P.J.Maker

ACS

ACS: The Architecture

Version:1.4 , 83 pages.

Original Design by: P.J.Maker
Revised: by L.Peppou
Document by: I.Stead

Released: 88/04/13
Printed: 88/04/13

Abstract

This document describes the architecture of the ACS implant software. The modular structure of the software is described, together with the policy decisions that affect the architecture, and an overview of the implant software is provided. The individual modules are listed, and an outline description is provided of the requirements and mission of each module, including the interfaces between modules.

It should be noted that the module descriptions in Chapter 7 have not yet been brought up to the same standard as those in Chapter 6.

Contents

List of Figures

Chapter 1

Introduction

1.1 Scope

This document describes the architecture of the ACS implant software. Detailed descriptions of the individual modules are to be found in the appropriate module design documents.

This document contains the following chapters:

Chapter 1 defines the scope of the document, lists references to associated documents and defines some ACS jargon;

Chapter 2 provides a brief overview of the ACS implant software;

Chapter 3 sets out the general policy decisions which affect the architecture of the system;

Chapter 4 provides a full list of all of the modules in the system and states the mission of each module;

Chapter 5 contains a description of the kernel, which is the 'operating system' for the ACS;

Chapter 6 contains a detailed description of each of the therapy controlling modules.

Chapter 7 contains a detailed description of each of the services modules.

1.2 References

1.2.1 Source references

The architecture design is based on the following documents:

1. Arrhythmia Control System: Formal Specification 1.5, sw118.
2. Arrhythmia Control System Specification Version 1.0, sw256, and draft of version 2.0.
3. ACS Programmer/Implant Telemetry Protocol.
4. Preliminary Overview of Safety Engineering.
5. Specification of the X92/Micro Interface, sw179.

1.2.2 Further reading

For further details regarding aspects of the implant software, consult the following documents:

1. OS-ACS: The Kernel, Version 1.12, sw270.
2. Module design documents for individual modules.

1.3 Terminology

The following terms are used in this document:

class - one of ten priority divisions into which processes are categorised. Within each class, further priorities are defined, to indicate the relative priority of each process.

context - this defines the internal condition of a process. It includes such information as the registers and stack variables.

module - a discrete functional component of the ACS software, eg. **pacing therapy**. A module performs one task.[1]

preemption - the suspension of one process to enable another process to start.

process - a software object that may be in one of four states: active, suspended, ready or dormant.

In this, and other ACS documents, the abbreviation MS is used to indicate a time interval of 0.977ms, which is the fundamental timing unit of the X92 control chip (10^{-10}s).

The following conventions have been adopted in this document:

1. Module names are given in bold type, eg. **pacing therapy**.

[1]See section 3.1.

2. Macros are given in upper case, eg. STOP_READY_RAISED.
3. Function names are given in lower case, eg. get_wallclock().

1.4 Guide to the figures

The symbols used in the figures in this document are defined in Figure 1.1.

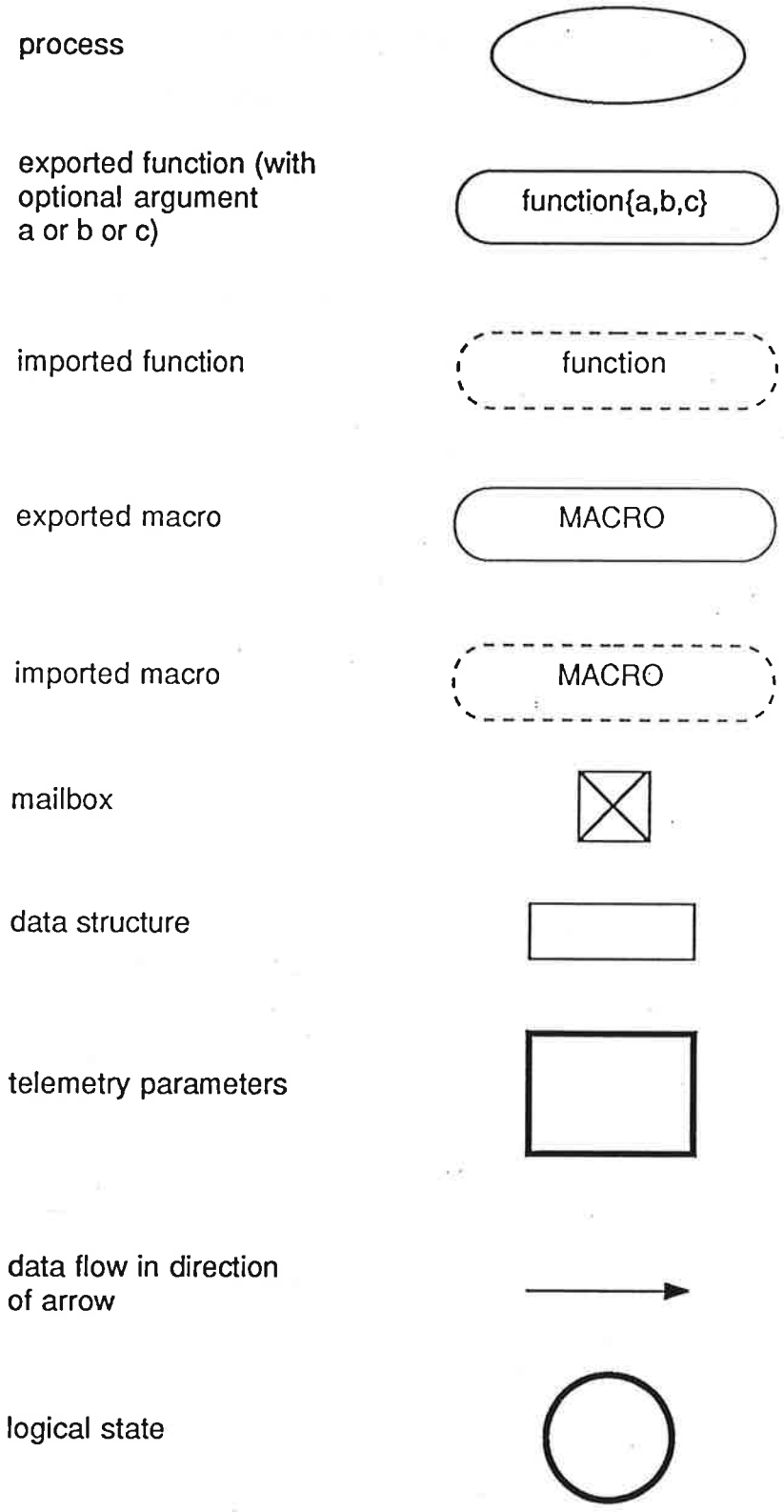

Figure 1.1: Key to Symbols

Chapter 2

Overview

2.1 General

The ACS implant software is constructed from a number of discrete modules, which are defined in terms of functions of the system. Thus, for example, the **noise detection** module is responsible for detecting noise. Some functional areas are handled by just one module (eg. exceptions, which is handled by **exceptions handling**), whereas other areas are split between a number of modules (eg. telemetry, which is handled by **telemetry primitives** and **telemetry commands**).

These modules are divided into the following categories:

1. The kernel - the kernel provides the essential scheduling, context saving and message passing facilities that enable the therapy controlling objects to operate.

2. Therapy controlling modules - these are the higher level components of the ACS which detect arrhythmias and perform the arrhythmia control therapy. The therapy controlling modules are:

   sequencer

   exception handling

   listener

   sensing and recent history

   detection

   confirmation

   pacing therapy

   pacing primitives

defib therapy

defib primitives

noise therapy

noise detection

magnet therapy

magnet handler

averager

telemetry primitives

telemetry commands

data logging

measurements

3. Services modules - these modules provide low level services to the rest of the system. The services modules are:
   self testing

   manufacturing testing

   execution tracing

   X92 handler

   X92 services

   timers

   start up and shutdown

   down line loader

   ports

   testing rig software

   utilities

2.2 ACS operation

2.2.1 Therapy states

The flow of control for therapies in automatic mode[1] is shown in Figure 2.1. The normal state of the ACS is arrhythmia detection, during which brady

[1] Refer to the ACS specification for details of system modes.

support pacing may be given, if required. If a tachyarrhythmia is detected, ACS enters pacing therapy state. If pacing therapy reverts the tachyarrhythmia, ACS returns to detection state; if the tachyarrhythmia continues, ACS enters defib therapy state where it stays until the tachyarrhythmia is reverted. If ACS detects noise when in detection state, it enters noise therapy state to prevent false tachyarrhythmia detection.

The pacing therapy state is shown in Figure 2.2. Confirmation of the tacharrhythmia is performed immediately on entering pacing therapy, and a check is then made to see if pacing trains are required. If no trains are required, ACS enters defib therapy state; if trains are required, the appropriate train scan and energy are determined and the train is delivered. After delivery of the train, the tachyarrhythmia is again confirmed. If confirmation reveals that the tachy has reverted, ACS returns to detection state (see Fig 2.1).

Defib therapy state is shown in Figure 2.3. Confirmation of the tacharrhythmia is performed immediately on entering defib therapy, and a check is then made to see if defib shocks are required. If no shocks are required, confirmation continues; if shocks are required, the appropriate shock energy is determined and the shock is delivered. After delivery of each shock, the tachyarrhythmia is again confirmed. If confirmation reveals that the tachyarrhythmia has reverted, ACS returns to detection state (see Fig 2.1). If the tachyarrhythmia is still present, ACS remains in defib therapy state.

2.3 The kernel

The kernel provides operating system facilities to the implant software. In particular it provides the following:

1. Process scheduling. The kernel defines the rules applied to the scheduling of processes in the system, including a limited amount of preemption.

2. Context saving. The kernel provides facilities for saving information on the state of processes that are preempted by other processes.

