INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

Development of Graphical User Interface for the PT Relays

by

Jingxuan Hu

A Thesis

Submitted to the Faculty of Graduate Studies

in partial fulfillment of the requirements

for the degree of

Master of Science

Department of Electrical and Computer Engineering

University of Manitoba

Winnipeg, Manitoba, Canada

THE UNIVERSITY OF MANITOBA

FACULTY OF GRADUATE STUDIES

*****

MASTER'S THESIS/PRACTICUM FINAL REPORT

The undersigned certify that they have read the Master's Thesis/Practicum entitled:

DEVELOPMENT OF GRAPHICAL USER INTERFACE FOR THE PT RELAY

submitted by

JINGXUAN HU

in partial fulfillment of the requirements for the degree of

M. Sc.

The Thesis/Practicum Examining Committee certifies that the thesis/practicum (and oral examination if required) is:

APPROVED

(Approved or Not Approved)

☑ **Thesis** ☐ **Practicum**

Advisor: Peter C. McLaren

Edward W. Castle

Date: 27/07/01

KW

(forms\thereptm - 08/95)

THE UNIVERSITY OF MANITOBA

FACULTY OF GRADUATE STUDIES

*****

COPYRIGHT PERMISSION

Development of Graphical User Interface for the PT Relays

By

Jingxuan Hu

A Thesis/Practicum submitted to the Faculty of Graduate Studies of The University of Manitoba in partial fulfillment of the requirement of the degree

of

MASTER OF SCIENCE

Acknowledgments

I wish to express my sincere gratitude to Dr. P.G. McLaren for choosing such an attractive topic for my master's studies. His guidance, advice and support encouraged me to implement this work.

I would also like to thank Mr. E. Dirks for his great assistance in developing this graphical distance relay.

Financial supports from Manitoba Hydro and NESRC are greatly acknowledged.

I extend my deep gratitude to my beloved husband Pei Wang for his strong support and encouragement during my studies. To my dearest parents and brother, I will keep all their supports in my heart.

Abstract

Computer relaying has played an important role in power system protection as technology advances. Due to the great flexibility of computer-based relays, it will be applicable to design a general user programmable relay to meet the various relaying applications in the utilities. The power group in University of Manitoba has successfully implemented such kind of distance relay prototype based on Digital Signal Processor (DSP), in which different applications adopted a generic piece of hardware capable of implementing all of the relaying functions but with a capability to expand as the need arises. However, the configuration/modification of custom relay algorithms still remain difficult for the relay engineers since a thorough understanding of C/Assembly language and hardware knowledge is required.

In this thesis, a Graphical User Interface (GUI) for the DSP-based distance relay was developed, which would assist the relay engineer to design a suitable algorithm for the application and then configure a general purpose relay hardware to run algorithms. The key part of the developed Graphical User Interface is a graphical block library of basic functions for distance relaying. Based on the developed GUI, a Graphical DSP-based Distance Relay (GDDR) was implemented. In addition, evaluation of the GDDR relay performance was carried out using laboratory tests, and the effects of System Impedance Ratio (SIR), fault types, and fault location on the fault pick-up time were investigated.

Table of Contents

List of Symbols

GDDR: Graphical DSP-based Distance Relay

FACTS: Flexible AC Transmission System

PSII: Positive Sequence Incremental Impedance

SIR: System Impedance Ratio (Source Impedance/Line Impedance)

DFT: Discrete Fourier Transform

FFT: Fast Fourier Transform

IIR: Infinite Impulse Filter

FIR: Finite Impulse Filter

DSA: Digital Signal Analyzer

RTP: Real Time Playback

DSP: Digital Signal Processor

L_{ld_ld}: Logic output signal of 'LI' block

L_{Dir}: Logic output of 'DI' block

DE: Direction Element

Dir: Logic output of 'DE' block

PS: Phase Selection

Φ_{ab1}: Ratio of ΔI_{ca} to ΔI_{ab}

Φ_{ab2}: Ratio of ΔI_{bc} to ΔI_{ab}

Φ_{bc1}: Ratio of ΔI_{ab} to ΔI_{bc}

Φ_{bc2}: Ratio of ΔI_{ca} to ΔI_{bc}

Φ_{ca1}: Ratio of ΔI_{bc} to ΔI_{ca}

Φ_{ca2}: Ratio of ΔI_{ab} to ΔI_{ca}

S_a: Phase selection flag for phase A

S_b: Phase selection flag for phase B

S_c: Phase selection flag for phase C

S_{ab}: Phase selection flag for phase AB

S_{bc}: Phase selection flag for phase BC

S_{ca}: Phase selection flag for phase CA

S_Φ: Phase Selection Flag, Φ is a, b, c, ab, bc or ca.

