Clinical Neutron Therapy System Implementation

Jonathan Jacky[1]

Radiation Oncology, Box 356043
University of Washington
Seattle, WA 98195-6043

Technical Report 2001-03-01

[1]email jon@u.washington.edu, telephone (206)-598-4117, fax (206)-598-6218

Abstract

This report describes the detailed design and code of the control program for the isocentric treatment unit of the Clinical Neutron Therapy System (CNTS). It is primarily intended for software developers. It describes design features which should be preserved when the program is modified, and explains some unobvious implementation details. The control program was developed from a detailed design expressed in the Z notation. It is coded in C. We used only the ANSI Standard C library and a few libraries needed to use the real-time operating system and the X window system (`Xlib` only, not Motif or any other toolkits). The program text (C data and functions) is partitioned into modules that make it easy for us to produce different program versions that run in different environments. The executing program comprises several concurrent tasks, coordinated by appropriate communication and synchronization methods. The behavior of the user interface is determined by a table. The graphical user interface is separable; the program can also read text commands from a keyboard or script file (for test automation). Most of the code was developed on a general-purpose workstation, simulating tasking in a single process while reading and writing simulated device controller commands and data from files. Many of the program source code files are re-used (without changes) in utility programs that run on a workstation.

Contents

Introduction

This report describes the detailed design and code of the control program for the isocentric treatment unit of the Clinical Neutron Therapy System (CNTS). It is primarily intended for software developers. It describes design features which should be preserved when the program is modified, and explains some unobvious implementation details. The control program was developed from a detailed design expressed in the Z notation. It is coded in C. We used only the ANSI Standard C library and a few libraries needed to use the real-time operating system and the X window system (`Xlib` only, not Motif or any other toolkits). The program text (C data and functions) is partitioned into modules that make it easy for us to produce different program versions that run in different environments. The executing program comprises several concurrent tasks, coordinated by appropriate communication and synchronization methods. The behavior of the user interface is determined by a table. The graphical user interface is separable; the program can also read text commands from a keyboard or script file (for test automation). Most of the code was developed on a general-purpose workstation, simulating tasking in a single process while reading and writing simulated device controller commands and data from files. Many of the program source code files are re-used (without changes) in utility programs that run on a workstation.

This report frequently refers to particular program source program files (C `.h` and `.c` files) and other project files such as `Makefile`. This report provides an overview, rationale, and high-level explanation of the source code but does not contain the code itself. Much detailed description that appears in the headers to the program source files is not repeated here. Readers who plan to revise the program or perform detailed analyses must also have access to the code, of course.

We chose the C programming language for this project because it is the native language of the real-time operating system and the X window system. These two systems are distributed (in part) as C header files, and all their documentation shows examples in C. Based on our experience with other projects, we expect that the practical difficulties of mating code written in any other language to these two systems would far outweigh any putative advantages other languages might provide. We understand the limitations and pitfalls of C very well and they have not caused difficulties for us. Programming errors discovered during development and use were not language-related and would have occurred if we had used another language.

We limited ourselves to C, the ANSI Standard C library, `Xlib`, and a few real-time operating system libraries in order to achieve stability and long product lifetime. These have been in wide use for many years, so the worst defects have been shaken out and the remaining limitations are well understood. We rejected other products which seemed overly complex, were still evolving rapidly, or which could make us dependent on particular suppliers.

Several reports describe other aspects of this project. To assess the correctness, safety, and fitness for use of the program described here it is necessary to consider all of their contents as well.

The facility description [17] provides an overview of the entire Clinical Neutron Therapy System (not just the control system). The reference manual [7] describes the control program whose internals are described here. It is sufficiently thorough to serve as the informal (natural language) specification for the program. The operations manual [3] explains how to install, configure, and maintain the control computer and control program. The therapist's guide [8] provides an orientation to the control console, screen and keyboard and explains how to use the control program to treat patients.

Much of the code was developed from a formal specification in Z [18, 2], a machine-readable notation for logic, set theory and arithmetic. The rationale for using a formal specification appears in [5]. The formal specification serves as the detailed design; we used no other design notation (besides English). The full text appears in [6]; excerpts and commentary appear in [10, 2, 11, 4].

Most project documents are available from `http://radonc.washington.edu/physics/cnts/`.

Chapter 1

Tracing from requirements to code

It is possible to trace from the requirements expressed in an early prose specification [9] to the detailed design in the formal specification [6] and finally to the implementation in the code.

The formal specification in [6] contains cross references that relate many of the Z formulas to section and page numbers in the corresponding prose requirements in [9].

We used similar names in the documents and code to help make the derivation clear. Modules whose contents are derived from the formal specification have names that begin with `z`: the `zfield` module implements *Field* state schema and the operations on it (section 6 in the formal specification). Types in the implementation that are derived from the formal specification have names that begin with `Z_`: the enumerated type `Z_Setting` in `znames.h` implements *setting* (section 3.1 in the formal specification). Other identifiers are made as similar as possible, subject to the requirements and conventions of each notation. The therapy parameter named GANTRY (prose requirements) is modelled by the set element named *gantry* (formal specification) and implemented by the enumerated type value named `gantry` (code).

The nomenclature in the formal specification and code is derived from the early requirements document [9]. We changed some nomenclature when we wrote the reference manual [7]. The *settings* (early documents and code) became *therapy parameters* (reference manual) and *measured settings* (early documents and code) became *actual settings* (reference manual).

The implementation of the detailed design (in Z notation) to code (in C) usually employs methods described in a textbook (chapter 28 in [2]).

Chapter 2

Modular structure

The program contains about 16,000 lines of C programming language code, divided among more than 90 source files. The source files are organized into more than 40 modules.

"Module" is not a C programming language construct; a module is a set of C files that we identify as one unit in our design.

Most modules consist of one header file (`.h` file) and one `.c` file with the same name: the `zprescription` module contains `zprescription.h` and `zprescription.c`. A few modules consist of only one file, for example `filenames.h` or `user.c`.

To design the *modular structure* of the program, we choose which modules to have, determine which code goes into each module, and work out the dependencies between modules.

We chose our particular modular design to make it easy for us to produce different program versions that run in different environments. We have to bring the system up in stages. At each development stage we must have a working, testable program version that provides a subset of the full functionality. We separate off the graphical user interface (GUI) so that we can have a command line interface for test automation, and we separate out the real time and multitasking capabilities so that we can do much development on an ordinary workstation.

The best source on modular design is chapter 5 in the textbook by Lamb [13], see also the papers by Parnas [16, 15].

Two strategies guide our modular design: *information hiding* and limiting *dependencies.*

2.1 Information hiding

Some modules are designed to achieve *information hiding.* Each of these modules contains some information which it uses but which other modules should not be able to change or even read. This information is said to be "hidden" in the module; the module is said to *hide* that information, which constitutes the module's "secret". For example, the `zprescription` module hides the organization of the in-memory prescription database and the format of the prescription data files.

The purpose of information hiding is to make the program easy to understand and change by localizing information in one place. When changes are made to information that is hidden in one module, the effects of the change can be completely understood by analyzing that one module in isolation from all the others. It is not necessary to consider whether some other module might be affected. For example, to change the organization of the in-memory database, or change the format of the disk files, we would only need to change one module: `zprescription`. We can be confident that no other modules would need to be changed[1].

To achieve information hiding, we separate the module's visible *interface* from its hidden *implementation.* A module is said to *provide* the items made visible in its interface. For example, many modules contain some data structures and the functions that operate on them. The function declarations are the interface and the data structures and function definitions are the implementation. As long as the items provided by the module interface (the function declarations) are not changed, it is possible to change the module implementation (data structures and function definitions) without having to change any other modules.

Information hiding results in a modular structure organized around access to data, not tasks. We have cases where a single module provides functions that are used by several different tasks.

The C programming language provides a way to separate the interface from the implementation: put the interface code in the module's `.h` file (header file) and the implementation code in the `.c` file. Information in a header file can be made available to other modules (certain selected ones, not all others). Information in a `.c` file is hidden from all other modules.

For example, the interface `zprescription.h` declares functions that access the in-memory prescription database, and other functions that load information from disk files into the in-memory database. The implementation `zprescription.c` defines variables that hold the contents of the in-memory database and contains function definitions that read the disk files.

We declare other items besides functions in header files: named constants (using `#define`), types (using `typedef`), macros (also using `#define`), and even some global variables. Declaring

[1] Changing the *contents* (rather than the organization or format) of the in-memory database or data files may affect the *behavior* controlled by code defined in other modules, of course.

global variables in header files requires un-obvious C programming techniques (appendix A).

2.2 Dependencies

When one module contains identifiers that are declared or defined in in another module's interface, we say that the first module *depends on* the second. The identifiers can be names of constants, variables, functions, or types, or can be macro text.

Dependencies are implemented in C by file inclusion. For example, the `zprescription` module depends on the `znames` module, so `zprescription.c` contains the directive `#include znames.h`. This directive enables code in `zprescription.c` to refer to identifiers declared in `znames.h`. For example, it can call functions declared there.

This makes dependencies explicit. To see what modules a module depends on, simply inspect the list of `#include` directives near the beginning of the `.c` file.

There are also dependencies among `.h` files, where one `.h` file contains identifiers defined or declared in another. However we usually avoid *nested includes* where `#include` directives appear in `.h` files. This prevents including the same item more than once when a `.c` file includes several `.h` files (we rejected alternatives involving conditional compilation as too complicated). Instead, dependencies among `.h` files are handled by order of inclusion in `.c` files. In every `.c` file, each header file must be included before any header files that depend on it. For example `znames.h` defines items used in `zprescription.h`, so `#include znames.h` must precede `#include zprescription.h` in any `.c` file that uses the `zprescription` module.

Three exceptions to the policy against nested includes are `util.h`, `xdefs.h`, and `rtdefs.h`, which include header files for the ANSI Standard C library, the X window system, and the real-time operating system, respectively. No other modules include the ANSI C header files (etc.), they all include `util.h` (etc.) instead. This avoids the problem of including the same header file more than once.

In this report the terms *uses*, *depends on* and *includes* are synonyms (when they are applied to modules). (This is not exactly the same meaning for *uses* as in Lamb [13], pps. 43 – 44. Our *uses* seems to be the same as Lamb's $uses_V$).

Dependencies are not transitive: if A depends on B and B depends on C, A need not depend on C. This is fortunate because it makes it possible to build up the program from separable, interchangeable parts. For example, the graphic utilities module `display` depends on the X window system library interface module `xdefs`. However the MLC graphics module `mlcdisp` depends on `dis-`

`play` but does not depend on `xdefs`. Therefore we could replace the X window system with some other graphics system with a quite different interface, without having to change the MLC graphics module or any of the other application graphics code.

Dependencies need not reflect data flow. For example data flows from `keyboard` to `zconsole` but neither depends on the other. Instead, the data flow is mediated by `user`, which depends on both (see p. 57 in Lamb [13]).

Dependencies that seem to be circular can be broken by splitting module contents between `.h` and `.c` files. For example `graphics` uses global variables in `zconsole`, while `zconsole` calls functions in `graphics`. This is not circular because `graphics.c` and `zconsole.c` both include `graphics.h` and `zconsole.h`, but the two `.h` files are independent; neither uses indentifiers declared in the other (see pps. 43–44 in Lamb [13]).

