SGS-THOMSON MICROELECTRONICS

TRANSPUTER ANSI C TOOLSET

Transputer software development and debugging tools

PRODUCT INFORMATION

FEATURES

- ANSI C compiler (X3.159-1989).
- Excellent compile time diagnostics.
- Global and local optimization.
- Assembler inserts and stand alone assembler.
- Support for EPROM programming.
- Support for placing code and data in user specified memory locations.
- Support for dynamically loading programs and functions.
- Small runtime overhead.
- Cross-development from PC and Sun-4 platforms.
- Support for parallelism.

INQUEST Interactive and post-mortem debugging:–

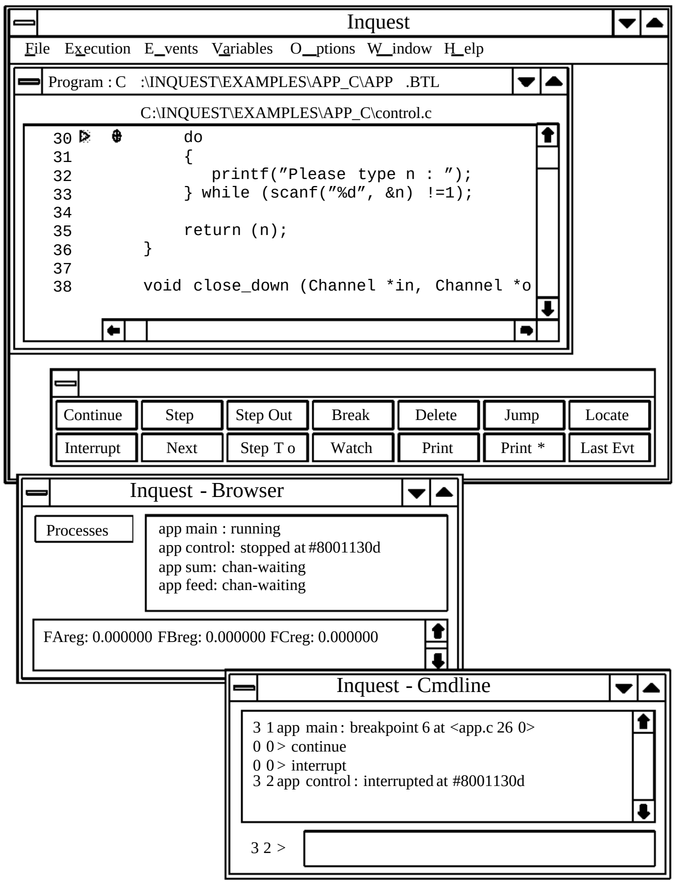
- Windowing interface using X Windows or Windows.
- Programmable command language.
- Source code or assembly code view.
- Stack trace-back facility.
- Variable and Memory display facility.
- C expression interpreter.

INQUEST Interactive debugging:–

- Process and thread break points.
- Single stepping of threads.
- Read/Write/Access watch point capability.
- Facilities to interrupt and find threads.

Performance analysis tools:–

- Analysis of time spent in each function.
- Analysis of processor idle time and high priority time.
- Analysis of number of times each block executed.



```
        do
        {
           printf(”Please type n : ”);
        } while (scanf(”%d”, &n) !=1);

        return (n);
    }

    void close_down (Channel *in, Channel *o
```

```
app main : running
app control: stopped at #8001130d
app sum: chan-waiting
app feed: chan-waiting
```

```
FAreg: 0.000000 FBreg: 0.000000 FCreg: 0.000000
```

```
3 1 app main : breakpoint 6 at <app.c 26 0>
0 0 > continue
0 0 > interrupt
3 2 app control : interrupted at #8001130d

3 2 >
```

DESCRIPTION

The Transputer ANSI C Toolset provides a complete high quality software development environment for T2xx, T4xx and T8xx transputers and ST20450 processors. The compiler supports the full ANSI C language definition and includes both local and global optimizing features. Improved embedded application support is provided by both configuration and symbol map utilities.

An *interactive windowing debugger* provides single stepping, breakpoints, watchpoints and many other features for debugging sequential and multi-tasking programs. *Execution profilers* give various post-mortem statistical analyses of the execution of a program.

Contents

1 Introduction

The ANSI C Toolset provides a complete high quality software development environment for all IMS T2xx, T4xx and T8xx series transputers and ST20450 processors. The compiler supports the full ANSI C language definition and includes both local and global optimizing features. The run-time library includes both standard C functions, supported by host target connections, and transputer specific functions to facilitate multi-processor programming and embedded control operations.

Networks of processors are supported by a software through-routing configurer that greatly simplifies the placement of code and communication paths in complex applications. Improved embedded application support is provided by both configuration and symbol map utilities. The real-time and multi-tasking support uses the on-chip hardware micro-kernel and timers, so for many applications no operating system or real-time kernel software is needed.

An interactive windowing debugger provides single stepping, breakpoints, watchpoints and many other features for debugging sequential and parallel programs running on one or more transputers. Execution analysis tools give post-mortem statistical analyses including execution profiles, processor utilization, test coverage and block profiles.

The host interface is provided by the AServer. This can be used simply as an application loader and host file server, invoked by the **`irun`** command. The INQUEST tools have their own commands which in turn load **`irun`** in order to load the application. The AServer may also be used to customize the host interface if required.

1.1 Applications

- Single processor embedded systems
- Multiprocessor embedded systems
- Porting of existing software and packages
- Massively parallel applications
- Evaluation of concurrent algorithms
- Low cost single chip applications
- Low level device control applications
- Scientific programming

2 Code-building tools

The Transputer ANSI C Toolset provides complete C cross-development systems for transputer targets. They can be used to build parallel programs for single transputers and for multi-transputer networks consisting of arbitrary mixtures of T2xx, T4xx and T8xx transputer types. Programs developed with the toolset are both source and binary compatible across all host development machines.

The ANSI C Toolset is available for the following development platforms:

- IBM 386/486 PC and compatibles under MSDOS 5 and Windows 3.1, or later versions.
- Sun 4 under SunOS 4.1.3 or Solaris 2.4 with X11 Release 4 server or OpenWindows 3, or later versions.

2.1 How programs are built

The toolset build process is shown diagrammatically in Figure 1.

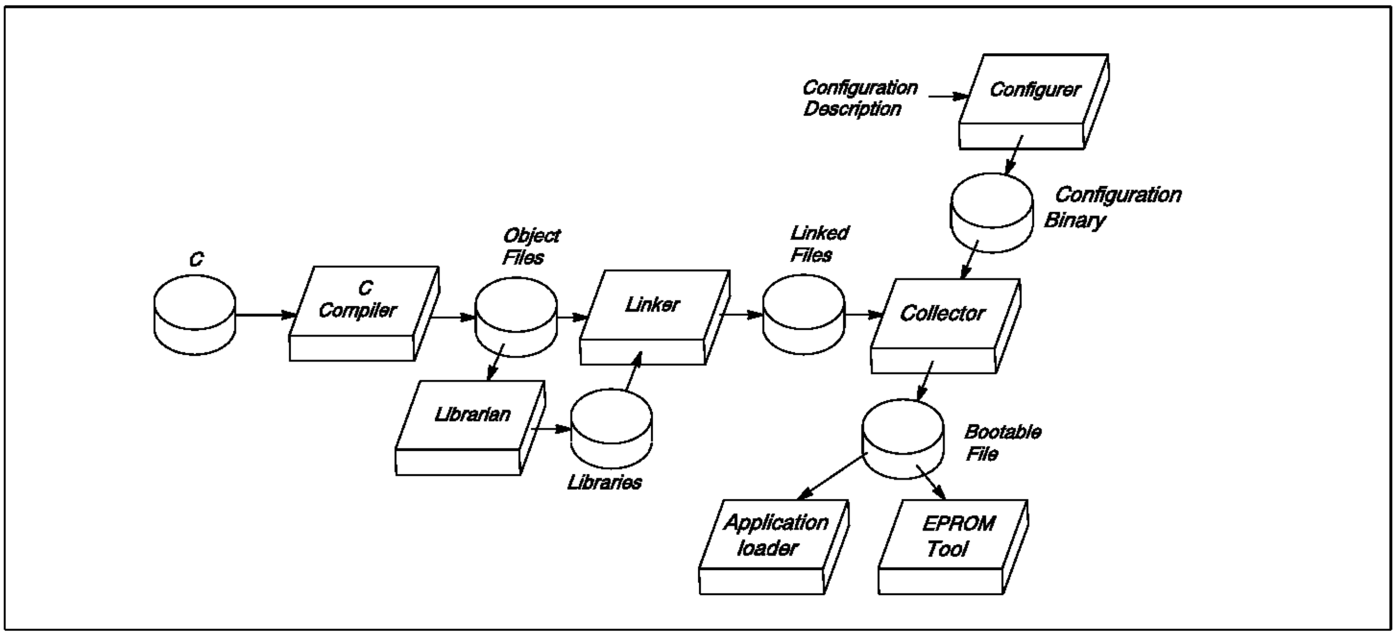


Figure 1 The tool chain

ANSI C source files may be separately compiled into *object files*. The compiler and libraries are described in section 2.2. The *librarian* may be used to collate object files into libraries. The *linker* links object files and libraries into fully resolved linked units.

