied. F.ex. public values may be shared between processes as long as they are constant during the lifetime of all shared processes, a property that is checked by the compiler. Mutual read is allowed provided none of the processes do a write. Processes may be parameterised through such (glob-

al variable) invariants or through a standard parameter regime. Channels may also be parameterised.

In such a safe message-passing world there is no problem to have scheduling points at semantically defined points (read: in some of the predefined functions or macros). Scheduling is present in the functions for start and stop of processes, communication and waiting on a timer. Occam can be implemented on top of a non-preemptive or preemptive scheduler, the foundation is CSP [2] and is suitable for either kind. It has no semaphores or monitors so we don't need to worry about correct usage. We analyse what we want to do, design the processes and message-passing, implement it in occam, push a button and see the programs appear, process for process, in the C-file. Analysis, design and implementation, prototyping and "real" typing are not as distinguishable as they would be in a product where we design some 'x' and see some unrecognisable 'y' at the coding level.

Observe that "user" state-machines are not machine generated.

SOUTHAMPTON PORTABLE OCCAM COMPILER

SPoC is a freeware system that was implemented as part of an Esprit-project [3]. It is based on a compiler construction toolkit from University of Karlsruhe. The accompanying implementation-note describes quite well the design criteria and implementation. SPoC is quite easily ported to other architectures: only two files need to be modified. Basically, one needs to define mapping of occam's primitive data-types to corresponding ANSI-C types. A timer interrupt routine has to

Ad EONIC Systems

be written, since occam does handle time. Also, other interrupt routines need to be plugged in, and sufficient buffering to make the interrupt-system acceptable for non-preemptive scheduling needs to be written. In our case, the dual-port and DMA automatically did this buffering for us. Connecting the producing interrupt routine to the sending side of a channel was not originally supported, we solved this in an acceptable way in two days.

Autronica has for two years now had a support contract with Southampton University. Updates have been placed back to public domain. The product has been quite stable and on the first sailing ship we don't know about any bugs coming from code-generation. The only stumbling stone was occam abbreviation (see below) of two-dimensional array into one-dimensional array, this had to be avoided.

SPoC builds quite readable C-code, and it was just a matter of getting used to running the system in the debugger. Viewing the variables of any process we only ever need to write "disp *FP" (see fig.2). SPoC builds a tree of structs and unions, hold the top-level struct and you may traverse to any process' data structure from there.

Observe layout of generated code and state machine. At time of writing a single file containing all 28000 lines

of code is generated (in a few seconds) from the bundle of occam files, since we have not been able to use SPoC's present separate compilation scheme. We are expecting to see this change.

PAR, ALT, REPLICATION AND ARRAY SLICING

Occam may be categorised as object-based since there is no object inheritance or polymorphism. Direct message passing (i.e. command-reply, client-server) goes for method calls. Nothing is ever exported up or beside in the process hierarchy and strict usage rules are enforced for "exporting" data down in the process hierarchy. Occam is simple to learn, the BNF description is only 6 pages including multi-processor configuration (not supported by SPoC) [4]. The fact that it has no pointers and has compiler-enforced usage rules makes it a safe language. It is fast, see the appendix for our speed measurements. And most importantly it is based on a sound theory (CSP) and is scalable to any number of processes.

However, the true story is that previously Inmos, now SGS-Thomson, is stopping both the transputer for which occam was first designed, and the occam support. But the sources of their occam compiler have been the basis of University of Kent's KRoC ports to

APPENDIX

The example does 1000 communications of INT's via OverWrBuff.INT. Each INT is sent 3 times, and one internal acknowledge is sent from the Output process back to Input process of OverWrBuff.INT. Our 40 MHz TMS320C32 with one wait-cycle and 16-bits bus (2 accesses per word) takes 504 ms to complete these 4000 communications, including 4000 process schedulings. Texas compiler is set to lowest optimisation level (o0) - not that it matters much. Occam process is high priority and no other process is running. This is 126 μ s per communication, on the average. Compensating for wait-cycles and bus width this should be closer to 30-40 μ s per communication. Dot is allowed in occam names:

```
PROC OverWrBuff.INT (CHAN OF INT in, out)
  PROC Input (CHAN OF INT input, CHAN OF BOOL acknowledged, CHAN OF INT output)
    INT buffer, noOfPackets:
    BOOL waitingForAcknowledge:
    SEQ
      noOfPackets := 0
      waitingForAcknowledge := FALSE
      WHILE TRUE
        PRI ALT
          ((waitingForAcknowledge = FALSE) AND (noOfPackets > 0)) & SKIP
        SEQ
          #C // This communication will never block:
          output ! buffer
          noOfPackets := 0
          waitingForAcknowledge := TRUE
    BOOL ack:
    acknowledged ? ack
      waitingForAcknowledge := FALSE
    input ? buffer
      noOfPackets := noOfPackets + 1
```