We use the `make` utility to compile and link the program, so we have to encode the dependencies among all the modules in the `Makefile`. The `Makefile` serves as an up-to-date record of the dependencies. (We maintain our `Makefile` by hand; we do not use a tool to keep track of dependencies automatically.)

2.3 Module groups

We try to minimize dependencies because they make the program more difficult to understand and change. In particular, they make it difficult to produce different versions suited for different environments.

There are two levels of information hiding. At the module level, information that appears only in a `.c` file is hidden from all other modules. At the program level, information that appears in an `.h` file is hidden from other modules that do not include that `.h` file. So we can selectively hide entire modules by using include directives sparingly, only where needed.

By using include directives selectively, we organize modules into several groups, where there may be many dependencies within each group or between particular groups but few or none between other groups. We should try to preserve these groups and not create new dependencies.

Here are descriptions of the most important module groups, including some of the modules they contain, and some of the dependencies within and between them. This is not a complete list but it reveals the main skeleton of the modular design. Consult the `Makefile` to confirm the dependencies currently in effect.

In the following discussion, module names are shown in typewriter font in lower case: `zfield`. Module group names are shown in typewriter font capitalized: `State`.

2.3.1 Utilities

These modules provide pervasive items that are independent of the therapy control application (they do not refer to data that are specific to this application):

- `switches` configuration options set at compile time
- `util` constants, types, macros, utility functions

`util.h` is one of the few `.h` files that includes other `.h` files. It includes all the header files for the ANSI Standard C library (`stdio.h`, `math.h`, etc.) and `switches.h`.

No other modules include the ANSI header files or `switches.h`. All other modules include `util.h` instead. Therefore all other modules depend on `Utilities` so we need not mention this explicitly in the following sections.

2.3.2 Real Time (RT)

These modules provide the real-time operating system and communication and synchronization between tasks:

- `rtdefs` real-time operating system
- `pipe` message pipes
- `delay` time delays
- `tasks` task management

The `rtdefs` module contains the header file `rtdefs.h` only. `rtdefs.h` is one of the few `.h` files that includes other `.h` files. It includes all the header files we need for the real-time operating system including `taskLib.h`, `semLib.h`, device driver headers and many others.

2.3.3 Messages

These modules handle event log messages and controller communication messages:

- `circbuf` circular buffers for event log messages and controller messages
- `messages` event log messages

`circbuf` uses semaphores to control access to the message buffers so it depends on `RT`. `messages` depends on `circbuf`. `messages` also depends on `graphics` (below) because the display may need to be updated when a new message appears.

Many modules use `Messages`, including the `zintlk` module and modules from the groups `Controllers`, `UI`, `Main`.

2.3.4 Constants

These modules provide names and constants for the therapy control application:

- `znames` parameter names etc.
- `zvalues` ranges, tolerances, units, discrete value encodings, etc.

`znames` uses no other modules (except `util`). `zvalues` uses `znames`.

2.3.5 Prescription

This module provides the prescription database. It hides the organization of the in-memory prescription database and the formats of the prescription data files:

- `zprescription` in-memory prescription database, prescription files

`Prescription` uses `Constants`.

2.3.6 State

These modules provide the variables that represent the state of the therapy equipment and treatment session:

- `zsession` session variables: patient, field, mode
- `zfield` parameter values: prescribed, preset, actual, accumulated
- `zintlk` interlocks, parameter readiness, subsystem readiness

These all use `Constants`. `zsession` and `zfield` also use `Prescription`. `zfield` uses `zsession`, and `zintlk` uses `zsession` and `zfield`.

`zintlk` contains the code for the main event loop in the interlock scanning task so it also depends on `RT`.

2.3.7 Controllers

These modules provide low-level device control. They hide the protocols and data representation used to communicate with the controllers:

- `devicenames` device or file names
- `ports` serial port utilities
- `scx` Scanditronix controller utilities
- `tmc, lcc, dmc` Scanditronix controllers
- `plc-ports` programmable logic controller port utilities
- `plc` programmable logic controller

`devicenames` selects whether to use actual device names, or to use file names instead (for configurations where device input/output is simulated by reading/writing files). The selection is determined by constants defined in `switches.h`.

`scx` and `plc_ports` use `ports`. `tmc, lcc` and `dmc` use `scx`. `plc` uses `plc_ports`. `tmc, lcc, dmc` and `plc` use `Constants` and `State`.

The `tmc, lcc, dmc`, and `plc` modules each contain the code for the main event loop in their controller task, so they all depend on `RT`. However the tasks are all spawned by `Main` (section 2.3.14).

In addition to handling the Dose Monitor Controller (DMC), the `dmc` module also mediates all activities initiated in response to messages or other events from the DMC. Therefore it manages the state transitions in the treatment sequence. This is some of the most important control logic in the program. We considered separating this out but the treatment sequence is so tightly coupled to the DMC that we left both in the same module. `dmc` is the second largest module in the program (almost 1700 lines), only a bit smaller than `zconsole` (section 2.3.12).

Data flows in both directions between `Controllers` to `State`. `Controllers` depends on `State`, not vice-versa. This enables us to write and test `State` before `Controllers` are ready (see p. 57 in Lamb [13]).

2.3.8 X window system (X)

This module provides the X window system:

- `xdefs` X window system

The `xdefs` module contains the header file `xdefs.h` only. `xdefs.h` is one of the few `.h` files that includes other `.h` files. It includes all the header files we need for the X window system library `Xlib` (`Xlib.h`, `keysym.h`, and a few others, but not the libraries for Motif or any other toolkits).

2.3.9 Display

These modules provide low-level graphics. They hide the low-level graphics system (X or whatever):

- `display` low-level but library-independent graphics
- `widgets` dialog boxes, menus, cursors

`display` depends on `X`. `widgets` depends on `display`, but not `X`.

It is important that `display` is the only graphics module that depends on `X`. The `display` module hides the X window system; the constants, types, and function declarations it provides in `display.h` are independent of X. To replace the X window system with some other graphics system, we would only need to rewrite `display.c`, which contains fewer than 700 lines. It is also possible to run the control program without `X` (or any other graphics) by providing a stub `display`.

2.3.10 Screens

These modules generate the many displays (screens) that the operator sees. They hide the appearance of the screens:

- `chartdisps dosecaldisp dosimdisp helpdisp listdisps logbox logindisp mlcdisp pagedisps plcdisp startupdisp statusbox treatmsg` display screens

No module in this group depends on any of the others. They all use `Display` (but none uses `X`). Most also depend on `Constants` and `State` to provide the data that they display.

The modules in `Screens` only read values from the modules they use, they never set them.

2.3.11 Graphics

This module provides graphic output. It hides the organization of the graphics facilities (the particular set of screens that are provided, etc.):

- `graphics` interface to graphics

This module depends on `Display` (but not `X`) and `Screens`.

The only way the control program can generate graphic output is to call a function in `Graphics`. The control program never calls functions in `Display` or `Screens` directly. It is possible to run the control program without `Display` or `Screens` by providing a stub `graphics` module.

Only three modules use `graphics`: `zconsole` in `UI` (section 2.3.12), `user` in `Main` (section 2.3.14), and `messages` in `Messages`.

2.3.12 User interface (UI)

These modules provide the operations of the user interface:

- `zedit` data editing
- `zconsole` user interface operations
- `transition` user interface state transitions

`zconsole` provide the user interface operations. It is the largest module in the program (just over 2000 lines). All activities initiated by the operator, including most graphic output and most file operations, are mediated by code in `zconsole`.

`zconsole` depends on `Constants`, `Prescription`, `State`, and `Controllers`. It also depends on `Graphics` (but not on `Screens`, `Display` or `X`).

`transition` depends on `zconsole` because the state transition table entries are pointers to functions provided there.

In addition to `transition.c` and `transition.h`, the `transition module` also contains `t.h` and `t_names.h`, which contain the state transition table itself.

We separate the operations of the user interface from the graphic output. This makes it easy to build a version of the program that generates no graphic output, but only writes text messages to the standard output.

We separate the operations of the user interface from the events that trigger them. This makes it easy to build a version of the program that does not require any interactive input, but only reads text commands from standard input.

2.3.13 Events

These modules provide user interface input event handling:

- `xevents` X window system event handling
- `keyboard` translation from X keys to commands

- `linereader` translation from strings to commands

`xevents` provides X window system events but is separated out from `display` so modules that do not produce graphic output do not need to depend on it. `xevents` depends on `X`.

`keyboard` processes X keys so it depends on `X` but the modules that use it do not.

`linereader` is an alternative to `keyboard` that processes strings from standard input.

Data flows from `Events` to `UI` but there are no dependencies between `Events` and `UI`. Instead, user interface events are mediated by `Main` (see p. 57 in Lamb [13]).

2.3.14 Main

The main program module is named `user` (because the real-time operating system does not allow a module named `main`, or so we thought):

- `user` main program module

The `user` module only contains `user.c`, there is no `user.h`.

`Main` starts all the tasks. Therefore `Main` depends on `RT` and `Controllers`.

`Main` contains the code for the main event loop in the user interface task. It intercepts events and translates them to commands which it passes to the user interface. Therefore `Main` depends on both `Events` and `UI`, so `UI` and `Events` need not depend on each other (see p. 57 in Lamb [13]).

`Main` intercepts events that cause the display to update. Therefore `Main` depends on `Graphics`.

No other modules depend on `Main`.

2.4 Experience and evaluation

We attempt to avoid creating dependencies. In practice this is difficult and the program has tended to become more interdependent as it has evolved. The modular structure is still satisfactory, but if we undertake an overhaul there are opportunities to remove some dependencies.

The most successful aspects of the modular design, which we have preserved, are the absence of dependencies on `Graphics` in almost all of the program, the absence of dependencies on `X` in `Graphics`, and the absence of dependencies between `Events` and `UI`.

Dependencies on modules in the `RT` group have become more pervasive than we anticipated. It should be possible to remove some of these, in the same way that the X window system is now isolated.

Our decision not to include `.h` files in other `.h files` can be a bit of a nuisance. Sometimes a `.c` file must include an `.h` file only because some other `.h file` needs it, but the `.c` file itself does not. This introduces the appearance of a module dependency where it seems there shouldn't be one. For example, `messages.h` uses the `circbuf_rec` type defined in `circbuf.h`, and `circbuf.h` uses the `SEM_ID` type defined in `rtdefs.h`. Therefore any module that uses `messages` — and many do — must include `circbuf.h` and `rtdefs.h` also. Perhaps we could mitigate this by relaxing our restriction on nested includes in a few more files, as we have already done for `util.h`, `xdefs.h`, and `rtdefs.h`.

Chapter 3

Resources

This chapter describes how the program uses resources such as memory, time, and file storage. We chose these strategies in order to make the program fast, simple, and robust.

3.1 Memory

The entire program and all data is memory resident. When the control computer boots (starts up or resets), it loads all of the control program and the entire contents of all data files into memory. After that, the program can continue to run if the file server becomes unavailable (shut down, disconnected, or inaccessible due to network problems).

Variables that represent the state of the therapy machine and treatment session (chapter 4), the prescription database (chapter 5), and the contents of the display (chapter 7) are all statically allocated (defined at file level, outside all functions). The number and size of these variables is fixed and is determined when the program is written. They persist in memory the entire time the program is running.