A *configuration description* is a text file describing the target hardware and how the software maps onto it. The *configurer* converts the configuration description file into a *configuration binary* file. The *collector* removes any debugging information, and uses the configuration binary to collect the linked files with bootstrap code to make an executable file called a *bootable file*.

During development and for hosted systems, the bootable file may be loaded down a hardware serial link onto the target hardware using the *application loader*. For stand-alone systems, the bootable file may be converted, using the EPROM tool, to an industry standard EPROM format for programming EPROMs.

In addition to loading programs down a hardware serial link, the application loader program provides access to host operating system facilities through a remote procedure call mechanism. This method is used to support the full ANSI C run-time library.

A memory configurer tool is supplied for describing a ST20450 memory configuration. This data is used to initialize the memory interface of the ST20450. The memory interface can be initialized either using the hardware serial link or from ROM.

2.2 ANSI C compilation system

2.2.1 Compiler operation

The compiler operates from a host command line interface. The preprocessor is integrated into the compiler for fast execution. The compile time diagnostics provided by the compiler are truly excellent, including type checking in expressions and type checking of function arguments.

2.2.2 ANSI conformance

The ANSI C Toolset supports the full standard ANSI C language as defined in X3.159-1989. The compiler passes all the tests in the validation suites from Plum Hall and Perennial.

2.2.3 Local optimized code generation

The compiler implements a wide range of local code optimization techniques.

Constant folding. The compiler evaluates all integer and real constant expressions at compile time.

Workspace allocation. Frequently used variables are placed at small offsets in workspace, thus reducing the size of the instructions needed to access them, and hence increasing the speed of execution.

Dead-code elimination. Code that cannot be reached during the execution of the program is removed.

Peephole optimization. Code sequences are selected that are the fastest for the operation.

Constant caching. Some constants have their load time reduced by placing them in a constant table.

Unnecessary jumps are eliminated.

Switch statements can generate a number of different code sequences to cover the dense ranges within the total range.

Special idioms that are better on transputers are chosen for some code sequences.

2.2.4 Globally optimized code generation

The ANSI C globally optimizing compiler extends the types of optimizations it performs to global techniques. These have typically given a 15–25% improvement in speed over the local optimizations as measured by a suite of internal benchmarks.

Common subexpression elimination removes the evaluation of an expression where it is known that the value has already been computed; the value is stored in temporary local workspace. This improves the speed of a program and reduces code size.

Loop-invariant code motion can move the position where an expression is evaluated from within a loop to outside it. If the expression contains no variables that are changed during the execution of a loop, then the expression can be evaluated just once before the loop is entered. By keeping the result in a temporary variable, the speed of execution of the whole loop is increased.

Tail-call optimization reduces the number of calls and returns executed by a program. If the last thing a function does is to invoke another function and immediately return the value therefrom, then the compiler attempts to re-use the same workspace area by just jumping to (rather than calling) the lower level function. The called function then returns directly to where the upper level function was called from. In the case where the call is a recursive call to the same function, then the workspace is exactly the right size, and a saving is also made because the stack adjustment instructions are no longer needed either. This optimization saves speed and total stack space.

Workspace allocation by coloring reduces the amount of workspace required by using a word for two variables when it can be determined that they are not both required at the same time. In addition the variables that are most frequently used are placed at lower offsets in workspace. This reduces the number of prefixes needed to access the variables and so reduces the total code size.

The optimizing compiler also implements a pragma, **`IMS_nosideeffects`**, whereby the user can indicate that a function does not have any side-effects on external or static variables. The optimizer can then use this information to make assumptions about what state can be changed by a function invocation and hence about the validity of any previously computed expressions.

2.2.5 Libraries

The full set of ANSI libraries is provided for all transputer types. The standard mathematics library operates in **double** precision. Versions of the mathematical functions are provided that operate on **float** arguments and return **float** values. These libraries provide improved performance for applications where performance requirements override accuracy requirements.

The standard C mathematical functions provided use the same code as in the occam compiler. This ensures identical results and accuracy.

A reduced C library is supplied to minimize code size for embedded systems applications. This library is appropriate for processes which do not need to access host operating system facilities.

Collections of functions can be compiled separately with the C compiler and then combined into a library. The linker is used to combine separately compiled functions into a program to run on a single processor. The linker supports selective loading of library units.

2.2.6 Mixed language programs

The ANSI C Toolset allows occam procedures and (single valued) occam functions to be called from C just like other C functions. Pragmas are provided to tell the compiler not to generate the hidden static link parameter (required by C but not by occam), and to change the external name of a function, since occam procedures and functions may have names which are not legal C function names.

The associated occam toolset supports calling C functions directly from occam. Functions which require access to static variables are supported.

occam and C processes may be freely mixed when configuring a program for a single transputer or a network of transputers.

2.2.7 Assembler inserts

The ANSI C Toolset provides a very powerful assembler insert facility. The assembler insert facility supports:

- Access to the full instruction set of the transputer;
- Symbolic access to C automatic and static variables;
- Pseudo-operations to load multiword values into registers;

- Loading results of C expressions to registers using the pseudo operations;
- Labels and jumps;
- Directives for instruction sizing, stack access, return address access etc.

2.3 Target systems

The compiler will generate code for the full range of first generation transputers (IMS M212, T212, T222, T225, T400, T414, T425, T800, T801, and T805) and the ST20450.

The processor target type and other compilation options are specified by command line switches. Libraries may contain code compiled for several different target processors; the linker will select the correct target at link time. The compiler, linker and librarian additionally support code compiled to run on a range of processor types achieving a space saving in libraries.

Mixed networks may consist of any combination of any processor types. The configuration tools, EPROM support tools and interactive and post-mortem debugging tools all support mixed networks.

2.4 Support for parallelism

The ANSI C Toolset supports parallelism on individual transputers, and parallelism across networks of transputers.

Parallel processes may be created dynamically by using library calls. Processes may be created individually in which case the function call will return immediately with the called process executing concurrently with the calling process, or created as a group, in which case the function call return will indicate completion of all processes within the group.

Processes have their own stack space (typically allocated from the heap of the main process) but share static and heap space. Processes created in this way can communicate by message passing over channels or by shared data protected by semaphores. Processes may be created at high and low priority levels. Interrupt routines are typically implemented as high priority processes. Functions are provided to read a message from one of a list of channels, to timeout on channel input, and to access the high and low resolution timers built into the transputer. The microcoded transputer scheduler provides extremely efficient scheduling of these processes and efficiently implements many features provided by real-time executives.

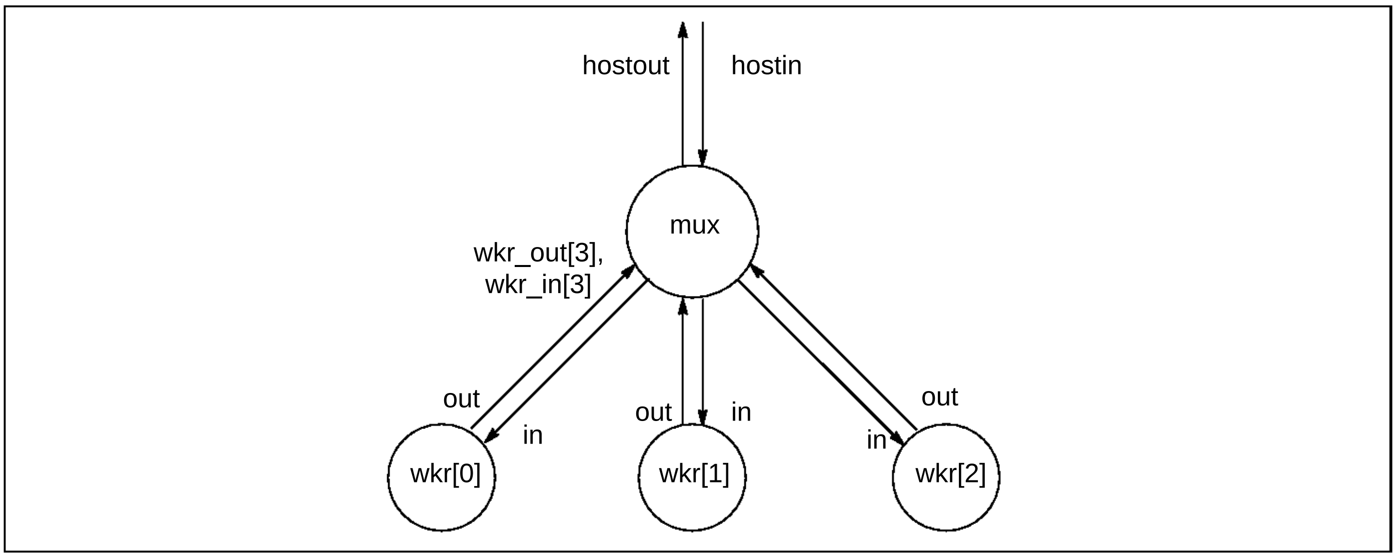


Figure 2 Software Network

The example code in figure 3 illustrates one way to program the collection of parallel processes shown in figure 2, using dynamically generated threads for the four processes. The functions **`fmux`** and **fwkr** contain the executable code of the processes. These processes will communicate using message passing functions.

