Durham E-Theses

The development and application of ambulatory monitor for measuring weight-bearing during fracture healing

Philip John Aranzulla

How to cite:

Aranzulla, Philip John (1995) *The development and application of ambulatory monitor for measuring weight-bearing during fracture healing.* Masters thesis, Durham University.

Use policy

The Development and Application of an Ambulatory Monitor for Measuring Weight-Bearing during Fracture Healing

Submitted by

Philip John Aranzulla

to the University of Durham

as a thesis for the degree of

Master of Science

in the School of Engineering

September 1995

I certify that all the material in this thesis which is not my own work has been identified and that no

material is included for which a degree has previously been conferred upon me.

P. Aranzulla

1 3 NOV 1995

List of Contents

List of Figures

List of Tables

Abstract

The aim of this study was to measure the weight-bearing during healing in a series of patients with tibial fractures, and to examine how this changed with time post-fracture. Weight-bearing indicates the force through the leg as a percentage of the total body weight. An ambulatory monitoring system was developed, comprising of the monitor and analysis software for a PC. The ambulatory monitor measures the force via pressure transducers attached to the load bearing areas on the underside of the foot to obtain the weight-bearing through the fractured leg. The software was developed in the C programming language by using a PC host executing a cross-compiler, the program then being down-loaded via the serial line to the monitor hardware for execution and testing. Once a basic monitor was operational, the clinical trials commenced, these being conducted at fracture clinic sessions at Middlesbrough General Hospital. Further development work occurred throughout the patient trials which led to increases in the accuracy and consistency of the results obtained.

Results were obtained from 37 patients with tibial fractures, and these all demonstrated that there was a non-linear increase in weight-bearing with time post-fracture. An increase in step duration relative to the step duration of the normal leg also occurred, indicating a gradual change in the gait pattern adopted, tending towards a normal gait pattern with time. A similar pattern was found with the stride length, this indicating a gradual change towards a more normal gait pattern with time. An increase in velocity of gait was also observed over the healing period, suggesting greater confidence in walking as healing progressed.

Such results lead to the hypothesis that a feedback mechanism operates which controls the weight-bearing applied to the fracture depending on the stiffness of the fracture. The clinical relevance of this work is to aid the clinician in fracture healing assessment enabling the prescription of more applicable treatment methods.

<u>Acknowledgements</u>

I wish to thank my supervisor, Dr. J.L. Cunningham for his continued support and encouragement throughout the project, and particularly during the writing of the thesis.

I also gratefully acknowledge Mr. Muckle and Mr. Pye of Middlesbrough General Hospital for their help during the patient trials.

Finally I would like to acknowledge SPARKS who have funded this study.

1. Introduction and Biological Background

Fractures require a stable mechanical environment for healing, and so fractures in long bones are particularly problematic due to the larger moments and forces at the fracture site. Therefore to permit healing, the fracture is often stabilised by use of a fixation method.

Tibial fractures are especially prone to non-union or delayed union (Oni *et al.*, 1988), and so previous clinical practice was to stop the patient weight-bearing on the fractured leg until the healing process was well advanced. However due to increased understanding of the healing process, more recent clinical practice has been to encourage patients to partially weight-bear early to stimulate healing. The aim of this study was to monitor the weight-bearing over the fracture healing period.

In this chapter is discussed the biological aspects of bones, fracture healing and treatment, this leading to the various methods possible for weight-bearing measurement, and concluding with the chosen method.

1.1.   Bone Function and Composition

Although the main purpose and function of individual bones may vary, in general each bone has three basic functions. The first is that when in combination to form a skeleton, the many different bones support the soft tissues and protect the internal organs from damage. They also provide for muscle, tendon and ligament attachments and by acting as levers and struts enable movement. Finally each bone stores various minerals and blood generating cells.

To be able to perform its first and second functions, bones are required to be strong. To minimise energy expenditure in movement, there is also a requirement for them to be as light as possible. The molecular and histological structures of bones are such as to satisfy both these requirements. Therefore the human skeleton, which constitutes less than 20% of the weight of the entire body, can endure high loads such as five times body weight on the bones of the knee joint when running. The equivalent skeletal framework made out of steel would weigh four to five times more than does the bony skeleton.

1.2.   Bone Structure

1.2.1.   General Structure

Examining bones at the gross level results in the discerning of two basic material structures: compact and spongy bone.

Compact bone, otherwise known as cortical bone, is solid, dense bone that is found in the walls of bone shafts and on external bone surfaces. Spongy, cancellous or trabecular bone has a more porous and lightweight honeycomb structure. It is found under protuberances where tendons attach, in vertebral bodies, at the ends of long bones such as the tibia and fibia, in short bones, and sandwiched between flat bones. The traberculae are arranged to withstand the stresses to which they are normally subjected, so that those lying along the lines of force are intersected by others acting as struts and ties (Gray, 1964). The benefit of spongy bone occurring at specific locations in a bone is that its weight is reduced due to its greater porosity. Figure 1.1 shows the difference in appearance between both types of bone, the example given being the proximal end of a tibia.

The porosity of spongy bone is greater than 70% which contrasts with that of cortical bone at 15% (Le Veau, 1992). As bone strength and stiffness varies inversely with increasing porosity, the mechanical properties of these two types of bone differ considerably. However the density and porosity of the bone do not alone dictate its mechanical characteristics, for these can vary as much as two orders of magnitude depending on its location and therefore use (Goldstein *et al.*, 1983). Wolff's law states that the physical characteristics of bone are matched to the routine structural demands placed upon it (Wolff, 1892). Therefore mechanical characteristic variations between bones exist due to a response or remodelling to different loading conditions existing

across a bone according to Wolff's law (Mow *et al.,* 1991). This mechanism is explained in fuller detail later on in this chapter.

During life the outer surface of bones is covered with periosteum, with the inner surface being covered with endosteum. Both are osteogenic tissues meaning that they contain bone forming cells which are numerous and active during youth, but reduced in number and relatively inactive in adulthood. However they can be stimulated to deposit bone when the periosteum is traumatised due to a fracture.

1.2.2. Molecular Structure

Whatever the type of bone, its molecular structure is the same. Bone tissue is a composite material, 90% being made up of the large protein molecule collagen. Each molecule of collagen intertwines with others to form flexible, slightly elastic fibres which are then stiffened by a dense inorganic filling of hydroxyapatite, which is a form of calcium phosphate. This mineral gives bone its hardness and rigidity, and when removed by immersing in acid, the bone becomes a flexible rubber like structure. In contrast, when the collagen is removed from bone, for example by heating, it becomes extremely brittle and crumbles easily. Therefore the composition of these two materials result in a tissue that is strong and rigid.

1.2.3. Histological Structure

Histology is the study of tissues, usually at the microscopic level. Such studies of mammalian bone results in two distinct histological types, immature and mature bone. Immature bone, otherwise known as coarsely bundled or woven bone, develops first and its existence is usually temporary as it is replaced by mature bone. It has a higher

proportion of osteocytes, which are bone cells, and is coarse and fibrous in microscopic appearance with bundles of collagen fibres arranged in a random pattern. Mature or lamellar bone tissue on the other hand has an organised structure due to the repeated addition of uniform lamellae to bone surfaces during appositional growth, this being apparent at two levels. The first is that the inner and outer bone surfaces are encircled by the inner and outer circumferential or primary lamellae, as shown by the second diagram of Figure 1.1. These lamellae's fibres are each oriented in a different direction as shown, so that the bone's strength is increased by being able to endure tensile and compressive forces in various directions. There are also the secondary lamellae which encircle the Haversian canal, each one's fibres again being oriented in a different direction than the next to increase strength.

Compact bone is too dense to be nourished by diffusion from surface blood vessels, therefore Haversian systems with their canals and canaliculi are present throughout the bone, as shown by Figure 1.1. The diagram on the right of this figure shows a cross section of an osteon or Haversian system, the lamellae indicated being called Haversian or secondary lamellae. An examination of each reveals a bed of parallel collagen fibres, with fibres in successive lamellae being oriented in different directions, again strengthening the structure. Through the Haversian canal there passes blood, nymph, and nerve fibres. Haversian canals run longitudinally within the bone, and are connected to each other by Volkmann's canals which are transversely oriented, not being surrounded by concentric secondary lamellae.
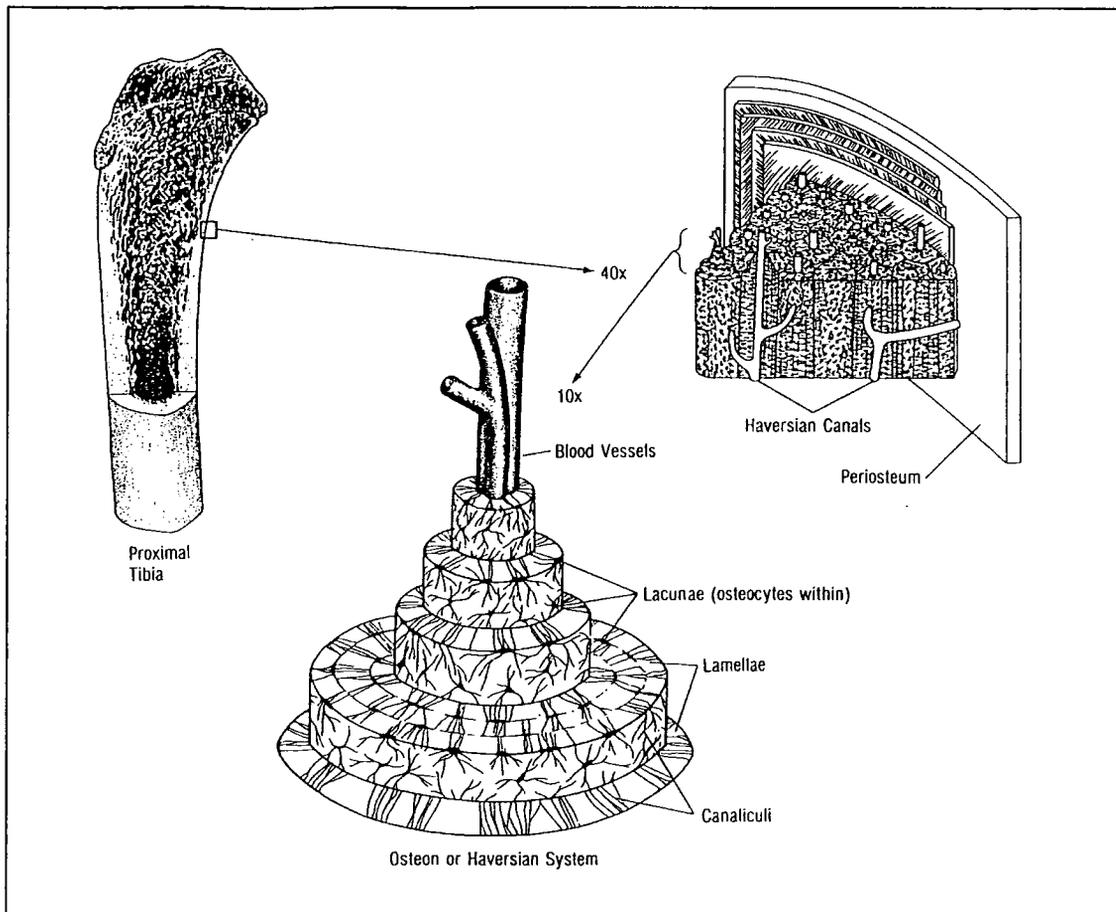
Figure 1.1 - Gross and microscopic structure of bone (White, 1991).

1.2.4.   Bone Cells

There are three major types of bone cells involved in forming and maintaining bone tissue, called osteoblasts, osteocytes, and osteoclasts, as shown in Figure 1.2.

Osteoblasts are bone forming cells which produce many collagen molecules arranged in a matrix, this being called osteoid. Osteoblasts derive from osteoprogenitor cells which are present in the periosteum and also the blood vessels of the Haversian vascular system. A proliferation of these bone forming cells occurs at fracture sites as osteoprogenitor cells divide frequently in such areas.

Once the collagen is calcified, the osteoblast being surrounded by a bony matrix, it is called an osteocyte. Osteocytes' function changes from one of bone formation to one of bone regulation they resorb calcium or phosphate from the surrounding tissues in response to hormonal signals, this process being called osteolysis. The space where each is found is called a lacuna, with long dendritic arms, called canaliculae, acting as communication and nutritional channels.

Osteoclasts have the opposite function to osteoblasts in that they resorb rather than form bone tissue. These are found in hollow depressions in the bone tissue known as Howship's lacunae. At the surface of the cell are apatite crystals and collagen removed, the cell therefore moving through the bony tissue as it is resorbed.
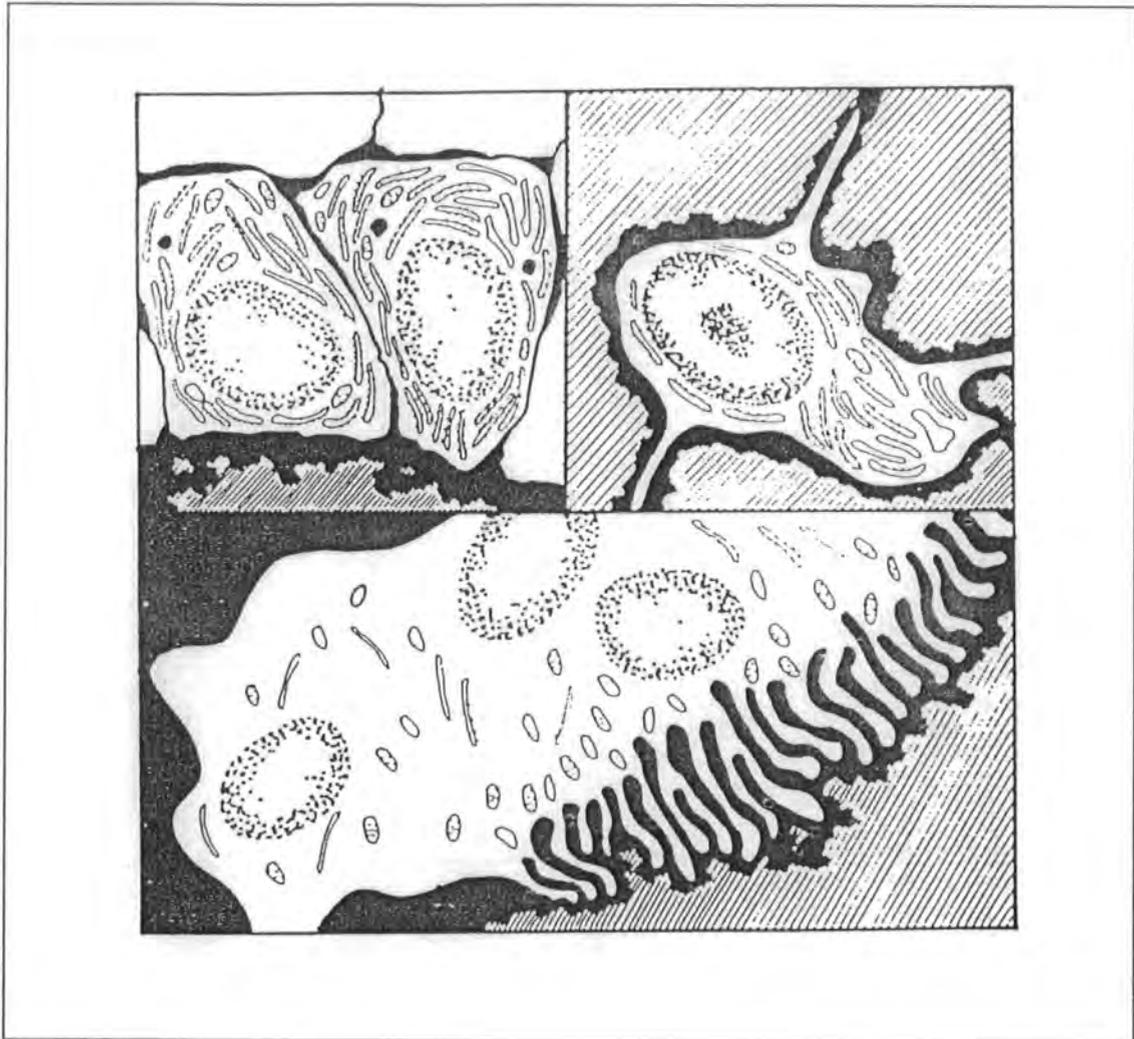
Figure 1.2 - Bone cells, with the diagonally shaded areas being bone. Osteoblasts are shown in the top left diagram. An osteocyte is shown in the top right diagram. Osteoclasts are large cells with many nuclei, part of one being shown in the bottom diagram. (Shipman et al., 1985).

1.3.   Fracture Healing

Fracture healing can be divided into phases, with events described in one phase

persisting into the next. This is shown by Figure 1.3 where the three basic phases

occurring during fracture healing are displayed, with an approximation of the relative

amounts of time for each. There follows explanatory text for each phase, with the

reparative phase being further sub-divided into the cartilaginous phase, and the

mineralisation phase. This phase division clarifies the events occurring during

fracture healing, and have been described over the years in investigative reports and

review articles (Ham, 1974).
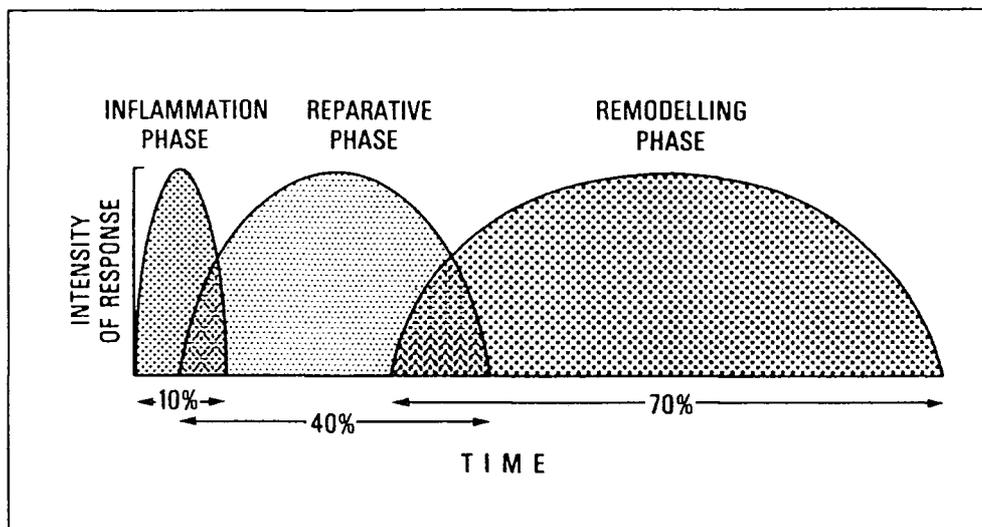
Figure 1.3 - An approximation of the relative amounts of time devoted to the inflammation, reparative, and remodelling phases in fracture healing (Rockwood et al., 1984).

1.3.1.   Inflammatory Phase

Figure 1.4 shows that after a fracture the soft tissue envelope is torn and the numerous

blood vessels crossing the fracture line are ruptured. Therefore an accumulation of

hematoma within the medullary canal occurs, this blood rapidly coagulating to form a

clot. As the blood supply is damaged, the osteocytes are deprived of oxygen and nutrients and so die as far back as the junction of collateral channels. Severely damaged soft tissues may contribute to the necrotic material in the region (Ham, 1974).

The presence of the necrotic material elicits an immediate inflammatory response. Vasodilation occurs with the blood vessels increasing in diameter and an exudate of proteins, plasma and white cells escape into the trauma region. A soft tissue cuff forms around the fracture site which increases both the cross-sectional area and the moment of inertia of bone, thereby greatly increasing the stiffness of the fracture. Prostaglandins are also released, these being associated with bone resorption, bone collagen synthesis and general cleanup of the fracture (Pan *et al.*, 1992).
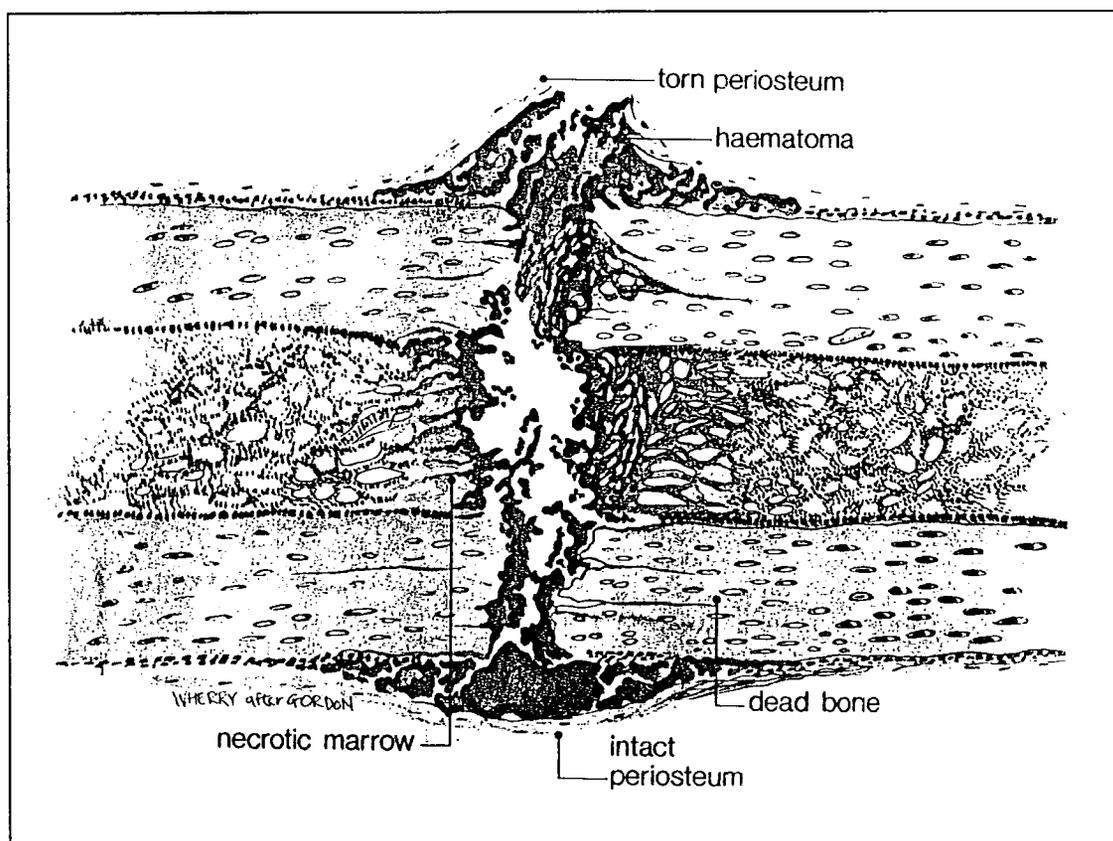
Figure 1.4 - The initial events involved in fracture healing of long bone. The periosteum is torn opposite the point of impact, and in many instances is intact on the other side. There is an accumulation of hematoma beneath the periosteum and between the fracture ends. There is necrotic marrow and dead bone close to the fracture line.

1.3.2.   Cartilaginous Phase

This phase is otherwise known as the proliferative or soft callus phase. At this stage the microenvironment about the fracture is acidic, and during the repair process the pH level returns to neutral and then slightly alkaline (Heppenstall, 1980). Electronegativity is also found in the region, and unlike currents measured in intact bones, is not generated by stress. This degree of electronegativity slowly diminishes until the fracture is united (Rockwood *et al.*, 1984). Both these factors are stimuli for cellular activity aimed at fracture repair.

Repair is indivisibly linked with the ingress of capillary buds into the hematoma (Rockwood *et al.*, 1984), these first appearing from the periosteal vessels with the nutrient medullary artery becoming more important later in the process. The periosteum is usually torn at the fracture site which stimulates its osteogenic layer (White, 1991) and so many new active bone cells are found in this area during fracture healing (McKibbin, 1978), having ingress via these capillary buds. These cells differentiate into fibroblasts, chondroblasts and osteoblasts, depending on the local requirements (Heppenstall, 1980). The callus tissue shown in Figure 1.5 is formed by the mesenchymal cells which produce fibrous tissue, cartilage and osteoid. This leads to a gradual increase in fracture site stability, although not being related to the radiographic size of callus formed (Panjabi *et al.*, 1985), with medullary callus being formed later. Bone or cartilage is formed according to the oxygen tension; cartilage being formed at greater distances from the blood supply where oxygen tensions are fairly low. This cartilage is eventually resorbed with bone taking its place. Bone resorption also occurs at the fracture site for the removal of the necrotic bone fracture ends. These must be removed for new cartilage and bone to form in its place.
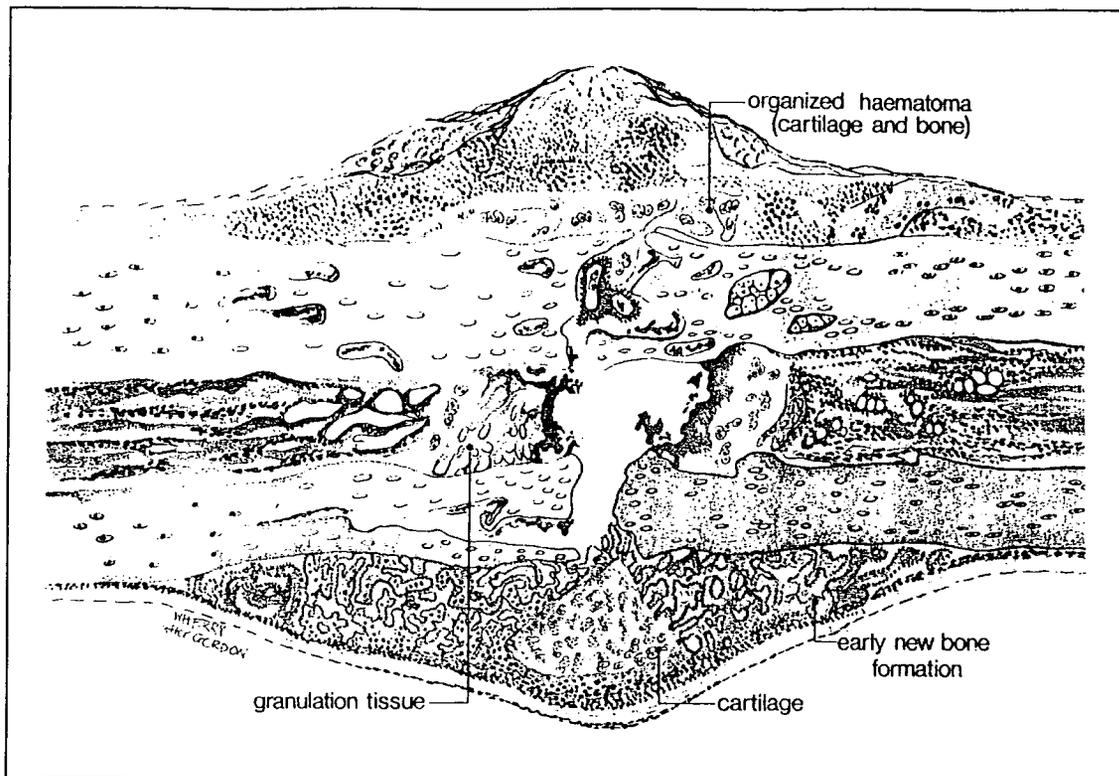
Figure 1.5 - Early repair. There is organisation of the hematoma, early primary new bone formation in subperiosteal regions, and cartilage formation in other areas (Rockwood et al., 1984).

1.3.3.   Mineralisation Phase

This stage is also known as the hard callus and bony phase. It begins at 3 to 4 weeks post-fracture and continues until new bone unites the bone fragments. This varies in time according to the type of bone and fracture, but is in the region of 3 to 4 months post-fracture for long bones in adults. There is an accumulation of calcium hydroxyapatite crystals which occurs for the mineralisation of the collagen. The increase in fracture strength and stiffness seems to be related to the amount of new bone connecting the fracture fragments (Black *et al.*, 1984).

1.3.4.   Remodelling Phase

As explained in Section 1.2.4, the function of osteoclasts is to resorb bone. The remodelling of the bone occurs by cutting cones made up of osteoclasts, these being followed by osteoblasts which deposit collagen matrix. These filling cones are tapered and extend a further distance longitudinally than the cutting cones due to the greater time required for collagen deposition. The cutting cones can advance a distance of 50 to 60 microns every 24 hours (Heppenstall, 1980).

These cones gradually resorb the woven bone of the callus, replacing it with the Haversian bone that was present prior to the fracture (Figure 1.6). These new struts of bone are deposited along the lines of force, the control mechanism being thought to be electrical (Rockwood *et al.*, 1984). Bone is known to be a piezoelectric material so that when subjected to stress electropositivity occurs on the convex surface and electronegativity on the concave surface. Circumstantial evidence indicates that regions of electropositivity are associated with osteoclastic activity and regions of electronegativity with osteoblastic activity (Bourne, 1971). Therefore Wolff's Law is explainable in terms of electrical activity which has a direct effect on cellular behaviour causing the bone to be altered according to the function demanded of it.

Cortical bone heals more slowly than cancellous bone due to the greater amount of bony tissue required, its more regular structure which requires more remodelling, and also because the marrow around the cancellous bone provides a source of osteoblasts local to the area of bone deposition. With favourable conditions cancellous bone may be united after just 4 weeks (Radin, 1987). However cortical bone usually requires about 8 to 12 weeks to heal (Figure 1.7).
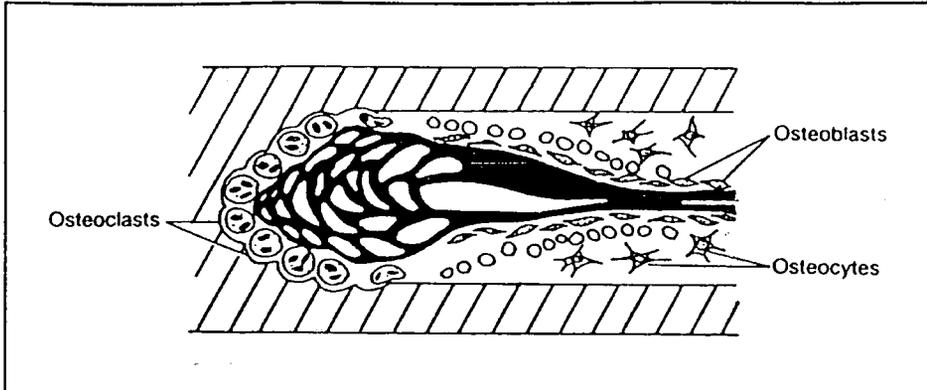
Figure 1.6 - The schematic cutting cone is moving from right to left through the bone. At the tip osteoclasts resorb bone; osteoblasts deposit new bone, are engulfed by the matrix they form, and so become osteocytes. New osteoblasts are produced from the capillary walls as the cutting cone moves through the bone tissue (Radin, 1987).
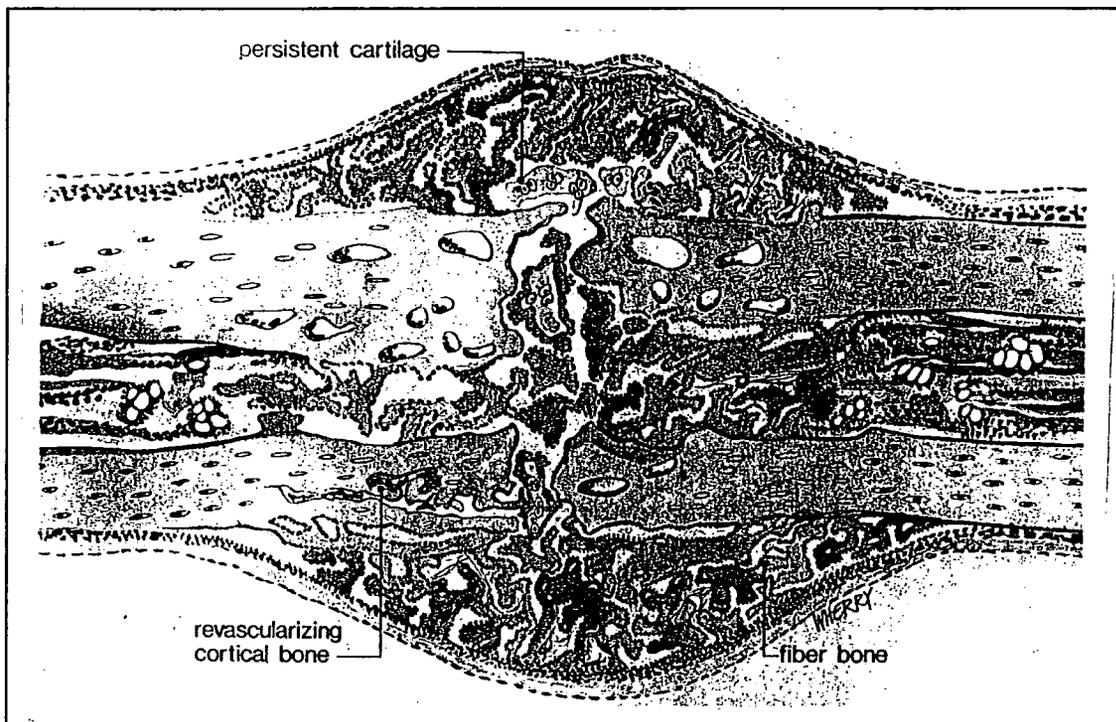
Figure 1.7 - At a later stage in the repair, early immature fibre bone is bridging the fracture gap. Persistent cartilage is seen at points most distant from ingrowing capillary buds. In many instances, these are surrounded by young new bone (Rockwood et al., 1984).

1.4.   Factors Affecting the Speed of Fracture Healing

Factors which affect the rate of fracture healing can be conveniently sub-divided under two headings; factors which are local, and others which are systemic.

1.4.1.   Local Factors

The degree of immobilisation with the amount of soft tissue trauma are probably of paramount importance to fracture healing, inadequate immobilisation leading to delayed union or non-union. This is probably because the initial fibrin scaffolding which is the first step of fracture repair is disrupted, causing the bony bridge of the external callus not to form properly. Fractures involving soft tissue trauma show retarded healing due to a decrease of differentiation of the mesenchymal cells and in their total number.

Factors which contributed most to delayed or non-union seem to be initial displacement, comminution, associated soft tissue injury and infection (Nicoll, 1964). Rockwood indicates that the fracture should be completely immobilised during the inflammatory stage so that the vascular supply could be reinstated (Rockwood *et al.*, 1984).

1.4.2.   Systemic Factors

Fractures in young people heal more rapidly than those of adults, with the rapid remodelling that accompanies growth also allowing correction of a greater degree of deformity in the young. The reasons for this might be given by the results of experimental work with animals which showed that when young there is a more rapid differentiation of cells from the mesenchymal pool (Rockwood *et al.*, 1984).

As has been indicated before, electronegativity in bone has been linked with osteoblastic activity and so bone formation. It has been hypothesised that the application of electric currents directly to a human fracture or via the use of non-invasive electromagnets might therefore increase the rate of healing. A double blind patient trial to study the effect of pulsed electromagnetic fields on 45 tibial fractures with delayed union was carried out, with the conclusions being that significant improvements in the healing of patients occurred with active electromagnetic stimulation (Sharrard, 1990). However some reports believe the result of improved healing times are inconclusive (McKibbin, 1978).

It has been found that the healing rate can be increased by allowing small movements of the fracture site to occur. For example cyclic loading producing a small (≤ 1 mm) amount of micromovement of a fracture was applied and was found to improve healing (Panjabi *et al.*, 1977; and Goodship and Kenwright, 1985). It is probable that bone formation is stimulated by forces acting across the fracture site, as the lack of weight-bearing has been shown to decrease the amount of woven bone that is formed (Meadows *et al.*, 1990). This hypothesis is in agreement with Klein-Nulend *et al.* (1986), who found that compressive forces at the fracture sites in foetal mice stimulates rapid mineralisation of uncalcified matrix.

1.5.   Post-trauma Osteoporosis

Osteoporosis is where there is a loss of bone, and changes in the cancellous pattern (Oxnard, 1993), so reducing the bone strength. During many rat experiments conducted in the 1950s, a rapid increase in bone mineral content following trauma occurred at the fracture site. However a decrease in the bone mineral quantity was noticed in the rest of the limb when compared with its opposite counter-part. This difference seemed to last for a longer period of time than that required for healing (Ulivieri *et al.*, 1990). This was found to be due to an increased rate of bone resorption (Wand *et al.*, 1992) as large resorption cavities were visible in the cortical bone (Young *et al.*, 1983).

Paavolainen and associates studied the healing of experimental fractures in rabbit tibiofibular bones treated by plates (Paavolainen *et al.*, 1979). During the first 9 weeks there was a progressive improvement in torsional strength reflecting the advancement of the union. From 9 to 24 weeks the torque capacity and energy absorption decreased while the torsional rigidity reached a steady state, concluding that after healing the continued presence of the implant has an adverse affect on the strength of cortical bone. This was verified by histological studies where after 9 weeks there was a rapid excavation and breakdown of the cortical wall, its porosity increasing from 9% to 37.5%. The same has been seen in human fractures, with loss of bone mineral being shown to have no correlation with the treatment method (Sarangi *et al.*, 1993). This effect is of long duration, for Nilsson (1966) found that this difference in porosity between the limbs took 6 to 7 years in males and 15 years in females to disappear. Post-traumatic osteoporosis can therefore lead to a weakness of the bone for many years which gives it a higher probability of refracture (Wand *et al.*, 1992).

A main factor for osteoblastic activity has been shown experimentally to be the amount of physical activity (Wand *et al.*, 1992). It is therefore thought that the main cause for

post-traumatic osteoporosis is a reduced functional loading of the limb (Le Veau, 1992 and Whalen *et al.*, 1988), with the increased bone resorption in response to fracture probably being a contributory factor (Sarangi *et al.*, 1993).

Therefore the previous rationale in treating fractures by non weight-bearing until the healing was far advanced, has been discarded in favour of early partial weight-bearing. This avoids post-trauma osteoporosis and encourages healing by producing forces which result with naturally induced micromovement at the fracture site.

1.6. Fracture Treatment and Technique

The main causes for tibial fractures involve direct violent impact such as motorcycle and car accidents or indirect injuries such as from sport accidents and falls (Rockwood *et al.*, 1984). Normally a high energy impact results in greater soft tissue damage, skin loss, bone displacement and comminution, with the fracture often being transverse. In contrast, a low energy impact usually results in an oblique or spiral fracture.

As an important permissive factor of fracture healing is the degree of immobilisation, so fixation is required to maintain alignment and give stability during union of the bone. Various fixation methods are available, with the particular method chosen being dependant on the extent of damage to the soft tissue and bone, and also to the pre-disposition of the surgeon preferring one method to another.

1.6.1. Bracing and Casts

As a plaster cast provides good control of angulation but poor control of rotation and length (Latta *et al.*, 1991), it is normally applied after reduction from the knee to the ankle for simple low impact fractures. Braces are easily adjustable to compensate for the changing leg volume so that stability can be maintained, being used for fractures with minimal initial shortening and soft tissue damage. Forty years ago the routine treatment for an uncomplicated tibial fracture was a closed reduction followed by a non-weight bearing long leg cast which was worn for 10 weeks before another cast was applied allowing partial weight-bearing. Since then however, early weight-bearing is encouraged starting at between 10 to 16 days post-fracture, as this seems to reduce muscle atrophy and tissue edema and also shorten the post cast rehabilitation time (Rockwood *et al.*, 1984). This is probably due to the stimulating of osteoblastic activity so decreasing of disuse osteoporosis.

1.6.2.   Intramedullary Nailing

Internal fixation is the general term for both intramedullary fixation using an intramedullary nail, and extra-medullary fixation by using plates and screws. Intramedullary fixation is good for short oblique fractures where a large displacement has occurred, and is also often used in comminuted fractures (Rockwood *et al.*, 1984). This is because the use of intramedullary nails results with minimal interfragmentory movement as the nail takes most of the bending, torsion and compression loads applied, so that direct healing of the cortical bone by remodelling occurs. However if a relatively flexible nail is used, some secondary healing may occur with the presence of external callus. The main disadvantage of this treatment method is that it is surgically traumatic and has a relatively high probability of infections, non-union and refracture (Gautier *et al.*, 1992). When the nail requires that the bone be reamed, up to 70% of the cortical blood supply can be disrupted (Whittle *et al.*, 1992) so slowing the healing rate. There is also a risk of mechanical failure of either the nail or the fixation screws due to the high loading they sustain (Latta *et al.*, 1991).

1.6.3.   Plating

Plates are often applied to segmental and intra-articular fractures involving the tibial shaft and knee or ankle joint (Rockwood *et al.*, 1984). Although the plate is not able to resist the high bending moments and rotations which can be borne by an intramedullary nail, it does provide very rigid fixation due to the fracture site being compressed (Gautier *et al.*, 1992). As with an intramedullary nail, the strength and rigidity obtained with a plate is sufficient to enable immediate early limb function. The main problem observed with plate fixation is the devitalisation of adjacent tissue,

subsequent skin breakdown and wound sepsis (Rockwood *et al.*, 1984). Also there is

an immediate alteration of stresses in the bone from those to which the bone is

accustomed, leading to possible stress fractures at the junction of the plate with the

bone. Temporary osteopenia is caused by having stress protection under the plate, so

that for over 6 months post-fracture the overall bone strength is reduced (Latta *et al.*,

1991). Care must be taken in weight-bearing shortly after removal of the plate, for

refracture of the tibia may occur because of disuse osteoporosis (Rockwood *et al.*,

1984).

1.6.4.   External Fixation

External fixation seems most useful in instances involving severe soft tissue wounds. It

reduces the requirement to dissect soft tissue adjacent to the fracture site and may also

be applied rapidly. An external fixator can also be adjusted to satisfy a particular

treatment course, allowing very rigid fixation or more flexible fixation so inducing

micromovement. The normal cyclical mechanical loading and strain in a tibia is

disrupted if a very rigid fixator is applied, so by introducing micromovement improved

osteogenesis at the fracture site may be observed (Goodship *et al.*, 1985, Egger *et al.*,

1993). Circular frames, whilst difficult to apply, have the advantages of resisting

rotary and angulatory deformation whilst still allowing axial deflection which

theoretically improves fracture healing. Problems encountered with external fixators

involve the infection of the pin-tracts which can lead to loosening and decreased

stability (Latta *et al.*, 1991).

1.7.   Measuring Weight-Bearing

All these treatment methods allow the patient some degree of freedom of movement as the fracture is given stability and stiffness due to the fixation device used. As noted previously, the patient being allowed to walk on the fractured limb also has positive effects on the healing of the fracture, for osteoblastic activity is stimulated by the amount of physical activity (Wand *et al.*, 1992). It has also been noted that allowing micromovement at the fracture site has been shown experimentally to favour healing (Goodship and Kenwright, 1985). Rather than directly inducing this micromovement at the fracture site as did Kenwright *et al* (1991) via pneumatic pump attached to a sprung external fixator, the patients monitored during this study have been encouraged to weight-bear early, with the assumption that weight-bearing on fractured limb will naturally induce micromovement at the fracture site because the fixation device can never be infinitely stiff.

The basic feedback mechanism which ensures that the patient does not transmit too much weight through the fractured limb during fracture healing is pain (Dehne, 1980). This occurs via pain receptors at the fracture ends, which indicate pain when these ends move against each other, or more often discomfort when this movement is small. If the fracture is in an early stage of healing, its stiffness is less, and so the limb is more unstable than at a later period in the healing process.

Therefore measuring the amount of weight-bearing of a patient should give an increase with time as the fracture stiffness increases due to the fracture healing. The lack of change in weight-bearing over time might indicate complications in the fracture healing process, or the patient not weight-bearing as requested due to the gait pattern developed or because of laziness. Both of these are of clinical interest; the former being verified by a radiograph and perhaps requiring surgical intervention, the latter

being of interest due to the prospect of a longer time for healing and subsequent

rehabilitation if the patient continues in the same manner.

1.7.1. Weight-Bearing Measurement Methods

To measure weight-bearing in this study, the method of pressure sensing under the foot

has been investigated. Another more limited and less accurate method of assessing

weight-bearing might be to measure the micromovement induced at the fracture site

(via sensors on an external fixator for example, as did Richardson *et al.*, 1992) and by

estimating the combined fracture and fixator stiffness, calculating the weight-bearing.

However estimating the current fracture stiffness is rather inaccurate as the fracture

stiffness increases over time due to healing. Also this method is only feasible using an

external fixator as the treatment method. Therefore for accuracy and flexibility of

treatment, an attempt was made to measure the weight-bearing directly.

Lord *et al.*, (1986) reviewed a number of systems which have been devised in an

attempt to identify high pressure areas underneath the foot which is of particular

interest for conditions where pressure may be excessive, such as diabetic neuropathy

and rheumatoid arthritis. Most foot pressure measurement systems are floor mounted,

it being more difficult to measure pressure beneath the foot inside the shoe.

1.7.1.1. Floor Mounted Systems

Floor mounted systems have the benefit of measuring over the whole area of the foot.

The main disadvantage of their use is that the patient can normally only take one step

on the measuring area. This leads to the patient 'aiming' the foot for the measuring

area when walking up to it, so increasing tension and altering the gait pattern, probably

leading to a reading for weight-bearing on that step being different than that during the

patient's normal gait (Whittle, 1991). A number of floor mounted systems have been used up to date some of which are detailed below.

Simple systems giving coarse readings of pressure which can be converted to force by multiplying by the area, include the Harris mat which is made of thin rubber whose upper surface consist of a series of ridges of different heights. This surface of the mat is coated with ink, paper is put on top of this mat, and the patient is asked to walk over the mat. The highest ridges compress under light load, with the lower ones requiring progressively greater pressures, therefore making the transfer of ink to the paper greater in areas of highest pressure. Other similar schemes include using pressure sensitive film instead of the paper and ink. The Pedobarograph uses an elastic mat laid over an edge-lit glass plate which, when the mat is compressed due to load, loses its reflectivity so becoming darker, this providing a quantitative measurement when recorded by a camera. Load cells have also been used which are placed as an array and walked on. Each measures the vertical force beneath a particular area of the foot, so when added together result in the weight-bearing. However the most accurate reading is obtained by using a force platform or force plate which measures the ground reaction force as a subject walks on it.

1.7.1.2.   In-Shoe Devices

The advantage of using an in-shoe measuring system is that measurements can be taken for each step and so the patient is able to relax into their normal gait pattern, this resulting in more accurate measurements. Also by recording each step taken, variations in the gait whilst walking can be quantitatively measured by observing the changes in the weight-bearing value. However there are difficulties in obtaining accurate measurements due to the curvature of the surface of the sole of the foot, a lack of space for transducers, and the requirement for a large number of wires from inside the shoe to the measuring equipment. For these reasons, such systems usually

measure pressure only in selected areas of the foot, contrasting with floor mounted systems which measure over the whole area of the foot (Whittle, 1991).

This study has sought to develop a weight-bearing measuring system that patients can use whatever the fixation method used for the fracture. An in-shoe measuring system has been devised and developed for this purpose, this method being chosen for its capability of monitoring all steps taken during walking, so that a more accurate average value can be gained for the weight-bearing with the standard deviation showing the variability of the gait. For ease of use and accuracy of data, the equipment had to be portable and small so that the patient would not be encumbered by it and so could walk using their normal gait. The aim of using such a system was to discover whether weight-bearing increased over time and whether any differences were apparent according to the treatment method employed.

1.8.   History of Ambulatory Monitoring

The requirement for ambulatory monitoring was first envisaged by Dr. Norman J.

Holter who was concerned with monitoring the heart. Certain heart conditions

occurred for a small period of time with their effects lasting for the rest of the person's

life, with these conditions being undetectable in an isolated laboratory (Meldrum,

1992). Holter wrote that more physical freedom was desirable to study the heart under

realistic conditions of daily life (Holter, 1961) and also that significant

electrocardiographic (ECG) changes might occur during the normal active day of a

clinically normal individual (Holter, 1957). Therefore there was a need for ambulatory

monitoring over an extended period of time.

For some time telephone ECG transmission occurred. The patient was confined at

home, and the ambulatory monitor intermittently transmitted data of the patient's heart

to a receiver linked to the telephone which in turn transmitted the data along the

telephone line to the laboratory for analysis (Pratt *et al.*, 1988). Later generations

stored the data in memory so that monitoring could occur at any time and the data

transmitted via the telephone when the line became accessible. Other systems used a

cassette tape to store continuously monitored data which was later analysed at the

laboratory using a computerised scanning system (Pratt *et al.*, 1988).

With the advances in microprocessor technology, recording has moved from analogue

to digital recording. The advantages are of speed and ease of data transfer and

evaluation, without requiring an expensive piece of equipment. Also calculations can

be performed during the recording period, such as discarding unessential data points

(Pfister *et al.*, 1989). Using this feature, microprocessor based ambulatory monitors

are able to monitor over extended periods of time without their memory being

exhausted, for only data of interest is stored (Besag *et al.*, 1989). The storage method

for such monitors is normally solid-state RAM technology, which having no

mechanical 'moving parts' means is more robust and smaller, resulting in a monitor which is more compact and lighter, and so less obtrusive to the patient.

Attempts have been made at real-time automatic ECG recording and processing by the monitor, with only the results being stored in memory. There have however been concerns over the sensitivity and specificity of the results in relation to artefacts and frequencies of ventricular arrhythmias, especially with complex and repetitive forms (Kennedy *et al.*, 1987). However ambulatory monitors which perform real-time processing of data offer the ideal solution for applications requiring simpler analysis of data for the extraction of results.

Since this application requires fairly simple data analysis, which will be explained in the next Chapter, this can be performed in real-time with the results only being stored in memory. Therefore due to benefits associated with ready access to the results and with the solid-state technology involved, this type of ambulatory monitor was selected for design and development.

2.   The Monitoring System

This chapter details the monitoring system, which comprises of the hardware and software of the ambulatory monitor, its interface for communication with the PC, and the PC's analysis and file manipulation program for the storage and display of the calculated results.

2.1.   Hardware of the Ambulatory Monitor

This section describes the hardware and other components used to make up the ambulatory monitor. Section 2.1.1 deals with the internal contents of the monitor which give it its functionality. Sections 2.1.2 describe the extra components which are necessary to provide its user interface, and also details the monitor's housing. Finally, Sections 2.1.3 detail the RS-485 to RS-232 interface.

2.1.1.   Internal Monitor Hardware

The intelligence or functionality of any computer is derived from the micro-processor, or Central Processing Unit (CPU), as it processes the string of commands which forms the program being executed. These commands perform very simple tasks and are represented as numbers in the computer. Many commands manipulate data in some form, and so a computer program consists of the string of command numbers interleaved with the required data numbers. This type of program code, which the CPU can execute directly is called machine code. Although it is theoretically possible for programmers to write machine code, the likelihood of mistakes is high because the

code is extremely difficult to read, since it is just a long line of numbers. Therefore if there is a requirement for the programmer to have this level of command execution control, assembly language is used; this uses labels and symbols to represent the command names, jumps, etc.. This program is then 'assembled' which tokenises the labels into their corresponding machine code numbers resulting with the machine code program which can then be executed by the CPU. Assembly language is much more readable than machine code itself, and therefore the probability of errors is reduced. Both are called low-level languages because the programmer has control on the exact commands and the order in which they are executed.

However, most programming occurs in high-level languages. These are more readable still than assembly language, being closer to logical constructs of the English language itself. Rather than each command in the high-level language corresponding directly to a machine code instruction, each command corresponds to a number of them. These two factors combine to provide a great increase in programmer productivity due to the greater readability of the code which results in fewer errors and facilitates the finding of errors that do occur. Also, due to the conciseness of the code that is written since each command corresponds to a number of machine code instructions there is a smaller probability of errors being in the code as there is a smaller number of commands. The translation of the high-level language instructions to machine code instructions can occur in either of two ways; through interpreting the code written by the programmer, or by compiling it. Interpreting the code is where the programmer's code is stored in memory and each statement is translated into its machine code equivalent as the program is being executed. Compiling the programmer's code involves the computer first translating the whole program into the machine code equivalent, and then executing the translated version. Interpreting the code therefore causes the execution speed of the program to be much slower than the execution speed of the compiled version. However, the time involved in first compiling the programmer's code into machine code before it can be executed must also be considered, and when debugging

a program (trying to find the errors and removing them) this causes the productivity of the programmer to decrease since there will always be a noticeable delay (of compilation) between each small change in the program implemented. If however the code were interpreted, the effect of any code changes would be observed almost instantaneously, even though the actual program execution would be slower.

The drawbacks of using high-level languages when writing code is that the programmer does not have full control over the commands executed by the CPU. This is because there is normally no access to the machine code commands, the compilation stage translating the programmer's code into a set of machine code instructions rather than just one. For most applications this is not a concern, but for some (for example the CPU accessing other parts of the computer like the display drivers) the programmer needs direct control at a low level (i.e. direct execution control) of the CPU functionality. This is because the high-level language can never be large enough in its different commands to be able to have separate commands which translate into every possible combination of machine code commands. This problem can be circumvented by the program being written mainly in a high-level language; but with sections which require the direct CPU command execution control (i.e. if the functionality required is not included in the high-level language) written in assembler.

The requirement for this application was to write a program to give the CPU and surrounding peripherals the functionality of an ambulatory monitor. To facilitate the writing of the program, a high-level language called 'C' was chosen as it also includes some low-level functions. However, as will be explained later, not all the monitor functionality was able to be written in C. Therefore a part of it was written in assembler.

Having decided upon the programming language requirements, the type of processor had to be chosen. Rather than decide on a specific processor for which the

programming tools were available and then design the computer system (the CPU with all the other peripherals needed such as memory and Analogue to Digital converters or ADCs) a pre-fabricated computer system was chosen. This was obtained from P.S.I. Systems (17-18 Chelmsford Rd. Industrial Estate, Essex CM6 1XG) and is called the 'Mini-Module'. The other major benefit of choosing a complete system was that it was already interfaced to different programming languages which could therefore be used to write programs, these being Modula-2, C and assembler. This software interfacing included the writing of various routines and functions for the controlling of the peripherals by the high-level language in question, which results in a great deal of time being saved for the programmer, as little or no interfacing in assembler needs to be performed oneself.

The Mini-Module Printed Circuit Board (PCB) measures some ten by eight by a half centimetres in size. It contains a Philips 93C100 CPU (which is Motorola 68000 software compatible), EPROM (Erasable Programmable Read Only Memory) for program storage, RAM (Random Access Memory) for data storage with a lithium battery back-up for when the power is disconnected, a real-time clock, sixteen digital channels which can be independently configured for output or input, four analogue to digital converters, one digital to analogue converter, an RS-485 serial interface, a watch-dog timer, a power fail detector, an LCD (Led Crystal Display) adapter, a keyboard adapter, an expansion bus, and finally a 68000 compatible bus port for connection to external peripherals. In Appendix 1 are included further details and purposes for each component of the Mini-Module that was utilised.

2.1.2. Supplementary Monitor Hardware

The extra physical components required for the ambulatory monitor are described below. These include LEDs, switches, a power supply, and a box to house these and the Mini-Module.

2.1.2.1. The LEDs

As was mentioned previously, four LEDs are needed to provide the monitor's operator with its status. Each one is lit by setting the digital line to 0 so that it is driven by the 100 µAmp source to 5 Volts. This current is sufficient to light low power LEDs feebly but visibly, and so adequately. Another way of lighting the LEDs would have been to have one side connected to the 5 volt battery voltage, and the other to the digital line. When the line was 1 (meaning that it was being driven by the FET to 0 Volts) current would pass from the battery to the FET so lighting the LED, a resistor in series limiting this current. The former option was used since this results in a lower power consumption which is important in this application. Although the LEDs were not lit as brightly as they would be by using the latter option, they were still visible.

2.1.2.2. The Switches

Two switches were required; one for the power connection which would determine whether the monitor is switched on or off, and the other a 'depress on' switch to enable the operator to access different functions of the monitor program.

For the power supply connection switch, the power supply's ground was permanently connected to the Mini-Module's ground. The power supply's positive was connected to one side of the switch, the middle switch connector going to the Mini-Module's +5V rail. Therefore only when the switch was in one of the two possible positions would both leads would be connected so that current could flow.

For the other switch, a 'depress on' type was chosen, where the switch is closed only when the switch is being depressed. An 'input' digital line is attached to one side of the switch, and ground being attached to the other. The unset state for a line is logically zero which corresponds to the 100 µAmps source driving the line at 5 Volts. When the switch is closed, the line voltage is pulled down to 0 Volts (corresponding to 'on' when being read by the CPU) and the current is dissipated as heat by the internal resistance of the batteries and the wire. The CPU can easily access just the one digital line by applying a mask over the other lines when it is being read. Hence the CPU will ignore the other lines by comparing the port's value to a number using an AND function. This function compares the bit values representing two numbers, resulting in a 1 or 'on' if both numbers have a 1 in that position, and a 0 if not. For example 34 AND 66 returns 2 because their binary representations of 100010 AND 1000010 result in 0000010 for that is the only set bit common to both. So for this application, reading the port and performing an AND function with 00000001 will mask out the top seven bits, the answer returned being the last line value which is connected to one side of the switch.

2.1.2.3.   The Power Supply

As the monitor's main requirement was of portability, a power supply made up of batteries was required so that the monitor would be freed from requiring the mains electricity supply. The voltage requirement for the monitor was determined by the Mini-Module; this being a maximum of about 5 Volts and a minimum of 4.75 Volts, which is where the power fail detector begins to operate. It was also thought important to use rechargeable batteries so that there would not be an on-going expense due to disposable batteries having to be replaced when discharged. The other criterion for the selection of batteries was physical size; to minimise battery size for them being housed in the same casing of the Mini-Module.

As ni-cad batteries are freely available and relatively inexpensive, this type of re-chargeable battery was chosen. Battery capacity is in the units of Amp Hour (Ah) which is equivalent to the sustainable current discharge for one hour. For example a 3Ah battery of output voltage 1.5 Volts could supply 3 Amps at 1.5 Volts (or 4.5 watts) for one hour before being discharged. Figure 2.1 lists the different battery types, their voltage output, and the maximum capacity that each holds.

Battery Type	Voltage	Capacity (Ah)
AAA	1.2 V	0.22
AA	1.2 V	0.65
C	1.2 V	1.5
PP3	9 V	0.11

Figure 2.1 - Various ni-cad battery types with their voltage and capacity

AA size batteries were chosen for their volume (including a battery holder) per capacity was smaller than for any other type.

Although their official rating is 1.2 Volts, the voltage output is not static but changes during discharge. Figure 2.2 shows how the voltage decreases for different discharge rates, with 'C' being the discharge rate to exhaust the battery in one hour. It can be seen that the voltage before discharging starts at 1.35 Volts, and reaches 1.2 Volts only a little before the battery is fully discharged. Therefore by using 4 batteries connected in series, a starting voltage of about 5.4 Volts occurs which decreases to 4.6 Volts just before the batteries are fully discharged. As the power fail detector occurs at about 4.75 Volts, this is a useful indicator to the operator of when the batteries need

re-charging. The Mini-Module still functions correctly at the higher voltage of 5.4

Volts and so four AA size ni-cad batteries in series were chosen for the power supply.
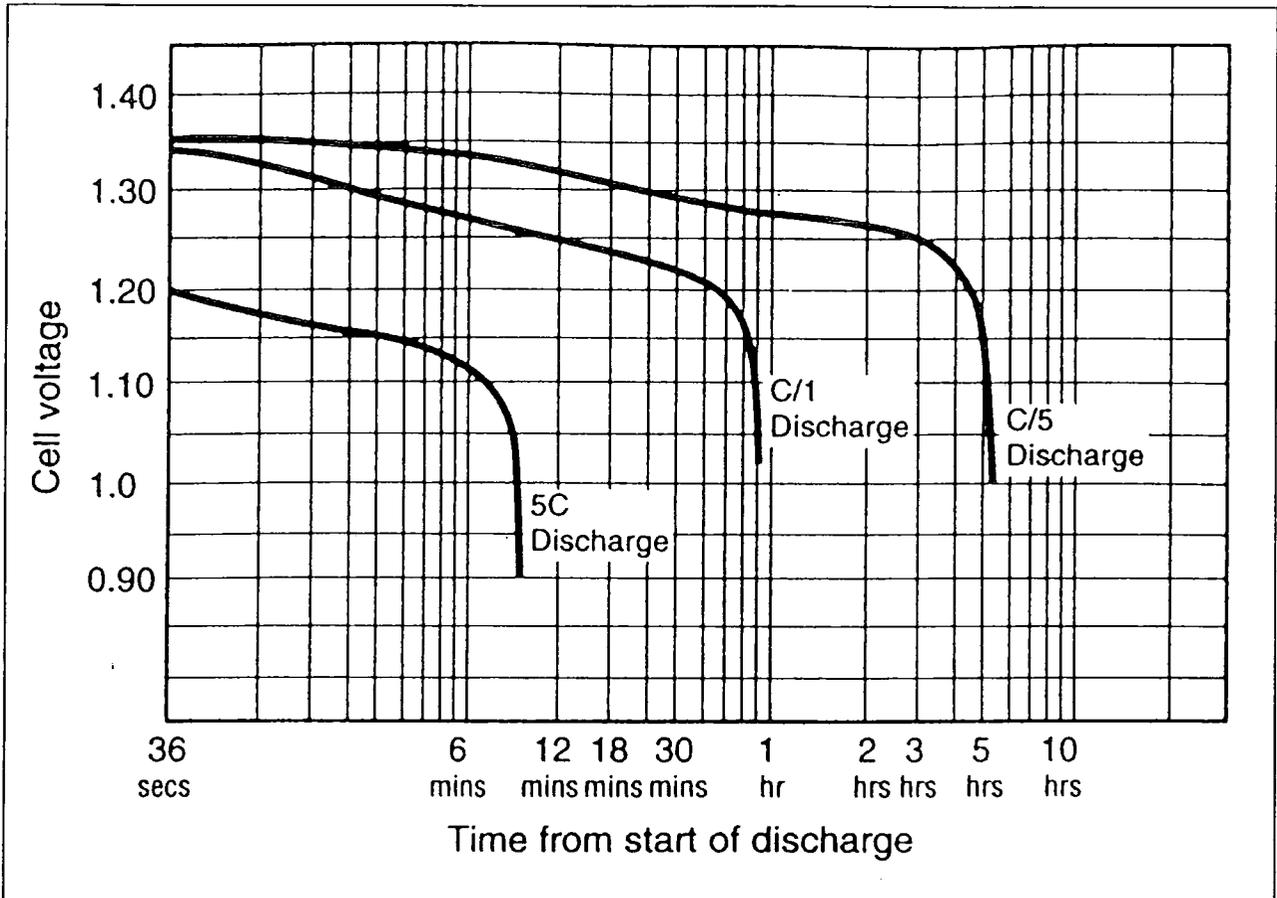
Figure 2.2 - Graphs showing typical ni-cad batteries' discharge times versus cell voltage for different discharge rates (RS Data Library, 1994).