3. Message passing. The kernel defines the message handling facilities provided for communication between processes.

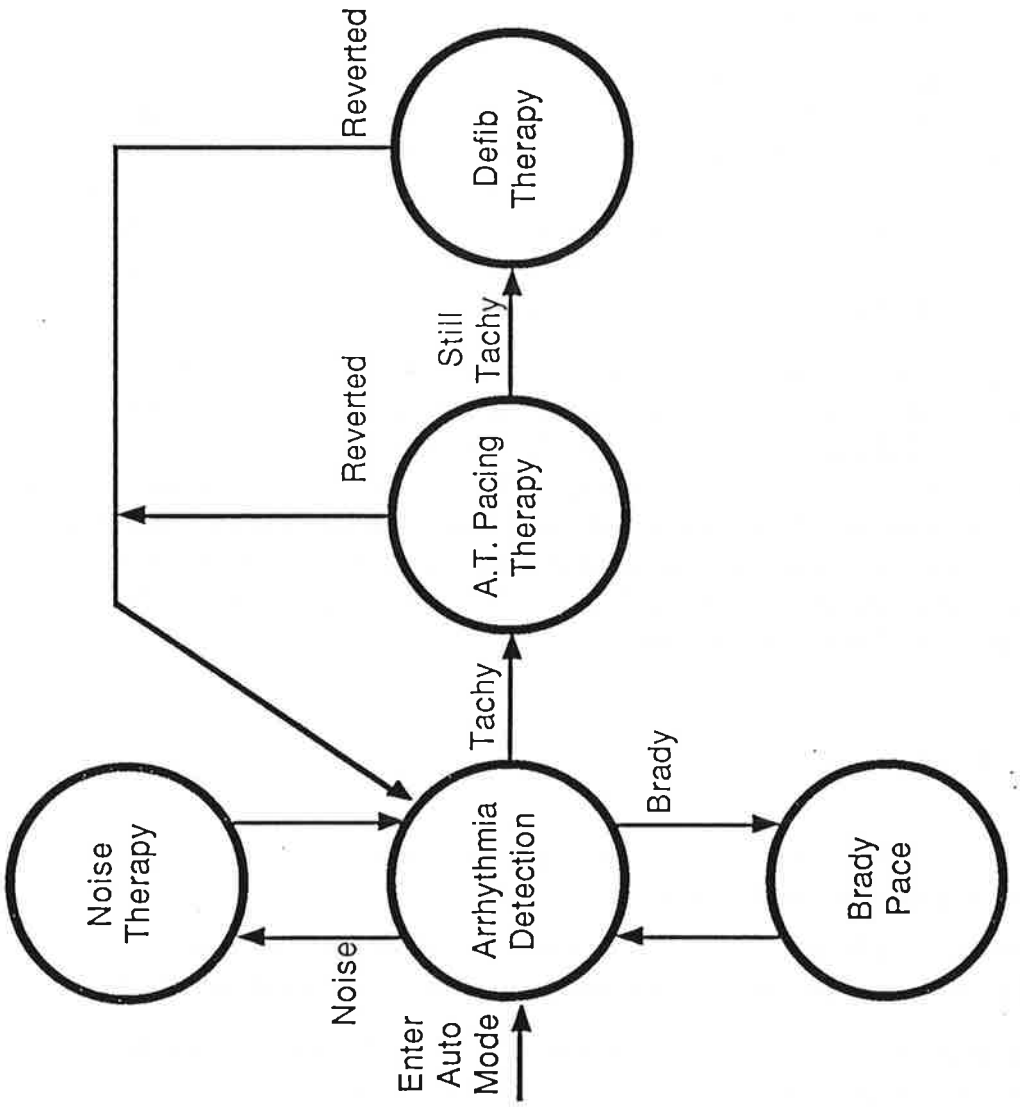

Figure 2.1: Flow of Control for Therapies in Automatic Mode

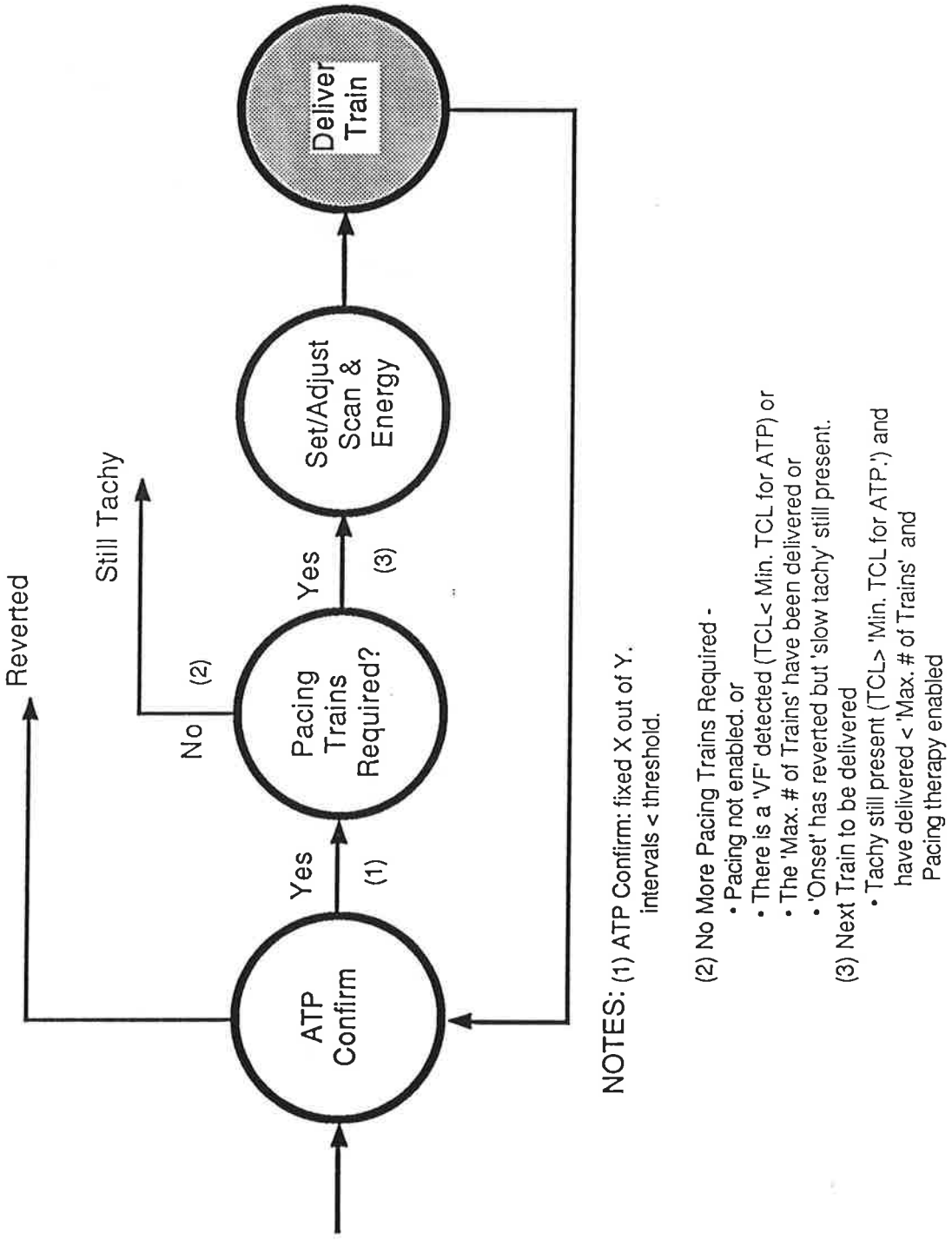

Figure 2.2: Pacing Therapy State

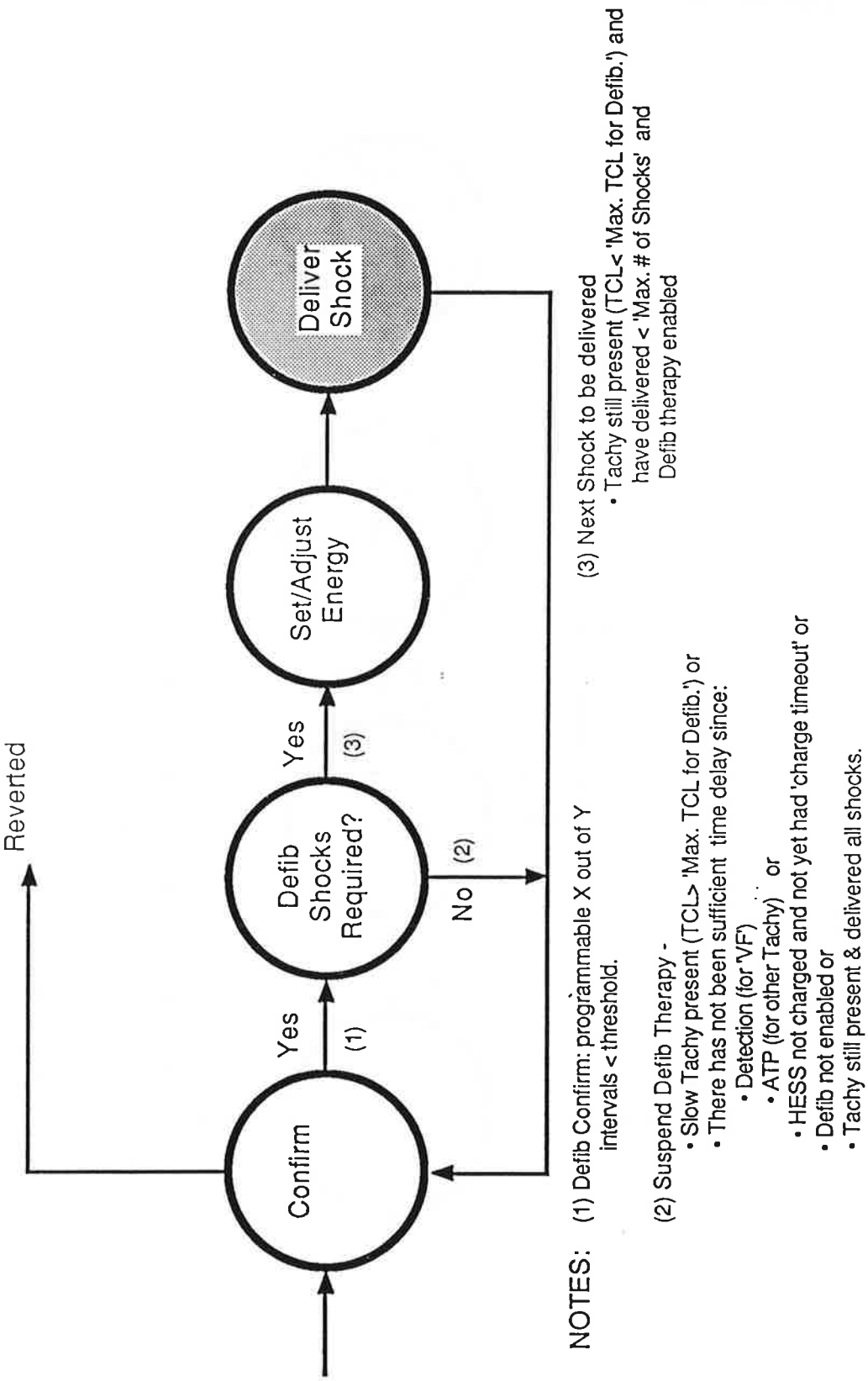

Figure 2.3: Defib Therapy State

2.4 Therapy controlling modules

2.4.1 Therapy sequencing

The **sequencer**[2] is responsible for moving ACS through the various therapy states. It responds to changes detected in the outside world (eg. tachyarrhythmia present, noise present, magnet present) which should produce a change of state (eg. moving from detection state to pacing therapy state). The possible state transitions are shown in Figure 2.4 (this figure is an expansion of Figure 2.1).

Before leaving one state to move to another, the **sequencer** performs necessary administrative tasks (eg. switching detection off immediately after detecting a tachyarrhythmia).

Some changes in the outside world (eg. magnet present) need to stop the system in its tracks in a controlled way.[3] When such an event occurs, the part of the system responsible for detecting the event (in our example, **magnet handler**) raises an exception (see section 2.4.2 for details). This causes any current therapy to abort and return to the **sequencer**. The **sequencer** then takes appropriate action in response to the exception and also clears the exception (analogous to servicing and clearing an interrupt).

2.4.2 System behaviour under exception conditions

Exceptional events that occur in the system are reported to **exception handling**[4] by other modules, using a set of exception codes.[5] **Exception handling** classifies the events into three degrees of urgency. The most urgent exceptional events represent a shutdown condition and result in an immediate call to a SHUTDOWN macro in **start up and shutdown**; the state of the system is saved for debugging and the μP is reset. Less urgent exceptional events result in one of two exception conditions being called:

1. STOP_WHEN_READY
2. STOP_ASAP

The exception conditions are distributed to other modules in the system and cause them to stop operation and hand control back to sequencer. **Sequencer** then polls **exception handling** to determine the cause of the

[2] See section 6.1.

[3] Rather like doing an emergency stop without locking the rear wheels or stalling the engine.

[4] See section 6.2.

[5] This is referred to as *raising an exception.*

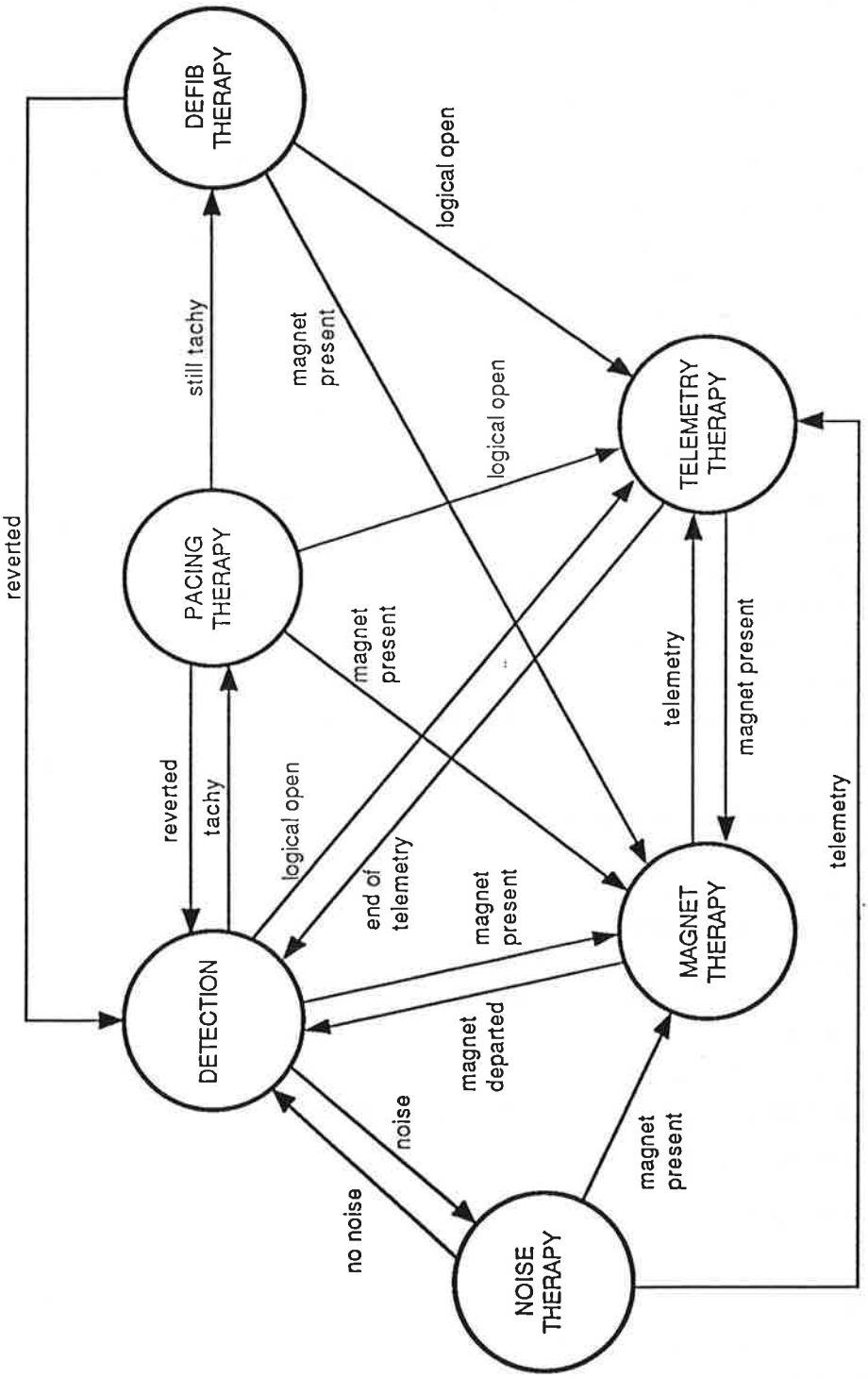

Figure 2.4: Transitions Between ACS Therapy States

exception condition and takes appropriate action. The speed with which an individual module reacts to the exception conditions and returns to **sequencer** is determined within the module, not by **exception handling.**

2.4.2.1 Magnet therapy

When a magnet is detected, **X92 services**[6] informs **magnet handler**[7] which raises an exception. The **sequencer** calls **magnet therapy**[8] to provide therapy to the patient while the exception remains raised. **Magnet handler** extends the effect of the magnet for a preset period after the magnet departs.

2.4.2.2 Telemetry

When the telemetry link is opened, **telemetry primitives** raises one of two exceptions and the **sequencer** instructs the telemetry subsystem to process the telemetry interaction (see section 2.4.7).

2.4.3 The listener

Modules that observe cardiac events, and initiate actions in response to those events, are integrated in a meta-module called **listener**[9] which coordinates the operations of the individual modules. The following modules are integrated within **listener**:

- **sensing and recent history**
- **averager**
- **noise detection**
- **detection**
- **confirmation**

Listener provides input and output facilities, and defines the order in which the modules within **listener** do their work. For example, **listener** ensures that a sensed event is passed to **averager** before **detection** when the ACID algorithm is being used. An overview of **Listener** is shown in Figure 2.5

Telemetry parameters for modules within **listener** are written/read via **listener.**

[6]See section 7.5.

[7]See section 6.9.

[8]See section 6.15.

[9]See section 6.3.

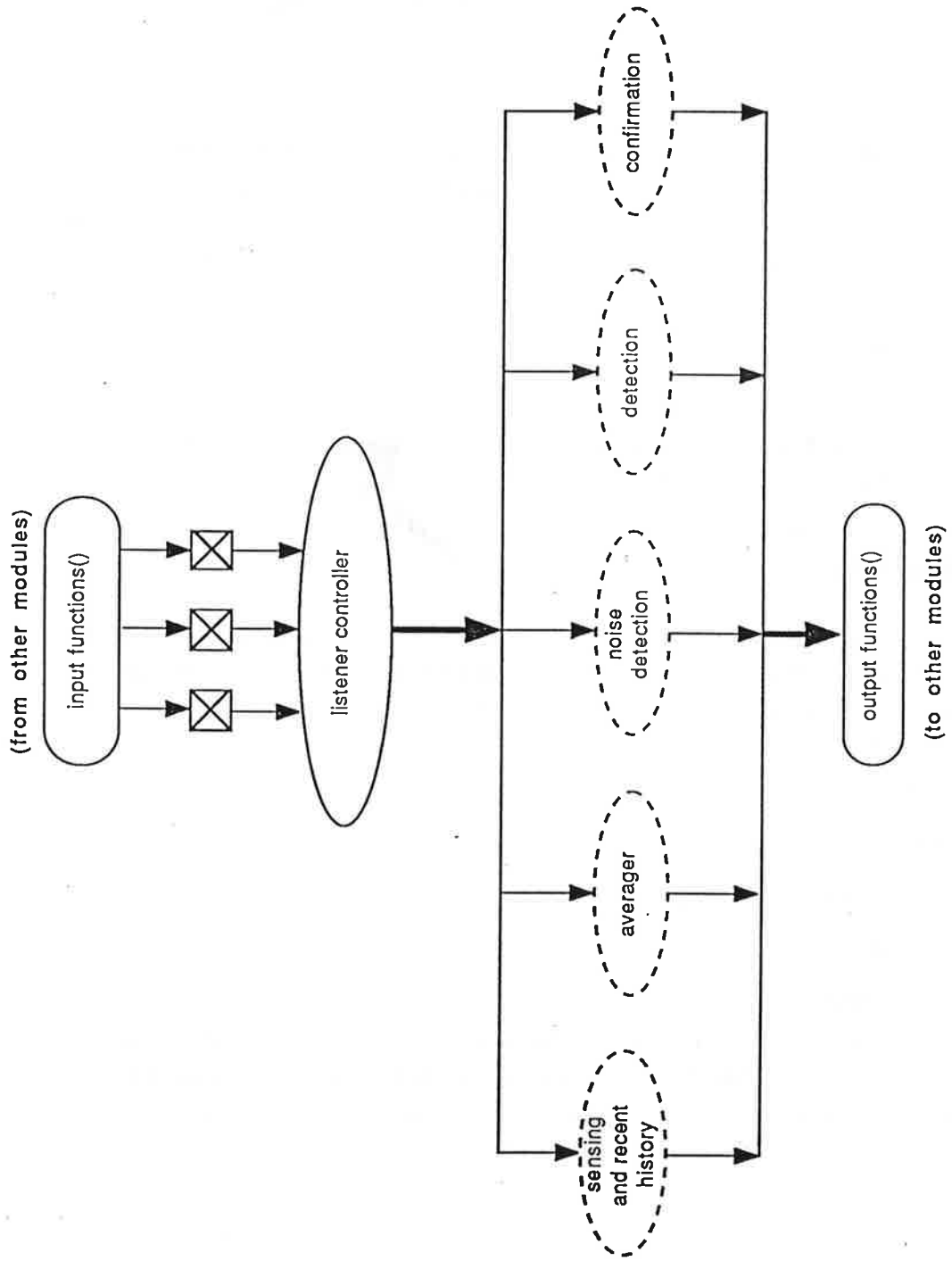

Figure 2.5: Overview of The Listener

2.4.4 Pacing subsystem

The pacing subsystem delivers anti-tachycardia pacing (ATP) to the patient.