The linker produces processes in the form of fully linked units. These units can be distributed over a network of processors using the configuration tools. Processes communicate by message passing over channels.

A configuration language is used to describe the transputer network, the network of processes and interconnections, and the mapping of the processes onto the transputer network. Multiple processes may be mapped onto the same transputer.

```
#include <process.h>
#include <misc.h>
#include <stdlib.h>

/*  Define prototypes for process functions   */

void fmux (Process*, Channel*, Channel*, Channel *[], Channel *[], int);
/*  process, hostin, hostout, in, out, no_workers  */

void fwkr (Process*, Channel*, Channel*, int);
/*  process, in, out, worker_id  */


/*  Main program  */

int main (int argc, char *argv[], char *envp[],
         Channel *in[], int inlen, Channel *out[], int outlen)
{
   int i;

   /*  Declare processes and channels  */
   Channel *hostin, *hostout, *wkr_in[3], *wkr_out[3];
   Process *mux, *wkr[3];

   /*  Allocate channels  */
   hostin = in[1];
   hostout = out[1];
   for (i = 0; i < 3; ++i)   {
      wkr_in[i] = ChanAlloc ();
      wkr_out[i] = ChanAlloc ();
   }

   /*  Allocate processes  */
   mux = ProcAlloc (fmux, 0, 5, hostin, hostout, wkr_in, wkr_out, 3);
   for (i = 0; i < 3; ++i)   {
      wkr[i] = ProcAlloc (fwkr, 0, 3, wkr_in[i], wkr_out[i], i);
   }
   /*  Start the processes running in parallel  */
   ProcPar (mux, wkr[0], wkr[1], wkr[2], NULL);

   exit (0);
}
```

Figure 3 Parallel processes in C

Although transputers can only transmit messages along hardware links between neighboring nodes in a network, it is possible for software processes to communicate along channels that span several nodes of a network. The configurer can add special processes to handle the routing and multiplexing of messages between the user’s processes. The user need do nothing extra to make use of this facility, known as *software through-routing*. If the application is restricted to communications between neighboring processors and all channels are explicitly placed on hardware links, then no extra code will be incorporated by the configurer.

Three examples of possible configurations follow, to illustrate just how easy it is to configure programs for transputers. In all cases we assume the **`mux`** and **`wkr`** processes in the software network have been compiled and linked into the files **`mux.lku`** and **`wkr.lku`** respectively.

Figure 4 shows another method of defining a multi-tasking program running on a single transputer, using configuration text instead of C library calls. The configuration text for mapping the software network in figure 2 onto the hardware shown in figure 5: a single T805 with 1 Mbyte of memory, connected to the host by link 0. The software description is given in figure 6, and would replace the code in figure 3. In this example all the processes run on the root transputer.

Using the configuration text in this way creates processes statically instead of dynamically. Configuration-level processes can be be mapped onto one or several transputers without altering the C code or the software description.

```
/* Configuration example – 1 processor */

/* Hardware description */

T805 (memory = 1M) root;
connect root.link[0] to host; /* Host is pre–defined edge */

/* Software description */

#include ”software.inc”

/* Mapping description */

/* Define linked file units for processes */
use ”mux.lku” for mux;
rep i = 0 for 3
  use ”wkr.lku” for wkr[i];

/* Map processes to processors and external channels to edges */
rep i = 0 for 3
  place wkr[i] on root;
place mux on root;
place hostinput on host;
place hostoutput on host;
```

Figure 4 Configuration text – one processor

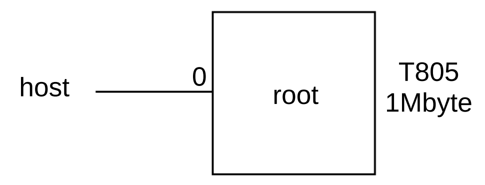


Figure 5 Hardware network – one processor

```
    /* Software description */

    /* Define process memory sizes and interfaces */
    process (stacksize = 2K, heapsize = 16k); /* Define defaults */
    rep i = 0 for 3
      process (interface (input in, output out, int id)) wkr[i];
    process (interface (input hostin, output hostout,
                        input in[3], output out[3])) mux;

    /* Define external channels, interconnections and parameters */
    input hostinput;  /* Host channel edges */
    output hostoutput;
    connect mux.hostin to hostinput;  /* Host channel connections */
    connect mux.hostout to hostoutput;
    rep i = 0 for 3
      {
        wkr[i] (id = i); /* Set worker process id parameter*/
        connect mux.in[i] to wkr[i].out;
        connect mux.out[i] to wkr[i].in;
      }
```

Figure 6 Configuration text – software description

Figure 7 shows the configuration text for mapping the same software network onto the hardware shown in figure 8; four T805s with 1 Mbyte of memory, arranged in a tree. In this example the **mux** process runs on the root transputer while the individual **wkr** processes run on each of the node transputers.

```
/* Configuration example – 4 processor tree */

/* Hardware description */

T805 (memory = 1M) root, p[3];
connect root.link[0] to host; /* Host link connection */
rep i = 0 for 3
  connect root.link[i + 1] to p[i].link[0];

/* Software description */

#include ”software.inc”

/* Mapping description */

/* Define linked file units for processes */
use ”mux.lku” for mux;
rep i = 0 for 3
  use ”wkr.lku” for wkr[i];

/* Map processes to processors and external channels to edges */
rep i = 0 for 3
  place wkr[i] on p[i];
place mux on root;
place hostinput on host;
place hostoutput on host;
```

Figure 7 Configuration text – four processor tree

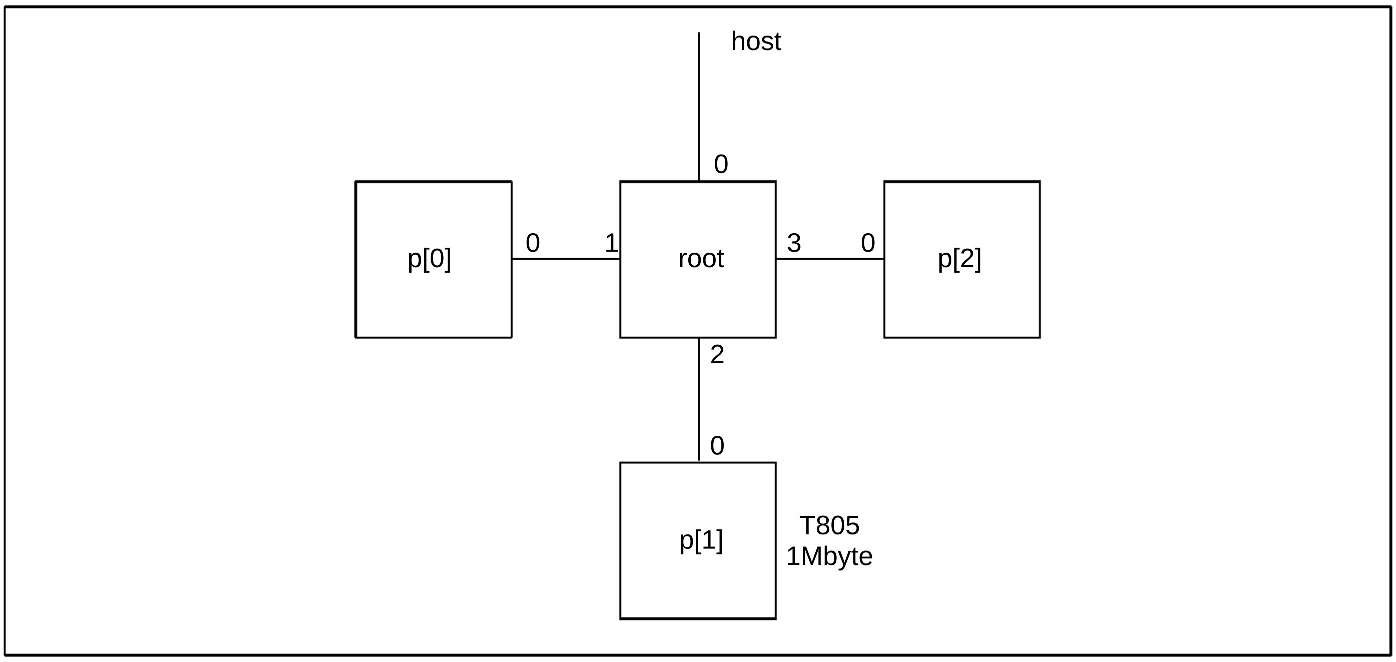


Figure 8 Hardware network – four processor tree

Only four lines of this configuration example are different from the previous example; the description of the software network is the same in both cases. Regardless of the target configuration it will always be possible to reconfigure the application for a single transputer, providing a useful first stage for target debugging.