Variables that are used temporarily (such as loop indices) are allocated on the stack (defined inside the functions that use them). They occupy memory only when the function that uses them is executing.

The control program code that we wrote does not do any dynamic memory allocation. It does not use the heap. It does not call `malloc`, `calloc`, or `free`.

Operating system code and X window system code that is included in the control program may do dynamic memory allocation, but we have tried to ensure that most of this activity is performed at startup (section 3.2).

We try to avoid using C pointers, but sometimes there is no alternative. Some library functions that we must use have pointer parameters. Sometimes we must use pointer parameters in our own functions to update variables (achieving the effect of passing variables by reference).

3.2 Time

We try to allocate resources at startup to avoid taking time or risking failure while the program is running.

At startup the program allocates most of the operating system resources it needs, including tasks, semaphores, pipes, and watchdog timers (chapter 8). Each resource then persists the entire time the program is running.

Opening and closing a file causes the operating system to allocate a file descriptor, then free it. It is not possible to perform all file operations at startup but we limit the activity as much as we can (section 3.3).

At startup the program allocates all of the X window system resources it needs. It opens the display, loads fonts, allocates colors, creates graphic contexts, and creates windows.

3.3 File storage

The control program uses a small, fixed set of input data files (section 4.3 in the operations manual [3]). Most have a fixed size, but `prescr.dat` and `accum.dat` grow when patients and fields are added and shrink again when patients are archived (the variables that store their contents in memory can accommodate the largest possible files). When the control program starts up it reads the contents of all the input data files into memory. The operator can cause the program to reread certain files to update memory after file contents have changed.

When the control program uses an input file it opens the file, reads its entire contents into memory, and closes it. Each input file is only open during the brief time when the control program is reading it.

The control program writes a large and ever-growing collection of log files and treatment records (section 4.4 in [3]). It opens a new event log file and treatment record file each time it starts up, and each night at midnight. These files grow each time a log message is written or a treatment is performed. Other log files can be written at the operator's command. It is the staff's responsibility to archive these files at intervals.

The treatment record file and event log file are opened when they are created (at startup or at midnight) and remain open until the next versions are created the following midnight, or until the control program shuts down.

Chapter 4

State variables

The variables that represent the state of the therapy equipment and treatment session are described in chapter 1 in the reference manual. They are modelled in the *Session*, *Field*, and *Interlock* state schemas in the formal specification and implemented in the `zsession`, `zfield` and `zintlk` modules in the code.

4.1 Names

Groups of names are modelled in Z by free types and implemented in C by enumerated types.

For example the names of the therapy parameters (table 1.1 in the reference manual) are modelled in Z by the set named *setting* (section 3.1 in the formal specification) and are implemented in C by the enumerated type named `Z_Setting` defined in `znames.h`.

Declaring an enumerated type in C associates the name of each value in the type with a different small integer (enumerated type values are just named integer constants). Therefore the values in each enumerated type have an order. The various subsets of *setting* defined in the formal specification (such as *motion*) are implemented by ordering the setting names in the enumerated type declaration such that members of the same subset appear together (they are adjacent). Therefore settings that belong to the same subset have successive numerical values and they all belong to a particular numerical range. Set membership tests are implented by range checks, for example the `in_motion` boolean function provided by the `znames` module.

4.2 Values

Groups of state variables are modelled in Z by functions from names to values and implemented in C by arrays indexed by enumerated types. (the values of C enumerated types are small integers so they can be used as array indices).

For example the collection of actual therapy parameter values (section 1.2.3, reference manual) is modelled in Z by the function named *measured* in the *Field* state schema whose domain is a set of setting names (section 6.1, formal specification). The actual value of the gantry rotation parameter is modelled in Z by the value of the function application *measured gantry*. The Z function *measured* is implemented in C by the array named `measured` in the `zfield` module. The Z function application *measured gantry* is implemented by the C array access expression `measured[gantry]`.

The representation of these state variables as C arrays is not hidden; the arrays themselves appear in `zfield.h` (they are not hidden in `zfield.c`). They are both declared and defined in `zfield.h`, using the coding technique described in Appendix A. Every module that needs to use the actual parameter values includes `zfield`. Any module that includes `zfield` could read or assign the value of `measured[gantry]`. Of course, it only makes sense for certain code in the `tmc` (treatment motion controller) module to assign the value, but this is not enforced by any programming language construct. It is ensured only by our understanding of the application, expressed through careful coding and quality assurance.

4.3 Operations

The operation schemas on the *Field* state (section 6, formal specification) are implemented by C functions declared in `zfield.h` and defined in `zfield.c`. Likewise for *Session* and `zsession`, *Intlk* and `zintlk`.

Chapter 5

Prescription database

The in-memory prescription database and the prescription files are handled by the `zprescription` module. It is one of the largest modules in the program (over 900 lines).

The in-memory prescription database is the most complicated data structure in the program. It stores identification information about all the patients under treatment and stores the prescribed settings for all their treatment fields. It also keeps track of the number of fractions and the total dose for each field as they accumulate over each patient's course of treatment. Complexities arise because the database contents change frequently and in a piecemeal fashion, as new patients and fields are added and others are archived.

The operations for viewing, selecting, and loading the in-memory database are described in the reference manual [7] (sections 5.4.2 – 5.4.5). The operations for adding patients and fields to the prescription file, and then archiving them, are described in the therapist's guide [8] (appendices A and B). The contents and formats of the prescription file and accumulating doses file are described in the operations manual [3] (sections 4.3.18 and 4.3.6, respectively). A high-level design appears in the formal specification report [6] (section 4), but most of the design details appear in the unpublished fragment `files.tex`[1].

[1] A printable version of `files.tex` is available as PostScript or DVI from `ftp://ftp.radonc.washington.edu:/pub/cnts-reports/z/`.

5.1 Prescription data files

The accumulating fractions and doses are stored in the file `accum.dat`. All other prescription information is stored in `prescr.dat`. Both kinds of information are stored in a single data structure in the in-memory database, however.

The control program reads all of `prescr.dat` and `accum.dat` when it starts up. At this time it makes the in-memory database consistent with both files.

Patients and fields are not added by the control program, but by a separate treatment planning program called Prism.

To add a patient and fields, the Prism user chooses a patient and one or more fields. Prism appends the patient record and field records to the end of `prescr.dat`. No changes are made to the in-memory database or to `accum.dat` at this time.

It is usual to add more fields for the same patient days or weeks later. Again, the program appends the patient record and field records to the end of `prescr.dat`. As a result, records for the same patient (followed by records for some of that patient's fields) usually occur at several separate locations in `prescr.dat`.

The archive operation is not performed by the control program, but by a separate program called Preview (prescription viewer utility).

The archive operation is performed when the patient has completed the entire course of treatment. At this time the archiver program user selects a patient. The archiver program rewrites both `prescr.dat` and `accum.dat`, removing the archived patient and all of his or her fields wherever they occur, but leaving the order and contents of the other patients and fields unchanged. No changes are made to the in-memory database at this time.

Each time the operator invokes the **Patient List** operation, the control program rereads all of `prescr.dat`. At this time the in-memory database is made consistent with any recent changes in `prescr.dat` that might have been made by Prism or the archiver program.

At the end of each treatment run, the control program rewrites all of `accum.dat`. At this time `accum.dat` is made consistent with recent changes in the in-memory database, including the dose from the run just completed and any changes that were propagated from `prescr.dat`.

The program also rewrites all of `accum.dat` when the operator invokes the **Write File** operation when the field list (for any patient) is displayed. This provides a way to write out the contents of the in-memory database of accumulated doses and fractions on demand for diagnostic purposes.

The control program never reads `accum.dat`, except when it starts up. While it is running, it keeps its own record of accumulating doses and fractions current in the in-memory database. Each time it rereads `prescr.dat` and rebuilds the in-memory database, it must retain the accumulating doses and fractions and keep them associated with the correct fields.

Each time the control program reads `prescr.dat` or `accum.dat`, it checks the format, range, and consistency of each item in the entire file. If it discovers any errors, it does not update the in-memory database at all. It merely issues a message to the operator indicating there is a problem with the file and continues running with the previous in-memory database contents.

The control program reads files from a file server computer which is separate from the control computer. Each time the control program attempts to read or write a file, if the server does not respond within a timeout interval (currently 15 seconds, set in a configuration file), the control program abandons the attempt, issues a message to the operator, and continues running with the previous in-memory database contents.

The database of experiment studies and fields is stored in `exper.dat`, which has essentially the same format as `prescr.dat`, with experimental studies taking the place of patients. The program reads all of `exper.dat` when the operator invokes the **Patient List** operation while the program is in experiment mode. At this time the program copies the contents of `exper.dat` into the in-memory prescription database. At this writing the therapy (patient) prescription database and the experiment database both occupy the same memory (program variables), so only one database can be in memory at a time.

5.2 Prescription data structures

The organization of the in-memory database and the files are hidden in the `zprescription` module. No other module can read or set the variables that comprise the in-memory database. The only way the program can access the in-memory database or the files is by calling functions provided by the `zprescription` module.

The in-memory database is implemented by arrays of records. There is a patient array `Names` with a record for each patient. There are four field arrays, each with a record for each field. The records at the same index in all four field arrays refer to the same field. `Prescribed_Info` stores the field name and other identifying information from `prescr.dat`. `Prescribed_Fields` stores the prescribed parameter values from `prescr.dat`. `Accumulated_Fields` stores the accumulating fractions and doses intialized from `accum.dat` and `Accumulated_Status` stores the dose warning flag initialized from `accum.dat`. See the header to `zprescription.c` for more details and cross references to variables in the formal specification.

There are actually two sets of arrays. The first set is the master copy used in all consequential operations. The second set is a temporary buffer and consists of the arrays `pat_lst`, `flds`, `prescr`, `accum`, and `accum_status`. When the control program reads the files, it loads the new data into the temporary buffer. If the program reads the entire files without any errors and the data passes all the range and consistency checks, the control program copies the temporary buffer contents into the master copy, replacing everything that was there (`pat_lst` is copied into `Names` etc.). If any errors occur or the data does not pass the checks, the master copy is left unchanged.

The order of records in the in-memory database is the same as in `prescr.dat`. In the patient array, patients appear in the order they first appear in in `prescr.dat` (a patient may appear at several locations in `prescr.dat`, but only appears once in the patient array). In the fields arrays, fields appear in the order they appear in `prescr.dat`.

The running program identifies each patient by his (her) index in the patient array `Names`. The currently selected patient is indicated by the value of the global variable `patient` (provided by the `zsession` module) which is the array index of the current patient in `Names`.

The patient number that the operator sees is not the array index, it is merely one of the data items stored in the patient record at that index.

The first patient in `prescr.dat` is stored at index 1. The index 0 signifies no patient (`patient` has this value when the operator has not selected a patient). The first element of the patient array (the element with index 0) stores the patient name string `No patient`.

The running program identifies each field by its index in the field arrays `Prescribed_Fields` etc. The currently selected field is indicated by the value of the global variable `field` (provided by the `zsession` module), which is the array index of the current field in the field arrays.

The first field in `prescr.dat` gets index 1. The index 0 signifies no field (`field` has this value when the operator has not selected a field). The first element of the field array (the element with index 0) stores the field name string `No field`.