The symbols below represent phasors

Z_a: Ground Impedance Element for Phase A

Z_b: Ground Impedance Element for Phase B

Zc: Ground Impedance Element for Phase C

Z_{ab}: Phase Impedance Element for Phase AB

Z_{bc}: Phase Impedance Element for Phase BC

Z_{ca}: Phase Impedance Element for Phase CA

Z_{l0}: Zero Sequence Impedance of Protected Transmission Line

Z_{l1}: Positive Sequence Impedance of Protected Transmission Line

ΔZ_l: Positive Sequence Incremental Impedance

Z_s: Positive Sequence Impedance from the relay busbar to the source E_s

Z_s': Positive sequence impedance from the fault point to the source E_s'

V_a: Phase A Voltage

V_b: Phase B Voltage

V_c: Phase C Voltage

E_a: Phase A Voltage

E_b: Phase B Voltage

E_c: Phase C Voltage

E_0: Zero Sequence Voltage

E_1: Positive Sequence Voltage

E_2: Negative Sequence Voltage

I_a: Phase A current

I_b: Phase B current

I_c: Phase C current

I_0: Zero Sequence Current

I_1: Positive Sequence Current

I_2: Negative Sequence Current

ΔI_{ab}: Incremental Phase A to Phase B current

ΔI_{bc}: Incremental Phase B to Phase C current

ΔI_{ca}: Incremental Phase C to Phase A current

ΔI_1: Incremental Positive Sequence Current

DI: Directional Impedance

LI: Load Impedance

List of Figures

List of Tables

Chapter 1 Introduction

1.1 Computer Relaying

1.1.1 History of Protection Relaying

Protection relaying is widely adopted to ensure the safe and continuous power supply of a power system in which a fault has developed [1-5]. As an important role in power protection, protection relaying provides two main functions. Firstly, protective relaying functions to remove as speedily as possible any section of a power system when it suffers a short circuit, or when it starts to operate in any abnormal manner that might cause damage or interfere with the effective operation of the rest of the system. The secondary function of protective relaying is to provide indication of the location and type of failure, which may not only assist in expediting repair, but also provide means for analyzing the effectiveness of the fault prevention.

A protection relay scheme is composed of one relay or a group of relays. The most typical relay schemes may be sorted into five categories: Overcurrent relay, Directional relay, Distance relay, Unit protection and Balanced current protection.

The ***Overcurrent relay***, which is widely used on low voltage distribution networks, operates when the quantity of current exceeds the high set current in the line section which develops a fault. The ***Directional relay*** aims to detect the direction using the product of the current and voltage, and they usually are used to obtain the

directional sensitivity for other relays such as overcurrent and impedance relays. The ***Distance relay*** defines the relays whose response to the input quantities is primarily a function of the electrical circuit distance between relay location and the point of the fault. ***Unit protection*** protects a system by comparing the current entering and leaving it, which should be the same under normal condition and during an external fault. ***Balanced current protection*** is the protection scheme for parallel circuits of the same impedance. Normally, parallel circuits carry an equal current (balanced current), which will change on the occurrence of a fault on all of the parallel circuits.

Different types of relay hardware have been developed and employed in power system protection as technology advances. Typically, they may be classified into three categories according to their nature: Electromechanical Relays, Static Relays and Computer Relays.

1.1.2 Computer Relays

The field of computer relaying started with relay engineers' interests in using digital computers in power system protection. Over the last forty years, computer relaying has made great strides with the fast improvement of computer techniques, from the first application using a digital computer in the 1960's to the present microprocessor based relays. Computer relays are currently playing a major role in power system protection and the typical architecture of a computer relay is shown in Figure 1.1. Compared to the other types of relays, computer relays offer the followings advantages.