Pacing therapy[10] applies confirmation to the recent past and, if necessary, calls **pacing primitives**[11] to deliver the next train. **Pacing primitives** calculates the timing and pacing energy of each train, in accordance with telemetry parameters, and delivers the appropriate number of trains.

2.4.5 Defib subsystem

The defib subsystem delivers defib therapy to the patient.

Defib therapy[12] applies confirmation to the recent past and, if necessary, tells **defib primitives**[13] to prepare for the next shock. **Defib primitives** has access to the shock train (the table of shocks programmed by the physician); it charges the capacitors for the next shock in the sequence and informs **defib therapy** when it is ready to shock. If **defib therapy** still wants to deliver the shock, it issues a command to **defib primitives** to deliver the shock.

Defib primitives has sole responsibility in the system for dumping energy internally; it is the only module that knows when dumping should take place. Similarly, it has sole responsibility for periodically reforming the capacitors.

2.4.6 Data logging

Certain system events are logged as they happen to be uploaded at a later time by a programmer.

Data logging[14] is responsible for storing all the logged events, for which it uses two types of record:

1. Incrementing counters which can be read and reset by the programmer.
2. Episode buffers, which map the progression of an episode of tachyarrhythmia and the implant's response. These also can be read and reset by the programmer.

Other modules in the system inform **data logging** of the occurrence of an event; **data logging** then formats the data in a form suitable for uploading.

[10] See section 6.10.

[11] See section 6.11.

[12] See section 6.12.

[13] See section 6.13.

[14] See section 6.18.

2.4.7 Telemetry subsystem

The telemetry subsystem handles all telemetric interactions with the programmer. There are two main types of interaction:

1. Commands.
2. Data accesses of the telemetry space.

The command interactions consist of a command sent by the programmer followed by execution of the command and reporting of the result by the implant back to the programmer. Data accesses consist of single bytes read and written from the implant telemetry space.

The interactions take place across the telemetric link which can be in one of three states:

1. Closed — no communication taking place.
2. Physically open — the programmer is talking directly to the X92 but the μP is not involved.
3. Logically open — communication is established between the programmer and the μP.

The telemetry software is split into two distinct modules: **telemetry primitives**[15] and **telemetry commands**[16].

Telemetry primitives maintains the telemetry address space and the state of the logical link. It also recognises the completion of a command sequence and informs the **telemetry commands** module that a command is to be processed.

Telemetry commands does all of the intelligent command processing work. When it receives a command from **telemetry primitives**, it performs the actions required to process that command. Upon completion of the command, **telemetry commands** informs **telemetry primitives** that the command has been processed.

2.5 Services modules

Services modules provide low level services essential to the operation of the rest of the software. Some of these services relate to interactions with the ACS hardware (eg **X92 services, ports**), whilst others provide common software facilities used by other areas of the implant software (eg **timers, utilities, self testing**).

[15]See section 6.16.

[16]See section 6.17.

Chapter 3

Policies

This chapter describes the policy decisions that have been made in designing this architecture. These policy decisions affect the details of the implementations of each of the component modules of this system.

3.1 General policies

1. A module should be defined to perform each distinct task in the system, even if this task is rather small.
2. Modules will not be organised into a hierarchical structure, but will be able to communicate with each other as necessary.
3. All modules shall be designed for testability.
4. The module breakup shall be designed to improve testablity.
5. All operations should be robust and safe under all possible and some impossible[1] error conditions.
6. Modules that have a critical effect on the system duty cycle must be time efficient.

3.2 Kernel policies

The following decisions have been made in the kernel:

[1]The software should aim to be safe even when presented with a theoretically impossible condition. For example the HESS control software should be safe when 'impossible' calling sequences occur.

1. Processes will be statically allocated.
2. Processes will be categorised into a number of priority classes.
3. Processes may preempt processes in lower classes.
4. Preemption of processes within a class will occur only on the request of the preempted process.
5. Each process will have a specified amount of μP time which it can use before rescheduling.

   For example, a low priority process in a class may be required to do a reschedule every 6ms to see if a higher priority process in the same class has become ready[2]. The highest priority process in a class will run to completion without rescheduling.
6. Message passing will be used as a means of communication between processes.

3.2.1 Allocating processes

If a module should be implemented as a process, then a number of decisions need to be made:

1. Which class of processes should this one be in?[3]
2. What are the scheduling requirements for this process?[4] In particular what sort of priority should be given to this process?
3. What method of context saving is required for this process?[5]

3.3 Communication

This section describes the methods that will be used to communicate between the components of the system.

[2]This constraint must be implemented in individual modules; it cannot be enforced by the kernel.

[3]See page 32 for further details.

[4]See page 33 for further details.

[5]See page 35 for further details.

3.3.1 Message passing

The message passing system defined by the kernel provides a first-in-first-out (FIFO) method of communication between processes in the system. Two types of data structure are defined for message passing in ACS:

1. Mailboxes. These are data structures which contain a single piece of user defined information.
2. Pipes. These are fixed length byte streams which provide greater capacity than mailboxes.

A message will contain any information that is required by the receiving process (the server). For example a 'sense arrived' message would normally also contain the interval between this sense and the previous one.

3.3.1.1 Reasons to use message passing

1. For communication between interrupt handling processes and the other processes in the system.

   This allows the arrival rate of interrupts to increase for a short period of time, outpacing the servers, without adversely affecting the system.

2. To decouple processes.

   For example, if a complicated pacing algorithm requires a large amount of computation, it could be implemented as two processes. One would do the individual pacing trains whilst the other would use the rest of the μP time to perform the calculations. The message mailbox or pipe between them is used to decouple these processes.

3. For general communications between processes.

3.3.1.2 Points to consider when using messages

1. Messages to be sent.
2. Sending processes for these messages.
3. Receiving processes for these messages.
4. The valid history for each message to be sent to this mailbox or pipe (a message may only be valid if it follows a certain sequence of messages, or if other conditions are in force).
5. The size of the mailbox or pipe.
6. Action on overflow of the mailbox or pipe.

3.3.2 Function calls

Function calls and argument passing can be used to communicate between processes.

3.3.2.1 Reasons to use function calls

1. Function calls are rather efficent since they are supported by the μP hardware.
2. If a set of operations need to be implemented which have the same priority as the calling process then they should be implemented as a function library.

   For example, a process which evaluates the checksum of a piece of store should be implemented as a function since it should run at the priority of the calling process. This is necessary so that a call to this process is running at the correct priority.

3.3.2.2 Points to be considered about function calls

1. Are the functions reentrant?
2. What are the valid histories of this process? For example must every call to function A be followed by a single call to function B?

3.3.3 Shared data structures

3.3.3.1 Points to be considered about shared data structures

1. Mutual exclusion.
   (a) Using the scheduler.

   If the processes accessing this data structure are in the same class then they can be assured of mutual exclusion because of the scheduling of the kernel.[6]

   (b) Disabling interrupts.

   Interrupts can be disabled for mutual exclusion, but must not be turned off for more than 2MS.

[6] The regions of mutual access should still be noted since a change to the the kernel scheduling is possible in the future (using the X.sys file).

2. Valid updates.

   If a process updates data in a shared data structure, the update must be complete before another process can access the data. Mutual exclusion will ensure that processes accessing the data structure do not access invalid data.

3.4 Exception conditions

3.4.1 Definition

"An exception is any event which causes an abrupt change of state of the system."[7]

3.4.2 Examples

1. "Magnet application"
2. "Telemetry polite open"
3. "Telemetry rude open"
4. "Store corruption"
5. "CPU self test failure"

3.4.3 Policy decisions

1. The exception process will define a mapping from the many possible exception conditions onto a smaller set which will be used by the rest of the processes. For example the exception condition A might be defined to occur whenever X or Y or Z is true.
2. The mapping described in the previous item will be static.
3. Exception conditions will be polled by the rest of the system.
4. A mechanism will be provided for processes which need to know if an exception has happened between two successive polls.

[7] This definition includes such events as the application of a magnet and opening of the telemetry link. It is a broadening of the idea of exceptions as being simply hardware/software error conditions.

Chapter 4

A summary of the modules

4.1 Kernel

The facilities provided by the kernel are listed below with a brief outline of their function.

Scheduling — responsible for sharing the μP between a number of distinct processes, each of which is allocated a priority rating within one of ten classes.

Context saving — provides the facilities for saving and restoring the context of a process. This allows other processes to be preempted by the scheduler while we preserve the current state of this process.

Message passing — provides facilities for sending and receiving messages between the component processes of ACS.

4.2 Therapy controlling modules

The therapy controlling modules are listed below with a brief statement of their missions.

Sequencer — the overall manager of all therapies in ACS.

Exception handling — reports on exceptional events in the ACS. An exceptional event is one that causes an abrupt change of state of the system.

Listener — integrates the operation of modules that listen to cardiac events.

Sensing and recent history — decides what to do with sensed events and maintains a representation of sensed events in the recent past.

Detection — responsible for detecting tacharrhythmias and salvos.

Confirmation — responsible for confirmation of tachyarrhythmias after pacing therapy or defib therapy.

Noise detection — detects noise.

Averager — maintains interval averages.

Magnet handler — tells the system when a magnet is present.

Pacing therapy — controls the delivery of anti-tachycardia pacing therapy to the patient.

Pacing primitives — delivers individual pacing trains.

Defib therapy — controls all defib therapy given to the patient.

Defib primitives — delivers individual shocks.

Noise therapy — provides therapy to the patient when noise is present.

Magnet therapy — provides therapy to the patient when a magnet is present.

Telemetry primitives — handles the logical link protocol.

Telemetry commands — processes programmer commands received over the telemetry link.

Data logging — provides a collecting place for all logged data in the system.

Measurements — handles all internal and external measurements in the system.

4.3 Services

The services modules are listed below with a brief statement of their missions.

Self testing — defines the routines used for self testing of the ACS, which may be initiated by a telemetry command.

Manufacturing testing — provides support for the manufacturing test system.

Execution tracing — maintains a log of the execution of the system for debugging of failed ACS implants.

X92 handler — responsible for the handshaking and control of the X92.

X92 services — provides a higher level interface to the X92 handler.

Timers — provides timing services to the rest of the system.

Startup and shutdown — defines the actions that will occur upon system startup and shutdown.

Down line loader — responsible for bringing code/data into the implant via the telemetry link.

Ports — provides a reliable interface to the μP ports. It is required because certain operations on the I/O ports must be atomic.

Testing rig software — provides an interface to the outside world for testing purposes.

Utilities — provides general utilities used in this system.[1]

[1] The place for things without another place.

Chapter 5

Details of the kernel

5.1 Introduction

The kernel[1] provides the following facilities for the implementation of ACS:

1. Scheduling the μP between a number of competing processes.
2. A simple form of context saving to enable processes to suspend and resume execution.
3. Inter-process communication using mailboxes and pipes.

5.2 Process scheduling

5.2.1 Process classes

This system provides scheduling for a constant number of predefined processes. These are divided into ten classes:

Class 8 processes - hardware interrupt handlers and other processes that must not be preempted.

Class 7 to 0 processes - processes in classes 7 to 0 perform the majority of operations of the system.

Class -1 processes - this is a special class containing only one process which runs when no other process is ready to run. This process places the μP into its low power mode.

[1]Softer (usu. edible) part within hard shell of nut (Concise Oxford Dictionary).

Within classes 0 to 8, each process is allocated a priority relative to the other processes in the same class.

5.2.2 Process states

Each process may be in one of the following states:[2]

active - a process is active if it is currently running on the μP.

preempted - a process is preempted if it has been halted by the scheduler to allow a higher priority process to run.

ready - a process is ready if it is waiting for the scheduler to make it active.

dormant - a process is dormant when it is not active, preempted or ready to run.

5.2.3 Scheduling

Process scheduling is performed at two levels:

1. Processes are scheduled according to the class that they are in. Thus, a class 5 process may preempt processes in classes 4, 3, 2, 1, 0, and -1, but may itself be preempted by processes in classes 8, 7 or 6.
2. Processes within classes are each allocated a priority which is used to determine which process takes precedence when two or more processes are ready to run in the same class. However, once a process is running, it cannot be preempted by another process in the same class except at its own request.

A process may need to be able to prevent its preemption by higher priority processes for the following reasons:

1. To ensure mutual exclusion of access to data.
2. Because of timing constraints.

This is achieved by allowing a process to temporarily promote itself into a higher class (its *effective class*). For example, a class 2 process could set its effective class to class 4 and so prevent its preemption by processes in classes 3 and 4. At some later time it would reset its effective class back to its own level (its *scheduling class*).

The scheduling scheme presented here results in the following conditions:

[2]The exception to this is the process in class -1, which will never be dormant.

1. Data structures passed between processes within a single class will always be consistent since a context switch within a class cannot occur without the active request of the rescheduled process.

2. If a low priority process within a class is running then it will run to completion or rescheduling even if a higher priority process in the same class becomes ready. This means that processes which take a large amount of time may need to call the scheduler at frequent intervals in order to give fair use of the processor.

5.3 Message passing

Message passing is the main means of communication between processes in this system.[3]

The following terms are used when describing message passing:

Message - a message is a record which contains a unique identifier and some optional arguments.[4]

For example, a message may be sent to indicate that a sense has arrived with an inter-sense interval of 500ms. Another message may indicate that a tachyarrhythmia has been detected and therapy should commence.

Sender - a process that constructs and sends messages to a particular mailbox. There can be many senders to one mailbox.

For example, the sender for the "tachyarrhythmia detected" message would be the "tachyarrhythmia detection module" which would send a message to the "therapy sequencer" when a tachyarrhythmia is detected.

Server - the process that receives messages from a mailbox and acts on them. There can be only one server for each mailbox.

The server process will be made ready to run whenever a message arrives in the mailbox. It may also be necessary to change the server process for a particular mailbox dynamically.

Overflow handler - this function defines the operations that are performed when a mailbox overflows. A mailbox may have more than one overflow handler.

---

[3] The other methods of global variable access and function calling are used merely for efficiency reasons. They suffer from the problems of mutual update and consistency of data structures.

[4] For any given message identifier the number and type of arguments will be fixed in this system.