The Mini-Module by itself requires almost 200 mA when running and almost 125 mA

when the CPU is in stand-by mode. However the whole monitor consumes much more

than this due to the need to power the signal conditioning units and when connected

the RS-485 to RS-232 converter. When operating with the powering down of the

processor in between samples enabled, the monitor consumes a total of about 225 mA.

When the processor is not powered down, this rises to about 295 mA. When the

converter is attached to the monitor for communication with the PC, an extra 105 mA is required, with the processor not being powered down in between samples.

As the monitoring sessions comprised of calibrating the transducers for each individual patient, walking the route with the patient, and finally down-loading the data to the PC, one can assume that the serial converter was attached to the monitor for about half the time. This gives an average current consumption of about 315 mA, assuming that the processor was powered-down in between samples when monitoring. This gives a discharge ratio of about C/8 with four batteries, with Figure 2.2 showing an estimated 6-7 hours of battery life. As another requirement was to have an ambulatory monitor which could function for a day without re-charging, it was therefore decided to include a second set of four batteries to be connected in parallel with the first set. The rating for the new power supply was therefore 5.2 Ah, which gave a discharge ratio of about C/16 resulting in an estimated battery life of 12-14 hours. This was deemed sufficient for the application's purposes, but for possible future purposes the option of further increasing battery life by the addition of external battery packs remains.

2.1.2.4.   The Ambulatory Monitor's Box

The box's dimensions are 13 cm by 13 cm wide, and 7.5 cm deep. As can be seen in Figure 2.3, one side holds the switches and LEDs, another the connector for attachment to the signal conditioning units, and the third side houses the connector for attachment to the RS-485 to RS-232 converter. The Mini-Module is clearly displayed in the housing, whilst the eight batteries comprising the power supply can be seen in the lid.

Figure 2.4 shows the ambulatory monitor being worn by a patient. The patient is not unduly encumbranced when walking due to its small size and weight.
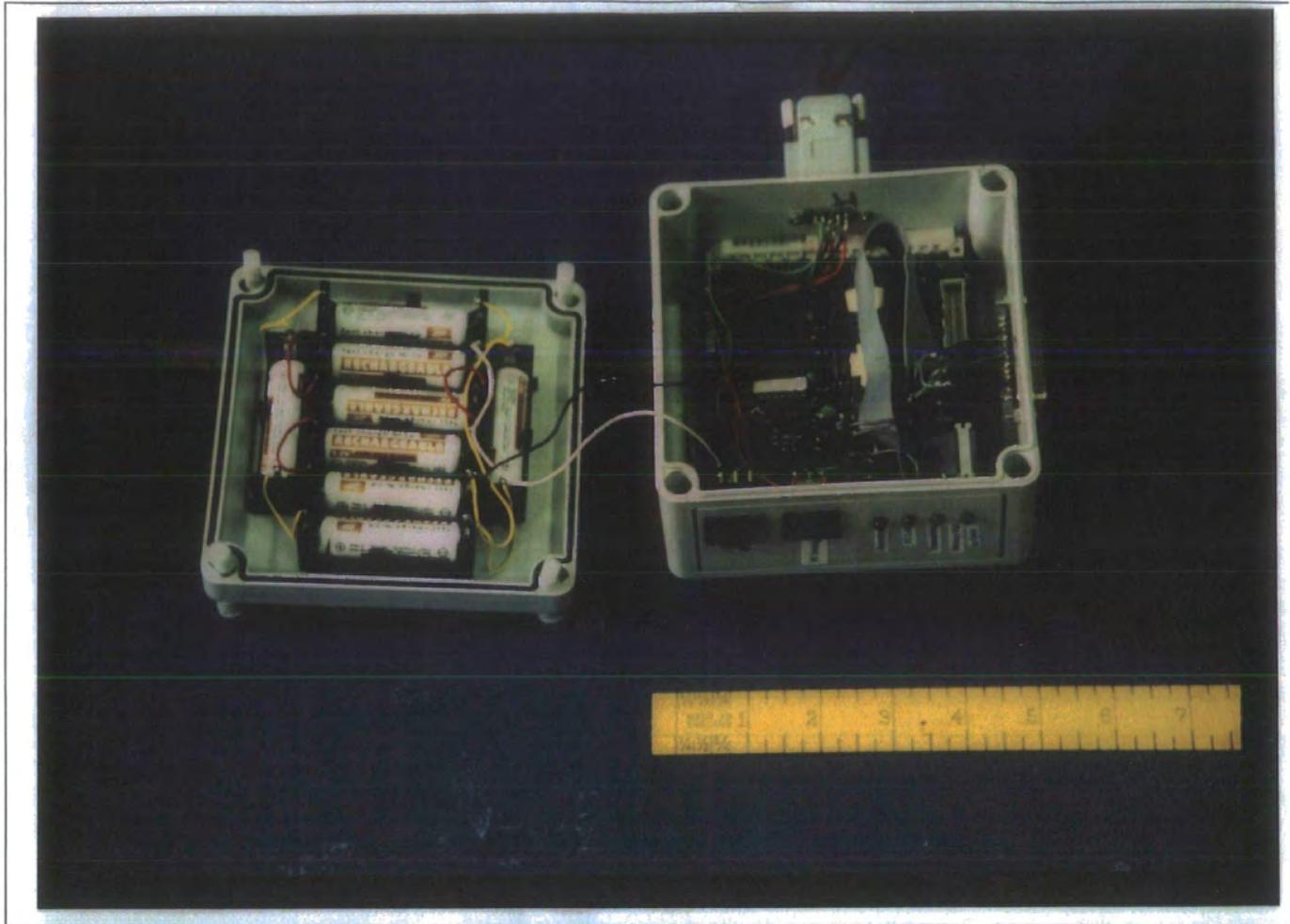
Figure 2.3 - A photograph of the ambulatory monitor with the casing opened, revealing the internal hardware

Figure 2.4 - A photograph of the ambulatory monitor being worn

2.1.3. Hardware Interfacing between the Ambulatory Monitor and the PC

The requirement of PC interfacing was necessary primarily to enable the data collected by the monitor to be down-loaded onto the PC for storage and analysis. The easiest means for this was to use the serial connection which is available to both the PC and the Mini-Module. However a conversion was needed for the Mini-Module's serial interface conforms to the RS-485 standard whilst PCs' conforms to the RS-232 standard. The converter's circuit is shown in Figure 2.5.
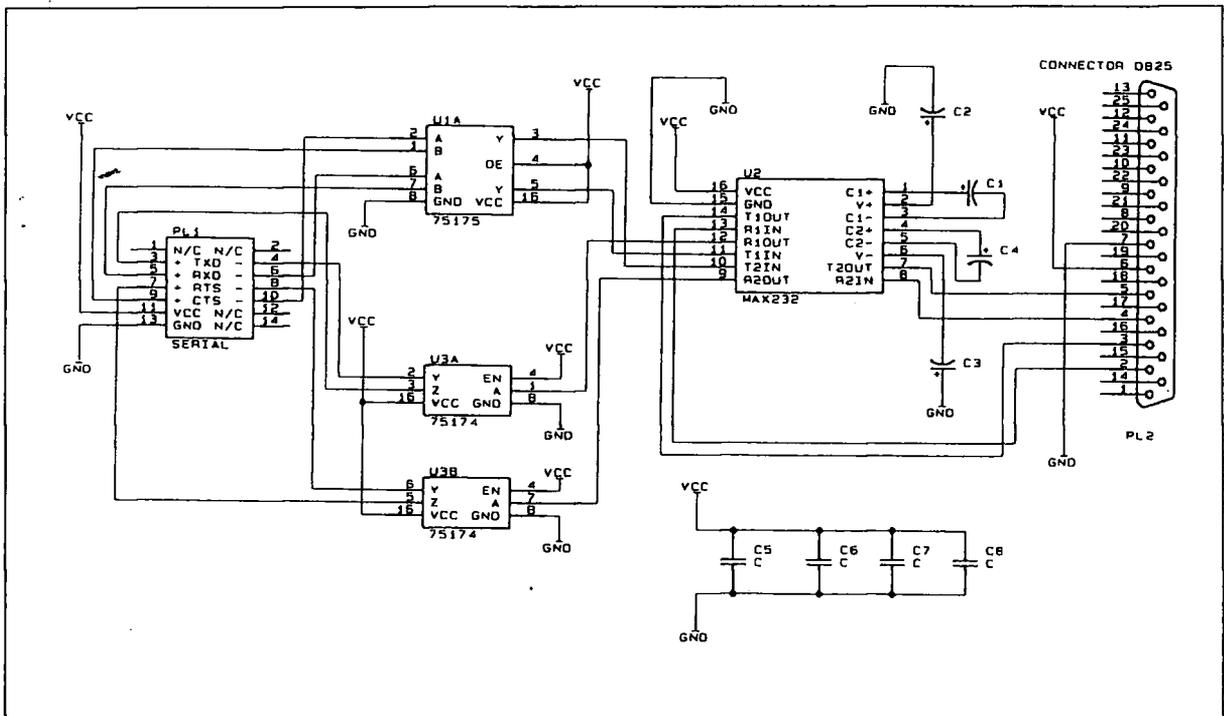
Figure 2.5 - The RS-485 to RS-232 converter circuit

The circuit was housed in a box 8.5 cm by 5.5 cm wide, and 4 cm deep. A photograph of the monitor connected to the PC via the converter is shown in Figure 2.6.

Figure 2.6 - A photograph of the whole monitoring system, with the ambulatory monitor connected to the PC.

The converter circuit is fast enough to allow the standard RS-232 serial communication speed of 9600 baud, which is equivalent in this case to 9600 bits per second or 1200 bytes per second of data being transmitted.

2.2.   Software of the Ambulatory Monitor

As explained in the previous section, the programming language chosen for the monitor program implementation was C. This gave the necessary low-level functionality required for every part of the implementation except for the powering down of the processor to decrease the battery consumption, this being performed using a mixture of C and assembler.

The Mini-Module has a C compiler available for which the necessary interfacing of libraries has already been performed by P.S.I. Systems, which was used as the development package.

The development system consisted of a host PC and program development hardware for the Mini-Module. The host executed the text editor, the compiler, it having its own file storage capabilities for storing the C, assembler, and machine code files. By so doing, the Mini-Module does not require the large amounts of RAM needed to execute the compiler and store the files. Standard compilers generate machine code for the same processor type as that which executes the compiler. As the processors in PCs, being of the Intel '86 family, differ greatly from the Motorola 68000, a cross-compiler was used which executes on one processor type, generating machine code for another. During program development, the generated machine code program was tested by down-loading it from the host to the Mini-Module and then executing it. When connected to the host the Mini-Module could use the keyboard and screen directly by the host executing a terminal emulating program so causing it to function as a terminal to the Mini-Module. This aided testing by being able to view program states and variables on the screen.

The hardware attachment to the Mini-Module also aided program testing as it interfaced both the Mini-Module's inputs and outputs to easily controllable and

viewable components. For example the ADCs are connected to potentiometers, the digital lines to switches and LEDs thus giving the capability of viewing and setting each one's state, and the DAC to an easily attachable point for a voltmeter or oscilloscope. It also includes two RS-232 serial ports, facilitating interfacing to the single or multiple hosts, however this latter option was not used during the application development. The use of twin hosts is sometimes beneficial as each host can run separate programs on the same Mini-Module, each being executed simultaneously by the multi-tasking Minos operating system.

A faster programming cycle was obtained by using a multi-tasking operating system on the host PC. This occurs because the text editor, compiler, and terminal emulating program can be simultaneously in the host's memory thus avoiding the time taken to continually load each program separately in turn during each compile and link cycle. Also whilst the compiler is compiling and linking, this being the greatest part of the cycle, other tasks can also be performed such as program editing or testing on the Mini-Module through the terminal emulating program.

2.2.1. General Program Overview

The following section highlights the main program workings by describing the general

program flow during its execution. The explanation of the C language 'main()' function

of the program is deemed sufficient for this.
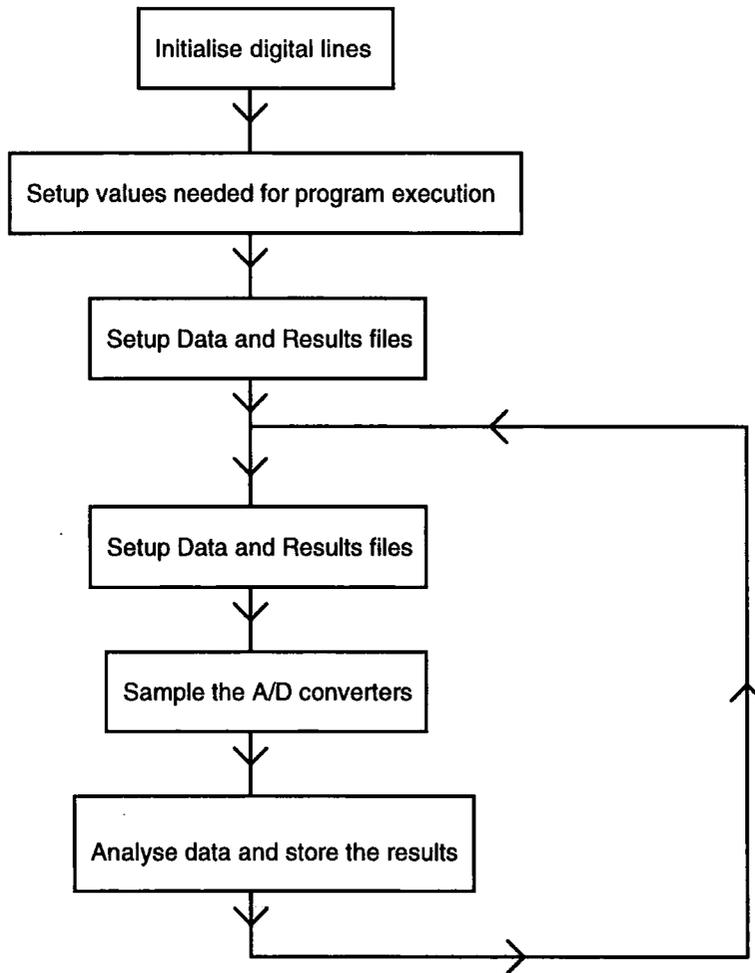
Figure 2.7 - A flowchart outlining the monitor program's general flow during execution

When program execution first commences, a number of initialisation stages are

necessary before the monitoring of the ADCs can occur. First the digital lines are

initialised. As each line can be set to function logically as an input or as an output line,

the former or latter must be specified before usage. Four LEDs were used, and so the

digital line that each was connected to was specified to be an output line by using the

'outch(line no.)' library routine. As the use of a switch for function selection was also required, another digital line was set to be an input line by using the 'inch(line no.)' routine. When the switch was depressed, the digital line became connected to ground so that its state changed from high to low voltage since the normal state for a digital line is to float high. By monitoring the value of this digital line the program could therefore determine whether the switch was being depressed.

The data required and generated by the program is stored in RAM in the format of files. The required data is stored in the 'Events' file, holding values such as the threshold value for an event to occur. The file is not deleted when the external battery supply is disconnected because it is designated as battery backed to the operating system. The program next checks for the existence of an Events file, and if not a new one is created with default values which can be subsequently modified by the operator.

The Data file stores the ADCs' sampled values. Any Data file present is next cleared from the RAM and a new one created. The Results file stores the results of the analysed data, with each result recorded corresponding to one weight-bearing event and including various information, for example the time of its occurrence. If no Results file already exists, for one could still be present from a previous session, it is created.

After the various initialisation stages, the program enters an infinite loop composing of various sections. The first periodically samples the ADCs, storing the data in the Data file. When the capacity of this file is filled, the program enters the next section where it is analysed, with any weight-bearing events being stored in the Results file. Finally the Data file is cleared and the ADCs are again sampled.

This execution loop can be halted to allow the operator to download the results onto a PC, or do a number of other functions. By connecting the Mini-Module to the PC via

the serial link, and depressing the switch connected to the digital line, the Mini-Module

can use the PC as a terminal if the PC executes the terminal emulating program. Since

the majority of the loop execution time is spent in the ADCs' sampling stage, it is in

this section that the switch is monitored. If the switch is depressed whilst loop

execution is in either of the other stages, then there will be a delay of the execution of

these stages being finished before the digital line is read. However, due to the this time

period being in the order of milli-seconds, the delay will not be noticeable to the

operator.

The following sections explain in greater detail the program's functionality. Each

section deals with the stages of the general program flow diagram given in Figure 2.6.

Section 2.2.2 elucidates further on the RAM files used for data storage and their

creation. Section 2.2.3 details the ADCs' sampling stage, with the final section

explaining the analysis of the Data file and the calculation of the results. For each of

these sections, line numbers refer to Appendix 3.

2.2.2.   Data Storage during Program Execution

As mentioned previously, the various types of data are stored in RAM as files. Space

in RAM could have been directly allocated by the program by using the C function

'malloc', with the different types of data being stored in separate sections of memory.

However, with this latter method of storage the data cannot be retained when the

battery supply is disconnected as the contents of the RAM are cleared. This is not the

case when using the files for data storage, for by using the 'datamod' library routine to

specify a file, the Minos OS can ensure that its contents remain intact after the power

supply has been disconnected. This is achieved by using the lithium battery back-up

which is mounted onto the PCB and connected to the RAM. The data in RAM which

is not found in a pre-assigned backed-up file is still contained in RAM but is

subsequently overwritten, for the Minos OS only keeps files which have been set to be backed-up with the 'backup' library routine.

2.2.2.1.   The Events File

The Events file is never erased and is always present in RAM since it contains the values required to control the monitor program operation. These are as follows; the threshold value above which an event occurs, the power down flag to indicate whether to power down the processor in between each ADCs' sample, the display flag indicating whether to display the ADCs' values on the screen, whether one or both legs are being simultaneously monitored, and the four scaling values used for each of the four pressure transducers' outputs. The monitor and PC software were under continuous development throughout the study, with the next addition to its functionality being the simultaneous monitoring of both legs. Unfortunately there was insufficient time to complete this, but what was written is included in the program to aid future development. An example of this redundancy is shown by the value in the Events file which will currently always be set to 1.

The 'open_event_file()' function of the monitor program performs the Events file creation (lines 1935 to 1947). A check is performed for whether an Events file already exists, and if not a new one is created. The size for this file is indicated at the start of the program by the label EVENT_SIZE, with DATA_SIZE and RESULTS_SIZE being used for the Data and Results files respectively. This increases the readability of the code and eases its modification, this being more important for the Data and Results files as the sizes for these are used extensively throughout the program, whilst EVENT_SIZE is only used once at file creation.

2.2.2.2.   The Data File

The previous Data file is erased and a new one created after each main loop in the program, because once its contents are analysed they are no longer required. Rather than actually deleting the file and creating a new one in its place, the pointer to the file could have been set to the start of the file once again; with the next set of data from the sampled ADCs simply overwriting it. The reason for not adopting this functionally simpler approach is historical. During the initial monitor program development, it was useful to check the contents stored in the Data file. Therefore an option, which has now been removed, was to view and download the actual contents of the Data file when the Mini-Module was connected to the PC. If the contents of the Data file were to be viewed, then one would not be able to differentiate where the current data ended, and the previous loop's data began; therefore the former approach was adopted. The actual C program is not made more complex by using this former method, for to erase the file takes two lines of code (for example lines 1783 and 1784); whilst as has been seen before, its creation takes one line. As this is not a speed sensitive application, in the sense of requiring the greatest speed possible in program execution, the time overhead to delete and create a new file, rather than simply resetting its pointer to the start of the file, is not of significance.

To be able to specify the size of the file at its creation, an assessment of the amount of RAM that would be available was made. RAM is not only used as file storage space by the Minos OS, but also for the storage of the program's variables and pointers, and also the OS's own data whose amount varies during any program's execution. Just taking the OS's RAM use into account means that the overall amount which is usable by programs and data storage decreases from the 128 KBytes which is mounted on the PCB to about 100 KBytes. As the information stored in the Data file is transitory, for after each main program loop the contents are erased, its importance is far less than the contents of the Results file which are stored until downloaded to the PC. Therefore the size of the Data file was made much smaller (1 KByte) than that of the Results file

(90 KBytes). Not knowing exactly how much RAM was required by the Minos OS in file creation and maintenance or by the program's variables and pointer storage, resulted with not all the 100 KBytes being allocated for information storage in files. Where two legs to be simultaneously monitored, the file sizes would be halved to obtain 506 Bytes for each Data file and 45 KBytes for each Results file.

As will be explained in Section 2.2.3, the monitor program requires the time to be stamped at the start of the data file for the calculation of the inter-sample time. The time stamp contains the year, month, day of the month, hour, minute and second, and so is contained in six bytes. The remainder of the file stores the sum of the ADCs' values, giving a result of between 0 and 255 which can be stored in one byte. The maximum value of 255 corresponds to 255 kg of mass or 2448 Newtons sensed by the pressure transducers, and so is more than adequate for every eventuality.

2.2.2.3.   The Results File

As explained previously, this file is 90 Kbytes in capacity, and it stores information about each event calculated during the analysis stage.

Before any event information is stored, the Results file is date stamped as this figure will be the same for all the events recorded. This occurs in lines 281 to 320, and as can be seen, the year, month and day of the month is recorded at the beginning of the file. This information takes up three bytes of storage, but by using a compression routine that will be detailed in Section 2.2.4, these three bytes of storage are compressed into two (lines 290 to 296). The next two bytes are set to 255 which indicates to the program the end of the stored results data. When events occur in the data file being analysed, these bytes are overwritten by the event information and the two bytes after that are set to 255. Therefore this end of file (EOF) indication, or more accurately the end of event information stored, as the Results file is a fixed

length, traverses through the Results file as event information is recorded. This is so that the monitor displays or downloads only event information and not the whole Results file therefore saving transmission time. Also if there were no EOF indication every byte in the Results file would have to be set to zero at creation as any other value would be taken as event information.

The data stored for each event is as follows; the time of its occurrence, its peak value, its duration and the time between samples. The time stamp of its occurrence is the hours, minutes and seconds of the start of the event. The duration is the number of samples comprising the event, which is given two bytes of storage, and with the inter-sample time indicates the actual event duration in time. A future enhancement would be to store the calculated duration in time, as this would reduce the amount of storage required for each event. However currently the information stored for each event comprises seven bytes. These seven bytes of information are compressed into five bytes and then stored in the Results file, as will be explained in Section 2.2.4. Therefore the number of events which can be stored in the Results file is 18,431 this being obtained from the following calculation, there being three bytes required for the date stamp:

$$\frac{(90 \times 1024) - 3}{5}$$

Attempting to calculate the amount of time that the ambulatory monitor could function before all the memory was filled can only give an estimate as the exercise patterns adopted by subjects will vary. Assuming a worst-case scenario of the patient walking continuously with crutches at two steps per second (so resulting in one step per second being taken by the monitored leg) an event will be recorded and stored for every second of monitoring.

Therefore the Results file will be filled in 5 hours 7 minutes, this being calculated by:

$$\frac{18,431}{60 \times 60}$$

This almost gives a working day of monitoring before the results need to be downloaded onto a PC so that the Results file can be cleared and monitoring can continue. This is a worst case however, and it is highly improbable that a patient with crutches would walk continuously for five hours without taking rests. A more realistic calculation might be to think of a patient trying to exercise as much as possible during the day, whilst taking frequent rests. Assuming that the patient exercises by walking for ten minutes, resting for another quarter of an hour and resting for more time when eating a meal, means that there is enough memory for monitoring of a 24 hour period, as:

per hour exercising: 24 minutes

3 meals rest: 3 hours

In a 14 hour day: $(14-3) \times 0.4 = 4.4$ hours

Therefore the limiting factor for length of unsupervised monitoring is the battery supply rather than the storage space in RAM.

2.2.3. The Sampling of the ADCs Stage

To aid the explanation of this section of the main program loop, a flowchart is given in Figure 2.8.

The first part of the sampling stage of the program checks to make sure that the required Data file exists in RAM. This should always be true, because due to the sequential nature of the program execution a new one will already have been created. If this is not the case, then due to a transitory fault a program error has occurred. Therefore all the LEDs are extinguished and the 'error' LED lit; an error message is sent to the serial interface, and the program's execution halted. Normally the ambulatory monitor would not be connected to the PC, so the error message would not be seen by the operator. However the LED would inform the operator that a fault had occurred, and the monitor could then be connected to the PC to view the error message and so give information on the reason why the program has halted.

The time and date is then stamped at the start of the Data file so that the time of occurrence for any sample in the file can be calculated, by time stamping the end of the file and dividing the difference by the number of samples.

Program execution then enters its sampling loop, through which it iterates for the number of bytes available for data storage in the Data file. This is equal to the DATA_SIZE label at the beginning of the program, minus the bytes used for the date and time stamps.

Figure 2.8 (overleaf) - A flowchart outlining the sampling of the ADCs stage
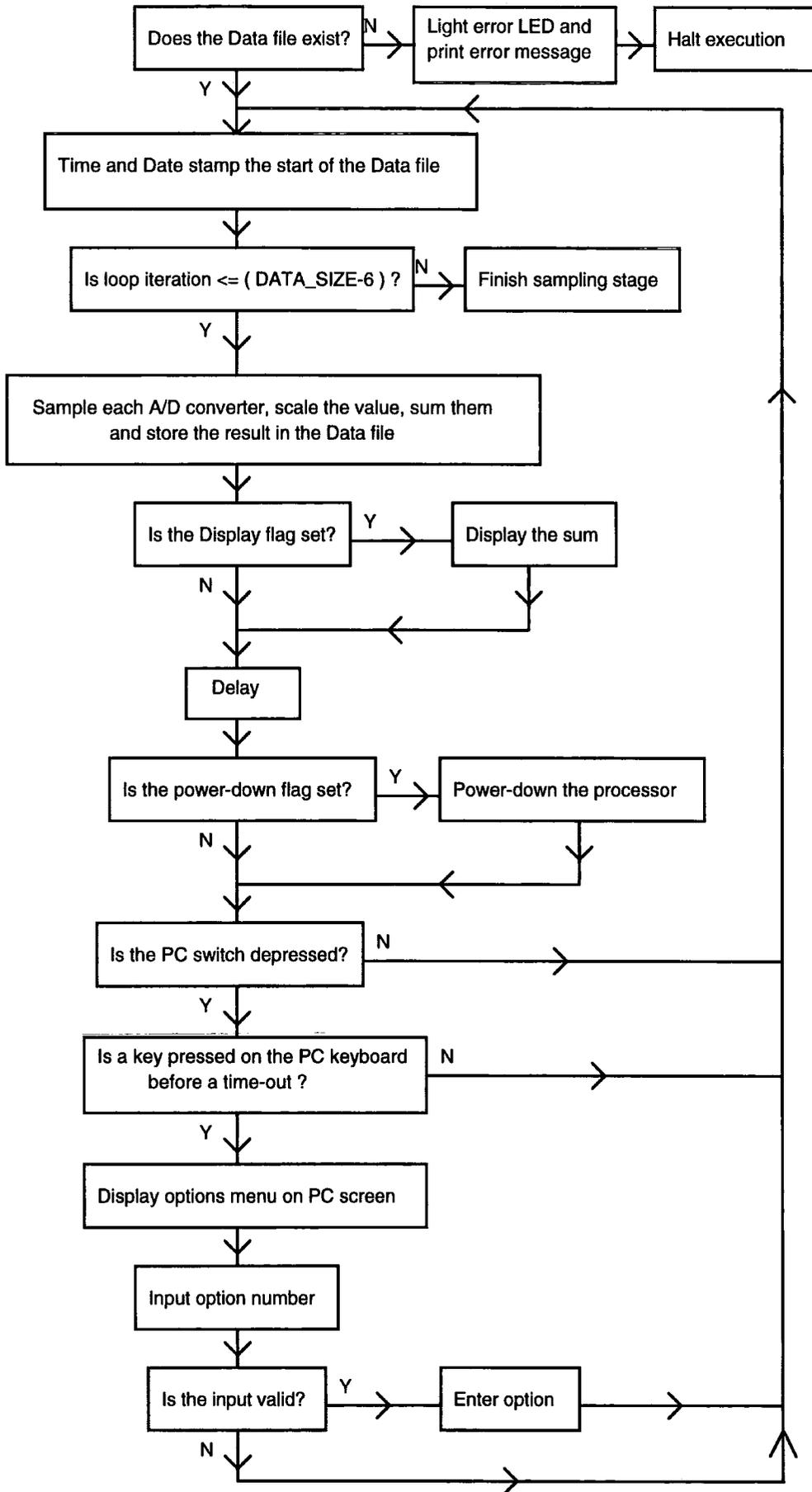
The ADCs are then sampled, scaled by their respective scaling values in the Event file, and stored in an 'unsigned char' variable which is one byte in length. As the scaling values are integers of one byte wide, rather than floats which use a greater number of bytes for storage, each one is divided by a hundred and then multiplied with the sampled value. Therefore a scaling range of between 0 and 2.55 is available, in increments of 0.01. When the summed value was first viewed on the PC screen, it was found that even though no pressure was being applied to any transducer, a reading ranging from zero and twenty Newtons was displayed. This was due to the signal conditioning units giving a minimum voltage reading of between 0 and 0.02 Volts. So to make the final reading more accurate, each transducer's baseline reading was viewed and according to the amount of variation an equal amount was taken away from it. Lines 856 to 860 show this for the second ADC, the first's reading always being at zero Newtons and therefore requiring no modification. This ADC's reading corresponded for the majority of the viewed samples to twenty Newtons, so two decrements occur from its reading, each occurring only if its value is greater than zero otherwise the value would be stored as 255 because it is an 'unsigned char' variable. In the extremely unlikely event of the sum of the ADCs being found to be greater than 255, the sum is modified to 255 so that it can be stored as a byte in the Data file. Once written to the Data file, the file pointer is incremented so that it points to the next location of memory for storing the next value (line 912).

If the display flag in the Event file is set to one, then the sum is printed on the screen of the PC through the serial link. A delay of 0.01 seconds is needed immediately after printing so that the Mini-Module has time to send all the information along the serial line before the subsequent iteration round the loop when the next sampled sum is displayed. This delay occurs by using the 'delay(no_of_tens_of_milliseconds)' library routine in line 906. The effects of not having this delay is that after a number of successive iterations round the loop, the serial buffer will be filled. If further

information is required to be transmitted, then the whole system hangs with the display, keyboard and file storage system of the PC also hanging.

Another delay of 0.01 seconds is programmed to occur which slows down the sampling rate from the kilo Hertz range, to tens of Hertz. If there were no delay the Data file would be filled every second and if a step was being taken which lasted two seconds, then two events would be recorded to have taken place as the Data file will have been filled twice. The slight decrease in accuracy of the peak value and the duration of the event by slowing the sampling rate is of negligible importance because the potential error introduced is of a much smaller order of magnitude being hundredths of seconds in comparison with the minimum of half a second that the foot is in contact with the ground when a patient using crutches takes a step.

After this delay has passed, a further delay occurs if the power down flag in the Event file is set to 'ON' (defined as 1 in line 22). This instructs the program to power down the processor to stand-by mode for 0.1 of a second, the processor then consuming less power for that period. When in stand-by mode, certain parts of the processor do not function, and other parts function at the slower clock speed. Hence all interrupts must be disabled before setting the processor to stand-by mode, and only a limited number of its functions are available in this mode which are sufficient to perform checks for when the condition to power the processor back up occurs. Since the monitor program is written in C, no interrupts are used during its execution as these are only generally accessible using assembler routines. However the monitor program runs under the Minos OS which does use interrupts, for example to enable it to run various programs simultaneously using multi-tasking, and so the registers' contents had to be saved before the processor could be powered-down. To perform both these functions, C could not be used as it does not have the mechanism to access registers or peripherals directly. A routine was therefore written in assembler by P.S.I. Systems, using C to perform some initialisation tasks for it, such as the length of time required

for the processor to be powered down. Lines 2024 to 2086 give the C initialisation functions for the function, which write the length of time which the processor is to be powered-down in the registers of the real-time clock (line 2031 in the 'writereg' function). The assembler listing was not available, and so is not listed in the appendices. However its functionality is as follows; the interrupt and register values are saved on the stack, and then the processor is powered-down. It waits for the required length of time, sets the processor to be driven by the faster oscillator, waits until this oscillator has become stable, and frees the processor so that it can continue operating.

Having performed all the stages of the ADCs' sampling, the program execution enters the second half of the pressure input stage loop which deals with the PC communication. This is effected by using the switch connected to the digital input line, which when depressed causes the line's value to drop from the 5 Volts to 0 Volts, as the other side of the switch is connected to ground. Even if the switch is depressed during the ADCs' sampling, the operator does not have enough time to release the switch before its state is read on entry to the second half of the loop. If the switch is being depressed, the digital line's value will have dropped from 1 to 0. The test occurs at line 932, with the switch being connected up to the digital line number 8, having an alias of 'SWITCH_PC_LINK'.

However, this switch might be accidentally depressed whilst monitoring and so would not be connected up to the PC. A check is required to ensure the ambulatory monitor is in fact connected to the PC and the switch was depressed intentionally by the operator, otherwise monitoring would cease whilst the program continually attempted to access the PC's screen to display its menu of options. This verification is performed by the 'link_test()' function of lines 1741 to 1758. The function executes a loop 500 times, each time monitoring the 'stdin' file to see if a key on the PC's keyboard is

pressed by using the 'ready' library routine, and waiting for 0.01 seconds, thus taking a little over 5 seconds to execute. If a key on the PC is not depressed within that time limit, the program goes back to the ADCs' monitoring by commencing another pressure input stage loop.

If a key is depressed then its value is not lost from the 'stdin' file. However as the menu is not yet displayed on the screen, the operator might not remember the various options available, and each one's key for selection. Therefore the key depression is used simply to confirm to the monitor that it is in fact connected up to the PC, and its value is then discarded by using the 'scanf("%c", &input)' command (line 946) to clear the 'stdin' file buffer. Next the menu options are printed on the PC screen (lines 951 to 961) and another 'scanf' reads the operator's input. According to the input, the program execution enters the option's code. If the input doesn't correspond to any of the available options, the program continues sampling by commencing another pressure input stage loop.

The menu information is shown in Figure 2.9.

```
    Possible options are:
    1: Record results
    2: Edit event level
    3: Calibrate transducers
    4: Restart Results1.dat module
    5: Power down processon ON/OFF
    6: List results to date
    7: Go to shell program
    8: Real time clock
    9: Display toggle

    Please input a number (1-9):
```

Figure 2.9 - The menu of options for the ambulatory monitor

The various options are clearly shown in this Figure. If option one is selected, the results currently stored are downloaded onto the PC. For this to occur, a file name must be chosen that it will be saved under, and which does not already exist in the current directory as otherwise the old file will be replaced by this one, its contents being lost. The choosing of a file name is performed by the 'get_outfile_name' function in lines 1688 to 1732. This specifies a file name which shows it to be a results file from the ambulatory monitor and which contains in the file name the date of the monitoring session. As the date when downloading the results might be different to that of when the monitoring session occurred, for example if the results were downloaded onto a PC the following day, the date of the Results file is incorporated into the file name. As a PC filename can have eight characters before the '.', and three characters after, the name format chosen is as follows. The first three characters are 'DAT', indicating to the PC analysis program that the file contains data downloaded from the ambulatory monitor. The next two characters are the day of the month, the next two the month of the year, and the eighth is the unit of the year; for example the year being 1994, the

eighth character would be 4. This leaves the last three characters after the '.' unused. It was initially envisaged that there would be up to five different monitors simultaneously monitoring different patients. Therefore the last three characters of the filename were originally set to be from '001' to '005' according to the ambulatory monitor which was used for the patient. However it was subsequently decided to use just one monitor as the number of patients which were first envisaged did not materialise, and so the need for simultaneous monitoring sessions was not required. This left the last three characters of the file name redundant. Another use for them was found in that they could indicate a certain patient so that their record could easily be found by the data and analysis files having the last three characters of the file name set to the same number, for example '001' or '012' for the first or twelfth patient ever recorded. However as this patient information is not stored by the monitor, the last three characters are simply set to '000' and then modified by the PC analysis program during the analysis of the monitor results data. So for the monitoring session of date 19/04/94, the filename under which the results would be downloaded onto the PC would be 'DAT19044.000'.

If option two is selected, then the event threshold level can be changed (lines 1063 to 1091). Using a pointer to the RAM location storing the threshold level, this being the location in the Event file , this location can be directly set to the newly inputted value. To guard against operator input error, if the inputted value is less than 0 or greater than 65 kilograms then it is rejected and the operator is requested to key in another value.

The transducers can be calibrated by selecting the third option (lines 1099 to 1360). Calibration is required because each transducer has a fixed area, whilst the different parts of the foot under which each is attached have different areas through which the weight is transmitted, this being compounded by different feet sizes. Therefore the parts of greater area require a larger scaling value to ascertain the total weight

transmitted across the area; whilst parts of smaller area require a smaller scaling value. These reasons are examined in greater detail in Chapter 3. Each scaling value is stored as an integer of one byte in the Event file and before scaling the sampled input, the value is divided by one hundred. Therefore the scaling range is from between '0' and '2.55'. The user can select which transducer's scaling value to change, and before doing so, the monitor displays fifty samples of just that transducer's output on the screen so that the operator can make a better judgement of what the scaling value should be. In between samples, the processor was powered-down as it was envisaged that this would normally occur during a monitoring session.

When selected, the fourth option shown in Figure 2.9 erases the contents of the Results file or files (lines 1372 to 1382). This option was implemented for cases when the commencement of a monitoring session is required, but the ambulatory monitor already holds some unwanted test results in memory. By selecting this option these results can be erased and the current date stamped at the start of the file in readiness for the monitoring session.

By selecting the fifth option, the operator can set the power down flag in the Event file (lines 1390 to 1408). By selecting the option so that it is set to 'ON' (defined as 1 in line 22) the processor is powered down in between samples, thus saving battery power.

The sixth option displays on the PC screen the results currently stored (lines 1417 to 1445). First the LED indicating that the monitor is transmitting information to the PC is lit. Next two temporary file pointers are set to the start of the Results file. The end of the stored data is indicated by having two consecutive bytes set to 255, so one of the temporary file pointers is incremented one byte and the two are then incremented simultaneously, the monitor displaying the value of each byte being pointed to. When both read 255, the end of the stored results has been reached and the monitor stops the incrementation and display of the temporary file pointers.

The seventh option (lines 1456 to 1458) synchronously enters the 'shell()' program which is distributed by P.S.I. Systems with the Mini-Module. This function (lines 2118 to 2504) is a simple command line interface to the Minos OS and was included for debugging purposes to verify the size of the RAM files that had been created.

Option eight deals with the time setting of the real-time clock (lines 1466 to 1584). The stored date and time is initially displayed on the screen, and the user can then request to change them if either is incorrect. For each value inputted by the operator, a check is made to validate it, for example in line 1483 a check is made to ensure that the hour value inputted is between zero and twenty-four. To access the time stored in the real-time clock, the 'getime(struct tm *time)' (line 1467) library routine is used. To store the date and time inputted by the operator, the 'setime(struct tm *time)' library routine is used (line 1579).

Finally, by selecting option nine, the operator can access the display flag in the Events file. Its setting is initially displayed, and the operator is given the option of changing it. If the display flag is set to ON, then the sampled total value for each leg during monitoring is displayed on the screen. This option is useful for the general verification that the transducer scaling values are correct, so that during initial monitoring the operator can confirm that the data values are feasible.

2.2.4. The Calculation of the Results Stage

The flowchart shown in Figure 2.10 highlights the general program execution flow during this stage.

The calculation of the results is performed by the 'calc_results()' function (lines 475 to 660). Initially a test is performed to verify that the Results file exists, and if not then a transient fault or program error has occurred. All the LEDs are therefore extinguished apart from the error LED, and program execution enters into an endless loop, effectively halting operation (lines 646 to 657).

If the Results file is present, as should always be the case, the program next obtains the inter-sample time, this being calculated by the 'time_increment()' function (lines 665 to 724). The start and the end of the Data file was time stamped, and as its size and therefore the number of samples is known, the sampling rate can be calculated. The inter-sample time varies according to whether the processor is powered-down in between samples, assuming that the monitor is not displaying the data on the PC screen for this incurs extra delays. Currently the inter-sample time is stored for each event in the Results file, and downloaded for the analysis program on the PC for the calculation of its duration in time. Possible future development will results with the time duration for each event being stored, so saving RAM space.

The program therefore obtains the date and time at the start of the Data file and then enters into the main loop where the Data file will be analysed to obtain the event information.

Figure 2.10 (overleaf) - A flowchart outlining the calculation of the results stage

Does the Results file exist? →N→ Light error LED and print error message → Halt execution

Y ↓

Obtain the sampling time

↓

Get the date and time from the Data file and increment Data file pointer to start of sampled data

↓

Is loop iteration <= (DATA_SIZE-6) ? →N→ Write an EOF indication in the Results file

Y ↓ ↓

Mark its position in the Data file ←Y← Is this the start of an event?   Finish the calculation of the Results stage

N ↓

Is this the continuation of an event? →N→ Is this the end of the Data file? →Y→

N ↓ N ↓

Is this the end of an event? ←N←

Y ↓

Set event finished flag   Increment event length

↓

Is the event flag set? →N→

Y ↓

Return to the start of the event   Calculate its time of occurrance

↓ ↓

Find event peak   Compress event information and store it in the Results file

↓

Increment Data file pointer to next sample

The following checks (lines 521 to 547) record the start of an event, and calculates the number of samples which constitutes it. As mentioned before, its time occurrence and duration can then be calculated.

If the final check shown in Figure 2.10 is true, then the Data file pointer is taken back to the start of the event and the samples constituting the event are again examined to calculate the peak value of the event (lines 553 to 563). The time of its occurrence is then calculated with the date and time being first set to that at the start of the Data file. Even though the Data file is not large in size, the time of the event's occurrence might be in the next minute, and the next hour, and the next day, and the next year. Therefore when the seconds value is incremented by the samples into the Data file multiplied by the sampling time, thus obtaining the time of the event, a check is performed for whether it is now greater than 59. If this is so, the minute value is incremented, and the seconds value decreased by 60 until the seconds value is less than 60 (lines 575 to 578). As the minute value might be now greater than 59 the same is performed with relation to hours, next for the hour in relation to the day, the day in relation to the year, and then for the year itself. Finally the event information calculated is stored in the Results file.

The seven bytes of information (hour, minute, second of the event, its peak, its duration in number of samples which takes up two bytes, and the inter-sample time) are compressed and stored as five bytes of data. The compression methods used are bit shifting, and decreasing the accuracy of the data.

The first method is concerned with utilising all eight bits which comprise a byte of information. All computers store information as numbers in a binary format. A single byte can therefore store a number of between 0 and 255. However some of the values that are required to be stored have maximums less than this. For example the hour can be between 0 and 23 which in binary is represented as 10111 leaving the top three bits

of the byte unused. By shifting the number up by three bits, the lower three can now be used for the start of the next number to be stored, and so on.

To elucidate further, the following example is given. Supposing that an event occurs at the 23rd. hour, at the 59th. minute, at the 59th. second. The binary representation for each is as follows:

Hour: 23   10111   - leaving 3 bits unused

Minute: 59   111011   - leaving 2 bits unused

Second: 59   111011   - leaving 2 bits unused

The bytes used to store this information would have their bits set as follows:

```
1 0 1 1 1 1 1 1    0 1 1 1 1 1 0 1    1 x x x x x x x
\____v___/\___v___/ \___v___/
   hour     minute      second
```

with the third byte having seven unused bits. Calculating the decimal equivalent of these binary representations, results with 191 for the first byte, and 125 for the second. The bit shifting operation is facilitated in C by the '>>' and '<<' operators (lines 605 to 614).

By limiting the event maximum to 127 Kg or 1219 Newtons, it can be stored in seven bits. The duration of an event in counts of samples might be from one to a very large number if the patient was standing for a long time without walking. However even if the duration of an event is long, as the Data file size is 1024 bytes, the longest event recorded will be 1024 sample counts. By assuming a maximum count duration of 511, especially as the patients were encouraged to walk during the monitoring session, it can be stored in 9 bits. However, by utilising the second compression method, that of decreasing the accuracy of the data, this nine bits of information can be stored in eight,

so reducing the required two bytes to one. The duration value is simply divided by two, and before it is downloaded onto the PC, is multiplied by two. This means that even values will be accurate, and odd values will be rounded down to the lower even value. However, as each count constitutes 0.14 seconds if the processor is powered down in between samples, this slight inaccuracy is negligible.

Therefore as can be seen, the seven bytes of information requiring storage is compressed into four, giving a compression ratio of 33%. An improved Huffman coding scheme which codes each byte rather than just characters as in the original paper (Huffman, 1952) was also initially examined as a possible data compression method, but was discarded for the reason that the data requiring compression would be variable, with different patients monitored at different times throughout the day giving different results of weight-bearing, so that such methods that work on repeatability of data would not be very efficient if the code was previously calculated. In Section 5.3 of Chapter 5, there is explained how improved Huffman coding might be used for further compression by calculating the code during monitoring.

When the event information has been recorded, an EOF indicator is written to the Results file afterwards, so that if the monitor is switched off and then back on at a later time, the data already stored is retained and new results appended.

2.3.   The PC Analysis Software

The following section describes the various parts of the program which analyses and stores the differing data and results files on the PC. As its general purpose and functionality have been explained in the overview at the start of this chapter, this section explains the various parts of this program in greater detail. To this end the various file structures and contents which are used by the program are detailed: these are then followed by the main menu which is shown in Figure 2.11 below, and then followed by Figure 2.12 which shows a flowchart of its general workings. The main body of this section deals with the explanations and screen displays of every option available to the operator during the program's execution.

2.3.1.   The Various Files Used

The program requires and manipulates three different types of file; data, analysis and patient files. Each of these files is stored in different directories to ease file maintenance. Data files are generated by the ambulatory monitor, and down-loaded onto the PC. After having been analysed, they are moved to the '\data' directory of the hard-disk. Patient files are generated by the analysis program by the user inputting information on the patient; and are stored in the '\patients' directory. Analysis files are also generated by the program by analysing the data files, and are stored in the '\analysis' directory.

The data file name format has been explained in Section 2.2.3. It consists of twelve characters; eight then a '.' and three more at the end. The first eight indicate the file as a data file and also include the date of its creation. The first three characters are 'DAT', with the next two being the day, the next two the month, and the final one the (year - 1990). The final three characters give the number of the patient file which

holds the details for the patient. So for example a data file generated on the 23rd.
April 1993 for a patient with the patient file number being 030, would have the
following data file name: 'DAT23043.030'. The first integer in the file corresponds to
the number of legs being simultaneously monitored, and the second is the event level
threshold. Next, the events information is stored which forms the bulk of the file.
Each event recorded requires seven integers to store the information of hour, minute,
second, event peak value, the duration of the event held by two integers, and the inter-
sample time multiplied by one hundred. Finally, to indicate the end of the file after all
the stored events, a single integer value '999' is stored.

The patient file name indicates to which patient it corresponds by its extension number
after the '.'. So the thirtieth patient file generated would be given the name
'PATIENT.030'. The file itself contains information about the patient which is inputted
by the operator before the file is created by the program. The information stored in
this file remains unchanged during fracture healing, and so can be abstracted into this
file so that it is not duplicated in all the patient's analysis files. Separated by carriage
returns, the following information is stored; the patient's name, the date of birth, the
hospital number, a 'R' or a 'L' indicating which of the right or the left leg is broken, the
type of fracture, the position of fracture along the tibia, the treatment method, the
patient's mass, and the date the fracture occurred.

The analysis file name takes a similar form to the data file name, except that its letter
prefix is 'AN' rather than 'DAT'. The contents of each file consists of the following
information separated by a carriage return. The patient name, the number of legs being
monitored, the event threshold level, a list of twenty-four numbers indicating the
number of events recorded in each hour of the day, the mean peak value weight for the
events, the weight variance, the weight standard deviation, the mean number of counts
duration of the event, the duration variance, the duration standard deviation, the

number of weeks before the next appointment, and a string containing the notes made after the monitoring session.

2.3.2.   General Program Structure

When execution of the program first begins, the main menu of options is displayed on the screen to the operator. This is shown in Figure 2.11 below.

```
Current data file:
Current analysis file:
Current patient name:

            1: Change any of above details

            2: Analyse data, storing results in

            3: Display analysis

            4: Examine patient's history

            5: Delete a patient's files

            6: Exit



            Please input a number between 1 and 6:
```
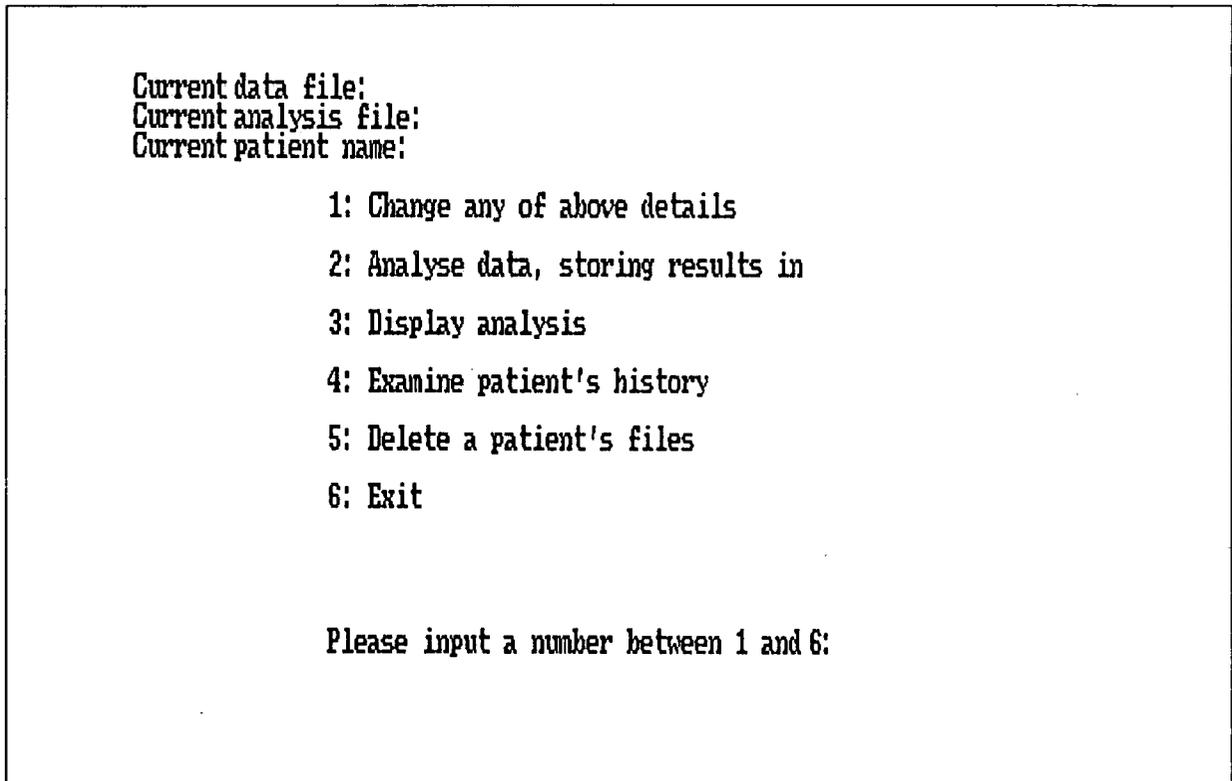
Figure 2.11 - A screen display showing the main menu

As can be seen, there are five main options available to the user. The first option enables one or more of the current files to be changed (the current files are the ones currently being used, and are shown above the main menu, there being a data file, its

corresponding analysis file, and the patient file). The second option allows the data file to be analysed and both the ensuing results and the initial file to be stored in their corresponding directories of the hard disk. The third option displays the analysis data on the screen in a graphical format. The fourth option allows the operator to examine the patient's history, this being both the notes taken for each monitoring session, and a graphical display of the patient's progress (in terms of weight-bearing on the fractured limb) over time. Finally, by selecting the fifth option the program is exited, and the operator is returned to the DOS prompt. Figure 2.12 is a flowchart which gives an overview of the program's workings, the code listing being given in Appendix 4. Subsequent line number references in Sections 2.3.* refer to this appendix.
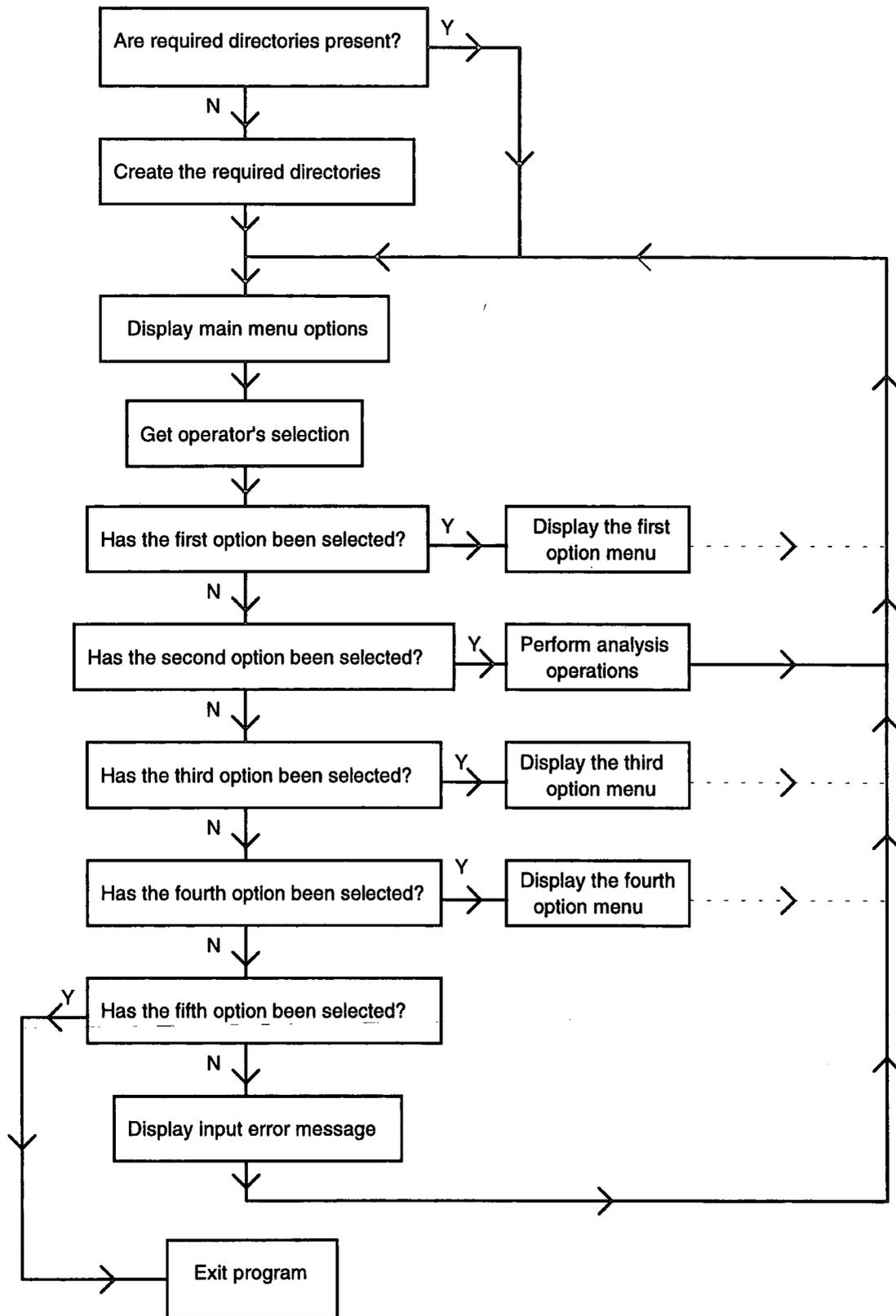
Figure 2.12 - A flowchart giving an overview of the program's general workings. The solid lines automatically proceed from the previous state, whilst the dashed lines show that the program will eventually proceed to the next state (after having entered and exited various sub-menus).

2.3.3. Program Initialisation

As there are a number of different display modes available to PCs, one had to be explicitly chosen for the displaying of the program's text and graphics. If the default option was kept, then older PCs whose display was of lower resolution would not be able to run the program properly as the display would be corrupted. Therefore the display was set to be black and white high-resolution CGA mode (line 74), which nearly all PCs have available, and although of lower resolution than the currently used VGA (640 by 480 pixels) and SVGA (800 by 600 pixels) standards, is still sufficient at 300 by 200 pixels to display all that is required.

The program then creates the required directories for the subsequent storage of the various types of file. Therefore the operator does not have to create these directories manually, when running the program for the first time on a PC.

The program checks to see which directories are present from the root directory, and only creates the ones missing. This check is performed in lines 90 to 107, using the 'system' command. Line 90 shows this, with a '\\' signifying the root directory. Just having the single '\' did not function as it is interpreted by the compiler as a control character. The directory names are saved in the present directory in a file called 'temp.dat'. This file is then opened for reading (line 95) and the finding of any of the three required directories' names in this file precludes these from being created in lines 112 to 117. This method of writing directory and file names to the 'temp.dat' file and subsequently opening the file for reading to access the information is used throughout the program for file and directory searching.

2.3.4. The First Option Menu

On having selected the first option, the program next displays the following menu (generated by lines 142 to 163).


```
Current data file:
Current analysis file:
Current patient name:

            1: Change data file name

            2: Change patient name

            3: Return to main menu




        Please input a number between 1 and 3:
```

Figure 2.13 - A screen display showing the first option menu

The details of the sub-menus accessible from this menu are detailed below, with their workings and functionality explained.

2.3.4.1. The First Option

Selecting the first option allows the operator, via further sub-menus, to either: input the new data file name, display the data files on the disk which are currently not yet analysed, display the analysed data files, or return to the previous menu. The middle

two options have been implemented to aid the operator in the selection of the data file name.

Lines 238 to 417 are executed when the option to change the file name is selected. The 'input_file' function is called which calculates the data file name according to the date inputs from the operator (lines 2231 to 2339). The day of the month is first requested, next the month of the year, and finally the year itself. Each input is tested for validation, with the operator being asked to input another value until one is accepted. From these inputs, a string representation of the file name is built up, the date representation in the file name having been previously explained. This function simply returns the date followed by the '.' and a number corresponding to a previously specified patient name, leaving lines 256 to 260 to add the 'DAT' to the beginning of the file name indicating it as a data file name. If a patient name has not been previously specified (so that a patient file is not currently 'active') then one is requested from the operator (lines 2286 to 2294) and its validity is tested by lines 2296 to 2314. All the patient files are opened in turn (lines 2297 to 2299) to try and match the inputted patient name to the name held in one of the patient files. If a match does not occur, the patient name is rejected and the operator returned to the first option menu. Conversely if a match does occur, the patient file name extension is stored, for the patient's data and analysis files will have the same number extension.