The configuration tools can create a multi-processor program from this configuration description and the linked process units. Bootstraps and program distribution code are automatically added resulting in a program which will distribute itself across a transputer network with no additional programming required by the user. Multi-processor programs can be loaded from a host machine using the **`irun`** application loader, or can be converted into a range of industry standard EPROM file formats for programming into EPROM.

The interactive and post-mortem symbolic debugging tools support both forms of parallelism; using the parallel processes library and using the configuration text.

In the previous two examples, processes which communicate with each other are mapped onto transputers which are neighbors in the network. If the network looked like figure 9, then the configuration text would be as in figure 10. The software description and the mapping are the same; only the hardware description has been changed.

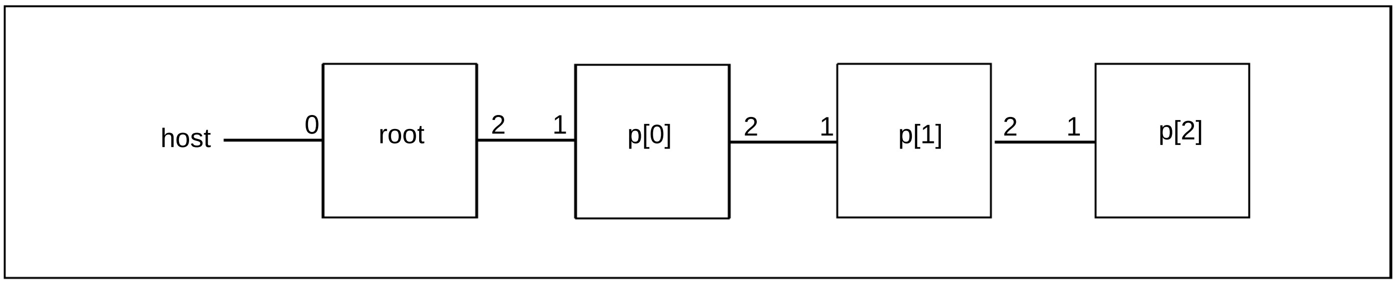


Figure 9 Hardware network – four processor pipeline

```
/* Configuration example – 4 processor pipeline */

/* Hardware description */

T805 (memory = 1M) root, p[3];
connect root.link[0] to host; /* Host link connection */
connect root.link[2] to p[0].link[1];
rep i = 1 for 2
   connect p[i–1].link[2] to p[i].link[1];

/* Software description */

#include ”software.inc”

/* Mapping description */

/* Define linked file units for processes */
use ”mux.lku” for mux;
rep i = 0 for 3
  use ”wkr.lku” for wkr[i];

/* Map processes to processors and external channels to edges */
rep i = 0 for 3
  place wkr[i] on p[i];
place mux on root;
place hostinput on host;
place hostoutput on host;
```

Figure 10 Configuration text – four processor pipeline

2.5 Support for embedded applications

The toolset has been designed to support the development of embedded applications. Features include facilities to:

- place code and data at particular places in memory,
- locate where functions and variables reside in memory,
- access the transputer instruction set efficiently from C,
- reduce the C run-time overhead to suit the application.

2.5.1 Placing code and data

At configuration level, a process consists of its code, stack, static and heap segments. The configurer allocates a place for each of these separate segments in the memory of the processor.

By default, the configurer allocates all the code and data segments to a contiguous default block of memory. The location of this default block and the order of the segments may be defined in the configuration description. The configuration description can also specify that particular segments of an application are to be allocated to particular places in memory outside the default block.

The compiler, linker and collector each will optionally produce a listing of how the various parts of an application are mapped into the segments and memory. A tool is provided that can read all these map files and produce a summary of the whole application, giving the locations of all the functions and static variables. Information is collated about code and data segments including start address and word size.

2.5.2 Access to the instruction set

ANSI C is a good language for writing embedded applications, since it combines the high level constructs of a programming language with low level access to the hardware through assembler inserts.

To make the access to some of the transputer's instructions even more effective, a number of special library functions have been defined which the optimizing compiler can render as in-line code. This removes the overhead of a library call, but it also gives the optimizer more information on what the program is doing.

Normally, when the optimizer sees a function containing some assembler code, it must make very conservative assumptions about the effect the code has on its surroundings, e.g. on static variables and parameters. By using the functions defined to access the instructions, the optimizer knows exactly what the effects will be and can make the minimally correct assumptions for the side-effects of the code.

The transputer instructions that can be accessed in this way include block moves, channel input and output, bit manipulation, CRC computations and some scheduling operations.

2.5.3 Runtime overhead reduction

In order to support the full ANSI C language, a significant run-time library is necessary. The toolset is supplied with another library, known as the *reduced library*, which does not support any of the features of the language which depend on a host. The host-dependent features include terminal and file system access and environment requests.

If some of the other features in ANSI C are not used, then it may be possible to reduce the overhead further by modifying the run-time initialization, the source of which is provided.

2.5.4 Assembler inserts

Within this implementation of the ANSI C language, assembler code can be written at any point to achieve direct access to transputer instructions for reasons of speed and code size. Full access is available to the transputer instruction set and C program variables.

2.5.5 Assembler

If there is no other way to obtain the code required, for example when writing customized bootstrap mechanisms, then the toolset contains an assembler as the final phase of the compiler. This can be invoked to assemble user-written code.

2.5.6 Dynamic loading of processes

The toolset can encapsulate the code and data of a process in a file called a *relocatable separately-compiled unit* or **`rsc`**. This format is suitable for dynamic loading and calling by another process. A set of functions is provided for this purpose. The **`rsc`** can be loaded from a file, loaded from memory, or input from a channel. The memory variant allows an **`rsc`** to be placed into ROM and executed if required. For example, if an application wishes to select a device driver to be placed in on-chip memory, then a number of possible drivers can be placed in ROM. The application can choose one for the occasion, copy it into low memory and execute it.

2.5.7 Bootstraps

The source code of the standard bootstraps are provided. The user can write bootstraps that are tailored to a specific application by using the standard ones as templates.

If the application is spread over several processors in a network, then it is possible to modify that part of the bootstrapping process which loads remote processors in order to perform special initializations on those processors before they receive any application code.

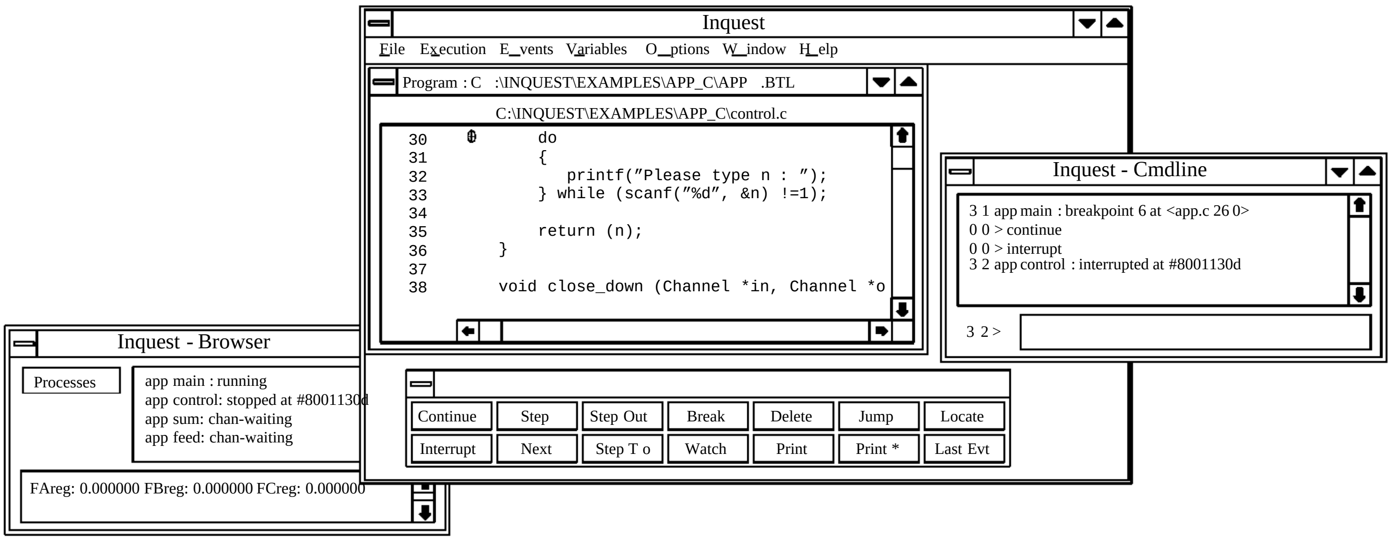
3 INQUEST windowing debugger

The INQUEST debugger can debug ANSI C programs either interactively or post-mortem. It supports single transputer applications and multi-processor networks. A user interface displays source code or disassembled code and enables the user to interact with the debugger by means of buttons, menus, a mouse and the keyboard. The interface is built using the X Window System and OSF/Motif for Sun-4s or Microsoft Windows for PCs.