5.4 Context saving

Processes can have three different forms of saved context:

No saved context - a process when called from the scheduler simply runs to completion each time. No information is saved about the state of the process on exit. Processes with no saved context include:

1. Interrupt processes - these will normally handshake a few lines and then send off a message.
2. A process which applies a function to its view of the past each time it is started. Detection, for example, applies a function to the history of recent cardiac events and then stops, waiting for the next sense or pace.
3. A finite state machine process which maintains its state within a single variable. Every time this process is called by the scheduler it will perform an operation based on the value of the state variable and its inputs. The state variable is then changed according to some function of state and inputs before the system returns to the scheduler.

Saved program counter - a process switch will merely save the program counter of this process. The following points should be noted:

1. This is the least amount of saved context that can be implemented. Because of this it is reasonably fast and easy to implement.
2. No stack-based variables can exist in a process of this type since the stack can be corrupted between calls to this process from the scheduler.
3. It is only possible to do a process switch when we are in the outermost function of a process since the return addresses from any subfunctions will also be corrupted.

Separate stack and registers [5] - The full version of separate processes requires separate stacks and the saving of registers. The consequences of this are:

1. This type of process can be switched at any time.
2. A process with this type of context switching can have local variables and return to the scheduler from within a called function with no ill-effects.
3. Context switching costs are larger than for the previous method.

[5] This will not be implemented in the current version of the kernel.

4. Hardware interrupts which occur must check to see which stack they are running in before saving all the registers. If they do not do this then each process stack must have enough space for all interrupt processes.

Chapter 6

Details of therapy controlling modules

This chapter contains a detailed description of each of the therapy controlling modules. For each module it gives the mission of the module, a detailed list of requirements and the proposed implementation. Any unresolved issues will be solved during the design phase of these modules.

6.1 Sequencer

Mission

The **sequencer** is the overall manager of all therapies in ACS. See section 2.4.1 for a description of its role in the system.

Requirements

1. Select and initiate the current therapy state of ACS.
2. Perform the administrative tasks involved in changing therapy states.
3. Handle all therapy issues that transcend any single type of therapy.
4. Handle all therapy issues not supported by other therapy modules. This includes work that is done at the instant that the **sequencer** actually uses data that is available over a period of time.
   - after tachyarrhythmia detection:
     (a) Initiate an ECG snapshot.
     (b) Send **measurements** a *tachy detect* MTE.

(c) Send **data logging** a *tachy detect* message and give **data logging** the TCL.

- after noise detection:
  (a) Send **measurements** a *noise detect* MTE.
  (b) Send **data logging** a *tachy detect* message
- after magnet detection:
  (a) Send **data logging** a *magnet detect* message.

5. Handle exceptions that occur whilst the **sequencer** is running.
6. Handle exceptions that cause therapy modules to abort and return to the **sequencer**.
7. Clear exception conditions to show that they have been noticed.
8. Inform **data logging** when noise is detected.
9. Inform **data logging** when a magnet is detected.

Telemetry parameters

None.

Implementation

Conceptually, the **sequencer** knits all ACS therapies together into one massive process. In actuality, each therapy will be defined as a separate process to facilitate development.

The implementation of the **sequencer** is shown schematically in Figure 6.1.

6.2 Exception handling

Mission

To report on exceptional events in the ACS. An exceptional event is one which causes "an abrupt change of state"[1]. See section 2.4.2 for details of the effects of exceptions on the system.

[1] See ACS: Formal Specification, version 1.5, p27.

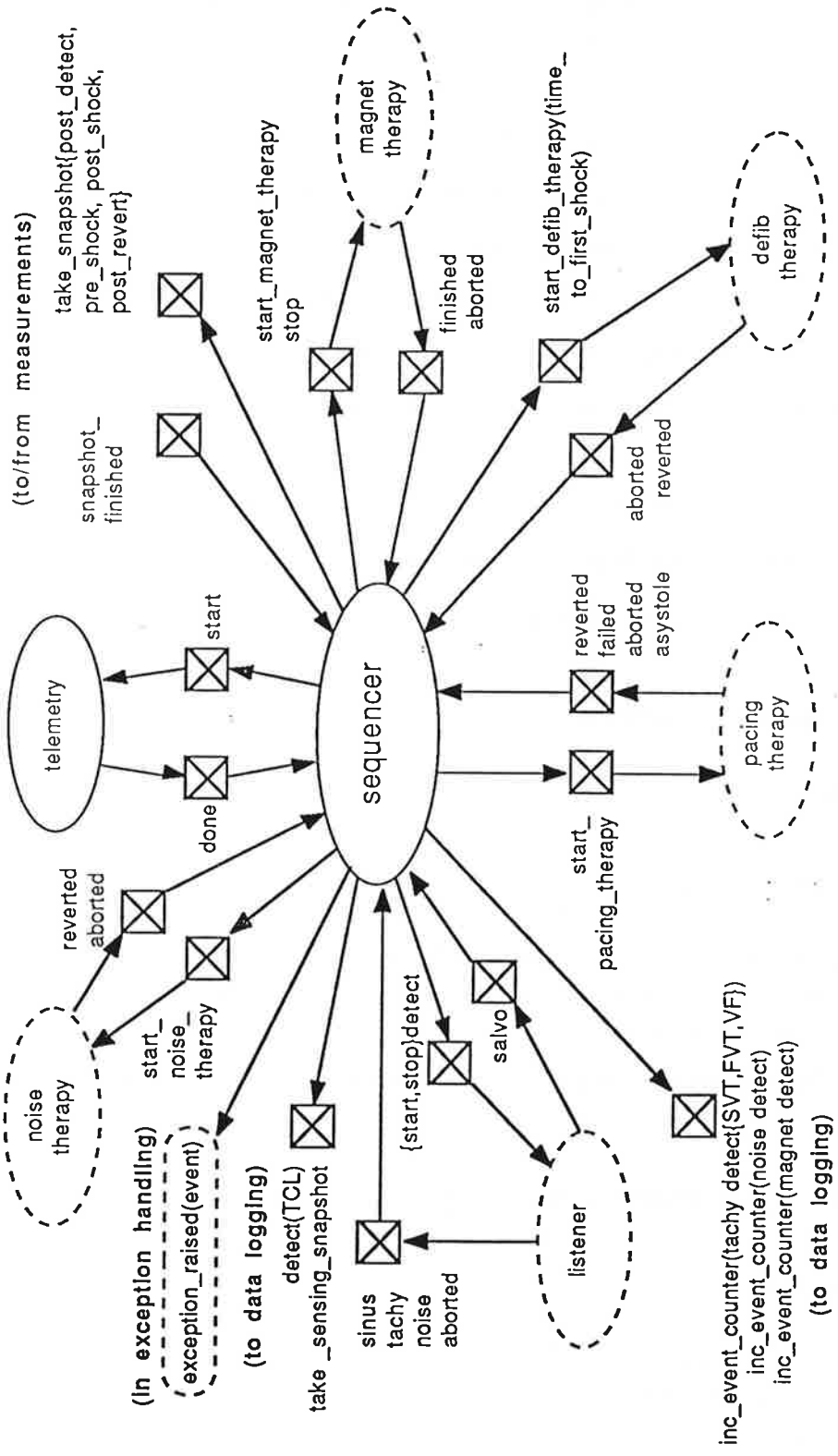

Figure 6.1: Implementation of The Sequencer

Requirements

1. Map many exceptional events onto a smaller number of exception conditions.
2. Allow any module to poll an exception condition.

Implementation

The implementation of **exception handling** is shown schematically in Figure 6.2.

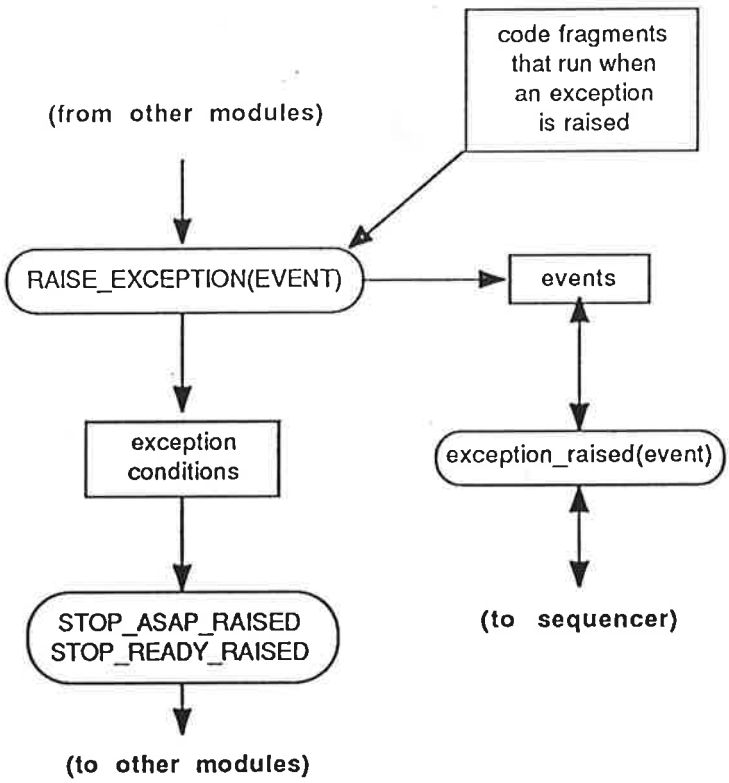

Figure 6.2: Implementation of Exception Handling

6.3 Listener

Mission

Listener integrates the operation of the modules that observe cardiac events and initiate actions in response to those events. See also section 2.4.3.

Requirements

1. Provide a general interface for **sensing and recent history, averager, noise detection, detection** and **confirmation.**
2. Time stamp cardiac events.
3. Define how work is divided among the modules within **listener.**
4. Ensure that the modules within **listener** operate in the correct order.

Telemetry parameters

Telemetry parameters for modules incorporated into **listener** are listed under the individual module descriptions.

Implementation

The implementation of **listener** is shown schematically in Figure 6.3.

6.4 Sensing and recent history

Mission

Sensing and recent history is responsible for deciding what to do with sensed events and maintaining a representation of sensed events in the recent past. See also section 2.4.3.

Requirements

1. Be time efficient in order to keep the system duty cycle reasonable.[2]
2. Inform **pacing primitives** or **defib primitives** when a sense arrives (**pacing primitives** synchronizes the start of a train to a sense, and **defib primitives** may synchronize a shock to a sense).
3. Handle sensed events on high gain or low gain channels (these events can occur simultaneously).
4. Provide a telemetry interface to enable/disable low power detect.

[2]See section 3.1.

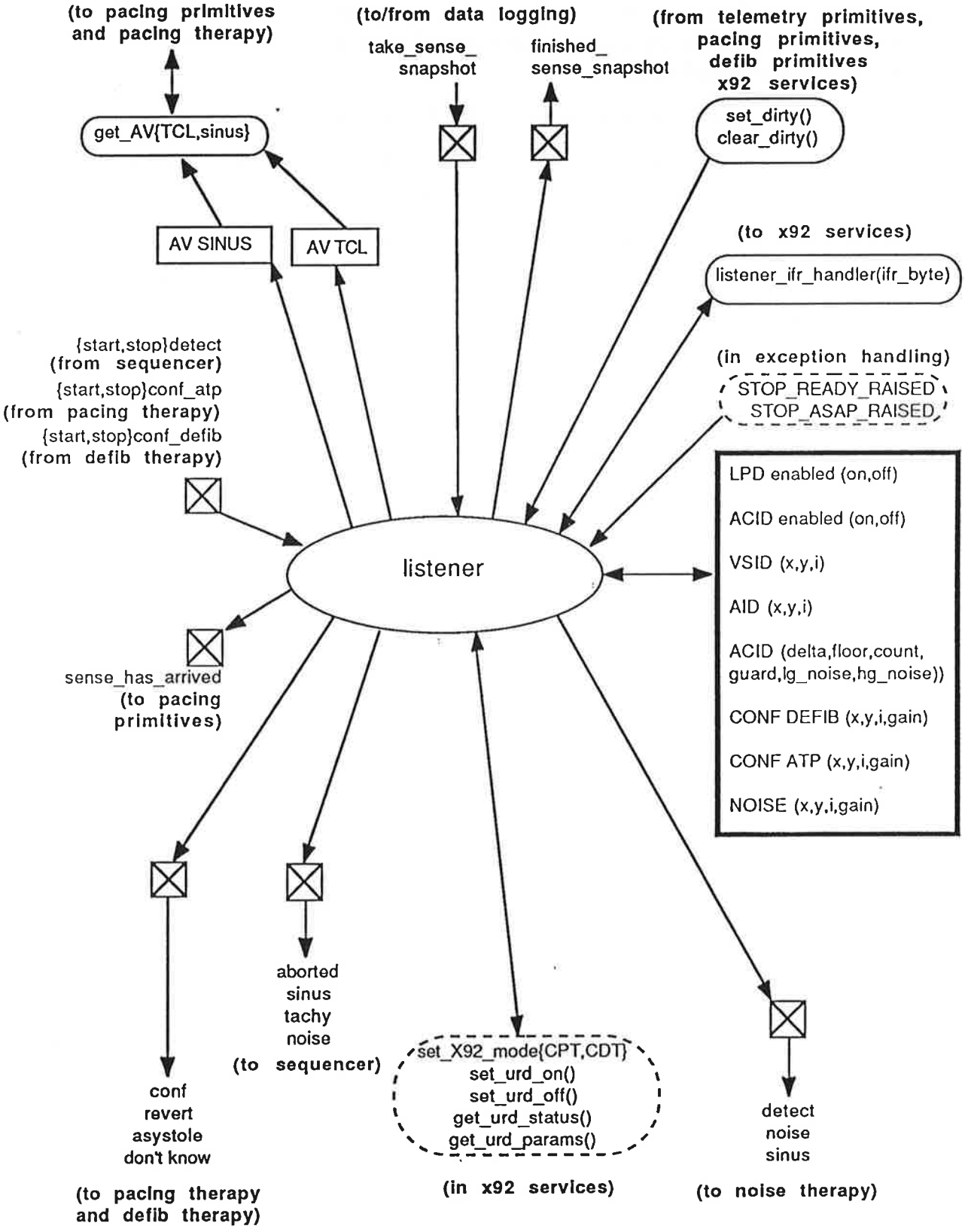

Figure 6.3: Implementation of Listener

5. Provide a mechanism to mark an event (and intervals enclosing the event) as having suspect timing (some independent X92 transactions could cause a sensed event to be delayed).
6. Record the intervals between events in the recent past in a history queue.
7. Provide an interface to the history queue for detection modules (**detection**, **noise detection** and **confirmation**).
8. Provide a facility to take a snapshot of the queue for **data logging**.
9. Handle sense overruns (another sensed event arriving before the previous one was processed).

Telemetry parameters

None.

Implementation

The implementation of **sensing** and **recent history** is included in the implementation of **listener** which is shown schematically in Figure 6.3.

6.5 Detection

Mission

Detection is responsible for the detection of tachyarrhythmias and salvos. See also section 2.4.3.

Requirements

1. Support tachyarrhythmia detection for the following algorithms:
   (a) VSID.
   (b) AID.
   (c) ACID.
   See the informal specification for further details.
2. Allow any combination of the above algorithms to run concurrently.
3. Provide the following outputs:

(a) Tachy detected.

(b) Salvo detected.

(c) Sinus rhythm.

(d) Don't know.

