CSP/85 Manual

David Middleton

June 1985

CSP/85: User and implementation manual.

David Middleton

March 1985

Introduction.

CSP/85 is a re-implementation of the CSP/80 system described in:

> **Design and Implementation of a language for Communicating Sequential Processes** by Jazayeri et al. in 1980 Int. Conf. on Parallel Processing (August) [IEEE].
>
> **CSP/80: A Language for Communicating Sequential Processes** by Jazayeri et al. in Compcon Fall 80 (September) [IEEE].

Significant changes were made to provide a more robust, rapid and traceable language system, while maintaining, where possible, the original skeletal structure. This report represents the ending of further modifications and improvements. It should provide both a user's manual, and an implementation manual for further modification of the system. The implementors of CSP/80 are deeply indebted to Steve Bellovin for his help with the implementation of the concurrency within the operating system level, and I am grateful to John Zimmerman for his useful suggestions which resulted in much more rapid compilation stages in CSP/85.

CSP is a notation developed by Hoare [CACM Aug 78] for expressing parallelism in terms of sequential processes that can communicate with each other. CSP/85 is one of a series of implementations of a subset of this notation. Each sequential process is described by a separate file of CSP/85 statements. The communication links between these processes are described in the *channel file*. A series of translation steps results in a group of distinct Unix processes. An invisible parent process oversees the interaction between

these processes, handling requests, and reactivating processes that were suspended. The series of implementations leading up to CSP/85 attempts to minimise the impact of system limits upon the CSP processes that can be run. Since it turns out that there are myriad ways in which to miss-program in CSP/85, the system attempts to provide exhaustive error checking.

1. Stages of a running system.

The shell file **csp** runs the system. It takes optional flags that are passed on to internal programs, a channel file that describes the interconnections, and the names of the CSP processes.

The program **parse** translates each CSP process into C code, and generates on the side a *linktable* which is a file describing the communication ports used by that process. That C code is compiled into an executable program; a set of separately compiled routines are included to support the communication operations.

The program **link** reads the channel file and the linktables generated by **parse**, and generates C data structures that describe the network in a file named **config.c**. These data structures are compiled into a parent process called the **monitor**.

Finally, **monitor** is invoked. It invokes each CSP process in turn, and then handles requests written in a Unix 'pipe' by any of the actual processes. An I/O operation in one of the CSP processes is translated into a function call in C. The common template for these function calls is to send the appropriate request for a particular service to the **monitor** (using an atomic write to a pipe), and suspend execution. The **monitor** will process requests read from the pipe, awakening particular processes at appropriate times. The actual data transfers are via a file called **msgfile**, to which each process has an independent file descriptor (in order that **lseek()**s in one process will not interfere with **read()**s and **write()**s in another).

The shell file `csp` encapsulates this behavior, saving option flags, running the translators, creating an executable file named `clean` that will remove the large number of intermediate files that are generated, and finally invoking `monitor`. Invoking `monitor` after `csp` has been run will re-execute the system of processes without repeating any translations. Changes to individual CSP/85 programs can be locally recompiled with just `parse` and `cc` unless changes to the port declarations have occurred, in which case the `link` and the `cc cspmon.o` steps must also be repeated.

2. The CSP/85 language.

A CSP/85 system consists of a set of modules containing sequential code, each held in a file with a `.csp` suffix, and a description of the interconnections between these modules held in the *channel file*. The runtime options to CSP/85 and the generation of output are described as part of the CSP/85 language in this section.

2.1 The module language.

Appendix 1 shows the BNF description of the syntax for CSP/85 code files, and the following highlights refer to those production rules. This grammar is extracted automatically from the source for the 'yacc' parser generator, by deleting the actions and peripheral code; and so reflects the conventions of yacc. Note particularly that terminal symbols are in upper case, and non-terminals are in lower case. For the following description, non-terminals are represented by italicised items, sometimes in angle brackets, and ϵ represents the empty string.