The program being debugged may consist of any number of tasks (or threads of execution) some of which may be running while others are stopped or being single stepped. The host debugger program is asynchronous and holds a copy of the last stopped state of each thread, so values may be inspected by the host while the user program is running on the network. Multiple debugging windows may be opened to view different parts of the program simultaneously.

The INQUEST debugger has three debugging modes:

- interactive debugging, i.e. monitoring the application as it executes on the target processor;
- post-mortem debugging on the target processor when the application has stopped;
- post-mortem debugging on the host from a dump file.

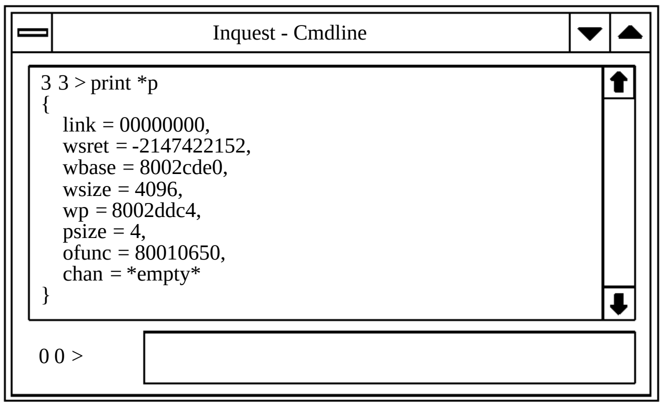


```
        do
        {
           printf("Please type n : ");
        } while (scanf("%d", &n) !=1);

        return (n);
    }

    void close_down (Channel *in, Channel *o
```

```
3 1 app main : breakpoint 6 at <app.c 26 0>
0 0 > continue
0 0 > interrupt
3 2 app control : interrupted at #8001130d
3 2 >
```

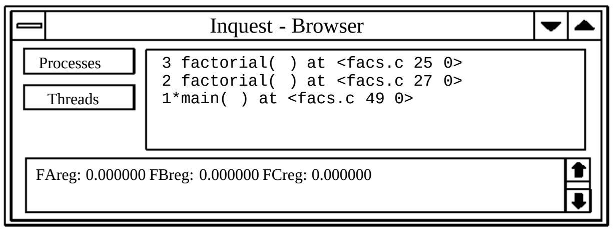
```
app main : running
app control: stopped at #8001130d
app sum: chan-waiting
app feed: chan-waiting
FAreg: 0.000000 FBreg: 0.000000 FCreg: 0.000000
```

Figure 11 The Microsoft Windows debugger display



```
3 3 > print *p
{
   link = 00000000,
   wsret = -2147422152,
   wbase = 8002cde0,
   wsize = 4096,
   wp = 8002ddc4,
   psize = 4,
   ofunc = 80010650,
   chan = *empty*
}
0 0 >
```

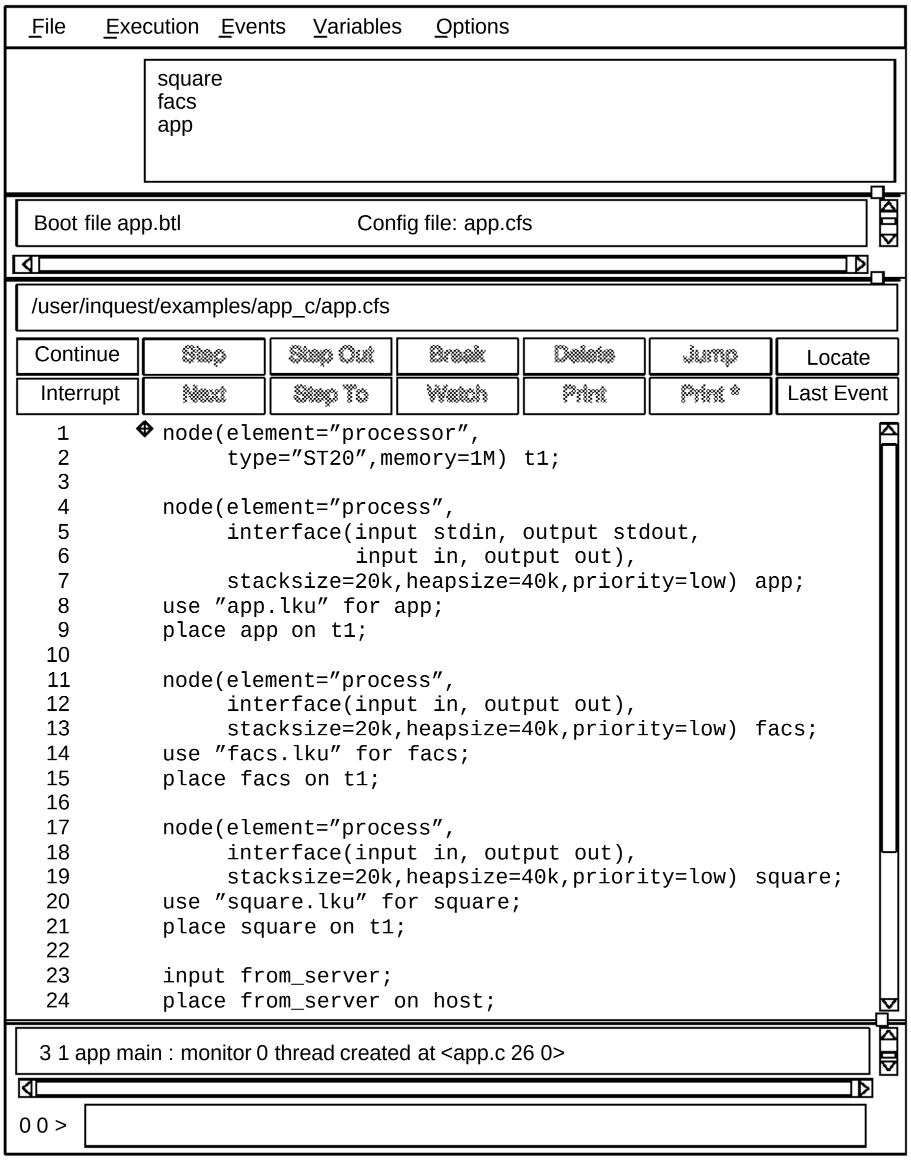
Figure 12 Displaying a structured variable on Microsoft Windows



```
3 factorial( ) at <facs.c 25 0>
2 factorial( ) at <facs.c 27 0>
1*main( ) at <facs.c 49 0>
```

FAreg: 0.000000 FBreg: 0.000000 FCreg: 0.000000

Figure 13   A Microsoft Windows stack trace



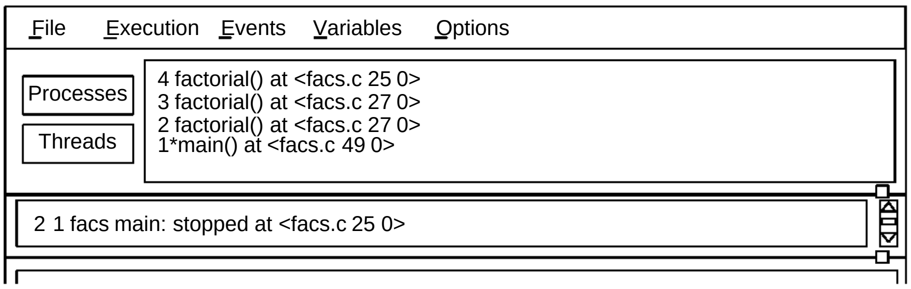
Boot file app.btl   Config file: app.cfs

/user/inquest/examples/app_c/app.cfs

```
node(element="processor",
     type="ST20",memory=1M) t1;

node(element="process",
     interface(input stdin, output stdout,
               input in, output out),
     stacksize=20k,heapsize=40k,priority=low) app;
use "app.lku" for app;
place app on t1;

node(element="process",
     interface(input in, output out),
     stacksize=20k,heapsize=40k,priority=low) facs;
use "facs.lku" for facs;
place facs on t1;

node(element="process",
     interface(input in, output out),
     stacksize=20k,heapsize=40k,priority=low) square;
use "square.lku" for square;
place square on t1;

input from_server;
place from_server on host;
```