4. Provide a telemetry interface to specify the tachy detection algorithm and the parameters of each algorithm.

5. This module must be time efficent, whether or not detection is enabled, in order to keep the system duty cycle reasonable.[3]

Telemetry parameters

Type	Name	Values or Range	Remarks
	LPD enabled	ON,OFF	All parameters are currently under review.
	ACID enabled	ON,OFF	
	VSID x	?	
	VSID y	?	
	VSID interval	?	
	AID x	?	
	AID y	?	
	AID interval	?	
	ACID delta	?	
	ACID floor	?	
	ACID count	?	
	ACID guard	?	
	ACID lg_noise	?	
	ACID hg_noise	?	

Implementation

The implementation of **detection** is included in the implementation of **listener** which is shown schematically in Figure 6.3.

[3]See section 3.1.

6.6 Confirmation

Mission

This module is responsible for the confirmation of tachyarrhythmias after pacing therapy or defib therapy. See also section 2.4.3.

Requirements

1. Apply confirmation algorithms to the **sensing and recent history** queue.
2. Provide separate confirmation parameters for **pacing therapy** and defib **therapy**.
3. Allow confirmation to be enabled or disabled.
4. Be time efficient when confirmation is enabled or disabled.
5. Provide the following outputs:
   (a) Confirmed.
   (b) Reverted.
   (c) Asystole.
6. Provide a telemetry interface to specify confirmation parameters.
7. Vary confirmation threshold with TCL.

Telemetry parameters

Type	Name	Values or Range	Remarks
	ATP x	?	
	ATP y	?	
	ATP interval	?	
	ATP gain	?	
	Defib x	?	
	Defib y	?	
	Defib interval	?	
	Defib gain	?	

Implementation

The implementation of **confirmation** is included in the implementation of **listener** which is shown schematically in Figure 6.3.

6.7 Noise detection

Mission

This module is responsible for detecting noise. See also section 2.4.3.

Requirements

1. Apply a noise detection algorithm to the recent past.
2. Provide a 'noise detected' output.
3. Be time efficent, since we may spend a large amount of time in a noise field, in order to keep the system duty cycle reasonable.[4]

Telemetry parameters

Type	Name	Values or Range	Remarks
r/w	lg_noise x	?	
r/w	lg_noise y	?	
r/w	lg_noise interval	?	
r/w	hg_noise x	?	
r/w	hg_noise y	?	
r/w	hg_noise interval	?	

Implementation

The implementation of **noise detection** is included in the implementation of **listener** which is shown schematically in Figure 6.3.

[4] See section 3.1.

6.8 Averager

Mission

Averager maintains interval averages. See also section 2.4.3.

Requirements

1. Maintain sinus and Tachy Cycle Length (TCL) averages.
2. Allow the input to the sinus averager to be enabled or disabled.
3. Allow the input to the TCL averager to be enabled or disabled.
4. Allow access to the averages at all times. If the input to an averager is disabled, the average becomes a record of the last known TCL or sinus (as applicable).
5. This module must be time efficent in order to keep the system duty cycle reasonable.[5]

Telemetry parameters

Not known.

Implementation

The implementation of **averager** is included in the implementation of **listener** which is shown schematically in Figure 6.3.

6.9 Magnet handler

Mission

Magnet handler informs the system when a magnet is present. See also section 2.4.2.1.

[5] See section 3.1.

Requirements

1. Detect the presence of a magnet, and raise a "magnet arrived" exception.
2. Detect the absence of a magnet.
3. Extend the effect of a magnet as specified in the formal specification.
4. Tell **magnet therapy** when the magnet effect has ended.
5. Respond to STOP_WHEN_READY and STOP_ASAP exceptions.

Implementation

The implementation of **magnet handler** is shown schematically in Figure 6.4.

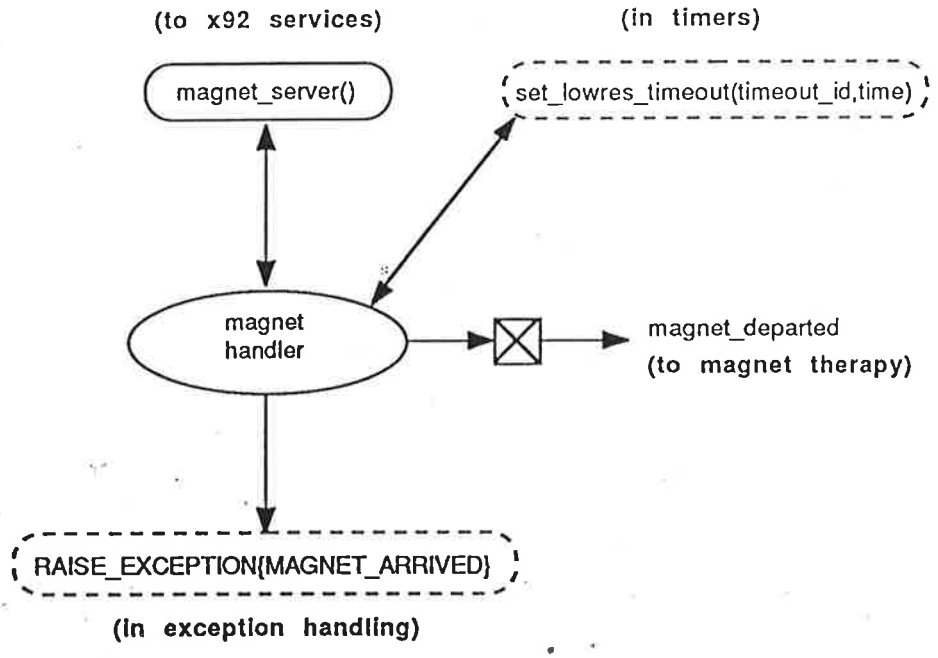

Figure 6.4: Implementation of Magnet Handler

6.10 Pacing therapy

Mission

Pacing therapy controls the delivery of anti-tachycardia pacing therapy to the patient. See also section 2.4.4.

Requirements

1. Set X92 into correct mode for confirmation, unless X92 is already in the correct mode.
2. Allow ATP to be enabled or disabled by telemetry parameters.
3. Inhibit ATP if TCL is less than *minimum TCL for ATP*.
4. Do not deliver more than the *maximum number of trains*.
5. Test **confirmation** before each train.
6. Use **pacing primitives** to deliver a train.
7. Poll the STOP_WHEN_READY exception and abort if it is raised.
8. Abort if **pacing primitives** aborts.
9. Determine when a scan is successful and instruct **pacing primitives** to save the scan value.
10. Send a message to **data logging** after each confirmation, giving the result of confirmation and TCL.
11. Send a message to **data logging** after a reversion, giving the number of trains delivered in this episode before reversion.
12. Switch off **confirmation** when therapy is finished.

Telemetry parameters

Type	Name	Values or Range	Remarks
r/w	max_no		The maximum number of ATP trains to be delivered.
r/w	min_TCL_for_ATP		The minimum value of TCL for which ATP is allowed.
r/w	atp_enable		Whether ATP is allowed for this patient.

Implementation

The implementation of **pacing therapy** is shown schematically in Figure 6.5.

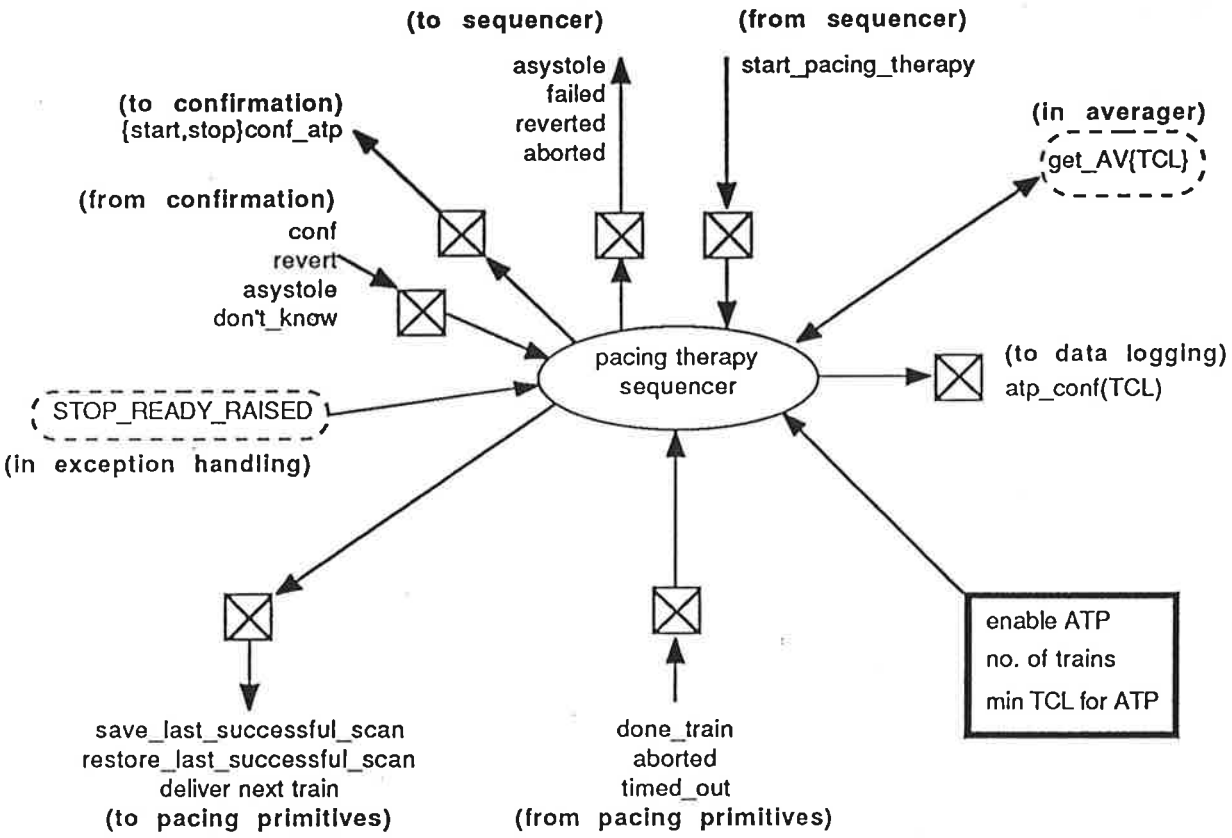

Figure 6.5: Implementation of Pacing Therapy

6.11 Pacing primitives

Mission

Pacing primitives delivers individual pacing trains. See also section 2.4.4.

Requirements

1. Configure the X92 into the correct mode for delivering a pacing train just before the first pulse.
2. Provide a command to start delivery of a train.
3. Synchronise the start of the train to a sensed event.
4. Terminate on the arrival of a STOP_ASAP exception.
5. Scan the pacing trains.
6. Save the values of the last scan.
7. Provide a command to save the last scan values into an internal 'last successful scan' buffer.
8. Provide a command to start the next scan on the last successful scan.
9. Send a message to **pacing therapy** when a train is finished.
10. Send a message to **data logging** after a train is delivered, describing that train (type of description to be determined).
11. Send a message to **data logging** saying how many paces were delivered.
12. Set the dirty word (in **sensing and recent history**) around a pacing train[6].
13. Implement a machine to interpret the specification of a pacing train as described in the Formal Specification and further detailed in *ACS Programmer/Implant Telemetry Protocol.*[7] Note that the specification is a general one that allows at least the following algorithms to be used:
    - PASAR
    - Orthorhythmic PASAR

[6] In detail: from just before the first pace is delivered to just after the last pace is delivered.

[7] This document has yet to be reviewed.

- Auto Decrement
- Auto Increment

14. Deliver each pace to an accuracy of ±5MS, with one pace per train allowed to be ±20MS.[8]
15. Provide a method to stop a pacing train on a sense.
16. Possibly deliver a tachy inducing train.

Telemetry Parameters

Type	Name	Values or Range	Remarks
r/w	N1, N2		Code segment to calculate the number of pulses for this train
r/w	I1, I2		Code segment to calculate initial interval
r/w	F1, F2		Code segment to calculate a factor which gives the relative size of the following intervals
r/w	timeout		Timeout time to allow for waiting for the synchronising sense
r/o	ls-scan		Last succcessful scan value

Implementation

The implementation of **pacing primitives** is shown schematically in Figure 6.6.

6.12 Defib therapy

Mission

Defib therapy controls all defib therapy given to the patient. See also section 2.4.5.

Requirements

1. Allow defib therapy to be started, with a caller-supplied minimum time to first shock in the train (this can be 0 for shock_asap).

[8] This is a suggested target only, and is open to negotiation.

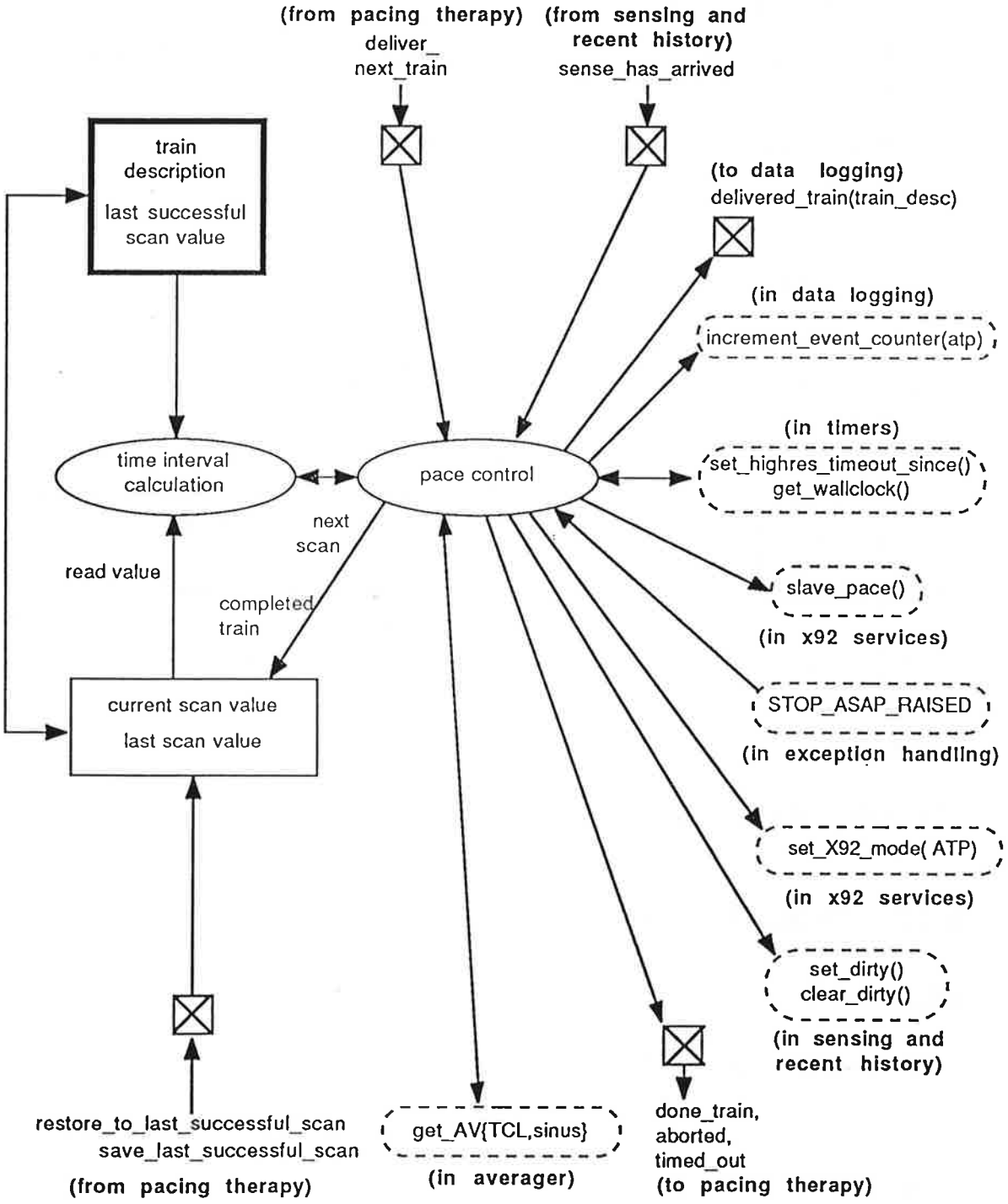

Figure 6.6: Implementation of Pacing Primitives

2. Deliver a train of shocks, the train being defined by telemetry parameters.
3. Inhibit shock delivery whilst TCL is less than *max tcl for defib.*
4. Allow defib therapy to be disabled, ie only continuous confirmation performed.
5. Abort defib therapy when STOP_WHEN_READY exception is raised, otherwise continue applying therapy until **confirmation** returns 'reverted'.
6. Set up the X92 into confirmation defib mode after the first shock is delivered.
7. Start and stop the confirmation process.
8. Supply a unique identifier for each shock in a train. This will ensure that prepare_next_shock(n) followed by deliver_train_shock(n) is the only valid combination of commands accepted by **defib primitives.**
9. Tell **data logging** when the patient reverted, after how many shocks, or if it was spontaneous.
10. Change behaviour when no more shocks are to be delivered (to be defined).