The program next ascertains whether the specified data file has already been analysed (lines 278 to 411). If a data file has not yet been analysed, it does not have a patient number extension, but '.000', and is to be found in the current directory. Therefore lines 278 to 290 search the current directory for the data file, and if found, rename it with the number extension which represents the current patient name (lines 335 to 343). An analysis file name is then generated (lines 366 to 372) and if an analysis file of the same date and extension is not found on the hard-disk then a message is displayed for the operator to analyse the data file. This check is performed as a guard

against an already analysed data file being found in the current directory and so confusing the program into believing that it had not yet been analysed.

The other two options in this menu are performed using the 'system' command in each case (lines 428 and 439), followed by a 'getch()' function call which waits until a key is pressed before continuing program execution.

2.3.4.2. The Second Option

Selecting the second option from the first menu allows the operator, via further sub-menus, to either: change the patient name, display the names of the patients with information files present on the hard-disk, input details for a new patient, or return to the previous menu.

Lines 506 to 533 are executed when the option to change the current patient name is selected. The operator inputs the first and last name of the patient, whose characters are all converted to lower case apart from the first letter of each word which is set in upper case. This is performed by the 'case_convert' function (lines 2346 to 2377), and so ensures that no errors in matching the two name pairs due to case variation occurs. Once converted, the inputted name is compared to the patient names of all the patient files on the hard-disk using the 'get_patient_file' function (lines 2190 to 2223). This iteratively opens each patient file to read the stored patient name to try and match the inputted name with the name in the patient file. If successful, the patient file name is returned, otherwise the string 'unsuccessful' is returned. Therefore if 'unsuccessful' is not returned, the patient name and patient file name are stored in global variables (lines 520 to 523) so that the patient's related information can be readily accessed. A new patient name having been accepted, the current data and analysis file names are deleted, as they will refer to the previous patient.

Lines 505 to 589 are executed when the option to view the names stored in the patient files is selected. Each of the patient files are iteratively opened and the patient name read and displayed on the screen. This displaying is performed by lines 576 to 585 which print the name, displaying three columns of names before inserting a carriage return.

When the option to input new patient details is selected, lines 602 to 867 are executed. The operator is asked to input a new patient name which is converted to lower case apart from the first letter of each name which is in upper case to avoid subsequent mismatching. This inputted name is next compared to every name stored in the patient files, to ensure that the operator does not input details for the same patient twice. If no match occurs, then the patient file names are iteratively obtained and the last three characters of each being converted to an integer (lines 657 to 668). By storing the greatest of these integer numbers, the patient file's number which was last generated is obtained. Incrementing this and converting it back to a three character string results with the new patient file name extension. The following inputs are then requested from the operator, with a validation check being performed when possible; the date of birth, the hospital number, whether the right or the left leg is fractured, the fracture type, the position of the fracture, the fracture treatment, the patient's body mass, and the date of when the fracture occurred. These details which include the patient's name are written to the newly generated patient file.

2.3.5. The Second Option Menu

This option is selected by the operator to analyse the data file. Before doing so, a check is performed to ensure that the currently selected data file name is valid, since this was found to be useful during the debugging stage. This will also guard against the possibility of the stored data file name not having been specified, or having become

corrupted during program execution. A check is also performed to make sure that a patient name has been specified, and if either is invalid the operator is returned to the main menu.

After having read in the first two values of the file, these being the number of legs monitored and the event level threshold, the analysis of the events information takes place. The complete analysis of the data file occurs by a number of read passes through it, each pass calculating different analysis results. The first pass is performed by lines 943 to 955, with the number of events in each hour being recorded as well as a sum of the peak event values and the number of the events. Next the new analysis file is generated with the patient name, the number of legs monitored, and the event level threshold being written to it. Twenty-four numbers corresponding to the number of events for each hour of the day are next printed to the file, each being separated by a space. The peak value mean is calculated by dividing the sum by the number of the events, which is then also written to the file. Using the C command 'rewind()' , the data file pointer is taken back to the start of the file and the next pass occurs (lines 974 to 995). This time the event duration is summed and the square deviation of the peak mass value is calculated in line 985 by using the following formula:

$$\sum [(peak - massMean)^2]$$

When the end of the data file is reached, the mass variance is calculated by dividing the square deviation by the number of events. The standard deviation is then calculated as the square root of the variance. Both the variance and the standard deviation are finally written to the analysis file.

The final pass through the data file occurs at lines 1006 to 1023. The duration variance and standard deviation is calculated during this pass, with the duration mean, variance and standard deviation being written to the analysis file. The operator is then

asked to input the number of weeks before the next appointment which is also written to the file, '0' being inputted if the patient is being discharged. This was to be used by the program to calculate how many monitoring sessions were to be expected at any day. Unfortunately there was insufficient time to implement this function, but the data is still recorded so that the information is present and the file structure is constant were this to be implemented in the future.

Finally the operator has an opportunity to type in notes and observations taken during the monitoring session (lines 1052 to 1104), these being printed to the analysis file as a single string. As each character types is inputted directly into a string, deletes are recorded as an extra character in the string even though on the screen the previous character displayed is automatically deleted. Therefore before printing the string to the file, its characters are iteratively compared to the delete character which has the ASCII value 8 (lines 1085 to 1104). Two character indices into the string are used, one which holds the position of the current character being read in, and the other the being the position of the current character being written to. Each loop iteration then increments both indices, and replaces the character at the write index with that of the read index. A copy of the original string is not required as a delete character simply results in the read index being incremented whilst the write index is decremented so subsequently deleting the previous character. The conclusion of this operation is the writing of the newly formatted string to the file, and the moving of the data file from the current to the '\data' directory (lines 1111 to 1123).

2.3.6. The Third Option Menu

Selecting this option displays the analysis of the data file on the screen in a partly graphical, partly textual form. A check is performed to ensure that the analysis file

exists, and if so the program execution proceeds with line 1157 onwards. An example

of the screen display is shown in Figure 2.14 below.

```
Patient name:                                          Date: 06/04/93
D.O.B.: 25/11/36        Total No. of events = 404  Body Mass = 72
Hospital No.:           Time Mean = 2.13           Weight Mean = 13.735
Leg Fractured: Left     Time Variance = 1.29       Weight Variance = 8.239
Fracture Type: Double   Time Std. Dev. = 1.14      Weight Std. Dev. = 2.870
Position of Fracture: Tibia
Fracture Treatment: Nail              Weight Bearing = 19% of Body Mass
```
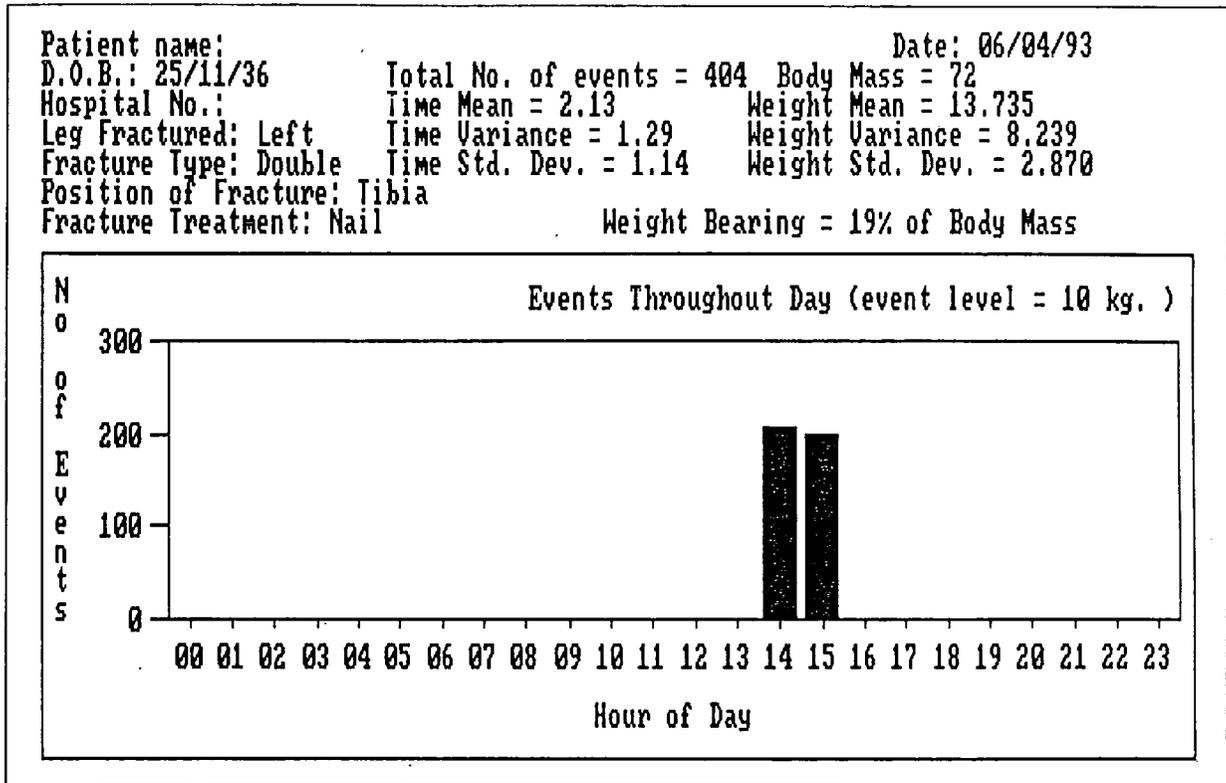

Figure 2.14 - A screen display showing a monitoring session's results. The patient name and number have been deleted for confidentiality.

The patient name is first read in from the analysis file, and the corresponding patient

file name obtained using the 'get_patient_file' function. This is required to obtain the

following information for the screen display: the date of birth, the hospital number,

which leg is broken, the fracture type, the fracture position, the fracture treatment, and

the body mass (lines 1171 to 1183). From the analysis file name, the date of the

monitoring session is obtained (lines 1185 to 1195). The file is then read to obtain the

following information; the number of legs monitored, the event level threshold, the

number of events in each hour, the peak mass mean, the peak mass variance, the peak

standard deviation, the duration mean, the duration variance, and the duration standard

deviation (lines 1197 to 1206). These are displayed on the screen with the percentage weight bearing, which is calculated by:

$$\frac{meanMass}{bodyMass} \times 100$$

The Microsoft C presentation graphics library routines are used to display a bar chart (lines 1289 to 1314), the chart window size being specified by setting the 'env.chartwindow' part of the structure (lines 1310 to 1313), and the chart displayed by the '_pg_chart' command in line 1314.

2.3.7.   The Fourth Option Menu

The patient's history can be examined in various ways by selecting this option. Before the sub-menu is displayed, a check is made for whether a patient name has been specified. If not, then the operator has the option to input one, or return to the menu (lines 1353 to 1366). The inputted name is compared to the name in the patient files for validation, the operator being offered the opportunity of inputting another name until validation occurs. Figure 2.15 below shows the sub-menu which is then displayed.

Current patient name:

    For the above patient:

        1: List the dates of the recorded monitoring sessions.

        2: Examine the notes from the monitoring sessions.

        3: Display a graph of patient's weight-bearing progress up to date.

        4: Return to the previous menu.

    Please input a number between 1 and 4:

Figure 2.15 - A screen display showing the fourth option menu. The patient name has been deleted for confidentiality.

2.3.7.1.   The First Option

When selected this option displays the dates of all the monitoring sessions that have occurred for the patient. All the analysis files for the patient, i.e. all those whose extension is the same number as the patient file are iteratively obtained, and by using the 'get_date' function, are converted to a standard date representation and printed on the screen (lines 1445 to 1467).

The get_date function is listed in lines 2147 to 2182. Its input is a file name, and as this might be a data or an analysis file a check of the first character is performed to determine which. This is needed because for a data file the prefix before the date section is 'DAT' being three characters, whilst with the analysis file it is 'AN' being two. The date is then extracted from the name, and put character by character into a static

string 'date' in the format 'day/month/year'. A pointer to this is then returned, but as 'date' is defined as a static its contents are not deleted when this function is exited.

2.3.7.2.   The Second Option

By selecting this option the notes taken during the monitoring sessions are displayed on the screen. The initial code is similar to that of the previous option, for all the analysis files of the patient need to be accessed, as they contain the notes.

The date of each monitoring session is obtained using the 'get_date' function having given the analysis file name as its input (line 1502). This is printed on the screen followed by the string containing the patient notes which has been extracted from the analysis file character by character (lines 1529 to 1540) to circumvent any maximum string length problems which might occur using the 'fscanf' command to obtain the string in one operation.

2.3.7.3.   The Third Option

Selecting this option displays a graph of the patient's weight-bearing progress up to date. An example screen is shown in Figure 2.16.
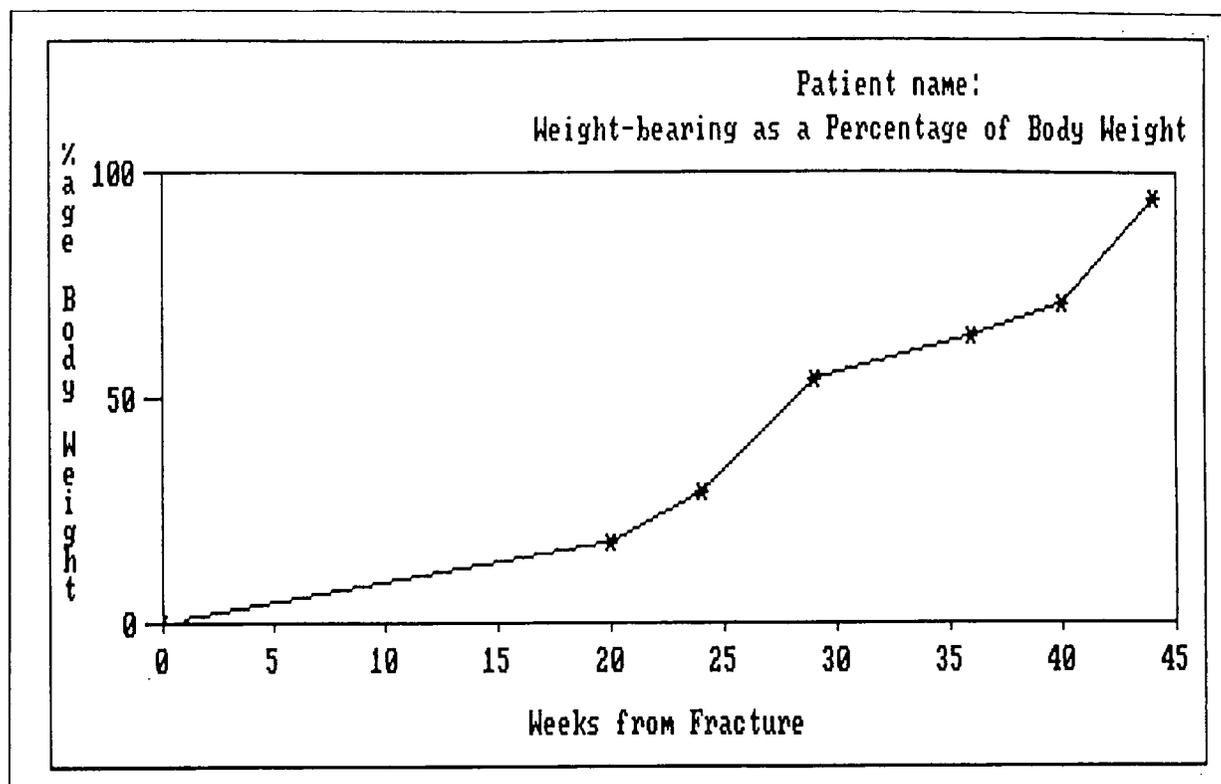
Figure 2.16 - A screen display showing a patient's weight-bearing progress up to date. The patient name has been deleted for confidentiality.

The initial code is again similar to that of the previous options as each analysis file requires accessing for the peak mass mean value. With the body mass value which is gained from the patient file, this is used to calculate the mean percentage weight-bearing. From this file is also obtained the date of fracture.

For each point to be plotted on the graph, the number of weeks from the fracture and the mean percentage weight-bearing is required. The number of weeks is calculated by the 'calc_no_of_weeks' function, which takes as its inputs the year, month, day of when the fracture occurred, and the current year, month and day. The code for this function is given in lines 2116 to 2140. The weight-bearing is calculated by the 'calc_weight_bearing' function which is given in lines 2083 to 2104, and takes the analysis file name and the patient's body mass as its inputs. This then opens the file, reads the mean peak mass of the session, calculates and returns the weight-bearing

figure. As was the case for the displaying the bar chart, the drawing of this graph uses some of the Microsoft C presentation graphics library routines.

2.3.8.   The Fifth Option Menu

As there are a number of files associated with each patient, a separate option has been implemented to allow the program to remove all the files, rather than the operator having to manually remove them. The subsequent patients' files are also renamed so that the patient file number does not remain unused. If selected the sub-menu shown in Figure 2.17 is displayed.

```
Current patient name:

    1: Delete current patient's records and tidy other files accordingly.
    2: List patients on record (number of monitoring sessions in brackets).
    3: Change current patient name.
    4: Return to the previous menu.




        Please input a number between 1 and 4:
```
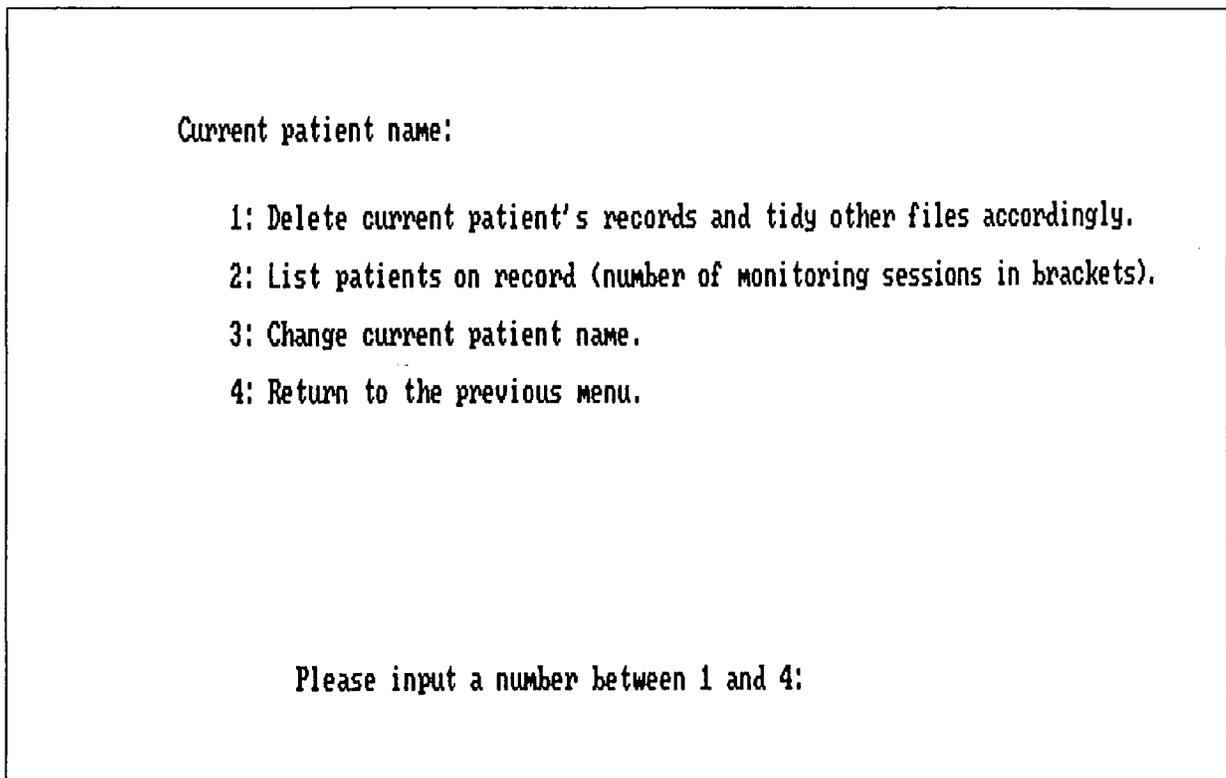
Figure 2.17 - A screen display showing the fifth option. The patient name has been deleted for confidentiality.

2.3.8.1. The First Option

This is the main option of the menu, and is selected when the operator requires the deletion of a patient's data, analysis and patient files. This would not be frequently selected as old patient details are normally retained, but during the patient trials it was found that with a couple of patients only one monitoring session took place, the patient not returning for further monitoring sessions. Therefore it might be useful in the future to delete all files relating to a patient.

Firstly the number of patient files is counted as this figure will be required later. Next the suffix of the patient file corresponding to the patient name is obtained (lines 1780 to 1784) and by using the 'system' command all the data, patient, and analysis files having that same suffix are deleted (lines 1790 to 1803). However, there will now remain a blank patient number allocation in the midst of a block of allocated numbers, for example if files for patient 020 have been deleted when there are patient files up to 030. When assigning a number for newly inputted patient details, this free number will not be allocated as 031 will be chosen which is the next free number after the last generated patient file. To avoid the possibility of exhausting of all the one thousand possible numbers over time, the next section of the code for this option (lines 1811 to 1921) decrements the patient number extensions of all the files whose numbers are greater than the deleted one. This therefore ends with the greatest patient number being one less than previously for the patient, data, and analysis files.

2.3.8.2. The Second Option

This is an enhanced version of the option available when changing patient details, and is useful for the operator to check the number of files that will be deleted. When selected, the names stored in all the patient files are displayed, with a number in brackets after each one indicating the number of monitoring sessions that have occurred for that patient.

This latter function is performed by lines 1967 to 1991. All the data files for the patient having the patient file extension are counted, and the patient name and number of monitoring sessions are then printed on the screen in a single column, there being three columns displayed across the screen (lines 1973 to 1991).

2.3.8.3.   The Third Option

This option was included to enable the operator to immediately change the patient name after having decided with the aid of option two which required deletion.

The 'get_patient_file' function is used to obtain the patient file name from the patient name inputted by the operator. Only if a patient file exists with the same patient name is the name accepted.

2.3.9.   The Sixth Option

The final option available on the main menu is to exit the program. This is included so that the program can set the display mode back to the default mode (line 2072) which was being used when program execution first began.

3.   Pre-Clinical Trials

3.1.   The Initial Sensory Equipment Configuration

To measure weight-bearing, the sensory equipment is required to measure force, for weight-bearing is the force transmitted through the fractured leg. It was decided to use pressure transducers for this purpose. Although just measuring pressure and thus force over a certain area, each pressure transducer output can be calibrated by a scaling value in the monitor software so that the reading corresponds to the force across a greater or lesser area. Different calibration values could therefore be used for the various parts of the underside of the foot for different patients each of which have differing areas.

Initially the weight-bearing sensory equipment consisted of two pressure transducers of half an inch in diameter and their respective signal conditioning unit which had inputs for two transducers. Both of these were purchased from M.I.E. Medical Instrumentation, Leeds, UK. The pressure transducers work on an electrical resistive principle, and consist of two sheets of polymer laminated together, one holding a conducting track, and the other the force sensing resitor polymer area. This makes them durable, vibration insensitive, temperature and moisture resistant. Also, in comparing them with conductive rubber, little hysteresis is exhibited. Thus they were viewed as being very suitable for the in-shoe monitoring function.

The voltage outputs from the signal conditioning unit is read by the ADCs of the ambulatory monitor, but as the outputs are not linear with respect to the applied pressure at the transducer, they first had to be calibrated. This was performed on a Hounsfield testing machine and rather than apply the load directly to the transducer,

the elasticity of the underside of the foot and the insole was modelled by testing the transducers in between two sheets of resilient foam. This was important as the two surfaces between the transducers during the patient trials are elastic, which means that the pressure would not necessarily be linear with the load, since as the load increases the surface area through which the load was transmitted could also slightly increase. The area of foam in contact was about 4 cm^2, this being an estimate of the minimum area that each transducer would be required to indicate the force across. The calibration graph was found to be almost identical for both transducers, and so the average values of both graphs were stored in the monitor program for calculating the load. A graph showing the transducer output against the applied load is shown in Figure 3.1.

Initial Calibration Graph
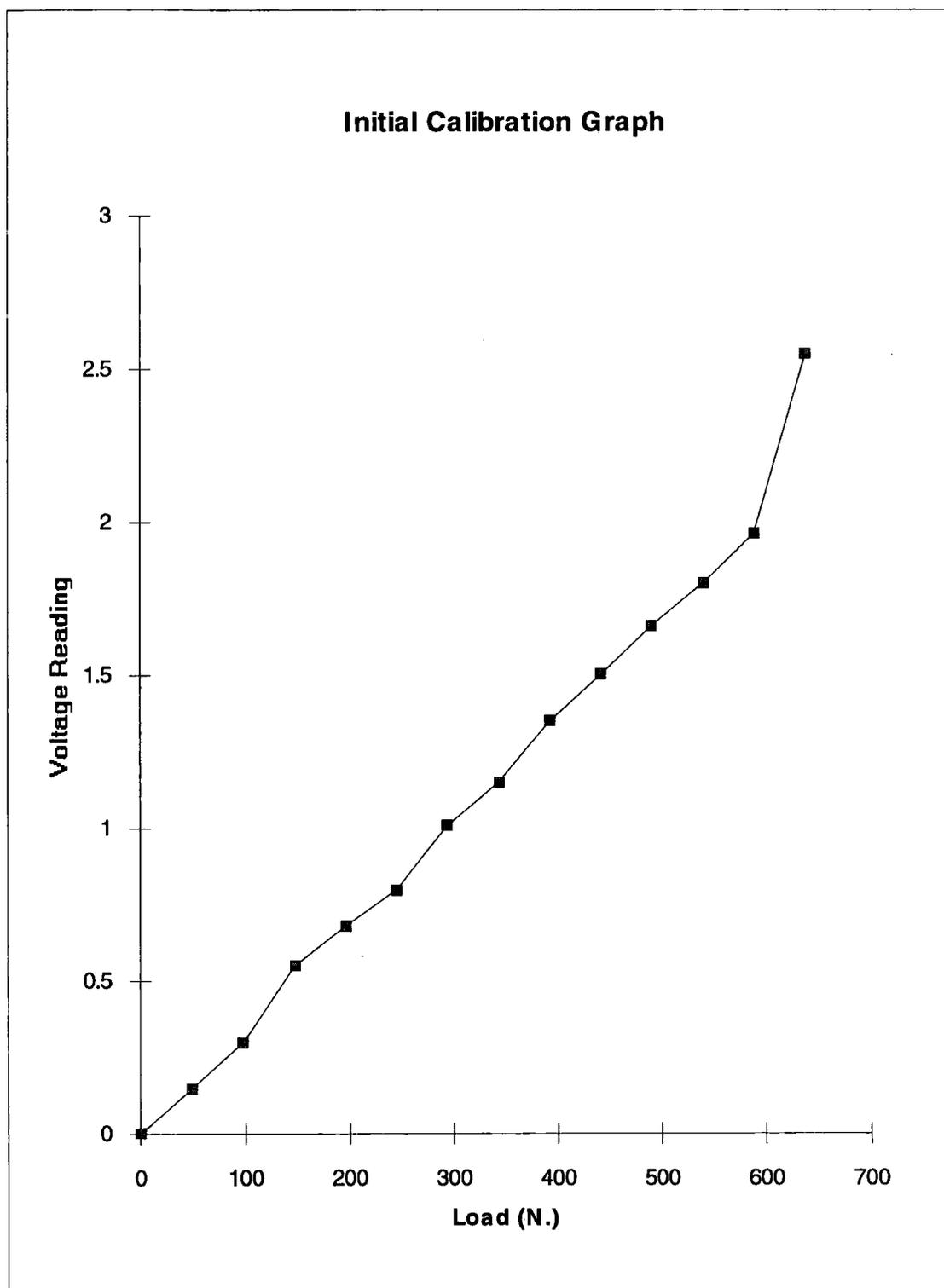
Figure 3.1 - Calibration graph for each transducer

Therefore the first calibration occurs in the monitor program obtaining the load by calculating the gradient in between the points on the graph, rather than assuming the graph as linear and storing just one scaling constant to convert the voltage read to load sensed. However this first calibration is not sufficient to give generically accurate load readings because the different parts of the underside of the foot vary in area as do the actual feet sizes. Also the elasticity of the underside of the foot varies between patients, mostly due to the soft tissues varying with age, the greater the age the less the elasticity, causing a smaller surface area which can lead to a greater pressure for the same applied load. Therefore a scaling factor was also used for each of the signal conditioner's inputs to the monitor, which could be modified separately by the operator through the monitor software. So in effect, two separate calibrations occur which in conjunction enable the monitor to store accurate weight-bearing data.

The transducers were attached by double-sided tape to the hospital plaster shoe which was used for the trials. The signal conditioning unit has inputs for two pressure transducers and was attached to a leather strap which fitted round the patient's ankle, as the wires of the transducers were relatively short. A benefit of this arrangement was that there was only one lead from the ankle to the monitor so reducing the possibility of it interfering with the patient's natural gait.

One transducer placed under the calcaneous, and the other under the first metatarsal head. The reason for this positioning of the transducers was suggested by, amongst others, Duckworth *et al.* (1982), who measured pressures under the foot and found that the highest pressure concentrations were under the calcaneous, and under the first and the second to the fourth metatarsal heads. However as the first metatarsal head was shown to have its pressure distributed almost directly over the head itself, rather than over a larger area, as with the other metatarsal heads, the area under the first metatarsal head was chosen for monitoring.

Before a monitoring session the monitor inputs' scaling values had to be especially calibrated for that patient. After having attached the equipment, the subject was asked to stand with the foot of the fractured leg on a set of bathroom scales and the other foot on the floor. The reading on the scales was therefore the amount of total weight being transmitted through the fractured leg, and the ambulatory monitor reading was required to be identical for the monitor's inputs to be correctly calibrated so that the local weight readings directly under the transducers equalled the total weight. This was achieved by altering the scaling values of the monitor's inputs. During this calibration phase, the operator also visually inspected the way in which the subject was standing on the scales so that an estimate of the weight ratio between the calcaneous and the metatarsal heads would be gained. This was compared to the individual monitor's input readings and further modification of the scaling values occurred as necessary. This calibration now having been performed, the monitoring session could commence by resetting the monitor and so clearing the data file, and allowing the subject to walk on a pre-defined route. At the end of the route, the monitor and sensory equipment was removed from the subject, and the results down-loaded to the PC for analysis and display of the data. If the subject had not been previously tested, then their details were inputted into the PC monitoring program first, with analysis occurring subsequently.

3.2.   Results from the Pre-Clinical Trials

A number of subjects with unfractured limbs were monitored initially, with the

expectation that each session's results would indicate an approximately 100% weight-

bearing average. However this was not found to be the case, with variations being

between 70% and 120%. It was thought that a factor affecting the accuracy of the

data was the positioning of the transducers in the plaster shoe. The shoe was affixed

to the foot by means of two pairs of velcro straps, which were thought to be

insufficient to prevent the foot moving from its original position during walking. Even

if this movement was small, due to the pressure concentrations under the foot

occurring only in specific areas (Duckworth *et al.*, 1992) there might be a large

resultant change in pressure and hence the weight-bearing reading. To circumvent this

problem the subject wore an elastic tubigrip sock, and the transducers were affixed

directly to the sock and therefore to the underside of the areas of interest of the foot.

Further trials were performed with this new configuration, and it was found that the

results obtained were more consistent and repeatable, an example being shown in

Figure 3.2. This shows the PC screen when running the PC analysis program and

displaying the monitoring session results. As can be seen, the average percentage

weight-bearing is 105%, the standard deviation being 4.8 and the average number of

samples comprising the event being 4.75, with standard deviation of 0.8. The

occurrence of the percentage weight-bearing reading above 100% of body weight is

expected due to the larger magnitude of the ground reaction force vector at the heel

contact and push off stages. As the gait pattern was normal and unchanging the

standard deviation of the weight-bearing at 46 on an average of 734 Newtons (i.e. 6%)

might be due to some of the peak weight-bearing values occurring in between samples,

for the sampling interval was 0.14 seconds, so that different events have slightly

different weight-bearing peaks.

```
Patient name: Philip Aranzulla                          Date: 20/04/93
D.O.B.: 7/04/70          Total No. of events = 64   Body Mass = 73
Hospital No.: 123        Time Mean = 4.75            Weight Mean = 76.547
Leg Fractured: Right     Time Variance = 0.69        Weight Variance = 23.154
Fracture Type: Spiral    Time Std. Dev. = 0.83       Weight Std. Dev. = 4.812
Position of Fracture: Middle
Fracture Treatment: Nail                    Weight Bearing = 105% of Body Mass
```
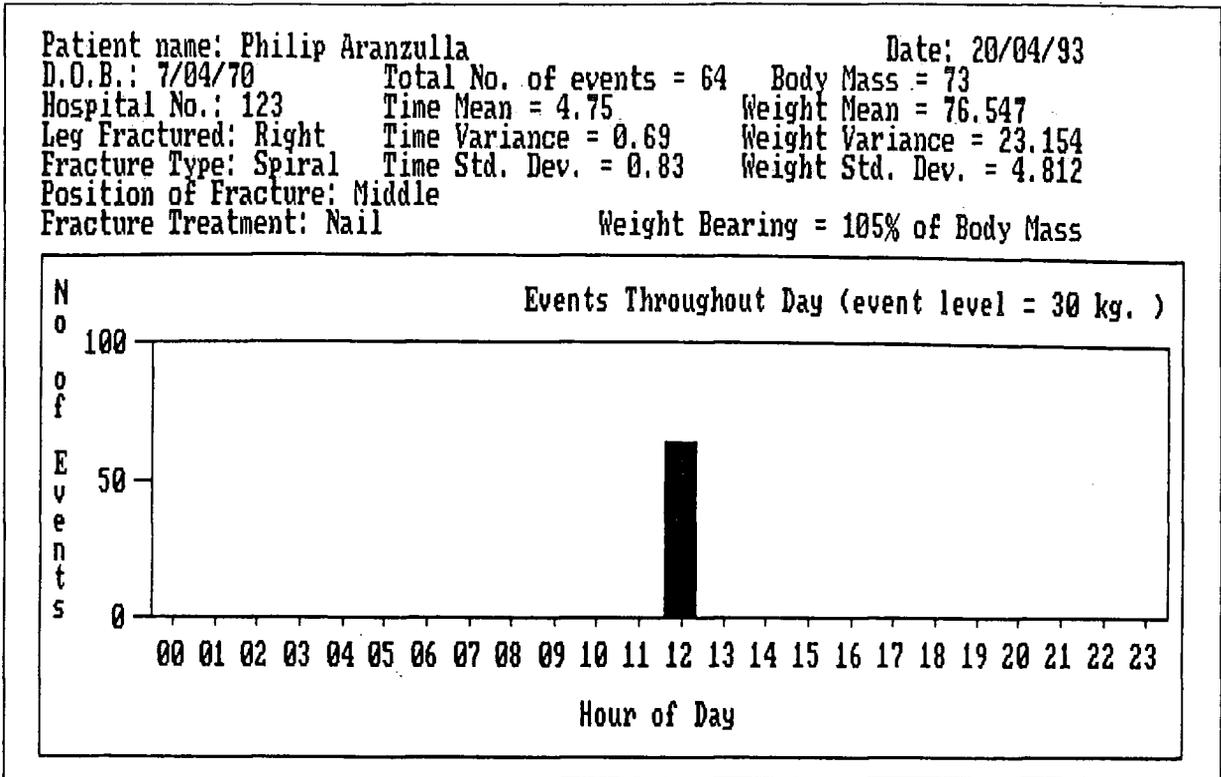

Figure 3.2 - A screen display of the results for the pre-clinical trail using a normal gait pattern.

Another trial was performed with the subject being asked to walk with a limp so entering immediately the flat foot stage on ground contact which was intended to imitate walking with a fractured tibia. By so doing the force throughout the leg is reduced, which is shown by the smaller magnitude of the ground reaction force vector during the mid-stance phases in Figure 3.5 and more clearly in Figure 3.3.

This occurs because the heel contact stage and the start of the flat foot stage is the weight acceptance stage, where the body decelerates its forward velocity. This deceleration is mostly due to the rising of the trunk between heel contact and foot flat as the horizontal kinetic energy is converted to potential energy. Whilst the trunk is ascending an acceleration upwards greater than gravity occurs, and from Newton's Third Law of Motion it can be seen that a force acts upwards, which by Newton's Second Law means that an equal force acts downwards combining with the body weight to generate a greater magnitude of the ground reaction force vector. This

trunk rising does not occur when the patient places the foot flat at ground contact because the trunk has not been lowered to its full extent which occurs at heel contact resulting in less force being transmitted through the leg as the acceleration does not occur.
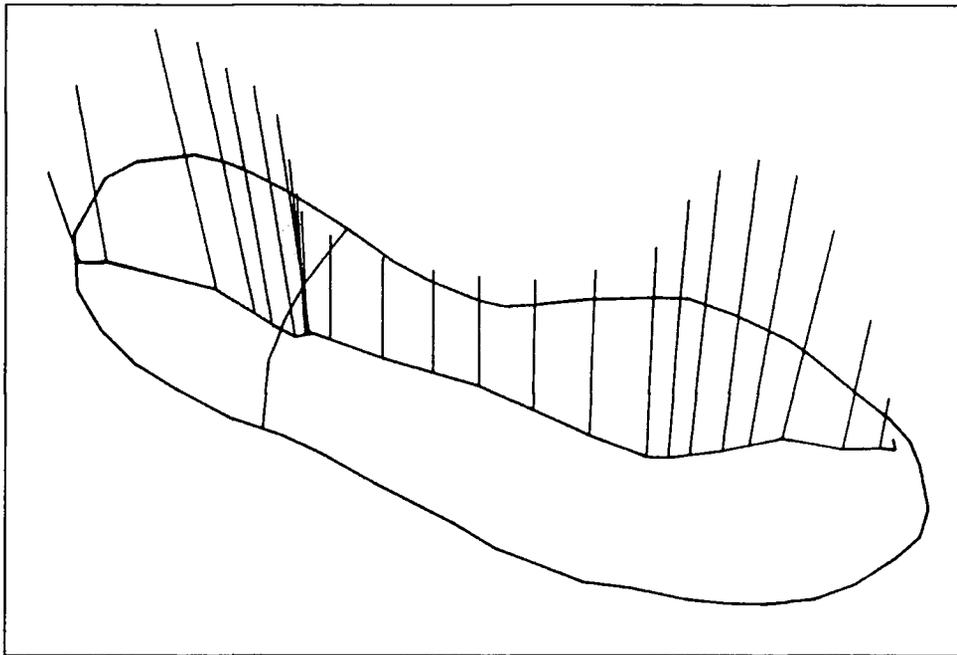
Figure 3.3 - Foot outline, centre of pressure and sagittal plane representation of ground reaction force vector; right foot of a normal male subject walking in shoes. As force is a vector, its magnitude and direction are indicated by the length of the lines and their orientation (Whittle, 1991).

The stance phase ending at heel off rather than push off also lowers the force through the leg as shown by the lower force vector magnitudes during the mid-stance phase in Figure 3.3 and Figure 3.5. The magnitude of the reaction force is greatest at the end of the heel off stage and at the start of the toe off stage as a large plantarflexing moment is generated to oppose the large external dorsiflexion moment, this extra force being transmitted to the ground which compounds with the weight to increase the magnitude of the reaction force. Therefore by excluding both the heel contact to foot flat stage and the push off stage, the force transmitted through the leg for the stance phase is fairly constant at the minimal amount for normal gait.

The results for the pre-clinical trial with limping (Figure 3.4) clearly show the reduced average weight-bearing at 95%. The standard deviation at 69 from 667 Newtons (i.e. 10%) is slightly higher than before, and probably indicates the variations between steps due to the subject having to 'imitate' a gait pattern which is not natural to them. Interestingly the average number of samples per event is similar at 4.7, even though the average stride time was lower. This indicates that the stance phase for the unmonitored leg was of longer duration than that of the monitored and limping leg which is reasonable as the limping mechanism's aim is to reduce the load on the limb. As the stride time is lower for pathological gaits, the effect of the peak value occurring in between samples lessens because the weight-bearing curve over time becomes more smoothed, so in effect the accuracy of the monitor readings will increase. As satisfactory results were obtained for both pre-clinical trials, the patient trials were commenced.

```
Patient name: Philip Aranzulla                          Date: 21/04/93
D.O.B.: 7/04/70          Total No. of events = 73   Body Mass = 73
Hospital No.: 123        Time Mean = 4.74            Weight Mean = 69.521
Leg Fractured: Right     Time Variance = 0.55        Weight Variance = 51.428
Fracture Type: Spiral    Time Std. Dev. = 0.74       Weight Std. Dev. = 7.171
Position of Fracture: Middle
Fracture Treatment: Nail                 Weight Bearing = 95% of Body Mass
```
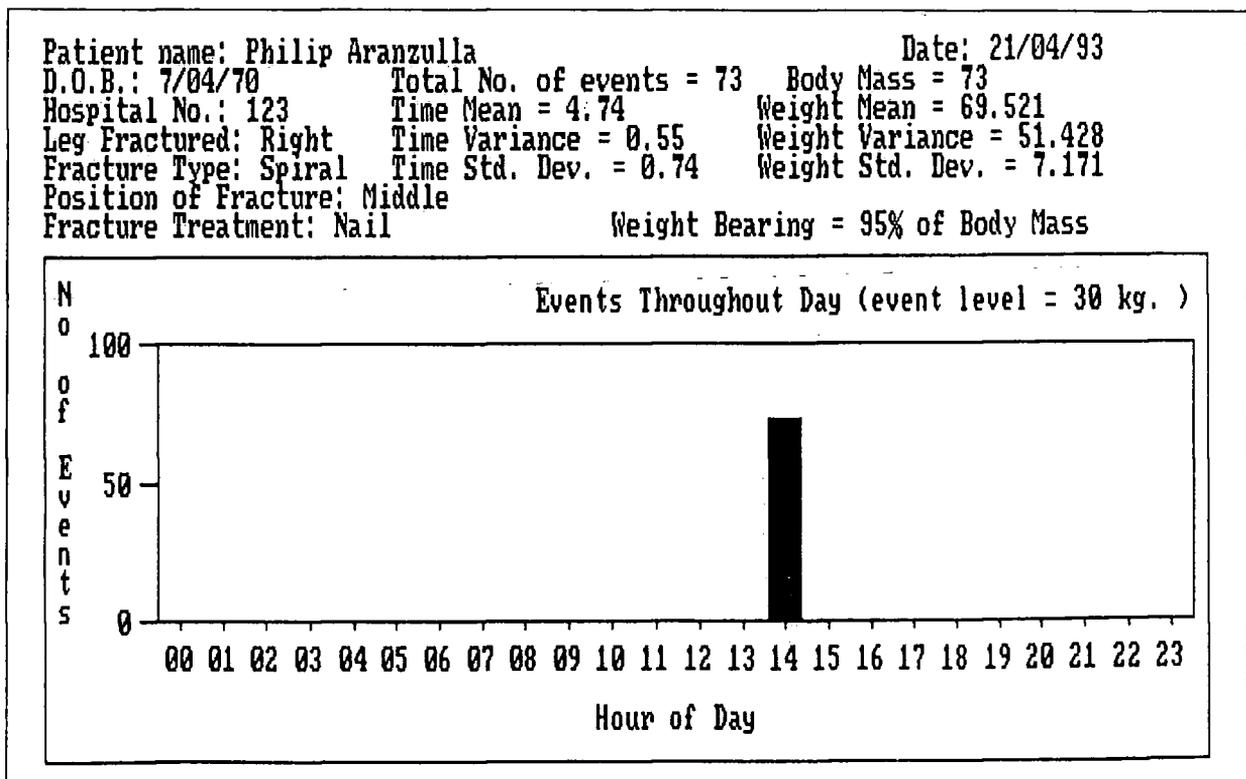

Figure 3.4 - A screen display of the results for the pre-clinical trial using a limping gait pattern.

3.3. The Final Sensory Equipment Configuration

However during the early stages of patient monitoring the results obtained were highly variable, there being great changes from one monitoring session to the next. The explanation for this follows, for there was another factor which affected the accuracy of the data. This was highlighted by Hutton *et al.* (1979), who studied the distribution of the load under the normal foot during walking. When the foot makes contact with the floor, the weight through the leg is transmitted to the floor across various areas of the underside of the foot, and due to Newton's Second Law equal and opposite ground reaction forces are produced. As force is a vector, having magnitude, position and direction, the resolving of all the force vectors results with a single ground reaction force vector. Hutton *et al.* used an array of load cells to calculate parts of the ground reaction force vector, which was plotted over time along the imprint of the foot. The ground reaction force vector moves from the heel to the ball and toes of the foot during the stance phase (which is when the foot is in contact with the ground) due to the weight transmitted moving during the stages between heel contact and toe off. Therefore the sampling of the ground reaction force at various intervals during the stance phase results with what is called a 'Butterfly diagram', this being a plot of all the instances of the ground reaction force vector over time during the stance phase, and an example is shown in Figure 3.5 which was obtained using a force plate.
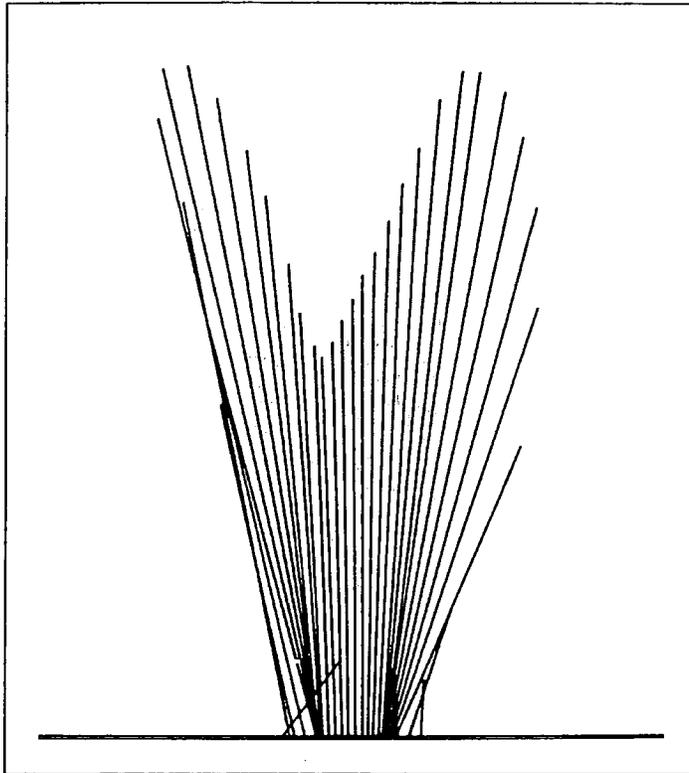
Figure 3.5 - 'Butterfly diagram' of the ground reaction force vector at 20 ms intervals, progression being from left to right (Whittle, 1991).

By using load cells Hutton *et al.* was not able to obtain the direction of the ground reaction force vector but only its position and magnitude. As Figure 3.6 shows, this information was sufficient however to be able to plot the vector's position with time, superimposed over the outline of the foot. By doing this for different subjects, it was discovered that there were quite significant variations in the centre of pressure line, especially when the gait pattern was affected by pathological factors, as in Figure 3.7. For example, with a fractured tibia the heel contact stage might be reduced as well as the toe off stage so that the foot enters the flat foot position immediately on ground contact, leaving the ground during the heel off stage, and so shortening the ground reaction force vector line. Therefore these variations between patients and between monitoring sessions over time for the same patient could result in different weight-bearing readings for the same weight transmitted due to the centre of pressure line being closer to or further away from the transducer underneath the first metatarsal head, since if it were closer then a higher reading would be registered and vice versa.

Therefore it was thought that two pressure transducers were not sufficient for the obtaining of accurate weight-bearing data. The monitor program was therefore modified in order to be able to accept another two transducers, and these were added with the aid of some simple hardware modifications which included the purchasing of another signal conditioning unit. Figure 3.8 shows the new sensory equipment configuration which incorporate both the affixing of the transducers directly to a tubigrip, and increasing the number of transducers to four.

The trials continued using the new configuration, and the subsequent results data found to be satisfactory. This confirmed that the equipment which had been developed is capable of obtaining fairly accurate weight-bearing data.
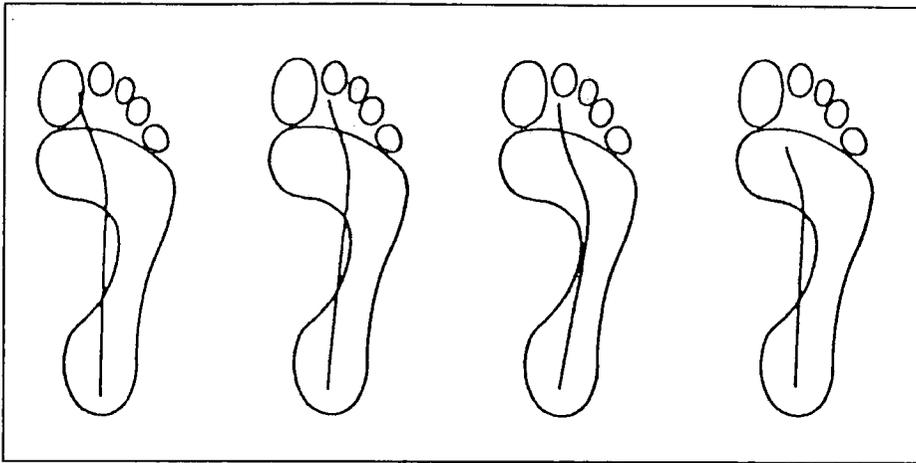
Figure 3.6 - Some examples of the variation seen in the centre of pressure line for normal subjects (Hutton et al., 1979).
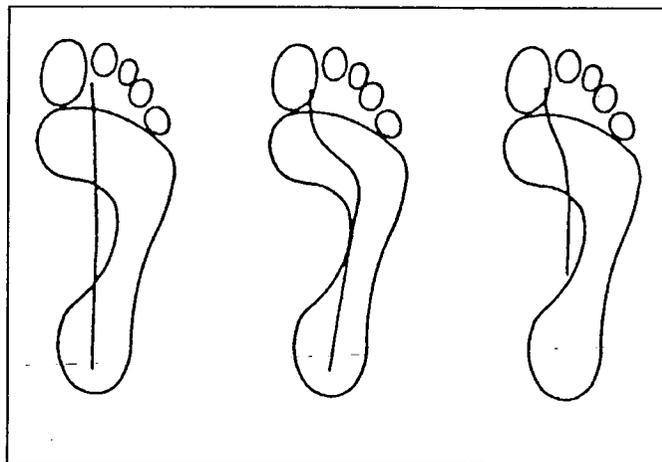
Figure 3.7 - Examples of the variation seen in the centre of pressure line for subjects with pathological gait patterns. The first two diagrams are from patients with rheumatoid arthritis, the third being from a patient with dropfoot (Hutton et al., 1979).
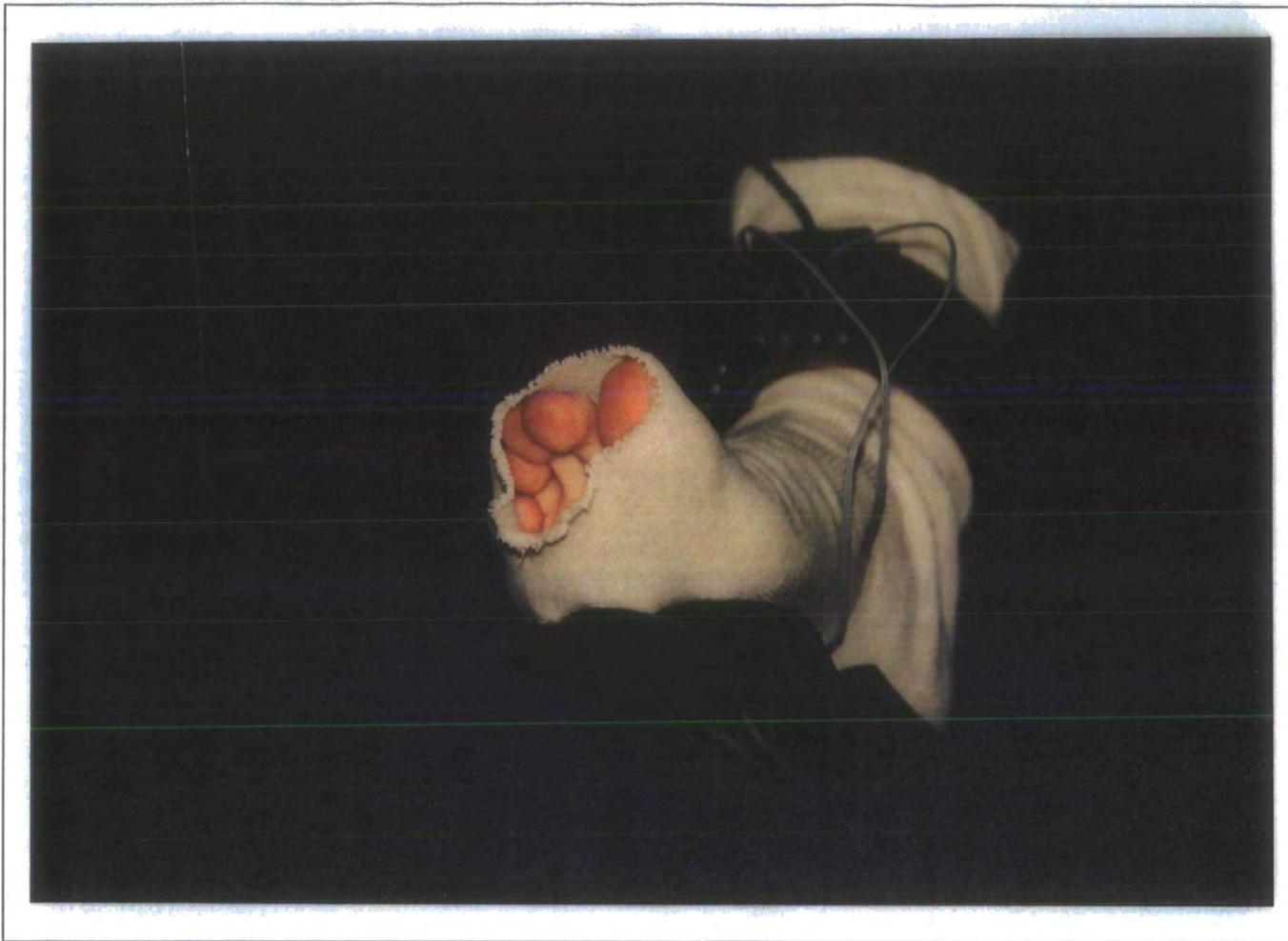
Figure 3.8 - The final sensory equipment configuration.

4.   Clinical Trials

4.1.   Introduction

The clinical trials took place in Middlesborough General Hospital in parallel with fracture clinic sessions. Two afternoon fracture clinics were attended each week from April 1993 to December 1993 and patients with tibial fractures at these clinics were tested with the ambulatory monitor. Although the fundamentals of the monitoring system were operational in April, further development of the equipment occurred throughout the patient monitoring period.

The procedure for the clinical trials was approved by the Hospital Ethics Committee, as shown in Appendix 2, and was as follows. After having attached the monitoring equipment, the subject was asked to stand with the foot of the fractured leg on a set of bathroom scales and the other foot on the floor. As the scales' reading was the amount of total weight being transmitted through the fractured leg, the scaling values of the monitor's inputs were adjusted until the monitor's reading was identical. During this calibration phase, the operator also visually inspected the way in which the subject was standing on the scales so that an estimate of the weight ratio between the calcaneous and the metatarsal heads would be gained. This was compared to the individual monitor's input readings and further modification of the scaling values occurred as necessary. This calibration now having been performed, the data file was cleared by resetting the monitor, and the subject guided to walk a pre-defined route which was about 250m in length, its geometry being shown by Figure 4.1. At the end of the route, the monitor and sensory equipment was removed from the subject, and the results down-loaded to the PC for analysis and display of the data. If the subject had

not been previously tested, then their details were inputted into the PC monitoring program first, with analysis occurring subsequently.

In total 37 different patients were monitored during these sessions. The original plan had been to monitor each patient from the initial weight-bearing period right through to when they were discharged. Each patient could then be categorised according to sex, age and fracture treatment method with the effect of differences within each group and between groups being noted. In practice however, a large number of these patients did not attend subsequent appointments or changed to another fracture clinic session and so further monitoring of such patients was not possible. In fact three or more monitoring sessions were obtained for only 9 of the patients. Seven of these patients' analysis results are detailed below, with a collation of the results following. The other two patients' results are not detailed as both suffered from problems during the development of the instrumentation, due to only two transducers being used.

For each patient there are four graphs plotted; one showing the weight-bearing against time another the other the step duration, or the time that the foot is in contact with the ground, against time; another the time taken to walk the prescribed route against time post-fracture; and the final one the number of events recorded whilst the patient was walking the prescribed route. The values for the second graph have to be calculated as the analysis program's output is the average number of samples which comprise an event. The number of samples was therefore multiplied by the known intersample period to obtain the average step duration. As the processor power down option on the ambulatory monitor program was selected for all the monitoring sessions, by multiplying the value by the known inter-sample period of 0.14 seconds the duration time is obtained. The values for the third graph are obtained by subtracting the time at the first event from that of the last for the monitoring session's Data file. Different patients' third and fourth graphs can be compared with each other because the route

prescribed for the monitoring session was the same on every occasion, and so each

monitoring session's distance is the same.
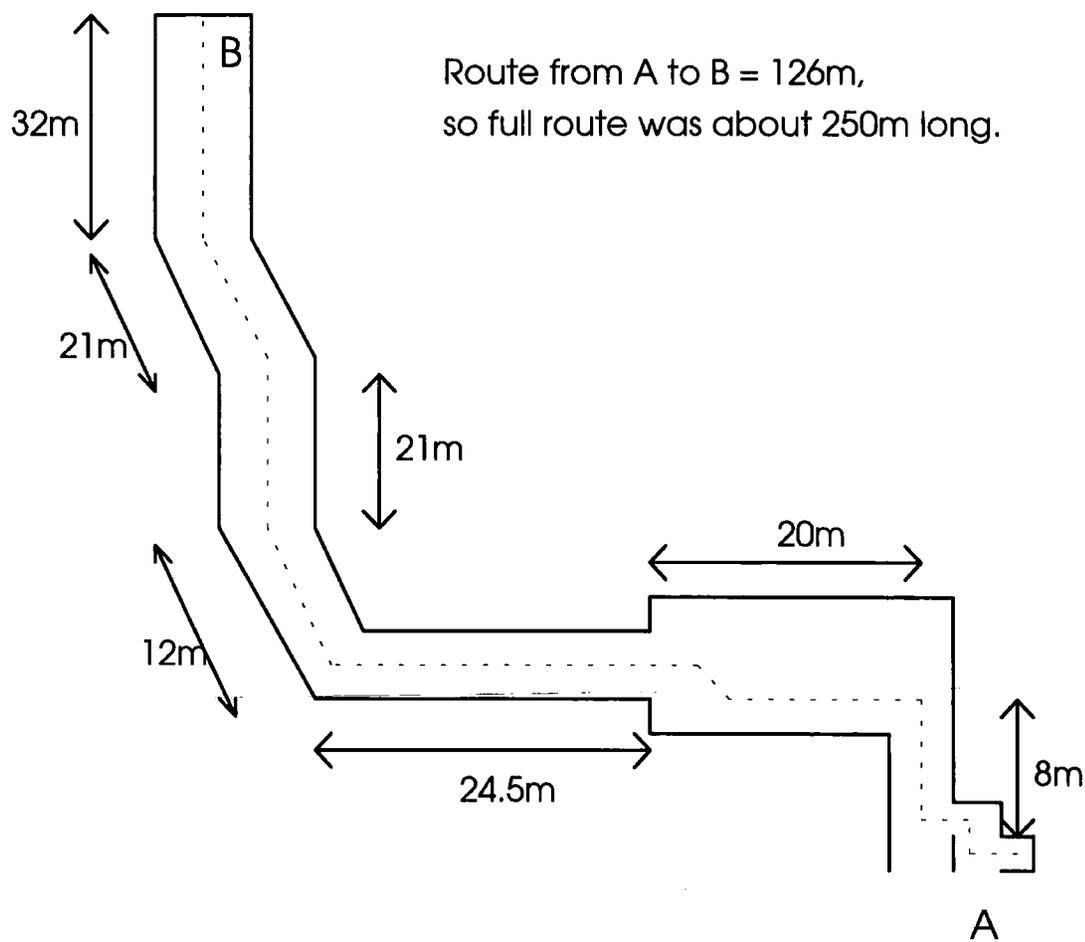
Figure 4.1 - The geometry of the route each patient walked when monitored. The patient walked from A to B, returning to A again.