2.1.1 Basic Expressions.

The syntax for expressions follows closely upon the expression forms in C. The four types allowed in CSP/85 consist of scalars or vectors of characters or integers. With the exception that negation is accomplished with ~ instead of !, most reasonable translations

of operations will work. The intermediate file of C code generated from CSP source will show the translation that occurred. It is stored under the same root name, with the suffix `.c` replacing `.csp`. The following set of CSP/85 operators are directly transcribed into C: {++,--,<<,>>,<=,>=,==,!=,&&,||,-,~,+,-,*,/,%,<,>}. Various parts of CSP/85, have associated variables beginning with the string "csp": conflicts may occur if names are used in CSP/85 programs beginning with this string. In general, there is little validation of identifier names. However, the process name, following the **process** keyword, is checked against the name of the file. In cases of disagreement, the file name takes precedence: `.c` and `.lt` files are named for the `.csp` file, and these are matched to processes named in the interconnection specifications.

2.1.2 Control structures.

The control structures in CSP/85 are based on Dijkstra's two alternation constructs, which contain a list of guarded statements. Each guarded statement consists of a boolean expression, ->, and a statement. When the compound statement is executed, all the boolean expressions are evaluated, and from the set of guarded statements with a true boolean expression, one statement is chosen to be executed.

```
[ guard-> stmt;
| guard-> stmt;
| guard-> stmt;
]
```

```
*[ guard-> stmt;
 | guard-> stmt;
 | guard-> stmt;
 ]
```

In the first form, the compound statement is executed exactly once, and the program fails if none of the guards is true. The second form is executed zero or more times, until none of the guards is true. Truth corresponds to C's interpretation, namely non-zero integers. In choosing from among more than one possible true guard, Hoare's CSP assumes non-deterministic choice, hopes for fairness, and leaves to the programmer the responsibility that the selection method could not affect correctness. The implementation

of CSP/85 relies upon the last assumption to exploit a simple though neither fair nor non-deterministic mechanism to accomplish this construct.

2.1.3 Communication.

Interprocess communication is accomplished in CSP/85 through a channel that joins the output port in one process to an input port of another; the channels are capable of transferring any of the CSP/85 data types. The two communication primitives are: ? *var* = *port* and ! *port* = *expn*. Ports exist both as scalars, and as vectors; in the latter case, *port* includes a subscript, (*expn*), following the *name* that was defined in the port declarations. A port declaration looks like:

guarded	input		int	[*num*]		(*num*)	
ϵ	output	port	char	ϵ	*name*	ϵ	;

For example, `input port int [5] plan (7) ;` declares an array of seven ports named `plan(0)` through `plan(6)`, each of which is an input port that receives five element integer arrays.

2.1.4 Properties of communication.

Communication between two CSP processes within a network is accomplished by a pair of such statements, one performing output, and the other receiving input. The two ports involved become associated by a connection statement in the channel file. As the two processes execute, the first to reach one of the two statements suspends execution until the other process reaches the other statement. After the transfer is performed, the two processes continue separately. This implicit synchronization is an important aspect of CSP; the first process is committed to completing this transfer before it can continue execution. It only completes after the far end has communicated; if the far end terminates before communicating, this process will terminate with an error.

The communication primitives described above are excessively rigid. All that a process can do is commit to performing an I/O operation, and once begun, it cannot complete until the process at the far end of the channel also acts on this particular channel. The language needs a mechanism to test the status of a channel, for example, in order to build processes that could support resource managers. Hoare's CSP allows for input to be guarded, that is the final element of a guard may be an input operation. The guard is now true if the preceding boolean expressions are all true, and the process at the far end of the associated channel has committed to sending output. If this guard is selected, then the input operation is performed and the statement corresponding to the guard is executed, otherwise no change takes place. The guard is false if the other end of a channel is a terminated process, otherwise, the value of this guard is indeterminate. In evaluating an alternative statement, CSP/85 chooses a true guard if one is available. Otherwise, if there was a guard of indeterminate value, the process is suspended until the situation changes, at which point it retries the whole statement.

CSP/80 was designed so that output operations could also occur in a guard, with similar results. If the boolean expressions were true, and the process at the far end of the channel had committed to receive input, the output operation would be performed, and the corresponding statement executed. Otherwise, if no subsequent guards could be satisfied, the process would become suspended, waiting for some change in status of one of the ports used in this guarded statement. (For reasons of simple implementation, a change in the status of any port is sufficient to prompt the process to retry the statement.) CSP/80 included a constraint preventing both ends of a channel from being able to perform guarded I/O. This was accomplished by declaring ports **guarded** if they were to be allowed to appear within guards, and then checking during the linking process that a channel did not connect two such guarded ports.