Telemetry parameters

Type	Name	Values or Range	Remarks
r/w	enable_defib	on, off	
r/w	max_TCL_for_defib	?MS	The maximum value of TCL for which defib is allowed.

Implementation

The implementation of **defib therapy** is shown schematically in Figure 6.7.

6.13 Defib primitives

Mission

Defib primitives is responsible for the delivery of individual shocks. See also section 2.4.5.

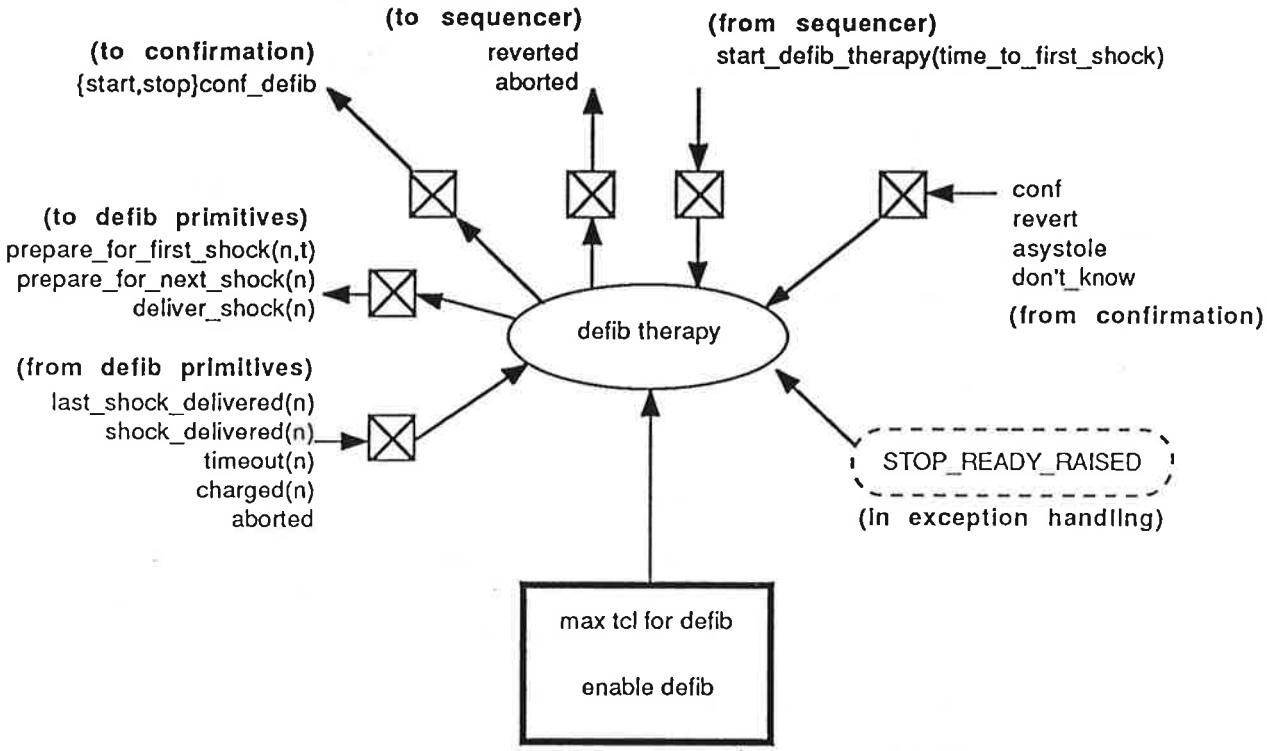

Figure 6.7: Implementation of Defib Therapy

Requirements

1. Provide a telemetry interface to supply a description of voltage-time pairs for a train of shocks.
2. Provide a telemetry parameter to enable/disable shocking.
3. Provide a telemetry parameter 'max TCL for defib'.
4. Calculate how long it takes to give a particular voltage, taking into account voltage required, state of battery and state of capacitors.
5. Predict an average time for charging and allow an extension to a maximum time for charging at which point charging will be terminated and the voltage attained will be used for the shock.
6. Provide a *shock_now* command.
7. Set the X92 and **sensing and recent history** into the correct mode around a shock.
8. Implement a 'shock ARP' after all delivered shocks. In particular, mark sense information as dirty within the shock ARP.

9. Measure the voltage on the capacitors immediately before and after a shock, and send the measurements to data logging.

10. Provide safety features to prevent unintended shocks:
    (a) A defib driver sub-module should be dedicated to driving the hardware safely.

    (b) Two sequential actions are required to give a shock. These actions should not be in a linear piece of code.

    (c) A grammar for requesting shocks.
    For example, the *prepare_for_shock()/deliver_shock()* commands use a unique identifier for the shock position within a train. **Defib primitives** should only allow *prepare_for_shock(n)* followed by *deliver_shock(n)*; any other combination should cause a shutdown. Furthermore, both **defib primitives** and **defib therapy** may calculate n+1 independently.

11. Inform **measurements** when charging starts and stops.

12. Dump charge internally when appropriate and make the HESS safe.

13. Inform **data logging** when a dump has occurred.

14. Respond to a STOP_ASAP exception by doing a dump.

15. Be aware of the limitation of the HESS. e.g. Maximum charge/dump operations allowed per unit time.

16. Provide *prepare_for_max_shock()* and *deliver_max_shock()* commands.

17. Send the following MTEs to **measurements**:
    - charge on
    - charge off
    - shock
    - dump

18. Periodically re-form the capacitors (to be clarified by hardware team).

Telemetry interface

Type	Name	Values or Range	Remarks
r/w	shock n voltage	? MS	Voltage for shock n in a train, where $1 \leq n \leq 7$
r/w	shock n time	? MS	Time for shock n in a train, where $1 \leq n \leq 7$
r/w	shocks enable	on, off	
r/w	shock ARP		Absolute refractory period of shock (to 256 MS resolution)

Implementation

The implementation of **defib primitives** is shown schematically in Figure 6.8.

6.14 Noise therapy

Mission

Noise therapy provides therapy to the patient when noise is present.

Requirements

1. Set the X92 into VOO mode to give noise reversion. The rate of VOO pacing will be determined within **X92 services**.
2. Keep applying detection until noise goes away.
3. Abort if the STOP_WHEN_READY or STOP_ASAP exception is raised.
4. Put the ACS into a low power mode after a certain amount of noise has been detected, to reduce power consumption (to be defined).
5. Continue noise therapy for some time after noise goes away. This hysteresis effect will reduce the possibility of ACS jumping in and out of noise therapy in a noise field with the consequent possibility of false tachy detection between periods of noise therapy.

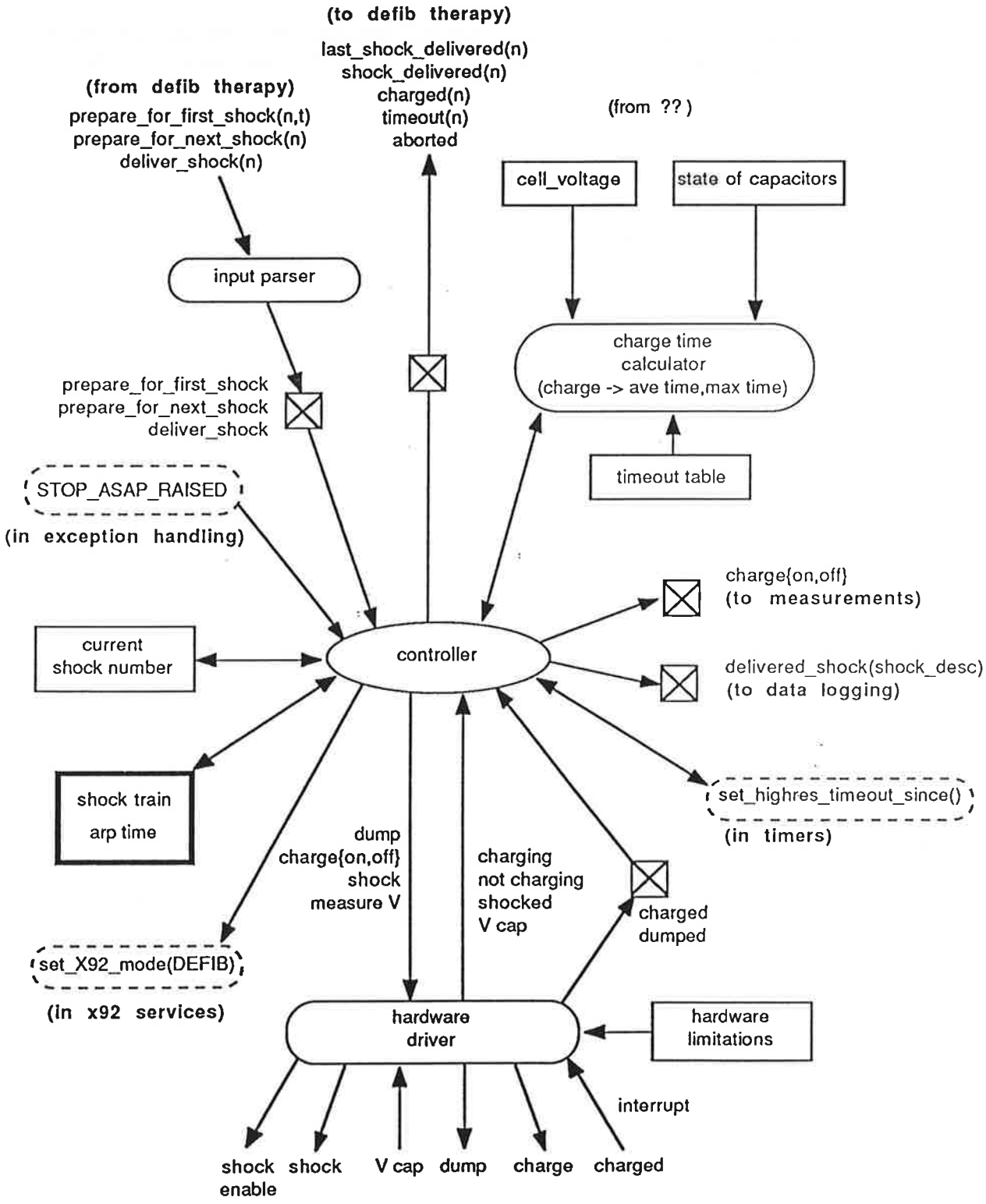

Figure 6.8: Implementation of Defib Primitives

Telemetry parameters

Parameters are required to implement requirements for low power mode (amount of noise and time in low power mode) and continuing noise therapy (hysteresis time).

Implementation

The implementation of **noise therapy** is shown schematically in Figure 6.9.

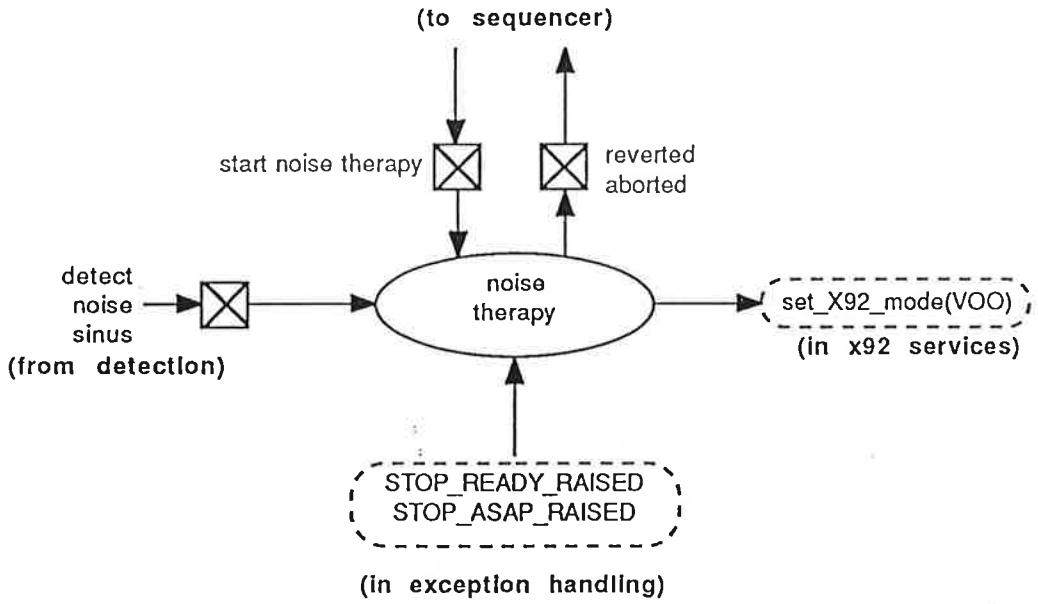

Figure 6.9: Implementation of Noise Therapy

6.15 Magnet therapy

Mission

Magnet therapy provides magnet therapy to the patient. See also section 2.4.2.1.

Requirements

1. Put the X92 into brady support mode.
2. Terminate when *magnet_removed* is received from **magnet handler.**

3. Terminate if the STOP_WHEN_READY or STOP_ASAP exception is raised.

Telemetry parameters

There are no telemetry parameters.

Implementation

The implementation of **magnet therapy** is shown schematically in Figure 6.10.