With both the mean and standard deviation plotted on the first two graphs, a quantitative representation of the weight-bearing and parts of the gait pattern are available. The standard deviation of the weight-bearing quantitatively shows the amount of patient uncertainty and 'testing' of the weight-bearing potential of the fractured leg. The standard deviation of the step duration also quantitatively indicates patient uncertainty in walking, with a high value indicating that the patient was constantly modifying the gait pattern perhaps due to a feeling of instability in the limb. A high value for the weight-bearing standard deviation would normally, but not necessarily, be coupled with a high value for the step duration standard deviation. The monitor therefore provides extra information over visual gait analysis which is valuable in determining the state of the fractured limb.

Before the full patient results are given, the incorrect results for one patient obtained using two transducers are given. These highlight the problems encountered with just using two transducers, which forced the change to monitoring with four described in the previous chapter.

The following 66 year old female patient of weight 528 Newtons was treated with a cast. This is a relatively unusual treatment for an adult and especially elderly patient, but as the patient was very fit with a history of walking and dancing, it was thought unnecessary to use a plate, external fixator or intra-medullary nail even though the fracture was of both the tibia and fibula. The first monitoring session occurred at 14 weeks post-fracture and although a slow pace with a noticeable limp was employed, the recorded average weight-bearing value was 113% of body weight which seemed high. At 19 weeks the patient was again tested and an average weight-bearing reading of 49% of body weight was obtained, this seeming much too low as the patient had by now returned to dancing four times a week. The final reading of 52% of body weight at 23 weeks post-fracture again was too low as the patient was walking unaided for at least a mile each day. The reason why the average weight-bearing and standard

deviation values are incorrect, as explained in the previous chapter, is because only two transducers were used. Therefore their accurate positioning is vital as even a slight displacement of a couple of millimetres can result in a large difference in the pressure sensed and thus the weight-bearing value. Also one expects the patient's gait pattern to change in some measure over the healing period, this affecting the position of the centre of pressure line, and therefore altering the pressure sensed by a transducer in the same position as during a previous measurement. These two factors combine together to affect the accuracy of the weight-bearing data when just two pressure transducers are used for monitoring, for ultimately it can only be by chance that the transducer at the metatarsal heads is positioned correctly. The outcome of incorrect positioning is shown below, with an almost inverted average weight-bearing graph, for it should have started at less than 100% of body weight and possibly risen to that value. However all the step duration readings are correct for these are not influenced by the weight-bearing value and therefore incorrect transducer positioning because for all the monitoring sessions an event threshold value of 96 Newtons (ie.10 kg) was used. The fact that the step duration is greater for week 14 than for the other weeks tallies with the observation during the monitoring session that the patient was walking slowly and carefully as the route was further than the patient had walked since the fracture occurred. As the fracture healed the patient become more confident adopting a faster pace of walking which is shown by the lower step duration.

(Female, Age 66yrs., Body Weight 540N.)
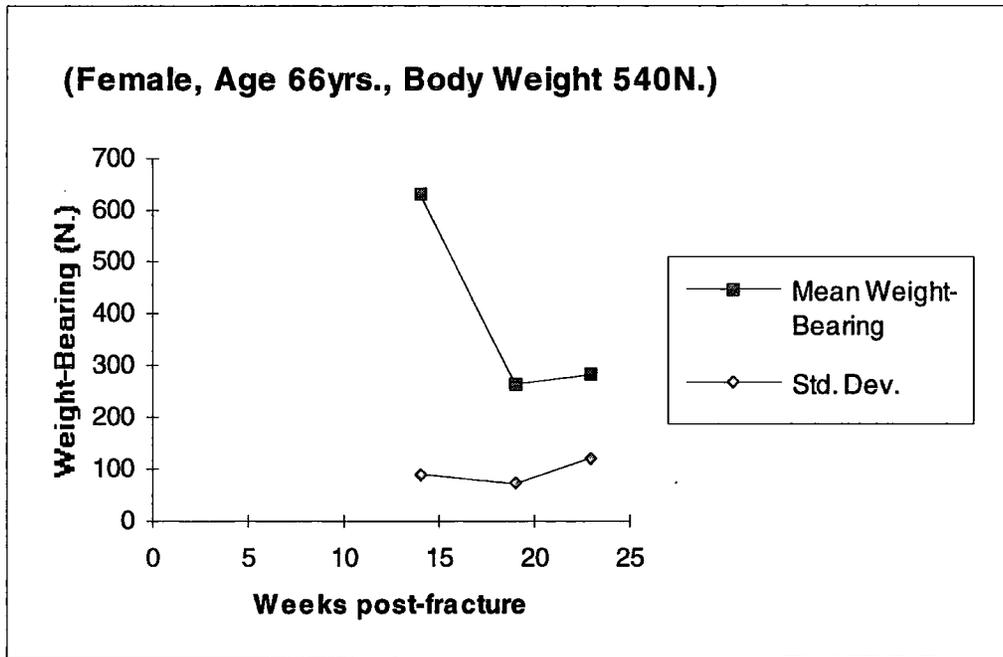
Figure 4.2 - Graphs showing the mean and standard deviation of the weight-bearing with time post-fracture for a patient
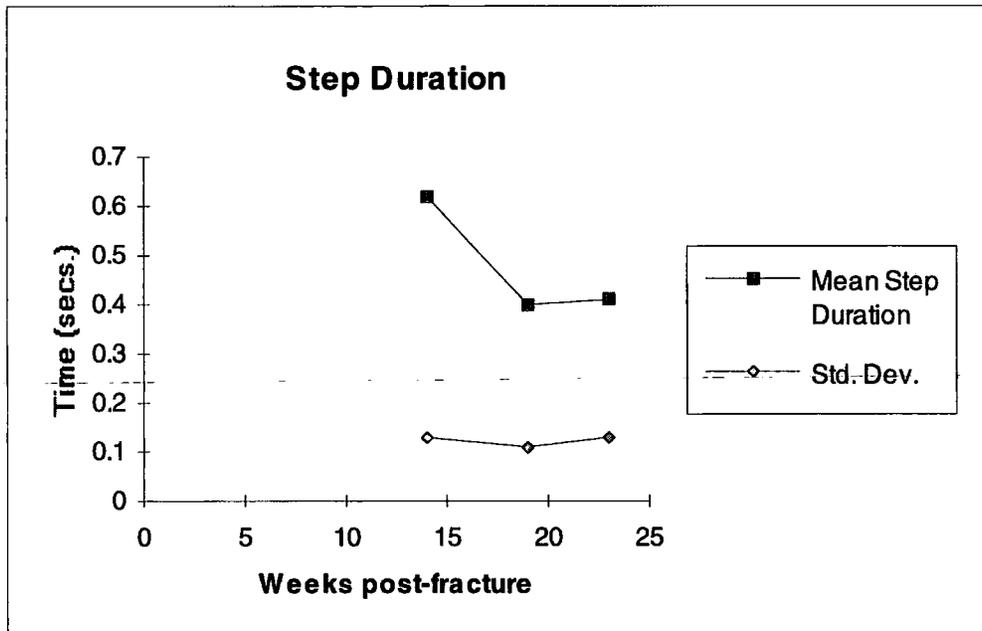
Step Duration

Figure 4.3 - Graph showing the mean step duration with time post-fracture for a patient

4.2.   Individual Patient Results

4.2.1.   Patient 1

The following patient was a 57 year old male of 72 kg who was treated by an intra-medullary nail for a segmental mid-shaft right tibial fracture. During the monitoring period the mean weight-bearing increased from 19% to 100% of body weight.

Radiographs at week 9 indicated that very little callus had formed, so the patient remained in a non weight-bearing cast. The patient therefore began weight-bearing relatively late at week 17 which accounts for the low weight-bearing value at week 20.

As can be seen in Figure 4.4, a large increase in mean weight-bearing occurred between weeks 24 and 29 post-fracture, this being from 30% to 54% of body weight. The patient was walking with the aid of crutches, and by week 29 the gait pattern was much more fluid and confident, with no perceptible limp, the stiffness and strength of the leg obviously having increased so that the patient felt more confident to increase weight-bearing. This however does not explain the large decrease in step duration recorded during this monitoring session and shown in Figure 4.5. Examining the data obtained from the ambulatory monitor indicated that there was a fault with the monitor software as there was only a small number of events with most being of 1 sample in duration, so explaining the values for week 29 in Figures 4.6 and 4.7. This fault occurred due to the changing of the monitor software during its continuous development and enhancement, so that the previous data was reliable even though the current was not. The fault was rectified after the monitoring session, so that subsequent data was again accurate.

By week 36, the patient was using one stick in opposition, with a slight limp being noticeable. This limp was still apparent at week 40, but the patient confirmed this to be habitual and was consciously trying to rectify this. A stick in opposition was still being used, but not in the house. Excluding week 29, Figure 4.5 shows a general trend towards a longer step duration up to week 40. Examining Figures 4.6 and 4.7 reveals that the velocity and stride length increased up to week 36 post-fracture for the distance traversed was the same for all monitoring sessions, with the stride length increasing still further by week 40. Although the patient was actually using a faster velocity the increased time that the foot was in contact with the floor and the increased stride length indicate a more symmetrical gait pattern as both stance phases tend towards equal length. This must occur as the stride length is increasing due to the fractured leg being more extended as its stance phase has more normal heel contact and foot flat stages, this being indicated by the increasing time that the foot is in contact with the floor. Therefore a perceived increased stability and strength of the leg by the patient is indicated.

Finally by week 44, the patient was walking much more confidently and with greater fluidity without the aid of a stick, the limp having been almost totally eradicated. The velocity was also greater which was shown by the slight decrease in the step duration in Figure 4.5, and the lower session duration in Figure 4.6. Figures 4.6 and 4.7 show a general increase in step duration during the healing period.
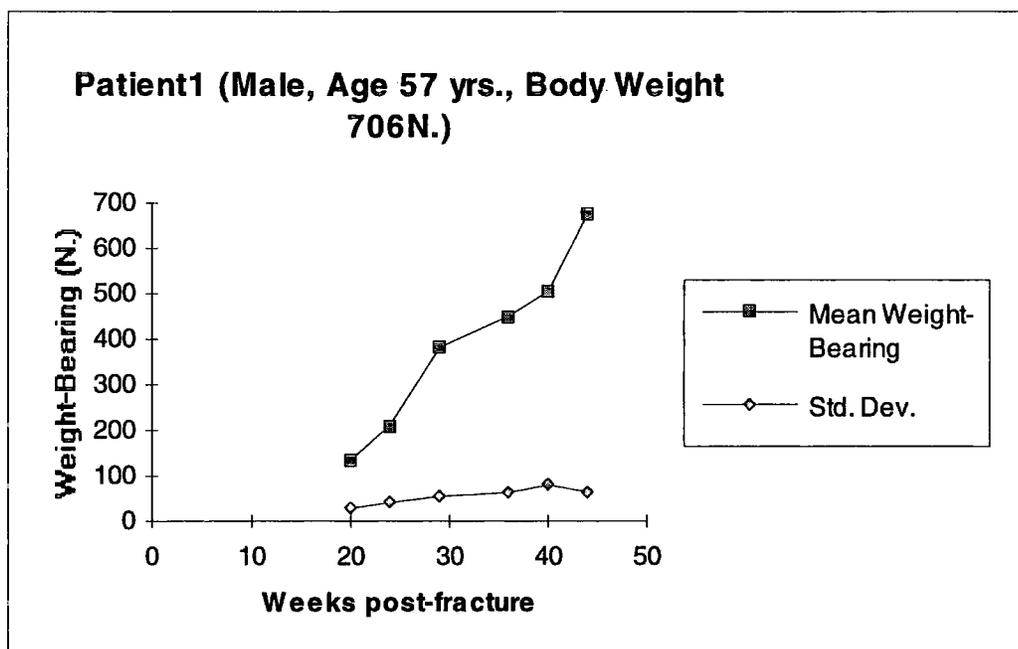
Figure 4.4 - Graphs showing the mean and standard deviation of the weight-bearing with time post-fracture for a Patient 1
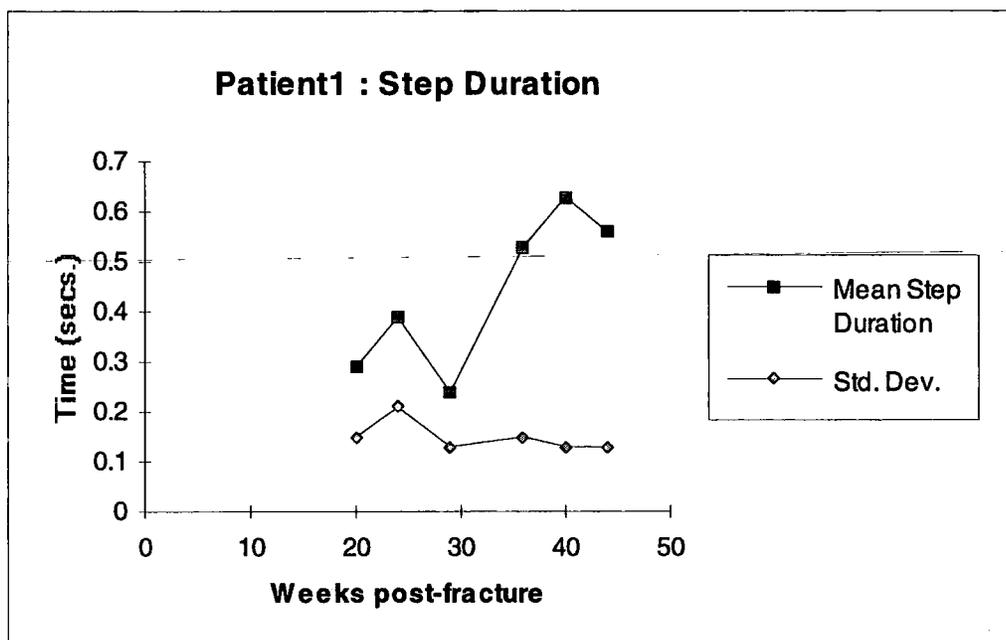
Figure 4.5 - Graphs showing the step duration with time post-fracture for Patient 1
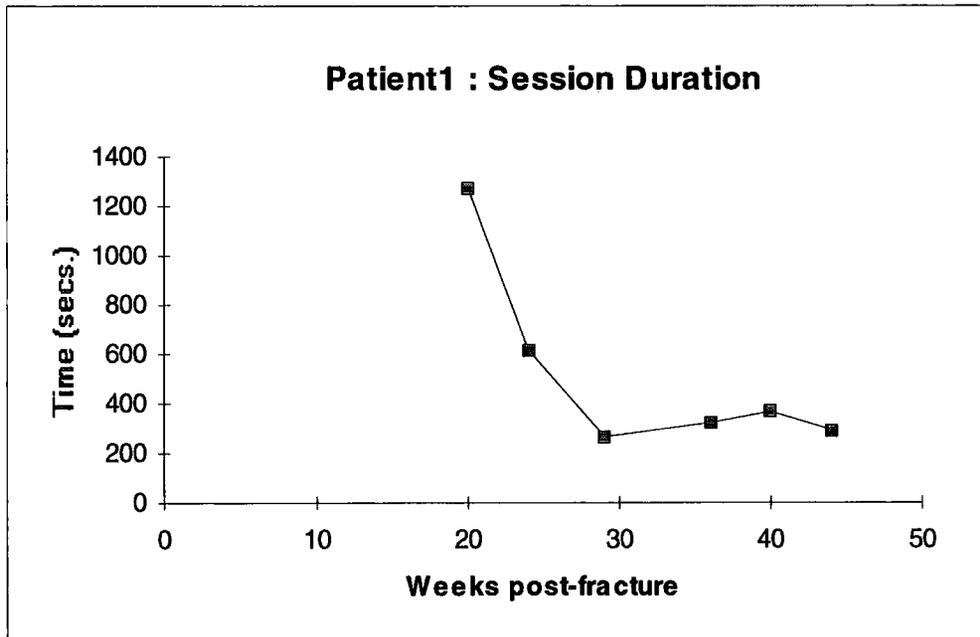
Figure 4.6 - Graph showing the monitoring sessions' duration with time post-fracture for Patient 1
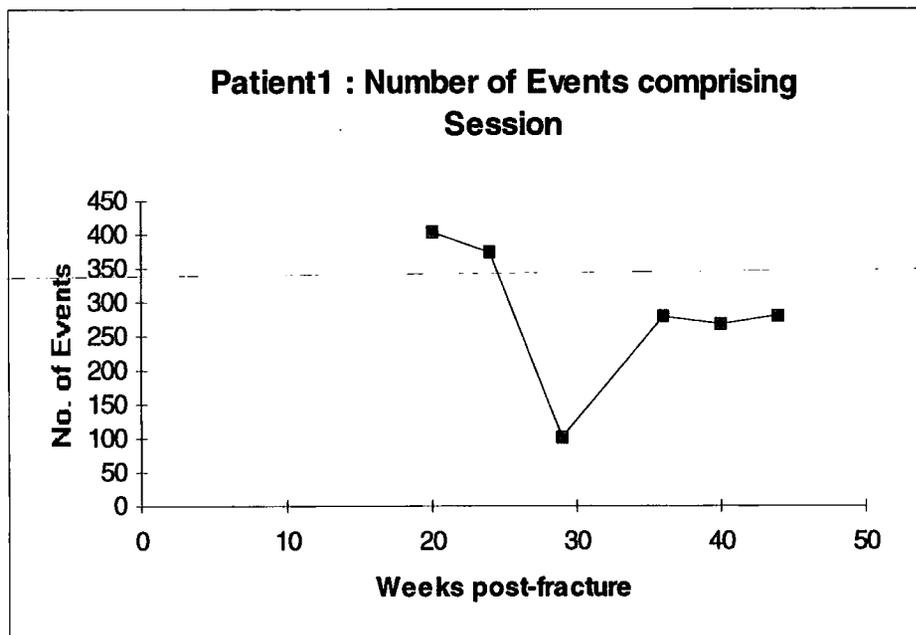
Figure 4.7 - Graph showing the number of events for a session with time post-fracture for Patient 1

4.2.2. Patient 2

This patient was a 61 year old female of 63 kg who was fit and was treated by an external fixator for a comminuted proximal tibial fracture of the left leg. Five monitoring sessions occurred between weeks 37 and 49 post-fracture. The long healing time for the fracture was probably due to it having been initially treated conservatively with a non weight-bearing plaster and due to this not providing satisfactory results, changing the treatment method to an external fixator.

By week 3 post-fracture the X-rays showed that early callus formation was occurring, and at week 6 the plaster was changed to a walking plaster. However at week 10 the X-rays indicated that no bony union had taken place, and although X-rays for subsequent weeks indicated the union as progressing, by week 28 a slight movement at the fracture site was observable even though the radiographs revealed a bridging of bone across the fracture site. Therefore at week 33 an external fixator was applied to provide a more stable and stiffer reduction of the fracture to facilitate its healing.

The monitoring session at week 37 recorded the first time the patient was weight-bearing since the operation to apply the fixator. This resulted in the higher standard deviation reading of Figure 4.8, as the patient was unsteady in her steps. It also resulted in a long average step duration and high standard deviation shown in Figure 4.9, indicating that the patient walked with slower steps and again with more uncertainty and unsteadiness, the relatively small velocity being shown by the long duration of the standard walking circuit in Figure 4.10.

It is interesting to see the weight-bearing value decreasing from 57% to 48% by week 42. This was due to the patient adopting a three-point swing through gait pattern with the crutches (Whittle, 1991), which decreases the weight-bearing as the weight is transmitted through both the legs during the stance phase of the gait cycle, even

though the patient was walking with greater confidence. This is also shown by the large decrease in step duration, and the low standard deviation value indicating that this gait pattern had become habitual to the patient. Figure 4.10 shows that the patient was actually walking with the greatest velocity of all the monitoring sessions that took place.

By week 45 the average weight-bearing had decreased further to 32% of body weight. The pin tracts had become infected two weeks previously which caused discomfort during weight-bearing and also resulted with a limp which had not been noticed previously. To minimise this discomfort and pain the patient therefore automatically reduced the weight-bearing, one indication of this being the adopted limping gait pattern, another being the decreased step duration, as the time when weight-bearing occurred was decreased. This had the effect of decreasing the velocity, shown by the increase in time taken in Figure 4.10, and Figure 4.11 shows the stride length being decreased as more steps were taken to finish the route.

The infection was treated by a course of antibiotics, and the external fixator was removed the following week and the patient again monitored, the equipment indicating an average weight-bearing of 34% of body weight. During this session, the patient walked with the aid of crutches and with a noticeable limp. The removal of the fixator decreased the total stiffness of the leg and so increased the perceived instability of the fracture. This is shown in this instance by the higher weight-bearing standard deviation value, the decreased step duration and the high standard deviation value of the step duration.

The above statements are supported by the step duration increasing at the last monitoring session because the fracture had become more rigid as the healing progressed, the patient herself indicating that the leg felt stronger. Also the weight-bearing standard deviation value is greatly decreased as the gait pattern adopted

changed to being more uniform. However the most conclusive datum is the average

weight-bearing for week 49, which shows a large increase to 52% of body weight.

Nevertheless this is actually a relatively low weight-bearing value, which was due to

the patient becoming accustomed to using the three-point swing through gait pattern

with the crutches which decreases the weight-bearing.
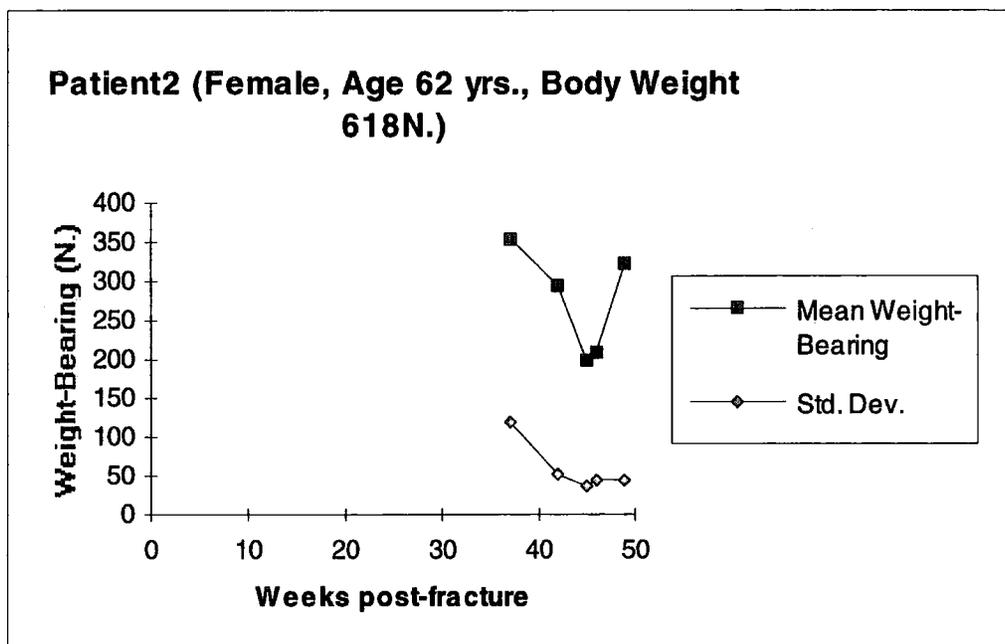
Figure 4.8 - Graphs showing the mean and standard deviation of the weight-bearing with time post-fracture for Patient 2
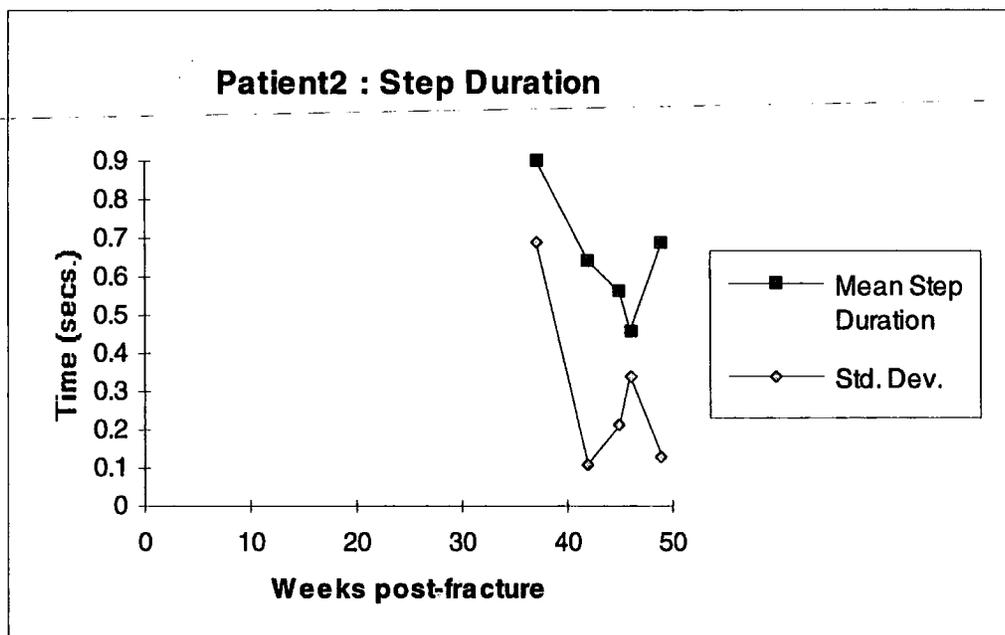
Figure 4.9 - Graphs showing the step duration with time post-fracture for Patient 2
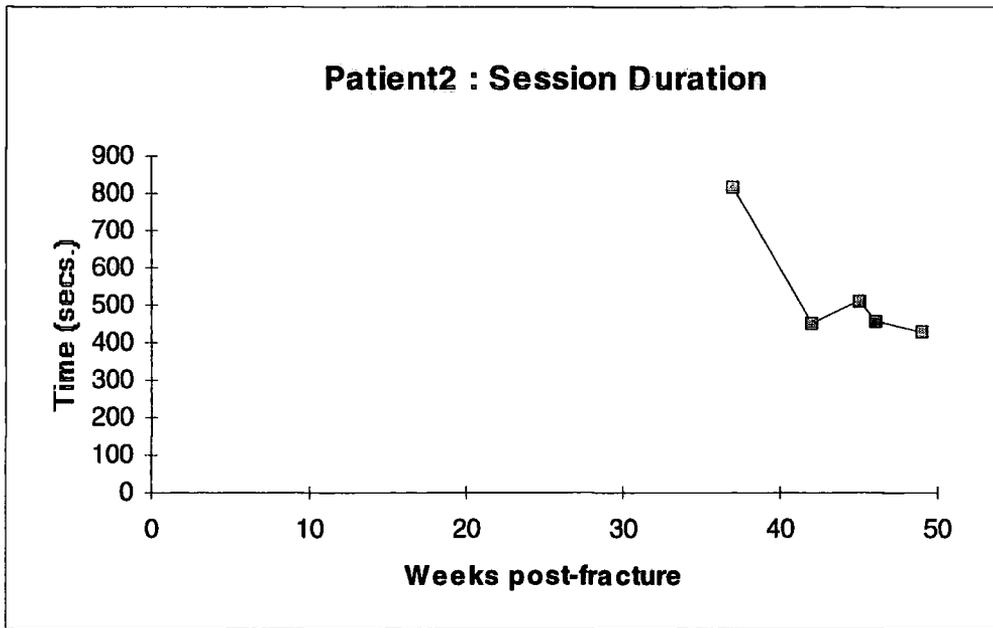
Figure 4.10 - Graph showing the monitoring sessions' duration with time post-fracture for Patient 2
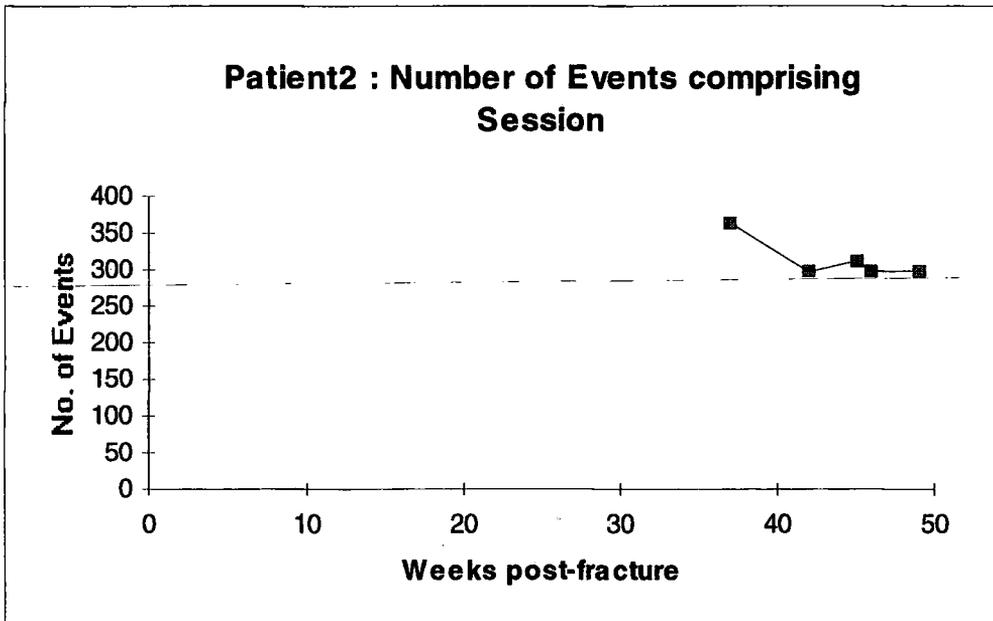
Figure 4.11 - Graph showing the number of events for a session with time post-fracture for Patient 2

4.2.3.   Patient 3

This 43 year old male patient of mass 83 kg was treated with a buttress plate. The fracture was a comminuted displaced mid-shaft fracture of the left tibia. The use of a buttress plate resulted in a fracture which was as stiff or stiffer than other treatment methods (Mow *et al.*, 1991). This effect is shown by the greater weight-bearing recorded at a relatively short time post-fracture. Unfortunately this patient's monitoring sessions occurred at the start of the monitoring period and so only two pressure transducers were used. The mean weight-bearing data from these last two monitoring sessions again illustrate the problems that occur with using only two pressure transducers.

The monitoring session at 5 weeks post fracture recorded the first time the patient bore weight since the time of fracture, an average weight-bearing of 20% of body weight being measured as shown in Figure 4.12. Unlike the other patients, this patient immediately adopted a regular gait pattern as was indicated by the low weight-bearing and step duration standard deviation values. By week 8, when the X-rays showed evidence of fracture union, the patient was walking regularly with the aid of a stick in opposition, recording a reading of 104% of body weight which seems slightly high considering the early stage in fracture healing. By week 15 the patient was walking confidently without the aid of the stick and with a slight limp. The recorded average weight-bearing reading was 72% of body weight.

The fairly constant weight-bearing standard deviation value might indicate that the patient started and continued using their normal gait pattern. This is corroborated by Figure 4.13 which shows that the mean step duration for each monitoring session is fairly constant, at slightly over 0.5 seconds, with the standard deviation being comparatively low at about 0.14 seconds. The first session's higher value is understandable due to it being the patient's first weight-bearing occasion since fracture.

fracture. However the decreases in Figures 4.14 and 4.15 show that a gradual increase in velocity of gait and stride length occurred with time post-fracture, and if the same gait pattern were being used a decrease in the mean step duration should be visible. Therefore some increase in the step duration relative to the gait cycle time must have occurred, but in relation to other patients it was relatively small.
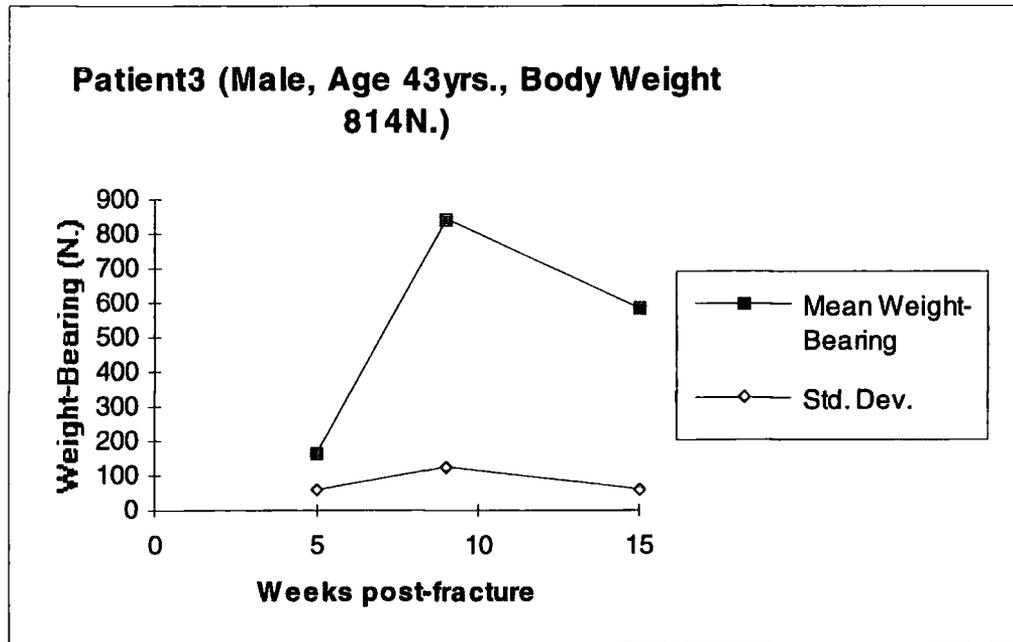
Figure 4.12 - Graphs showing the mean and standard deviation of the weight-bearing with time post-fracture for Patient 3
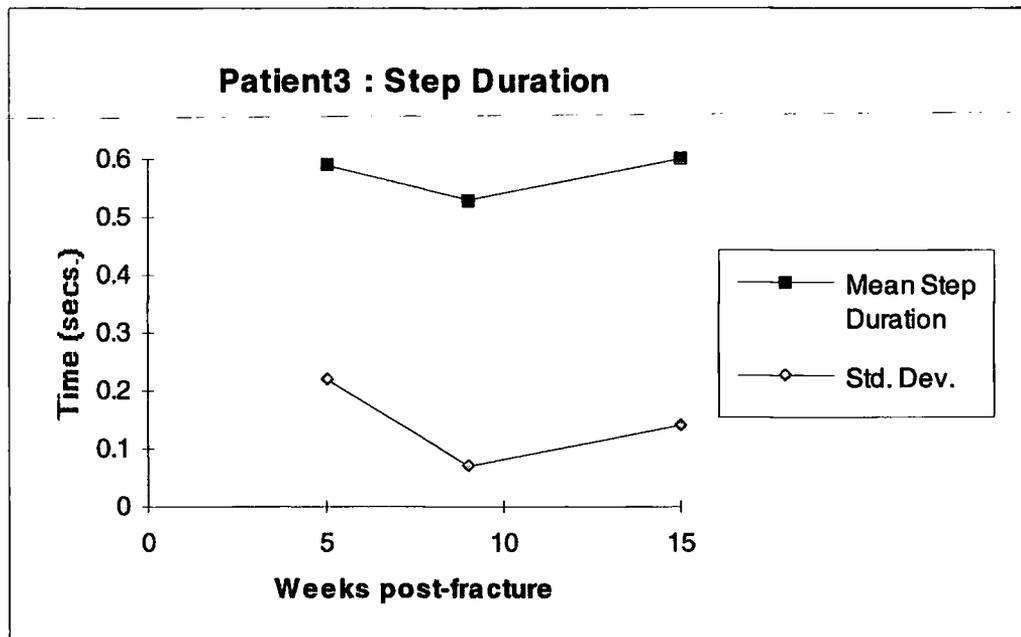
Figure 4.13 - Graphs showing step duration with time post-fracture for Patient 3

Patient3 : Session Duration
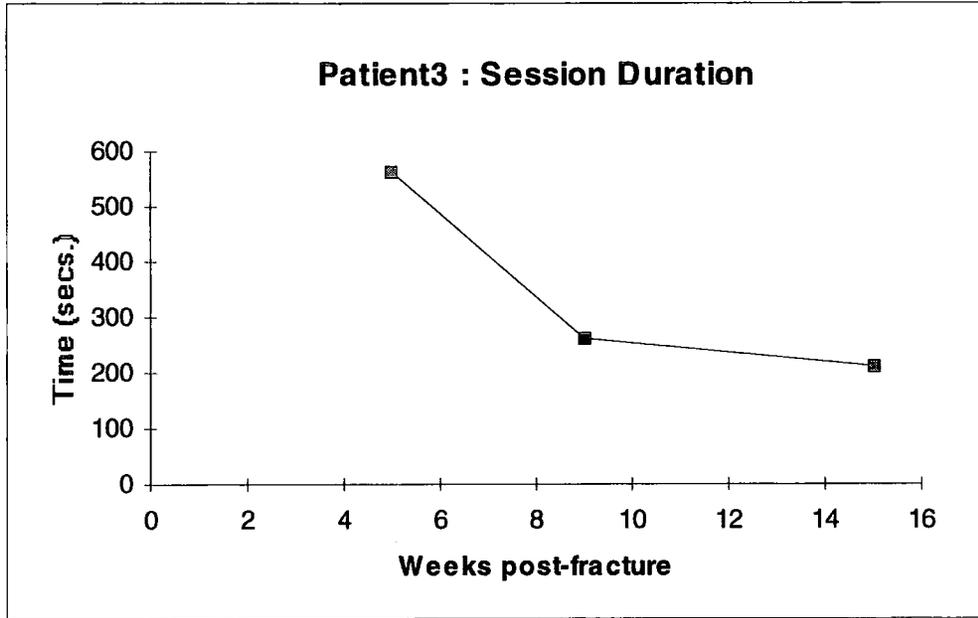
Figure 4.14 - Graph showing the monitoring sessions' duration with time post-fracture for Patient 3

Patient3 : Number of Events comprising Session
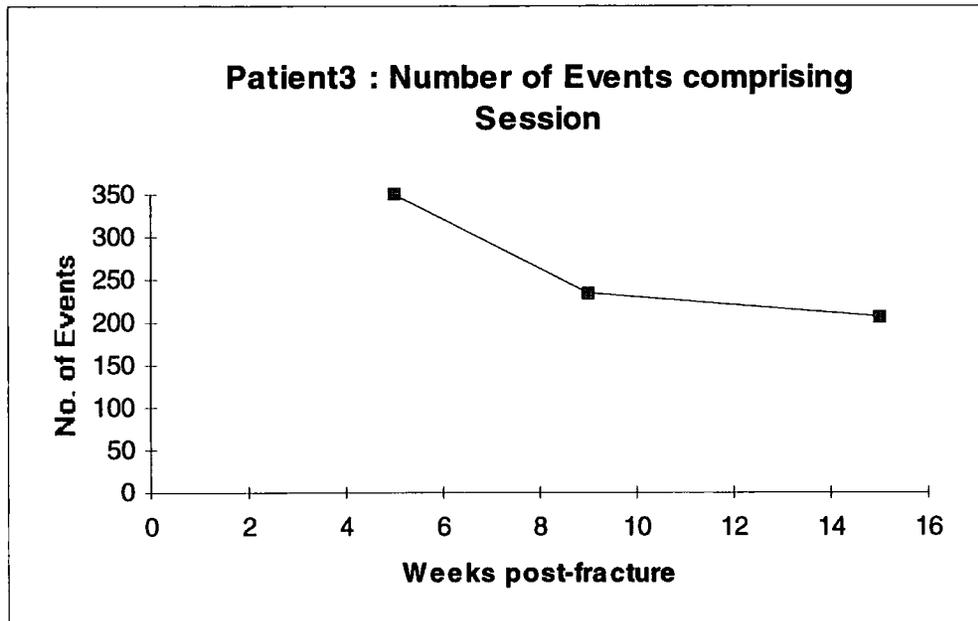
Figure 4.15 - Graph showing the number of events for a session with time post-fracture for Patient 3

4.2.4.   Patient 4

This 74 year old male patient of mass 62 kg had a spiral mid-shaft fracture of the left tibia, and was treated with a compression plate.

By the time of the first monitoring session at week 14, the patient was walking unaided apart from a stick in opposition. The weight-bearing value of 53% of body weight recorded for this session as shown in Figure 4.16 therefore seems low. Examining the actual monitor data reveals that a number of events have a duration of 1 sample. This occurred due to the same software error as for patient 1 at week 29 as these monitoring sessions occurred during the same fracture clinic. This explains the high step duration standard deviation and possibly the low mean step duration in Figure 4.17.

The subsequent monitoring sessions' readings of 95% of body weight at 19 weeks and 90% of body weight at 35 weeks are more reasonable. The decreasing standard deviation values, from about 10 to 5 indicate that the patient was easing into a more normal gait pattern over this period of healing. This view is further corroborated by the average duration for each event increasing over time and the number of events decreasing in Figure 4.19 for the same session duration shown in Figure 4.18. These indicate that the patient was gradually using a more symmetrical and normal gait pattern for the stride length increased by the stance phase of the fractured leg becoming more normal, therefore causing both stance phases of the gait cycle to become more equal in duration.

The relatively high step duration standard deviation at week 35 was due to the patient having to occasionally stop when walking due to obstructing groups of people which explains the higher session duration and higher number of events as the patient had to stand still at various times, these of course being recorded as events. By again

examining the actual monitor data, it can be seen that in fact most of the events'

duration is 5 samples or 0.7 seconds, leading to a low standard deviation value if the

few long duration events are omitted.
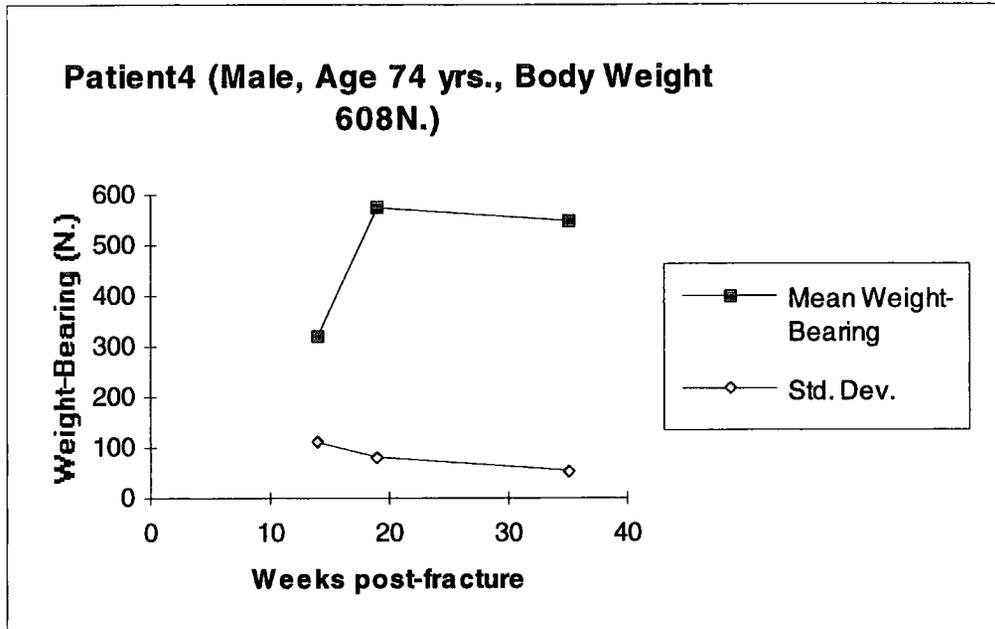
Figure 4.16 - Graphs showing the mean and standard deviation of the weight-bearing with time post-fracture for Patient 4
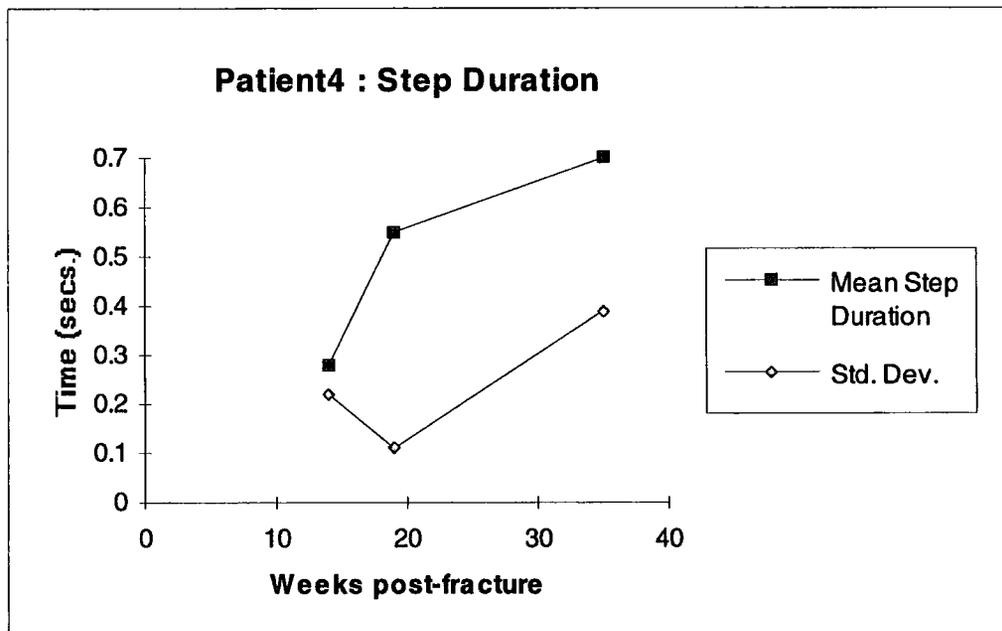
Figure 4.17 - Graphs showing the step duration with the time post-fracture for Patient 4
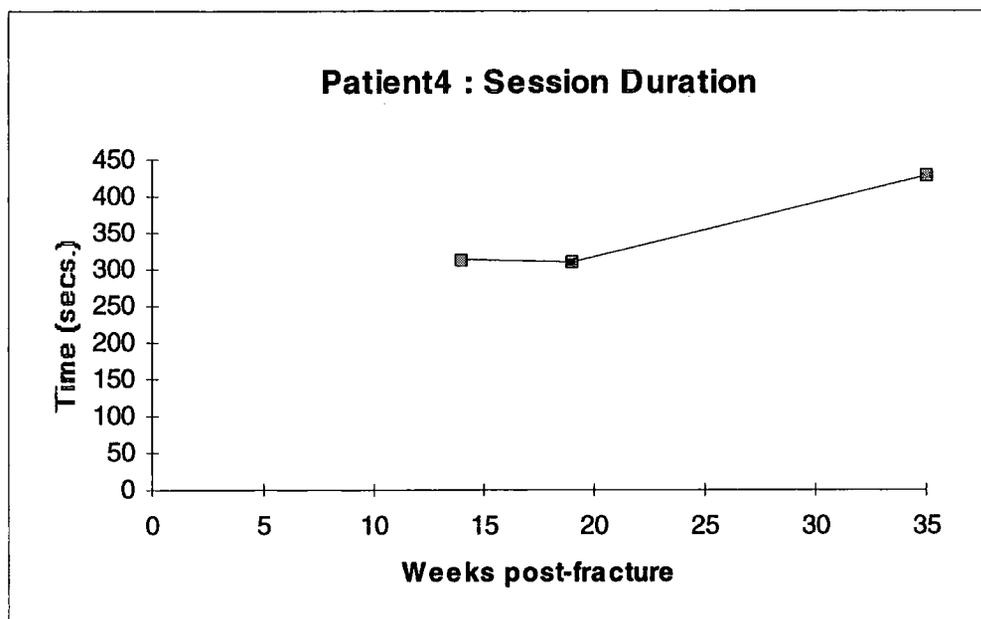
Figure 4.18 - Graph showing the monitoring sessions' duration with time post-fracture for Patient 4
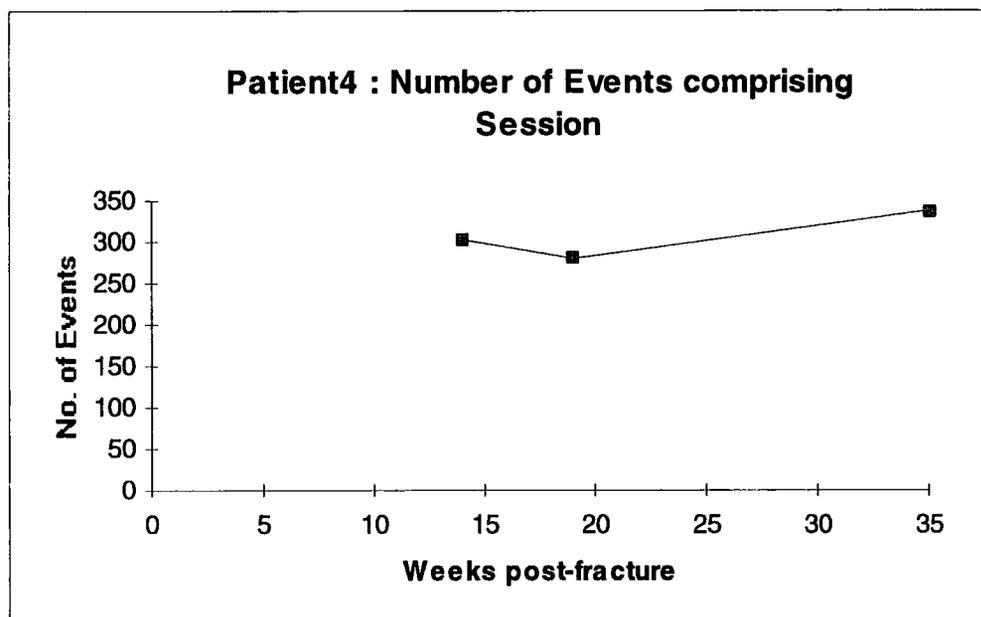
Figure 4.19 - Graph showing the number of events for a session with time post-fracture for Patient 4

4.2.5.   Patient 5

This 17 year old male patient of mass 75 kg sustained a compound comminuted mid-shaft fracture of the left tibia, and was treated with an intra-medullary nail.

For the first monitoring session at week 8 post fracture, the patient had stopped using crutches and sticks, the long leg plaster had just been removed and a brace being worn instead for this and all the subsequent sessions. A 56% of body weight average weight-bearing value was recorded for this session as shown in Figure 4.20, this being the first time the patient was weight-bearing without a plaster. Even though the patient compensated for this with less weight-bearing, the standard deviation values are small, signifying that the patient was confident about walking with minimal unsteadiness in the fractured limb because an unvarying gait pattern was adopted. This is also shown by the step duration mean in Figure 4.21 being fairly constant throughout all the monitoring sessions, and the standard deviation being relatively low, apart from the results at week 24 post-fracture.

However this does not mean that a normal gait pattern was adopted for Figures 4.22 and 4.23 show a decreasing session duration and less number of events for the session. Therefore an increased stride length and relative step duration occurs implying that the stance phase is modified but not to the same degree as with the other patients for the real step duration increases only slightly.

By week 13 the average weight-bearing had increased to 90% of body weight, the patient walking with a slight limp. The final two sessions at weeks 18 and weeks 24 recording average values of 81% and 83% of body weight respectively which seem low even though the interlocking screws of the nail had by this time been removed. The abnormally high step duration standard deviation for week 24 is explained by examining the monitor data, for this reveals the patient stood still a number of times

during the monitoring session, as indicated by the increased session duration. The route normally walked was obstructed and so was cut short, explaining the lower number of events for this session.

The continual increase in step duration, velocity and stride length indicates that as the fracture healed, the patient adopted a more uniform and regular gait pattern for the leg could transmit more weight and was stiffer.
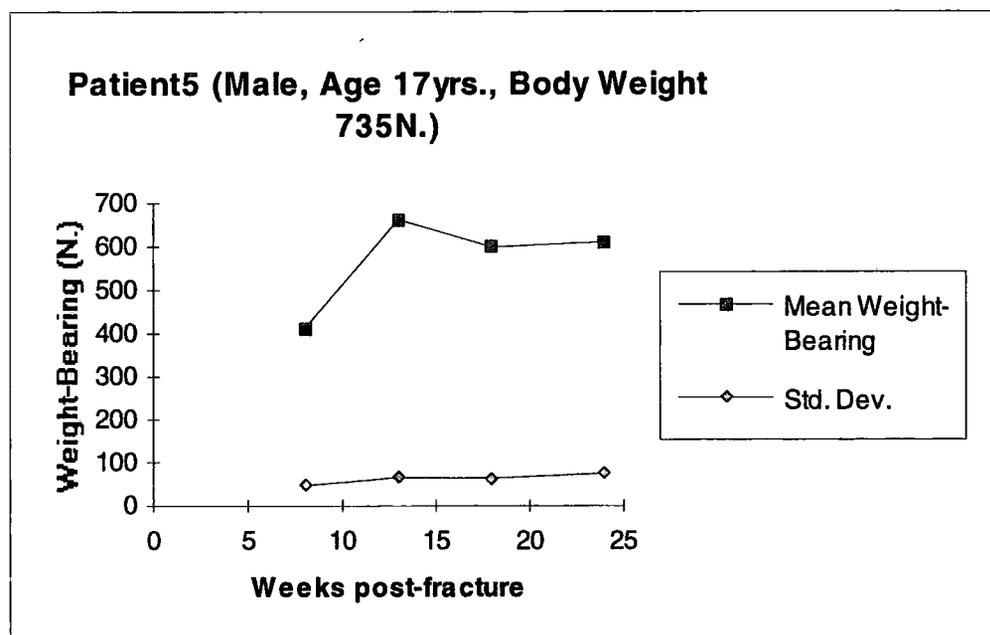
Figure 4.20 - Graphs showing the mean and standard deviation of the weight-bearing with time post-fracture for Patient 5
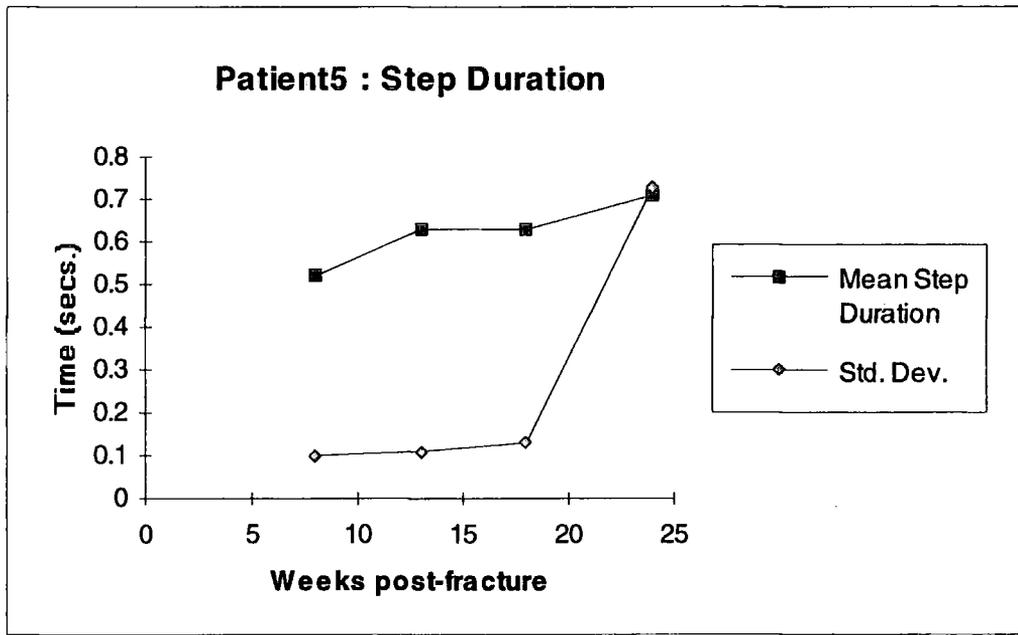
Figure 4.21 - Graphs showing the step duration with time post-fracture for Patient 5
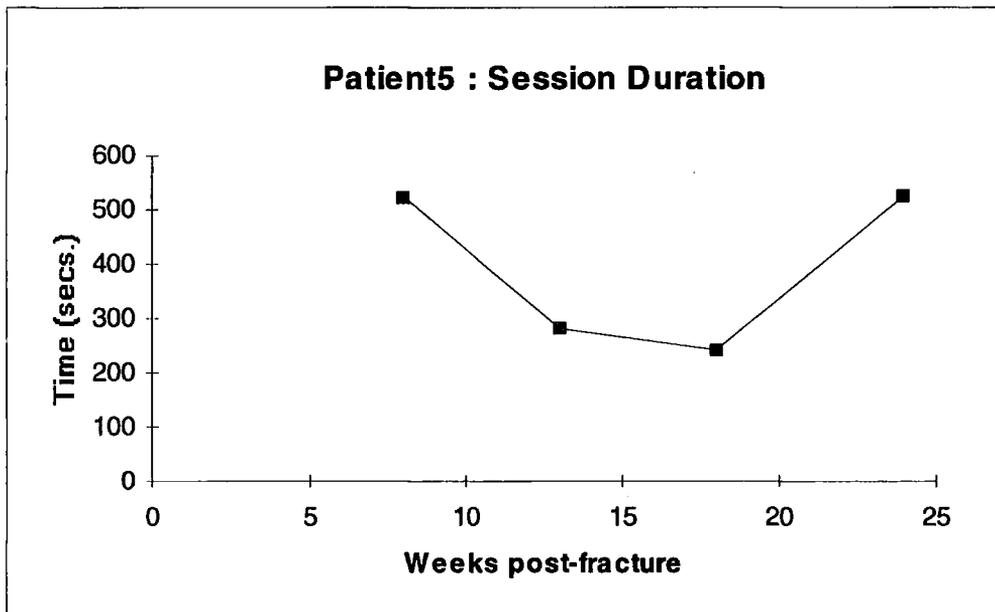
Figure 4.22 - Graph showing the monitoring sessions' duration with time post-fracture for Patient 5

Patient5 : Number of Events comprising Session
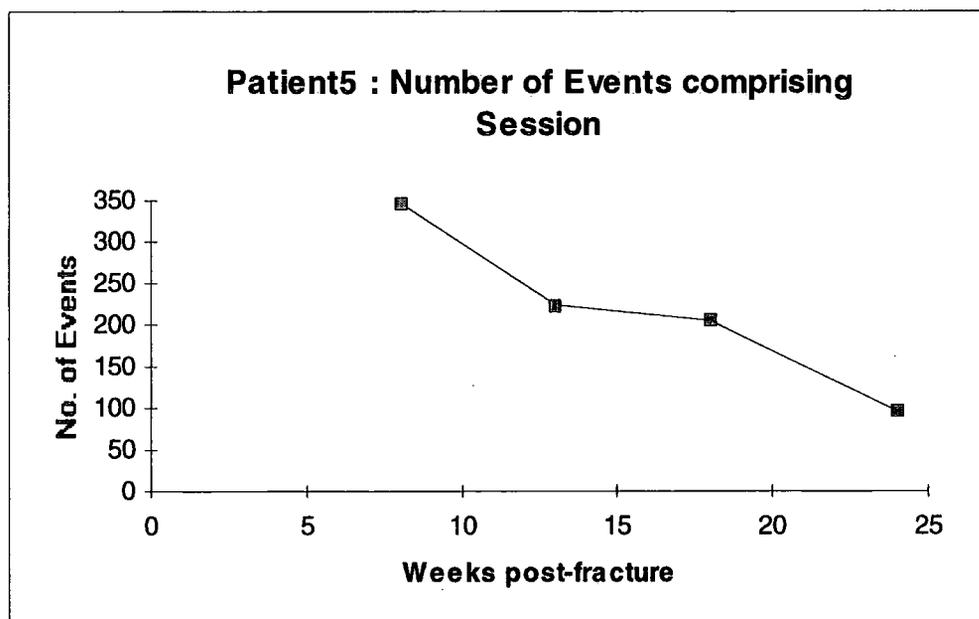
Figure 4.23 - Graph showing the number of events for a session with time post-fracture for Patient 5

4.2.6.   Patient 6

This 27 year old male patient of mass 67 kg sustained a displaced mid-shaft fracture of the left tibia, and was treated with an external fixator.

For the monitoring session at week 6 the patient was using crutches and was weight-bearing for the first time. This is perhaps why a low 34% body weight average weight-bearing in Figure 4.24 and the long average step duration and standard deviation in Figure 4.25 were measured, as the patient walked slowly and carefully using the crutches to minimise the weight-bearing on the fractured leg. This is shown by Figure 4.26, with the carefulness when walking being indicated by the large number of events in Figure 4.27.