CSP/85 has evolved under the motivation of circuit specification and this path of extension exposes some problems. I/O in Hoare's CSP has two important characteristics; it synchronizes the two processes, and it commits a process to performing an operation blindly. As far as performing output goes, a process must commit to beginning the operation, and once begun, cannot escape until the far end sends a value (or terminates). Consider the following two situations:

```
process produce ::
output port int Consumer ;
*[ true -> /*produce value*/
        !Consumer = value ;
 ]
end process
```

```
process consume::
input port int Producer ;
*[ ?value = Producer ->
                    /* use value */
 ]
end process
```

In this case, the programs will work as expected, but a naïve hardware implementation may not. This is because explicit handshake is required in hardware to keep the two processes synchronized whereas in CSP, this facility is provided implicitly by the statement definitions.

The second situation involves trying to rotate the contents of a ring network consisting of copies of the same CSP process. Since the code is duplicated for each process, either the processes will all suspend trying to send output, or they will all suspend waiting for input. Thus, the natural CSP description of the system will suffer deadlock.

These problems arise because of excessive constraint in the definition of CSP, (at least from the point of view of describing hardware). In the former case, the constraints confined a dangerous definition into behaving correctly, and in the second case, the constraints confined a reasonable definition into failing. An important characteristic for CSP/85 is to provide a descriptive notation tool, as well as an executable one.

CSP's constraints can be viewed as having two parts: commitment and synchronization; all a process can do is start an I/O operation blindly, and then it must wait for

help from a distant process before finishing. These constraints were relaxed by Hoare's introduction of guarded input. CSP/80 relaxed this constraint further by also allowing output operations to appear in guards. These approaches are weakening the commitment constraint, a process can discover information about of an operation before attempting it. Guardable output is not, however, appropriate for describing hardware; it suggests that a channel can detect if the far end is being read. Another way to relax the problem with over-constraint is to weaken the synchronization. CSP/85 does this by allowing an output operation to complete before the corresponding input operation has occurred. This proposal only removes half of the synchronization; input operations will still be delayed until the corresponding output operation occurs. From this behavior, it is still possible to accomplish the full synchronization during communication by programming an explicit acknowledgement of receipt; CSP/85 also allows the fully synchronised operation to be chosen statically for each channel.

This new interpretation of communication leads to some interesting changes in behavior that now models hardware more accurately. The circular buffer will now avoid deadlock if the CSP/85 code for each cell sends its current value to one neighbour before reading from its other neighbor. The CSP/85 description of the producer-consumer pair is now capable of overwriting values, more closely reflecting how the straightforward hardware implementation would behave. An advantage of CSP/85 is that it more closely models real hardware, while providing explicit messages when overwriting does occur. This change also creates more separation between processes; a write operation may fail without the sending process finding out (until a subsequent operation).

Both forms of relaxed constraints are available in CSP/85, and as far as possible can be combined, although without any reason being apparent. The `-g` option causes `link` to allow output ports to be guarded. Both synchronized and non-synchronized output operations are available depending on the channel specification in channel file.

2.1.5 Embedded C code.

A **PASSTHROUGH** non-terminal is a patch originally added to allow I/O with a terminal before we had decided how to model terminal I/O as CSP process communication. It consists of a line with a # in the first column, and the following text is transcribed directly into the object C code. It has since been extended to allow the use of C preprocessor statements to define compile time constants, and include other files (with the *globals* non-terminal). It can be used to allow general function calls, which are not supported in the CSP/85 syntax. In the latter case, the **csp** shell file can be modified so that the C compilation of the process also involves the **.o** file containing separately compiled functions [Gross 85]. Strong modularity is advisable. Calling a distinct function maintains a well defined interface between explicit C code and that which the CSP/85 code generated, but building structure that extends from one **PASSTHROUGH** to another around intervening CSP code is dangerous and unnecessary.

2.2 The interconnections: the channel file language.

The channel file, given as the first file argument to **csp**, describes the interconnections among the csp processes. It consists of two parts, the declarations and the connections, each consisting of line orientated commands. Comments, any lines beginning with #, may be placed anywhere.