CONTINUED ON PAGE 87

several other architectures [5]. They show tremendous process switching times, and have implemented some extensions to the language. And SPoC is around, so we do have compilers.

I will briefly sum of some other facets of occam. PAR starts processes that communicate over CHANS according to used-defined PROTOCOLS. Occam 2, like Java, does not have support for user-defined structs, but occam 2.1 has. At time of writing SPoC does not have error-free support of the occam 2.1 RECORD. The occam ALT is very powerful. It makes you able to wait for a channel or channels with or without timeout and optionally fall through if neither of the other ALT's branches are taken. And you can dynamically switch the channel off so that a process server can block a client as long as it can find it appropriate, an important property. The occam ALT-construct cannot timeout on a failing channel output. This functionality may be covered with memory-mapped I/O or calls to low-level native procedures. Occam is able to take slices of arrays and give them new names, as well as picking out one dimension of an array and give it a new name, all usage checked. This mechanism is called abbreviation and replaces many pointer operations. An interesting facet of occam is that it will not let

you give any portion of memory two names and then get conflicting use of that cell: aliasing errors are thus provided. Occam has a rather unusual syntax of indenting blocks instead of curly brackets. Occam has semantically "orthogonal" FUNCTIONS and PROCs, the first is not allowed to modify external state. This implies that for SPoC all FUNCTIONS are converted one-to-one to ANSI-C, no state-machine is generated.

It is reassuring to see some CSP/occam philosophy revealed in Bell Lab's Limbo native language for the new Inferno operating system; it has both alt and chan of data-type, is based on synchronous communication and has array slicing [6].

Occam has 4 replicators: SEQ, PAR, IF and ALT. I know of no other language that has anything else than a "for-loop", this is the same as occam's SEQ-loop. The PAR-loop starts a process a (compile-time known) number of times. The IF-loop searches until something happens, this makes a "break"-statement unnecessary. The ALT-replicator makes it possible to wait for communication on an array of channels. Observe that communication is done on named channels, not with named processes. This makes it possible to have well-designed and usable libraries.

CONTINUED FROM PAGE 86

```

        -- Overflow is no problem, we'll just get the last value through
:
PROC Output (CHAN OF INT input, CHAN OF BOOL acknowledge, CHAN OF INT output)
  INT buffer:
  WHILE TRUE
    SEQ
      input ? buffer
      -- This communication may block:
      output ! buffer
      acknowledge ! TRUE
:
CHAN OF INT local:
CHAN OF BOOL acknowledge:
PAR
  Input (in, acknowledge, local)
  Output (local, acknowledge, out)
:
-----
PRI PAR
  CHAN OF INT start,end:
  VAL NoOfRuns IS 1000:
  PAR
    SEQ i = 0 FOR NoOfRuns
      start ! i
    OverWrBuff.INT (start, end)
    SEQ i = 0 FOR NoOfRuns
      INT value:
      end ? value
  SKIP

```

CONTINUED ON PAGE 88

Occam as implemented on the transputer does not cause stack overflow, as the compiler calculates each process' exact need. (Occam has no recursion or dynamic memory handling.) The picture is more complicated with SPoC: variables with a lifetime between descheduling points (atomic) are local and therefore allocated on the stack, all other values reside in the large struct-tree. A consequence of the non-preemptive scheduling is that we only have one stack for all processes. We check for stack-overflow at run-time.