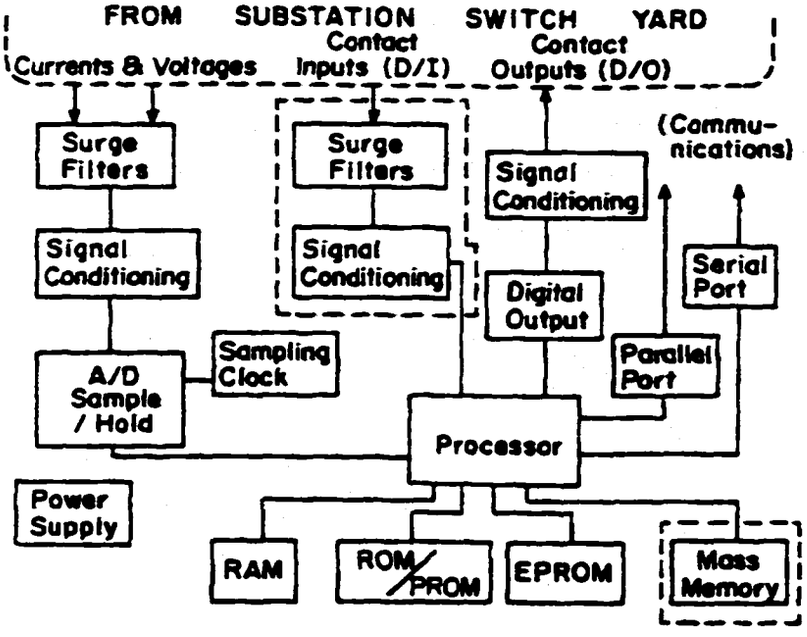


Figure 1.1 A Typical Architecture of Computer Relay [5]

- Cost- using a computer relay in some situations will be more cost effective than the traditional relays.
- Self-Checking and Reliability- this feature provides more ways to test the relay.
- Functional Flexibility - Relay algorithms can be programmed to perform several functions to meet the requirements of different protection schemes. The settings can be adjusted conveniently and promptly because of programmability and communication capability of the computer based relays.
- Adaptive *Relaying* - Adaptive features can be added.

1.1.3 Adaptive relaying

Adaptive protection has been used in power system protection to some extent. Time-delay overcurrent relays adapt the operating time to fault current magnitude, and directional relays adapt to the direction of the fault current. However, these are permanent characteristics of relay systems and are included as part of the original relay system to perform a pre-defined function. None of these has implemented the adaptive protection concept in a comprehensive sense as an on-line, real-time tool.

With the development of digital relays, the possibility of using digital techniques to implement adaptive protection occurred to many researchers. They addressed different adaptive protection concepts in different perspectives. Horowitz, Phadke and Thorp [6] described the results of an investigation into the possibilities of using digital techniques to adapt transmission system protection and make real-time changes. They defined adaptive protection as "*a protection philosophy which permits and seeks to make adjustments to various protection functions in order to make them more attuned to prevailing power system conditions*". Jampala, Venkata and Damborg [7] offered a different description of the concept with a different perspective, which is "*ability of the protection system to automatically alter its operating parameters in response to changing network conditions to maintain optimal performance*".

There exist two important characteristics in any protection scheme: security and dependability. Dependability measures the relaying equipment's ability to correctly clear a fault while security is a measure of the relaying equipment's' tendency not to trip incorrectly. Once a non-adaptive relaying system has been designed and installed, its security and dependability are fixed and cannot respond to changing system conditions.

Thus, there is always a compromise between security and dependability for non-adaptive relaying schemes. On the contrary, adaptive relaying schemes, which make the adjustment to various protection functions, are capable of improving relaying reliability and power system security plus achieving other benefits, such as the improvement of speed and sensitivity to various faults under different conditions without losing selectivity.

1.2 Background of this work.

Commonly, the design of the relays depends mainly on the manufacturers to meet the utility relaying problem. Since there are various specific relaying applications in the utilities, it will be a tendency to design a general user programmable relay and the configuration/modification of the relays can be easily controlled and mastered by the utility relay engineers. With the great flexibility of the microprocessor-based relays, this approach may become a possibility and occurred to many researchers.

Dr. McLaren et al [8] pointed out the concept of " open" system relaying and successfully implement its prototype, in which different applications adopted a generic piece of hardware capable of implementing all of the relaying functions but with a capability to expand as the need arises. Based on their work, the power group in ECE department of University of Manitoba has developed several versions of computer (DSP)-based distance relays currently called the PT relay for the protection of the power transmission lines. The PT relay is an adaptive quadrilateral relay and Figure 1.2 shows the structure of the PT relay [9].