Declarations either define constants, using the command **set** *var expn*, or establish arrays of processes, using the command **array** *<process name> <expn> : <expn>* where the two expressions provide the inclusive limits of the subscripts for the process being duplicated. The variable **csp_proc_num** can be used in CSP/85 code to provide elements of such an array with their identity; it is the programmer's responsibility not to modify this variable. Expressions combine decimal constants and previously defined variables using

the C operators {+,-,*,/,%}. NO PRECEDENCE RULES are applied; all operations associate to the left, with a warning, which can be suppressed by using parentheses.

Connections join two ports from separate processes. A port is specified by a process name, followed by a subscript in parentheses when that name has been defined as an array, a period, and a port name that is declared within that CSP/85 process. The port name may include a subscript expression in parentheses; scalar ports are indistinguishable from a port name with a zero subscript. The statement **connect** *port* **to** *port* creates a channel in which the process performing the output operation can continue execution without waiting for the reading process to catch up. Replacing **connect** with **sync** creates a channel in which the output operation does not complete until the input operation occurs. Gross noted that the full synchronization behaviour provides a higher level of descriptiveness when using the language for VLSI module specification.

These connection statements can be enclosed in iterative statements. The statement **for** <*var*> <*expn*> : <*expn*> repeats a group of connections with the variable <*var*> taking on successive values in the inclusive range specified. The scope of the **for** statement extends to the next **endfor** statement which closes out *all* **for** loops.

```
for i 1 :  n
      sync control.od(i-1) to doutpt(i).od
endfor
```

2.3 Execution: the **csp** *shell file language.*

A CSP/85 program is a group of files of the form <*process*>**.csp** as specified in section 2.1, and a channel file as specified in section 2.2. This program can be run using:

csp *options* <*channel file*> <*process names*>

where the *process names* are the roots of the **.csp** files.

2.3.1 Runtime options.

The following options apply to various programs invoked by the **csp** shell file.

-x causes **monitor** to leave a trace of transactions in the file **logfile**. A separate line describes the result of each transaction. It consists of the request type, the process and port id's, and the state of all the processes after that transaction has been performed. **R** is running, **|** is in I/O, **-** is terminated, and **W** is suspended in an activity check. The process and port id's can be used with the **config.c** file (manually) to find the corresponding CSP/85 instruction.

-g causes **link** to allow output ports to be declared as guardable. (**Parse** would prevent ports being used that way without declaration.)

-d*num* causes the **link** and **monitor** to generate tracing information. The number ranges from 1 for minimal output to 9 for far too much.

-p In some versions of **csp**, this option started up a delayed **ps&** command that provided UNIX process id's for dealing with lost signals that were causing CSP/85 to hang, under the 4.1BSD operating system.

2.4 Output.

Monitor will generate output on the **stderr** file in the case where it detects deadlock. Deadlock is defined to be the situation where none of the CSP/85 processes are still executing, and some of them are not terminated. The **monitor** indicates the status and last request for each of the CSP/85 processes.

Beyond this, the programmer can generate output using a line such as

```
#printf("%d %d\n",v1,v2) ; fflush(stdout) ;.
```

It is appropriate to choose one module that provides the environment for a running system, and perform all interaction with the terminal from that module. In these cases, the CSP/85 module contain a line **##include <stdio.h>** before the **process** keyword.

3. Implementation.

The system uses the following files:

`csp`	The shell file that runs a system
`camel`	lex source code of the CSP/85 syntax analyser
`llama`	yacc source code to generate **parse**, the CSP/85 to C compiler
`parse`	generates C code and port descriptions from CSP/85 source
`*.csp`	CSP/85 source code
`*.c, *.lt`	generated by **parse**: code and link table
`cspio.c`	source for the 6 csp functions: init, read, write, . . .
`cspio.o`	to be combined with *.c to yield the CSP processes
`cspio.h`	various general constants
`dirio.h`	three access routines to **msgfile**: read, write and copy
`linker.c`	**link** source, which turns channel file and *.lt into config.c
`config.c`	process and channel structures describing global interactions
`cspmon.c`	supervisor and arbitrator of separate Unix CSP processes
`cspmon.o`	completely compiled except for the config.c of a particular run
`monitor`	`cc cspmon.o config.c` performed in csp
`msgfile`	CSP processes communicate here, with dirio.h through cspio.o
`logfile`	-x option to monitor takes a log of all requests
`clean`	generated by csp to remove files after system has run

3.1 Request mechanism for CSP/85 processes.

The file `cspio.c` contains 7 routines by which the CSP/85 processes request support services from **monitor**: **cspinit**, **cspread**, **cspwrite**, **cspcc**, **cspac**, **cspexit**, and **cspabort** (which is unused in the current implementation). **Parse** generates calls to these functions in its object code. These routines use a Unix pipe that was open when **monitor** created the processes with **fork()**, to send requests to **monitor**. They buffer data and receive a return status in the file **msgfile**. This mechanism is, as much as possible, independent of the number of processes running.