HAND-CODED C

We have some 19000 lines of occam and in the run-time (as opposed to boot-time) program over 28000 lines of generated C. We also have some 17000 lines of hand-written C consisting of 3 "C-tasks" called from 3 corresponding occam tasks. Calling C from occam is straightforward, SPoC has two ways to do it: inline, or through a header-file concept. The hand-coded "C-tasks" constitute a next level of communicating state machines, this level is hand-coded. These C-tasks always return to occam, they can be considered as 3 "main" programs with one event-input each. They all have their separate malloc's. This pattern has functioned quite well; even if this is where we had the

greatest safety-hazard: we had to align the C-structs with occam hand-indexed arrays. This will be removed when SPoC occam 2.1 RECORD support is correct. Only interrupts are handled when a "C-task" runs, so its maximum process time is a dimensioning factor of the reactivity of the system as a whole, since it yields at own will. When a C-task yields just to "time-slice" itself we inserted manual descheduling with a dummy communication.

DSP BOOTING

The DSP may be booted through an on-chip boot loader. Since we have a Lon-controller on the other side of the dual-port memory, we are able to boot the DSP via the dual-port, controlled from an NT-hosted configuration client. A file system is loaded to RAM with an extra boot loader we had to write to load programs larger than the size of the dual-port. A completely empty flash is programmed with the run-time program file. After this the DSP is started from flash, and user files may be programmed via the file system. The file system is typical: process code is in occam, its often used "flash driver" process is in occam, and the low level flash driver is in C. Dual-port driver and A/D conversion also have the lowest level code in C.

CONTINUED FROM PAGE 87

The generated C-code of the Input process spawned by OverWrBuff.INT follows. Observe that the occam ALT construct needs both set-up and teardown code: ENBS/DISS=Enable/Disable skip; ENBC/DISC=Enable/Disable channel. This contributes substantially to the average communication time of 126 μ s. I had to break the DISS line to get it onto the paper. Each time Scheduler schedules the process the top-most switch is taken. The code will not be discussed any further.

```
static void P_Input_3885 (tSF_P_Input_3885 *FP)
{
    while(true)
    {
        switch(FP->_Header.IP)
        {
            CASE(0):
                FP->noOfPackets_3882 = 0;
                FP->waitingForAcknowledge_3883 = false;
                GOTO(1);
            CASE(2):
                ALT();
                ENBS(((FP->waitingForAcknowledge_3883 == false) && (FP->noOfPackets_3882 >
0)));
                ENBC(true,FP->acknowledged_3879);
                ENBC(true,FP->input_3878);
                ALTWT(3);
            CASE(3):
                {
                    BOOL TMP = false;
                    TMP |= DISS(4,((FP->waitingForAcknowledge_3883 == false) &&
(FP->noOfPackets_3882 > 0)));
                    TMP |= DISC(5,true,FP->acknowledged_3879);
                    TMP |= DISC(6,true,FP->input_3878);
```

CONTINUED ON PAGE 90

DEADLOCK

Deadlock, starvation and livelock are inherent properties of any multi-threaded system. They must be handled, preferably at design-time. Occam makes reasoning about these things simple. We have not used any formal tool to discover such malign behaviour because there is no tool that takes occam code and does it. Whenever we saw a possible pathologic communication path between any number of processes, we either did a redesign or broke the cycle with an overwrite-buffer process (Appendix) - or we got a run-time stop (below).

The program below deadlocks immediately, as receiving is attempted in wrong order to sending. Compiling it with SPoC goes fine, but it will crash at run-time, with a message "Terminating application" since there are no more processes to run. Occam does send and receive with the '!' and '?' operators. New blocks are defined with indenting of 2 characters. Two processes are started and run here:

```
CHAN OF INT a,b:
PAR
  SEQ
    a ! 1
    b ! 2
  INT x,y:
  SEQ
    b ? x
    a ? y
:
```

It displays the clarity of occam: we can reason about this easily. If we do not see it, an analysing tool could stop us, or the run-time system can. Compare this to the subtleties of the monitor concept implemented in Java. Thanks to Peter Welch in the occam-for-all team [5] for pointing out this example.

CONCLUSIONS

The experience we had in occam and embedded software design made it feasible to use SPoC.

The main problem we had was with mixing occam and C and aligning C struct's and occam arrays.