The field numbers that the operator sees are not stored in the data at all, they are merely the order of occurence in `prescr.dat`, within that patient. So for each patient, the first field for that patient that occurs in `prescr.dat` is that patient's field number one, and so on. This numbering scheme ensures that the same field always gets the same number because we only add fields to `prescr.dat`, until we remove a patient and all of his or her fields at once when we archive the patient.

Each patient record in `Names` stores an array of field identifiers (array indices) of all the fields for that patient, in the order they occur in `prescr.dat`. The first element in the array (index 0) stores the index of the first field for that patient in the field arrays (this is the field number one for that

patient). And so on.

When the operator selects a patient, the index of that patient's record in `Names` is assigned to the global variable `patient` provided by the `zsession` module. The identifying information from the patient's record is displayed on the screen.

When the operator selects a field, the index of that field's records in all four field arrays is assigned to the global variable `field` provided by the `zsession` module. The identifying information at that index in `Prescribed_Info` is displayed on the screen. The contents at that index in the other three arrays are assigned to the variables that represent the current field's prescribed settings etc. in the `zfield` module.

At the end of each run, the program copies the new values of the accumulated dose, number of fractions, and dose warning flag from variables in `zfield` into the records at the `field` index in the arrays `Accumulated_Status` and `Accumulated_Fields`. Then the program rewrites the entire `accum.dat` file with contents of both arrays.

Each time the control program reads `prescr.dat` successfully, it overwrites the patient array `Names` and two of the field arrays `Prescribed_Info` and `Prescribed_Fields` to make them consistent with the file contents. The array contents may change because patients and fields may have been added or removed (archived). If a patient (with fields) has been archived, some of the remaining patients and fields will appear nearer the beginning of the arrays (they will have smaller array indices) as they take the positions formerly occupied by the archived patients and fields.

After the control program reads `prescr.dat` it is necessary to reassign the variables `patient` and `field` and move the contents of the field arrays `Accumulated_Fields` and `Accumulated_Status` to make them all consistent with the (possibly changed) array indices in the field arrays `Prescribed_Info` and `Prescribed_Fields`. This is performed by code near the end of `Read_Prescr_File` in `zprescription.c`.

In experiment mode the program reads the contents of `exper.dat` into the `Prescribed_Info` array. At this writing the same `Prescribed_Info` array is used to store the patient prescriptions from `prescr.dat` and the experiment field setups from `exper.dat`. Therefore only one of these two data sets can be in memory at a given time. The `Accumulated_Fields` and `Accumulated_Status` arrays are not used in experiment mode.

5.3 Prescription displays and user interaction

During some operations, the control program displays a list of patients or a list of fields (for one patient). These displays are managed by the `listdisps` module. The operator can select a new patient or a new field by selecting an item from one of these lists. This interaction is managed by the `zconsole` module. Both `listdisps` and `zconsole` depend on `zprescription`.

The `zprescription` module provides the global variable `name_list` that holds the list of patients or fields that is displayed. The module also provides functions `Generate_Names` and `Generate_Field_Names` that generate the displayed list of patients and fields, respectively, from the arrays in the in-memory database. `Generate_Names` is simple because every patient in the in-memory patient array `Names` appears in `name_list`, and they appear in the same order, so each patient has the same index in `Names` and in `name_list`. `Generate_Field_Names` is more complex because only the fields for the selected patient appear in `name_list`, moreover fields in `name_list` do not appear in their order of appearance in the in-memory field arrays; `Generate_Field_Names` sorts the fields into different categories. The array `field_map` hidden in `zprescription.c` records the relation between indices in `name_list` and the field arrays. The `zprescription` module provides functions that translate between the index in `name_list` and the index in the field arrays, using `field_map`.

The operation of selecting a patient or field is managed by code in `zconsole`. It provides the global variable `list_item` which is the index of the currently selected item in `name_list`.

The list of patients or fields in `name_list` may be too long to fit on the screen. The `listdisps` module displays the list one page at a time and places the display cursor on the current page. It hides the array `list_entries` which contains one screenful of consecutive entries from `name_list`, and the variable `list_cursor` which is the index of the cursor position in `list_entries`.

5.4 Prescription database summary

There are several representations of the prescription database:

1. The files `prescr.dat` and `accum.dat`.

2. The in-memory patient array `Names` in `zprescription`, indexed by `patient` in `zsession`. The in-memory field arrays `Prescribed_Info`, `Prescribed_Fields`, `Accumulated_Fields`, and `Accumulated_Status` in `zprescription`, all indexed by `field` in `zsession`.

3. The array `name_list` in `zprescription`, indexed by `list_item` in `zconsole`.
4. The array `list_entries` in `listdisps`, indexed by `list_cursor` also in `listdisps`.

At particular times, each of these representations must be made consistent with the one that precedes it in the list.

Chapter 6

User interface

The behavior of the user interface is determined by a table. To add a new operation to the user interface, it is not necessary to add or change any control structure. You just write three functions and add a row to the table (whose entries the names of the new functions). This makes it easy to add operations and build different program versions that provide different subsets of the whole collection of operations.

6.1 Operations, events, and states

Operations are central in the design. An operation is a unit of work that is triggered by an event. In the user interface, all events are keystrokes (we do not use the mouse). Every single keystroke triggers an operation in the user interface (there is an "ignore" operation that handles unassigned or disabled keys). The detailed design expressed in the formal specification [6] decribes about 40 user interface operations. We have also coded about 20 more that are not formally specified.

The user interface is designed and coded as a state machine. Our operations are the state transitions, and events (keystrokes) are the inputs. States are also important. Each operation is only allowed to occur in certain states, so a key might or might not trigger a particular operation, depending on the state. For example, it is only possible to select a new field when a run is not in progress. Moreover, a key can trigger different operations in different states (the **Select** key triggers many different operations). There is no single program variable that represents the state. The state is determined by the values of several (many) different program variables.

Three C functions define each state transition. There is a boolean function that tests whether the

program is in a state where that operation is allowed. There is a boolean function that tests whether the event (the most recently pressed key) is the one that triggers that operation. Finally, there is a procedure that performs that operation (a procedure is a C function of type `void`: it returns no value, but updates global variables). All of these functions are provided by the `zconsole` module, which invokes all operations driven by the user's activities. It the largest module in the system (over 2000 lines). Almost all of this module is code for the operation procedures.

6.2 State transition table

All the control logic for the user interface is represented in the state transition table in the file `t.h`. Fig. 6.1 shows the first 30 rows (at this writing there are 73 rows in all).

The table is an array of records (the array is named `t` and is defined in `transition.c`, which includes `t.h` in the definition). There is a row in the table for each state transition. Each row is a record with four members. From left to right, the members are: a number that indicates the indentation level (explained below in section 6.3), the name of the state function, the name of the event function, and the name of the operation procedure. The exact order of the rows in the table is significant because the program traverses the table from top to bottom and the indentation level in each row depends on the preceding rows.

Usually we do not create the file `t.h` by hand. Instead, we create a more nicely formatted file `console-stt.txt` (Fig. 6.2), which better shows the indentation levels indicated by the numbers. We run the script `maketrans` to create `t.h` from `console-stt.txt`. This script also creates `t_names.h`, which is also included by `transition.c` and contains information useful during development, `console-stt.tex`, which is suitable for putting in a LaTeX document, and declarations and code skeletons suitable for putting in `zconsole`.

6.3 State transitions

The X event loop in `user` intercepts each keystroke. It calls `Translate_Key` in `keyboard` to translate the X keysym to our application-specific command identifier, a value of the `Z_Input` enumeration defined in `znames.h`. Then it calls `Transition` in `transition` to traverse the state transition table in `t.h` and select and execute the operation procedure.

Each time `Transition` procedure is called, it traverses the state transition table once, starting at the first (top) row. At each row, the procedure executes the state function. If it returns *true*, the procedure executes the event function. If that also returns *true*, the procedure executes the operation

```
{ 0, Z_True, RefreshKey, Refresh },
{ 0, Z_True, AsynchKey, AsynchOp },
{ 0, Unlocked, NoKey, NoOp },
{ 1, AckWarnMsg, CancelKey, ClearMsg },
{ 1, Started, CancelRunKey, CancelRunC },
{ 1, PausedStopped, CancelRunKey, CancelRunC },
{ 1, TermWait, SelectKey, TerminateC },
{ 1, TermWarn, CancelRunKey, TerminateC },
{ 1, Available, DisplayKey, SelectDisplay },
{ 1, NULL, PatientKey, SelectPatientList },
{ 1, NULL, TableKey, SelectTable },
{ 1, NULL, MessageKey, TypeMessage },
{ 1, NULL, HelpKey, SelectHelp },
{ 1, NULL, CollCalKey, SelectCollCal },
{ 1, NULL, PageKey, SelectPage },
{ 2, Page, VArrowKey, PageUpDn },
{ 2, NULL, FileKey, WritePageFile },
{ 2, List, VArrowKey, GetListArrow },
{ 2, Table, ArrowKey, GetSettingArrow },
{ 2, PatientSelected, FieldKey, SelectFieldList },
{ 2, Setup, LoginKey, SelectLogout },
{ 3, Physicist, ExptModeKey, ExptModeC },
{ 3, PatientList, SelectKey, SelectPatientC },
{ 3, FieldList, SelectKey, SelectFieldC },
{ 3, NULL, FileKey, WritePageFile },
{ 3, PatientSelected, StoreFieldKey, EditField },
{ 3, FieldSelected, NoKey, NoOp },
{ 4, AutoSetupDisplay, AutoSetupKey, AutoSetupC },
{ 4, OverrideTable, NoKey, NoOp },
{ 5, NoItem, OverrideKey, OverrideAck },
```

Figure 6.1: State transition table (excerpt)

```
0, Z_True, RefreshKey, Refresh
0, Z_True, AsynchKey, AsynchOp
0, Unlocked, NoKey, NoOp
1, AckWarnMsg, CancelKey, ClearMsg
1, Started, CancelRunKey, CancelRunC
1, PausedStopped, CancelRunKey, CancelRunC
1, TermWait, SelectKey, TerminateC
1, TermWarn, CancelRunKey, TerminateC
1, Available, DisplayKey, SelectDisplay
1, ,  PatientKey, SelectPatientList
1, ,  TableKey, SelectTable
1, ,  MessageKey, TypeMessage
1, ,  HelpKey, SelectHelp
1, ,  CollCalKey, SelectCollCal
1, ,  PageKey, SelectPage
2,   Page, VArrowKey, PageUpDn
2, ,  FileKey, WritePageFile
2,   List, VArrowKey, GetListArrow
2,   Table, ArrowKey, GetSettingArrow
2,   PatientSelected, FieldKey, SelectFieldList
2,   Setup, LoginKey, SelectLogout
3,     Physicist, ExptModeKey, ExptModeC
3,     PatientList, SelectKey, SelectPatientC
3,     FieldList, SelectKey, SelectFieldC
3, ,    FileKey, WritePageFile
3,     PatientSelected, StoreFieldKey, EditField
3,     FieldSelected, NoKey, NoOp
4,       AutoSetupDisplay, AutoSetupKey, AutoSetupC
4,       OverrideTable, NoKey, NoOp
5,         NoItem, OverrideKey, OverrideAck
```

Figure 6.2: Formatted source for state transition table (excerpt)

function and stops traversing the table. The operation function performs the work that the user requested by pressing the key. If the state function or the event function returns *false*, the procedure proceeds to the next row. If the procedure traverses the entire table without finding an operation procedure to execute, it sounds the terminal bell to notify the user that he (she) pressed a disabled or unassigned key.