At week 10 post-fracture the fixator was dynamised as radiographs indicated that a satisfactory amount of callus was present. The monitoring session for that week recorded an average weight-bearing of 82% of body weight, with the patient walking more quickly and with greater confidence, whilst still using crutches. This is shown by the lower mean step duration and greatly decreased session duration and number of events. Although the patient walked more quickly than Patient 2 on their second monitoring session, as seen by comparing Figure 4.10 with Figure 4.26, the mean duration of each step was slightly higher, indicating that this patient was walking with a more natural and symmetrical gait pattern.

By week 27, the fixator had already been removed a month prior to this session, the radiographs showed evidence of union. For this session the patient was walking normally and without crutches. The low weight-bearing and step duration standard deviations indicate a regular gait pattern, with the relatively long step duration and fast pace of walking adopted indicating a normal gait pattern. This implies that the leg had healed to such a degree that it was of sufficient stiffness to permit (almost) normal loading.
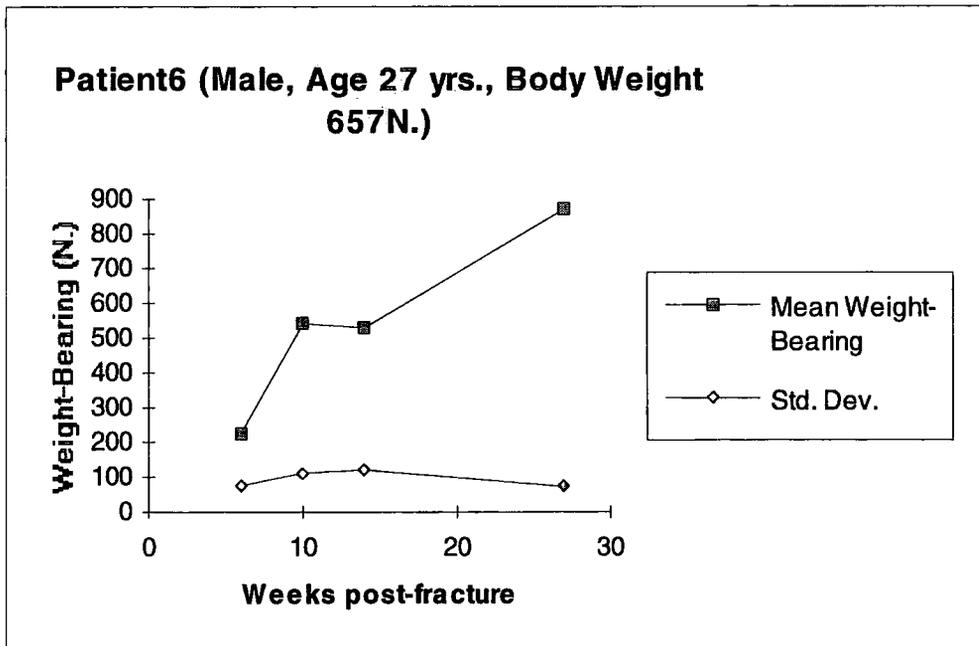
Figure 4.24 - Graphs showing the mean and standard deviation of the weight-bearing with time post-fracture for Patient 6
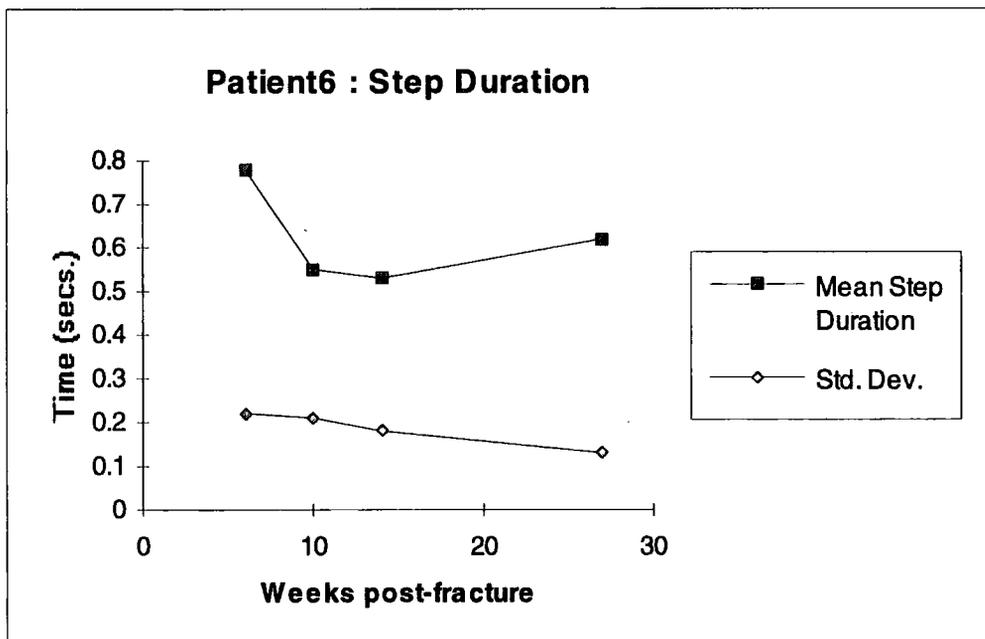
Figure 4.25 - Graphs showing the step duration with time post-fracture for Patient 6
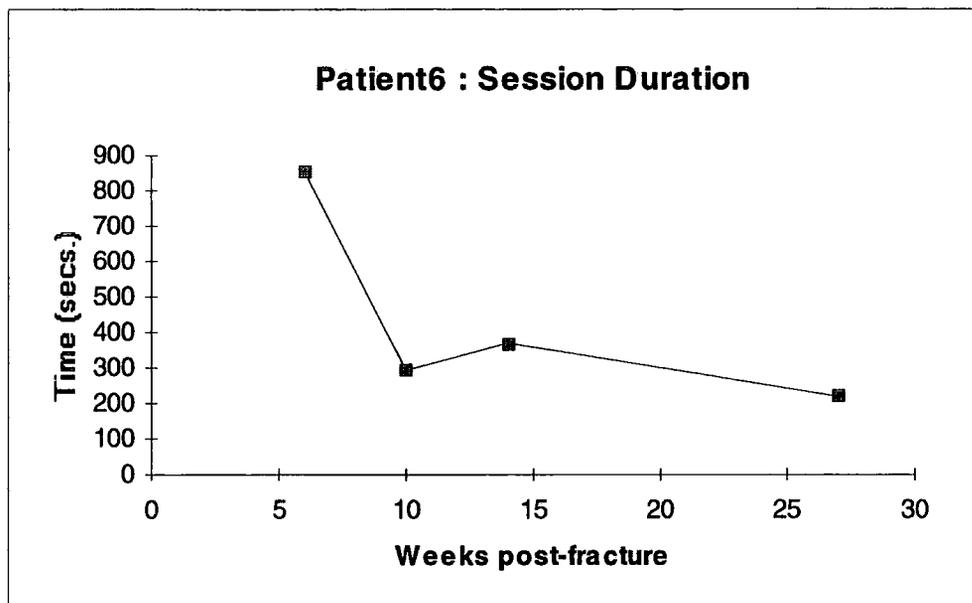
Figure 4.26 - Graph showing the monitoring sessions' duration with time post-fracture for Patient 6
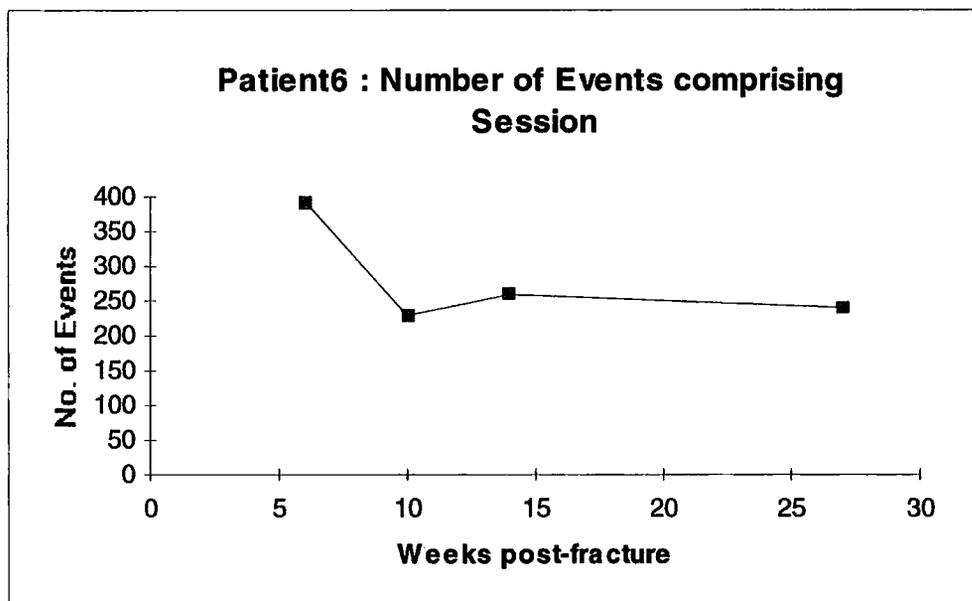
Figure 4.27 - Graph showing the number of events for a session with time post-fracture for Patient 6

4.2.7. Patient 7

This 60 year old male patient of mass 85 kg was treated with an external fixator for a non-union of a previous compound comminuted fracture.

For the first monitoring session at 17 weeks post fracture the patient had the external fixator removed but was still using crutches. This accounted for the low 57% of body weight weight-bearing average recorded and shown in Figure 4.28. The higher weight-bearing and step duration standard deviations shown in Figures 4.28 and 4.29 are probably due to the patient not being confident in weight-bearing on the fractured leg, due to the fixator having been dynamised just before the monitoring session. The higher session duration in Figure 4.30 and greater number of events for that session in Figure 4.31 support this.

The crutches were discarded by week 19, with a corresponding increase in the average weight-bearing to 91% body weight. By week 25, the average weight-bearing was 94% body weight with a corresponding increase in the average duration for each event as the patient resumed a normal gait pattern, as shown by the higher walking speed and stride length.
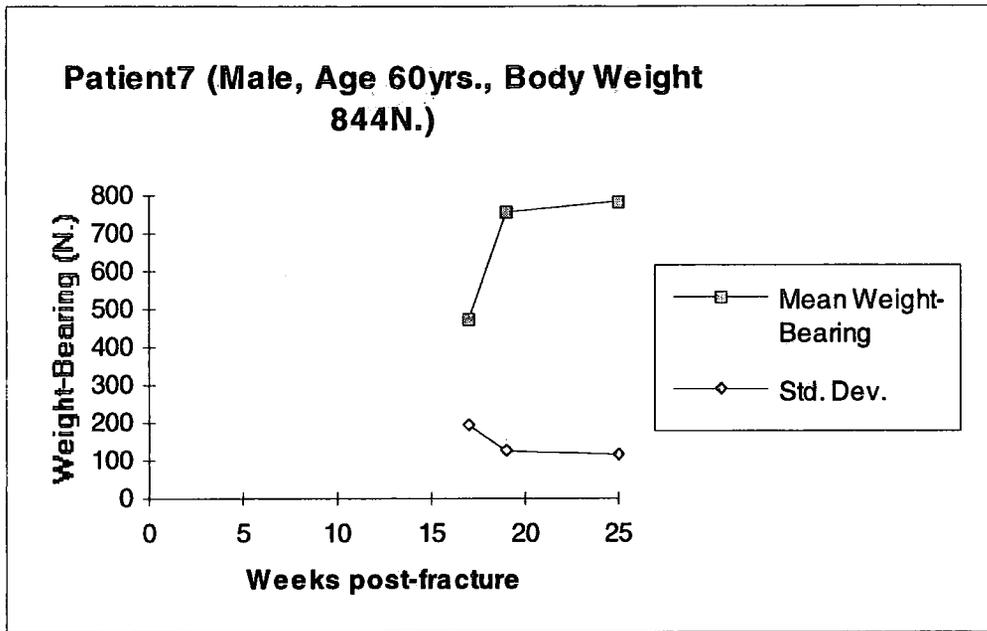
Figure 4.28 - Graphs showing the mean and standard deviation of the weight-bearing with time post-fracture for Patient 7
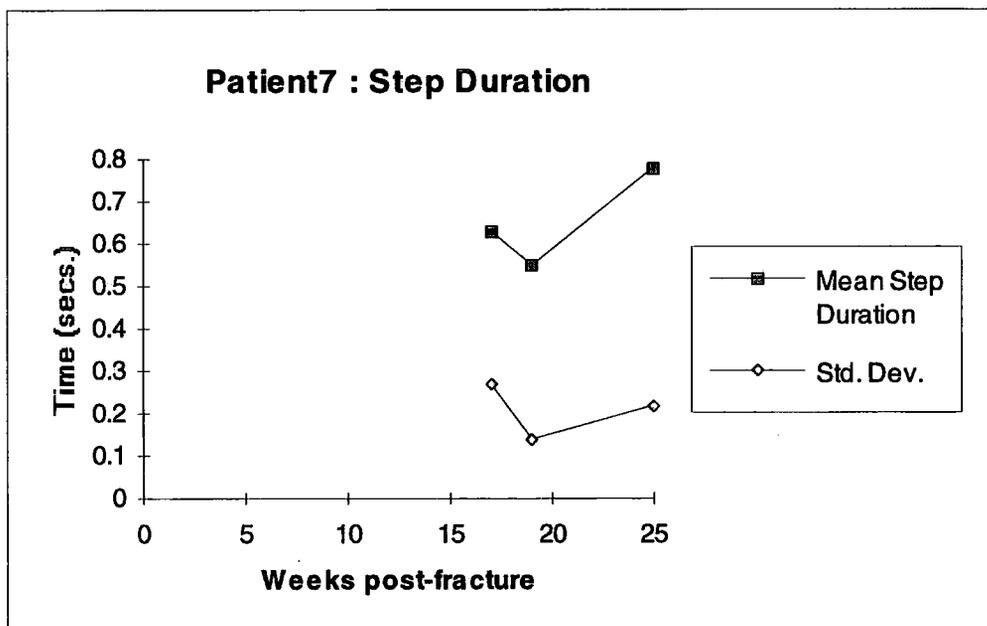
Figure 4.29 - Graphs showing the step duration with the time post-fracture for Patient 7
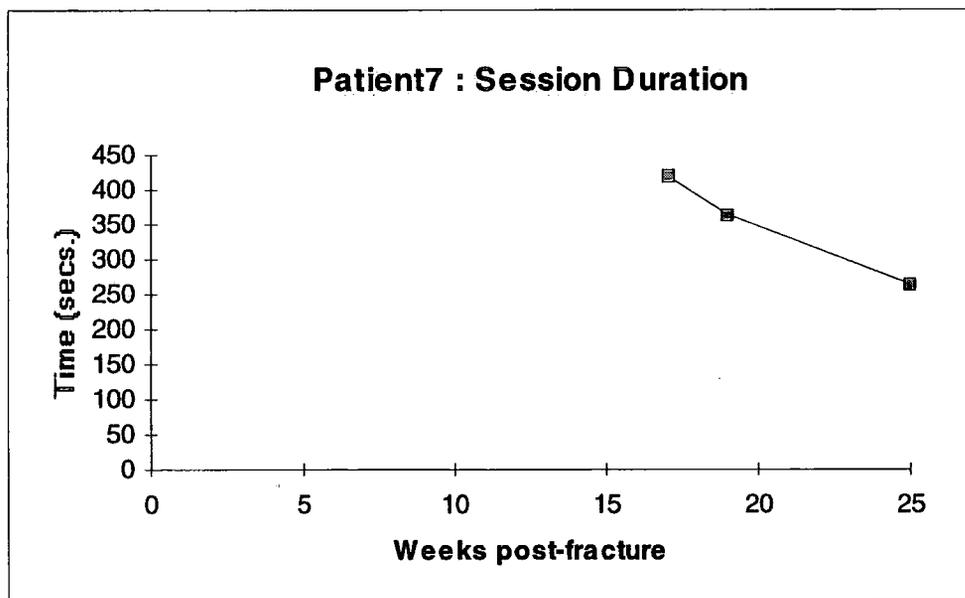
Figure 4.30 - Graph showing the monitoring sessions' duration with time post-fracture for Patient 7
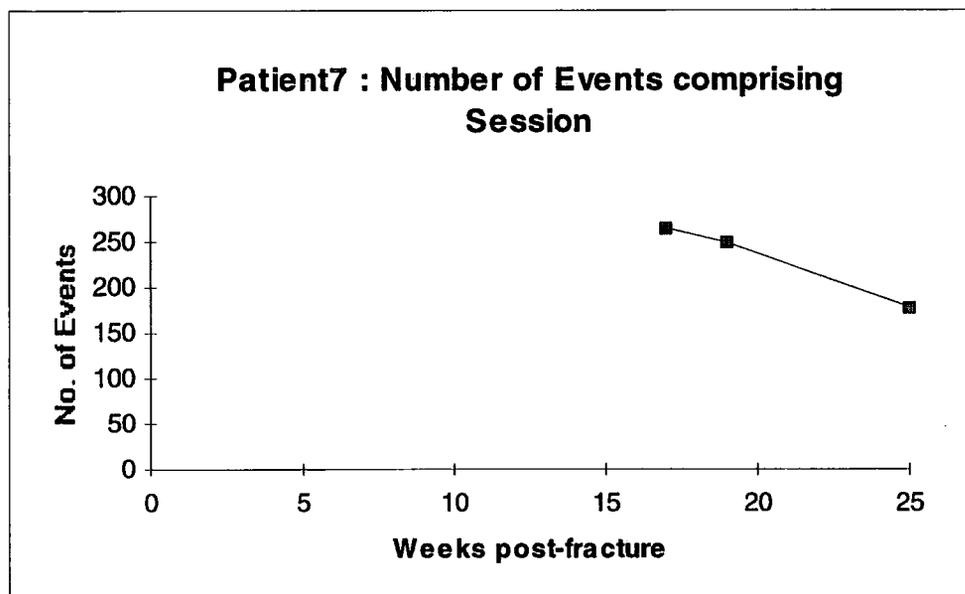
Figure 4.31 - Graph showing the number of events for a session with time post-fracture for Patient 7

4.3.   Group Results

It was first envisaged that group result bar charts could be constructed for the different categories of fracture and fracture treatment method, however the lack of patient data precluded this occurring. Therefore only the general results and the children's, adults', and elderly adults' results follow. The results are presented as the mean of all the patients' average weight-bearing value, with the standard deviation of this mean also being shown.

4.3.1.   General Patients' Results

Figure 4.31 below shows the mean weight-bearing results for all the patients regardless of age, sex, treatment method or type of fracture. Even though these factors which affect the rate of fracture stiffness over time and so weight-bearing are not considered, it can be seen that there is a general trend of increased weight-bearing up to week 28 post-fracture.

Keeping in mind the fact that only a small data set was obtained therefore possibly causing abnormalities in the group results, the following observations from the results are noted. The increasing weight-bearing over time does not seem to be linear with time post-fracture. Weeks 4 to 14 post-fracture seems to indicate a linear increase in weight-bearing. Week 14 to 24 seems to show a fairly constant mean weight-bearing value, with weeks 24 to 28 showing a large linear increase in mean weight-bearing. Week 36 onwards show the results for patients with delayed fracture union, with an increase in mean weight-bearing over time being observable.

Biologically, these times post fracture tie in with certain stages of fracture healing. Week 14 occurs in the latter half of the mineralisation phase. It seems reasonable that

when the fracture increases in stiffness due to the callus being progressively calcified that a progressive increase in weight-bearing should occur. Week 14 to 24 is when the majority of bone remodelling occurs, so a fairly constant weight-bearing value seems reasonable for a stage where old bone is being resorbed and new Haversian bone laid instead. Week 24 to 28 is possibly when the Haversian bone along the lines of force fully unite the fracture ends and so greatly strengthen and stiffen the fracture site, by this stage the fracture being said to be healed.
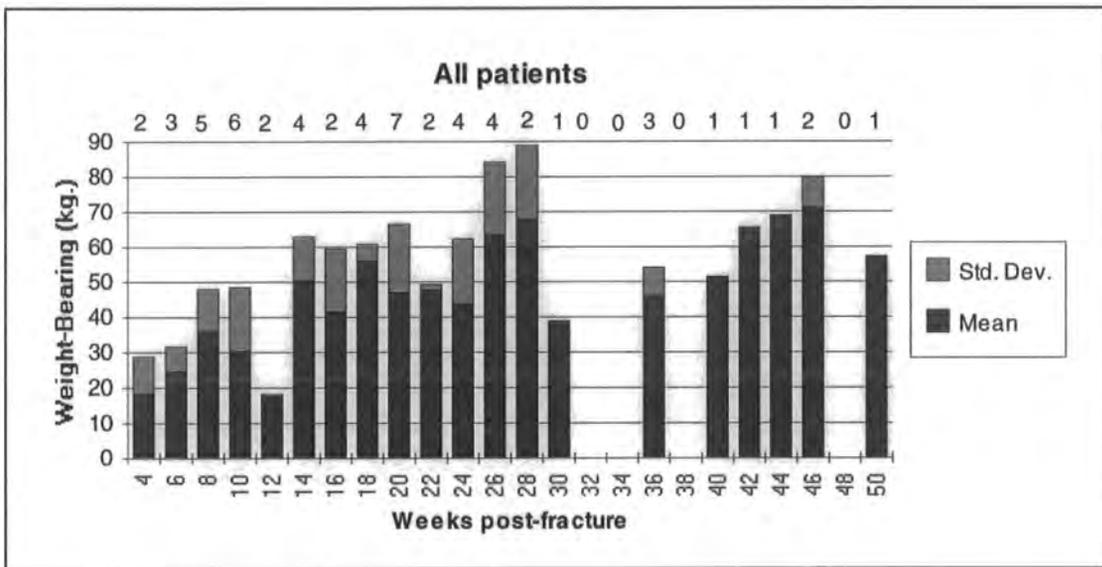
Figure 4.31 - Weight-bearing over time post-fracture for all the patients. The number of patients indicated at each week is given above the each column.

4.3.2. Children Patients' Results

The data obtained for children up to 16 years is shown below in Figure 4.32. Due to the small sample set no observations about general trend can readily be made. What can be observed however is that the fractures healed more quickly than with the adult patients, even though all the children's fractures were treated conservatively with a plaster. Even though this provides less support and so causes less overall stiffness at the fracture site, these young patients also bore significant percentages of body weight much earlier than the adult patients.
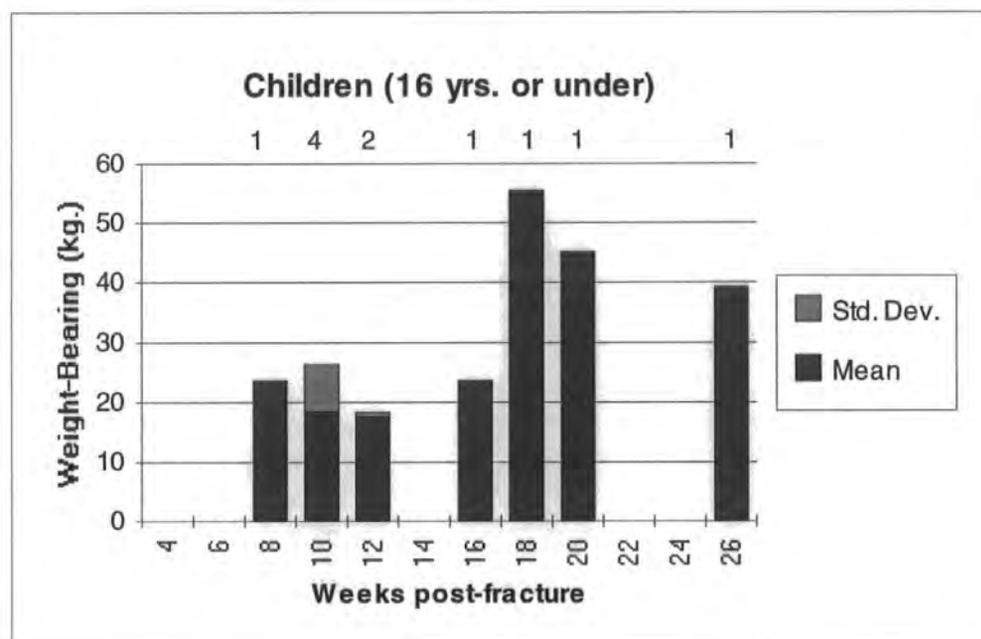
Figure 4.32 - Weight-bearing over time post-fracture for the patients 16 years old or under.

4.3.3.   Adult Patients' Results

The mean weight-bearing results over time for the adult patients between 16 and 55 years old are shown in Figure 4.33. As with the general results, it can be observed that there is an increase in weight-bearing up to week 14, this being similar to a previous study (Cunningham *et al.*, 1989). A fairly constant mean weight-bearing value occurs from week 14 to week 24, and a large increase in mean weight-bearing from week 24 to week 28. The similarity in trends between these and the general results might be expected however, for the bulk of the general results are composed of these adult ones. This factor is evidenced by the trend described with the general results being shown more clearly here.

However rather than with there being a linear increase in weight-bearing up to week 14, an increase showing a decreasing positive gradient is noticeable. This is more reasonable than a linear increase, for whatever the treatment method, initially a greater increase in rigidity occurs with lesser subsequent increases. This is because the fixation method will give an initial stiffness at the fracture site, with a slow gradual increase in stiffness from then on as calcification of the callus occurs. This increase in stiffness was shown by Richardson *et al.* (1992) to be exponential with time post-fracture. Due to the main feedback mechanism limiting the weight-bearing being pain or discomfort which is governed by the amount of movement at the fracture site, one might expect there it be a close correlation with fracture stiffness and weight-bearing over time post-fracture. Although the sample set is too small to make definitive empirical deductions regarding this hypothesis, one can see from Figure 4.33 that an exponential increase in weight-bearing after offsetting the weight-bearing possible due to the fixator stiffness is not at great variance from the recorded results.

The results for weeks 42 onwards are for patients who were treated conservatively with a plaster the fracture subsequently ending with delayed union and requiring another fixation method.
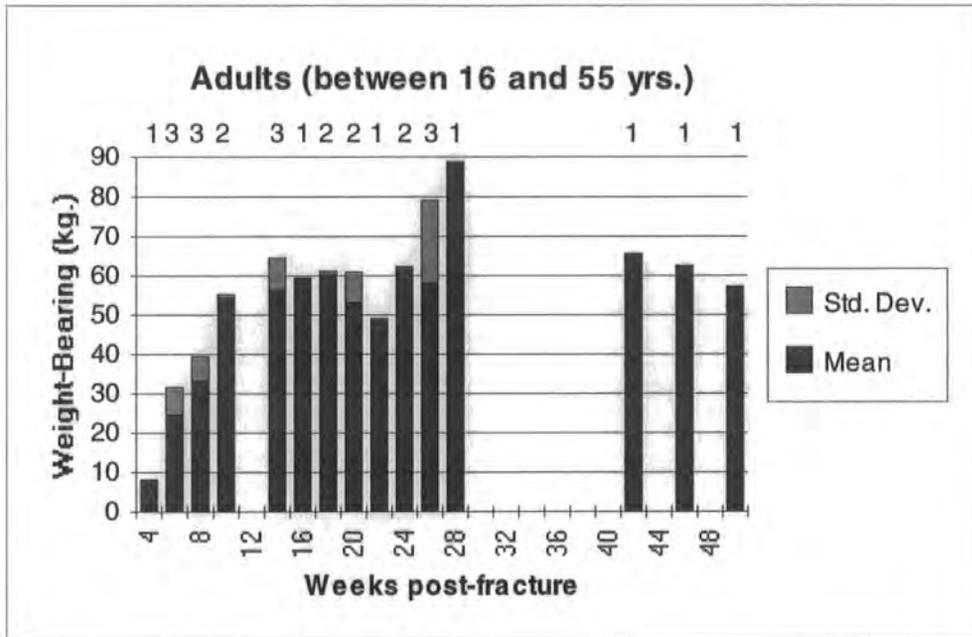
Figure 4.33 - Weight-bearing over time post-fracture for the patients between 16 and 55 years of age.

4.3.4.   Elderly Adults' Results

The results for the adult patients aged 55 years and over are shown in Figure 4.34. As with the children's results, there are too few elderly adult results to be able to make mean weight-bearing trend observations. However what is noticeable is that the fractures seem to take longer to heal for there is a spread of similar weight-bearing results throughout all the weeks post-fracture recorded.
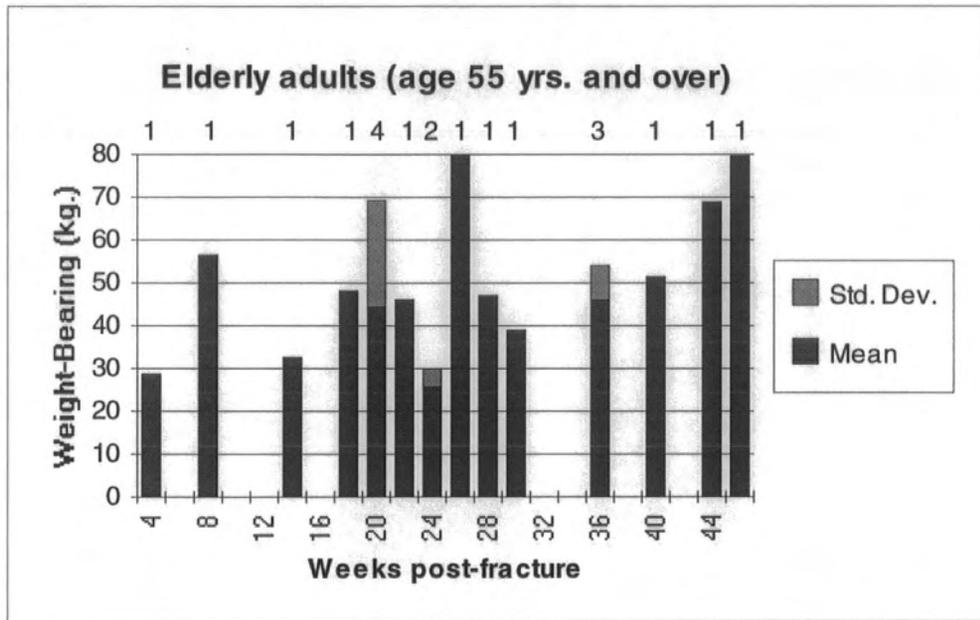
Figure 4.34 - Weight-bearing over time post-fracture for the patients aged 55 years and over.

5.   Discussion and Conclusions

This chapter is composed of three sections; discussions regarding the global aspects of the results obtained, the clinical benefits of using ambulatory monitoring for weight-bearing, and the scope for further development and applications of the monitoring system.

5.1.   Discussion of the Results

The ambulatory monitor software performed as desired, but when the monitor was operated during the software's development, the data obtained would occasionally be inaccurate. The inaccuracies have been detailed in the previous chapter.

5.1.1.   Step Duration

A patient with a fractured tibia should walk with a pathological gait pattern. This is because the patient will attempt to minimise the weight-bearing through that leg. As has been explained in Chapter 3, this in general takes the form of the stance phase for the fractured leg omitting, or greatly modifying, the normal heel contact, foot flat, and heel off, toe off stages. Since the ground reaction force magnitude is greater at these stages, if the patient omits or modifies these stages then the weight-bearing is consequently reduced. This is because during these stages the centre of gravity of the trunk rises from its lowest to its highest position, and this raising of the trunk results in an increased load through the leg. This pathological gait pattern has visible

characteristics, for example the knee flexes upon entering and during the mid-stance stage to minimise the trunk rising, which is observed as a 'limp'.

As the stance phase for the fractured leg will have some stages omitted, its duration will be less than with the non-fractured leg's stance phase. During the healing process the fracture gradually becomes stronger and stiffer, so causing the pathological gait pattern to change towards a more normal gait pattern, therefore extending the stance phase time for the fractured leg towards that of the other leg. During the early stages of fracture healing the patient probably walks carefully and slowly, feeling instabilities in the fracture leg, but as healing progresses the patient changes to a higher cadence, so causing the actual step duration to remain constant or possibly decrease. Hence, during healing, the step duration for the fractured leg increases provided the cadence does not increase too much during the healing period.

Group step duration column charts have not been included in the previous chapter because it was felt that due to the large variations of cadence in the normal gait patterns of different subjects, changes in an individual's step duration could be obscured by the varying cadences of others. Hence one must look to the individual patient results to ascertain whether the above hypothesis is empirically justified.

Table 5.1 shows both the step duration and the session duration for the first and last monitoring session of each patient. Examining only the step duration, it can be seen that in the majority of instances there is an increase over the fracture healing period. This is not however true for Patient 2, 3 and 6 where there seems to be a constant or decrease in the step duration. Therefore one cannot only look at the step duration but must take into account the gait velocity to assess the step duration relative to the step duration of the other foot. As it has not been possible to monitor this during the study, an indication of this relative step duration can be gained by examining the session duration, since a constant step duration with a decreased session duration means an

increased relative step duration; therefore the session durations have also been included in Table 5.1.

Patient	1	2	3	4	5	6	7
First Session at Week post-fracture	20	37	5	14	8	6	17
Step Duration (secs.)	0.3	0.9	0.6	0.28	0.5	0.78	0.63
Session Duration (secs.)	1272	816	562	314	524	852	420
Final Session at Week post-fracture	44	49	15	35	24 (18)	27	25
Step Duration (secs.)	0.6	0.7	0.6	0.7	0.7 (0.65)	0.6	0.8
Session Duration (secs.)	294	428	212	428	526 (242)	240	264

Table 5.1 - Table showing the step duration and session duration over time

It can be seen that in almost every case there is a large decrease in session duration, with five patients showing at least a halving in the time taken to complete the standard circuit. As Patient 5 had a non-standard last monitoring session, having to stand a number of times and walking a shorter route, if the results from the second from last monitoring session are used instead (as shown in brackets) we find that this figure rises to six out of the seven patients. To obtain a quantitative measurement for the step duration change relative to the un-fractured leg's step duration, the following formula was used:

$$RSD = \frac{relativeStepDurationChange}{relativeSessionDurationChange}$$

relative in the calculation meaning relative to itself, for example:

$$relativeStepDurationChange = \frac{NewStepDuration}{OldStepDuration}$$

Table 5.2 shows this quantitative figure which combines the effects of both the step and session duration. This clearly shows that over the fracture healing period there is an increase in the relative step duration indicating that the shorter stance phase relative to the normal, non-fractured, leg increases in relative duration, therefore becoming more like the normal leg's in duration and characteristics. In cases where the monitoring began when patients were just beginning to weight-bear, as with patients 3, 5, and 6, the relative change over the healing period was a lengthening in relative step duration of over 2.5 times.

Patient	1	2	3	4	5	6	7
Relative change	8.7	1.5	2.7	1.8	2.8	2.7	2

Table 5.2 - Table showing the relative changes in step duration for each patient

5.1.2.   Stride Length

Stride length is linked with the relative step duration, a more normal gait pattern producing a greater stride length, as including the heel contact and toe off stages enables the leg to be extended to a greater degree in the swing phase. Therefore it would seem reasonable that an increased relative step duration should be coupled with an increased stride length.

During this study stride length was not measured directly. However an indication of stride length can be gained by examining the number of events recorded for the monitoring session, this being the number of steps taken by the fractured leg while completing the standard circuit, and thus the number of gait cycles needed to complete

the circuit. Table 5.3 shows this for each patient. Patients 4 and 5 are indicated in brackets since the data from the second from last session was used instead of the final one due to the final one being corrupted by obstructions where the patient had to stand occasionally in one case, and a shorter route walked in the other. As the number of events is inversely proportional to the stride length, by inverting the figures in Table 5.3 an indication of the stride length is obtained, these data being shown in Table 5.4.

Table 5.4 shows that for each patient there is an increase in stride length over the healing period, up to a maximum of a 70% increase. However data is not well correlated with the step duration data given in Table 5.2.

Patient	1	2	3	4	5	6	7
Relative change	0.7	0.82	0.59	(0.92)	(0.59)	0.56	0.67

Table 5.3 - Table showing the relative changes in the number of events per sessions for each patient

Patient	1	2	3	4	5	6	7
Relative change	1.43	1.22	1.7	(1.09)	(1.7)	1.79	1.49

Table 5.4 - Table showing the relative changes in the stride length for each patient

Whilst it can generally be said that the greater is the increase in step duration the greater is the increase in stride length, this does not apply to Patient 1. This is probably due to the effect of using crutches where a swing through gait can be employed, so causing a greater stride length with a small step duration.

5.1.3.   Weight-Bearing

The main applicationary object of this study was to monitor weight-bearing changes over the period of fracture healing. It was hypothesised that a gradual increase in weight-bearing would occur over the period, as the fracture site became stiffer and stronger as healing progressed.

As the individual and group results show, a gradual increase in the weight-bearing did occur over the healing period, this increase generally being non-linear. It has been previously speculated that the reason for this is that the increase in stiffness at the fracture site is non-linear over time, as was experimentally demonstrated by Richardon *et al.* in 1992, who found that, discarding the stiffness of the fixation device, the stiffness at the fracture site increased exponentially with time post-fracture.

This gradual increase in weight-bearing over time post-fracture leads to the conclusion that in the absence of pain or discomfort to the patient, there is another feedback mechanism regulating the amount of weight-bearing through the fractured leg. The results of this study certainly indicate that another feedback mechanism is active, but without having obtained measurements local to the fracture site, for example movement at the fracture site during weight-bearing, these results do not lend direct support to the above hypothesis. Perhaps this feedback mechanism is linked with the interfragmentary strain, for if this is too high, then healing cannot progress. This is often seen by resorption at a small fracture gap to allow granulation tissue and callus to form in a lower-strain environment (O'Sullivan *et al.*, 1989).

A progression in fracture healing does not however necessarily mean an increase in weight-bearing, as Figure 4.7 for Patient 2 shows. The mean weight-bearing for the second monitoring session was lower than that measured previously because the patient had adopted a three-point swing through gait pattern even though the velocity

of gait was increased as shown by Figure 4.9. Even the decreased step duration shown in Figure 4.8 was not visible to the observer for the patient walked without a limp or noticeable discomfort. Three weeks later the monitoring session recorded a mean weight-bearing even lower, for a pin tract infection had occurred which caused discomfort to the patient when weight-bearing thus causing the patient to lower weight-bearing to minimise this discomfort. However when a course of antibiotics had healed the infection, a much higher weight-bearing average was recorded. Even so, this was thought to be low because of the gait pattern adopted by the patient. Removing the crutches and allowing the patient to walk with sticks could have forced a higher weight-bearing value.

5.2. Clinical Benefits of using Ambulatory Monitoring for Measurements

The measurement of weight-bearing via ambulatory monitoring can aid the clinician in forming an assessment of the fracture healing by providing further information on limb function which is not available from radiographs. Although this study's aim was not to provide a method to assess the stage of fracture healing, the results from the monitoring of weight-bearing indicate that such data might be useful in such assessment. However as extensive experience in data interpretation are required to give advice based on such measurements, X-rays or other investigations will always be needed for a final decision (Bergmann *et al.*, 1990).

However although some indication of the healing is given, such data really shows the patient's ability to weight-bear on the fracture, this being a combination of conscious and sub-conscious awareness. Normally the only information from the patient that can be gleaned regards pain, discomfort, and unsteadiness. However these are rather subjective, and with no quantitative data the clinician is often forced to conjecture as to the state of the fracture with the aid of previous experience of the length of time normally required for the fracture to heal. Obtaining a quantitative measurement from the patient allows the clinician to include this to aid his assessment of healing. As ambulatory monitoring is non-invasive, data can be quickly and easily obtained from the patient. This data might enable the clinician to provide a more effective course of mobilisation for the treatment of the fracture, for example with the use of crutches, sticks or neither. An example of this was shown with the results obtained for Patient 2, who although was walking quicker and with a greater stride length by week 42 post-fracture, was actually weight-bearing less due to the gait pattern adopted with the crutches. The lower weight-bearing is contrary to expectations, for an increased velocity in walking normally results with an increase in weight-bearing (Jahnke *et al.*, 1992). Therefore the decrease is due to the change in gait pattern, for differing gait patterns result with differing weight-bearing (Olsson, 1992). However the effect of

this gait pattern was not easily visible to the observer and so in such cases the patient might be encouraged to continue in the same manner because of the increase in patient confidence suggested by the higher gait velocity. A more effective treatment however would be to force the patient to produce a greater weight-bearing by perhaps replacing the crutches with two sticks.

Using ambulatory monitoring to measure the relative step duration might also aid the clinician as this also gives an indication of the progress of fracture healing. This is because with a longer step duration relative to the unfractured leg's step duration, the patient will have a greater weight-bearing average, so it is in effect an estimate of the weight-bearing. Ambulatory monitoring of the relative step duration is easier and more accurate than the weight-bearing directly, for accurate weight-bearing requires a greater number of transducers which have to be accurately placed at the load bearing areas of the foot. However changes in relative step duration can only indicate a change in weight-bearing not its magnitude, and so such data on its own might not be so helpful to the clinician. Therefore the direct monitoring of weight-bearing and relative step duration would be the best solution because the relative step duration data also gives useful indications of the gait pattern.

5.3. Possible Future work on the Further Development and Application of the System

To increase the accuracy of the weight-bearing data obtained from the monitor, more pressure transducers are required to be placed at the load bearing areas of the foot so that the scaling values for all the transducers will tend to one as the total load will tend towards being the actual load rather than a scaled estimate. This will also results with less cumbersome monitoring sessions for the personal calibration of the scaling values for each patient will not be required. Zweifel *et al.*, demonstrated in 1992 a weight-bearing monitor whose shoe insole had between five and seven pressure transducers affixed to it during different tests. Although this was not strictly speaking an ambulatory monitoring system, in that the cables from the transducers trailed across the floor to a desktop computer, the insole measuring system is relevant to the equipment developed during this study. By using more transducers and modifying the program, it might be also possible to obtain data of other aspects of the gait pattern, such as the areas of greater loading bearing.

Direct monitoring of the step duration of the fractured leg relative to the normal leg would be of great benefit, for one could then perform more detailed experiments and monitor how the relative step duration changes over the monitoring session, and according to the distance walked. By so doing, more informative information might be obtained from a patient as regarding the fracture condition. Also the correlation between increased relative step duration and fracture healing might be tested further. To record this data, both legs are required to be simultaneously recorded. Therefore the software and hardware of the ambulatory monitor would require modification, as well as the PC analysis program.

Previous studies, for example Richardson *et al.* (1992), have shown that fracture stiffness increases exponentially with time post-fracture. This greater stability should

result with greater weight-bearing being possible. Although the results from this study are not at variance with this hypothesis, further trials are needed for its confirmation. Were these trials to be performed in conjunction with fracture stiffness measurements, then this hypothesis might be quantitatively proved or disproved. Although both Richardson *et al.* (1992) and Kenwright *et al.* (1991) measured fracture stiffness via the external fixator used during fracture treatment, this would also be possible with internal fixation by using strain gauge transducers and telemetry (Bergmann *et al.*, 1990).

Increases in the amount of data recorded would require an improvement in the size of storage space for the monitor to keep the capability of 24 hour monitoring sessions. Rather than developing the compression capability further, the first modification that would be made regards the information stored for each event. As has been explained previously, by calculating and storing the step duration in time instead of the duration in samples and the inter-sample time, the amount of storage required for the information of one event would decrease by one byte.

However the greater increase in storage capability would be obtained by further data compression, which as has been indicated previously, would occur by using an improved Huffman compression technique. Traditional implementations of Huffman compression techniques have calculated and stored the code using sample data, before the actual data has been recorded. This method is not feasible for this application because the data differs between patients and also between different monitoring sessions of the same patient. However if the code were to be calculated when the Results file became full, then the optimum compression would be obtained for the stored data. The currently stored data would then be compressed, and using the same code, subsequent event data from the monitoring session would be compressed in real-time.

5.4.   Conclusions

An ambulatory monitor has been developed which records the weight-bearing, duration, and time for each step of the fractured leg that occurs during the monitoring session. By calibrating the transducer scaling values for each individual patient, reliable data are obtained. A program executing on a PC which analyses, displays and manipulates the various files has also been written.

This system is able to quantitatively record the patient's weight-bearing and step duration over the monitoring session, storing and displaying the mean, standard deviation of each, and the weight-bearing as a percentage of body weight. Also the weight-bearing progress with time gained from all the monitoring sessions recorded can be displayed. By comparing the session duration for the standard circuit, an indication of the change in the velocity of gait can be obtained. Also the change in the relative stride length over the fracture healing period is gained by the alteration in the number of events recorded for the standard circuit.

This equipment was used to monitor 37 patients with tibial fractures. After further modification, the equipment was found to record reliable data, and subsequently weight-bearing with time was shown to increase non-linearly with time post-fracture. An increase in step duration relative to the step duration of the normal leg also occurred, indicating a gradual change in the gait pattern adopted, tending towards a normal gait pattern with time. The same was found with the stride length, again indicating a gradual change towards a more normal gait pattern with time. An increase in velocity of gait was also observed over the healing period, suggesting greater confidence in walking as healing progressed.

These results are not by themselves sufficient to diagnose the state of the fracture, but they do give an indication of the progression of the fracture healing. Further trials are

required to quantify the expected weight-bearing over the fracture healing period for the general case, and were these to be performed in conjunction with fracture stiffness measurements, a relationship between the two might be derived.

6.   References

BERGMANN G, GRAICHEN F, ROHLMANN A. Implantable Telemetry in Orthopaedics. *Forschungsvermittlung der FU, Berlin*, 1990.

BESAG FMC, MILLS M, WARDALE F, ANDREW CM, CRAGGS MD. The validation of a new ambulatory spike and wave monitor. *Electroencephalography and clinical Neurophysiology* 1989; 73: 157-161.

BLACK J, PERDIGON P, BROWN N, POLLACK SR. Stiffness and strength of fracture callus. Relative rates of mechanical maturation as evaluated by a uniaxial tensile test. *Clinical Orthopaedic Related Research* 1984; 192: 278-288.

BOURNE GH. The Biochemistry and Physiology of Bone, 2nd Ed. *Academic Press, New York*, 1971.

COOK JE. Assessment of tibial fracture healing using Dual Energy X-ray Absorptiometry. *M.Sc. Thesis*, 1993.

CUNNINGHAM JL, EVANS M, KENWRIGHT J. Measurement of fracture movement in patients treated with unilateral external skeletal fixation. *Journal of Biomedical Engineering* 1989; 11: 118-122.

DEHNE E. The rationale of early functional loading in the healing of fractures: a comprehensive gate control concept of repair. *Clinical Orthopaedics and Related Research* 1980; 146(Jan.-Feb.): 18-27.

DUCKWORTH T, BEETS RP, FRANKS CI, BURKE J. The measurement of pressures under the foot. *Foot & Ankle*, 1982; 3: 130-141.

EGGER EL, GOTTSAUNER-WOLFF F, PALMER J, ARO HT, CHAO EYS. Effects of axial dynamisation on bone healing. *The Journal of Trauma* 1993; 34(2): 185-192.

GAUTIER E, PERREN SM, GANZ R. Principles of internal fixation. *Current Orthopaedics* 1992; 6: 220-232.

GRAY H. Gray's Anatomy, 33rd Ed. *Longman's, UK*, 1964.

GOLDSTEIN SA, WILSON DL, SONSTEGARD DA, MATTHEWS LS. The mechanical properties of human tibial trabercular bone as a function of metaphyseal location. *Journal of Biomechanics* 1983; 16(12): 965-969.

GOODSHIP AE, KENWRIGHT J. The influence of induced micromovement upon the healing of experimental tibial fractures. *The Journal of Bone and Joint Surgery* 1985; 67-B(4): 650-655.

HAM AW. Histology, 7th Ed. *JB Lippincott, Philadelphia*, 1974.

HOLTER NJ. New method for heart studies. *Science* 1961; 134: 1214.

HOLTER NJ. Radioelectrocardiography: a new technique for cardiovascular studies. *Annual New York Academic Science* 1957; 65: 913.

HUFFMAN D. A method for the construction of minimum redundancy codes. *Proceedings of the IRE* 1952; 40(9): 1098.

HUTTON WC, DHANENDIAN M. A study of the load under the normal foot during walking. *International Orthopaedics*, 1979; 3: 153-157.

JANKE MT, HESSE S, SCHREINER C, MAURITZ K. Dependency of ground reaction forces, loading and unloading rates of gait velocity, stride length, and constitutional factors in hemiparetic patients. *Proceedings of the European Symposium on Clinical Gait Analysis* 1992: 164-167.

KELLEY A, POHL I. A book on C. *Benjamin/Cummings Publishing Co.*, 1990.

KENNEDY HL *et al.* Ambulatory electrocardiography and computer technology: practical advantages. *American Heart Journal* 1987; 113: 186.

KENWRIGHT J, RICHARDSON JB, CUNNINGHAM JL, WHITE SH, GOODSHIP AE, ADAMS MA, MAGNUSSEN PA, NEWMAN JH. Axial movement and tibial fractures. *The Journal of Bone and Joint Surgery* 1991; 73-B: 654-659.

KLEIN-NULEND J, VELDHUIJZEN JP, BURGER EH. Increased calcification of growth plate cartilage as a result of compressive force in vitro. *Arthritis and Rheumatism* 1986; 29(8): 1002-1009.

LATTA LL, ZYCH GA. The mechanics of fracture fixation. *Current Orthopaedics* 1991; 5: 92-98.

LE VEAU BF. Williams and Lissners Biomechanics of Human Motion, 3rd Ed. *WB Saunders Company, USA*, 1992: 29-59.

LORD M, REYNOLDS DP, HUGHES JR. Foot pressure measurement: a review of clinical findings. *Journal of Biomedical Engineering* 1986; 8: 729-736.

McKIBBIN B. The biology of fracture and healing in long bones. *The Journal of Bone and Joint Surgery* 1978; 60-B(2): 150-162.

MEADOWS TH, BRONK JT, CHAO EYS, KELLY PJ. Effect of weight-bearing on healing of cortical defects in the canine tibia. *Journal of Bone and Joint Surgery* 1990; 72-A, 7: 1074-1080.

MELDRUM SJ. Ambulatory monitoring: an evolving concept. *Biological Engineering Soceity, Physiological Monitoring Group*;18[th.] November 1992.

MICROSOFT. C for yourself. *Microsoft Corporation*, 1990.

MOW VC, HAYES WC. Basic Orthopaedic Biomechanics. *Raven Press, USA*, 1991: 93-142.

NICOLL EA. Fractures of the tibial shaft: A survey of 705 cases. *The Journal of Bone and Joint Surgery* 1964; 46-B(3): 373-387.

NILSSON BER. Post-traumatic Osteopenia: Quantitative study of the bone mineral mass in the femur following fracture of the tibia in man using americium-241 as a photon source. *Acta Orthopaedica Scandinavica* 1966; 91(37): 14-24.

OLSSON E. Partial weight-bearing ambulation - the unloading effect of assistive devices and gait patterns. *Proceedings of the European Symposium on Clinical Gait Analysis* 1992: 104-106.

ONI OOA, HUI A, GREGG PJ. The healing of closed tibial shaft fractures. *Journal of Bone and Joint Surgery* 1988; 70-B: 787-790.

O'SULLIVAN ME, CHAO EYS, KELLY PJ. The Effects of Fixation on Fracture-Healing. *Journal of Bone and Joint Surgery* 1989; 71-A: 306-310.

OXNARD CE. Bone and bones, architecture and stress, fossils and Osteoporosis. *Journal of Biomechanics* 1993; 26: 63-79.

PAAVOLAINEN P, SLATIS P, KARAHARJU E, HOLMSTROM. The healing of experimental fractures by compression osteosynthesis I Torsional strength. *Acta Orthopaedica Scandinavica* 1979; 50: 369-374.

PAAVOLAINEN P, SLATIS P, KARAHARJU E, HOLMSTROM. The healing of experimental fractures by compression osteosynthesis II Morphometric and chemical analysis. *Acta Orthopaedica Scandinavica* 1979; 50: 375-383.

PAN WT, EINHORN TA. The Biochemistry of Fracture Healing. *Current Orthopaedics* 1992; 6: 207-213.

PANJABI MM, WHITE AA, SOUTHWICK WO. Temporal changes in the physical properties of healing fractures in rabbits. *Journal of Biomechanics* 1977; 10: 689-699.

PANJABI MM, WALTER SD, KARUDA M, WHITE AA, LAWSON JP. Correlations of radiographics analysis of healing fractures with strength: a statistical analysis of experimental osteotomies. *Journal of Orthopaedic Research* 1985; 3: 212-218.

PFISTER CJ, HARRISON MA, HAMILTON JW, TOMPKINS WJ, WEBSTER JG. Development of a three-channel, 24-h ambulatory esophageal pressure monitor. *IEEE Transactions on Biomedical Engineering* 1989; 36(4): 487-490.

PRATT CM *et al.* Ambulatory electrocardiographic recordings: the Holter monitor. *Current Problems in Cardiology* 1988; 13(8): 519-586.

PSI SYSTEMS. Mini-Module manual. *P.S.I. Systems*, 1991.

RADIN EL. Orthopaedics for the Medical Students. *JB Lippencott Company, Philadelphia*, 1987: 9-34.

RICHARDSON JB, KENWRIGHT J, CUNNINGHAM JL. Fracture stiffness measurement in the assessment and management of tibial fractures. *Clinical Biomechanics* 1992; 7: 75-79.

ROCKWOOD CA, GREEN DP. Fractures in Adults, 2nd Ed. *JB Lippincott Company, Philadelphia*, 1984.

RS Data Library, 1994.

SARANGI PP, WARD AJ, SMITH EJ, STADDON GE, ATKINS RM. Algodystrophy and osteoporosis after tibial fractures. *The Journal of Bone and Joint Surgery* 1993; 75-B: 450-452.

SHARRARD WJW. A double-blind trial of pulsed electromagnetic fields for delayed union of tibial fractures. *The Journal of Bone and Joint Surgery* 1990; 72-B(3): 347-355.

SHIPMAN P, WALKER A, BIRCHELL D. The human skeleton. *Harvard University Press, Massachusetts*, 1985: 18-63.

ULIVIERI FM, BOSSI E, AZZONI R, RONZANI C, TREVISAN C, MONTESANO A, ORTOLANI S. Quantification by Dual Photon Absorptiometry of local bone loss after fracture. *Clinical Orthopaedics* 1990; 250: 291-296.

WAND JS, SMITH T, GREEN JR, HESP R, BRADBEER JN, REEVE J. Whole-body and site specific bone remodelling in patients with previous femoral fractures: Relationships between reduced physical activity, reduced bone mass and increased bone resorption. *Clinical Science* 1992; 83: 665-675.

WHALEN RT, CARTER DR, STEELE CR. Influence of physical activity on the regulation of bone density. *Journal of Biomechanics* 1988; 21(10): 825-837.

WHITE TD. Human Osteology. *Academic Press, USA*, 1991.

WHITTLE AP, RUSSEL TA, TAYLOR CJ, LAVELLE DG. Treatment of open fractures of the tibial shaft with the use of interlocking nailing without reaming. *The Journal of Bone and Joint Surgery* 1992; 74-B(8): 1162-1171.

WHITTLE MW. Gait Analysis: an introduction. *Butterworth-Heinemann* 1991.

WOLFF J. Das gaetz der transformation. Transformation der knochen. *Hirshwald, Germany*, 1892.

YOUNG DR, NIKLOWITZ WJ, STEELE CR. Tibial changes in experimental disuse osteoporosis in the monkey. *Calcified Tissue International* 1983; 35: 304-308.

ZWEIFEL HJ, KESSELRING J, ARLANCH C, WILLI P, BERNEGGER U, JEHLE A. Erfahrungen mit p-gait-analysis. *Proceedings of the European Symposium on Clinical Gait Analysis* 1992: 260-263.

Appendices

Appendix 1: The Mini-Module PCB Components

The following sections detail the Mini-Module P.C.B. components which are referred to by the Hardware section of Chapter 2. Where relevant explanations of the necessity and function of components is also included.

A1.1.   The Micro-Processor

The CPU on the Mini-Module is a Motorola 68000 software compatible processor; the Philips 93C100. The older Motorola 68000 processor has a slower clock-speed, and needs a number of extra external peripheral interfacing chips to design and build a computer, which the 93C100 includes on-board the processor chip. These are a clock or oscillator, external vectored interrupts, memory interfacing chips, and (for the bus used on the Mini-Module) an I2C bus interface.

Apart from the faster clock speed of 30 MHz (the 68000 having a maximum of 12 MHz) the main functional difference is that the 93C100 also has a second on-board oscillator which drives it at the slower speed of 5 MHz; this feature being used for when the processor is in 'stand-by' mode. When in this state the processor consumes less power which is important in power sensitive applications such as that of ambulatory monitoring.

A1.2.   The Erasable Programmable Read Only Memory (EPROM)

A computer system needs memory for the purpose of storing the program whilst it is being executed by the processor, and for storing and manipulating the data that is produced.

EPROM is 'programmed' (meaning that each memory location's content is set to a value) by applying different voltages to various pins of the casing. This is done automatically by an EPROM programmer, which stores the file and transfers it to the EPROM. Depending upon the size of the program or data being stored, this can take one or two minutes. The EPROM chips can then be inserted into the sockets provided on the PCB of the Mini-Module, and their contents read by the processor. To erase the memory of its contents, the silicon chip is exposed through the clear 'window' in the casing to ultra violet light for some twenty minutes.

As programming and re-programming of an EPROM is a long process, taking up to half an hour, this type of memory can not be used for applications which involve constantly changing values; such as the data generated by a program, or in this case read in by the monitor. However it can be used for unvarying data such as the program code itself, and initialising data which does not change and is needed when commencing program execution. A benefit of using EPROM rather than other types of memory for program storage is that the contents are not lost when the memory is disconnected from the power supply, which means that battery power is saved and the monitor need only be powered for the time period when the data is being gathered, rather than having to constantly power it in order to keep the program in memory.

The Mini-Module is flexible on the differing sizes of EPROMs that it can use. Either CMOS or NMOS types can be used. These are based on different technologies and function differently in operation although performing the same task. CMOS type of EPROM was chosen for that consumes less power than the equivalent sized NMOS EPROM. The size of memory of the EPROM can be from 16 KBytes to 256 KBytes each, giving an overall memory of between 32 KB and 512 KB as two EPROMs are used. The memory size chosen was of 128 KB each (giving 256 KB in total), to

ensure that there would be ample room in which to store the program code and the
initialising data.

The EPROM speed of operation (when returning a specified memory location's value)
is slower than for other types of memory, and much slower than the CPU operational
speed. To circumvent such problems, the Mini-Module uses an asynchronous bus
interface which means that the speed of each access cycle is controlled by the device
being accessed, and not by the CPU. Therefore the Mini-Module has some external
(to the CPU) timer logic which forces each EPROM read cycle to be at least 350 ns
allowing the use of EPROMs with access times of up to 250 ns. However this
application is not adversely affected by the slower memory speed because for most of
its execution time the program will be periodically monitoring and storing the ADCs'
values. In fact it will have to be slowed down even further in its processing speed in
between taking individual readings from the ADCs, otherwise the sampling rate would
be in the thousands rather than in the tens of hertz range.

A1.3.   The Random Access Memory (RAM)

RAM is available in two types; static and dynamic. Each type of is of different
technology and construction; each location in dynamic memory being a capacitor and
resistor, whilst in static memory it is a transistor. Dynamic memory is therefore much
easier and more compact to manufacture on silicon and so costs less than its static
equivalent. However as each location value is stored by the capacitor charge (a zero
value being no charge stored and a one being charge stored) it has to be 'refreshed'
periodically for it to be maintained. This means that dynamic memory has a greater
power consumption than static memory since when a small charge is given to the base
of the transistor (signifying a one for current will now flow from the collector to the
emitter) it remains there until it is changed or the power is switched off. Static RAM is
also much faster in operation than dynamic RAM as the dynamic memory is limited in

speed to the capacitor discharge rate. In quantitative terms this gives an access time of 20 ns for static RAM and 80 ns for dynamic RAM.

The Mini-Module is fitted with 128 KB of static RAM which is therefore of benefit over dynamic RAM in its overall power consumption. Since the program and initialising data is stored on EPROM, all of this memory area (apart from that required by the operating system) can be utilised by program generated data. As there is an interface to the 68000 bus, external memory can be added to form a total of 2 MBytes (as the address bus has a total of 20 lines) should applications require it. For this application however, 128 KB of RAM was deemed to be sufficient.

A1.4.   The Battery Back-up

The Mini-Module also has a nickel cadmium (ni-cad) battery mounted on the PCB, which is connected to the static RAM when the external power supply, which in this case is the set of batteries, is disconnected. This battery can supply enough power for the RAM to keep its contents for up to about 250 hours, because static RAM consumes very little power when in an 'idle' state, which is when its contents are not being accessed or set. As explained previously, the life for this battery would be very much shorter if dynamic RAM was used, due to the different technology it employs.