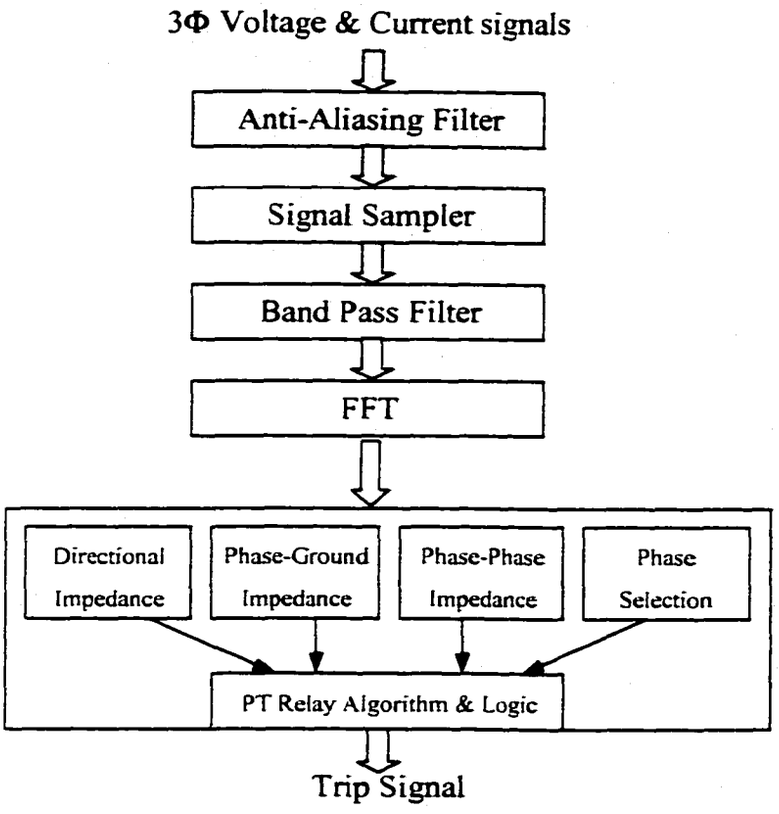
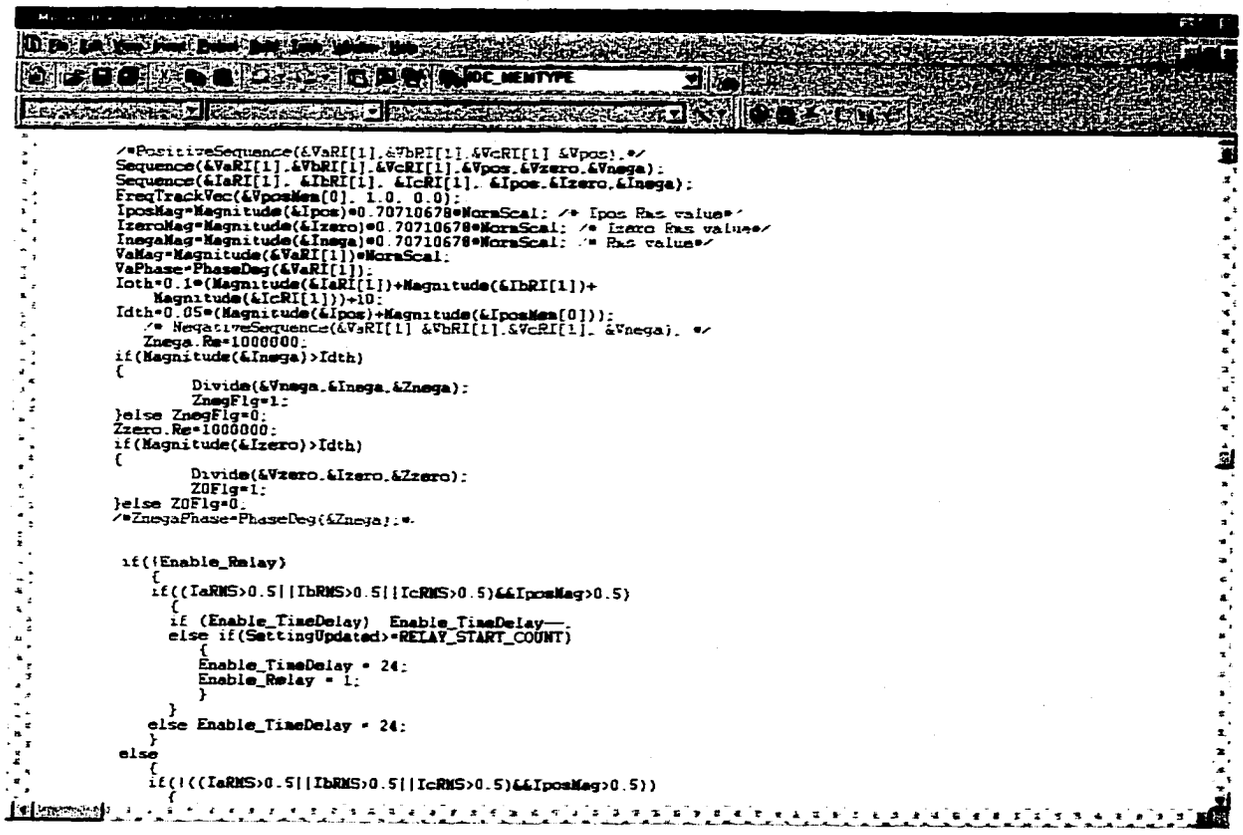