The account in the preceding paragraph is somewhat simplified. Testing the state is actually more complicated. At each row, the procedure often uses the results of the state functions from several preceding rows also. It uses the conjunction of the results from the state functions of preceding rows where the indentation level is less (using the result from the last row at each level). If the table entry for the state function in a row is `NULL` (blank in Fig. 6.2), the procedure uses the result from the last preceding non-null state function instead. This method enables us to write simple state functions because more complex conditions are encoded in the ordering and indentation of the table.

For example, the procedure executes the operation procedure `AutoSetupC` (third line from the bottom in the figures) when the event function `AutoSetupKey` returns *true* and when the state functions `Unlocked`, `Available`, `Setup`, `FieldSelected`, and `AutoSetupDisplay` all return *true*.

The state transition table and the workings of the `Transition` function are explained more fully in a paper [10]. The paper decribes how the ordering and indentation of the rows in the table are derived from the formal specification [6].

Chapter 7

Graphics

This chapter describes the code that generates the screen displays (chapter 8 and appendix A in the reference manual [7]). This code comprises about a third of the program, about five thousand lines in almost twenty modules. However the bulk arises from the large number of different displays (twenty) and displayed data items (about four hundred). The design is simple but repetitious. Once it is understood, it is easy to add or change displays.

7.1 Modular structure

The control program uses the X window system to produce graphic output (`Xlib` [14] only, not Motif or any other toolkits). We isolate the X dependencies in a single module so we can easily switch to a different graphics system (or none). Moreover, we separate graphics from the control logic and operations of the user interface (section 2.3.12, chapter 6). This makes it easy to produce program versions that generate no graphics, but only write text messages to standard output (chapter 9).

Graphics are provided by the module groups `X`, `Display`, `Screens`, and `Graphics` (section 2.3). The `xdefs` module in `X` provides all the header files we need for `Xlib`. `Display` is the interface to the X window system. To replace the X window system with some other graphics system, we would only need to rewrite `display.c`, which contains fewer than 700 lines. It is also possible to run the control program without `X` (or any other graphics) by providing a stub `display`. The many modules in the `Screens` group generate the displays (screens) that the operator sees. They all depend on `Display` but none depend on `X`. `Graphics` provides the interface that is used by the rest of the program. It depends on `Display` and `Screens` but not `X`. The rest of

the program only depends on `Graphics`, not `Display`, `Screens` or `X`. It is possible to run the control program without `Display` or `Screens` by providing a stub `graphics` module.

Only a few modules invoke `Graphics`: `zconsole` in `UI`, `user` in `Main`, and `messages` in `Messages`. Almost all the calls to `Graphics` occur in `zconsole`.

Most of the functions provided by `Graphics` indicate events that change the value of a particular variable or group of variables. For example there are functions named `New_Patient`, `New_Field`, `Update_Settings` etc. When an event occurs, the control program calls the appropriate function. The function may (or may not) cause the display to indicate the new values of the variables (depending on which display is currently on the screen). `Graphics` keeps track of which display is on the screen, and determines how each display should be changed after each event. Therefore the rest of the program only needs to indicate events. Information about the appearance of the displays, including which displays are available, is hidden in the graphics modules.

7.2 Resources

At startup the program allocates all of the X window system resources it needs. It opens the display, loads fonts, allocates colors, creates graphic contexts, and creates windows. No additional X resources are allocated while the program is running.

7.3 Display modules

Most of the graphics code is in the many modules in the `Screens` group that generate the displays that the operator sees. The `statusbox` and `logbox` modules generate the rectangular regions that are always visible at the top and bottom of the screen. The `treatmsg` module generates pop-up message boxes. The other modules generate the displays that appear in the central rectangular region. Some of these, for example `logindisp` and `dosecaldisp`, generate a single display. Others, for example `chartdisps`, `mlcdisp`, and `pagedisps`, generate several similar displays.

All of the modules in the `Screens` group use the same design: each contains similar collections of variables and functions.

7.3.1 Data

All variables that represent display contents are defined at file level in the module's `.c` file. All are declared `static`.

Each module represents one window (in the X window system sense).

All window contents are represented by instances of a C structure type called `Cell_Record` declared in `display.h`. A `Cell_Record` represents a colored rectangular region that may contain text. The members of the record store the text, font, text color, background color etc.

Each module defines all the instances of `Cell_Record` needed to represent its window contents. Usually there are one or more arrays of records and several individually named record variables.

7.3.2 Functions

Each display module provides several functions to generate display contents and update the display. The `graphics` module is the only module that calls these functions. The `graphics` module provides the event-based functions used by the rest of the program and translates from these to the display-based functions provided by the display modules.

Each display module provides a similar collection of functions. Module `A` provides an initalization function `Init_A`, a build function `Build_A`, several update functions `Update_B`, `Update_C` etc., a refresh function `Refresh_A`, and (usually) an unmap function `Unmap_A`. For example the `StatusBox` module provides `Init_Status_Box`, `Build_Status_Box`, `Update_Patient`, `Update_Field`, `Update_Lamps`, and `Refresh_Status_Box`. There are a few exceptions: `StatusBox` and `LogBox` have no `Unmap` function because they are always visible. `StartupDisp` has no `Update` functions because its contents are fixed (computed by `Init_Startup`).

Here is what each function does:

`Init_A`: Initialization, called once at startup. Creates window for A, computes fixed contents such as background and captions. Does not display window A.

`Build_A`: Displays window A (maps A, in X jargon). Called when window is first displayed, or is displayed again, replacing a previous (different) window in the same location.

`Update_B`: Recomputes contents of B portion of window A from application data, then draws contents. Called after new data becomes available and window is already displayed (mapped).

`Refresh_A`: Redraws contents of all regions of window A without recomputing contents. Called after exposure events, etc. (when overlaying pop-up box disappears, etc.)

`Unmap_A`: Causes window A to disappear (unmaps A), possibly other cleanup as well. Called right before another window replaces window A.

Chapter 8

Task structure

The therapy control program comprises several concurrent tasks. This chapter describes the tasks and explains how they communicate and synchronize.

The control program runs on a commercially available real-time operating system [19, 20] and uses the tasking facilities that it provides. However, it uses facilities that are also provided by general-purpose (Unix-like) operating systems wherever this is possible. This enables us to build program versions that run in a single process on ordinary workstations for development and testing.

8.1 Design rationale

Operations, not tasks, are central in the design. An operation is a unit of work that the program can complete quickly. The detailed design expressed in the formal specification [6] decribes more than one hundred operations. An operation becomes enabled when certain conditions are true (one kind of condition indicates that an event has occurred recently). An enabled operation can execute. Concurrency arises because more than one operation can be enabled at the same time. The design allows enabled operations to execute in any (nondeterministic) order. In the formal specification there are no tasks, just a big collection of operations that execute on demand. Allocation of operations to tasks is a feature of each implementation, not the design. In program versions that run on a general-purpose operating system, all operations execute in the same task. In versions that run on the real-time operating system, operations execute in several tasks.

We designed the tasks to eliminate needless waiting. When there is just one task, all work has to stop to wait for any event. Allocating operations to independently scheduled tasks allows some

tasks to work while others are waiting. Unnecessary waiting is minimized when there is a separate task to monitor and control each independent (concurrent) device or activity in the outside world. Therefore we have one task for each of the four controllers: TMC, LCC, DMC and PLC. Another task handles the user interface and file operations (this task monitors the keyboard and controls the display and disk). Finally, there is an interlock task that uses information provided by all the other tasks to compute software interlocks and system status, and to dispatch other tasks when certain conditions occur.

The program is event-driven. Each task can respond to a particular set of events. The main routine in each task is an event loop where the task waits for one of these events, processes it, and then waits again for the next event. The task determines whether each event has caused one of its operations to become enabled. If so, the task executes the operation.

Most sequencing and control is internal to each task and does not require communication among tasks. Some sequencing and control applies to the entire program, for example the treatment phase (which indicates when a therapy run is in progress, among other things). This global control is exercised by only two tasks, which communicate with the other tasks in order to send commands. The user task exercises the control driven by the user's activities, while the interlock task exercises the control driven by sensors (via the device controllers) and by conditions the arise in the internal program state.

8.2 Tasks

The control program comprises six tasks (Table 8.1). The columns in the table show the task name used by the operating system, the name of the C function that comprises the task's main routine, the name of the module which provides that function, the task priority (higher numbers mean lower priority), and a brief description. Modules are organized around access to data, not tasks; every task also uses functions (etc.) provided by other modules besides the one listed in the table, and every module in the table provides functions to other tasks as well. See chapter 2.

In addition to the six control program tasks, there are also four operating system tasks (see Appendix H in [3]). They can be ignored here.

8.2.1 Creation and deletion

All six tasks are created when the control program starts up and are deleted when the control program exits. No tasks are created or deleted while the control program is running. Each task has a fixed priority that never changes. Higher numbers mean lower priority, so the user interface task has

Name	Function	Module	Priority	Description
tUser	user	user	99	User interface, file operations
tTMC	TMC_Task	tmc	92	Treatment motion controller
tLCC	LCC_Task	lcc	92	Leaf collimator controller
tDMC	DMC_Task	dmc	90	Dose monitor controller
tPLC	PLC_Task	plc	90	Treatment motion controller
tIntlk	Intlk_Task	zintlk	90	Software interlocks, system status

Table 8.1: Therapy control program tasks

the lowest priority.

8.2.2 Scheduling

The control program uses *pre-emptive priority scheduling*: the highest priority ready task runs (section 2.3 in [19]). A task is *ready* when it is not *blocked* (waiting for an event, such as input from a controller). A task runs until it blocks (pauses itself to wait for an event), or until a higher priority task becomes ready (because an event has occured, for example input has arrived).

8.3 Communication and synchronization

8.3.1 Shared data

All tasks share a single address space — all of the data is potentially accessible to all of the tasks (subject to the data hiding described in chapter 2). The state of the equipment and treatment session is stored in global variables provided by the `zsession`, `zfield` and `zintlk` modules (chapter 4). Each task updates particular variables and all the other tasks may read them. Usually there is no need for any explicit sychronization because only one task can update each variable and it is permissible for updating and reading to interleave in any order.

8.3.2 Critical sections

In a few cases we need to ensure that a *critical section* of code will be executed to completion, without being preempted by other tasks. We achieve this by assigning task priorities. We assign equal high priorities to the DMC, PLC and interlock tasks. No other application tasks have higher

priorities than these. Therefore these tasks cannot be preempted (section 2.3 in [19]) and every non-blocking sequence of code in these tasks is a critical section.

8.3.3 Semaphores

Tasks use *semaphores* to signal the occurence of events (section 2.3.4 in [19], especially pages 60 - 61). For example, the user task gives a semaphore to a controller task to signal that program is ready to perform an automatic setup operation. The controller task then performs its part of the setup operation, reading the preset parameter values from the global variables in `zfield`. Tasks can wait for semaphores with an optional *timeout*; we use this to provide periodic polling.