The ni-cad battery has a discharge ratio of ten to one. This means that an external power source must be connected for 10% of the time for the battery to remain charged. For this application, the Mini-Module would be powered only when it was being used to monitor a patient, but when the monitoring trial finishes the results are down-loaded onto a PC for storage. Therefore even if patient trials were infrequent, so causing the battery-back up to fully discharge, no important data remains in the RAM after a trial, except some initialising values which can be re-inputted, meaning that no important data will be lost.

A1.5. Real Time Clock

The real time clock provides a clock facility which counts in 1/100ths. of a second. It also includes a calendar, and a timer which can count for up to 99 days. Also an alarm facility is included, which can generate an interrupt at a particular date or time of the clock timer. This clock can give time facilities to the Mini-Module's programs which has been utilised for this application.

A1.6. Digital Input/Output Communications

Analogue communication consists of a varying voltage signal, the amplitude indicating the 'number' being transmitted. Digital communication does not have this flexibility of a varying signal amplitude as the voltage level can be either 'on' or 'off', corresponding to either 5 Volts or 0 Volts respectively. Representing a one or zero is therefore straight-forward, and for other values a number of digital lines can be used in parallel, with the value being encoded in a binary format. It has become standard to have digital lines in multiples of eight; so that an 8-bit processor would normally communicate with other peripherals across an eight or sixteen line data bus, so being able to directly manipulate an eight or sixteen bit number (i.e. between 0 and 255 or 0 and 65535 respectively).

The Mini-Module does not only permit external analogue communications (accepting inputs via the ADCs, and generating an output through the Digital to Analogue converter or DAC) but also external digital communication facilities through four eight bit digital ports, which are basically four sets of eight parallel digital lines. Each port is quasi bi-directional which means that although physically it is only an output port, it can also be programmed to be an input port. To understand why this is possible, it is necessary to examine the digital line more closely.

The high level for each digital line is provided by a 100 μAmps current source with the low level output being provided by a high current field effect transistor (FET) which can accommodate an input current of up to 25 mAmps. Therefore each line will read as high (5 Volts) when not being driven, by having been set by the program to a logical 'off', and it will be read as low (0 Volts) when it is set to 'on', as the FET will then be 'active'. When being driven by an outside source, a high voltage value will cause a digital line's voltage to remain at 5 Volts, and a low voltage value will drive the line to 0 Volts as the relatively small 100 μAmps will be dissipated by the external equipment since the current will flow from the Mini-Module to the connected external equipment, effectively acting as an earth for the digital line.

Each port has a change of state detector which periodically compares the state of the pins of the port with a copy of the state of the pins when the port was last read. When a difference is noted, an interrupt is generated. This is then removed by either the port returning to its original state or it being read by the CPU. Therefore a port can be used for input purposes by either waiting for the interrupts to occur, as the initial state is known, or by periodically sampling the port and ignoring the interrupts that will be generated on each change of state.

For this application there is a requirement to use five digital lines. The first is needed as a digital input to 'read' the state of a switch, so that when the switch is closed different program functions can be enabled. The other four are needed to be used as outputs, to drive four Light Emitting Diodes (LEDs) which display to the operator the different states of the program executing on the Mini-Module. Both are detailed in the next part which deals with peripherals required for the monitor which were not found directly mounted on the PCB.

A1.7.   Analogue to Digital Converters (ADCs)

The Mini-Module has four of ADCs which include a sample and hold amplifier. Each ADC is an 8 bit device (meaning that the range of possible digital outputs is from 0 to 255) and the input range is from 0 to 2.55 Volts. This therefore gives a sensitivity of 10 mV per bit over its input range. The ADCs can be configured for a number of different input modes, giving four single ended inputs or two differential inputs.

With single ended input mode, the input is connected to the positive input of the ADC, and the voltage measured between the input and the analogue ground of the Mini-Module. To make sure that the ground voltage levels are the same for both the Mini-Module and the external voltage source which is being measured, the external source's ground can be connected to the analogue ground of the Mini-Module. The use of this connection method gives the possible utilisation of four ADCs, and it works satisfactorily using short cables in low noise environments. However if the environment is noisy (i.e. there is a relatively high amount of electromagnetic radiation in the area) a voltage will be induced in the cable which will superimpose on the voltage being measured to give a higher or lower voltage reading at the ADC than that generated by the external voltage source. This problem is exacerbated the greater the cable length as a greater voltage can be induced. In low noise environments the use of a long cable will result in a voltage drop due to its internal resistance, for the longer the cable the greater the resistance its resistance, so giving a lower voltage reading at the ADC.

Differential input mode works by connecting the two inputs from an external voltage source to two separate ADCs, one to its positive input and the other to its negative input, the voltage reading then being the difference between the two. This method has the advantage of noise immunity, for if a voltage is induced it will be induced to the same degree on both inputs because they are normally tied together so there is no possibility of each one being affected by different electromagnetic radiation sources, as

might happen if the leads were metres apart. However since the voltage in one cable is different than in the other cable, a voltage can be induced from the higher voltage to the lower voltage cable by coupling. Therefore shielded cables are used, with the shields being connected to the analogue grounds of both the Mini-Module and the external voltage source, so reducing the common mode voltage, and also further reducing the noise sensitivity. Since two ADCs are required for each voltage source being measured, using this method means that only two different voltage sources can be monitored.

The final connection mode possible is the quasi differential mode. Each channel's negative input is connected to the same negative ADC input, with the positive voltage source inputs connected to separate ADCs' positive inputs. By using this method of connection, three different ADCs are available for monitoring use. Also this mode offers noise immunity for the cables and for voltage drops over long distances, but not to the same degree as with using differential input mode.

For this application, each pressure transducer was connected to a signal conditioning unit which returned a separate voltage reading for each transducer to the monitor. Therefore each transducer's reading, via the signal conditioning unit, would be monitored by a separate ADC. The environment where the equipment would be in use is in a hospital's outpatients department, where there would not be any extra-ordinary levels of electro-magnetic radiation. Also because the cables connecting the signal conditioning units would be less than a metre in length, for the ambulatory monitor would be worn on a belt round the waist and the signal conditioning units positioned at the ankle, no noticeable voltage drop should occur. All these factors, plus the fact that some measure of immunity to noise is possible by the use of shielded cables, combined to the decision to use single ended input mode, so that up to four transducers could be monitored. As shall be seen later on, this was important to be able to obtain accurate weight-bearing data.

A1.8.   The RS-485 Port

The RS-485 serial communication standard provides serial communication using two differential lines for each channel. This allows the use of simple twisted pair cabling, and so will provide a high degree of noise immunity when the cable has to traverse long distances.

An RS-485 port is provided on the Mini-Module as standard, which provides it with serial communication capability. For this application the serial connection is required to interface to an I.B.M. compatible Personal Computer (PC) . However PCs are fitted with RS-232 standard serial ports which are not compatible with the RS-485 standard. Therefore an RS-485 to RS-232 converter was built which provided the necessary conversion so that the PC and Mini-Module could communicate with each other. This is detailed in a subsequent part of this Chapter.

A1.9.   The Watch-Dog

A watch-dog is a timer chip which is reset by a pulse on its trigger line. If a pulse does not occur within a specified time, the watch-dog generates an interrupt. Using a watch-dog gives a computer some fault-tolerance capabilities for a program or more usually for an operating system. This functions in the following manner; a pulse is regularly transmitted to the watch-dog, but if a fault occurs so that a pulse is not sent to it, an interrupt is generated which can be specified to jump to a memory location for the execution of a specific part of the program, which might for example jump back to the start of the program or function that was being executed, the benefit being that no data would be lost. This is a feasible scenario because most faults that occur are transient faults rather than hardware faults; for example connecting or disconnecting a high current device to the same mains supply as the computer will generate a voltage spike and possibly a transient fault if the computer power supply is not sufficiently shielded. If this were to occur, the program counter might become corrupted and so

send the CPU to a different memory location possibly sending the program or operating system into an infinite loop. Therefore the instruction to send a pulse to the watch-dog would not be processed and so the pulse would not be sent. An interrupt would therefore be generated, and the specified code processed could then send the program counter to restore control to the start of the operating system, so enabling the computer to overcome the transient fault whilst keeping the majority of its previously generated data.

A watch-dog timer is present on the PCB, to which if a pulse is not received on its trigger line by 400 ms, it resets the Mini-Module. When using the multi-tasking Minos operating system, the individual programs do not have to periodically send a pulse to the watch-dog as this is done by the operating system. If the watch-dog feature is not required in an application it can be disabled by removing a link on the Mini-Module.

In this application, the monitor program was executed under the Minos operating system and so the watch-dog was enabled, as its functionality would be beneficial if the ambulatory monitor were to be used in an environment where there was relatively high electro-magnetic radiation that might affect the Mini-Module circuitry. When executing programs under the Minos OS, a reset is generated if the watch-dog interrupt occurs. Therefore if a transient fault were to occur and a reset was generated by the watch-dog timer, the CPU would go to the start of the program but with the collected data, which had not yet been downloaded onto the PC, still being intact since it is stored in its own RAM file.

A1.10.   The Power Fail Detector

Present on the PCB is also a power fail detector. This is connected to the power supply and monitors its voltage. If the voltage level drops below about 4.75 Volts, the power fail detector resets the Mini-Module. This feature is useful in this particular

application because the power supply will be a set of batteries with their general discharge characteristic being that the voltage decreases as they are used (the battery characteristics will be detailed in Section 2.1.2.3). Therefore to have the Mini-Module being reset when the batteries' voltage is low, will indicate to the user that they need changing. If no reset occurred, the ambulatory monitor would continue to appear to function normally as the status LEDs would continue to light; but the Mini-Module would produce some transient faults in program operation since the voltage is not high enough to drive the transistor transistor logic (TTL) circuitry properly.

A1.11. Other PCB Components

The PCB holds a number of other components which are not utilised for this application. There is a DAC, a keyboard port and an LCD port. The keyboard and LCD ports are not required since through the RS-485 port the PC's keyboard and screen are utilised when required.

The Mini-Module also has three different bus standards available for external connection. An I2C bus is used by the 93C100 to interface to other peripherals on the PCB, and a connector is also provided for external peripherals. A 64 pin expansion connector is also provided, this bus having three basic modes of operation which allows access to 68000 memory (for external memory expansion), 68000 peripherals and 8051 peripherals. Each mode uses the same address and data lines, but a different set of control lines.

As all the hardware needed for this application was already provided on the Mini-Module PCB, no external peripherals were needed so that neither of the expansion connectors were used.

Appendix 2 : Ethical Approval and Original Project Protocol

SOUTH TEES
HEALTH

District Offices
POOLE HOSPITAL
Nunthorpe
Middlesbrough
Cleveland TS7 0NJ
Telephone: Middlesbrough (0642) 320000
Fax: (0642) 324176

JRCS/DD

7 December 1990

Mr D Muckle
Consultant Orthopaedic Surgeon
Middlesbrough General Hospital
Middlesbrough
Cleveland

Dear David

90/46 - THE INFLUENCE OF THE MAGNITUDE AND DURATION OF WEIGHT BEARING ON THE HEALING OF TIBIAL AND FEMORAL FRACTURES

Thank you for submitting this protocol to the Ethics Committee. We do not see any ethical problems, and are happy for you to proceed with the study.

I presume that the microprocessor based data logging device has now been developed, and you are moving into the stage of testing it in patients with tibial and femoral fractures, as outlined in method (c).

I would remind you that you should obtain informed consent from the patients who participate in the study, and we look forward to receiving a report of your results in due course.

With kind regards.

Yours sincerely
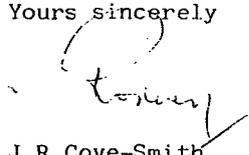
J R Cove-Smith
Chairman
Ethics Committee

PROJECT PROTOCOL

TITLE: The influence of the magnitude and duration of weight-bearing on the healing of tibial fractures.

BACKGROUND

Fracture healing is influenced by the prevailing mechanical environment at the fracture site (1-4). Fractures which have been accurately reduced and in which there is minimal interfragmentary strain, heal directly by primary means (1,2), whereas fractures which are less rigidly fixed heal by secondary bone healing with external callus formation, the amount of callus depending on the rigidity of the fixation (1, 3, 4).

The rate of increase of fracture stiffness and strength can be influenced by the rigidity of the fixation system, this being seen in both experimental (3-5) and clinical studies (7). Most conservatively treated tibial fractures show incomplete reduction and so indirect healing with external callus formation leads to the most effective and rapid healing of the fracture. The potential therefore exists to use weight bearing to produce axial loading of conservatively treated tibial fractures and hence stimulate callus formation.

To encourage fracture healing by secondary means, early weight bearing is prescribed and encouraged to provide the axial strain at the fracture site necessary to promote callus formation (6). In a photogrammetric study, Lippert and Hirsch (8) demonstrated that large amounts of movement at the fracture site (up to 5 mm) are possible during normal activities in fractures treated by cast. In studies of patients being treated by cast braces for femoral shaft fractures, the loading of the fractured limb during healing has been measured (9, 10) and has been shown to increase with increasing time post-fracture. More recently, Cunningham *et al* (11) studied weight bearing and fracture movement at set intervals during healing in a small group of patients treated with unilateral external skeletal fixation. Despite being encouraged to weight-bear on their fractured limb, weight bearing was less than 50% of body weight during the first two months post fracture. It is in this early stage of healing that axial strain at the fracture site appears to be most effective in promoting healing (4). In all of these studies the results represent the weight bearing achieved during the tests and direct inference cannot be made that such weight bearing was the norm when the patient was at home.

As healing of the fracture progresses, the ability of the patient to weight bear on the fractured limb increases, perhaps as a result of a bio-feedback mechanism of biological self-control of fracture site strain, as suggested by Lazo-Zbikowski *et al* (12).

In fractures, the level and frequency of weight bearing will affect fracture healing, and information on weight bearing during treatment would be invaluable in assessing treatment methods, (ie casts, internal and external fixation) patient motivation and injury and fracture type on the ability of the patient to weight bear. By being able to determine favourable influences on weightbearing, then increased callus formation and more rapid fracture healing could result. The information obtained from measurements of weight bearing could also be used as an indication of the extent of fracture repair if correlated with clinical, radiological and mechanical (13) assessments of union. This technique, when developed, would be potentially applicable to other orthopaedic treatments where a measure of patient activity either pre- or post-surgery (e.g. total hip and knee replacement) is required.

METHODS

Microprocessor based data logging devices

Ambulatory monitoring of patients has become a widespread clinical diagnostic technique over the past 25 years. Probably the best known example is recorded electrocardiography (Holter monitoring) which was reported as early as 1961 (14) and detected ST segment changes in patients during symptomatic anginal attacks.

Three distinct recording methods are presently available, these are continuous, intermittent (patient or time activated) and real time analytical recorders. Continuous, two-channel analogue Holter tape-recorders are the most widely used in the field. Most are now cassette based and offer reasonable recording fidelity, but are bulky and thus inconvenient to use. Patient-activated recorders are limited in their use since they must be activated by the patient in response to symptoms so large amounts of important data may be missed. An additional drawback is the limited memory of many of the present systems and the lack of input channels.

Real-time analysers are recorders with the ability to analyse the incoming signal in real-time, subsequently storing examples of abnormalities. Unfortunately these real-time analysers have

difficulty in reading through ambient noise, this is a major problem especially when dealing with ECG signals.

In order to investigate weightbearing achieved during fracture treatment it is proposed to develop a microprocessor based instrumentation and data acquisition system. This system would need to be portable, self powered, unobtrusive and be able to monitor patients for long unsupervised periods, possibly away from the hospital environment. The use of a microprocessor-based system would enable the device to be "intelligent" - making decisions as to whether data was useful or erroneous. This would allow data compression to take place allowing an extended monitoring period. This monitoring period could also be prolonged by the use of real-time analysis of the data.

Force measurement system

Forces and pressures under the foot have been measured using single and multi-element force plates and optical methods (15, 16), however the use of such systems is restricted to a laboratory environment,. Sensors have been developed which fit inside the shoe and allow a continuous measure of activity (17), although not of the magnitude of the loading applied. In this project the aim is to enable a continuous measurement of limb loading to be made over a period of time. The distribution of the loading over the foot is not considered to be as important as the magnitude of the load applied to the limb, and so it is initially intended to develop a force measurement system based on two miniature pressure transducers, positioned over the area of the fore-foot and the heel. Alternative methods of measuring pressures under the foot will also be explored, including the use of piezoelectric polymers, specifically polyvinylidene fluoride (PVDF), although the costings in this proposal are based on the available technology of a pressure transducer system.

PLAN OF INVESTIGATION

a) Force measurement system

It it proposed to develop a compact force measurement system which will enable continuous measurements of loading and the duration of loading applied by the foot of the fractured leg. This system would consist of a pressure measuring element or elements utilising an array of small pressure transducers or alternatively could be constructed from a piezo-electric

polymeric material. A suitable power supply and amplification for such a pressure sensitive element will be developed and incorporated within the data recording equipment.

b) Data logging system

It is proposed to base this system around a commercially available microprocessor system (15) which uses an industry standard Motorola 68000 compatible microprocessor chip. This system is based on a small (100 mm x 115 mm) printed circuit board that contains the microprocessor, 128K of static RAM (random access memory), an interactive programming language and sufficient input/output for this stand-alone application.

To allow a specific system for the investigation of weightbearing during fracture treatment to be designed, finance for the development of a system comprising:

>   IBM compatible PC (386)
>   Printer
>   PSI Systems development system (PSI-J100)
>   Cross Assembler for 68000 cpu or Cross Compiler for 68000 cpu
>   ROM splitter software
>   S-Record generator

is requested. In addition, once developed, individual systems would be required to enable clinical trials to be carried out on fracture patients. This would require finance for:

>   10 off PSI-K100 mini-module controllers and a budget for miscellaneous analogue and digital electronic components.

c) Clinical testing

Preliminary testing of the force measurement and data logging system will be carried out on a small series of volunteer subjects to determine the accuracy and reliability of these systems. Subsequently, a series of about 20 - 30 patients being treated by cast,, internal and external fixation for tibial fractures will be fitted with the measurement and logging system, and measurements of the amount and duration of fracture loading will be made continuously throughout treatment. In addition to the usual clinical and radiological assessment of healing,

mechanical assessments of healing will be made using either ultrasound (16) or a direct method of measuring fracture stiffness (13, 17).

JUSTIFICATION FOR SUPPORT REQUESTED

The support requested will enable a postgraduate research assistant to carry out the investigations described in detail in Section 4 above. The research assistant would be employed by the University of Durham, and housed in the Bioengineering Laboratory in the School of Engineering and Computer Science at that University. The research assistant would be responsible to Dr J L Cunningham. During the clinical testing, the research assistant would be required to make frequent visits to Middlesbrough General Hospital, and travel costs associated with these visits have been included in the application.

REFERENCES

1.   McKibben, B. The biology of fracture healing in long bones. J. Bone and Joint Surg. 60B, 152-162, 1978.

2.   Perren, S.M. Physical and biological aspects of fracture healing with special reference to internal fixation. Clin. Orthop. Rel. Res. 138, 175-196, 1979.

3.   Sarmiento, A., Schaeffer J.F., Beckerman L, Latta L.L. and Eris, J.E. Fracture healing in rat femora as affected by functional weight bearing. J. Bone and Joint Surg. 58A, 369-375, 1977.

4.   Goodship A.E. and Kenwright J. The influence of induced micromovement upon the healing of experimental tibial fractures. J. Bone and Joint Surg. 67B, 650-655.

5.   Woolf J.W., White A.A., Panjabi M.M. and Southwick, W.O. Comparison of cyclic loading versus constant compression in the treatment of long bone fractures in rabbits. J. Bone and Joint Surg. 63A, 805-810, 1981.

6.   Sarmiento, A. Function bracing of tibial fractures. Clin. Orthop. Rel. Res. 105, 202-219, 1974.

7.   Kenwright, J., Richardson, J.B., Cunningham, J.L., White, S.H., Goodship, A.E., Adams M.A., Magnussen, P.A. and Newman J.H.   Axial movement and tibial fractures.   A controlled randomised trial of treatment.   J. Bone and Joint Surg. 73-B,654-659, 1991.

8.   Lippet, F.G. and Hirsch, C.   The three-dimensional measurement of tibial fracture motion by photogrammetry.   Clin. Orthop. Rel. Res. 105, 130-143, 1974.

9.   Meggit, B.F., Juett, D.A. and Smith, J.D.   Cast-bracing for fractures of the femoral shaft.   A biomechanical and clinical study.   N. Bone Joint Surg. 63-B,12-23, 1981.

10.   Wardlaw, D. M^{C}Lauchlan, J., Pratt, D.J. and Bowker, P.   A biomechanical study of cast-brace treatment of femoral shaft fractures.   J. Bone Joint Surg. 63-B, 7-11, 1981.

11.   Cunningham, J.L., Evans M and Kenwright J.   Measurement of fracture movement in patients treated with unilateral external skeletal fixation.   J. Biomed. Eng. 11, 118-122, 1989.

12.   Lazo-Zbikowski, J. Aguilar, F., Mozo, F., Gonzales-Buendia, R. and Lazo, J.M.   Biocompression external fixation: sliding external osteosynthesis.   Clin. Orthop. Rel. Res. 206, 169-184, 1986.

13.   Cunningham, J.L., Kenwright, J. and Kershaw, C.J.   Biomechanical measurement of fracture healing.   J. Med. Eng. and Technol. 14, 92-101, 1990.

14.   Holter, N. (1961).   New method for heart studies.   Science, 134, 1214-1220.

15.   Hutton, W.C. and Dhanendran, M.   A study of the distribution of load under the normal foot during walking.   Int. Orthop. 3, 153-157, 1979.

16.   Duckworth, T., Betts, R.P., Franks, C.I. and Burke, J.   The measurement of pressures under the foot.   Foot & Ankle 3, 130-141, 1982.

17.   Harris, D., Gwillim, J., Cochrane, G. and Hopkins, S.   Monitoring performance and activity. 10th Annual Report of the Oxford Orthopaedic Engineering Centre, 80-84, 1983.

18.   PSI Systems Mini-module Hardware Manual PSI-K100/3.

19. Cunningham, J.L. and Kershaw, C.J. Ultrasonic assessment of fracture healing. Brit. J. Radiol. <u>63</u>, 393, 1989.

20. Shah, K.M., Nicol, A.C. and Richardson, J.B. A method of non-invasive fracture stiffness measurement. Proc. 6th Meeting of the European Society of Biomechanics, C10, 1988.