3 1 app main : monitor 0 thread created at <app.c 26 0>

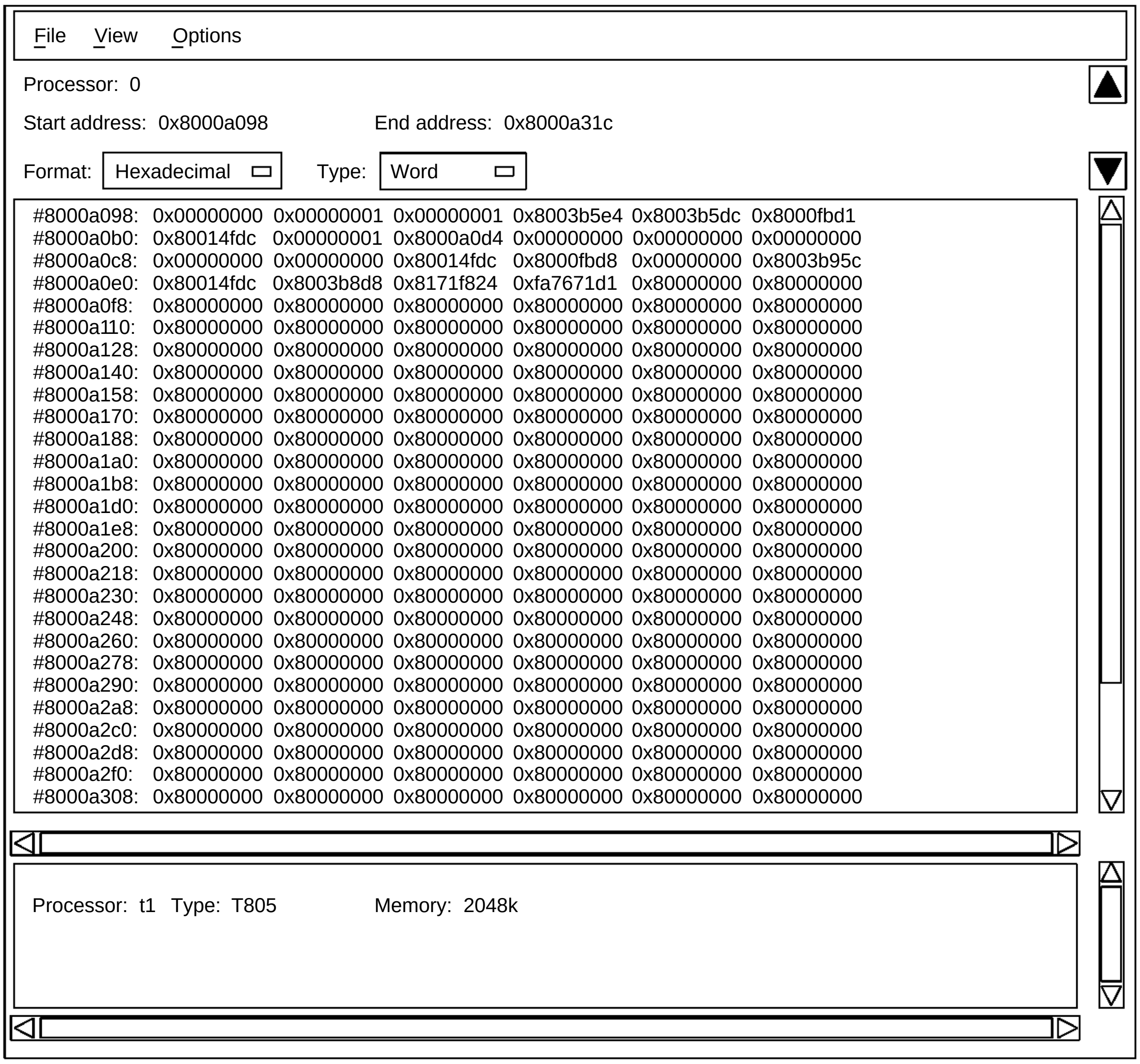
Figure 14   The X-Windows debugger display



4 factorial() at <facs.c 25 0>
3 factorial() at <facs.c 27 0>
2 factorial() at <facs.c 27 0>
1*main() at <facs.c 49 0>

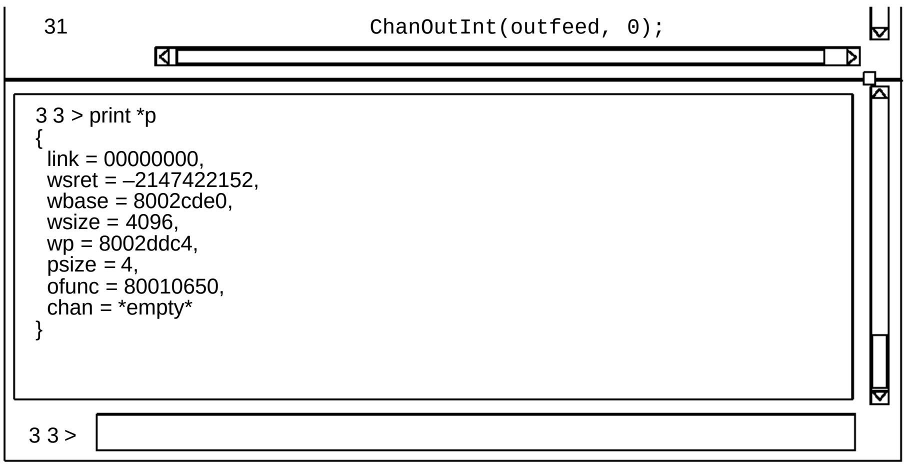
2 1 facs main: stopped at <facs.c 25 0>

Figure 15   An X-Windows stack trace



```
Processor:  0
Start address:  0x8000a098            End address:  0x8000a31c
Format:  Hexadecimal        Type:  Word

#8000a098:  0x00000000  0x00000001  0x00000001  0x8003b5e4  0x8003b5dc  0x8000fbd1
#8000a0b0:  0x80014fdc  0x00000001  0x8000a0d4  0x00000000  0x00000000  0x00000000
#8000a0c8:  0x00000000  0x00000000  0x80014fdc  0x8000fbd8  0x00000000  0x8003b95c
#8000a0e0:  0x80014fdc  0x8003b8d8  0x8171f824  0xfa7671d1  0x80000000  0x80000000
#8000a0f8:  0x80000000  0x80000000  0x80000000  0x80000000  0x80000000  0x80000000
#8000a110:  0x80000000  0x80000000  0x80000000  0x80000000  0x80000000  0x80000000
#8000a128:  0x80000000  0x80000000  0x80000000  0x80000000  0x80000000  0x80000000
#8000a140:  0x80000000  0x80000000  0x80000000  0x80000000  0x80000000  0x80000000
#8000a158:  0x80000000  0x80000000  0x80000000  0x80000000  0x80000000  0x80000000
#8000a170:  0x80000000  0x80000000  0x80000000  0x80000000  0x80000000  0x80000000
#8000a188:  0x80000000  0x80000000  0x80000000  0x80000000  0x80000000  0x80000000
#8000a1a0:  0x80000000  0x80000000  0x80000000  0x80000000  0x80000000  0x80000000
#8000a1b8:  0x80000000  0x80000000  0x80000000  0x80000000  0x80000000  0x80000000
#8000a1d0:  0x80000000  0x80000000  0x80000000  0x80000000  0x80000000  0x80000000
#8000a1e8:  0x80000000  0x80000000  0x80000000  0x80000000  0x80000000  0x80000000
#8000a200:  0x80000000  0x80000000  0x80000000  0x80000000  0x80000000  0x80000000
#8000a218:  0x80000000  0x80000000  0x80000000  0x80000000  0x80000000  0x80000000
#8000a230:  0x80000000  0x80000000  0x80000000  0x80000000  0x80000000  0x80000000
#8000a248:  0x80000000  0x80000000  0x80000000  0x80000000  0x80000000  0x80000000
#8000a260:  0x80000000  0x80000000  0x80000000  0x80000000  0x80000000  0x80000000
#8000a278:  0x80000000  0x80000000  0x80000000  0x80000000  0x80000000  0x80000000
#8000a290:  0x80000000  0x80000000  0x80000000  0x80000000  0x80000000  0x80000000
#8000a2a8:  0x80000000  0x80000000  0x80000000  0x80000000  0x80000000  0x80000000
#8000a2c0:  0x80000000  0x80000000  0x80000000  0x80000000  0x80000000  0x80000000
#8000a2d8:  0x80000000  0x80000000  0x80000000  0x80000000  0x80000000  0x80000000
#8000a2f0:  0x80000000  0x80000000  0x80000000  0x80000000  0x80000000  0x80000000
#8000a308:  0x80000000  0x80000000  0x80000000  0x80000000  0x80000000  0x80000000

Processor:  t1   Type:  T805          Memory:  2048k
```