A request contains the process index (into the `config.c hp[]` process structure) which is used for identification, the requested operation, and when appropriate, the port number related to the request. Having written such a request (an atomic write is important since many processes will likely be doing this concurrently), the process suspends itself using the Unix **pause()** function. When the **monitor** has satisfied the request, by transferring data within **msgfile** and appropriately changing process states, it writes the status of the operation in **msgfile** and sends a signal to awaken the requesting process. That process reads the status from **msgfile** and continues execution of that CSP program.

3.2 Critical sections.

Because these processes are all executing concurrently in the Unix environment, it is possible that the child process will be swapped out after attempting to write its request to the pipe. The `monitor` may then read and process the request, and send a signal to the child process. When the child continues, it suspends itself waiting for a signal that has already come. The original mechanism to avoid this, under the early Unix signal mechanism, involved clearing a flag before sending the request, setting the flag when a signal arrived, and then only performing the `pause()` if the flag was still cleared. There is still the chance of missing signals here, and such has occurred; but, in general, this system was adequate. The current system uses 4.2BSD signal capabilities. The function `sigblock()` allows certain signals to be delayed in arrival, and a corresponding function `sigpause()` suspends the process and allows prior permitted signals through again, as an atomic operation, (through the explicit storage of the old mask to be restored (without `WAKEUP`) which was possibly already inhibited). The chosen signal, `WAKEUP`, and the conversion macro `Mask()`, are defined in `cspio.h`. This interrupt mechanism requires an interrupt handling function, which for CSP/85 is null.

3.3 Data buffers.

Communication is accomplished by a `cspwrite()` call in the sending process and a `cspread()` call in the receiving process. The function `cspwrite()` copies its data into a buffer area in `msgfile`, and then sends a write request as explained above; When `monitor` awakens it, its buffer area has been emptied, so it can immediately perform another `cspwrite()` on the same port. The function `cspread()` performs a read request, and when that returns, it may read the incoming data from its own buffer area in `msgfile`. In general there is danger that once the transaction has completed, the sender may send the next value before the receiver has actually retrieved the data from the prior transfer; this a symptom of uniprocessor support for a parallel language. Rather than increase the

number of system calls for synchronising transfers, CSP/85 creates two buffer areas in **msgfile**, and **monitor** transfers between them when both requests have been received. With the possibility of non-synchronized output, a third buffer area has been added: every channel now has three buffers associated with it in **msgfile**. The first buffer is used by the write operation before it sends its request. The second buffer holds the data that a read operation will retrieve after a mutual transfer has been performed. The third buffer holds the data that a write operation sent after its request has been acted upon. This buffer needs to be distinct from the second for the cases where the reader has nominally received its data from a previous transfer, but has in fact not yet been rescheduled in the (uniprocessor) operating system. If data is transferred to the third buffer before a read operation emptied the previous contents, an error message is issued.

3.4 Handling guarded I/O.

The function **cspcc()**, a commitment check, requests the status of the channel connected to a port. It is used when an I/O operation appears in a guard, and the process will continue execution as soon as **monitor** receives and handles the request. There are three basic responses possible: the process at the far end of the channel has terminated, (this guard may not be selected); the process at the far end of the channel has performed its part of a communication operation, (this guard is selected, and the transfer completed); or the channel is inactive. In the last case, the process will try other guarded statements, looking for one that can be executed immediately.