Appendix 3 : The Ambulatory Monitor Program Listing

```
1   /* This programme is in a finished and working state, all options       */
2   /* having been fully tested.  However due to its evolution               */
3   /* during trials parts of one feature, that due to time limitations was  */
4   /* not finished, remain in the code (for aiding future extension of the  */
5   /* programme).  This feature is the simultaneous monitoring of 2 legs.   */
6   /* Therefore currently only 1 leg can ever be monitored during a trial   */
7   /* meaning that 'legs_monitored' is always 1.                            */
8
9   #include <stdio.h>
10  #include <minos.h>
11  #include <moddef.h>
12  #include <i2c.h>
13  #include <time.h>
14  #include <mriext.h>
15  #include <string.h>
16  #include <errno.h>
17  #include <stdlib.h>
18  #include "moddef.h"
19  #include "procs.h"
20  #include "minos.h"
21
22  #define ON                      1
23  #define OFF                     0
24  #define SUCCESSFUL              1
25  #define UNSUCCESSFUL            0
26  #define DATA_SIZE               212
27  #define RESULTS_SIZE            30000
28  #define EVENT_SIZE              8 /* event size, power down, display,
29  legs_monitored, calibrate_value0, 1, 2, 3 */
30  #define EVENT_LEVEL1            30
31  #define DATA_FILE               1
32  #define RESULTS_FILE            2
33  #define SECONDS                 0
34  #define HUNDREDTHS_OF_SECS      1
35  #define RECORDING               6
36  #define PC_LINK                 4
37  #define TRANSMITTING            2
38  #define ERROR                   0
39  #define MEMORY_FULL             0
40  #define SWITCH_PC_LINK          8
41
42  typedef char tname[10];
43
44  load ( char * );
45  extern int _paths[];
46  void *sysmem( int, int );
47  char *clearwhite( char * );
48  char *getcmd( char *, char * );
49  char *getarg( char *, char *, int );
50  void *link( char * );
51  int save_file( FILE * , int );
52  float time_increment( void );
53  void calc_results( void );
54  void pressure_input( void );
55  int open_data_file( int );
56  int open_results_file( int );
57  int open_event_file( void );
58  void setup_datamods( void );
59  void writereg( int, int );
60  void writebcdreg( int, int );
61  void sleep( int, int );
62  int link_test( void );
63  void stamp_results( int );
64  unsigned char get_adc( int );
65  void switch_on( int );
66  void switch_off( int );
67  unsigned char interpolate( int, unsigned char, unsigned char, unsigned char
68  );
```

```
69     void get_outfile_name( char * );
70     void error( void );
71     void startup_event_file( void );
72     void setup_event_file( void );
73
74
75     /* The following structures, pointers, etc. are defined globally for */
76     /* ease of implementation. */
77
78     struct moddef *results1;
79     struct moddef *results2;
80     struct moddef *event;
81     struct moddef *data1;
82     struct moddef *data2;
83     struct moddef *test;
84     unsigned char *cur_result1;
85     unsigned char *cur_result2;
86     unsigned char *cur_event;
87     unsigned char *power_down;
88     unsigned char *display;
89     unsigned char *legs_monitored;
90     unsigned char *calibrate_value0;
91     unsigned char *calibrate_value1;
92     unsigned char *calibrate_value2;
93     unsigned char *calibrate_value3;
94     unsigned char *cur_data1;
95     unsigned char *cur_data2;
96     struct tm tim;
97     struct tm *cur_time;
98     char outfile_name[13];
99     char module_name[]="000";
100
101    /* Below are variables used by shell(). */
102
103    tname types[] =
104    {
105            "Program",
106            "Dit",
107            "Driver",
108            "System",
109            "Modula",
110            "Data"
111    };
112
113    char Buffer[80];
114    FILE *in;                      /* Holds path to be used for input */
115    FILE *out;                     /* Holds path to be used for output */
116    int coproc;                    /* Flag to show concurrent execution */
117    int Inp_Path;
118    int Out_Path;
119
120
121    /* The following routine is called at the initialisation stage of the   */
122    /* program if this is the first time it is being run (ie. there is no    */
123    /* event file present in memory with assigned flags for program         */
124    /* operation and scaling values for the transducer calibration.         */
125    /* Therefore the flags and scaling values are set to initial defaults). */
126
127    void startup_event_file()
128
129    {
130
131    cur_event = (unsigned char *) event + event -> start;
132
133    *cur_event = EVENT_LEVEL1;      /* Threshold minimum value for the      */
134                            /* occurrance of an event.              */
135    cur_event++;
136    *cur_event = ON;            /* Processor power down = ON */
137    power_down = cur_event; /* The power_down (and subsequent pointers) are */
138                       /* strictly necessary, but they aid in program  */
139                       /* readability and give a slight speed increase */
140                       /* at the expense of extra memory usage.        */
```

```
141
142     cur_event++;
143     *cur_event = OFF;            /* Display = OFF */
144     display = cur_event;
145
146     cur_event++;
147     *cur_event = 1;              /* No. of legs_monitored = 1 */
148     legs_monitored = cur_event;
149
150     cur_event++;
151     *cur_event = 120;               /* Calibrate_value0 = 0.90 (Calcaneous  */
152                           /* transducer scaling value).            */
153     calibrate_value0 = cur_event;
154
155     cur_event++;
156     *cur_event = 85;         /* calibrate_value1 = 0.75 */
157     calibrate_value1 = cur_event;
158
159     cur_event++;
160     *cur_event = 120;           /* calibrate_value2 = 0.75 */
161     calibrate_value2 = cur_event;
162
163     cur_event++;
164     *cur_event = 120;           /* calibrate_value3 = 0.75 */
165     calibrate_value3 = cur_event;
166
167     cur_event = (unsigned char *) event + event -> start;
168
169     }
170
171
172     /* The following routine is called at the initialisation stage of the    */
173     /* program if the program was run previously and therefore the battery   */
174     /* backed RAM still contains the previous events file.  It therefore     */
175     /* just sets the pointers to point to the relevant part of the file      */
176     /* storing their value.                                                  */
177
178     void setup_event_file()
179
180     {
181     cur_event = (unsigned char *) event + event -> start;
182
183     cur_event++;
184     power_down = cur_event;
185
186     cur_event++;
187     display = cur_event;
188
189     cur_event++;
190     legs_monitored = cur_event;
191
192     cur_event++;
193     calibrate_value0 = cur_event;
194
195     cur_event++;
196     calibrate_value1 = cur_event;
197
198     cur_event++;
199     calibrate_value2 = cur_event;
200
201     cur_event++;
202     calibrate_value3 = cur_event;
203
204     cur_event = (unsigned char *) event + event -> start;
205
206     }
207
208
209     main()
210
211     {
212     int run=SUCCESSFUL,i, outcome;
```

```
213     FILE *fptr;
214
215     initi2c();       /* This initialises the Mini-module i2c bus for I/O */
216
217     cur_time = &tim;         /* The library functions which access the      */
218                      /* real-time clock return to a structure of type*/
219                      /* tm.  Therefore tim stores the values, and a  */
220                      /* pointer to it (cur_time) is used to access    */
221                      /* them.                                         */
222
223     /* The following lines set the digital channels to either inputs or    */
224     /* outputs (the input needed only for the channel connected to the     */
225     /* switch which when depressed indicates to the Mini-module to attempt */
226     /* to access the PC via the serial interface.                          */
227
228     outch( RECORDING );
229     outch( PC_LINK );
230     outch( ERROR );
231     outch( TRANSMITTING );
232     inch( SWITCH_PC_LINK );
233
234     /* The following simply switch off the leds.     */
235
236     switch_off( RECORDING );
237     switch_off( PC_LINK );
238     switch_off( ERROR );
239     switch_off( TRANSMITTING );
240
241     outcome = open_event_file();     /* An attempt is made to set up an     */
242                      /* events file which is only successful */
243                      /* if there is none already present.    */
244
245     if( outcome==SUCCESSFUL )         /* If first time round, setup variables */
246                      /* (pointers).                           */
247           startup_event_file();
248     else
249           setup_event_file(); /* Started up with event file already present */
250
251     setup_datamods();        /* This function deletes any data files present */
252                      /* and makes new blank ones.  It also creates   */
253                      /* new results file(s) if not already present.  */
254
255     while( 1 ) {     /* endless loop */
256
257           setup_datamods();        /* Clears data files.   */
258
259           pressure_input();        /* This function samples the A/D        */
260                      /* converters until the data file(s) are*/
261                      /* full.  In this function the sampling */
262                      /* can be interrupted for PC access for */
263                      /* downloading results etc..            */
264
265           calc_results();          /* This function analyses the data      */
266                      /* file(s) and stores the information   */
267                      /* for any events that occur.           */
268
269           }
270
271     error();         /* program execution should never get here */
272
273     }
274
275
276     /* This routine stamps each result file with the year, month, and day of*/
277     /* month.  As this information is the same for all events, it is only   */
278     /* stored once at the start of each Results file.  The tree items of    */
279     /* information are compressed into two bytes by bit shifting.           */
280
281     void stamp_results( int no_of_legs )
282
283     {
284     unsigned char ctime1, ctime2;    /* These 2 variables hold the compressed*/
```

```
285                               /* information.                       */
286    getime(cur_time);          /* The cur_time pointer of type tm structure   */
287                        /* points to a tim of type structure tm which   */
288                        /* stores the time information.                 */
289
290    ctime1 = (unsigned char) ((cur_time -> tm_year)-1);
291    ctime1 = ctime1 << 1;
292    ctime1 += (unsigned char) (cur_time -> tm_mon) >> 3;
293    ctime2 = (unsigned char) (cur_time -> tm_mon) & 7;
294
295    ctime2 = ctime2 << 5;
296    ctime2 += (unsigned char) (cur_time -> tm_mday) & 31;
297
298    if( no_of_legs == 2 ) {
299          *cur_result2 = ctime1;
300          cur_result2++;
301          *cur_result2 = ctime2;
302          cur_result2++;
303          *cur_result2 = 255;      /* Two subsequent 255s indicate the end */
304                        /* of the results currently stored in   */
305                        /* the Results file.                    */
306          cur_result2++;
307          *cur_result2 = 255;
308          cur_result2++;
309          }
310
311    *cur_result1 = ctime1;
312    cur_result1++;
313    *cur_result1 = ctime2;
314    cur_result1++;
315    *cur_result1 = 255;
316    cur_result1++;
317    *cur_result1 = 255;
318    cur_result1++;
319
320    }
321
322
323    /* The following two routines are written for better code readability.  */
324    /* When switching a LED on, the digital line is actually turned off,    */
325    /* and vice versa.                                                      */
326    /* For both, function is the digital line number (defined above as a    */
327    /* 'function' eg. TRANSMITTING).                                        */
328
329    void switch_on( int function )
330
331    {
332
333    turnoff( function );
334
335    }
336
337
338    void switch_off( int function )
339
340    {
341
342    turnon( function );
343
344    }
345
346
347    /* This function is called from the presure_input function, when the    */
348    /* operator selects option 1 (record results) from the options menu.    */
349    /* This function is then called with the 'function' variable being      */
350    /* RESULTS_FILE.  In fact, DATA_FILE is never called, but was originally*/
351    /* used for debugging purposes.                                         */
352
353    int save_file( FILE *fptr, int function )
354
355    {
356    int      i, temp_i, outcome=UNSUCCESSFUL, year, month, d_month,
```

```
357            hour, min, sec, temp, max_value, msb;
358    unsigned char *temporary;
359
360    switch( function ) {
361    case RESULTS_FILE:
362            outcome = SUCCESSFUL;
363            cur_result1 = (unsigned char *) results1 + results1 -> start;
364            temporary = (unsigned char *) results1 + results1 -> start;
365
366            year = (*cur_result1 >> 1) & 127;
367            month = (*cur_result1 & 1) << 3;
368            cur_result1++;
369            month += ((*cur_result1 & 224) >> 5);
370            d_month = *cur_result1 & 31;
371
372            cur_result1++;
373            temporary = cur_result1;
374            temporary++;
375
376            fprintf(fptr,"%d\r\n",*legs_monitored);
377            fprintf(fptr,"%d\r\n",*cur_event);
378
379
380            /* The while loop below continues until 255 255 is reached in   */
381            /* the file (which is the end of file marker.                   */
382
383            while ( (*cur_result1!=255) && (*temporary!=255) ) {
384                    hour = *cur_result1 >> 3;
385                    min = (*cur_result1 & 7) << 3;
386                    cur_result1++;
387                    min += (*cur_result1 & 224) >> 5;
388                    sec = (*cur_result1 & 31) * 2;
389                    if ( sec>59 ) {
390                            sec = 59;
391                            }
392                    cur_result1++;
393
394                    fprintf(fptr,"%d ",hour);
395                    fprintf(fptr,"%d ",min);
396                    fprintf(fptr,"%d ",sec);
397
398                    max_value = (*cur_result1 >> 1) & 127;
399                    msb = *cur_result1 & 1;
400
401                    fprintf(fptr,"%d ",max_value);
402                    fprintf(fptr,"%d ",msb);
403                    cur_result1++;
404
405                    for ( temp=0; temp<2; temp++ ) {
406                            fprintf(fptr,"%d ",*cur_result1);
407                            cur_result1++;
408                            }
409
410                    fprintf(fptr,"\r\n");
411
412                    temporary = cur_result1;
413                    temporary++;
414
415                    }
416
417            fprintf(fptr,"999");     /* This is the end of file marker for  */
418                            /* the PC data file.                    */
419
420            backup(results1,0);     /* These lines delete the Results1.dat  */
421            unfix("Results1.dat");  /* file, and open a new one (effectively*/
422            open_results_file(1);   /* just deleting the old contents.      */
423
424
425            /* Opening a new results file also writes 255 255 as the first  */
426            /* 2 unsigned char numbers.  Therefore the start for new data    */
427            /* for the results file is already incremented twice with the    */
428            /* end of file marker.  Therefore it is decremented twice so     */
```

```
429         /* that new results data can be written from the start of the    */
430         /* file.                                                          */
431
432         cur_result1--;
433         cur_result1--;
434
435         fprintf(fptr,"%c", 0x0D);
436         fprintf(fptr,"%c", 0x0A);
437         fprintf(fptr,"%c", EOF);
438
439         printf("\r\n");
440
441         break;
442
443  case DATA_FILE:
444         outcome = SUCCESSFUL;
445         cur_data1 = (unsigned char *) data1 + data1 -> start;
446
447         for ( i=0; i<DATA_SIZE; i++ ) {
448              if ( (i%20)==0 ) {
449                   fprintf(fptr,"%c", 0x0D);
450                   fprintf(fptr,"%c", 0x0A);
451                   }
452              fprintf(fptr,"%d ",*cur_data1);
453              cur_data1++;
454              }
455
456         fprintf(fptr,"%c", 0x0D);
457         fprintf(fptr,"%c", 0x0A);
458         fprintf(fptr,"%c", EOF);
459         break;
460
461         }
462
463  return ( outcome );      /* 'outcome' is SUCCESSFUL if either of the 2    */
464                          /* case branches have been entered; otherwise it*/
465                          /* is UNSUCCESSFUL.                             */
466
467  }
468
469
470  /* This function is called from the main() function, after the           */
471  /* pressure_input function has been called.  This function analyses the */
472  /* data file, storing the results file the information for any events    */
473  /* that occur.                                                           */
474
475  void calc_results()
476
477  {
478  int     data_counts, i, c, max_value=0, counts, start=-1, finish=0,
479          year, yday, month, sec_temp, min_temp, hour_temp, day_month,
480          isdst, hour, min, sec;
481  unsigned char ctime1, ctime2, ctime3;
482  float time_inc;
483  unsigned char m_days[]={31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
484
485
486  /* First a test is performed to ensure that the Results file exists for */
487  /* the number of legs specified (currently there can only be 1).        */
488
489  if ( (*legs_monitored==1) && ((test=link("Results1.dat"))!=NULL) ) {
490         time_inc = time_increment();     /* The inter-sample time */
491
492         if (*display==ON)
493              printf("\r\nTime inc:%f ", time_inc);
494
495         cur_data1 = (unsigned char *) data1 + data1 -> start;
496
497         year = *cur_data1;
498         cur_data1++;
499         month = *cur_data1;
500         cur_data1++;
```

```
501         day_month = *cur_data1;
502         cur_data1++;
503         hour = *cur_data1;
504         cur_data1++;
505         min = *cur_data1;
506         cur_data1++;
507         sec = *cur_data1;
508         cur_data1++;
509
510         sec_temp = sec;
511         min_temp = min;
512         hour_temp = hour;
513         start=-1; finish=0;
514
515         data_counts = DATA_SIZE-12;       /* -12 because the first 6 and  */
516                                  /* last 6 bytes are used for    */
517                                  /* the start and finish time    */
518                                  /* stamps.                      */
519
520         for ( i=0; i<data_counts; i++ ) {
521             if ( start == -1 )       /* start==-1 when current        */
522                              /* position is not in an event. */
523                 counts=0;            /* So current number of samples */
524                              /* comprising the event is 0.   */
525
526             if ( (*cur_data1 > *cur_event) && (start==-1) ) {
527             /* ie the start of an event */
528                 start = i;       /* The position of the start of */
529                              /* the event in the file.       */
530                 counts = 1;      /* Currently event comprises of */
531                              /* 1 samples.                   */
532             }
533
534             else if ( *cur_data1 > *cur_event ) {
535             /* ie already in an event */
536                 if ( i==data_counts-1 ) /* If at the end of the */
537                                  /* data file.           */
538                     finish = start+counts;
539
540                 else     /* Otherwise increment the number of    */
541                     /* samples comprising the event.        */
542                     counts++;
543             }
544
545             else if ( (*cur_data1 < *cur_event) && (start!=-1) )
546             /* ie the end of an event */
547                 finish = start+counts;
548
549
550             if ( finish != 0 ) {
551             /* ie an event has just finished */
552
553                 /* First rewind pointer to the start of the event*/
554                 for( c=0; c<counts; c++ ) {
555                     cur_data1--;
556                 }
557
558                 /* Next obtain the peak for the event.  */
559                 for ( c=0; c<counts; c++ ) {
560                     if ( *cur_data1 > max_value )
561                         max_value = *cur_data1;
562                     cur_data1++;
563                 }
564
565                 /* '.._temp' has previously been set to the time*/
566                 /* at the start of the data file.               */
567
568                 /* Now set 'sec' to the middle sample of the    */
569                 /* event.  This might take it over 60, so after */
570                 /* 'min', 'hour', 'yday', 'year' are incremented*/
571                 /* as required.                                 */
572                 sec = sec_temp+((finish-(counts/2))*time_inc);
```

```
573                         min = min_temp;
574                         hour = hour_temp;
575                         while ( sec > 59 ) {
576                                 sec -= 60;
577                                 min += 1;
578                                 }
579                         while ( min > 59 ) {
580                                 min -= 60;
581                                 hour += 1;
582                                 }
583
584                         /* This calculates the yday. */
585                         yday = 0; c=0;
586                         while( month != (c+1) ) {
587                                 yday += m_days[c];
588                                 c++;
589                                 }
590                         yday += day_month;
591
592                         while ( hour > 23 ) {
593                                 hour -= 24;
594                                 yday++;
595                                 }
596
597                         while ( yday > 365 ) {
598                                 yday -= 365;
599                                 year++;
600                                 }
601
602                         /* The time stamp information is now compressed */
603                         /* from 3 bytes to 2.                           */
604                         ctime1 = (unsigned char) hour;
605                         ctime1 = ctime1 << 3;
606                         ctime1 += (unsigned char) (min >> 3) & 7;
607                         ctime2 = (unsigned char) min & 7;
608                         ctime2 = ctime2 << 5;
609                         ctime2 += (unsigned char) ((sec/2) & 31);
610                         ctime3 = (max_value & 127) << 1;
611
612                         /* The top-most bit of the counts value is      */
613                         /* packed at the end of the 3rd byte.           */
614                         ctime3 += (counts >> 8) & 1;
615
616                         *cur_result1 = ctime1;
617                         cur_result1++;
618                         *cur_result1 = ctime2;
619                         cur_result1++;
620                         *cur_result1 = ctime3;
621                         cur_result1++;
622                         *cur_result1 = counts & 0xFF;    /* LSB of counts */
623                         cur_result1++;
624                         *cur_result1 = (unsigned char) (time_inc*100);
625                         cur_result1++;
626
627                         finish = 0;     /* Reset these variables to      */
628                         start = -1;     /* continue analysisng the data  */
629                         max_value = 0;  /* file for more events.         */
630
631                         }
632
633                 cur_data1++;
634
635             }
636
637     /* At the end of the current results data in the results file,  */
638     /* put 255 255 as the end of file marker.                       */
639     *cur_result1 = 255;
640     cur_result1++;
641     *cur_result1 = 255;
642     cur_result1--;
643
644     }
```

```
645
646    else {    /* If the Results1.dat file is not found, an error has occurred. */
647            switch_off( RECORDING );
648            switch_off( PC_LINK );
649            switch_off( TRANSMITTING );
650            switch_on( ERROR );
651
652            if( *legs_monitored==1 )
653                    printf("\r\n!!! ERROR !!! :- Results1.dat not found");
654
655            while(1) ;      /* Infinite loop, ie. programme halts at this point
656    */
657
658            }
659
660
661    }
662
663
664    /* This function calculates the inter-sample time. */
665
666    float time_increment()
667
668    {
669    int     i, start_year, start_month, start_day_month, start_isdst,
670            start_hour, start_min, start_sec, finish_year, finish_month,
671            finish_day_month, finish_isdst, finish_hour,
672            finish_min, finish_sec, hours, mins, secs;
673    unsigned char *file_time;
674    float time, inc_time;
675
676    file_time = (unsigned char *) data1 + data1 -> start;
677
678    start_year = *file_time;
679    file_time++;
680    start_month = *file_time;
681    file_time++;
682    start_day_month = *file_time;
683    file_time++;
684    start_hour = *file_time;
685    file_time++;
686    start_min = *file_time;
687    file_time++;
688    start_sec = *file_time;
689    file_time++;
690
691    for ( i=0; i<(DATA_SIZE-12); i++ )
692            file_time++;
693
694    finish_year = *file_time;
695    file_time++;
696    finish_month = *file_time;
697    file_time++;
698    finish_day_month = *file_time;
699    file_time++;
700    finish_hour = *file_time;
701    file_time++;
702    finish_min = *file_time;
703    file_time++;
704    finish_sec = *file_time;
705    file_time++;
706
707    /* If finish_hour<start_hour it means that the start was before      */
708    /* midnight with the finish being after.  Therefore add 24 to        */
709    /* finish_hour.                                                      */
710    if ( finish_hour < start_hour )
711            finish_hour += 24;
712
713    hours = finish_hour - start_hour;
714    mins = finish_min - start_min;
715    secs = finish_sec - start_sec;
716
```

```
717    time = (hours*60*60) + (mins*60) + secs;
718
719    /* The inter-sample time is the time duration for all the samples of    */
720    /* the data file divided by the number of samples.                      */
721    inc_time = time/(DATA_SIZE-12);
722
723    return ( inc_time );
724
725    }
726
727
728    /* This function is called from the pressure_input function.   It        */
729    /* calculates the mass from the pressure transducer reading.  This       */
730    /* occurs from its stored calibrated pressure transducer values for      */
731    /* masses applied in multiples of 5 kg.  It uses the interpolate         */
732    /* function to interpolate between these calibrated values to obtain the*/
733    /* corresponding mass for the inputted pressure transducer reading.      */
734
735    unsigned char get_adc( int no )
736
737    {
738    unsigned char value;
739    unsigned char ad[]={0,15,30,55,70,85,105,115,135,150,169,182,195,255};
740         /* These are the calibration pressure transducer values for each*/
741         /* multiple of 5 kg.                                            */
742
743    int flag=0, i;
744
745    value = adc( no );
746
747
748    /* If the pressure transducer values are either 0 or 255, then set to   */
749    /* 1 and 254 respectively so that the calibration values either side of */
750    /* can be obtained.                                                     */
751
752    if ( value==0 ) {
753         value=1;
754         flag=1;
755         }
756
757    /* It should not be possible to get a value above 255 but just in case
758    ...*/
759    else if ( value >= 255 ) {
760         value=254;
761         flag=1;
762         }
763
764
765    /* The following code calculates which calibration value is just above  */
766    /* the inputted value from the pressure transducers.  It then re-sets    */
767    /* the inputted values to 0 or 255 if required.                         */
768
769    for ( i=0; ad[i]<value; i++ ) ;
770    if ( flag==1 ) {
771         if ( value==254 )
772              value=255;
773         else
774              value=0;
775         }
776
777    value = interpolate( i, value, ad[i-1], ad[i] );
778
779    return( value );
780
781    }
782
783
784    unsigned char interpolate( int lower_val, unsigned char value,
785         unsigned char lower_calibration, unsigned char upper_calibration )
786
787    {
788    int i;
```

```
789     float answer;
790
791     /* During calibration it was found that a slight pressure was required  */
792     /* before the reading changed from 0.  This is added (0.275).           */
793
794     answer = 0.275+(5.0*(value-lower_calibration))
795                 / ((float)upper_calibration-lower_calibration);
796     answer += (lower_val-1)*5.0;
797
798     return( (unsigned char) answer );
799
800     }
801
802
803     /* It is in this function that the programme spends most time during     */
804     /* execution.  The function is called by the main() function after       */
805     /* having cleared the data files.                                        */
806
807     void pressure_input()
808
809     {
810     int sum, data_counts, i, temp, outcome=UNSUCCESSFUL, count, test, flag,
811             set_sec, set_min, set_hour, set_mday, set_mon,
812             set_year, set_wday, set_yday, set_isdst;
813     unsigned char adc0, adc1, adc2, adc3;
814     unsigned char *temp1_results;
815     unsigned char *temp2_results;
816     char file_name[10],input;
817     unsigned char m_days[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
818     static float calibrate_value=1.0;
819     FILE *fptr;
820
821     /* First a test is performed to ensure that there is a data file present. 
822     */
823
824     if ( (*legs_monitored==1) && ((data1=link("Data1.dat"))!=NULL) ) {
825             switch_on( RECORDING );
826             getime(cur_time);
827
828
829             /* The start of the data file is time stamped. */
830
831             cur_data1 = (unsigned char *) data1 + data1 -> start;
832             *cur_data1  = cur_time -> tm_year;
833             cur_data1++;
834             *cur_data1 = cur_time -> tm_mon;
835             cur_data1++;
836             *cur_data1 = cur_time -> tm_mday;
837             cur_data1++;
838             *cur_data1 = cur_time -> tm_hour;
839             cur_data1++;
840             *cur_data1 = cur_time -> tm_min;
841             cur_data1++;
842             *cur_data1 = cur_time -> tm_sec;
843             cur_data1++;
844
845             data_counts = DATA_SIZE-12;      /* -12 because there are 2 time */
846                                             /* stamps (one at the beginning */
847                                             /* and the other at the end)    */
848                                             /* each taking 6 bytes each.    */
849
850             for ( i=0; i<data_counts; i++ ) {
851                     adc0 = (unsigned char) ((get_adc(0))*(*calibrate_value0)/100);
852                     adc1 = (unsigned char) ((get_adc(1))*(*calibrate_value1)/100);
853
854
855                     /* The following decrements are needed for when no load */
856                     /* is applied on the transducers, a value of 1 or 2      */
857                     /* would be read from the A/D 1.                         */
858
859                     if( adc1>0 )
860                             adc1--;
```

```
861     if( adc1>0 )
862             adc1--;
863
864
865     adc2 = (unsigned char) ((get_adc(2))*(*calibrate_value2)/100);
866     adc3 = (unsigned char) ((get_adc(3))*(*calibrate_value3)/100);
867
868
869     /* Only one decrement is required for A/D 2 for its no  */
870     /* load value would be 0 or 1.                          */
871
872     if( adc2>0 )
873             adc2--;
874
875
876     /* By using 'sum' which is an int, the summation is     */
877     /* forced to be an int so causing there to be no        */
878     /* 'wrap-around' if the sum was greater than 255.       */
879
880     sum = adc0;
881     sum = sum + adc1;
882     sum = sum + adc2;
883     sum = sum + adc3;
884
885
886     /* If 'sum' is greater than 255, then set it to 255 so  */
887     /* it can be stored in 1 byte.  This gives the monitor a*/
888     /* range of up to 255 kg (more than enough)!            */
889
890     if( sum > 255 )
891             sum = 255;
892
893     /* The transducers' sum (kg) is then stored in the data */
894     /* file.                                                */
895     *cur_data1 = sum;
896
897
898     /* If the display toggle (accessed from the operator    */
899     /* menu is ON, then print 'sum' on the screen.          */
900
901     if ( *display==ON ) {
902             printf(" %d",*cur_data1);
903
904             /* A delay is now required, otherwise there is a*/
905             /* danger of the RS-485 buffer overflowing,     */
906             /* resulting with the display, keyboard and file*/
907             /* storage system hanging.                      */
908
909             delay(1);
910             }
911
912     /* The data file pointer is now incremented so that it  */
913     /* points to the next free byte.                        */
914
915     cur_data1++;
916
917
918     /* A delay is specified to slow the sampling rate, or it*/
919     /* would be in the kilo(?) Hertz range. */
920
921     delay(1);
922
923
924     /* If the operator has specified that the processor     */
925     /* should be powered down in between samples, then do so*/
926     /* for 0.1 seconds (excluding time for re-starting it.  */
927
928     if ( *power_down==ON )
929             sleep( 10, HUNDREDTHS_OF_SECS );
930
931
932     /* Test for the PC switch on the monitor having been    */
```

```
933                      /* triggered (ie requesting a PC link).              */
934
935              if ( ch(SWITCH_PC_LINK)==0 ) {
936                      switch_off( RECORDING );
937                      switch_on( PC_LINK );
938
939                      /* It might have been depressed by accident.    */
940                      /* Therefore check for a key being depressed on */
941                      /* the keyboard to confirm.                     */
942
943                      outcome = link_test();
944                      if ( outcome == SUCCESSFUL ) {
945
946                              /* Clear the key depressed from the     */
947                              /* buffer.                              */
948
949                              scanf("%c", &input);
950
951
952                              /* Print on the screen the options menu*/
953
954                              printf("\r\n\n Possible options are:");
955                              printf("\r\n 1: Record results");
956                              printf("\r\n 2: Edit event level");
957                              printf("\r\n 3: Calibrate transducers");
958                              printf("\r\n 4: Restart ""Results1.dat""
959    module");
960                              printf("\r\n 5: Power down processor ON/OFF");
961                              printf("\r\n 6: List results to date");
962                              printf("\r\n 7: Go to shell program");
963                              printf("\r\n 8: Real time clock");
964                              printf("\r\n 9: Display toggle");
965                              printf("\r\n\n Please input a number (1-9): ");
966
967
968                              /* Put a delay to let the serial port   */
969                              /* catch up with the program.           */
970
971                              delay(50);
972
973
974                              /* Get the operator input.              */
975
976                              scanf("%c", &input);
977                              printf("\r\n");
978
979
980                              /* If option '1' is chosen, save the    */
981                              /* results file onto the PC disk.  Then */
982                              /* clear the results and data files from*/
983                              /* memory and restart monitoring.       */
984
985                              if ( input=='1' ) {
986                                      switch_on( TRANSMITTING );
987
988                                      /* For ease of programmer change*/
989                                      /* the PC file's name is        */
990                                      /* abstracted into a separate   */
991                                      /* function (so can easily be   */
992                                      /* changed for the whole        */
993                                      /* programme).                  */
994
995                                      get_outfile_name( outfile_name );
996                                      printf("\r\nSaving 'Results1.dat' module
997    as '%s' on the hard disk.\r\n\n",outfile_name);
998
999                                      fptr = fopen(outfile_name,"w");
1000                                     outcome = save_file( fptr, RESULTS_FILE );
1001                                     fclose(fptr);
1002
1003
1004                                     /* Perform a kind of reset.      */
```

```
1005                                    /* Delete the data and results  */
1006                                    /* files, and create new ones.   */
1007                                    /* Also reset the loop variable */
1008                                    /* and data/results files        */
1009                                    /* pointers after time stamping.*/
1010
1011                                    /* First delete the data file.  */
1012
1013                                    backup(data1,0);
1014                                    unfix("Data1.dat");
1015
1016
1017                                    /* Next create a new one and     */
1018                                    /* time stamp it.                */
1019
1020                                    open_data_file(1);
1021
1022                                    getime(cur_time);
1023                                    cur_data1 = (unsigned char *) data1 +
1024    data1 -> start;
1025                                    *cur_data1  = cur_time -> tm_year;
1026                                    cur_data1++;
1027                                    *cur_data1 = cur_time -> tm_mon;
1028                                    cur_data1++;
1029                                    *cur_data1 = cur_time -> tm_mday;
1030                                    cur_data1++;
1031                                    *cur_data1 = cur_time -> tm_hour;
1032                                    cur_data1++;
1033                                    *cur_data1 = cur_time -> tm_min;
1034                                    cur_data1++;
1035                                    *cur_data1 = cur_time -> tm_sec;
1036                                    cur_data1++;
1037
1038
1039                                    /* Next delete the results file. */
1040
1041                                    backup(results1,0);
1042                                    unfix("Results1.dat");
1043
1044
1045                                    /* And create a new one.         */
1046
1047                                    open_results_file(1);
1048
1049                                    cur_result1--;
1050                                    cur_result1--;
1051                                            /* as created with 255 255 */
1052
1053                                    /* Finally reset the loop variable */
1054
1055                                    i=0;
1056
1057                                    switch_off( TRANSMITTING );
1058
1059                                    }
1060
1061
1062                            /* If option '2' is selected, the event */
1063                            /* level is printed on the screen, and  */
1064                            /* the operator has the option to change*/
1065                            /* its value.                           */
1066
1067                            else if ( input=='2' ) {
1068                                    printf("\r\nCurrent event level set at %d
1069    kg.",*cur_event);
1070                                    printf("\r\nDo you wish to change this
1071    level? (Y/N) ");
1072                                    scanf("%c",&input);
1073                                    if ( (input=='Y')||(input=='y') ) {
1074                                            printf("\r\nEnter new level: ");
1075                                            scanf("%d",&test);
1076                                            while ( (test>65)||(test<0) ) {
```

```
1077                                               printf("\r\nRange is 0 to 65
1078    kg. Please try again: ");
1079                                                   scanf("%d",&test);
1080                                                   }
1081                                               *cur_event = (unsigned char) test;
1082                                               printf("\r\n\nLevel set at %d
1083    kg.\r\n",*cur_event);
1084                                           }
1085                                       else {
1086                                           printf("\r\n");
1087                                           }
1088
1089                                       /* A delay is required to        */
1090                                       /* ensure that the serial port   */
1091                                       /* has caught up with the        */
1092                                       /* programme.                    */
1093
1094                                       delay(10);
1095                                       }
1096
1097
1098                           /* If option '3' is selected, the       */
1099                           /* operator can view the individual     */
1100                           /* transducers' input value and change  */
1101                           /* each one's scaling value.            */
1102
1103                           else if ( input=='3' ) {
1104                                   printf("\r\nWhich transducer do you wish
1105    to calibrate:");
1106                                   printf("\r\nCalcaneus(1), First Metatarsal
1107    Head (2), Fifth Metatarsal Head (3),");
1108                                   printf("\r\nThird Metatarsal Head (4), or
1109    monitor all three at once (5): ");
1110                                   scanf("%c",&input);
1111                                   printf("%c\r\n",input);
1112                                   fflush(stdin);
1113
1114                                   if ( input=='1' ) {
1115                                           printf("Current calibration value is
1116    %d.\r\n",*calibrate_value0);
1117                                           delay(50);
1118                                           flag=OFF;
1119                                           count=0;
1120                                           while( flag==OFF ) {
1121                                                   printf("%d ",(unsigned char)
1122    ((get_adc(0))*(*calibrate_value0)/100) );
1123                                                   delay(4);
1124                                                   sleep( 10, HUNDREDTHS_OF_SECS
1125    );
1126                                                   if ( count==50 ) {
1127                                                           printf("\r\nInput 1 to
1128    continue, 2 to change calibration value, 3 to end calibration: ");
1129                                                           scanf("%c",&input);
1130                                                           printf("%c\r\n",input);
1131                                                           if ( input=='1' )
1132                                                                   count=0;
1133                                                           else if ( input=='2' )
1134    {
1135                                                                   printf("\r\nSet
1136    calbration value * 100 (currently %d) to (max 255): ",*calibrate_value0);

1138            scanf("%d",&test);

1140            *calibrate_value0 = (unsigned char) test;

1142            printf("\r\nCalibration value set at
1143    %.2f\r\n",(*calibrate_value0/100.0));
1144                                                                   count=0;
1145                                                                   }
1146                                                           else
1147                                                                   flag=ON;
1148
```

```
1149                                                        printf("\r\n");
1150                                                        }
1151
1152                                             count++;
1153
1154                                             }
1155
1156                                    }
1157
1158
1159                          else if ( input=='2' ) {
1160                                    printf("Current calibration value is
1161    %d.\r\n",*calibrate_value1);
1162                                             delay(50);
1163                                             flag=OFF;
1164                                             count=0;
1165                                             while( flag==OFF ) {
1166                                                      adc0 = (unsigned char)
1167    ((get_adc(1))*(*calibrate_value1)/100);
1168                                                      if( adc0>0 )
1169                                                              adc0--;
1170                                                      if( adc0>0 )
1171                                                              adc0--;
1172
1173                                                      printf("%d ", adc0);
1174
1175                                                      delay(4);
1176                                                      sleep( 10, HUNDREDTHS_OF_SECS
1177    );
1178                                                      if ( count==50 ) {
1179                                                              printf("\r\nInput 1 to
1180    continue, 2 to change calibration value, 3 to end calibration: ");
1181                                                              scanf("%c",&input);
1182                                                              printf("%c\r\n",input);
1183
1184                                                              if ( input=='1' )
1185                                                                      count=0;
1186                                                              else if ( input=='2' )
1187    {
1188                                                                      printf("\r\nSet
1189    calbration value * 100 (currently %d) to (max 255): ",*calibrate_value1);
1190
1191          scanf("%d",&test);
1192
1193          *calibrate_value1 = (unsigned char) test;
1194
1195          printf("\r\nCalibration value set at
1196    %.2f\r\n",(*calibrate_value1/100.0));
1197                                                                      count=0;
1198                                                                      }
1199                                                              else
1200                                                                      flag=ON;
1201
1202                                                              printf("\r\n");
1203                                                              }
1204
1205                                                      count++;
1206
1207                                                      }
1208
1209                                             }
1210
1211
1212                          else if( input=='3' ) {
1213                                    printf("Current calibration value is
1214    %d.\r\n",*calibrate_value2);
1215                                             delay(50);
1216                                             flag=OFF;
1217                                             count=0;
1218                                             while( flag==OFF ) {
1219                                                      adc0 = (unsigned char)
1220    ((get_adc(2))*(*calibrate_value2)/100);
```

```
1221                                                    if( adc0>0 )
1222                                                            adc0--;
1223
1224                                                    printf("%d ", adc0);
1225
1226                                                    delay(4);
1227                                                    sleep( 10, HUNDREDTHS_OF_SECS
1228    );
1229                                                    if ( count==50 ) {
1230                                                            printf("\r\nInput 1 to
1231    continue, 2 to change calibration value, 3 to end calibration: ");
1232                                                            scanf("%c",&input);
1233                                                            printf("%c\r\n");
1234
1235                                                            if ( input=='1' )
1236                                                                    count=0;
1237                                                            else if ( input=='2' )
1238    {
1239                                                                    printf("\r\nSet
1240    calbration value * 100 (currently %d) to (max 255): ",*calibrate_value2);
1241
1242            scanf("%d",&test);
1243
1244            *calibrate_value2 = (unsigned char) test;
1245
1246            printf("\r\nCalibration value set at
1247    %.2f\r\n",(*calibrate_value2/100.0));
1248                                                                    count=0;
1249                                                                    }
1250                                                            else
1251                                                                    flag=ON;
1252
1253                                                            printf("\r\n");
1254                                                            }
1255                                                    count++;
1256
1257                                                    }
1258
1259                                            }
1260
1261
1262                            else if( input=='4' ) {
1263                                                    printf("Current calibration value is
1264    %d.\r\n",*calibrate_value3);
1265                                                    delay(50);
1266                                                    flag=OFF;
1267                                                    count=0;
1268                                                    while( flag==OFF ) {
1269                                                            adc0 = (unsigned char)
1270    ((get_adc(3))*(*calibrate_value3)/100);
1271
1272                                                            printf("%d ", adc0);
1273
1274                                                            delay(4);
1275                                                            sleep( 10, HUNDREDTHS_OF_SECS
1276    );
1277                                                            if ( count==50 ) {
1278                                                                    printf("\r\nInput 1 to
1279    continue, 2 to change calibration value, 3 to end calibration: ");
1280                                                                    scanf("%c",&input);
1281                                                                    printf("%c\r\n");
1282
1283                                                                    if ( input=='1' )
1284                                                                            count=0;
1285                                                                    else if ( input=='2' )
1286    {
1287                                                                            printf("\r\nSet
1288    calbration value * 100 (currently %d) to (max 255): ",*calibrate_value3);
1289
1290            scanf("%d",&test);
1291
1292            *calibrate_value3 = (unsigned char) test;
```

```
1293
1294          printf("\r\nCalibration value set at
1295    %.2f\r\n",(*calibrate_value3/100.0));
1296                                                  count=0;
1297                                              }
1298                                        else
1299                                            flag=ON;
1300
1301                                    printf("\r\n");
1302                                }
1303                            count++;
1304
1305                        }
1306
1307                    }
1308
1309
1310                    /* Whilst the above options    */
1311                    /* deal with just one transducer*/
1312                    /* value at a time, the last    */
1313                    /* option prints all 4 values   */
1314                    /* simultaneously.              */
1315
1316                    else if( input=='5' ) {
1317                        delay(50);
1318                        flag=OFF;
1319                        count=0;
1320                        while( flag==OFF ) {
1321                            adc0 = (unsigned char)
1322    ((get_adc(0))*(*calibrate_value0)/100);
1323                            adc1 = (unsigned char)
1324    ((get_adc(1))*(*calibrate_value1)/100);
1325                            adc2 = (unsigned char)
1326    ((get_adc(2))*(*calibrate_value2)/100);
1327                            adc3 = (unsigned char)
1328    ((get_adc(3))*(*calibrate_value3)/100);
1329                            if( adc1>0 )
1330                                adc1--;
1331                            if( adc1>0 )
1332                                adc1--;
1333                            if( adc2>0 )
1334                                adc2--;
1335
1336                            printf("%d;%d,%d,%d    ",
1337    adc0,adc1,adc3,adc2);
1338
1339                            delay(4);
1340                            sleep( 10, HUNDREDTHS_OF_SECS
1341    );
1342                            if ( count==100 ) {
1343                                printf("\r\nInput 1 to
1344    continue, 2 to end monitoring: ");
1345                                scanf("%c",&input);
1346                                printf("%c\r\n");
1347
1348                                if ( input=='1' )
1349                                    count=0;
1350                                else
1351                                    flag=ON;
1352
1353                            }
1354                        count++;
1355
1356                        }
1357
1358                    }
1359                else
1360                    printf("\r\nNo transducer chosen.");
1361
1362                delay(50);
1363
1364            }
```

```
1365
1366
1367                              /* If option '4' is selected, the old    */
1368                              /* results file is cleared and a new one*/
1369                              /* started (so that old results data is */
1370                              /* effectively deleted).  This is useful*/
1371                              /* so that the operator can be sure that*/
1372                              /* when starting a monitoring session    */
1373                              /* the date of the results file will be */
1374                              /* correct.                              */
1375
1376                              else if ( input=='4' ) {
1377                                      printf("\r\nStarting new 'Results1.dat'
1378      module with today's date.\r\n\n");
1379                                      backup(results1,0);
1380                                      unfix("Results1.dat");
1381                                      open_results_file(1);
1382                                      cur_result1--;
1383                                      cur_result1--;
1384                                      delay(10);
1385
1386                                      }
1387
1388
1389                              /* If option '5' is selected, the        */
1390                              /* current setting for the power-down    */
1391                              /* toggle is displayed, and the operator*/
1392                              /* has the option of changing it.        */
1393
1394                              else if ( input=='5' ) {
1395                                      printf("\r\nPower down toggle is currently
1396      ");
1397                                      if ( *power_down==ON )
1398                                              printf("ON.");
1399                                      else
1400                                              printf("OFF.");
1401
1402                                      printf("\r\nDo you want to change this
1403      setting? (Y/N) ");
1404                                      scanf("%c",&input);
1405                                      if ( (input=='y')||(input=='Y') ) {
1406                                              if ( *power_down==ON )
1407                                                      *power_down=OFF;
1408                                              else
1409                                                      *power_down=ON;
1410                                              }
1411
1412                                      }
1413
1414
1415                              /* If option '6' is selected, the data  */
1416                              /* currently held in the results file   */
1417                              /* is displayed.  Therefore all bytes   */
1418                              /* are displayed up to 255 255 which is */
1419                              /* the end of file marker.              */
1420
1421                              else if ( input=='6' ) {
1422                                      switch_on( TRANSMITTING );
1423
1424                                      printf("\r\nData in results1 file is as
1425      follows:\r\n\n");
1426                                      temp1_results = (unsigned char *) results1
1427      + results1->start;
1428                                      printf("%d ",*temp1_results);
1429                                      temp1_results++;
1430                                      printf("%d\r\n",*temp1_results);
1431                                      temp1_results++;
1432                                      temp2_results = temp1_results;
1433                                      temp2_results++;
1434                                      temp=0;
1435                                      while (
1436      (*temp1_results!=255)&&(*temp2_results!=255) ) {
```

```
1437                                              printf("%d ",*temp1_results);
1438                                              temp1_results++;
1439                                              temp2_results++;
1440                                              temp++;
1441                                              if ( (temp%5)==0 )
1442                                                     printf("\r\n");
1443                                              if ( temp<100 )
1444                                                     delay(1);
1445                                              }
1446
1447                                      delay(50);
1448                                      switch_off( TRANSMITTING );
1449                                      }
1450
1451
1452                      /* selecting option '7' enters the       */
1453                      /* operator in the shell programme.      */
1454                      /* This option is useful for debugging  */
1455                      /* purposes, but was left in so that if */
1456                      /* needed the operator could check as to*/
1457                      /* whether the various data files had    */
1458                      /* been instantiated as required.        */
1459
1460                      else if ( input=='7' ) {
1461                             shell();
1462                             }
1463
1464
1465                      /* Selecting option '8' prints the       */
1466                      /* current date and time, with the       */
1467                      /* operator having the option to change */
1468                      /* it.                                    */
1469
1470                      else if ( input=='8' ) {
1471                             getime(cur_time);
1472                             printf("\r\n\nCurrent settings are:");
1473                             printf("\r\n\nDate: %d/%d/%d",cur_time-
1474  >tm_mday, cur_time->tm_mon, ((cur_time->tm_year)-1));
1475                             printf("\r\nTime: %d:%d:%d",cur_time-
1476  >tm_hour, cur_time->tm_min, cur_time->tm_sec);
1477                             printf("\r\n\nDo you want to change the
1478  settings? (Y/N)");
1479                             scanf("%c",&input);
1480                             printf("%c\r\n",input);
1481
1482                             if ( (input=='Y')||(input=='y') ) {
1483                                    flag=OFF;
1484                                    while( flag==OFF ) {
1485                                           printf("\r\nInput hour: ");
1486                                           scanf("%d",&test);
1487                                           if ( (test>-1) && (test<24) )
1488                                                  flag=ON;
1489                                           else
1490                                                  printf("\r\nOh, really
1491  ??!!");
1492                                           }
1493                                    set_hour = test;
1494                                    flag=OFF;
1495                                    while( flag==OFF ) {
1496                                           printf("\r\nInput minute: ");
1497                                           scanf("%d",&test);
1498                                           if ( (test>-1) && (test<60) )
1499                                                  flag=ON;
1500                                           else
1501                                                  printf("\r\nOh, really
1502  ??!!");
1503                                           }
1504                                    set_min = test;
1505                                    set_sec = 0;
1506                                    flag=OFF;
1507                                    while( flag==OFF ) {
```

```
1508                                                   printf("\r\nInput day of
1509   month: ");
1510                                                   scanf("%d", &test);
1511                                                   if ( (test>0) && (test<32) )
1512                                                           flag=ON;
1513                                                   else
1514                                                           printf("\r\nOh, really
1515   ??!!");
1516                                                   }
1517                                           set_mday = test;
1518                                           flag=OFF;
1519                                           while( flag==OFF ) {
1520                                                   printf("\r\nInput month of
1521   the year (1-12): ");
1522                                                   scanf("%d",&test);
1523                                                   if ( (test>0) && (test<13) )
1524                                                           flag=ON;
1525                                                   else
1526                                                           printf("\r\nOh, really
1527   ??!!");
1528                                                   }
1529                                           set_mon = test;
1530                                           flag=OFF;
1531                                           while( flag==OFF ) {
1532                                                   printf("\r\nInput year (year-
1533   1900): ");
1534                                                   scanf("%d",&test);
1535                                                   if ( (test>-1) && (test<100)
1536   )
1537                                                           flag=ON;
1538                                                   else
1539                                                           printf("\r\nOh, really
1540   ??!!");
1541                                                   }
1542                                           set_year = test;
1543                                           flag=OFF;
1544                                           while( flag==OFF ) {
1545                                                   printf("\r\nInput day of week
1546   (Sunday = 0): ");
1547                                                   scanf("%d",&test);
1548                                                   if ( (test>-1) && (test<8) )
1549                                                           flag=ON;
1550                                                   else
1551                                                           printf("\r\nOh, really
1552   ??!!");
1553                                                   }
1554                                           set_wday = test;
1555                                           flag = OFF;
1556                                           while( flag==OFF ) {
1557                                                   printf("\r\nInput daylight
1558   saving time (0, 1): ");
1559                                                   scanf("%d",&test);
1560                                                   if ( (test>-1) && (test<2) )
1561                                                           flag=ON;
1562                                                   else
1563                                                           printf("\r\nOh, really
1564   ??!!");
1565                                                   }
1566                                           set_isdst = test;
1567                                           test = 1;
1568                                           set_yday = 0;
1569                                           while ( test != set_mon ) {
1570                                                   set_yday += m_days[test-1];
1571                                                   test++;
1572                                                   }
1573
1574                                           cur_time->tm_sec = set_sec;
1575                                           cur_time->tm_min = set_min;
1576                                           cur_time->tm_hour = set_hour;
1577                                           cur_time->tm_mday = set_mday;
1578                                           cur_time->tm_mon = set_mon;
1579                                           cur_time->tm_year = set_year;
```

```
1580                                                cur_time->tm_wday = set_wday;
1581                                                cur_time->tm_yday - set_yday;
1582                                                cur_time->tm_isdst = set_isdst;
1583
1584                                                setime( cur_time );
1585
1586                                                }
1587
1588
1589                                        }
1590
1591
1592                                /* Finally, selecting option '9' prints*/
1593                                /* the current power-down setting, with */
1594                                /* the option to change it.           */
1595
1596                                else if ( input=='9' ) {
1597                                        printf("\r\nDisplay toggle is currently
1598  ");
1599                                        if ( *display==ON )
1600                                                printf("ON.");
1601                                        else
1602                                                printf("OFF.");
1603
1604                                        printf("\r\nDo you want to change the
1605  setting? (Y/N) ");
1606                                        fflush(stdin);
1607                                        scanf("%c",&input);
1608                                        if ( (input=='Y')||(input=='y') ) {
1609                                                if ( *display==ON )
1610                                                        *display=OFF;
1611                                                else
1612                                                        *display=ON;
1613                                                }
1614                                        }
1615
1616
1617                                }
1618
1619                        switch_off(PC_LINK);
1620                        switch_on(RECORDING);
1621
1622                        }
1623
1624                }
1625        outcome = SUCCESSFUL;
1626
1627
1628        /* Finally at the end of the filling of the data file, the        */
1629        /* current date and time is again recorded (for the calculating */
1630        /* of the inter-sample time),                                    */
1631
1632        getime(cur_time);
1633
1634        *cur_data1  = cur_time -> tm_year;
1635        cur_data1++;
1636        *cur_data1 = cur_time -> tm_mon;
1637        cur_data1++;
1638        *cur_data1 = cur_time -> tm_mday;
1639        cur_data1++;
1640        *cur_data1 = cur_time -> tm_hour;
1641        cur_data1++;
1642        *cur_data1 = cur_time -> tm_min;
1643        cur_data1++;
1644        *cur_data1 = cur_time -> tm_sec;
1645        cur_data1++;
1646
1647
1648        /* It was thought beneficial to extend the functionality of      */
1649        /* this monitor programme to have the option of monitoring 1 or */
1650        /* 2 legs simultaneously (therefore using 4 or 2 transducers for*/
1651        /* each foot respectively).  Not enough time was found to        */
```

```
1652                 /* complete this extension, but the code that for now will never*/
1653                 /* be executed has been left to facilitate future work.           */
1654
1655             if( *legs_monitored==2 ) {
1656                     *cur_data2  = cur_time -> tm_year;
1657                     cur_data2++;
1658                     *cur_data2 = cur_time -> tm_mon;
1659                     cur_data2++;
1660                     *cur_data2 = cur_time -> tm_mday;
1661                     cur_data2++;
1662                     *cur_data2 = cur_time -> tm_hour;
1663                     cur_data2++;
1664                     *cur_data2 = cur_time -> tm_min;
1665                     cur_data2++;
1666                     *cur_data2 = cur_time -> tm_sec;
1667                     cur_data2++;
1668                     }
1669
1670             }
1671
1672     else    {
1673         switch_off( RECORDING );
1674         switch_off( PC_LINK );
1675         switch_off( TRANSMITTING );
1676         switch_on( ERROR );
1677
1678         /* Prints the appropriate error message for debugging purposes  */
1679         /* (as program execution should never arrive at this point).    */
1680
1681         printf("\r\n!!! ERROR !!! :- Data1.dat not found");
1682
1683         while(1) ;  /* Infinite loop, ie. programme halts at this point.*/
1684
1685         }
1686
1687     }
1688
1689
1690 /* This function builds up the file name for the file which will be     */
1691 /* saved to the PC disk using a pre-defined stub and the date of the    */
1692 /* results file.                                                        */
1693
1694 void get_outfile_name( char *file_name )
1695
1696 {
1697 int year, month, d_month, i, temp;
1698
1699 /* The results file date is stored at the start of the file. */
1700
1701 cur_result1 = (unsigned char *) results1 + results1->start;
1702
1703
1704 /* Then need to uncompress it.  */
1705
1706 year = (*cur_result1 >> 1) & 127;
1707
1708 printf("\r\nyear=%d\r\n",year); delay(5);
1709
1710 month = (*cur_result1 & 1) << 3;
1711 cur_result1++;
1712 month += ((*cur_result1 & 224) >> 5);
1713 d_month = *cur_result1 & 31;
1714
1715
1716 /* Finally build up the file name and return it via the inputted char   */
1717 /* pointer.                                                             */
1718
1719 *file_name = 'd';
1720 file_name++;
1721 *file_name = 'a';
1722 file_name++;
1723 *file_name = 't';
```

```
1724    file_name++;
1725    *file_name = (d_month/10) + 48;
1726    file_name++;
1727    *file_name = d_month-((d_month/10)*10) + 48;
1728    file_name++;
1729    *file_name = (month/10) + 48;
1730    file_name++;
1731    *file_name = month-((month/10)*10) + 48;
1732    file_name++;
1733    *file_name = year-((year/10)*10) + 48;
1734    file_name++;
1735    *file_name = '.';
1736    strcat( file_name, module_name );
1737
1738    }
1739
1740
1741    /* This function is called from the pressure_input function to check    */
1742    /* whether a key on the PC keyboard is depressed (after the PC switch on*/
1743    /* the monitor having been triggered).  The routine monitors the stdin  */
1744    /* stream for a short period of time, returning the SUCCESSFUL if a key  */
1745    /* has been depressed, and UNSUCCESSFUL if not.                          */
1746
1747    int link_test()
1748
1749    {
1750    int i, outcome=UNSUCCESSFUL;
1751    FILE *input;
1752
1753    input = stdin;
1754
1755    for ( i=0; i<500; i++ ) {
1756            if ( ready(fileno(stdin)) > 0 ) {
1757                    outcome = SUCCESSFUL;
1758                    }
1759
1760            delay(1);
1761
1762            }
1763
1764    return( outcome );
1765
1766    }
1767
1768
1769    /* This function is called from the main() function.  It deletes any     */
1770    /* data files present (creating new ones in their place) and creates new*/
1771    /* results file(s) if none are present.                                 */
1772
1773    void setup_datamods()
1774
1775    {
1776    int i=0, outcome;
1777    unsigned char *next_result;
1778
1779    /* If legs_monitored==2, then there should be 2 data/results files      */
1780    /* present (or 1 if previously only 1 leg was being monitored).         */
1781    /* Therefore first call open_data_file(1) which tries to create a new   */
1782    /* data file for 1 leg being monitored (successul only if no data files */
1783    /* are present) and then removes the data file for leg 1.               */
1784    /* Therefore 2 data files can now be created, and the programme has      */
1785    /* switched from having 1 data file to 2.                               */
1786
1787    if( *legs_monitored==2 ) {         /* This clears Data1.dat if present */
1788            open_data_file(1);
1789            backup(data1,0);
1790            unfix("Data1.dat");
1791            }
1792
1793    /* The required number of data files (currently always 1) are created.  */
1794    /* Returns SUCCESFUL only if no data files are present.                  */
1795
```

```
1796        outcome = open_data_file(*legs_monitored);
1797
1798        if ( outcome==UNSUCCESSFUL ) {         /* Therefore a data file is present. */
1799               /* If 2 legs are being monitored, data file for leg 1 has        */
1800               /* already been deleted.  Therefore just delete the data file    */
1801               /* for leg 2.                                                     */
1802               if( *legs_monitored==2 ) {
1803                      backup(data2,0);
1804                      unfix("Data2.dat");
1805                      }
1806
1807               else {                                   /* ie. legs monitored=1 */
1808                      backup(data1,0);
1809                      unfix("Data1.dat");
1810                      }
1811
1812
1813               /* Can now successfully create the required number of data files.*/
1814
1815               open_data_file(*legs_monitored);
1816
1817               }
1818
1819
1820        /* Set up the data file pointers to point to the start of the file.     */
1821
1822        if( *legs_monitored==2 )
1823               cur_data2 = (unsigned char *) data2 + data2 -> start;
1824        cur_data1 = (unsigned char *) data1 + data1 -> start;
1825
1826
1827        /* Now create the results files if none are already present. */
1828
1829        outcome = open_results_file(*legs_monitored);
1830
1831        if( *legs_monitored==2 ) {
1832               /* Move the results file pointer to the end of the recorded      */
1833               /* data by putting the file pointer to the start of the file and*/
1834               /* searching through it until the end of file marker is found    */
1835               /* (this being 255 255).                                         */
1836
1837               cur_result2 = (unsigned char *) results2 + results2 -> start;
1838               next_result = (unsigned char *) results2 + results2 -> start;
1839               next_result++;
1840               for (i=0;
1841        (!((*cur_result2==255)&&(*next_result==255)))&&(i<(RESULTS_SIZE/2)); i++) {
1842                      cur_result2++;
1843                      next_result++;
1844                      }
1845
1846               /* If the file memory is full, then can not store more results   */
1847               /* so stop monitoring, light the memory full LED, and print a    */
1848               /* relevant message.  If the monitor is not connected to the PC,*/
1849               /* the operator will just see the LED indication but will be     */
1850               /* also able to read the message indication by switching off the*/
1851               /* monitor, connecting to the PC and switching it back on (as    */
1852               /* this routine will be onw of the first executed when the       */
1853               /* programme is re-started).                                     */
1854
1855               if ( i == (RESULTS_SIZE/2) ) {
1856                      printf("\r\nMemory of Results2.dat is full...");
1857                      switch_on( MEMORY_FULL );
1858                      while(1) ;
1859                             /* Infinite loop, ie. programme halts at this point */
1860                      }
1861
1862               cur_result1 = (unsigned char *) results1 + results1 -> start;
1863               next_result = (unsigned char *) results1 + results1 -> start;
1864               next_result++;
1865               for (i=0;
1866        (!((*cur_result1==255)&&(*next_result==255)))&&(i<(RESULTS_SIZE/2)); i++) {
1867                      cur_result1++;
```

```
1868                          next_result++;
1869                          }
1870
1871              if ( i == (RESULTS_SIZE/2) ) {
1872                          /* Results1.dat is full, so stop recording */
1873                          printf("\r\nMemory of Results1.dat is full...");
1874                          switch_on( MEMORY_FULL );
1875                          while(1) ;
1876                                  /* Infinite loop, ie. programme halts at this point */
1877                          }
1878
1879              }
1880
1881      else {
1882              cur_result1 = (unsigned char *) results1 + results1 -> start;
1883              next_result = (unsigned char *) results1 + results1 -> start;
1884              next_result++;
1885              for (i=0;
1886      (!((*cur_result1==255)&&(*next_result==255)))&&(i<RESULTS_SIZE); i++) {
1887                          cur_result1++;
1888                          next_result++;
1889                          }
1890
1891              if ( i == RESULTS_SIZE ) {
1892                          /* Recording file is full, so stop recording */
1893                          printf("\r\nMemory is full...");
1894                          switch_on( MEMORY_FULL );
1895                          while(1) ;
1896                                  /* Infinite loop, ie. programme halts at this point */
1897                          }
1898
1899              }
1900
1901      }
1902
1903
1904      /* This function is called by the setup_datamods function.  According   */
1905      /* to 'no_of_legs', it creates the data file(s) only if not already     */
1906      /* present, returning SUCCESSFUL or UNSUCCESSFUL.                        */
1907
1908      int open_data_file( int no_of_legs )
1909
1910      {
1911
1912      if( no_of_legs == 2 ) {
1913              /* When called, the data file 1 has already been deleted.       */
1914              /* Therefore just check for whether data file 2 is already      */
1915              /* present.                                                      */
1916              if ( (data2=link("Data2.dat"))==NULL ) {
1917                          data2 = datamod("Data2.dat", (DATA_SIZE/2), 0);
1918                          backup(data2, 1);
1919                          data1 = datamod("Data1.dat", (DATA_SIZE/2), 0);
1920                          backup(data1, 1);
1921                          return ( SUCCESSFUL );
1922                          }
1923              }
1924
1925      else {            /* 1 leg being monitored. */
1926              if ( (data1=link("Data1.dat"))==NULL ) {
1927                          data1 = datamod("Data1.dat", DATA_SIZE, 0);
1928                          backup(data1, 1);
1929                          return ( SUCCESSFUL );
1930                          }
1931              }
1932
1933      return ( UNSUCCESSFUL );
1934
1935      }
1936
1937
1938      /* This function is called near the beginning of the main() function.   */
1939      /* It creates an event file if one does not already exist.              */
```

```
1940
1941    int open_event_file()
1942
1943    {
1944
1945    if ( (event=link("Event.dat"))==NULL ) {
1946          event = datamod("Event.dat", EVENT_SIZE, 0);
1947          backup(event, 1);
1948          return( SUCCESSFUL );
1949          }
1950
1951    return( UNSUCCESSFUL );
1952
1953    }
1954
1955
1956    /* This function is called from a number of places, and according to   */
1957    /* 'no_of_legs' creates the results file(s) and sets the results files  */
1958    /* pointers to the start og the file.                                   */
1959
1960    int open_results_file( int no_of_legs )
1961
1962    {
1963    int outcome;
1964
1965    if( no_of_legs == 2 ) {
1966          if ( (results2=link("Results2.dat"))==NULL ) {
1967
1968                /* ie. no of legs being monitored has just been changed */
1969                /* from 1 to 2 (or just initialising at beginning of    */
1970                /* execution. So if Results1.dat is present, it is       */
1971                /* cleared.                                              */
1972
1973                if( !((results1=link("Results1.dat"))==NULL) ) {
1974                      backup(results1, 0);
1975                      unfix("Results1.dat");
1976                      }
1977
1978
1979                /* Next the results files are created, with the file    */
1980                /* pointers being set to the start of the files.         */
1981
1982                results2 = datamod("Results2.dat", (RESULTS_SIZE/2), 0);
1983                backup(results2, 1);
1984                cur_result2 = (unsigned char *) results2 + results2 -> start;
1985
1986                results1 = datamod("Results1.dat", (RESULTS_SIZE/2), 0);
1987                backup(results1, 1);
1988                cur_result1 = (unsigned char *) results1 + results1 -> start;
1989
1990
1991                /* Finally the current date is stored at the start of    */
1992                /* the files.                                            */
1993
1994                stamp_results(2);
1995
1996
1997                return( SUCCESSFUL );
1998
1999                }
2000
2001          }
2002
2003    else {              /* ie. only 1 leg being monitored so only 1 results file */
2004
2005          if ( (results1=link("Results1.dat"))==NULL ) {
2006
2007                /* This means that execution is at the initialisation stage.
2008    */
2009
2010                results1 = datamod("Results1.dat", RESULTS_SIZE, 0);
2011                backup(results1, 1);
```

```
2012              cur_result1 = (unsigned char *) results1 + results1 -> start;
2013
2014              stamp_results(1);
2015
2016              return( SUCCESSFUL );
2017
2018              }
2019
2020          }
2021
2022    return ( UNSUCCESSFUL );
2023
2024    }
2025
2026
2027    /* This function is called by the sleep and writebcd functions, and is  */
2028    /* used when powering down the processor.  It writes 'data' into the     */
2029    /* specified 'reg' of the real-time clock.                               */
2030
2031    void writereg( int reg, int data )
2032
2033    {
2034    char buffer[3];
2035
2036    buffer[0] = reg;
2037    buffer[1] = data;
2038    i2c(buffer, buffer, 2, 0, 0x50);
2039
2040    }
2041
2042
2043    void writebcdreg( int reg, int data )
2044
2045    {
2046
2047    writereg( reg, (data/10)<<4+(data%10) );
2048
2049    }
2050
2051
2052    /* This function is called by the pressure_input function for the        */
2053    /* powering down of the processor in between samples.  Currently only    */
2054    /* the HUNDREDTHS_OF_SECONDS option is used, but the SECONDS case has     */
2055    /* been included to facilitate future possible future extension of the   */
2056    /* programme.  The amount of time to leave the processor in 'sleep' mode  */
2057    /* is stored in the RAM of the real-time clock, before the assembler      */
2058    /* routine pd() is called which saves the interrupts, disables them,      */
2059    /* and then powers down the processor for the specified period of time.   */
2060
2061    void sleep( int time, int function )
2062
2063    {
2064    writereg( 0, 0x0c);
2065
2066    switch( function ) {
2067    case SECONDS:
2068          if (time > 3600) {
2069                  time = time /3600;
2070                  writereg( 8, 0x0c);
2071                  }
2072          else if( time > 60 ) {
2073                  time = time / 60;
2074                  writereg( 8, 0x0b );
2075                  }
2076          else {
2077                  writereg( 8, 0x0a );
2078                  }
2079          writebcdreg( 7, 100-time );
2080          pd();
2081          writereg( 0, 0x08 );
2082          break;
2083
```

```
2084    case HUNDREDTHS_OF_SECS:
2085            writereg( 8, 0x09 );
2086            writebcdreg( 7, 100-time );
2087            pd();
2088            writereg( 0, 0x08 );
2089            break;
2090
2091            }
2092
2093    }
2094
2095
2096    /* This function is currently just called from the main() function.    */
2097    /* It is placed where it should never be executed, so that if it is ever*/
2098    /* entered then a possible hardware fault (or transient fault) has      */
2099    /* occurred.  It was left in as a separate function so that it would be */
2100    /* available for use for general error handling during future extension.*/
2101
2102    void error( void )
2103
2104    {
2105
2106    /* The program execution should never get here */
2107
2108    switch_off( RECORDING );
2109    switch_off( PC_LINK );
2110    switch_off( TRANSMITTING );
2111    switch_on( ERROR );
2112
2113    /* Helpful (?!) error message.... */
2114
2115    printf("\r\nThis is an impossible error (if that helps at all) ....");
2116
2117    while(1) ;       /* Infinite loop, ie. programme halts at this point */
2118
2119    }
2120
2121
2122    /* The following functions are the code and related functions used      */
2123    /* by the shell() programme accessed through the options menu.          */
2124
2125    shell()
2126    {
2127            int stopflag;
2128            int pid;
2129            char Cmd[32];
2130            char Arg[32];
2131            char *ptr;
2132
2133            stopflag = 0;
2134            printf("PSI Systems 'C' Support\n\n\r");
2135            fflush(stdout);
2136            do
2137            {
2138                    do
2139                    {
2140                            fputs("C > ",stdout);
2141                            fflush(stdout);
2142                            readln(0,Buffer,80);
2143                    }
2144                    while( Buffer[0] == 13 );
2145
2146                    Inp_Path = -1;
2147                    Out_Path = -1;
2148                    in = stdin;
2149                    out = stdout;
2150                    coproc = 0;
2151
2152                    if( doargs() )
2153                    {
2154                            ptr = getarg(Buffer,Cmd,0);
2155                            if( !cmpnam( "d",Cmd ) )
```

```
2156                              {
2157                                      ptr = clearwhite( ptr );
2158                                      if( *ptr < 32 )
2159                                              *ptr = 0;
2160                                      debug( ptr );
2161                              }
2162                              else if( !cmpnam( "load",Cmd ) )
2163                              {
2164                                      ptr = getarg(Buffer,Arg,1);
2165                                      load(Arg);
2166                              }
2167                              else if( !cmpnam( "mdir",Cmd ) )
2168                              {
2169                                      mdir();
2170                              }
2171                              else if( !cmpnam("procs",Cmd) )
2172                              {
2173                                      proc();
2174                              }
2175                              else if( !cmpnam("lock",Cmd) )
2176                              {
2177                                      ptr = getarg(Buffer,Arg,1);
2178                                      lock(Arg);
2179                              }
2180                              else if( !cmpnam("unlock",Cmd) )
2181                              {
2182                                      ptr = getarg(Buffer,Arg,1);
2183                                      unlock(Arg);
2184                              }
2185                              else if( !cmpnam("unload",Cmd) )
2186                              {
2187                                      ptr = getarg(Buffer,Arg,1);
2188                                      if( unfix(Arg) )
2189                                      {
2190                                              tsterror("Can't unload module");
2191                                      }
2192                              }
2193                              else if( !cmpnam("quit",Cmd) )
2194                              {
2195                                      stopflag = 1;
2196                              }
2197                              else
2198                              {
2199                                      if( (pid=chain(Cmd,4096,Inp_Path,Out_Path)) == -
2200      1)
2201                                      {
2202                                              tsterror("Can't create new process");
2203                                      }
2204                                      else if( !coproc )
2205                                      {
2206                                              death( pid );
2207                                              wait();
2208                                      }
2209                              }
2210                      }
2211              }
2212          while( stopflag != 1 );
2213      }
2214
2215
2216  /* This routine locates the named module then sets      */
2217  /* the battery backup marker on it's RAM                 */
2218  /*                                                       */
2219
2220  lock( name )
2221  char *name;
2222  {
2223          void *Pointer;
2224          if( (Pointer = link(name)) == NULL )
2225          {
2226                  tsterror("Can't find module");
2227          }
```

```
2228            else
2229            {
2230                    if( backup( Pointer,1 ) )
2231                    {
2232                            tsterror("Can't lock module");
2233                    }
2234            }
2235    }
2236
2237
2238    /* This routine locates the named module then clears     */
2239    /* the battery backup marker on it's RAM.                */
2240
2241    unlock( name )
2242    char *name;
2243    {
2244            void *Pointer;
2245            if( (Pointer = link(name)) == NULL )
2246            {
2247                    tsterror("Can't find module");
2248            }
2249            else
2250            {
2251                    if( backup( Pointer,0 ) )
2252                    {
2253                            tsterror("Can't unlock module");
2254                    }
2255            }
2256    }
2257
2258    load( name )
2259    char *name;
2260    {
2261            int Path;
2262            int Size;
2263            unsigned char *Buffer;
2264
2265            Path = open(name,3);
2266            if( Path == -1 )
2267            {
2268                    tsterror("Can't open file");
2269            }
2270            else
2271            {
2272                    Size = fsize(Path);
2273                    if( Size == -1 )
2274                    {
2275                            tsterror("file size");
2276                            return(0);
2277                    }
2278                    else
2279                    {
2280                            if( (Buffer=sysmem( Size, 0 )) == NULL )
2281                            {
2282                                    tsterror("Can't allocate memory");
2283                            }
2284                            else
2285                            {
2286                                    read(Path,Buffer,Size);
2287                                    if( fixmod( Buffer ) )
2288                                    {
2289                                            tsterror("Can't attach module");
2290                                    }
2291                                    else
2292                                    {
2293                                            printf("loaded at %x\n\r",Buffer);
2294                                            fflush(stdout);
2295                                    }
2296                            }
2297                    }
2298                    close(Path);
2299            }
```

```
2300        }
2301
2302        tsterror( string )
2303        char *string;
2304        {
2305                printf("%s error %d\n\r",string,errno);
2306                fflush(stdout);
2307        }
2308
2309
2310        /* This routine checks along the command line looking for the   */
2311        /* re-direct arrows < and > and for concurrent process flag &    */
2312
2313        int doargs()
2314        {
2315                char *name;
2316                char *ptr;
2317                char arg[32];
2318                int n;
2319                FILE *fp;
2320
2321                n = 1;
2322                while( (ptr=getarg(Buffer,arg,n++)) != NULL )
2323                {
2324                        switch(arg[0])
2325                        {
2326                        case '&' :        coproc = 1;
2327                                        break;
2328                        case '>' :        name = &arg[1];
2329                                        fp = fopen(name,"w");
2330                                        if( fp == NULL )
2331                                        {
2332                                                tsterror("Can't re-direct output");
2333                                                return(0);
2334                                        }
2335                                        setbuf(fp,NULL);
2336                                        Out_Path = _paths[fileno(fp)];
2337                                        out = fp;
2338                                        break;
2339                        case '<' :        name = &arg[1];
2340                                        fp = fopen(name,"r");
2341                                        if( fp == NULL )
2342                                        {
2343                                                tsterror("Can't re-direct input");
2344                                                return(0);
2345                                        }
2346                                        setbuf(fp,NULL);
2347                                        Inp_Path = _paths[fileno(fp)];
2348                                        in = fp;
2349                                        break;
2350                        }
2351                }
2352                return(1);
2353        }
2354
2355
2356        /* This function extracts an argument from a string. The args      */
2357        /* are separated by tabs or spaces and the input line can end with */
2358        /* 0 or 13. The required argument is coppied into the buffer pointed */
2359        /* to by arg and is zero terminated.                               */
2360
2361        char * getarg( string, arg, count )
2362        char *string;
2363        char *arg;
2364        int count;
2365        {
2366                string = clearwhite( string );
2367                while( count )
2368                {
2369
2370                while((*string!=32)&&(*string!=9)&&(*string!=13)&&(*string!=0))
2371                        string++;
```

```
2372
2373                    if( (*string==13) || (*string==0) )
2374                    {
2375                            return(NULL);
2376                    }
2377                    else
2378                    {
2379                            string = clearwhite( string );
2380                            --count;
2381                    }
2382            }
2383            while((*string!=32) && (*string!=9) && (*string!=13) &&
2384    (*string!=0))
2385                    *arg++ = *string++;
2386            *arg = 0;
2387            return(string);
2388    }
2389
2390
2391    char *clearwhite( string )
2392    char *string;
2393    {
2394            while( (*string==9) || (*string==32) )
2395                    string++;
2396            return( string );
2397    }
2398
2399
2400    /* This routine checks that to character arrays are the       */
2401    /* same regardless of the case of the alpha characters in the */
2402    /* two strings.                                               */
2403
2404    cmpnam( s1, s2 )
2405    char * s1;
2406    char * s2;
2407    {
2408            register char c1;
2409            register char c2;
2410            do
2411            {
2412                    c1 = *s1++;
2413                    c2 = *s2++;
2414                    if( (c2>='A') && (c2<='Z') )
2415                            c2 += ('a'-'A');
2416            }
2417            while( (c1 != 0) && (c2 != 0) && (c1 == c2) );
2418            if( (c1 == 0) && (c2 == 0) )
2419                    return(0);
2420            else
2421                    return(-1);
2422    }
2423
2424
2425    mdir()
2426    {
2427            struct md **mdirglob;
2428            struct md *mdir;
2429
2430            mdirglob = (struct md **) 0x80408;
2431            mdir = *mdirglob;
2432            fprintf(out,"\r\n Address |   Size   | Module Name |  Type    |
2433    Memory");
2434            fprintf(out,"\r\n=================================================
2435    =");
2436            while(1)
2437            {
2438                    ShowPage(mdir);
2439                    if( (mdir=mdir->next) == NULL )
2440                            break;
2441            }
2442            fprintf(out,"\r\n");
2443            fflush(out);
```

```
2444    }
2445
2446    ShowPage(mdir)
2447    struct md *mdir;
2448    {
2449            int n;
2450            struct moddef *module;
2451            char *string;
2452            for(n=0; n<30; n++)
2453            {
2454                    if( (module = mdir->modules[n].module) != NULL )
2455                    {
2456                            string = (char *) module + module->name;
2457                            fprintf(out,"\r\n %06x   | ",module);
2458                            fprintf(out,"%6x | ",module->size);
2459                            fprintf(out,"%-13s| ",string);
2460                            fprintf(out,"%-8s|",types[module->type]);
2461                            module->header = 0;
2462                            if( module->header )
2463                            {
2464                                    fprintf(out,"  Rom");
2465                            }
2466                            else
2467                            {
2468                                    module->header = 0x4afc;
2469                                    fprintf(out,"  Ram");
2470                            }
2471                    }
2472            }
2473    }
2474
2475    proc()
2476    {
2477            int n;
2478            struct pd *desc;
2479            pdt *pdtable;
2480            pdt **pdtptr;
2481            struct moddef *mod;
2482            pdtptr = (pdt **)0x80424;
2483            pdtable = *pdtptr;
2484            fprintf(out," PID |  Module Name |  Status   |  Signal | Sleep   |
2485    Death \r\n");
2486            fprintf(out,"====================================================
2487    ===\r\n");
2488            for( n=0; n<64; n++)
2489            {
2490                    if( (desc=(*pdtable)[n]) != NULL )
2491                    {
2492                            if( (mod = desc->module) == 0 )
2493                                    fprintf(out,"%4d | %-13s|",n,"  raw code");
2494                            else
2495                                    fprintf(out,"%4d | %-13s|",n,(char *)mod + mod-
2496    >name);
2497                            if( desc->status == 1 )
2498                                    fprintf(out," Sleeping |");
2499                            else if( desc->status == 2 )
2500                                    fprintf(out," Running  |");
2501                            else
2502                                    fprintf(out," Waiting  |");
2503                            fprintf(out," %7d | %6d |",desc->signal,desc->sleep);
2504                            if( desc->death == 0xffff )
2505                                    fprintf(out," none\r\n");
2506                            else
2507                                    fprintf(out,"%5d\r\n",desc->death);
2508                    }
2509            }
2510
2511            fflush(out);
2512
2513    }
2514
```