The real-time operating system does not provide any built-in mechanism to wait for several different events and respond to the first that occurs[1]. Therefore we have only one semaphore per task, which is filled when any pertinent event occurs.

We use *request flags* to distinguish between events. Request flags are not an operating system facility, they are ordinary boolean variables that we use for this purpose. There is a separate request flag for each kind of event. When a sender task notifies a receiver task of an event, it sets the request flag and then gives the semaphore. The receiver task takes the semaphore, and then checks the request flags. When it finds a flag that is set, it clears that flag and dispatches to the code that handles the request.

It might seem that there are potential race conditions because access to the request flags is not synchronized. We avoid them by observing the following design rules. Each request flag can only be set by one sender task. The sender task can only set the flag. The receiver task is the only task that can clear the flag. The sender task always sets the flag first, then gives the semaphore. The receiver task always takes the semaphore first, then tests the flags. If it finds a set flag, it clears it and handles that request. The receiver task always checks the request flags in a fixed priority order. If more than one request flag is set (multiple requests are pending), the receiver task handles each request in priority order (in a few cases, the task also clears the low priority request flags when it handles the high priority request). When the receiver task is the DMC, PLC, or interlock task, all of these activities are uninterruptible because every non-blocking sequence of code in these tasks is a critical section (section 8.3.2).

The module that provides each task's main routine (Table 8.1) also provides a function for each kind of request. The only way another task can communicate an event is to call that function. Note that the function is provided by the module that provides the receiver task, but the function executes in the sender task. The body of the function sets the request flag and gives the semaphore. The request flags and semaphore are hidden in the receiver task module and cannot be directly manipulated

[1]The `select` mechanism (section 8.3.5) only applies to data streams, not events.

or even read by other modules. In this way intertask communication is made to appear like an ordinary function call in the sender module. In effect, the receiver task module hides the fact that the receiver is running in a separate task. Therefore the sender module need not depend on any real-time operating system facilities and can run on an ordinary workstation.

We also use one semaphore for mutual exclusion (section 2.3.4 in [19], especially page 60). The `mutex` semaphore in the `circbuf` module ensures that only one task at a time can update the circular buffer of log messages.

8.3.4 Pipes

Tasks use *pipes* to communicate sequences of data items called *messages* (section 2.4.5 in [19]). Pipes provide *buffering*: temporary storage for (several or many) messages in transit between tasks. Therefore they provide *asynchronous* communication. A sender task can write messages to the pipe, then proceed with other work. The receiver task need not respond immediately; it can complete other work, then read the messages from the pipe later. The control program sets the capacity of each pipe when it starts up.

In our program no task ever waits reading or writing a pipe. A sender task always checks that the pipe is not full before it attempts to write (otherwise, it would have to wait for the receiver task to free up space by reading from the pipe). If the pipe is full, the sender discards the message and proceeds (it also indicates that messages were lost). If the pipe is empty, the reader task proceeds without waiting for a message.

These pipes are like the named pipes provided by most general-purpose operating systems, so code that uses them can also run on a workstation.

8.3.5 Select

Tasks use the operating system `select` function to do non-blocking reads from devices, pipes, and sockets (section 3.3.8 in [19]). Tasks call `select` to check whether data is available any of several sources, and dispatch to one of them. Tasks can optionally wait for input from any of the sources, with an optional timeout. We also use `select` with a timeout to implement delays (see `delay.c`). The `select` function is provided by most general-purpose operating systems, so code that uses it can also run on a workstation.

8.3.6 Signals

We use *signals* and the C library `setjmp` facility to handle a few abnormal situations or *exceptions* (section 2.4.7 in [19], Appendices B8 and B9 in [12]). A task or interrupt handler can send a signal to another task. The task that receives the signal abandons its current activity and executes the specified signal handler (a function) instead. The task responds promptly to the signal even when it is blocked. Signals and `setjmp` are provided by the ANSI Standard C library, so code that uses these facilities can also run on a workstation.

8.3.7 Watchdog timers

Tasks use operating system *watchdog timers* for some timeouts (section 2.6 in [19]). A task starts a timer and then continues with other activities. When the timer times out, the program executes a specified *interrupt service routine* (section 2.5 in [19]) that can give a semaphore, send a signal, or update data (such as a flag to indicate that the timeout has occurred).

8.4 Task list

This section describes each of the six application tasks.

8.4.1 User task

The user task is the main program that starts and coordinates most other activities. The startup script `isostart` runs each time the control computer boots. A commmand in this script spawns the user task. The user task then spawns the other five tasks and enters its event loop. When the operator issues the shutdown command, the user task deletes the other five tasks and exits.

Like most tasks, the user task handles a device in the outside world. The device that the user task handles is the operator's workstation: it monitors the keyboard, updates the display, and reads and writes files on disk. No other tasks access the display or file system. (In the present configuration, these are actually two devices: the keyboard and display are provided by an X terminal and the file system is provided by a separate file server. See the installation guide [3]).

The user task runs at the lowest priority; it can be preempted by any other task.

```
do /* while not shutdown */
   select(xfd, mfd, ifd)

   xfd: /* X window system event: keystroke */
      decode keystroke and perform requested action
      if display should change, update it
      if work is ready for interlock task (new status etc.),
         give semaphore
      if work is ready for controller task (auto setup etc.),
         give semaphore

   mfd: /* Log messages in pipe */
      write messages to log file
      if log message display selected, update it

   ifd: /* Display update request in pipe */
      if pertinent display selected, update it
      update clock on display
      if periodic housekeeping needed, do it

while not shutdown
```

Figure 8.1: User task event loop

The user task is not a real-time task. Display and file operations may take an indefinite (upredictable) amount of time. All input to the user task is queued or buffered so immediate response is not necessary. We considered a system design where the user task was an application program running on an ordinary workstation under a general-purpose operating system, communicating over the network with the other tasks running on a real-time computer[2]. The present design still bears many traces of this (although the user task now runs on the real-time computer as well).

The user task is an event loop that handles three kinds of events: keystrokes, log messages, and display update requests (fig. 8.1). Keystrokes come from a socket to the X server, log messages come from a *message pipe* that is filled by all tasks, and display update requests come from an *interlock pipe* that is filled by the interlock task (but indicates activity by all four controller tasks). These three sources are identified by the C file descriptors `xfd`, `mfd`, `ifd`, respectively. The user task dispatches on events from all three sources using `select`. When input arrives at any source, the task branches to the code that handles that source. Therefore the task can update the display as often as needed and still respond promptly when the user strikes a key.

[2]This system design is used in EPICS [1], which we are planning to use for the cyclotron control system.

Keystrokes from `xfd` may indicate that the operator has requested activity from other tasks, for example an `Auto Setup` operation. In those cases the user task issues commands to other tasks by giving semaphores (the user task never takes a semaphore, it only gives them).

All tasks write event log messages into a single pipe. The `mfd` branch reads from this pipe to write the log message file (and update the display if needed). The pipe mechanism ensures that messages are not lost and do not overwrite each other (as might occur if all tasks attempted to write the log message file).

Each message in the interlock pipe indicates that a controller has aquired new data (the content of the message says which controller). The `ifd` branch reads each message and updates the display only when the current display contents could be affected by that controller.

Controller activity ensures that the `ifd` branch is taken frequently, so this branch also performs periodic housekeeping. It checks to see if certain conditions are true, and if they are, it performs the appropriate actions. It calls `Check_ClockEvent` to check the calendar so it can reset the daily doses and open new log files when the date rolls over. It calls `Check_RunEvent` to determine when to write treatment log messages. It calls `New_Time` to update the on-screen clock.

The user task reads and writes files on a separate file server computer, using the NFS protocol. If the server is unreachable or not working, NFS may wait several minutes before timing out. During this long interval the control program would appear "frozen" or "hung" — it would not respond to keystrokes and the display would not update. That would be unacceptable. We used a watchdog timer and the `setjmp` facility to provide a much shorter timeout[3], see `Read_File` and `Write_File` in `zconsole.c`.

8.4.2 Controller Tasks

All four controller tasks (PLC, TMC, LCC, DMC) have the structure shown in Fig. 8.2.

`Sync_Error` checks for unsolicited messages from the controller. These usually indicate that the controller has lost synchronization with the control program and must be reset.

The user task can set a request flag and give (fill) the controller semaphore `ctlrSem` (actually `tmcSem` or `lccSem` etc.) to request the controller task to perform some action (an **Auto Setup** operation, for example).

If the user task has no request for the controller, `ctlrSem` is empty. The controller task waits at the `semTake` call for a time interval up to `period`. Then `semTake` times out and returns a value

[3]Currently 15 seconds, set in the `timeouts.dat` file, section 4.3.20 in [3]

```
while (true) {
   status = success;
   status = Sync_Error();

   if (status == success)
   {
       if (OK == semTake(ctlrSem, period)) {
           status = handle_event();
           PutSettings(ctlr);
       }
       else {
           if (polling && iclear == interlock[ctlr]) {
               status = do_poll();
               PutSettings(ctlr);
           }
       }
    }
    if (status != success) handle_error();
}
```

Figure 8.2: Controller task event loop

different from OK. If this controller is in polling mode and its error interlock is clear, the controller task calls do_poll to poll the controller once.

Usually there are no requests for the controller, semTake times out repeatedly, and the controller task polls periodically (some controller tasks poll only during certain treatment phases). During each polling cycle the controller task commands the controller to read and report its input values. Then the controller task uses these values to update certain variables or array elements that represent the state of the treatment machinery (chapter 4). All other tasks may read these variables at any time (they do not synchronize with the controller task, they just read the most recent values).

Occasionally there is a request for the controller so ctlrSem is full, semTake returns OK, and the controller task executes handle_event (which checks the request flags and handles the request). Some requests (for example, auto setup) involve a sequence of controller commmands and responses that take much longer than a single polling cycle. The controller task always completes handling the request before resuming polling or handling the next request. Therefore polling is not strictly periodic, and appreciable delays can occur before a request is handled.

After returning from do_poll or handle_event, the controller task calls PutSettings to signal the interlock task that there may be new data from the controller, so system status should be recalculated. PutSettings gives a semaphore to the interlock task and updates a variable that indicates which controller is the source of the new data. PutSettings is provided by the zintlk module but runs in the controller task.

The following sections describe special features of each controller task.

8.4.3 PLC task

The PLC task is the controller task with the simplest structure. It simply polls the PLC periodically. On each polling cycle it commands the PLC to set outputs and read inputs. Unlike the other controller tasks, PLC polling never stops or pauses to handle other requests[4]. There is no separate auto setup operation for the PLC task; it can set outputs on any polling cycle.

At the end of each polling cycle, the PLC task signals the interlock task by calling PutSettings. Since PLC polling continues without pause as long as the control program is running, this acts as a periodic pacemaker for the interlock task, and (through the interlock pipe) for the user task as well.

If the control program detects errors in its communication with the PLC, it shuts itself down. If the PLC detects that the control program has stopped polling, it sets an interlock in the hardwired safety trace. The only way to recover is to (manually) restart the control program and reset the interlock.

[4]It is possible to stop PLC polling to use the pass-through facility but this is never used in normal operation.

8.4.4 TMC task

The TMC task works as described above (section 8.4.2), but there are a few additional complications.