Figure 16   Displaying memory contents on X-Windows



```
                ChanOutInt(outfeed, 0);

3 3 > print *p
{
  link = 00000000,
  wsret = –2147422152,
  wbase = 8002cde0,
  wsize = 4096,
  wp = 8002ddc4,
  psize = 4,
  ofunc = 80010650,
  chan = *empty*
}

3 3 >
```

Figure 17   Displaying a structured variable on X-Windows

3.1 Interactive debugging

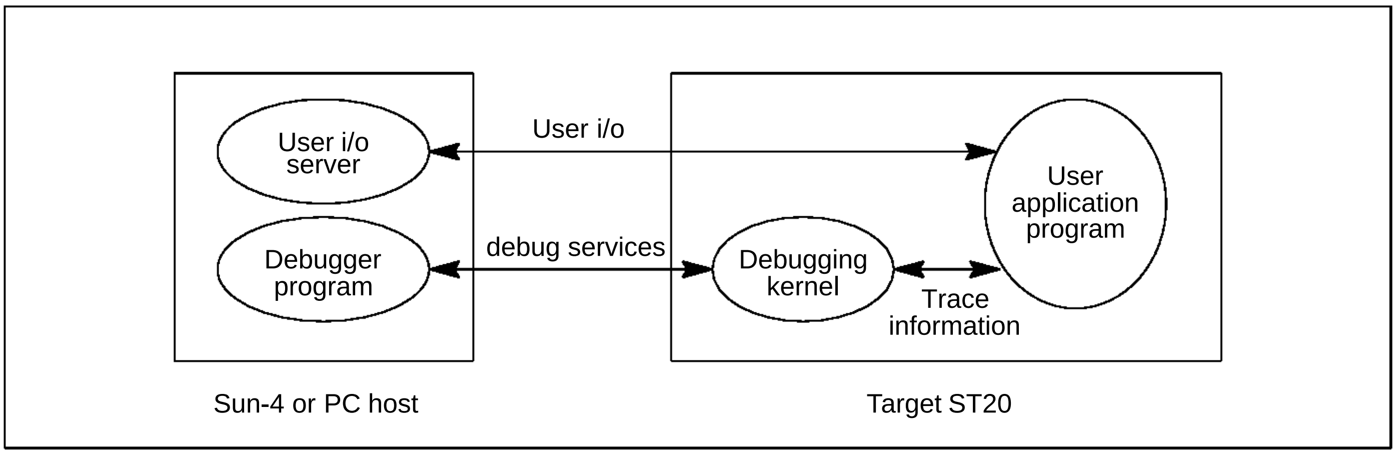


Figure 18 Interactive debugger architecture

The interactive debugger consists of a host-based symbolic debugger and a distributed debugging kernel that is configured into the application program on each transputer.

The interactive debugger provides the following features:–

- A break point facility that can be used on all or selected threads of execution.
- A single stepping facility that allows a thread of execution to be single stepped at the source level or at the assembly code level.
- A watch point facility that enables the program to be stopped when variables are to be written to or read from.
- A facility to find the threads of execution of a program and set break points on them.
- A stack trace facility.
- A facility to monitor the creation of threads of execution.
- Commands to print the values of variables and display memory.
- A simple interpreter to enable C aggregate types (i.e. structures and arrays) to be displayed.
- A programmable command language that allows complex break points and watch points to be set and enables debugging scripts to be generated.
- A source and object browser to select a process, a thread and source code.
- An interface to the real-time operating system for dynamic code loading and thread creation.

3.2 Post-mortem debugging

The post-mortem debugger provides the following features:–

- A source and object browser to select a process, a thread and source code.
- Commands to print the values of variables and display memory.
- A simple interpreter to enable C aggregate types (i.e. structures and arrays) to be displayed.
- Creation of dump files.
- Debugging from a dump file.

4 Execution analysis tools

Three profiling tools are supplied for analyzing the behavior of application programs; the execution profiler, the utilization monitor, and the test coverage and block profiling tool. The execution profiler estimates the time spent in each function and procedure, the processor idle time and various other statistics. The utilization monitor displays a Gantt chart of the CPU activity of the processor as time progresses. The test coverage and block profiling tool counts how many times each block of code is executed.

The monitoring data is stored in the target processor's memory, so the profiling tools have little execution overhead on the application. After the program has completed execution, the monitoring data is extracted from the processor and is analyzed to provide displays on the program execution.

4.1 Execution profiler

The execution profiler gives an analysis of the total time spent executing each function on each processor.

It provides the following information on program execution:–

- The percentage time spent executing each low priority function.
- The percentage time spent executing at high priority.
- The percentage idle time of the processor.

The execution of the user program is monitored by a profiling kernel. The presence of the profiler kernel will slow the execution of the program by less than 5%.

```
Processor ”Root”
Idle time 35.3% (19516)
High time 0.1% (37)
Wptr Misses 0
Iptr Misses 0
Resolution 4

-----------------------------------------------------------------------------
Process ”ex” (99.9% processor) (35.666s)
Stack 100.0% (35666)    Heap 0.0% (0)    Static 0.0% (0)
Function Name                               | Process | Processor |Samples
-----------------------------------------------------------------------------
libc.lib/getc                               |    11.4 |      11.4 |4081
cc/pp.c/pp_rdch0                            |    10.1 |      10.1 |3605
cc/bind.c/globalize_memo                    |     6.9 |       6.9 |2467
cc/pp.c/pp_process                          |     4.3 |       4.3 |1525
cc/pp.c/pp_rdch3                            |     4.2 |       4.2 |1497
cc/pp.c/pp_rdch2                            |     3.9 |       3.9 |1380
cc/pp.c/pp_rdch1                            |     3.8 |       3.8 |1354
cc/pp.c/pp_rdch                             |     3.5 |       3.5 |1252
cc/pp.c/pp_nextchar                         |     3.3 |       3.3 |1189
cc/pp.c/pp_checkid                          |     3.2 |       3.2 |1150
cc/lex.c/next_basic_sym                     |     2.7 |       2.7 |979
libc.lib/strcmp                             |     2.3 |       2.3 |812
libc.lib/DummySemWait                       |     2.2 |       2.2 |784
libc.lib/sub_vfprintf                       |     1.7 |       1.7 |617
```

Figure 19 Example output from the execution profiler

4.2 Utilization monitor

The utilization monitor shows in graphical form the utilization of the processor over the time of the program execution. This is displayed by an interactive program that draws a chart of processor execution against time using X Window System and OSF/Motif on a Sun-4 or Microsoft Windows on a PC.

As with the execution profiler, the user program is monitored by a profiling kernel. The kernel will slow the program by less than 5%.

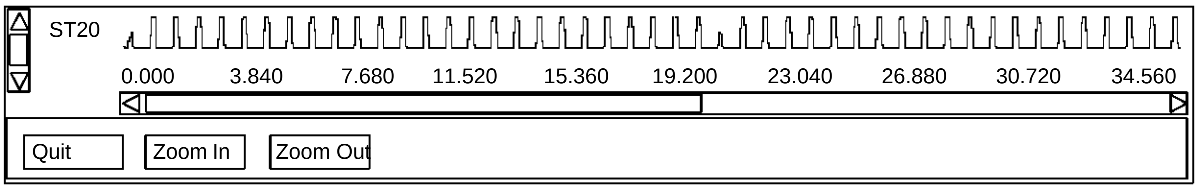


Figure 20 Example X-Windows display from the utilization monitor

4.3 Test coverage and block profiling tool

This tool monitors test coverage and performs block profiling for an application which has been run on target hardware.

This tool is able to:

- provide an overall test coverage report;
- provide per module test coverage reports;
- accumulate a single report from multiple test runs;
- provide a detailed basic block profiling output by creating an annotated program listing;
- provide output that can be fed back into the compiler as a part of its optimization process.