Appendix 4 : The PC Analysis Program Listing

```c
1   #include <stdio.h>
2   #include <string.h>
3   #include <graph.h>
4   #include <math.h>
5   #include <stdlib.h>
6   #include <pgchart.h>
7
8   #define ON                1
9   #define OFF               0
10  #define SUCCESSFUL        0
11  #define UNSUCCESSFUL      1
12  #define NO_OF_POINTS      50
13
14  typedef enum {FALSE, TRUE} boolean;
15
16  char * get_outname( char * );
17  char * get_name( char * );
18  void case_convert( char * );
19  void input_file ( char *, char *, char * );
20  char * get_patient_file( char * );
21  char * get_date( char * );
22  int calc_no_of_weeks( int, int, int, int, int, int );
23  float calc_weight_bearing( char *, int );
24
25  FILE *infile, *outfile;
26
27
28  main()
29  {
30  char *temp, *inname, *outname, *name, *temp_outname;
31  char in_name[13];
32  char out_name[13] = "";
33  char pat_name[40] = "";
34  char *char_array_pointer;
35  char pat_file[40] = "";
36  char temp_out_name[13];
37  char temp_name1[40];
38  char temp_name2[40];
39  char temp_name3[40];
40  char temp_name4[40];
41  char patient_notes[2000];
42
43  int hour[24], body_mass=0, i, j, k, input, flag1, flag2, flag3, flag4,
44  flag5, legs_monitored, event_level, n, input_value, outcome;
45  int no_of_weeks, year_end, month_end, day_end, year_start, month_start,
46  day_start, year_new, year_cur, month_new, month_cur, day_new, day_cur;
47  char input_char;
48  double sum, sq_dev;
49  float weight_mean, weight_variance, weight_std_dev, duration_mean,
50  duration_variance, duration_std_dev;
51  int cur_read, cur_write, day_ob, month_ob, year_ob;
52  char hospital_no[20];
53  char leg_broken;
54  char fracture_type[40], fracture_pos[40], fracture_treat[40];
55  float hours[24], weight_bearing[NO_OF_POINTS], week_no[NO_OF_POINTS];
56  float temp_float;
57  chartenv env;
58
59  char far *hour_name[24] =
60  {
61      "00", "01", "02", "03", "04", "05", "06", "07", "08", "09",
62      "10", "11", "12", "13", "14", "15", "16", "17", "18", "19",
63      "20", "21", "22", "23"
64  };
65
66
67  /* Initialisation */
68
```

```
69    /* Not knowing the type and age of PC which might in the future execute */
70    /* this programme, the graphics mode with the lowest common denominator */
71    /* has been chosen (CGA).  The black and white mode was chosen for its  */
72    /* increased resolution over the colour modes.                          */
73
74    _setvideomode(_HRESBW);
75
76    inname = "";
77    outname = "";
78    name = "";
79
80
81    /* This programme uses the following directories for the storage of its */
82    /* files: '\patients', '\data', '\analysis'.  The following code checks */
83    /* for whether they are present on the c: drive, and if not (ie this is */
84    /* the first time that the programme has been executed on this          */
85    /* computer), they are created.                                         */
86
87    /* First print the list of directories from the root into the file      */
88    /* called 'temp.dat'.                                                   */
89
90    system( "dir c:\\ /A:D /B > temp.dat" );
91
92    /* Next search through this file comparing each directory name with     */
93    /* DATA, ANALYSIS, and PATIENTS, recording the matches found.           */
94
95    infile = fopen( "temp.dat", "r" );
96    flag1 = OFF;
97    flag2 = OFF;
98    flag3 = OFF;
99    while ( fscanf( infile, "%s", temp_name1) != EOF ) {
100           if ( strcmp( temp_name1, "DATA" ) == 0 )
101                   flag1=ON;
102           else if ( strcmp( temp_name1, "ANALYSIS") == 0 )
103                   flag2=ON;
104           else if ( strcmp( temp_name1, "PATIENTS" ) == 0 )
105                   flag3=ON;
106           }
107    fclose( infile );
108
109
110    /* Finally for any that a match was not found, create it.        */
111
112    if ( flag1 == OFF )
113           system( "mkdir c:\\data" );
114    if ( flag2 == OFF )
115           system( "mkdir c:\\analysis" );
116    if ( flag3 == OFF )
117           system( "mkdir c:\\patients" );
118
119
120    /* Enter the main part of the programme, which as an infinite loop will */
121    /* never be exited from (except when the programme execution is         */
122    /* terminated).                                                        */
123
124    while (1) {
125           if ( strcmp( outname, "" ) != 0 )
126                   strcpy( temp_out_name, outname );
127
128           else {
129                   if ( strcmp( inname, "" ) == 0 )
130                           strcpy( temp_out_name, "" );
131                   else
132                           strcpy( temp_out_name, get_outname( inname ) );
133
134                   }
135
136           temp_outname = temp_out_name;
137           outname = temp_out_name;
138
139
140           /* The following is the root or main menu. */
```

```
141
142          _clearscreen(_GCLEARSCREEN);
143          _settextposition(1,5);
144          printf("Current data file: %s",inname);
145          _settextposition(2,5);
146          printf("Current analysis file: %s",outname);
147          _settextposition(3,5);
148          printf("Current patient name: %s",pat_name);
149          _settextposition(5,20);
150          printf("1: Change any of above details");
151          _settextposition(7,20);
152          printf("2: Analyse data, storing results in %s",temp_outname);
153          _settextposition(9,20);
154          printf("3: Display analysis");
155          _settextposition(11,20);
156          printf("4: Examine patient's history");
157          _settextposition(13,20);
158          printf("5: Delete a patient's files");
159          _settextposition(15,20);
160          printf("6: Exit");
161
162          _settextposition(20,20);
163          printf("Please input a number between 1 and 6: ");
164          scanf("%d",&input);
165
166
167          /* Selecting option '1' allows the operator to change any of the*/
168          /* above name details via various sub-menus.                    */
169
170          if (input==1) {
171                  flag1=OFF;
172                  while ( flag1==OFF ) {
173                          _clearscreen(_GCLEARSCREEN);
174                          _settextposition(1,5);
175                          printf("Current data file: %s",inname);
176                          _settextposition(2,5);
177                          printf("Current analysis file: %s",outname);
178                          _settextposition(3,5);
179                          printf("Current patient name: %s",pat_name);
180                          _settextposition(5,20);
181                          printf("1: Change data file name");
182                          _settextposition(7,20);
183                          printf("2: Change patient name");
184                          _settextposition(9,20);
185                          printf("3: Return to main menu");
186
187                          _settextposition(20,20);
188                          printf("Please input a number between 1 and 3: ");
189                          scanf("%d",&input);
190
191
192                          /* Having selected option '1', the operator can */
193                          /* access other options to aid in the selecting */
194                          /* of a new data file name.  These are to list  */
195                          /* the data files not yet analysed, and/or list */
196                          /* those already analysed.                      */
197
198                          if (input==1) {
199                                  flag2=OFF;
200                                  while( flag2==OFF ) {
201                                          _clearscreen(_GCLEARSCREEN);
202                                          _settextposition(1,5);
203                                          printf("Current data file: %s",inname);
204                                          _settextposition(2,5);
205                                          printf("Current analysis file:
206  %s",outname);
207                                          _settextposition(3,5);
208                                          printf("Current patient name:
209  %s",pat_name);
210                                          _settextposition(5,20);
211                                          printf("1: Change data file name");
212                                          _settextposition(7,20);
```

```
213                               printf("2: List data files not analysed");
214                               _settextposition(9,20);
215                               printf("3: List analysed data files");
216                               _settextposition(11,20);
217                               printf("4: Return to the previous menu");
218
219                               _settextposition(20,20);
220                               printf("Please input a number between 1
221    and 4: ");
222                               scanf("%d",&input);
223
224
225                               /* When selecting the          */
226                               /* following option, the        */
227                               /* operator inputs the date of  */
228                               /* the monitoring session he    */
229                               /* wishes to access the data of.*/
230                               /* If a patient name has not yet*/
231                               /* been specified the operator  */
232                               /* is also asked to input one.  */
233                               /* From all this information,    */
234                               /* the name of the required data*/
235                               /* file is constructed and the  */
236                               /* accessed.                     */
237
238                               if ( input==1 )
239                                       {
240                                       /* The following function performs
241    most */
242                                       /* of the file name construction.
243    */
244
245                                       input_file( temp_name2, pat_name,
246    pat_file );
247
248
249                                       /* 'temp_name2' is OFF only if the
250    */
251                                       /* inputted patient name was not
252    valid. */
253
254                                       if ( strcmp( temp_name2, "OFF" ) !=
255    0 ) {
256                                               strcpy( temp_name1, "DAT" );
257                                               strcat( temp_name1,
258    temp_name2 );
259
260                                               strcpy( temp_name2,
261    temp_name1 );
262                                               temp_name2[9] = '0';
263                                               temp_name2[10] = '0';
264                                               temp_name2[11] = '0';
265
266
267                                               /* List the unanalysed data
268    file*/
269                                               /* names in 'temp.dat', and
270    then*/
271                                               /* check to see that the data
272    */
273                                               /* file name constructed
274    exists */
275                                               /* among these.
276    */
277
278                                               system( "dir dat*.* /B >
279    temp.dat" );
280
281                                               infile = fopen( "temp.dat",
282    "r" );
283                                               flag3=OFF;
```

```
284                                                      while ( fscanf( infile, "%s",
285    temp_name3 ) != EOF ) {
286                                                          if ( strcmp(
287    temp_name2, temp_name3 ) == 0 )
288                                                                  flag3=ON;
289                                                          }
290                                                      fclose( infile );
291
292                                                      if ( flag3 == OFF ) {
293                                                          /* Check to see whether
294    the constructed */
295                                                          /* data file name
296    exists for the          */
297                                                          /* analysed data files
298    (ie those stored */
299                                                          /* in the '\data'
300    directory.              */
301
302                                                          system( "dir
303    c:\\data\\dat*.* /B > temp.dat" );
304                                                          infile = fopen(
305    "temp.dat", "r" );
306                                                          flag4=OFF;
307                                                          while ( fscanf( infile,
308    "%s", temp_name3 ) != EOF ) {
309                                                              if ( strcmp(
310    temp_name1, temp_name3 ) == 0 )
311                                                                      flag4=ON;
312                                                              }
313                                                          fclose( infile );
314
315                                                          if ( flag4==OFF )
316                                                              printf("\nThe
317    inputted data file '%s' does not exist !! ",temp_name1);
318                                                          else
319                                                              goto
320    The_unmentionable_command;
321                                                          }         /* Although
322    using a 'goto', no danger of stack overloading because */
323                                                          /* the 'if'
324    statement's opening bracket is cancelled by the 'else' */
325                                                      else {          /*
326    statement's closing bracket. */
327
328                                                          /* The file is not
329    analysed yet (so has '.000'  */
330                                                          /* as its suffix).
331    Therefore rename the file    */
332                                                          /* the patient number
333    suffix.                  */
334
335                                                          strcpy( temp_name3,
336    "rename " );
337                                                          strcat( temp_name3,
338    temp_name2 );
339                                                          strcat( temp_name3, " "
340    );
341                                                          strcat( temp_name3,
342    temp_name1 );
343                                                          system( temp_name3 );
344
345                                                          /* The following code
346    is executed for whether   */
347                                                          /* the data file name
348    has or has not been       */
349                                                          /* analysed.
350    */
351
352                    The_unmentionable_command:            strcpy(
353    in_name, temp_name1 );
354                                                          inname = in_name;
```

```
355                                         printf("\nThe inputted
356    data file '%s' is accepted.",inname);
357
358
359                                         /* Now generate the
360    analysis file name, and    */
361                                         /* check for whether an
362    analysis file exists    */
363                                         /* with the same name.
364    */
365
366                                         strcpy( temp_name3,
367    "AN" );
368                                         for ( i=0; i<9; i++ )
369                                                 temp_name3[i+2]
370    = temp_name1[i+3];
371    temp_name3[i+2] = 0;
372
373                                         system( "dir
374    c:\\analysis\\an*.* /B > temp.dat" );
375                                         infile = fopen(
376    "temp.dat", "r" );
377                                         flag3=OFF;
378                                         while( fscanf( infile,
379    "%s", temp_name4 ) != EOF ) {
380                                                 if ( strcmp(
381    temp_name3, temp_name4 ) == 0 )
382                                                         flag3=ON;
383                                                 }
384                                         fclose( infile );
385
386
387                                         /* If not, then print a
388    message to remind the    */
389                                         /* operator to analyse
390    the data file.           */
391
392                                         if ( flag3==OFF ) {
393                                                 printf("\nThe
394    corresponding analysis '%s' file does not exist.",temp_name3);
395                                                 printf("\nThe
396    user must analyse the data file first.");
397                                                 printf("\nTo do
398    so, the data file must be in the current directory.");
399                                                 }
400                                         else {
401                                                 strcpy(
402    out_name, temp_name3 );
403                                                 outname =
404    out_name;
405                                                 printf("\nThere
406    is a corresponding analysis file, '%s'.",outname);
407                                                 }
408
409                                         }
410
411                                 }
412
413                         printf("\n\nPress a key to
414    continue.");
415                         getch();
416
417                 }
418
419
420                         /* List the data files not yet  */
421                         /* analysed by printing the ones*/
422                         /* on the screen from the       */
423                         /* current directory, as they   */
424                         /* haven't been moved to '\data'*/
425                         /* directory yet.               */
426
```

```
427                                        else if (input==2) {
428                                              system( "dir dat*.* /B | more" );
429                                              getch();
430                                              }
431
432
433                                        /* By printing the data files in the
434      '\data' directory, the        */
435                                        /* analysed files are printed.
436      */
437
438                                        else if (input==3) {
439                                              system( "dir c:\\data\\dat*.* /B |
440      more" );
441                                              getch();
442                                              }
443
444                                        else if (input==4)
445                                              flag2=ON;
446
447                                        else {
448                                              printf("\nInput range is from 1 to
449      3. Please try again.");
450                                              getch();
451                                              }
452
453                                  }
454
455                            }
456
457
458                      /* By selecting option '2', the operator can    */
459                      /* change the patient name to another; list the */
460                      /* names of the patients having been monitored; */
461                      /* input details of a new patient for storage.  */
462
463                      else if (input==2) {
464                            flag2 = OFF;
465                            while ( flag2==OFF ) {
466                                  _clearscreen(_GCLEARSCREEN);
467                                  _settextposition(1,5);
468
469                                  if ( strcmp(get_name(pat_file),
470      "unsuccessful") )
471                                        printf("Current patient name:
472      %s",get_name(pat_file));
473                                  else
474                                        printf("Current patient not
475      specified.");
476
477                                  _settextposition(3,20);
478                                  printf("1: Change current patient name");
479                                  _settextposition(5,20);
480                                  printf("2: List patients");
481                                  _settextposition(7,20);
482                                  printf("3: Input details of a new
483      patient");
484                                  _settextposition(9,20);
485                                  printf("4: Return to the previous menu");
486
487                                  _settextposition(20,20);
488                                  printf("Please input a number from 1 to 4:
489      ");
490                                  scanf("%d",&input);
491
492
493                                  /* The first option having      */
494                                  /* been selected, the operator  */
495                                  /* is asked to input the patient*/
496                                  /* name, whose letters are all  */
497                                  /* converted to lower case      */
498                                  /* except the first for each    */
```

```
499                                     /* word which is converted to    */
500                                     /* upper.  The name is then       */
501                                     /* compared to the name stored    */
502                                     /* in every patient file and if */
503                                     /* there is a match then the      */
504                                     /* patient name is accepted.      */
505
506                                     if ( input==1 ) {
507                                         printf("\nWhat is the new patient's
508     name? : ");
509                                         scanf("%s %s", temp_name1,
510     temp_name2);
511                                         case_convert( temp_name1 );
512                                         case_convert( temp_name2 );
513                                         strcat ( temp_name1, " " );
514                                         strcat ( temp_name1, temp_name2 );
515
516                                         strcpy( temp_name3,
517     get_patient_file( temp_name1 ) );
518                                         if ( strcmp( temp_name3,
519     "unsuccessful") ) {
520                                             strcpy( pat_name, temp_name1
521     );
522                                             strcpy( pat_file, temp_name3
523     );
524                                             printf("\nPatient name
525     accepted.");
526                                             strcpy( inname, "" );
527                                             strcpy( outname, "" );
528                                         }
529                                         else
530                                             printf("\nPatient name does
531     not exist !!");
532
533                                     }
534
535
536                                     /* The second option having       */
537                                     /* been selected, the names of    */
538                                     /* all the patients which have    */
539                                     /* been monitored are displayed */
540                                     /* in three columns on the        */
541                                     /* screen.  Each patient file is*/
542                                     /* accessed in turn and the       */
543                                     /* patient name displayed.        */
544
545                                     else if ( input==2 ) {
546                                         system( "dir c:\\patients\\patient.*
547     /B > temp.dat" );
548                                         infile = fopen("temp.dat","r");
549
550                                         i=0;
551                                         while ( fscanf(infile, "%s",
552     temp_name1) != EOF ) {
553                                             i++;
554
555                                             strcpy( temp_name2,
556     "c:\\patients\\" );
557                                             strcat( temp_name2,
558     temp_name1 );
559                                             strcpy( temp_name1,
560     temp_name2 );
561
562                                             outfile =
563     fopen(temp_name1,"r");
564                                             fscanf(outfile, "%s
565     %s",temp_name1,temp_name2);
566                                             fclose(outfile);
567
568                                             strcat( temp_name1, " " );
569                                             strcat( temp_name1,
570     temp_name2 );
```

```
571
572                                        /* The following lines   */
573                                        /* display three columns*/
574                                        /* of names.            */
575
576                                            if( (i%4)==0 )
577                                                 i=1;
578                                            if( ((i%2)==0) || ((i%3)==0)
579   )
580
581           printf("%25s",temp_name1);
582                                            else
583
584           printf("\r\n%25s",temp_name1);
585
586                                                }
587
588                                        fclose(infile);
589
590                                            }
591
592
593                                    /* Selecting the third option   */
594                                    /* allows the operator to input */
595                                    /* the details of a new patient.*/
596                                    /* The operator inputs the      */
597                                    /* patient name which is        */
598                                    /* compared to all the patient  */
599                                    /* names already stored and is  */
600                                    /* only accepted if no match    */
601                                    /* occurs.                      */
602
603                                    else if ( input==3 ) {
604                                            _clearscreen(_GCLEARSCREEN);
605
606                                            printf("What is the new patient's
607   name? : ");
608                                            scanf("%s %s", temp_name1,
609   temp_name2);
610                                            case_convert( temp_name1 );
611                                            case_convert( temp_name2 );
612                                            strcat( temp_name1, " " );
613                                            strcat( temp_name1, temp_name2 );
614                                            system( "dir c:\\patients\\patient.*
615   /B > temp.dat" );
616                                            outfile = fopen("temp.dat","r");
617                                            flag3=OFF; flag4=OFF;
618                                            while( (flag3==OFF) && (flag4==OFF)
619   ) {
620                                                if ( fscanf(outfile,
621   "%s",temp_name2) != EOF ) {
622                                                    strcpy( temp_name3,
623   "c:\\patients\\" );
624                                                    strcat( temp_name3,
625   temp_name2 );
626                                                    strcpy( temp_name2,
627   temp_name3 );
628
629                                                    infile =
630   fopen(temp_name2,"r");
631                                                    fscanf(infile,"%s %s",
632   temp_name3, temp_name4);
633                                                    fclose( infile );
634
635                                                    strcat(temp_name3, "
636   ");
637                                                    strcat(temp_name3,
638   temp_name4);
639                                                    if ( strcmp(temp_name3,
640   temp_name1) == 0 ) {
641                                                        printf("\n%s
642   already exists !!", temp_name1);
```

```
643                                                           flag3 = ON;
644                                                       }
645                                                   }
646                                               else
647                                                   flag4=ON;
648                                           }
649
650                                   /* Execute the following lines only
651  if no match */
652                                   /* has occurred.
653  */
654
655                                   if ( flag3!=ON ) {
656                                           rewind( outfile );
657                                           input=0;
658                                           while( fscanf(outfile, "%s",
659  temp_name2) != EOF ) {
660                                                   for ( i=0; i<3; i++ )
661                                                       temp_name3[i] =
662  temp_name2[i+9];
663
664                                                   temp_name3[i] = 0;
665                                                   i = atoi(temp_name3);
666                                                   if ( i > input )
667                                                           input = i;
668
669                                           }
670
671                                       input += 1;
672
673                                       temp_name3[0] =
674  (input/100)+48;
675                                       temp_name3[1] = ( (input-
676  ((temp_name3[0]-48)*100))/10 ) + 48;
677                                       temp_name3[2] = ( input-
678  ((temp_name3[0]-48)*100)-((temp_name3[1]-48)*10) ) + 48;
679                                       temp_name3[3] = 0;
680
681                                       strcpy( pat_file,
682  "c:\\patients\\patient." );
683                                       strcat( pat_file, temp_name3
684  );
685                                       strcpy( pat_name, temp_name1
686  );
687                                       infile = fopen( pat_file, "w"
688  );
689                                       fprintf( infile, "%s\n",
690  temp_name1 );
691
692                                       flag3=OFF;
693                                       while( flag3==OFF ) {
694                                               printf("\n\nWhat is the
695  patient's date of birth?\n");
696                                               printf("Day: ");
697                                               scanf("%d",&input);
698                                               if ( (input>0) &&
699  (input<32) )
700                                                       flag3=ON;
701                                               else
702                                                       printf("\nThe
703  range is from 1 to 31. Please try again.\n");
704                                       }
705                                       fprintf( infile, "%d ", input
706  );
707
708                                       flag3=OFF;
709                                       while( flag3==OFF ) {
710                                               printf("Month: ");
711                                               scanf("%d",&input);
712                                               if ( (input>0) &&
713  (input<13) )
714                                                       flag3=ON;
```

```
715                                                    else
716                                                            printf("\nThe
717    range is from 1 to 12. Please try again.\n");
718                                                    }
719                                            fprintf( infile, "%d ", input
720    );
721
722                                            flag3=OFF;
723                                            while( flag3==OFF ) {
724                                                    printf("Year (eg.
725    1970): ");
726                                                    scanf("%d",&input);
727                                                    if ( (input>1900) &&
728    (input<2000) )
729                                                            flag3=ON;
730                                                    else
731                                                            printf("\nThe
732    range is from 1900 to 2000. Please try again.\n");
733                                                    }
734                                            fprintf( infile, "%d\n",
735    input );
736
737                                            flag3=OFF;
738                                            while( flag3==OFF ) {
739                                                    printf("Hospital
740    Number: ");
741                                                    scanf("%s",temp_name4);
742                                                    flag3=ON;
743                                                    }
744                                            fprintf( infile, "%s\n",
745    temp_name4 );
746
747                                            flag3=OFF;
748                                            while( flag3==OFF ) {
749                                                    printf("Right or Left
750    leg fractured (R/L): ");
751                                                    fflush( stdin );
752
753            scanf("%c",&input_char);
754                                                    if ( (input_char=='r')
755    || (input_char=='l') )
756                                                            input_char -=
757    32;        /* put into upper case */
758                                                    if ( (input_char=='R')
759    || (input_char=='L') )
760                                                            flag3=ON;
761                                                    else
762                                                            printf("\nInput
763    either R or L. Please try again.\n");
764                                                    }
765                                            fprintf( infile, "%c\n",
766    input_char );
767
768                                            flag3=OFF;
769                                            while( flag3==OFF ) {
770                                                    printf("Fracture type:
771    ");
772                                                    scanf("%s",temp_name1);
773                                                    case_convert(
774    temp_name1 );
775                                                    flag3=ON;
776                                                    }
777                                            fprintf( infile,"%s\n",
778    temp_name1 );
779
780                                            flag3=OFF;
781                                            while( flag3==OFF ) {
782                                                    printf("Position of
783    fracture: ");
784                                                    scanf("%s",temp_name1);
785                                                    case_convert(
786    temp_name1 );
```

```
787                                                     flag3=ON;
788                                             }
789                                     fprintf( infile,"%s\n",
790     temp_name1 );
791
792                                     flag3=OFF;
793                                     while( flag3==OFF ) {
794                                             printf("Fracture
795     treatment: ");
796                                             scanf("%s",temp_name1);
797                                             case_convert(
798     temp_name1 );
799                                             flag3=ON;
800                                     }
801                                     fprintf( infile, "%s\n",
802     temp_name1 );
803
804                                     flag3=OFF;
805                                     while( flag3==OFF ) {
806                                             printf("Patient Body
807     Mass: ");
808                                             scanf("%d", &input);
809                                             if ( (input<0) ||
810     (input>120) )
811                                                     printf("Input
812     range is from 0 to 120 kg..  Please try again.\n");
813                                             else
814                                                     flag3=ON;
815                                     }
816                                     fprintf( infile, "%d\n",
817     input );
818
819                                     printf("\nInput the date when
820     fracture occurred.\n");
821                                     flag3=OFF;
822                                     while( flag3==OFF ) {
823                                             printf("Day: ");
824                                             scanf("%d", &input);
825                                             if (
826     (input<1)||(input>31) )
827                                                     printf("Input
828     range is from day 1 to 31 of the month.  Please try again.\n");
829                                             else
830                                                     flag3=ON;
831                                     }
832                                     fprintf( infile, "%d ", input
833     );
834
835                                     flag3=OFF;
836                                     while( flag3==OFF ) {
837                                             printf("Month (1-12):
838     ");
839                                             scanf("%d", &input );
840                                             if (
841     (input<1)||(input>12) )
842                                                     printf("Input
843     range is from month 1 to 12 of the year.  Please try again.\n");
844                                             else
845                                                     flag3=ON;
846                                     }
847                                     fprintf( infile, "%d ", input
848     );
849
850                                     flag3=OFF;
851                                     while( flag3==OFF ) {
852                                             printf("Year (eg.
853     1993): ");
854                                             scanf("%d", &input);
855                                             if (
856     (input<1992)||(input>1994) )
857                                                     printf("Input
858     range is from 1992 to 1994.  Please try again.\n");
```

```
859                                                        else
860                                                             flag3=ON;
861                                                        }
862                                                   fprintf( infile, "%d\n",
863    input );
864
865                                                   }
866
867                                         fclose(infile);
868                                         fclose(outfile);
869                                         }
870
871
872                              /* Selecting the fourth option returns the
873    operator to  */
874                              /* the previous menu.
875    */
876
877                              else if ( input==4 )
878                                   flag2 = ON;
879
880                              else
881                                   printf("\nInput range is from 1 to
882    4. Please try again.");
883
884
885                              if( flag2==OFF ) {
886                                   printf("\n\nPress a key to
887    continue.");
888                                   getch();
889                                   }
890
891                              }
892                         }
893
894                    /* Selecting the third option returns the operator to
895    */
896                    /* the root or main menu.
897    */
898
899                    else if (input==3)
900                         flag1=ON;
901
902
903                    else {
904                         printf("\nInput range is from 1 to 3. Please try
905    again.");
906                         getch();
907                         }
908
909               }
910          }
911
912
913    /* Selecting this option analyses the currently specified data  */
914    /* file.  This occurs only if a data file and patient name are   */
915    /* specified.  An analysis file is created for the storage of    */
916    /* the analysis results.  The operaor also has the opportunity   */
917    /* input notes of the monitoring session which are also stored   */
918    /* in this file.                                                 */
919
920    else if (input==2) {
921          strcpy( temp_name1, "c:\\data\\" );
922          strcat( temp_name1, inname );
923          if( ((infile=fopen(inname,"r"))==NULL) &&
924    ((infile=fopen(temp_name1,"r"))==NULL) ) {
925               printf("\nCurrent input file does not exist !!");
926               getch();
927               }
928          else if ( strcmp( pat_name, "" ) == 0 ) {
929               printf("\nNo patient name specified !!");
930               getch();
```

```
931                                }
932                   else {
933                        _clearscreen(_GCLEARSCREEN);
934                        printf("Analysing data file and writing results to
935     %s\n",outname);
936                        for ( i=0; i<24; i++ )
937                             hour[i]=0;
938                        fscanf(infile,"%d",&legs_monitored);
939                        fscanf(infile,"%d",&event_level);
940                        outcome = 0;
941                        input_value=0;
942                        sum = 0;
943                        n = 0;
944                        fscanf(infile,"%d",&input_value); /* This is outside so
945     can check if any data events are in file (unlikely but possible) */
946                        while ( input_value != 999 ) {
947                             hour[input_value] += 1;
948                             for ( i=0; i<3; i++ )
949                                  outcome =
950     fscanf(infile,"%d",&input_value);
951                             sum += input_value;
952                             n++;
953                             for ( i=0; i<3; i++ )
954                                  outcome =
955     fscanf(infile,"%d",&input_value);
956
957                             fscanf(infile,"%d",&input_value);
958                             }
959
960                        strcpy( temp_name1, "c:\\analysis\\" );
961                        outname = out_name;
962                        outname = get_outname( inname );
963                        strcat( temp_name1, outname );
964                        outfile = fopen(temp_name1,"w");
965
966                        fprintf(outfile,"%s\n",pat_name);
967                        fprintf(outfile,"%d\n",legs_monitored);
968                        fprintf(outfile,"%d\n",event_level);
969
970                        for ( i=0; i<24; i++ )
971                             fprintf(outfile,"%d ",hour[i]);
972                        fprintf(outfile,"\n");
973
974                        weight_mean = sum/n;
975                        fprintf(outfile,"%f\n",weight_mean);
976
977                        rewind(infile);
978                        input_value = 0;
979                        sq_dev = 0;
980                        duration_mean = 0.0;
981                        fscanf(infile,"%d",&input_value);
982                        fscanf(infile,"%d",&input_value);
983
984                        fscanf(infile,"%d",&input_value);
985                        while ( input_value != 999 ) {
986                             for ( i=0; i<3; i++ )
987                                  fscanf(infile,"%d",&input_value);
988                             sq_dev += (input_value-weight_mean)*(input_value-
989     weight_mean);
990
991                             fscanf(infile,"%d",&input_value);
992                             duration_mean += input_value*256;
993                             fscanf(infile,"%d",&input_value);
994                             duration_mean += input_value;
995
996                             fscanf(infile,"%d",&input_value);
997                             fscanf(infile,"%d",&input_value);
998                             }
999
1000                       weight_variance = sq_dev/n;
1001                       weight_std_dev = sqrt(weight_variance);
1002
```

```
1003                    fprintf(outfile,"%f\n",weight_variance);
1004                    fprintf(outfile,"%f\n",weight_std_dev);
1005
1006                    duration_variance = 0.0;
1007                    duration_mean /= n;         /* mean hour for all events
1008    */
1009
1010                    rewind(infile);
1011                    fscanf(infile,"%d",&input_value);
1012                    fscanf(infile,"%d",&input_value);
1013
1014                    fscanf(infile,"%d",&input_value);
1015                    while ( input_value != 999 ) {
1016                            for ( i=0; i<4; i++ )
1017                                    fscanf(infile,"%d",&j);
1018
1019                            j *= 256;
1020                            fscanf(infile,"%d",&input_value);
1021                            j += input_value;
1022                            duration_variance += (j-duration_mean)*(j-
1023    duration_mean);
1024
1025                            fscanf(infile,"%d",&input_value);
1026                            fscanf(infile,"%d",&input_value);
1027                            }
1028
1029                    duration_variance /= n;
1030
1031                    duration_std_dev = sqrt( duration_variance );
1032
1033                    fprintf(outfile,"%f\n",duration_mean);
1034                    fprintf(outfile,"%f\n",duration_variance);
1035                    fprintf(outfile,"%f\n",duration_std_dev);
1036
1037                    flag1=OFF;
1038                    while ( flag1==OFF ) {
1039                            printf("\nInput the next appointment date in
1040    weeks from today (or 0 if discharged): ");
1041                            scanf("%d",&input);
1042                            if ( (input<0) || (input>52) )
1043                                    printf("\nThe input range is from between
1044    0 and 52 weeks.  Please try again.");
1045                            else
1046                                    flag1=ON;
1047                            }
1048                    fprintf(outfile,"%d\n",input);
1049
1050
1051                    /* The patient notes are stored in the patient_notes
1052    */
1053                    /* array before being written to the file.
1054    */
1055
1056                    printf("\nDo you wish to record some notes in the
1057    analysis file (Y/N): ");
1058                    fflush(stdin);
1059                    scanf("%c",&input_char);
1060                    if ( (input_char=='Y') || (input_char=='y') ) {
1061                            printf("\nType in the details, then press RETURN
1062    to insert them into the analysis file.\n");
1063                            fflush(stdin);
1064                            i=0;
1065                            input_char = getch();
1066                            while( input_char != 13 ) {      /* 13 signifies a
1067    carriage return */
1068                                    putch(input_char);
1069                                    patient_notes[i++] = input_char;
1070                                    input_char = getch();
1071                                    }
1072                            patient_notes[i] = 0;
1073
1074                    /* Have to edit the 'patient_notes' array for    */
```

```
1075                    /* deletes.  A delete is indicated by ASCII 8.  */
1076                    /* Therefore use two indices into the           */
1077                    /* patient_notes array; one for the current read*/
1078                    /* position, the other for the write position.  */
1079                    /* For each iteration through the loop the read  */
1080                    /* position character is written to the write    */
1081                    /* position character, there being no change in  */
1082                    /* the patient_notes array when both the read    */
1083                    /* and write indices point to the same place.    */
1084                    /* When a delete is encountered, the read index  */
1085                    /* is incremented, and the write index is        */
1086                    /* decremeneted so that the previous character   */
1087                    /* will be overwritten so deleteing it.          */
1088
1089                        cur_read = 0;
1090                        cur_write = 0;
1091                        for ( cur_read=0; patient_notes[cur_read]!=0;
1092    cur_read++ ) {
1093                            while( patient_notes[cur_read]==8 ) {
1094                                    cur_read++;
1095                                    if ( cur_write>0 )
1096                                            cur_write--;
1097                                    }
1098
1099                            patient_notes[cur_write] =
1100    patient_notes[cur_read];
1101                            cur_write++;
1102
1103                            }
1104
1105                        patient_notes[cur_write] = 0;
1106                        fprintf(outfile,"%s\n",patient_notes);
1107
1108                        }
1109
1110                    /* Finally the analysed data file is moved from the
1111    */
1112                    /* current directory to the '/data' directory.
1113    */
1114
1115                    fclose(infile);
1116                    strcpy( temp_name1, "copy " ); /* moving data file into
1117    DATA directory */
1118                    strcat( temp_name1, inname );
1119                    strcat( temp_name1, " c:\\data" );
1120                    system( temp_name1 );
1121                    strcpy( temp_name1, "del " );
1122                    strcat( temp_name1, inname );
1123                    system( temp_name1 );
1124
1125                    fclose(outfile);
1126                    fflush(stdin);
1127                    }
1128
1129                fclose(infile);
1130
1131                }
1132
1133
1134        /* Selecting option 3 allows the operator to view the analyses  */
1135        /* of the currently selected analysis file.                     */
1136
1137        else if (input==3) {
1138            flag2=OFF;
1139            while( flag2==OFF ) {
1140                _clearscreen(_GCLEARSCREEN);
1141                _settextposition(1,5);
1142                printf("Current data file: %s",inname);
1143                _settextposition(2,5);
1144                printf("Current analysis file: %s",outname);
1145                _settextposition(3,5);
1146                printf("Current patient name: %s",pat_name);
```

```
1147                        _settextposition(5,20);
1148                        printf("1: Display analyses for '%s' file",outname);
1149                        _settextposition(7,20);
1150                        printf("2: Return to the previous menu");
1151
1152                        _settextposition(20,20);
1153                        printf("Please input a number from 1 to 2: ");
1154                        scanf("%d",&input);
1155
1156
1157                   if( input==1 ) {
1158                        strcpy( temp_name1, "c:\\analysis\\" );
1159                        strcat( temp_name1, outname );
1160                        if ( (outfile = fopen(temp_name1,"r"))==NULL ) {
1161                             printf("\nCurrent output file does not
1162    exist !!");
1163                             getch();
1164                             }
1165                   else {
1166                             fscanf(outfile,"%s
1167    %s",pat_name,temp_name1);
1168                             strcat( pat_name, " " );
1169                             strcat( pat_name, temp_name1 );
1170
1171                             strcpy( temp_name1, get_patient_file(
1172    pat_name ) );
1173                             strcpy( pat_file, temp_name1 );
1174
1175                             infile = fopen( pat_file, "r" );
1176                             fscanf( infile, "%s", temp_name1 );
1177                             fscanf( infile, "%s", temp_name1 );
1178                             fscanf( infile, "%d %d %d", &day_ob,
1179    &month_ob, &year_ob );
1180                             fscanf( infile, "%s", hospital_no );
1181                             fscanf( infile, "%s", temp_name1 );
1182                             leg_broken = temp_name1[0];
1183                             fscanf( infile, "%s", fracture_type );
1184                             fscanf( infile, "%s", fracture_pos );
1185                             fscanf( infile, "%s", fracture_treat );
1186                             fscanf( infile, "%d", &body_mass );
1187                             fclose( infile );
1188
1189                             strcpy( out_name, get_outname( inname ) );
1190
1191                             temp_name4[0] = out_name[2];
1192                             temp_name4[1] = out_name[3];
1193                             temp_name4[2] = '/';
1194                             temp_name4[3] = out_name[4];
1195                             temp_name4[4] = out_name[5];
1196                             temp_name4[5] = '/';
1197                             temp_name4[6] = '9';
1198                             temp_name4[7] = out_name[6];
1199                             temp_name4[8] = 0;
1200
1201                             fscanf(outfile,"%d",&legs_monitored);
1202                             fscanf(outfile,"%d",&event_level);
1203                             for ( i=0; i<24; i++ )
1204                                  fscanf(outfile,"%d",&hour[i]);
1205                             fscanf(outfile,"%f",&weight_mean);
1206                             fscanf(outfile,"%f",&weight_variance);
1207                             fscanf(outfile,"%f",&weight_std_dev);
1208                             fscanf(outfile,"%f",&duration_mean);
1209                             fscanf(outfile,"%f",&duration_variance);
1210                             fscanf(outfile,"%f",&duration_std_dev);
1211                             n = 0;
1212                             for ( i=0; i<24; i++ )
1213                                  n += hour[i];
1214
1215
1216                             _clearscreen(_GCLEARSCREEN);
1217
1218                             _settextposition(1,1);
```

```
1219                                      printf("Patient name: %s",pat_name);
1220
1221                                      _settextposition(1,60);
1222                                      printf("Date: %s",temp_name4);
1223
1224                                      _settextposition(2,25);
1225                                      printf("Total No. of events = %d",n);
1226
1227                                      _settextposition(2,52);
1228                                      printf("Body Mass = %d",body_mass);
1229
1230                                      _settextposition(3,50);
1231                                      printf("Weight Mean = %.3f",weight_mean);
1232
1233                                      _settextposition(4,50);
1234                                      printf("Weight Variance =
1235     %.3f",weight_variance);
1236
1237                                      _settextposition(5,50);
1238                                      printf("Weight Std. Dev. =
1239     %.3f",weight_std_dev);
1240
1241                                      _settextposition(7,40);
1242                                      printf("Weight Bearing = %.0f%% of Body
1243     Mass",(weight_mean/body_mass*100) );
1244
1245                                      _settextposition(3,25);
1246                                      printf("Time Mean = %.2f",duration_mean);
1247
1248                                      _settextposition(4,25);
1249                                      printf("Time Variance =
1250     %.2f",duration_variance);
1251
1252                                      _settextposition(5,25);
1253                                      printf("Time Std. Dev. =
1254     %.2f",duration_std_dev);
1255
1256                                      _settextposition(2,1);
1257                                      printf("D.O.B.:
1258     %d/%2d/%d",day_ob,month_ob,(year_ob-1900));
1259                                      if( (month_ob/10) == 0 ) {
1260                                          if ( (day_ob/10)==0 )
1261                                              _settextposition(2,(1+10));
1262                                          else
1263                                              _settextposition(2,(1+11));
1264
1265                                          printf("0");
1266                                      }
1267
1268                                      _settextposition(3,1);
1269                                      printf("Hospital No.: %s", hospital_no );
1270
1271                                      _settextposition(4,1);
1272                                      strcpy( temp_name1, "Leg Fractured: " );
1273                                      if ( leg_broken=='R' )
1274                                          strcat( temp_name1, "Right" );
1275                                      else
1276                                          strcat( temp_name1, "Left" );
1277                                      printf("%s",temp_name1);
1278
1279                                      _settextposition(5,1);
1280                                      printf("Fracture Type: %s",fracture_type);
1281
1282                                      _settextposition(6,1);
1283                                      printf("Position of Fracture: %s",
1284     fracture_pos);
1285
1286                                      _settextposition(7,1);
1287                                      printf("Fracture Treatment: %s",
1288     fracture_treat);
1289
1290                                      for ( i=0; i<24; i++ )
```

```
1291                                                    hours[i] = (float) hour[i];
1292
1293                                            _pg_initchart();
1294                                            _pg_defaultchart( &env, _PG_COLUMNCHART,
1295     _PG_PLAINBARS );
1296
1297                                            strcpy( temp_name3, "Events Throughout Day
1298     (event level = " );
1299                                            temp_name2[0] = ((event_level/10)+48);
1300                                            temp_name2[1] = 0;
1301                                            strcat( temp_name3, temp_name2 );
1302                                            temp_name2[0] = (event_level-
1303     ((event_level/10)*10)+48);
1304                                            temp_name2[1] = 0;
1305                                            strcat( temp_name3, temp_name2 );
1306                                            strcat( temp_name3, " kg. )" );
1307                                            strcpy( env.maintitle.title, temp_name3 );
1308                                            env.maintitle.justify = _PG_RIGHT;
1309                                            strcpy( env.yaxis.axistitle.title, "No of
1310     Events" );
1311                                            strcpy( env.xaxis.axistitle.title, "Hour
1312     of Day" );
1313                                            env.chartwindow.border = TRUE;
1314                                            env.chartwindow.x1 = 0;
1315                                            env.chartwindow.y1 = 60;
1316                                            env.chartwindow.x2 = 639;
1317                                            env.chartwindow.y2 = 199;
1318                                            _pg_chart( &env, hour_name, hours, 24 );
1319
1320                                            getch();
1321                                            fclose(outfile);
1322
1323                                            }
1324
1325                                    }
1326
1327
1328                        else if( input==2 )
1329                                flag2=ON;
1330
1331                        else
1332                                printf("\nInput range is from 1 to 2.  Please try
1333     again.");
1334
1335
1336                        if( flag2==OFF ) {
1337                                printf("\n\nPress a key to continue.");
1338                                getch();
1339                                }
1340
1341                    }
1342
1343
1344                }
1345
1346
1347        /* Selecting option 4 enables the operator to view the         */
1348        /* patient's history through another sub-menu.  If a patient    */
1349        /* name is not specified when selecting this option, then the   */
1350        /* operator has the option to input one which is then validated,*/
1351        /* or return to the root menu by typing 'go back'.              */
1352
1353        else if (input==4) {
1354                _clearscreen(_GCLEARSCREEN);
1355                _settextposition(1,5);
1356
1357                if ( strcmp( pat_name, "" ) == 0 ) {
1358                        printf("\nNo patient name has been specified.");
1359                        flag1=OFF;
1360                        while ( flag1==OFF ) {
1361                                printf("\nInput the patient's name, or enter 'go
1362     back' to return to the previous\nmenu: ");
```

```
1363                              scanf("%s %s", temp_name1, temp_name2);
1364                              case_convert( temp_name1 );
1365                              case_convert( temp_name2 );
1366                              strcat( temp_name1, " " );
1367                              strcat( temp_name1, temp_name2 );
1368
1369                              if( strcmp( temp_name1, "Go Back" ) == 0 )
1370                                      break;
1371
1372                              system( "dir c:\\patients\\patient.* /B >
1373     temp.dat" );
1374                              outfile = fopen( "temp.dat", "r" );
1375                              flag2=OFF;
1376                              flag3=OFF;
1377                              while( (flag2==OFF) && (flag3==OFF) ) {
1378                                      if ( fscanf( outfile, "%s", temp_name2 )
1379     != EOF ) {
1380                                              strcpy( temp_name3, "c:\\patients\\"
1381     );
1382                                              strcat( temp_name3, temp_name2 );
1383                                              strcpy( temp_name2, temp_name3 );
1384                                              infile = fopen( temp_name2, "r" );
1385                                              fscanf( infile, "%s %s", temp_name3,
1386     temp_name4 );
1387                                              fclose( infile );
1388                                              strcat( temp_name3, " " );
1389                                              strcat( temp_name3, temp_name4 );
1390                                              if ( strcmp( temp_name3, temp_name1
1391     ) == 0 ) {
1392                                                      strcpy( pat_file, temp_name2
1393     );
1394                                                      strcpy( pat_name, temp_name3
1395     );
1396                                                      flag2=ON;
1397                                                      flag1=ON;
1398                                                      }
1399                                              }
1400                                      else {
1401                                              flag3=ON;
1402                                              printf("\n\nThe inputted patient
1403     name '%s' has not been found.",temp_name1);
1404                                              }
1405
1406                                      }
1407
1408                              fclose( outfile );
1409
1410                              }
1411
1412                      }
1413
1414              if ( strcmp( pat_name, "" ) != 0 ) {
1415                      flag1=OFF;
1416                      while( flag1==OFF ) {
1417                              _clearscreen( _GCLEARSCREEN );
1418                              _settextposition(1,5);
1419                              printf("Current patient name: %s",pat_name);
1420                              _settextposition(4,10);
1421                              printf("For the above patient:");
1422                              _settextposition(6,14);
1423                              printf("1: List the dates of the recorded
1424     monitoring sessions.");
1425                              _settextposition(8,14);
1426                              printf("2: Examine the notes from the monitoring
1427     sessions.");
1428                              _settextposition(10,14);
1429                              printf("3: Display a graph of patient's weight-
1430     bearing progress up to date.");
1431                              _settextposition(12,14);
1432                              printf("4: Return to the previous menu.");
1433
1434                              _settextposition(20,14);
```

```
1435                                    printf("Please input a number between 1 and 4:
1436      ");
1437                                    scanf("%d",&input);
1438
1439
1440                                    /* Selecting option 1 lists the dates     */
1441                                    /* of the monitoring session already      */
1442                                    /* recorded for this patient.  All the    */
1443                                    /* analysis file names with this patient*/
1444                                    /* number as their suffix are written to*/
1445                                    /* 'temp.dat' file and then iteratively */
1446                                    /* read and the date extracted from the */
1447                                    /* file name and then printed on the      */
1448                                    /* screen.                                */
1449
1450                                    if ( input==1 ) {
1451                                            for( i=0; pat_file[i]!=0; i++ ) ;   /*
1452      Getting the patient number by reading the patient filename */
1453                                            for( j=3; j>-1; j-- ) {
1454                                                    temp_name2[j] = pat_file[i];
1455                                                    i--;
1456                                                    }
1457
1458                                            strcpy( temp_name1, "dir
1459      c:\\analysis\\an*." );
1460                                            strcat( temp_name1, temp_name2 );
1461                                            strcat( temp_name1, " /B > temp.dat" );
1462
1463                                            system( temp_name1 );
1464                                            outfile = fopen( "temp.dat", "r" );
1465                                            while( fscanf( outfile, "%s", temp_name1 )
1466      != EOF ) {
1467                                                    strcpy ( temp_name2, get_date(
1468      temp_name1 ) );
1469                                                    printf("\n%s",temp_name2);
1470                                                    }
1471
1472                                            }
1473
1474
1475                                    /* Selecting option 2 displays the        */
1476                                    /* notes taken after each monitoring      */
1477                                    /* session of the patient.  The same      */
1478                                    /* code as for option 1 is used, but      */
1479                                    /* there is the addition that each        */
1480                                    /* analysis file is accessed and the      */
1481                                    /* notes read in character by character */
1482                                    /* into the patient_notes char array.    */
1483                                    /* When the character 32 is encountered */
1484                                    /* (which signifies a carriage return)    */
1485                                    /* then the patient_notes array is        */
1486                                    /* displayed on the screen as string.    */
1487
1488                                    else if ( input==2 ) {
1489                                            for( i=0; pat_file[i]!=0; i++ ) ;   /*
1490      Getting the patient number by reading the patient filename */
1491                                            for( j=3; j>-1; j-- ) {
1492                                                    temp_name2[j] = pat_file[i];
1493                                                    i--;
1494                                                    }
1495
1496                                            strcpy( temp_name1, "dir
1497      c:\\analysis\\an*." );
1498                                            strcat( temp_name1, temp_name2 );
1499                                            strcat( temp_name1, " /B > temp.dat" );
1500
1501                                            system( temp_name1 );
1502                                            outfile = fopen( "temp.dat", "r" );
1503                                            _clearscreen(_GCLEARSCREEN);
1504
1505                                            while( fscanf( outfile, "%s", temp_name1 )
1506      != EOF ) {
```

```
1507                                                strcpy ( temp_name2, get_date(
1508    temp_name1 ) );
1509                                                printf("\n\n%s",temp_name2);
1510
1511                                                strcpy( temp_name3, "c:\\analysis\\"
1512    );
1513                                                strcat( temp_name3, temp_name1 );
1514
1515                                                infile = fopen( temp_name3, "r" );
1516
1517                                                fscanf( infile,"%s %s", temp_name2,
1518    temp_name3 );
1519
1520                                                for( i=0; i<26; i++ )
1521                                                        fscanf( infile, "%d", &j );
1522                                                for( i=0; i<6; i++ )
1523                                                        fscanf( infile, "%f",
1524    &weight_mean );
1525                                                fscanf( infile, "%d", &j );
1526
1527                                                strcpy( patient_notes, "" );
1528
1529                                                fscanf( infile, "%s",
1530    patient_notes);
1531                                                for( i=0; patient_notes[i]!=0; i++ )
1532    ; /* This is to find out starting position for next word (stored in
1533    variable i) */
1534                                                fscanf( infile, "%c", &input_char );
1535                                                while( input_char != 0 ) {
1536                                                        patient_notes[i] =
1537    input_char;
1538                                                        fscanf( infile, "%c",
1539    &input_char );
1540                                                        i++;
1541
1542                                                        if( input_char < 32 )
1543                                                                input_char=0;
1544
1545                                                }
1546                                                patient_notes[i] = 0;
1547
1548                                                printf( "\n%s",patient_notes);
1549
1550                                                fclose( infile );
1551
1552                                                getch();
1553
1554                                                }
1555
-1556                                        }
1557
1558
1559                                    /* By selecting option 3 a graph of      */
1560                                    /* weight-bearing over time               */
1561                                    /* post-fracture is displyed.  Each       */
1562                                    /* analysis file for the patient is       */
1563                                    /* again accessed and the %age            */
1564                                    /* weight-bearing value calculated using*/
1565                                    /* the calc_weight-bearing function.      */
1566                                    /* The week no. and corresponding %age    */
1567                                    /* weight-bearing value are stored in     */
1568                                    /* week_no and weight-bearing arrays.     */
1569                                    /* These are used to print the graph.     */
1570
1571                                    else if( input==3 ) {
1572                                            _clearscreen(_GCLEARSCREEN);
1573
1574                                            strcpy( temp_name3, get_patient_file(
1575    pat_name ) );
1576                                            strcpy( pat_file, temp_name3 );
1577
1578                                            infile = fopen( pat_file, "r" );
```

```
1579                                      fscanf( infile, "%s", temp_name2 );
1580                                      fscanf( infile, "%s", temp_name2 );
1581                                      fscanf( infile, "%d %d %d", &day_ob,
1582      &month_ob, &year_ob );

1583                                      fscanf( infile, "%s", hospital_no );
1584                                      fscanf( infile, "%s", temp_name2 );
1585                                      leg_broken = temp_name2[0];
1586                                      fscanf( infile, "%s", fracture_type );
1587                                      fscanf( infile, "%s", fracture_pos );
1588                                      fscanf( infile, "%s", fracture_treat );
1589                                      fscanf( infile, "%d", &body_mass );
1590                                      fclose( infile );
1591
1592                                      infile = fopen( pat_file, "r" );
1593                                      for( i=0; i<12; i++ )
1594                                          fscanf( infile, "%s",temp_name2 );
1595                                      day_start = atoi(temp_name2);
1596                                      fscanf( infile, "%s %s",temp_name2,
1597      temp_name4 );
1598                                      fclose(infile);
1599                                      month_start = atoi(temp_name2);
1600                                      year_start = atoi(temp_name4);
1601                                      year_start -= 1990;
1602
1603                                      outfile = fopen( pat_file,"r" );
1604                                      for( j=0; j<11; j++ )
1605                                          fscanf(outfile, "%s", temp_name1);
1606                                      body_mass = atoi( temp_name1 );
1607                                      fclose( outfile );
1608
1609                                      for( i=0; pat_file[i]!=0; i++ )  ;
1610                                      temp_name1[2] = pat_file[--i];
1611                                      temp_name1[1] = pat_file[--i];
1612                                      temp_name1[0] = pat_file[--i];
1613                                      temp_name1[3] = 0;
1614
1615                                      strcpy( temp_name2, "dir
1616      c:\\analysis\\an*." );

1617                                      strcat( temp_name2, temp_name1 );
1618                                      strcat( temp_name2, " /B > temp.dat" );
1619                                      system( temp_name2 );
1620                                      infile = fopen( "temp.dat","r" );
1621                                      i=1;
1622                                      week_no[0] = 0.0;
1623                                      weight_bearing[0] = 0.0;
1624                                      while( fscanf(infile,"%s",temp_name2) !=
1625      EOF ) {
1626                                              year_cur = temp_name2[6]-48;
1627                                              month_cur = ((temp_name2[4]-
1628      48)*10)+(temp_name2[5]-48);
1629                                              day_cur = ((temp_name2[2]-
1630      48)*10)+(temp_name2[3]-48);
1631                                              no_of_weeks =
1632      calc_no_of_weeks(year_start,month_start,day_start,year_cur,month_cur,day_cu
1633      r);
1634
1635                                              week_no[i] = (float) no_of_weeks;
1636                                              weight_bearing[i] =
1637      calc_weight_bearing(temp_name2,body_mass);
1638                                              i++;
1639                                              }
1640
1641                                      fclose( infile );
1642                                      j = i;
1643
1644                                      for( ; i<NO_OF_POINTS; i++ ) {
1645                                              week_no[i] = 99999.9;
1646                                              weight_bearing[i] = 99999.9;
1647                                              }
1648
1649                                      flag3=OFF;
1650                                      while( flag3==OFF ) {
```

```
1651                                                    flag3=ON;
1652                                                    for( i=0; i<49; i++ )
1653                                                            if( week_no[i] > week_no[i+1]
1654    ) {
1655                                                                    temp_float =
1656    week_no[i];
1657                                                                    week_no[i] =
1658    week_no[i+1];
1659                                                                    week_no[i+1] =
1660    temp_float;
1661
1662                                                                    temp_float =
1663    weight_bearing[i];
1664                                                                    weight_bearing[i] =
1665    weight_bearing[i+1];
1666                                                                    weight_bearing[i+1] =
1667    temp_float;
1668
1669                                                                    flag3=OFF;
1670                                                                    }
1671                                                            }
1672
1673
1674                                    _pg_initchart();
1675                                    _pg_defaultchart( &env, _PG_SCATTERCHART,
1676    _PG_POINTANDLINE );
1677
1678                                    strcpy( temp_name1, "Patient name: " );
1679                                    strcat( temp_name1, pat_name );
1680                                    strcpy( env.maintitle.title, temp_name1 );
1681                                    env.maintitle.justify = _PG_RIGHT;
1682
1683                                    strcpy( env.subtitle.title, "Weight-
1684    bearing as a Percentage of Body Weight" );
1685                                    env.subtitle.justify = _PG_RIGHT;
1686
1687                                    strcpy( env.yaxis.axistitle.title, "%age
1688    Body Weight" );
1689                                    strcpy( env.xaxis.axistitle.title, "Weeks
1690    from Fracture" );
1691
1692                                    env.chartwindow.border = TRUE;
1693
1694                                    _pg_chartscatter( &env, week_no,
1695    weight_bearing, j );
1696
1697                                    getch();
1698
1699                                    }
1700
1701
1702                                    /* Selecting option 4 returns the operator to
1703    */
1704                                    /* the root menu.
1705    */
1706
1707                                    else if ( input==4 )
1708                                            flag1=ON;
1709
1710
1711                                    else
1712                                            printf("\nThe range is from 1 to 4.
1713    Please try again.");
1714
1715
1716                                    if( flag1==OFF ) {
1717                                            printf("\n\nPress a key to continue.");
1718                                            getch();
1719                                            }
1720
1721                            }
1722
```

```
1723
1724                               }
1725
1726                       }
1727
1728
1729          /* Selecting option 5 enters the operator into a sub-menu        */
1730          /* allowing him to delete a patient's records from the disk (the*/
1731          /* patient, data and analysis files).                           */
1732
1733          else if (input==5) {
1734                  flag1=OFF;
1735                  while( flag1==OFF ) {
1736                          _clearscreen(_GCLEARSCREEN);
1737                          _settextposition(1,5);
1738                          printf("Current patient name: %s",pat_name);
1739                          _settextposition(4,9);
1740                          printf("1: Delete current patient's records and tidy
1741   other files accordingly.");
1742                          _settextposition(6,9);
1743                          printf("2: List patients on record (number of
1744   monitoring sessions in brackets).");
1745                          _settextposition(8,9);
1746                          printf("3: Change current patient name.");
1747                          _settextposition(10,9);
1748                          printf("4: Return to the previous menu.");
1749
1750                          _settextposition(20,14);
1751                          printf("Please input a number between 1 and 4: ");
1752
1753                          scanf("%d",&input);
1754
1755
1756                          /* By selecting the first option, the operator  */
1757                          /* deletes all the files associated with the    */
1758                          /* current patient name.  This is a dangerous    */
1759                          /* but important option, for doing this manually*/
1760                          /* is too time intensive since all the          */
1761                          /* subsequent patient numbers and their         */
1762                          /* associated files have to be decremented so    */
1763                          /* that the program does not run out of unused   */
1764                          /* patient numbers.                             */
1765
1766                          if( input==1 ) {
1767                               if( strcmp( pat_name, "" ) == 0 ) {
1768                                      printf("\nNo patient name specified!!");
1769                                      getch();
1770                                      }
1771
1772                               else {
1773                                      /* Get total number of patients */
1774
1775                                      system("dir c:\\patients\\patient.* /B >
1776   temp.dat");
1777                                      infile = fopen("temp.dat","r");
1778                                      i=0;
1779                                      while( fscanf(infile,"%s",temp_name1) !=
1780   EOF )
1781                                              i++;
1782                                      fclose(infile);
1783                                      itoa( i, temp_name3, 10);
1784
1785                                      /* Get number corresponding to patient
1786   name (j) */
1787
1788                                      for( j=0; j<4; j++ )     /* gets patient
1789   number */
1790                                              temp_name3[j] = pat_file[j+20];
1791
1792                                      j = atoi( temp_name3 );
1793
1794                                      /* All the files associated with*/
```

```
1795                                    /* this patient number are now  */
1796                                    /* deleted.                     */
1797
1798                                    strcpy( temp_name1, "del c:\\data\\dat*."
1799    );
1800                                    strcat( temp_name1, temp_name3 );
1801                                    system( temp_name1 );
1802
1803                                    strcpy( temp_name1, "del
1804    c:\\patients\\patient.");
1805                                    strcat( temp_name1, temp_name3 );
1806                                    system( temp_name1 );
1807
1808                                    strcpy( temp_name1, "del
1809    c:\\analysis\\an*." );
1810                                    strcat( temp_name1, temp_name3 );
1811                                    system( temp_name1 );
1812
1813
1814                                    /* All the patient nunmbers      */
1815                                    /* higher than the one deleted   */
1816                                    /* are now iteratively renamed   */
1817                                    /* with a decremented suffix.    */
1818
1819                                    for( k=(j+1); k<=i; k++ ) {
1820                                            itoa( k, temp_name3, 10 );
1821                                            if( k<10 ) {
1822                                                    temp_name4[0] = '0';
1823                                                    temp_name4[1] = '0';
1824                                                    temp_name4[2] = 0;
1825                                                    strcat( temp_name4,
1826    temp_name3 );
1827                                                    }
1828
1829                                            else if( k<100 ) {
1830                                                    temp_name4[0] = '0';
1831                                                    temp_name4[1] = 0;
1832                                                    strcat( temp_name4,
1833    temp_name3 );
1834                                                    }
1835                                            else
1836                                                    strcpy( temp_name4,
1837    temp_name3 );
1838
1839                                            strcpy( temp_name3, temp_name4 );
1840
1841                                            itoa( (k-1), temp_name2, 10 );
1842                                            if( (k-1)<10 ) {
1843                                                    temp_name4[0] = '0';
1844                                                    temp_name4[1] = '0';
1845                                                    temp_name4[2] = 0;
1846                                                    strcat( temp_name4,
1847    temp_name2 );
1848                                                    }
1849
1850                                            else if( (k-1)<100 ) {
1851                                                    temp_name4[0] = '0';
1852                                                    temp_name4[1] = 0;
1853                                                    strcat( temp_name4,
1854    temp_name2 );
1855                                                    }
1856                                            else
1857                                                    strcpy( temp_name4,
1858    temp_name2 );
1859
1860                                            strcpy( temp_name2, temp_name4 );
1861
1862                                            strcpy( temp_name1, "rename
1863    c:\\patients\\patient." );
1864                                            strcat( temp_name1, temp_name3 );
1865                                            strcat( temp_name1, " patient." );
1866                                            strcat( temp_name1, temp_name2 );
```

```
1867                                        system( temp_name1 );
1868
1869                                        strcpy( temp_name1, "dir
1870   c:\\data\\dat*." );
1871                                        strcat( temp_name1, temp_name3 );
1872                                        strcat( temp_name1, " /B > temp.dat"
1873   );
1874                                        system( temp_name1 );
1875                                        infile = fopen( "temp.dat" ,"r" );
1876                                        while( fscanf(infile, "%s",
1877   temp_name4) != EOF ) {
1878                                                temp_name4[9] = 0;
1879                                                strcpy( temp_name1, "rename
1880   c:\\data\\" );
1881                                                strcat( temp_name1,
1882   temp_name4 );
1883                                                strcat( temp_name1,
1884   temp_name3 );
1885                                                strcat( temp_name1, " " );
1886                                                strcat( temp_name1,
1887   temp_name4 );
1888                                                strcat( temp_name1,
1889   temp_name2 );
1890                                                system( temp_name1 );
1891                                        }
1892                                        fclose( infile );
1893
1894                                        strcpy( temp_name1, "dir
1895   c:\\analysis\\an*." );
1896                                        strcat( temp_name1, temp_name3 );
1897                                        strcat( temp_name1, " /B > temp.dat"
1898   );
1899                                        system( temp_name1 );
1900                                        infile = fopen( "temp.dat", "r" );
1901                                        while( fscanf(infile, "%s",
1902   temp_name4) != EOF ) {
1903                                                temp_name4[8] = 0;
1904                                                strcpy( temp_name1, "rename
1905   c:\\analysis\\" );
1906                                                strcat( temp_name1,
1907   temp_name4 );
1908                                                strcat( temp_name1,
1909   temp_name3 );
1910                                                strcat( temp_name1, " " );
1911                                                strcat( temp_name1,
1912   temp_name4 );
1913                                                strcat( temp_name1,
1914   temp_name2 );
1915                                                system( temp_name1 );
1916                                        }
1917                                        fclose( infile );
1918
1919                                }
1920
1921                        strcpy( pat_name, "" );
1922                        strcpy( pat_file, "" );
1923                        strcpy( out_name, "" );
1924                        outname = out_name;       /* otherwise for
1925   some reason outname points to the string value of the total number of
1926   patients (ie. well done to Microsoft for another bug-free product) */
1927                        strcpy( in_name, "" );
1928
1929                }
1930
1931
1932
1933            }
1934
1935
1936                /* Selecting option 2 displays all the patient  */
1937                /* names currently on record, with the number of*/
1938                /* monitoring session s which have occurred in   */
```

```
1939                          /* brackets after each name.  For this, the    */
1940                          /* number of data files for each patient is     */
1941                          /* counted.                                     */
1942
1943                 else if( input==2 ) {
1944                          _clearscreen(_GCLEARSCREEN);
1945                          system( "dir c:\\patients\\patient.* /B >
1946     temp.dat" );
1947                          infile = fopen("temp.dat","r");
1948                          i=0;
1949                          while ( fscanf(infile, "%s", temp_name1) != EOF )
1950     {
1951                                  i++;
1952
1953                                  strcpy( temp_name2, "c:\\patients\\" );
1954                                  strcat( temp_name2, temp_name1 );
1955                                  strcpy( temp_name1, temp_name2 );
1956
1957                                  for( j=0; j<4; j++ )     /* gets patient
1958     number */
1959                                          temp_name3[j] = temp_name1[j+20];
1960
1961                                  outfile = fopen(temp_name1,"r");
1962                                  fscanf(outfile, "%s
1963     %s",temp_name1,temp_name2);
1964                                  fclose(outfile);
1965
1966                                  strcat( temp_name1, " " );
1967                                  strcat( temp_name1, temp_name2 );
1968
1969
1970                          /* Gets the number of data files for    */
1971                          /* this patient, storing result in j.   */
1972                          /* Assume that there are no unanalysed */
1973                          /* data files.                          */
1974
1975                                  strcpy( temp_name2, "dir c:\\data\\dat*."
1976     );
1977                                  strcat( temp_name2, temp_name3 );
1978                                  strcat( temp_name2, " /B > temp.dat" );
1979                                  system( temp_name2 );
1980                                  outfile = fopen( "temp.dat", "r" );
1981                                  j=0;
1982                                  while( fscanf(outfile,"%s",temp_name3) !=
1983     EOF )
1984                                          j++;
1985
1986                                  fclose(outfile);
1987                                  itoa( j, temp_name3, 10 );
1988
1989                                  strcat( temp_name1, "(" );
1990                                  strcat( temp_name1, temp_name3 );
1991                                  strcat( temp_name1, ")" );
1992
1993                                  if( (i%4)==0 )
1994                                          i=1;
1995                                  if( ((i%2)==0) || ((i%3)==0) )
1996                                          printf("%25s",temp_name1);
1997                                  else
1998                                          printf("\r\n%25s",temp_name1);
1999
2000                                  }
2001
2002                          fclose(infile);
2003
2004                          }
2005
2006
2007                  /* This option was included to aid the         */
2008                  /* operator, for if the patient name needed to  */
2009                  /* be changed to the one to be deleted then the */
2010                  /* operator would otherwise have to traverse    */
```

```
2011                          /* through many menus to get to the one where    */
2012                          /* he would be able to change the patient name. */
2013
2014                          else if( input==3 ) {
2015                                  printf("\nWhat is the new patient's name? : ");
2016                                  scanf("%s %s", temp_name1, temp_name2);
2017                                  case_convert( temp_name1 );
2018                                  case_convert( temp_name2 );
2019                                  strcat ( temp_name1, " " );
2020                                  strcat ( temp_name1, temp_name2 );
2021
2022                                  strcpy( temp_name3, get_patient_file( temp_name1
2023      ) );
2024                                  if ( strcmp( temp_name3, "unsuccessful") ) {
2025                                          strcpy( pat_name, temp_name1 );
2026                                          strcpy( pat_file, temp_name3 );
2027                                          printf("\nPatient name accepted.");
2028                                          strcpy( inname, "" );
2029                                          strcpy( outname, "" );
2030                                          }
2031                                  else
2032                                          printf("\nPatient name does not exist
2033      !!");
2034
2035                                  }
2036
2037
2038                          /* Selecting option 4 returns the operator to the root
2039      */
2040                          /* menu.
2041      */
2042
2043                          else if( input==4 )
2044                                  flag1=ON;
2045
2046
2047                          else
2048                                  printf("\nThe range is from 1 to 4.  Please try
2049      again.");
2050
2051
2052                          if( flag1==OFF ) {
2053                                  printf("\n\nPress a key to continue.");
2054                                  getch();
2055                                  }
2056
2057
2058                          }
2059
2060                  }
2061
2062
2063          /* Selecting option 6 exits the program by breaking from this    */
2064          /* loop, for there is no code (except to reset the video mode to*/
2065          /* what had been previously selected) in the main function       */
2066          /* afterwards.                                                   */
2067
2068          else if (input==6) {
2069                  break;
2070                  }
2071
2072
2073          else {
2074                  printf("\nThe range is from 1 to 6. Please try again.");
2075                  getch();
2076                  }
2077
2078
2079          }
2080
2081  _setvideomode(_DEFAULTMODE);
2082
```

```
2083     }
2084
2085
2086     /* This function is used to calculate the %age weight-bearing from the  */
2087     /* inputted analysis file name and patient body mass.  The analysis file*/
2088     /* is accessed and the session's average weight-bearing value obtained. */
2089     /* This is divided by the inputted patient's mass and multiplied by 100 */
2090     /* to obtain the %age weight-bearing.                                   */
2091
2092     float calc_weight_bearing( char *file_name, int body_mass )
2093
2094     {
2095     float weight_bearing;
2096     int i;
2097     char temp_str[50];
2098     FILE *fptr;
2099
2100     strcpy( temp_str, "c:\\analysis\\" );
2101     strcat( temp_str, file_name );
2102
2103     fptr = fopen( temp_str,"r" );
2104     for( i=0; i<29; i++ )
2105            fscanf( fptr, "%s", temp_str );
2106
2107     weight_bearing = atoi( temp_str );
2108     weight_bearing /= (float) body_mass;
2109     weight_bearing *= 100.0;
2110
2111     return( weight_bearing );
2112
2113     }
2114
2115
2116     /* The following function is used to calculate the number of weeks      */
2117     /* post-fracture.  The inputs are the date of the fracture             */
2118     /* ( '.._start' ) and the date post-fracture ( '.._end' ).  The        */
2119     /* intervening number of weeks is calculated by first calculating the  */
2120     /* intervening number of days, and then converting this to weeks.  If  */
2121     /* there are 4 days or over remaining, then this is rounded up to an    */
2122     /* extra week.                                                          */
2123     */
2124
2125     int calc_no_of_weeks(int year_start,int month_start,int day_start,int
2126     year_end,int month_end, int day_end)
2127
2128     {
2129     int no_of_weeks=0,i,day_month_start=0,day_month_end=0;
2130     int days_month[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
2131
2132     no_of_weeks = (year_end-year_start)*52;
2133
2134     for( i=0; i<month_start; i++ )
2135            day_month_start += days_month[i];
2136     for( i=0; i<month_end; i++ )
2137            day_month_end += days_month[i];
2138     day_month_end = day_month_end-day_month_start;
2139
2140     day_month_end += day_end-day_start;
2141
2142     if( (day_month_end%7) > 3 )
2143            no_of_weeks++;
2144
2145     no_of_weeks += day_month_end/7;
2146
2147     return( no_of_weeks );
2148
2149     }
2150
2151
2152     /* This function is used to obtain the date from an inputted data or    */
2153     /* analysis file name.  The date is returned in the standard            */
2154     /* day/month/year format.                                              */
```

```
2155
2156    char * get_date( char *file_name )
2157
2158    {
2159    static char date[10];
2160    char temp_str[15];
2161    int i;
2162
2163    strcpy( temp_str, file_name );
2164    strcpy( date, "" );
2165
2166    if ( temp_str[0]=='A' )
2167            i=2;
2168
2169    else if ( temp_str[0]=='D' )
2170            i=3;
2171
2172    else
2173            strcpy( date, "ERROR" );
2174
2175
2176    if( date[0] != 'E' ) {
2177            date[0] = temp_str[i++];
2178            date[1] = temp_str[i++];
2179            date[2] = '/';
2180            date[3] = temp_str[i++];
2181            date[4] = temp_str[i++];
2182            date[5] = '/';
2183            date[6] = '9';
2184            date[7] = temp_str[i++];
2185            date[8] = 0;
2186            date[9] = 0;
2187            }
2188
2189    return( date );
2190
2191    }
2192
2193
2194    /* This function returns the patient file name corresponding to the    */
2195    /* inputted patient name.  Each patient file is opened in turn and the  */
2196    /* stored patient name compared to the one inputted until a match is    */
2197    /* found, the file name being then returned.                           */
2198
2199    char * get_patient_file( char *patient_name )
2200
2201    {
2202    char static patient_file[40];
2203    char temp_name1[40], temp_name2[40], temp_name3[40], temp_name4[40];
2204    int flag=OFF, i;
2205    FILE *fptr1, *fptr2;
2206
2207    system( "dir c:\\patients\\patient.* /B > temp.dat" );
2208    fptr1 = fopen( "temp.dat", "r" );
2209    while( fscanf( fptr1, "%s", temp_name1 ) != EOF ) {
2210            strcpy( temp_name2, "c:\\patients\\" );
2211            strcat( temp_name2, temp_name1 );
2212            strcpy( temp_name1, temp_name2 );
2213            fptr2 = fopen( temp_name1, "r" );
2214            fscanf( fptr2, "%s %s", temp_name3, temp_name4 );
2215            fclose( fptr2 );
2216            strcat( temp_name3, " " );
2217            strcat( temp_name3, temp_name4 );
2218
2219            if ( strcmp( temp_name3, patient_name ) == 0 ) {
2220                    strcpy( patient_file, temp_name2 );
2221                    flag=ON;
2222                    }
2223
2224            }
2225
2226    if ( flag==OFF )
```

```
2227          strcpy( patient_file, "unsuccessful" );
2228
2229     fclose( fptr1 );
2230     return( patient_file );
2231
2232     }
2233
2234
2235     /* This function is called when changing the data file being accessed.  */
2236     /* The date is requested with each value being validated as reasonable. */
2237     /* If a patient name is not specified then this is also requested and    */
2238     /* the data file name constructed.                                       */
2239
2240     void input_file( char *file_name, char *pat_name, char *pat_file )
2241
2242     {
2243     int i, flag1, flag2, flag3, flag4, input;
2244     static char temp_name[40];
2245     char temp_name1[40], temp_name2[40], temp_name3[40];
2246     char temp_name4[40], temp_name5[40], temp_name6[15];
2247
2248     strcpy( temp_name, "" );
2249     strcpy( file_name, "" );
2250     flag1=OFF;
2251     while ( flag1==OFF ) {
2252          printf("\nInput date of the monitoring session:\nDay of month: ");
2253          scanf("%d",&input);
2254          if ( (input>0) && (input<32) )
2255               flag1=ON;
2256          else
2257               printf("\nThe range is from 1 to 31. Please try again.\n");
2258          }
2259     temp_name1[0] = input/10+48;
2260     temp_name1[1] = input-((input/10)*10)+48;
2261     temp_name1[2] = 0;
2262     strcat( temp_name, temp_name1 );
2263
2264     flag1=OFF;
2265     while ( flag1==OFF ) {
2266          printf("Month: ");
2267          scanf("%d",&input);
2268          if ( (input>0) && (input<13) )
2269               flag1=ON;
2270          else
2271               printf("\nThe range is from 1 to 12. Please try again.\n");
2272          }
2273     temp_name1[0] = input/10+48;
2274     temp_name1[1] = input-((input/10)*10)+48;
2275     temp_name1[2] = 0;
2276     strcat( temp_name, temp_name1 );
2277
2278     flag1=OFF;
2279     while ( flag1==OFF ) {
2280          printf("Year (eg. 1993): ");
2281          scanf("%d",&input);
2282          if (input>1991)
2283               flag1=ON;
2284          else
2285               printf("\nThe range is from 1993 onwards. Please try
2286     again.\n");
2287          }
2288     temp_name1[0] = input-1990-((input-1990)/10)*10+48;
2289     temp_name1[1] = 0;
2290     strcat( temp_name, temp_name1 );
2291     strcat( temp_name, "." );
2292
2293     strcpy( temp_name1, pat_name );
2294
2295     if ( temp_name1[0] == 0 ) {       /* ie. temp_name1 = "" */
2296          printf("\nInput patient name in the following format. ");
2297          printf("\nPatient Name ('first name' 'second name'): ");
2298          scanf( "%s %s", temp_name1, temp_name2 );
```

```
2299            case_convert( temp_name1 );
2300            case_convert( temp_name2 );
2301            strcat( temp_name1, " " );
2302            strcat( temp_name1, temp_name2 );
2303            }
2304
2305    system( "dir c:\\patients\\patient.* /B > temp.dat" );
2306    outfile = fopen("temp.dat","r");
2307    flag1=OFF;
2308    while( (fscanf(outfile, "%s", temp_name2) != EOF) && (flag1==OFF) ) {
2309            strcpy( temp_name6, temp_name2 );
2310            strcpy( temp_name3, "c:\\patients\\" );
2311            strcat( temp_name3, temp_name2 );
2312            strcpy( temp_name2, temp_name3 );
2313            infile = fopen(temp_name2,"r");
2314            fscanf( infile, "%s %s", temp_name3, temp_name4 );
2315            fclose( infile );
2316            strcat( temp_name3, " " );
2317            strcat( temp_name3, temp_name4 );
2318
2319            if ( strcmp(temp_name3, temp_name1) == 0 )
2320                    flag1=ON;
2321
2322            }
2323    fclose( outfile );
2324
2325    if ( flag1==ON ) {
2326            printf("\nPatient name accepted.");
2327            temp_name2[0] = temp_name6[8];
2328            temp_name2[1] = temp_name6[9];
2329            temp_name2[2] = temp_name6[10];
2330            temp_name2[3] = 0;
2331
2332            strcat( temp_name, temp_name2 );
2333            strcpy( pat_name, temp_name1 );
2334
2335            strcpy( pat_file, "C:\\PATIENTS\\PATIENT." );
2336            strcat( pat_file, temp_name2 );
2337
2338            }
2339
2340    else {
2341            printf("\nThe inputted patient name '%s' does not exist in the
2342    records.",temp_name1);
2343            strcpy( temp_name, "OFF" );
2344            }
2345
2346    strcpy( file_name, temp_name );
2347
2348    }
2349
2350
2351    /* This function is called after every name inputted, for it converts   */
2352    /* the case of that name.  All its letters are converted to lower case   */
2353    /* except the first which is converted to upper case.                    */
2354
2355    void case_convert( char *name )
2356
2357    {
2358    int i;
2359
2360    while( *name<65 )          /* in case any spaces before text in input string
2361    */
2362            name++;
2363
2364    if ( *name>90 )           /* put first character in upper case */
2365            *name -= 32;
2366    name++;
2367
2368    for ( i=0; *name!=0; i++ ) {
2369            if ( (*name<91) && (*name>64) )
```

```
2370                    *name += 32;     /* if any letters upper case, put in lower
2371    case */
2372         else if ( *name==32 ) {
2373                 while( *name==32 ) /* incase 2 or more spaces in between names
2374    */
2375                         name++;
2376                 if ( *name>90 ) /* put first character in upper case */
2377                         *name -= 32;
2378                 else if ( *name==0 ) /* in case a space at end of names */
2379                         break;
2380                 name++;
2381                 }
2382         else
2383                 name++;
2384
2385         }
2386
2387    }
2388
2389
2390    /* This function obtains the patient name by accessing the inpuuted    */
2391    /* patient file name.                                                  */
2392
2393    char * get_name( char *file_name )
2394
2395    {
2396    int i;
2397    FILE *file;
2398    static char string[] = "unsuccessful";
2399    static char patient_name[40];
2400    char tmp[40];
2401
2402    if ( (file = fopen(file_name,"r")) == NULL )
2403         return(string);
2404    else {
2405         fscanf(file,"%s",patient_name);
2406         fscanf(file,"%s",tmp);
2407         strcat( patient_name, " " );
2408         strcat( patient_name, tmp );
2409         fclose(file);
2410         return(patient_name);
2411         }
2412
2413    }
2414
2415
2416    /* This function returns the analysis file name from the inputted file  */
2417    /* name stub.                                                           */
2418
2419    char * get_outname(char *inname)
2420
2421    {
2422    FILE *file;
2423    float a=0.1;
2424    char *outname;
2425    static char tmpstr[13];
2426
2427    tmpstr[0]  = 'A';
2428    tmpstr[1]  = 'N';
2429    tmpstr[2]  = inname[3];
2430    tmpstr[3]  = inname[4];
2431    tmpstr[4]  = inname[5];
2432    tmpstr[5]  = inname[6];
2433    tmpstr[6]  = inname[7];
2434    tmpstr[7]  = inname[8];
2435    tmpstr[8]  = inname[9];
2436    tmpstr[9]  = inname[10];
2437    tmpstr[10] = inname[11];
2438    tmpstr[11] = 0;
2439    tmpstr[12] = 0;
2440
2441    outname = tmpstr;
```

```
2442
2443    return( outname );
2444
2445    }
2446
2447
2448
```