In the case where no other guards are satisfied, this process must wait; it cannot exit the alternative statement until all guards definitely fail. The function **cspac()** performs a general "activity check"; the process has suspended operation until something interesting happens to one of its ports, at which time **monitor** will signal it, and it will retry the entire alternative statement. There is the possibility that a distant process will make its communication request between the time when a particular process performs **cspcc()** to

discover that a channel is inactive, and the time when it performs **cspac()** to be put to sleep. Every channel has a flag called **ac_race** to detect this situation. When **monitor** receives a commitment check request upon a port, it sets this flag to **noted**. If something affects the far end of the channel, the **ac_race** is modified so that when the activity check request occurs, that process is immediately awoken to retry the alternative statement. Some false alarms may occur: the various **ac_race** flags remain at **noted** even when a subsequent guard is chosen. Care must be taken so that when the flags are cleared by an activity check request, only those associated with the process are altered. **Link** ensures that guardable ports always come first in a channel. The function **cspac()** clears the flags for all the ports the process owns that appear first in their channel, clearing both real notes and stale ones. These are exactly the ports that may be guarded by this process.

A call to **cspinit()** is the first executable statement in the C code **parse** generates. It performs various initialisation, including reading data placed on an initialisation pipe by **monitor** describing the port buffer positions within **msgfile** for this process. The function **cspexit()** is called both for proper termination and in the case of errors. **Cspabort()** provides for a more urgent termination, and is unused.

The code generated by **parse** uses a number of variables for temporary results: **csp_proc_num** is the single variable meant to be visible to the programmer, it contains the subscript that distinguishes otherwise identical copies of a process. Other such variables are supposed to be hidden, and the programmer should not modify any variables beginning with "csp". They include **csps** which holds the status of a port returned by the **cspcc()** commitment check call, **cspx** which holds the subscript of a target variable which is an element of an array, (for example, the value 4 in **?a[4] = A ;**), and **cspp** which holds the subscript of an array of ports, (for example, the value 1 in **!B(1) = 3 ;**).

3.5 Guarded statements.

In implementing the two forms of the CSP alternation command, the interpretation was chosen to be that on each cycle, the lexically first true guard would determine the statement to be executed. This is consistent with perceived intent of the CSP operation, provides simple and repeatable operation, but unfortunately does not drive execution through any sort of fair, or even reasonable, program path.

A conditional C statement is generated for each guard and statement pair; the C test comes from the CSP guard, and the C statement body comes from combining the CSP statement and a branch to the bottom of the block. If the guard is not true, then execution will fall into the code for the next guarded statement. If none of the guards is true, then the result depends on whether this statement is simple alternation, in which case this is a program error, or a repetition, in which case the program exits the alternation command.

When an I/O operation appears within a guard, `cspcc()` returns the status of the port which is saved in `csps`. There are three possible states: `terminated` which corresponds to a false guard, `committed` to I/O which corresponds to a true guard, and `inactive` which means the result depends on some future action. In the latter case, subsequent guarded statements are attempted, but the difference from a false guard is noted by setting `cspi`. If, at the end of the block, none of the other statements could be executed, and `cspi` is set, then the program performs an activity check with `cspac()`, which means this program will remain suspended until some activity occurs on one of its ports. Rather than specify particular ports of interest in this block, any port is considered, which leads to the possibility of false alarms in returning from `cspac()`. Thus, a simple alternative is encased in a while statement that waits for `cspi` to be cleared. It is cleared before any guards are attempted, it is set when any commitment check returns an indefinite answer, and it is cleared again by any chosen statement in case an indefinite guard preceded that choice. The repetition is further encased in a while loop whose controlling variable, `cspg`,

remains true until all the guards in the inner block fail, and execution drops through to a statement that clears cspg.

4. Installation.

A system should start with the following files: ../include/cspio.h, ../include/dirio.h, camel, csp, llama, linker.c, cspio.c, and cspmon.c.

The following steps should produce a running system:

```
lex camel
yacc llama
cc y.tab.c -ll -o parse
cc linker.c -o link
cc cspio.c -c cspio.o
cc cspmon.c -c cspmon.o
```

The variable CSPHOME in csp should be set to reflect the directory where these files are situated.