Figure 1.2 Schematic diagram of the PT relay system [9]

The algorithms of the PT relay are written in C with embedded Assembly language to optimize the codes and a typical code section of the PT relay algorithm is shown in Figure 1.3. With the consistent efforts of numerous researchers, the algorithms of the PT relay have been greatly improved and verified in either simulation or real-time environment. The PT relay can also provide good graphic interfaces to display measured results. However, Its text formatted C/assembly programming still remains difficult for the utility relay engineers since a thorough understanding of C/Assembly language and hardware knowledge is required to design the custom relay algorithms.



```
/*PositiveSequence(&VaRI[1],&VbRI[1],&VcRI[1] &Vpos);*/
Sequence(&VaRI[1],&VbRI[1],&VcRI[1],&Vpos,&Vzero,&Vnega);
Sequence(&IaRI[1], &IbRI[1], &IcRI[1], &Ipos,&Izero,&Inega);
FreqTrackVec(&VposMem[0], 1.0, 0.0);
IposMag=Magnitude(&Ipos)*0.70710678*NormScal; /* Ipos Rms value*/
IzeroMag=Magnitude(&Izero)*0.70710678*NormScal; /* Izero Rms value*/
InegaMag=Magnitude(&Inega)*0.70710678*NormScal; /* Rms value*/
VaMag=Magnitude(&VaRI[1])*NormScal;
VaPhase=PhaseDeg(&VaRI[1]);
Ioth=0.1*(Magnitude(&IaRI[1])+Magnitude(&IbRI[1])+
    Magnitude(&IcRI[1]))+10;
Idth=0.05*(Magnitude(&Ipos)+Magnitude(&IposMem[0]));
   /* NegativeSequence(&VsRI[1] &VbRI[1],&VcRI[1], &Vnega); */
   Znega.Re=1000000;
if(Magnitude(&Inega)>Idth)
{
        Divide(&Vnega,&Inega,&Znega);
        ZnegFlg=1;
}else ZnegFlg=0;
Zzero.Re=1000000;
if(Magnitude(&Izero)>Idth)
{
        Divide(&Vzero,&Izero,&Zzero);
        ZOFlg=1;
}else ZOFlg=0;
/*ZnegaPhase=PhaseDeg(&Znega);*/

 if(!Enable_Relay)
    {
    if((IaRMS>0.5||IbRMS>0.5||IcRMS>0.5)&&IposMag>0.5)
      {
      if (Enable_TimeDelay)  Enable_TimeDelay--;
      else if(SettingUpdated>=RELAY_START_COUNT)
        {
        Enable_TimeDelay = 24;
        Enable_Relay = 1;
        }
      }
    else Enable_TimeDelay = 24;
    }
 else
    {
    if(!((IaRMS>0.5||IbRMS>0.5||IcRMS>0.5)&&IposMag>0.5))
      {
```

Figure1.3 A Code Section of PT Relay Algorithm

With the advent of the commercial software RIDE, a graphic user interface may be possible which would assist the relay engineer to design a suitable algorithm for the application and then configure a general purpose relay hardware to run such an algorithm. RIDE is short for "Real-time Integrated Development Environment". Described as a superset of the Hypersignal Block Diagram visual environment, it adds support for the design, implementation and analysis of real-time DSP algorithms and systems. Figure 1.4 shows a typical block diagram application built in the Hypersignal RIDE environment. By dividing the PT relay algorithm into basic functions and rewriting them

with block diagram format in the RIDE environment, the configuration of relays may be implemented by relay engineers with more facility without special computer knowledge. Thus, it is of interest to investigate the possibility of developing such a programmable graphic user interface.