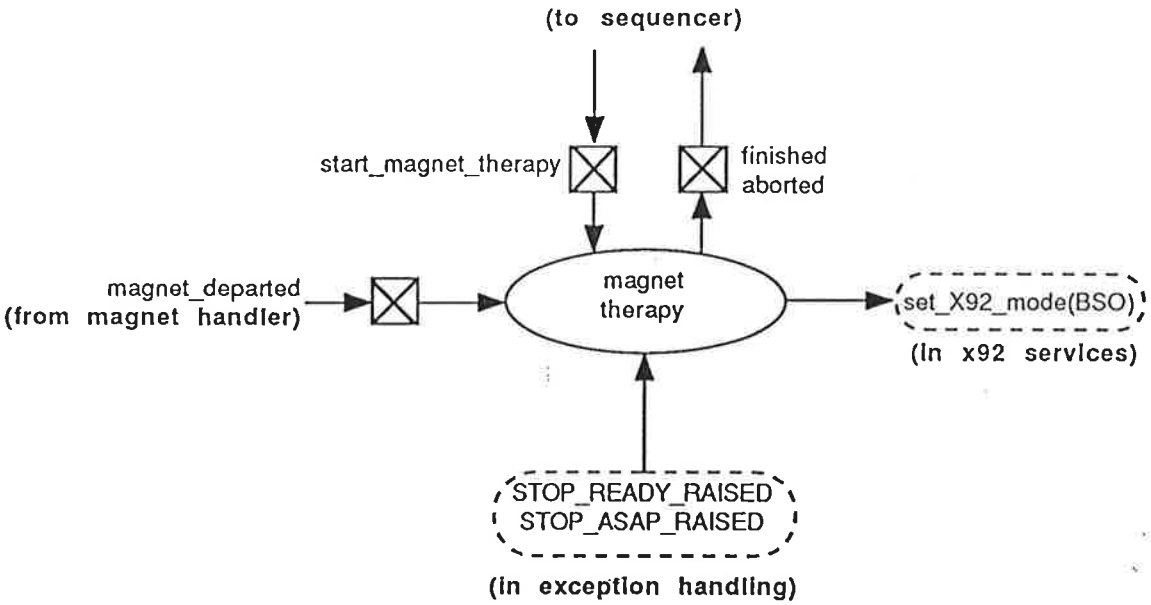

Figure 6.10: Implementation of Magnet Therapy

6.16 Telemetry primitives

Mission

Telemetry primitives handles the logical link protocol. See also section 2.4.7.

Requirements

1. Deal with tele_aborts, tele_writes and tele_reads at any time.

2. Prevent the ACS telemetry hardware being permanently left in 4MS mode after external RTD has finished.
3. Receive commands and pass them on to **telemetry commands**.
4. Pass command results back to the programmer.
5. Implement the logical link protocol.
6. Tell **telemetry commands** when the physical link has closed.
7. Raise exceptions for:
   (a) rude_open_request
   (b) polite_open_request

Telemetry parameters

Type	Name	Values or Range	Remarks
r/w	base_ptr	$0 - 2^{16}$	Address in the μP address space of telemetry address 0
r/w	num_writable	0 – 0xF0x	Number of writeable locations in the data buffer in the telemetry address space

Implementation

The implementation of **telemetry primitives** is shown schematically in Figure 6.11.

6.17 Telemetry commands

Mission

Telemetry commands processes programmer commands sent across the telemetry link. See also section 2.4.7.

Requirements

1. Receive commands and associated data from **telemetry primitives**.
2. Check the validity of the command.

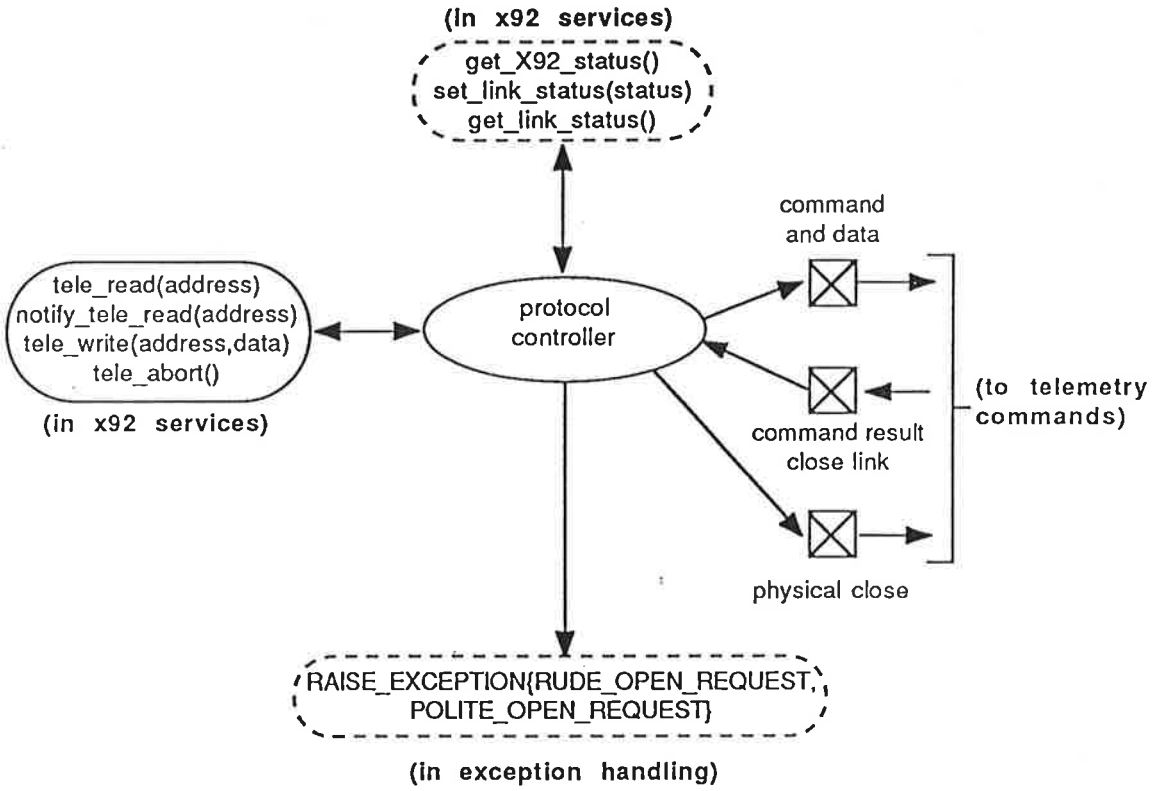

Figure 6.11: Implementation of Telemetry Primitives

3. Check the integrity of the data.
4. Determine what actions have to be done, call other modules to perform the actions and determine the result of the command (note: actions in response to a command should be completed within 100MS).
5. Pass the result of the command back to **telemetry primitives.**
6. Respond to STOP_WHEN_READY and STOP_ASAP exceptions (to be clarified).
7. Implement a grammar defining permissible command sequences.
8. Respond to 'physical close' message from **telemetry primitives.**

Telemetry parameters

There are no telemetry parameters.

Implementation

The implementation of **telemetry commands** is shown schematically in Figure 6.12.

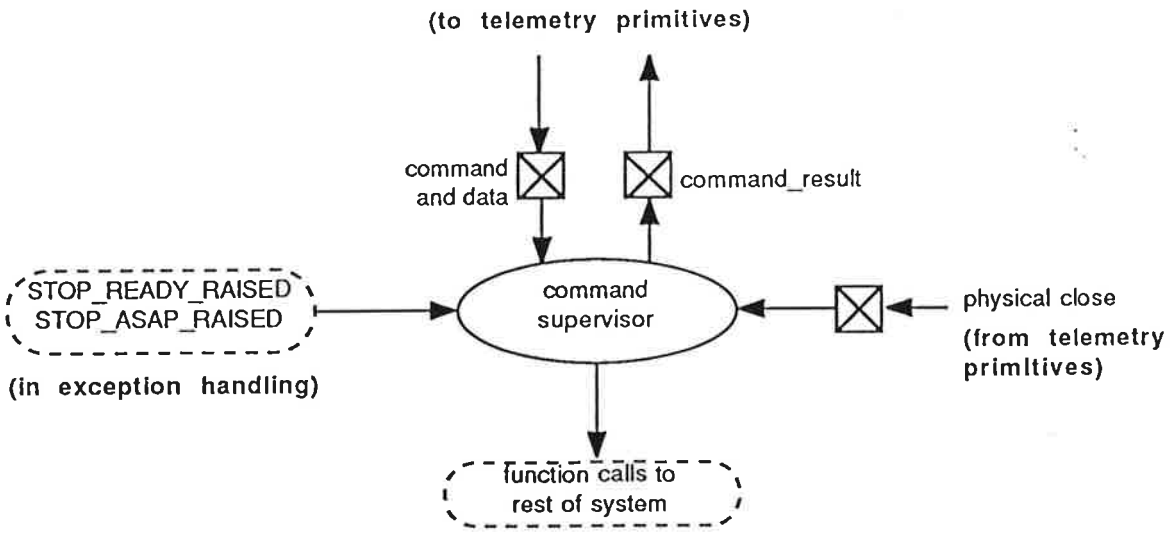

Figure 6.12: Implementation of Telemetry Commands

6.18 Data logging

Mission

Provide a collecting place for all logged data in the system. See also section 2.4.6.

Requirements

1. Provide counters to log system events[9].
2. Provide an interface to other modules to increment each counter.
3. Provide an interface to telemetry to read each counter.
4. Provide a telemetry command to reset any counter.
5. Provide storage for ECG snapshot buffers (ESBs), to be used by measurements.
6. Provide storage management for the ESB space so that oldest snapshots are lost to create space for newer ones.
7. Provide a telemetry interface to upload ESBs.
8. Provide a telemetry command to release ESB space.
9. Provide storage for sense snapshot buffers (SSBs).
10. Provide a command to take an SSB snapshot.
11. Provide storage management for the SSB space so that oldest snapshots are lost to create space for newer ones.
12. Provide a telemetry interface to upload SSBs.
13. Provide a telemetry command to release SSB space.
14. Provide an episode capture buffer (ECB). Episode number to be defined by sequencer.
15. Log the following events for each episode in an ECB generating a timestamp for each event:
    (a) detection including TCL at detection.
    (b) confirmation_ATP including TCL at confirmation.

[9]The counters required are defined in the informal spec.

(c) confirmation_defib including TCL at confirmation.

(d) delivery of an ATP train including details of that train (details to be determined).

(e) delivery of a shock including nominal and delivered energy for that shock.

(f) a number of SSBs. The SSBs logged are controlled by a telemetry parameter which selects SSBs:

i. at detection

ii. at detection and at each confirmation

iii. never

(g) a number of ESBs. The ESBs logged are controlled by a telemetry parameter which selects ESBs:

i. after detection

ii. after detection and after reversion

iii. after detection, at each confirmation, and after reversion

iv. never

16. Provide a telemetry parameter which selects full or shock-priority storage.

full If space runs out, the oldest episode log is overwritten

shock priority If space runs out, the *oldest episode that did not include any shocks* is truncated with only the following information retained:

(a) time stamp

(b) TCL

Telemetry parameters

Type	Name	Values or Range	Remarks

Implementation

The implementation of **data logging** is shown schematically in Figures 6.13 and 6.14.

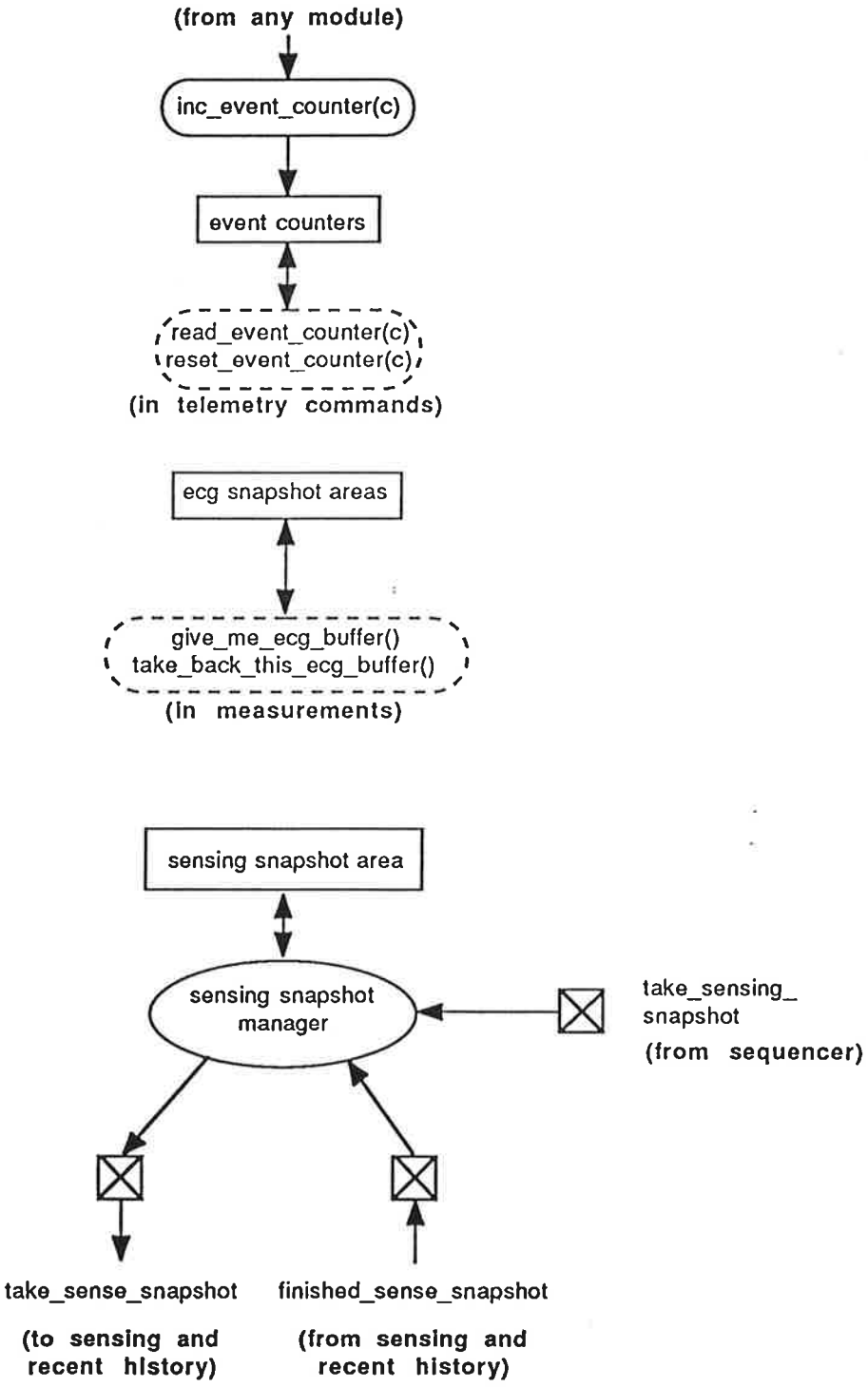

Figure 6.13: Implementation of Data Logging (sheet 1)