The application program (compiled with the appropriate compiler option) is run and accumulates the counts in the memory of the target processor. The tool is used to extract the results and save, accumulate or display them. This application writes the counts into the code area, so the tool cannot be used with code running from ROM.

```
Writing coverage file ”square.v” – 40% coverage
Writing coverage file ”comms.v” – 14% coverage
Writing coverage file ”app.v” – 75% coverage
Writing coverage file ”control.v” – 36% coverage
Writing coverage file ”feed.v” – 33% coverage
Writing coverage file ”sum.v” – 40% coverage
Total coverage for bootable 39% over 1 run
```

Figure 21 Example test coverage summary report

The following is an example of the contents of a coverage file:

```
|/*
| * facs.c
| *
| * generate factorials
| *
| */
|
|#include <stdio.h>
|#include <stdlib.h>
|#include <process.h>
|#include <channel.h>
|#include <misc.h>
|#include ”comms.h”
|
                |#define TRUE 1
                |#define FALSE 0
                |
                |


                |        /*
                |         * compute factorial
                |         *
                |         */
                |
                |int factorial(int n)
96              |{
                |  if (n > 0)
74              |    return ( n * factorial(n–1));
                |  else
22              |    return (1);
                |
                |}
                |
                |int main()
1               |{
                |   Channel *in, *out;
                |   int going = TRUE;
                |
                |   in = get_param(1);
                |   out = get_param(2);
                |
                |   while (going)
27              |   {
                |      int n, tag;
                |
                |      tag = read_chan (in, &n);
                |      switch (tag)
                |      {
                |         case DATA: {
22              |            send_data (out, factorial(n));
                |            break;
                |         }
                |         case NEXT: {   /* start a new sequence */
4               |            send_next (out);
                |            break;
                |         }
                |         case END: {    /* terminate */
1               |            going = FALSE;
                |            send_end (out);
                |         }
                |      }
                |      }
                |   }
                |}
                |

######################
# Summary of results #
######################
Source file      : facs.c
Number of runs   : 1
Processors       : All
>From linked unit : facs.lku

Top 10 Blocks!!
```

```
Line 25 – 96 times
Line 27 – 74 times
Line 42 – 27 times
Line 29 – 22 times
Line 49 – 22 times
Line 53 – 4 times
Line 34 – 1 time
Line 57 – 1 time

Total number of basic blocks 8
Basic blocks not executed    0
Coverage 100%
```

5 Host interface and AServer

The host interface is provided by the AServer. This can be used simply as an application loader and host file server, invoked by the **`irun`** command. The INQUEST tools have their own commands which in turn load **`irun`** in order to load the application. The AServer may also be used to customize the host interface if required.

5.1 The application loader – `irun`

`irun` performs three functions, namely:

1. to initialize the target hardware;
2. to load a bootable application program onto the target hardware via the hardware serial link;
3. to serve the application, i.e. to respond to requests from the application program for access to host services, such as host files and terminal input and output.

These steps are normally performed when **`irun`** is invoked.

5.2 AServer

The AServer (Asynchronous Server) system is a high performance interface system which allows multiple processes on a target device to communicate via a hardware serial link with multiple processes on some external device. The AServer software acts as a standard interface which is independent of the hardware used. A simple example is shown in Figure 22, in which the external device is the host.

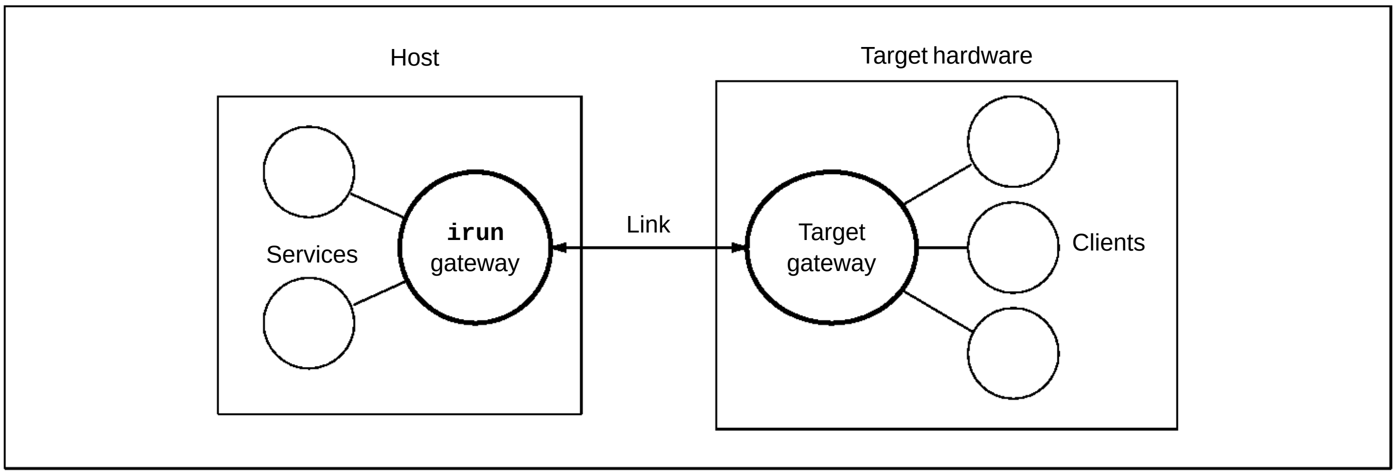


Figure 22 A simple software host-target interface

The AServer is a collection of programs, interface libraries and protocols that together create a system to enable applications running on target hardware to access external services in a way that is consistent, extensible and open. The software elements provided are:

- a target gateway which runs on the target;
- an **`irun`** gateway which runs on the host;
- an **`iserver`** service which runs on the host;
- an **`iserver`** converter which runs on the target;
- a library of interface routines for use by client and service processes;

- simple example services.

5.3 AServer features

This type of architecture offers a number of advantages:

1 The AServer handles multiple services.

A number of services may be available on one device, handled by a single gateway. For example, new services may be added without modifying a standard server.

2 The AServer handles multiple clients.

Any process on any processor in the target hardware may open a connection to any service. The process opening the connection is called the client. Several clients may access the same service. The gateway will automatically start new services as they are requested.

3 Services are easy to extend.

The AServer enables users to extend the set of services that are available to a user's application. The AServer provides the core technology to allow users to create new services by providing new processes. For example, the **`iserver`** service provides terminal text i/o, file access and system services, which may be expanded by adding new AServer service processes such as a graphics interface.

4 AServer communications can be fast and efficient.

The communications over the link between gateways use mega-packets, which make efficient use of the available bandwidth. Messages between the client and the service are divided into packets of up to 1 kbyte. The packets are bundled into mega-packets to send over the hardware serial link. Packets from different clients and services can be interleaved to reduce latency.

5 AServer communications are independent of hardware.

When an AServer connection has been established the process can send data messages of arbitrary length to the service it is connected to, receive data messages of arbitrary length and disconnect from the service. The gateways are responsible for building and dividing mega-packets and complying with hardware protocols.

6 ANSI C Toolset product components

6.1 Documentation

- ANSI C toolset user guide
- Toolset reference manual
- ANSI C language and libraries reference manual
- INQUEST user and reference manual
- INQUEST debugger tutorial
- AServer programmer's guide
- Delivery manual

6.2 Software Tools

`icc`, **`ilink`**, **`ilibr`** – ANSI C compiler, linker and librarian

`icconf`, **`icollect`** – configuration tools

`imakef` – makefile generator

AServer including **`irun`** – host server program

`ilist`, **`imap`** – binary lister program and memory map lister

`ieprom`, **`iemit`**, **`imem450`** – EPROM and memory interface programming tools

`inquest` debugger

`imon`, **`iprof`**, **`iline`** – execution analysis tools

`rspy` – network analyzer

6.3 Software libraries

`libc.lib` – Full ANSI library plus parallel support

`libcred.lib` – Reduced library for embedded systems

`centry.lib` – Mixed language support library

7 Product Variants

7.1 IMS D7414 IBM PC version

All code building tools are provided in a form which will run on the host machine. Running the code, with or without interactive debugging, requires access to the transputer network. The INQUEST post-mortem debugger requires access to the transputer network unless it is using a post-mortem dump file. The profiling tools also require access to the target transputer network.

7.1.1 Operating requirements

The following configuration is required:

- IBM PC with 386 or 486 processor or compatible and at least 8 Mbytes memory;
- DOS 5.0 or later and Windows 3.1 running in enhanced mode;
- 15 Mbyte of free disk space;
- A transputer development board including associated board support software.

7.1.2 Distribution media

Software is distributed on 1.44 Mbytes 3.5 inch IBM format floppy disks.

7.2 IMS D4414 Sun 4 version

All code building tools are provided in a form which will run on the host machine. Running the code requires access to the transputer network. The INQUEST debugger requires access to the transputer network unless it is using a post-mortem dump file. The profiling tools also require access to the target transputer network.

7.2.1 Operating requirements

For hosted cross-development and debugging the following configuration is required:

- A Sun 4 workstation or server with 1/4 in. tape drive capable of reading QIC-24 format;
- One of:
  - SunOS version 4.1.3, or compatible versions or
  - Solaris 2.4, or compatible versions.
- 17 Mbytes of free disk space;
- An X11 Release 4 (or later) server or OpenWindows 3;
- A transputer development board including associated board support software.

7.2.2 Distribution media

Sun 4 software is distributed on DC600A data cartridges 60 Mbyte, QIC-24, tar format.

8 Support

TRAMs and transputer toolset development products are supported worldwide through SGS-THOMSON Sales Offices, Regional Technology Centers, and authorized distributors.

9 Ordering Information

Description	Order number
ANSI C Toolset for 386 PC.	IMS D7414A
ANSI C Toolset for Sun 4.	IMS D4414A