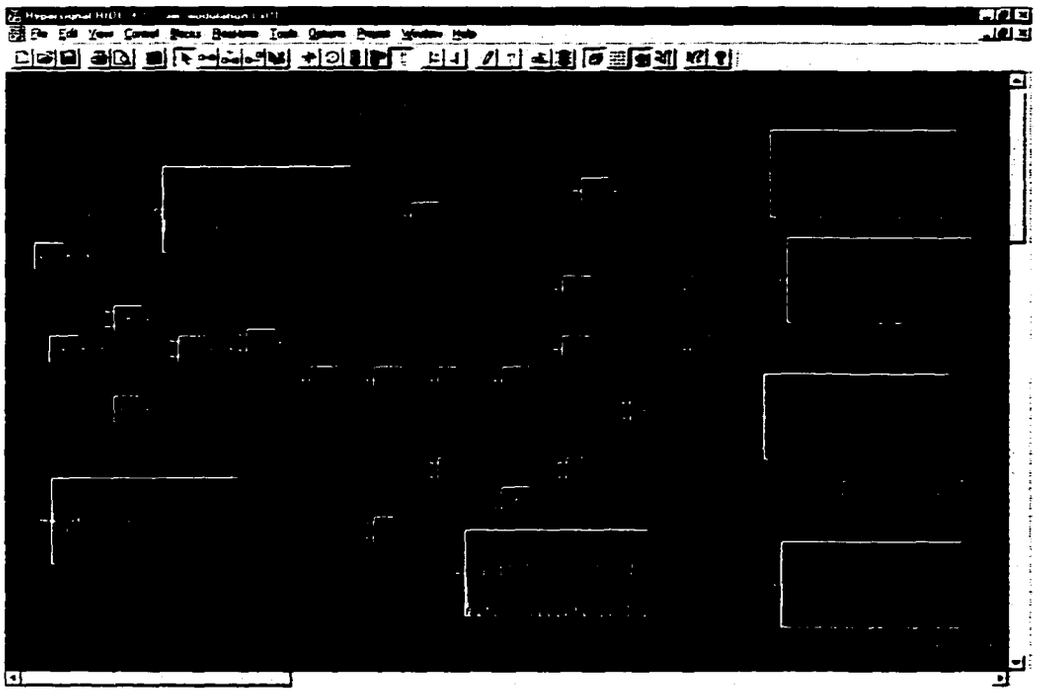


Figure 1.4 A Typical Block Interface of RIDE

1.3 Scope of the present work

This thesis aims to develop a Graphical User Interface using Digital Signal Processor hardware (TMS320C30 DSP board and PC) and commercial software Hypersignal RIDE, TI compiler and Visual C++. This interface will provide graphical blocks for the basic

relay algorithms, so relay engineers can build their custom relays with more flexibility. This thesis includes the following works:

1. To develop the 'windows driver' program for the TMS320C30 DSP board using Hypersignal Driver Wizard to set up the communication between the DSP and host PC.
2. To build a graphical block diagram library of the basic functions for a distance relay using Visual C++, TI compiler and Hypersignal RIDE.
3. To implement a Graphical DSP-based Distance Relay (GDDR) for the transmission line protection using the developed Graphical User Interface.
4. To verify the developed GDDR relay using the power system simulation program PSCAD/EMTDC and the RTP.
5. To study the effects of fault location, fault type and System Impedance Ratio (SIR) on the tripping time using the developed GDDR relay.
6. To investigate the effects of adaptive relaying using the developed GDDR relay under different fault conditions.

Chapter 2 Development of a graphical DSP-based Distance Relay (GDDR)

2.1 The GDDR relay

The GDDR relay was developed based on the PT (Power Tower) relay originally designed by the UM power system group. The GDDR relay is designed to protect power system transmission lines. A typical block diagram for the GDDR relay is shown in Figure 2.1.