At the auto setup operation, before commanding the TMC to initiate motions, the TMC task must cause the PLC to energize the motion enable relays. However, this task does not command the PLC or communicate to the PLC controller task; instead, it merely sets the enable requests by updating certain array elements in the `zintlk` module (chapter 4). Then the TMC task waits a few seconds and reads different array elements in `zintlk` to confirm that the enable relays are energized. If the relays are energized, the TMC task commands the controller to initiate motions. If not, this task clears the enable requests, abandons the auto setup operation, and displays a message to the operator. In this way the TMC task can coordinate with the PLC task without any explicit synchronization (other than the delay).

After the TMC task commands the controller to initiate motions, the motions can continue for more than a minute. The TMC task polls the controller while motions are in progress. When the task determines that a motion is complete (has reached its commanded ending position) it clears that motion's enable request to cause the PLC to disable the motion. Usually each commanded motion completes at a different time. If a commanded motion does not complete within its deadline, the task clears its enable request.

The TMC task polls the controller at all times, except when it is responding to a request.

8.4.5 LCC task

The LCC task works as described above (section 8.4.2).

This task also sets, tests, and clears motion enable requests for the PLC, much like the TMC task (section 8.4.4). The interval that the LCC task waits for the motion enable relays to become energized was determined by guessing, trial and error. Once when we replaced the PLC by a different model, the timing changed and we had to make the interval a bit longer.

The LCC semaphore `lccSem` is not just set by the user task. It is also set by the interlock task when certain array elements in the `zintlk` module indicate that the operator has pressed buttons on the X-ray control box or in the treatment room to request a leaf collimator setup. The code that monitors these elements and sets the semaphore and request flags is in the function `Scan_CollSensors` which is provided by the `lcc` module, although it executes in the interlock task (section 8.4.7).

The LCC task polls the controller at all times, except when it is responding to a request and when

leaf motion is in progress. The LCC does not respond to commands when the leaves are moving.

8.4.6 DMC task

The DMC task is the most complex controller task. In addition to controlling the DMC (whose protocol is more complex than the other controllers), this task also controls the treatment sequence by assigning values to the `phase` global variable in the `dmc` module.

The DMC semaphore `dmcSem` is not just set by the user task. It is also set by the interlock task when conditions indicate that certain actions should be taken (such as changing the treatment phase or sending a sequence of commands to the DMC). The code that monitors these conditions and sets the semaphore and request flags is in the function `Scan_TreatSensors` which is provided by the `dmc` module, although it executes in the interlock task (section 8.4.7).

The DMC task only polls the controller when a run is in progress, as indicated by certain values of the `phase` variable.

8.4.7 Interlock task

The interlock task monitors the system state, computes software interlocks and system status, and triggers activities in other tasks. This is the only task which is not driven primarily by events originating in the outside world (controller activity or the user's activity). Instead, this task is driven by the other tasks, and uses data updated by those tasks.

Fig 8.3 shows a sketch of the interlock task. The interlock task waits at `semTake` until any other task indicates it has new data by writing its own identify into `intlk_msg` and giving `intlkSem`. The interlock task waits without a timeout because controller task activity (periodic polling etc.) ensures that the semaphore will be given frequently.

Each time another task indicates there is new data, the interlock task recomputes all software interlocks and the status of every therapy parameter and every subsystem.

The interlock task calls `Scan_TreatSensors` in the `dmc` module to monitor conditions that indicate when the treatment phase should change. If necessary, it sets request flags and gives `dmcSem` to signal the DMC task. `Scan_TreatSensors` contains some of the most important control logic in the program. This logic is an implementation of the tables derived from the preconditions of Z operation schemas in the unpublished fragment of the formal specification named `dmc.tex`[5].

[5] A printable version of `dmc.tex` is available as PostScript or DVI from

```
while (true) {
    semTake(intlkSem, WAIT_FOREVER)
     /* return when another task has new data */

    Calculate software interlocks, parameter status,
      subsystem status

    /* if treatment phase changed, give dmcSem */
    Scan_TreatSensors

    /* if leaf setup button pressed, give lccSem */
    Scan_CollSensors

    if interlock pipe not full
      write(ifd, intlk_msg) /* indicates which task has new data */
}
```

Figure 8.3: Interlock task event loop

The interlock task calls `Scan_CollSensors` in the `lcc` module to monitor conditions that indicate the leaf collimator should be set up. If necessary, it sets request flags and gives `lccSem` to signal the LCC task.

Finally, the interlock task writes the controller identity stored in `intlk_msg` into the interlock pipe `ifd` to signal the user task to update the display and perform housekeeping functions. There is a potential race condition because access to `intlk_msg` is not synchronized and several tasks may write it. A different controller task may write `intlk_msg` again before the interlock task reads the first value. In practice this has no serious consequences because the important information (that system status should be recalculated) is conveyed by the semaphore; the value of `intlk_msg` is merely a hint to the display code.

`ftp://ftp.radonc.washington.edu:/pub/cnts-reports/z/`.

Chapter 9

Development stages

We developed the program in stages. At each stage we had several working, testable program versions. This chapter describes the sequence of stages.

Each stage before the last provided a subset of capabilities. We added capabilities primarily by adding new modules, not by changing existing modules.

The different program stages ran in different environments. Most of the code was developed on an ordinary workstation running a general-purpose (Unix-like) operating system. We selected and acquired the embedded computer and real-time operating system rather late in the project. Even then, we had only limited access to the actual therapy machine, and much of the real-time code had to be developed in a test environment where devices were simulated by stub routines.

We devoted considerable effort to test automation. It is possible to simulate operator's activity and device activity using stub routines that read from files. Test cases can be stored in files and run repeatedly with little effort. The test automation facilities are still present in the operational version of the program, although they are not usually enabled.

The following sections describe each program stage.

9.1 Workstation

Three early stages ran on an ordinary workstation running a general-purpose operating system. They used the workstations's disk file system and console (the local X server).

9.1.1 ANSI C only

The initial program stage used only ANSI C and the ANSI standard library (Appendix B in [12]).

Most of the control program code uses only ANSI C, including all the code that reads and writes files. This code could have been developed and tested on almost any system that provides a C compiler. It includes the module groups `Utilities`, `Constants`, `Prescription`, `State`, `UI`, and early versions of some of the modules in `Messages`, `Events`, and `Main` (section 2.3).

Instead of using a graphical user interface, this stage reads commands from standard input (the keyboard) and writes status messages to standard output (the terminal window). Test automation is accomplished by simply redirecting input from a file of commands (a *test script*).

In our modular design the user interface logic and operations are separated from graphical output and event handling (section 2.3). This early stage provided most of the same operations as the graphical user interface to come.

At this writing code and other items from this stage are stored on the HP cluster under `/radonc/cnts/history/dem`
and `.../xless`

9.1.2 ANSI C with X windows

The next stage adds a graphical user interface (GUI), using the X window system. The GUI only uses `Xlib` [14], not Motif or any other toolkits.

This stage adds the module groups `X`, `Display`, `Screens`, and `Graphics`, adds modules to `Events`, and revises `Main` (section 2.3).

The GUI does not replace the command line interface. The command line interface remains in all program stages because it supports test automation. In the `user` module, the code optionally reads and executes commands from a test script until it reaches end-of-file. Only then does it enter the X event loop. (In the operational version of the program, a command line option forces the program to bypass the test script and go directly to the X event loop).

At this writing code and other items from this stage, including test scripts, are stored on the HP cluster under `/radonc/cnts/history/demo.aug97`

9.1.3 Simulated tasks and device control

The next stage uses functions provided by the (Unix-like) workstation operating system to simulate tasking and device control in a single operating system process. Most of the code was developed and tested by the end of this stage, including most of the device controller modules.

At this stage we add the module groups `RT` and `Controllers` (section 2.3), add modules to `Messages` and revise `user` in `Main`.

At this stage we use the `select` function to wait for input from multiple sources and to provide delays (section 8.3.5). We also use *named pipes* to communicate sequences of data items (section 8.3.4).

In this stage the user task (fig. 8.1, section 8.4.1) is the main program (in `main.c`, the precursor of `user.c`). The main event loop at this stage is almost in the final form shown in fig. 8.1. The most important difference is that the `xfd` branch does not give any semaphores (there are none). At this stage there are no controller task event loops as in fig. 8.2 (there are no controller tasks). Instead, the main event loop code simply calls the appropriate controller functions and waits for them to return. For example, when the user requests an auto setup operation, the code calls a function in the affected controller module that sets the pertinent request flag (sections 8.3.3, 8.4.2) and calls that controller's `handle_event` function (section 8.4.2). The code in `handle_event` completes its work quickly and returns control to the main event loop. At this stage there is no interlock task event loop as in fig 8.3 (there is no interlock task). Instead the code in the body of this loop (including writing to the interlock pipe) is provided in a function named `Scan` that is called by the main event loop or by code in the controller modules when the values of any pertinent variables might have changed.

In this way almost all of the code can be exercised in a single process (with only one task). This is made easy by our modular design: the main event loop communicates with the controller modules and interlock module by calling functions whose interfaces (declarations) are the same whether or not there are separate tasks (see paragraph five in section 8.3.3). In effect, the controller modules and interlock module hide the fact that they are running in a separate task (or not).

Except for the absence of the event loops, the code in the Scanditronix controller modules `tmc`, `lcc` and `dmc` is almost identical to the final stage. So is the lower-level code in the `scx` module. The main difference is hidden in the lowest-level device module, `ports`. In this stage `ports` reads and writes files with simulated controller input and output, instead of reading and writing to the serial port. The files with simulated controller input serve as test scripts.

In this stage the `plc` module is just a stub; we do not simulate the activities of the PLC as realistically as the Scanditronix controllers. Instead there are operations in the user interface that allow the

user to simulate setting and clearing bits by pressing keys on the keyboard.

In this stage we simulate periodic polling by the (nonexistent) controller tasks. The `select` call at the top of the main event loop includes a timeout (not shown in fig. 8.1). When the timeout occurs, control transfers to a fourth branch (not shown in fig. 8.1) that writes a message to the interlock pipe. This ensures that the `ifd` branch in the main loop is taken periodically, just as it is when the controller tasks are polling.

At this writing code and other items from this stage are stored on the HP cluster under `/radonc/cnts/demo`

9.2 Embedded computer

Two later development stages, including the operational program, run on a single-board embedded computer running a real-time operating system. The embedded computer has no disk, display, or keyboard. It communicates over a network with an X terminal and a file server; connections to these are established by the startup script that runs when the computer boots. Control program code that uses the X console and file system is unchanged from earlier stages. For more information about the embedded computer, see the operations manual [3].

9.2.1 Tasks, simulated device control

At this stage we implement tasking as described in chapter 8. We run the controller code and interlock scanning code in five independently scheduled tasks and signal events by giving semaphores instead of calling functions. The `RT` and `Main` module groups are complete in this stage.

Most program source files are the same as in the final stage, except we continue to simulate the treatment equipment by using special versions of the `ports`, `plc_ports` and `plc` modules in the `Controllers` group. Other differences are handled by conditional compilation, using the `DEMO` and `DEBUG` constants defined in `switches.h`.

We continue to use this configuration to develop and test program revisions because it can run without access to the actual treatment equipment.

The code for this stage is in `/radonc/cnts/code`, except the variant `ports`, `plc`, and `plc_ports` modules are in `~jon/cnts/task`.