4.1 Important defines *in cspio.h.*

There are three important constants in the cspio.h file, Most_procs, Most_ports, and Most_Inst. Most_procs is the maximum number of processes that a CSP/85 system may start up, which is limited by a system defined constant, and must leave room for at least monitor and the user's shell. Most_procs allows link to provide more informative messages than an indication that the fork() system call failed in monitor (-27: no such process!). It also defines the appropriate size for the hp[] system table. Most_ports is the maximum number of port names that may be declared within a single .csp file. Arrays of ports still count for one. Most_Inst is the number of instances of ports in the complete CSP/85 system. It differs from Most_ports in that it covers ports in all the .csp files, rather than each file alone, and each array of ports counts for many entries. It is needed by link which checks that all declared ports are connected to exactly one

channel. It is difficult to choose a reasonable limit for this constant since it can easily grow quadratically with `Most_procs`, as well as linearly with `Most_ports`. We found it useful to raise `Most_procs` to 100 on our machine. `Most_ports` being 20 meant that a 200,000 element array is created. It is reasonable to choose something lower, and raise it when an error message along the lines of `Link: 'used' overflow` occurs.

```
process     :   globals PROCESS IDENT DELCOL
                portdec decls stmnts END PROCESS ;
globals     :   /* empty */
            |   globals PASSTHROUGH ;
portdec     :   /* empty */
            |   portdec guarded INPUT PORT TYPE dim /* element size */
                IDENT portdim /* number of ports */ SEMICOL ;
decls       :   /* empty */
            |   decls decl SEMICOL ;
decl        :   TYPE IDENT dim ;
dim         :   /* no bound => scalar */
            |   LBRA NUM RBRA ;
portdim     :   /* no bound => scalar */
            |   LPAREN NUM RPAREN ;
guarded     :   GUARDED
            |   /* empty */ ;
stmnts      :   command
            |   stmnts command ;
command     :   SKIP SEMICOL
            |   expn SEMICOL
            |   io SEMICOL
            |   alt
            |   PASSTHROUGH
            |   error ;
alt         :   choice alterns RBRA ;
choice      :   REP
            |   LBRA ;
alterns     :   altern
            |   alterns BOX altern ;
altern      :   guard ARROW decls stmnts ;
guard       :   bool decls
            |   bool SEMICOL decls io
            |   decls io ;
bool        :   expn
            |   bool SEMICOL expn ;
expn        :   NUM
            |   STRING
            |   QUOTE
            |   IDENT sub
            |   LPAREN expn RPAREN
            |   expn op expn
            |   expn op
            |   op expn ;
sub         :   /* empty */
            |   LBRA expn RBRA ;
op          :   OP
            |   EQUALS ;
io          :   QUERY target EQUALS port
            |   EXCLAM port EQUALS expn ;
target      :   IDENT tsub ;
tsub        :   /* empty */
            |   LBRA expn RBRA ;
port        :   IDENT psub ;
psub        :   /* empty */
            |   LPAREN expn RPAREN ;
```

Appendix 1.
BNF productions for CSP.

Appendix 2. Further work.

Various possibilities that would improve the system, but were not crucial to its operation have been indefinitely postponed. They include the following:

The system needs a makefile that will install it appropriately. This includes making the site for storing the include files less rigid.

The interface between `config.c`, `monitor`, and `cspio.c` should provide more descriptive messages. For example, `config.c` is already generated so that port and process names appear in a comment. If these were moved into strings in the `hp[]` and `ch[]` structures, then tracing information could refer to them directly. Line numbers identifying which of potentially many transfers through a particular port would also be useful.

The routines in `dirio.h` belong in a separately compiled `dirio.c` file.

A large fraction of the lines in the `logfile`, created by the `-x` option, are commitment transactions. These can often be deduced from the I/O or activity check transactions, and crowd out the useful entries. This option should take a parameter that can disable the reporting of commitment check transactions.

A common situation is exemplified by a clock module sending signals to each element of an array of processes. The clock module needs the number of processes for it to declare an array of ports for sending signals, and during execution for counting. There needs to be a mechanism for the compilations (`parse` and `cc`) to get information from the *channel file*. An example of this is shown in Appendix 3, where the number 7 appears in two distinct files.

A concise notation for iterating through alternative statements a specific number of times, rather than explicitly encoding such cases, appears to be useful in practice.

`Link` could use true variables in the middle of `for` statements. The tree example in Appendix 3 shows both the ugliness possible in expressions for connection statements; this can be reduced by suitable choice of variable range. Minor differences often occur among similar connections, for example, the different names of processes at the periphery of a network. These cause the number of connection statements to grow rapidly.