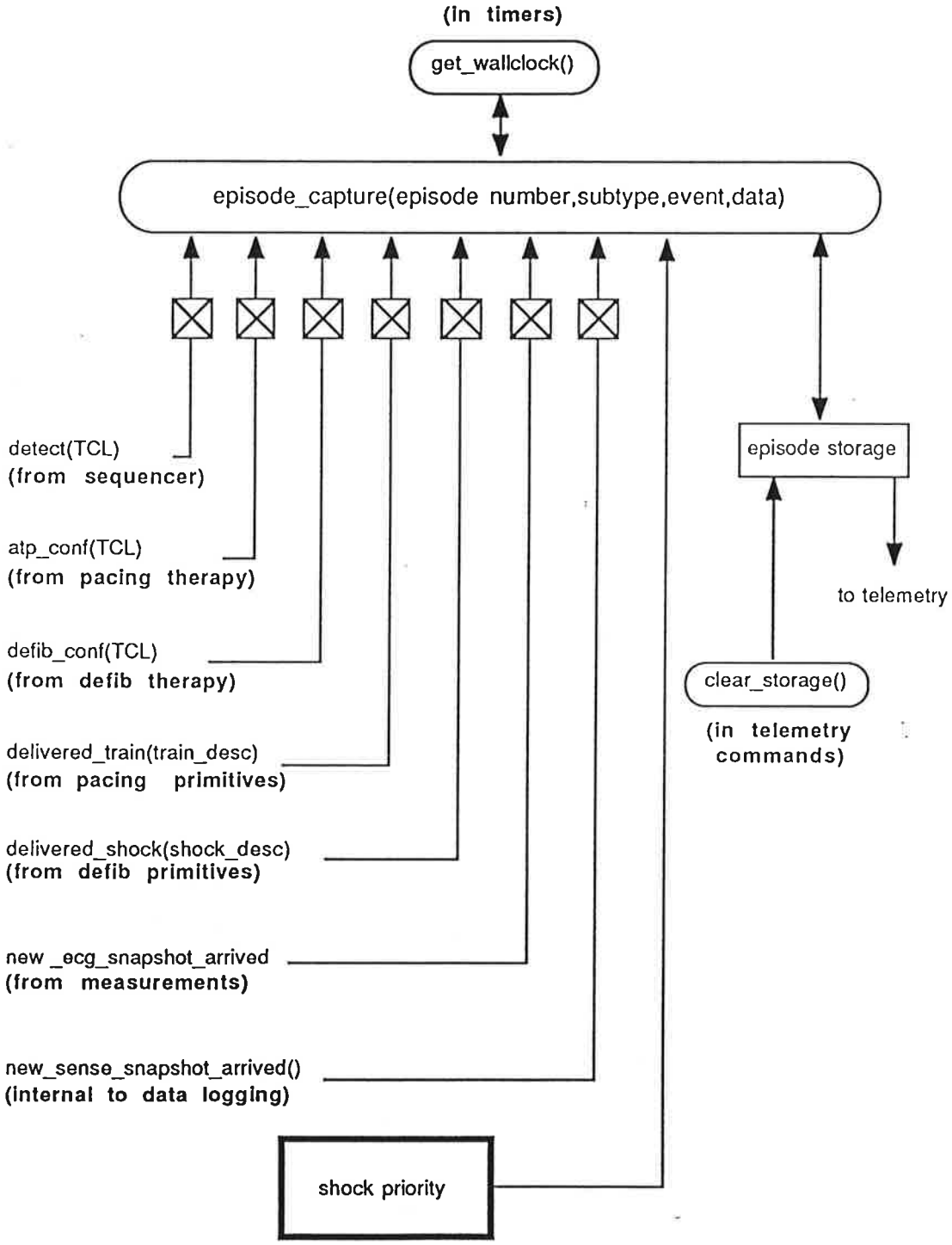

Figure 6.14: Implementation of Data Logging (sheet 2)

6.19 Measurements

Mission

Measurements handles all internal and external measurements in the system.

Requirements

1. Support the following measurements:
   (a) external programmer ECG.
   (b) external programmer MTE.
   (c) internal ECG snapshot.
   (d) internal cell voltage test measurements.
2. Provide a telemetry command to start external ECG.
3. Provide a telemetry command to start external MTE.
4. Provide a telemetry command to start external ECG and MTE.
5. Provide a telemetry command to stop RTD.
6. Provide a telemetry interface for self tests.
7. Provide a place for other modules to send ACS MTEs to, if MTEs are enabled.
8. Send MTEs to X92 services.
9. Provide a command to take an internal ECG snapshot and pass it on to **data logging**.
10. Prevent a snapshot being taken if external RTD is enabled, and pass this information on to **data logging**.
11. Detect corruptions within the ECG snapshot, discard the snapshot and inform **data logging**.
12. Take periodic measurements of the ACS cell voltage.
13. Provide a means of preventing an ACS cell voltage update for a period after an inverter charge.
14. Provide a telemetry interface to read the ACS cell voltage.

Telemetry interface

Parameters

Type	Name	Values or Range	Remarks
r/o	current_cell_voltage	to be defined	
r/o	self_tests_results	to be defined	
command	read_cell_voltage		
command	ext_ecgs_on		
command	ext_mtes_on		
command	ext_ecgs_mtes_on		
command	ext_rtd_off		
command	start_self_tests		

Implementation

The implementation of **measurements** is shown schematically in Figure 6.15.

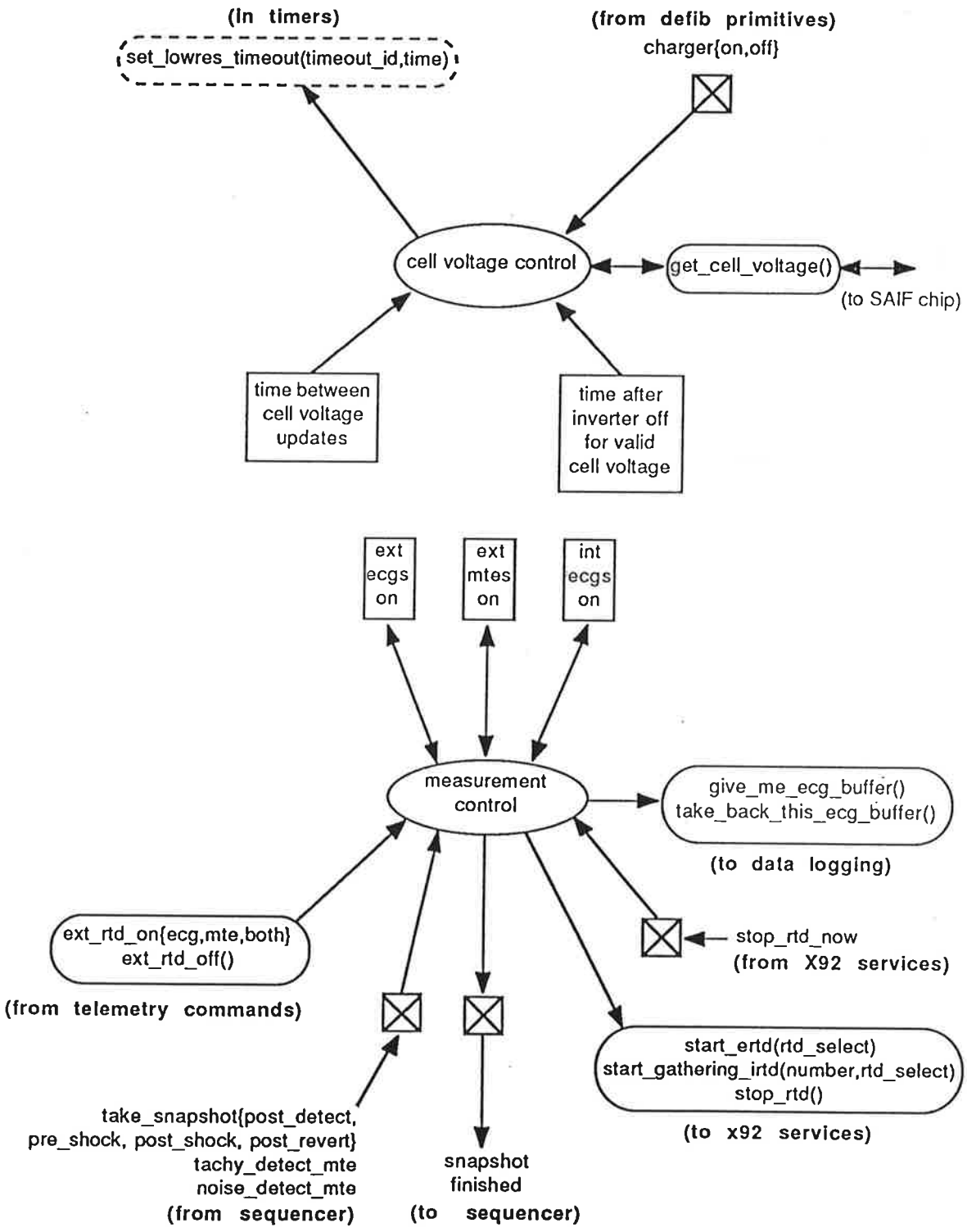

Figure 6.15: Implementation of Measurements

Chapter 7

Details of services modules

This chapter contains a detailed description of each of the services modules. For each module it gives the mission of the module, a detailed list of requirements and the proposed implementation. If the method for implementation is not obvious then it is indicated as "not known". Any unresolved issues will be solved during the design phase of these modules.

7.1 Self testing

Mission

This module is responsible for the self testing of the ACS in the field.

Requirements

1. Perform checksums of code and static data.
2. Provide functions for other system modules to check the consistency of their own data.
3. Check that the μP hardware ports are in their expected states.
4. Perform a battery test.
5. Support inline asserts and warnings.
6. Start self tests at timed intervals.
7. Start self tests either during therapy or when a therapy is completed.
8. Start self tests when requested by telemetry.

Implementation

The implementation of **self testing** is shown schematically in Figure 7.1.

Figure 7.1: Implementation of Self Testing(To be supplied)

7.2 Manufacturing testing

Mission

This module defines the support that will be provided for manufacturing testing.

Requirements

Not known.

Implementation

The implementation of **manufacturing testing** is shown schematically in Figure 7.2.

Figure 7.2: Implementation of Manufacturing Testing(To be supplied)

7.3 Execution tracing

Mission

This module is reponsible for maintaining a trace of the execution of the μP.

Requirements

1. Provide a means of tracing the execution of selected lines of code.
2. Provide a means of accessing the trace information after failure of an ACS.
3. Support the dynamic enabling/disabling of tracing for given lines of code.
4. Provide tracing for development and production systems.
5. Support the uploading of data from a breadboard ACS to the host computer via a UART.
6. Save only the most recent data in a production ACS.
7. Provide a means for displaying trace messages on a terminal for the development ACS.

8. Do not impinge on the operation of time-critical parts of the code.

Implementation

1. Function calls to log information will simply send a message to a trace pipe.
2. Development tracing: a single slow class process will be the server for the pipe. It will use the UART to print out each message that it receives. If the pipe overflow occurs then a message indicating this will be printed.
3. Production tracing: there will be no server for the pipe. Overflow of the pipe will be ignored, thus using the pipe as a circular buffer to contain the most recent traced events.

   A routine will provided for dumping the contents of this pipe to a UART.[1]

The implementation of **execution tracing** is shown schematically in Figure 7.3.

Figure 7.3: Implementation of Execution Tracing(To be supplied)

[1]This can be done by taking a copy of the store of the μP to another machine which has a UART.

7.4 X92 handler

Mission

X92 handler is responsible for the low-level handshaking with the X92 control chip.

Requirements

1. Perform handshaking of the X92 control lines for the following operations:
   (a) IFR transfer.
   (b) Telemetry read, write and abort.
   (c) Real time data transfer.
   (d) μP reads and writes of the X92.
   (e) μP direct control commands.
2. Be time efficient in order to prevent the μP being reset by the X92 and to keep the system duty cycle reasonable.[2]

Implementation

The implementation of **X92 handler** is shown schematically in Figure 7.4.

7.5 X92 services

Mission

X92 services provides a higher level interface to the X92 using **X92 handler** to perform handshaking of the X92 control lines.

Requirements

1. Provide a telemetry interface to X92 registers.
2. Provide a therapy interface to X92 functions.

[2]See section 3.1.

Figure 7.4: Implementation of X92 Handler(To be supplied)

3. Initialise the X92.
4. Perform X92 micro direct control commands (MDCCs).
5. Direct X92 interrupt events to their appropriate handler.
6. Reset the X92 MICE bit (to reset the μP).[3]
7. Maintain the X92 link_status location.
8. Clean up the X92 after a physical link break.
9. Read the X92 magnet status bit.
10. Read the X92 PCM bit.
11. Set the X92 exit code.
12. Switch normal real time data (RTD) on and off, and capture a block of RTD.
13. Transmit implant software generated MTEs to the programmer.

[3] ie jump before we are pushed.

14. Resolve interactions between the programmer and the μP in RTD and analogue measurements.
15. Support version 1 of the X92 control chip.[4]
16. Be time efficient in order to keep the system duty cycle reasonable.[5]

Implementation

The implementation of **X92 services** is shown schematically in Figure 7.5.

Figure 7.5: Implementation of X92 Services(To be supplied)

7.6 Timers

Mission

Timers provides general timing services to the rest of the system.

[4] Until version 2 appears.

[5] See section 3.1.

Requirements

1. Provide an absolute time wallclock with a resolution limited only by the available hardware and a wraparound time greater than the lifetime of the ACS family of devices.
2. Measure intervals between events of a particular class.
3. Enable timeouts to be set *t*MS in the future.
4. Enable timeouts to be set *t*MS after the last occurrence of a particular event.
5. Be time efficient in order to keep the system duty cycle reasonable.[6]

Implementation

The implementation of **timers** is shown schematically in Figure 7.6.

Figure 7.6: Implementation of Timers(To be supplied)

[6]See section 3.1.

7.7 Startup and shutdown

Mission

Startup and shutdown ensures that the system starts up in a controlled manner and shuts down safely when a fault occurs.

Requirements

1. At startup:
   (a) Initialise as much of the hardware as necessary to enable execution of the remainder of the startup code and communication with the programmer.
   (b) Store data about the current state of the system, unless such data is already stored from a previous shutdown.
   (c) Initialise and execute the programmer communications software.
   (d) Prevent the X92 from resetting the μP on startup.
2. At shutdown:
   (a) Store data about the current state of the system unless such data is already stored from startup.
   (b) Make the HESS safe.
   (c) Leave the X92 in an appropriate configuration to perform brady support pacing if required.
   (d) Ensure that the μP remains in a safe state until a programmer takes control.
   (e) Reset the μP if shutdown is called during a shutdown sequence.

Implementation

The implementation of **startup and shutdown** is shown schematically in Figure 7.7.

7.8 Down line loader

Down line loader is used to load code into a RAM based version of the ACS via the telemetry link.

Figure 7.7: Implementation of Startup and Shutdown(To be supplied)

Requirements

1. Write to any location in μP store via the telemetry link.
2. Read any location of μP store.
3. Start the μP executing at an arbitrary address.

Implementation

This will be implemented using the version 3 ROM down line loader or by providing dll-like commands to the telemetry commands module.[7]

The implementation of **down line loader** is shown schematically in Figure 7.8.

7.9 Ports

Ports provides structured access to the μP I/O ports.

[7] **The second option is the preferable one.**

Figure 7.8: Implementation of Down Line Loader(To be supplied)

Requirements

1. Initialise the ports.
2. Change the states of the I/O ports.
3. Prevent any other processes from destroying the values in the port (e.g. because of interrupts).
4. Trace the changes in values for some of the ports.

Implementation

The implementation of ports is shown schematically in Figure 7.9.

7.10 Test rig software

This provides a suitable interface to our testing environment.

Figure 7.9: Implementation of Ports(To be supplied)

Requirements

1. Define how the state of the system will be displayed on the ACS Status Panel. This format will also be used on chart recorders.
2. Provide routines to change this information.
3. Any other tasks required for testing.

Implementation

The implementation of test rig software is shown schematically in Figure 7.10.

7.11 Utilities

Mission

Utilities provides general utility routines for use by other modules in the system.

Figure 7.10: Implementation of Test Rig Software(To be supplied)

Requirements

Implementation

The implementation of **utilities** is shown schematically in Figure 7.11.

Figure 7.11: Implementation of Utilities(To be supplied)