9.2.2 Operational program

This is the final stage, now in use in the clinic. It actually operates the treatment equipment. Its code differs from the preceding version only by replacing the `ports`, `plc_ports` and `plc` modules in the `Controllers` group, and by undefining the `DEBUG` and `DEMO` constants in the `switches` module.

The code for this last stage is in `/radonc/cnts/code`.

Chapter 10

Utility programs

In addition to the control program that runs on the embedded computer, we also had to produce several utility programs that maintain the control program data files. These programs can run on an ordinary workstation under a general-purpose (Unix-like) operating system. Currently they run on any workstation in the HP cluster. The control program data files are accessible on the HP cluster via NFS.

Some of the utility programs use many of the same modules (exactly the same source files) as the embedded control program.

10.1 Prescription utility

Therapists use the prescription utility (named `preview`, for *prescription viewer*) to archive patients and fields, to change field completion status, and to view the prescription database. Instructions for using the prescription utility appear in Appendix B of the therapist's manual [8].

A therapist performs the archive operation when a patient has completed the entire course of treatment. The therapist uses the prescription utility to select the patient. The utility rewrites both `prescr.dat` and `accum.dat` (section 5.1), removing the selected patient and all of his or her fields. Then it writes the patient and field information to a permanent archive (section 4.5 in the operations manual [3]).

A therapist can change the field completion flag (for example from T (Treatment) to S (Superceded)) when appropriate. The therapist uses the prescription utility to select the patient and field. The utility

rewrites `prescr.dat` to change the value of the flag.

The prescription utility is mostly built from the same modules (exactly the same source files) as the control program. In particular, it uses the same `zprescription` module to read the files and maintain the in-memory prescription database, and uses the same user interface and display modules so the appearance of the prescription database on the screen, and the operations used to select patients and fields, are exactly the same in the utility program (at a workstation) as they are in the control program (at the control console).

At this writing the `preview` program source and executable are on the department HP cluster in `/radonc/cnts/preview`

The `preview` program uses only ANSI C and Xlib and could be built and run on any system that provides them. It does not use the real-time operating system nor does it require any particular variant of Unix (such as HP-UX).

Source files common to both programs are not duplicated in the `preview` directory. They can be found in the source directory for the control program which is `/radonc/cnts/code` at this writing.

The `preview` directory contains only the source files which are different from the control program files. The ones named `*-pr.{h|c}` (for example `user-pr.c`) are derived from the corresponding `.h` and `.c` files for the control program but are usually much shorter (because many control program functions are stubbed out).

Several source files are new for the `preview` program, for example `archive.h` and `archive.c`. They have different names from any control program source files.

The `preview` program uses the same user interface module `zconsole` as the control program. Most of the new functions in the `preview` program are in the new `archive` module. The user interface of the `preview` program is determined by providing a new state transition table, similar to the control program table in the file `t.h` (chapter 6). The table for the `preview` program is in `t-pr.h`. This table has entries for functions in the `zconsole` module and the new `archive` module. Most of the operations in `zconsole` are unreachable in `preview` because there are no corresponding table entries in `t-pr.h`.

We created a new makefile `Makefile-pr.h` that uses the workstation C compiler. To build the `preview` program, place the preview-specific source files and `Makefile-pr` in the `preview` directory. Then make symbolic links in that directory to all the control program source files in the `../code` directory. Then execute the makefile. More detailed instructions appear in the `AAAREADME` file in the `preview` directory.

Prescription utility source files that have different contents from control program files all have different names, so it would be possible to keep all the source files for both programs in a single directory. However it is necessary to build the `preview` program in a different directory so its `.o` files (for the workstation) can be distinguished from control program `.o` files (for the embedded computer) with the same names. To avoid confusion we keep the source files in separate directories also, with symbolic links to the source files that are the same in both programs.

10.2 Dose calibration utility

The dose calibration utility `dosecal` provides a graphical user interface for editing the contents of the dosimetry calibration file `dosecal.dat`.

The dose calibration utility is much like the prescription utility: it is mostly built from control program source files, which ensures that it has the same appearance and works the same way as the control program.

The dose calibration utility files are in `/radonc/cnts/dosecal`. This directory contains links to control program files in `../code` and to prescription utility files in `../preview`. Most of the new functions are in the new `dosecal` module, the user interface state transition table is in `t-dc.h`, and the makefile is `Makefile-dc`. Detailed instructions for building the utility are in the `AAAREADME` file in the `dosecal` directory.

10.3 Treatment planning program

New patients and fields are appended to the end of the prescription file `prescr.dat` by the treatment planning program, Prism (section 5.1). Instructions for using Prism appear in Appendix A of the therapist's manual [8].

Prism has no code in common with the control program. Prism is coded in Common Lisp. The code that writes the new prescription file contents is in the file `write-neutron.cl` kept under configuration control in `/radonc/prism/SCCS`.

Code in `write-neutron.cl` writes the new prescription file contents, then spawns the shell script `cnts_xfer` to actually append these contents to `prescr.dat`. At this writing the shell script and other documentation are in `/radonc/cnts/transfer`

10.4 Log file utilities

Several utilities summarize the log files and treatment record files written by the control program. These summaries are useful for machine maintenance and patient quality assurance. For example, the `accum` utility reads through several weeks' treatment records, produces a day-by-day listing of all the treatments for all the patients currently under treatment, and compares the accumulated total doses and fractions calculated from all the treatment records to those recorded in the `accum.dat` file. The `accum` utility and several others run every night, scheduled by the `cron` utility.

These utilities have no code in common with the control program. They are coded in Perl and sh (Bourne shell). At this writing utility code and other documentation are in `/radonc/cnts/qa`

Appendix A

Declaring global variables in header files

There are some global variables that are used by more than one module. Each such global variable must be defined in one module and declared external in all the other modules where it is used. We achieve the effect of either declaring or defining a variable in a single `.h` file by using conditional definitions as in the following example [1].

`zsession.c` begins with the list of include directives for all the other modules it depends on. Then it defines a macro named `ZSESSION_EXTERN` whose replacement text is the empty string, and then it includes its own `.h` file:

```
/* zsession.c */

#include "util.h"
#include "znames.h"
#include "zoper.h"
#include "zprescription.h"

#define ZSESSION_EXTERN
#include "zsession.h"
```

`zsession.h` begins with these lines that define the macro `ZSESSION_EXTERN` whose replacement text is the string `extern`, only in those cases where it is not already defined:

[1] I learned this technique from Gregg Traction of the Department of Radiation Oncology at the University of North Carolina. I haven't seen it in any published source.

```
/* zsession.h */

#ifndef ZSESSION_EXTERN
#define ZSESSION_EXTERN extern
#endif
```

Later `zession.h` and contains this line

```
ZSESSION_EXTERN Z_Mode mode;
```

The C pre-processor translates the macro `ZSESSION_EXTERN` to the replacement text `extern` in all files except `zsession.c`, where it translates it to the empty string instead. This has the effect of defining `mode` in the `zsession` module and declaring it external in all other modules that depend on it.

Likewise, every other module `foo` defines and uses a macro `FOO_EXTERN` in the same way.

We use a similar conditional compilation technique to initialize a variable in its definition, see for example `p_status_text` in `zfield.h` and `zfield.c`.

Bibliography

[1] L. R. Dalesio, M. R. Kraimer, and A. J. Kozubal. EPICS architecture. In C. O. Pak, S. Kurokawa, and T. Katoh, editors, *Proceedings of the International Conference on Accelerator and Large Experimental Physics Control Systems*, pages 278–282, 1991. ICALEPCS, KEK, Tsukuba, Japan.

[2] Jonathan Jacky. *The Way of Z: Practical Programming with Formal Methods.* Cambridge University Press, 1997.

[3] Jonathan Jacky. Clinical neutron therapy system installation and operations. Technical Report 99-08-01, Department of Radiation Oncology, University of Washington, Box 356043, Seattle, Washington 98195-6043, USA, August 1999.

[4] Jonathan Jacky. Lessons from the formal development of a radiation therapy machine control program. In Michael G. Hinchey and Jonathan P. Bowen, editors, *Industrial-Strength Formal Methods in Practice*, pages 185–206. Springer-Verlag, 1999.

[5] Jonathan Jacky. Formal safety analysis of the control program for a radiation therapy machine. In Wolfgang Schlegel and Thomas Bortfeld, editors, *The Use of Computers in Radiation Therapy: XIIIth International Conference*, pages 68–70. Springer, 2000.

[6] Jonathan Jacky, Michael Patrick, and Jonathan Unger. Formal specification of control software for a radiation therapy machine. Technical Report 95-12-01, Radiation Oncology Department, University of Washington, Seattle, WA, December 1995.

[7] Jonathan Jacky and Ruedi Risler. Clinical neutron therapy system reference manual. Technical Report 99-10-01, Department of Radiation Oncology, University of Washington, Box 356043, Seattle, Washington 98195-6043, USA, October 1999.

[8] Jonathan Jacky and Ruedi Risler. Clinical neutron therapy system therapist's guide. Technical Report 99-07-01, Department of Radiation Oncology, University of Washington, Box 356043, Seattle, Washington 98195-6043, USA, July 1999.

[9] Jonathan Jacky, Ruedi Risler, Ira Kalet, Peter Wootton, and Stan Brossard. Clinical neutron therapy system, control system specification, Part II: User operations. Technical Report 92-05-01, Radiation Oncology Department, University of Washington, Seattle, WA, May 1992.

[10] Jonathan Jacky and Jonathan Unger. From Z to code: A graphical user interface for a radiation therapy machine. In J. P. Bowen and M. G. Hinchey, editors, *ZUM '95: The Z Formal Specification Notation*, pages 315–333. Ninth International Conference of Z Users, Springer-Verlag, 1995. Lecture Notes in Computer Science 967.

[11] Jonathan Jacky, Jonathan Unger, Michael Patrick, David Reid, and Ruedi Risler. Experience with Z developing a control program for a radiation therapy machine. In Jonathan P. Bowen, Michael G. Hinchey, and David Till, editors, *ZUM '97: The Z Formal Specification Notation*, pages 317 – 328. Tenth International Conference of Z Users, Springer-Verlag, 1997. Lecture Notes in Computer Science 1212.

[12] Brian W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice-Hall, second edition, 1988.

[13] David Alex Lamb. *Software Engineering: Planning for Change*. Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[14] Adrian Nye. *Xlib Programming Manual*. O'Reilly and Associates, Inc., Sebastopol, CA, 1988.

[15] David Lorge Parnas and Paul C. Clements. A rational design process: How and why to fake it. *IEEE Transactions on Software Engineering*, SE-12(2):251–257, February 1986.

[16] David Lorge Parnas, Paul C. Clements, and David M. Weiss. The modular structure of complex systems. *IEEE Transactions on Software Engineering*, SE-11(3):259–266, March 1985.

[17] Ruedi Risler and Jonathan Jacky. Technical description of the clinical neutron therapy system. Technical Report 90-12-02, Radiation Oncology Department, University of Washington, Seattle, WA, December 1990.

[18] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, New York, second edition, 1992.

[19] Wind River Systems, Inc., Alameda, California. *VxWorks Programmer's Guide 5.3.1*, 1997.

[20] Wind River Systems, Inc., Alameda, California. *VxWorks Reference Manual 5.3.1*, 1997. Edition 1.