Up to first release 13000 lines of written occam code was converted by SPoC to 26000 lines of run-time C-code. Altogether some 19000 lines of occam had been written (the additional was for test-code, etc.). This had taken 2200 man hours, all phases included, from conception through analysis, design, coding, testing, system testing and some documentation along the line. Meetings and communication with other people are also included.

Two reasons for using SPoC are

1. occam is a superb tool for comfortably writing concurrent programs and

2. we now can run it on our DSP and almost any processor that has a C-compiler.

The reasons for not using SPoC are

1. occam's state is undecidable (dead?, dying?, alive?, ahead of its time?) and
2. automatically generated code does, after all, generate an extra level of complexity.

We would consider doing it again, and some of us (also people outside this project) want to try it out on a Microchip PIC-processor (which we use in vast numbers) since a C-compiler is now available. Programming an embedded 386ex processor is also a possibility, we now use ANSI C plus VxWorks on those machines.

After doing this job we may redesign a transputer-based product with 35000 lines of occam code to run on a Texas DSP. And we would know how to do it.

In [7] this project has been elongated in larger detail. Observe that a CSP-library also has been written for Java [8]. ■

REFERENCES

- [1] Øyvind Teig, "Ping-Pong scheme uses semaphores to pass dual-port memory privileges", EDN, 6th June

Ad EDS

- 1996
- [2] C.A.R. Hoare, "Communicating Sequential Processes". Prentice Hall, 1985
- [3] Mark Debbage, Mark Hill, Sean Wykes, Denis Nicole, "Southampton's Portable Occam Compiler (SPOC)", In: Miles, Chalmers (ed.), "Progress in Transputer and occam Research", IOS Press, Amsterdam, 1994 (WoTUG-17 proceedings), pp.40-55
- [4] "occam2 Reference Manual". INMOS Ltd, Prentice Hall, 1988 (C.A.R. Hoare is series editor) ISBN 0-13-629312-3
- [5] occam-for-all Team. occam-for-all Home Page. <URL: <http://www.hensa.ac.uk/parallel/occam/occam-for-all/index.html>>, 1997
- [6] Lucent Technologies Inc., "The Limbo Programming Language".
<URL: <http://inferno.lucent.com/inferno/limbo.html>>
- [7] Øyvind Teig, "PAR and STARTP Take the Tanks", In: P.H.Welch, A.W.P.Bakkers (ed.) "Architectures, Languages and Patterns for Parallel and Distributed Applications", pp. 1-18. 1998 (WoTUG-21 proceedings).
- [8] Gerald Hilderink et al, "A new Java Thread model for concurrent programming of real-time systems". Real-Time Magazine 98-1, pp.30-35

Øyvind Teig is a senior development engineer who has been working with hardware design and embedded real-time systems programming. After graduation from NTH, Trondheim with a M.Sc. degree in 1975 he has worked for Autronica. In the 80's he worked with microcontrollers (assembly with self-built real-time kernel), and 8088-type machines (PL/M and Modula-2 plus separate real-time kernels). In the 90's he has worked with transputers (occam) and PIC-controller (C). He has worked on fire detection, diesel engine monitoring and radar-based fluid level gauges. After the transputer/occam experience a natural interest has become real-time programming languages and practice. Teig has several times been an invited speaker, published in technical magazines and contributed to conferences. He is a member of IEEE.

CONTINUED FROM PAGE 88

```

    }
    ALTEND();
CASE(4):
    // This communication will never block:
    OUTPUT4(FP->output_3880, &FP->buffer_3881, 8);
CASE(8):
    FP->noOfPackets_3882 = 0;
    FP->waitingForAcknowledge_3883 = true;
    GOTO(7);
CASE(5):
    INPUT1(FP->acknowledged_3879, &FP->ack_3884, 9);
CASE(9):
    FP->waitingForAcknowledge_3883 = false;
    GOTO(7);
CASE(6):
    INPUT4(FP->input_3878, &FP->buffer_3881, 10);
CASE(10):
    FP->noOfPackets_3882 = (FP->noOfPackets_3882 + 1);
    GOTO(7);
CASE(7):
CASE(1):
    if (true)
    {
        GOTO(2);
    }
    RETURN();
default: SETERR(MSG_IP);
}
}
}

```