Two further facilities in `link` would be useful in modelling networks of identical hardware modules. Where one hardware unit is composed of many CSP/85 modules, some system of hierarchical grouping would improve the descriptiveness. The CSP/85 system is also intended to provide a software environment in which a hardware module can be tested in place of the CSP/85 code that describes it. This requires some kind of `reconnect` command in `link` that will allow the hardware module to replace a particular element of a CSP/85 network, without disrupting the iterative loops describing the structure.

Linenumbers reported in `link` ignore null lines.

Since `monitor` is already noting ports on which it has recently performed a commitment check, `cspac()` could be woken only by actions that affect ports that have the `ac_race` flag `noted`. This should reduce false alarms in awakening processes.

`Monitor` should print out the current state of the various CSP/85 processes on receiving some interrupt. This should be synchronised with the request handling cycle, to ensure internally consistent results and safe behaviour at the signal handling level.

Appendix 3. An example.

```
## include <stdio.h>
process env ::
        output  port int Left ;
        input   port int Right ;
        output  port int Child0 ;
        input   port int ChildI ;

int code; int val;

#printf("encode bits :") ;
#fflush(stdout) ;
#scanf("%d",&code) ;
*[ code>=0 ->
        !Left = code ;
        ?code = Right ;
#       printf("\n") ;
#       fflush(stdout) ;
        ?val=ChildI ;
        !Child0 = 0 ;
        !Left = code ;
        ?code = Right ;
#       printf("\nencode bits :") ;
#       fflush(stdout) ;
#       scanf("%d",&code) ;
]
end process
```

```
## include <stdio.h>
process t ::
        output  port int Parent0 ;
        output  port int Left0 ;
        output  port int Right0 ;
        input   port int ParentI ;
guarded input   port int LeftI ;
        input   port int RightI ;

int v1; int v2;

*[ ?v1=LeftI ->
        ?v2=RightI ;
        !Parent0 = v1 + v2 ;
        ?v2 = ParentI ;
        !Left0 = v2 ;
        !Right0 = v1 + v2 ;
]
end process
```

```
## include <stdio.h>
process 1 ::
        output  port int Right ;
guarded input   port int Left ;
        output  port int Parent0 ;
        input   port int ParentI ;
int result; int code;
result = 0 ;
*[ ?code=Left ->
        result = (code >> (7-csp_proc_num)) & 1 ;
#       printf("%2d ",result) ;
        !Right = code ;
#       fflush(stdout) ;
        !Parent0 = result ;
        ?result = ParentI ;
        ?code=Left ;
#       printf("%2d ",result) ;
#       fflush(stdout) ;
        !Right = code ;
]
end process
```

```
#number of T cells
set     n       7
array   t       1 :  n
array   l       0 :  n
for i       1 :  n/2
        connect         t(i).LeftI   to t(2*i).Parent0
        connect         t(i).RightI  to t((2*i)+1).Parent0
        connect         t(i).Left0   to t(2*i).ParentI
        connect         t(i).Right0  to t((2*i)+1).ParentI
endfor

for i       (n/2)+1         :  n
        connect         t(i).LeftI   to l(2*(i-((n/2)+1))).Parent0
        connect         t(i).RightI  to l((2*(i-((n/2)+1)))+1).Parent0
        connect         t(i).Left0   to l(2*(i-((n/2)+1))).ParentI
        connect         t(i).Right0  to l((2*(i-((n/2)+1)))+1).ParentI
endfor

for i       1 :  n
        connect l(i).Left    to l(i-1).Right
endfor
connect env.Left     to l(0).Left
connect env.Right    to l(n).Right
connect env.Child0   to t(1).ParentI
connect env.ChildI   to t(1).Parent0
```

This example builds a tree of t modules with l modules at the leaves and surrounds this structure with an env module. It performs an algorithm used by the FFP machine to generate cumulative sums. (It took about one hour and eight iterations to get running.)

There are some points of interest. The number 7 appears both in the *channel file*, as well as in the code. With a higher process limit, this might be changed to 15 or 31 (in both places). Choosing stranger odd numbers for n will yield a non full tree in which the zeroth l module will begin in the middle of the string of l modules. This requires more complex programming either in the `l.csp` code, or else with the `connections` in the *channel file*. Care *must* be taken with synchronization of the various print operations.An early run of this example ran into problems with the l modules not printing their values in left to right order, despite seemingly obvious constraints to do so.