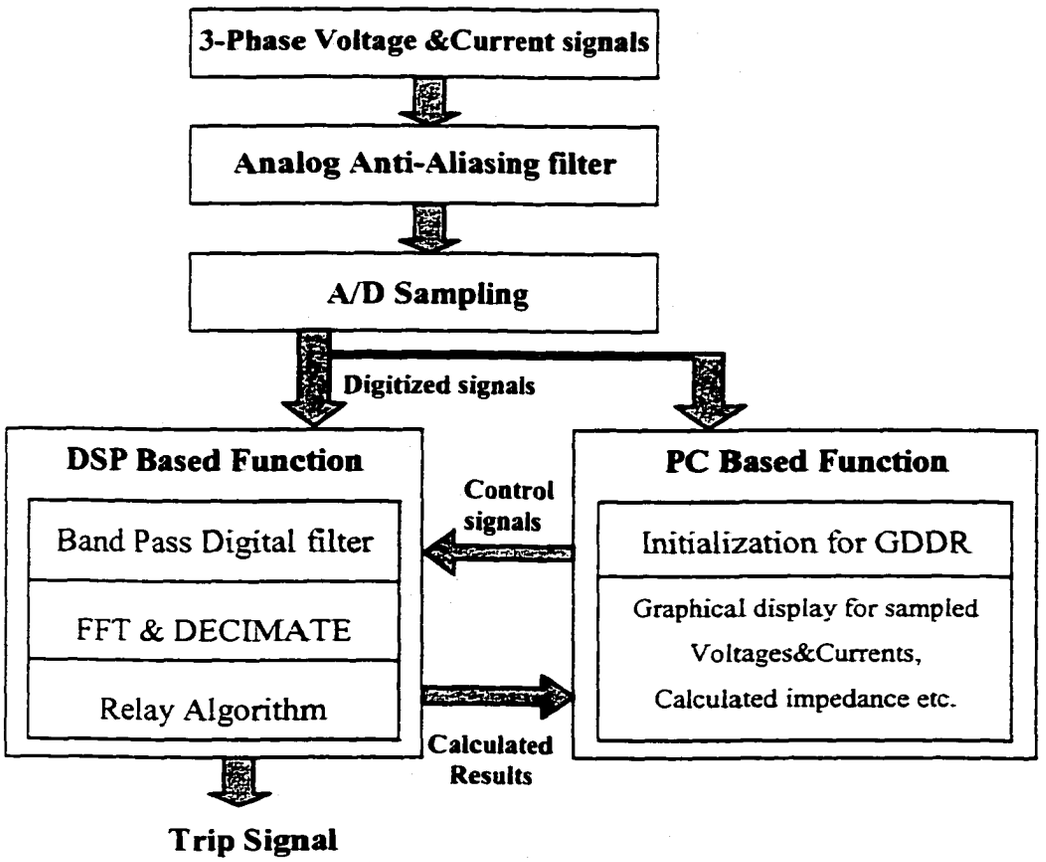


Figure 2.1 Typical Block Diagram of GDDR relay

2.1.1 Anti-Aliasing Filter

According to sampling theory, the system sampling rate must be greater than twice the highest frequency in the original analog signal. Otherwise, the original signal can not be recovered and represented by the sampled data. A low pass filter which removes the frequencies higher than half the sampling rate is referred to as an anti-aliasing filter.

In this work, the semicondutor chip LTC1602 is employed to implement the function of anti-aliasing. The LTC1602 is a 5^{th} order lowpass filter with no DC error. Its unusual architecture puts the filter outside the DC path so DC offset and low frequency noise problems are eliminated. In addition, the filter cutoff frequency is set by an internal clock which can be externally driven, so the anti-aliasing filter's cut-off frequency can be adjusted to follow the power system frequency variation.

2.1.2 Analog-to-digital conversion:

To utilize the power of the computer, analog signals obtained have to be converted to digital signals. The basic conversion scheme for most analog-to-digital conversion is shown in Figure 2.2(a) [10]. The unknown voltage is connected to one input of an analog comparator and a time-dependent reference voltage is connected to the second input of the comparator.

The transfer characteristic of the comparator is shown in Figure 2.2 (b). If the input voltage V_1 is greater than V_2, the output voltage will be at a positive level corresponding to a logic "1". If input V_2 is greater than V_1, the output voltage will be at a low level, corresponding to logic "0".

To perform a conversion, the reference voltage V_R is varied to determine which of the 2^n possible binary words is closest to the unknown voltage Vx. The reference voltage V_R can assume 2^n different values of the form

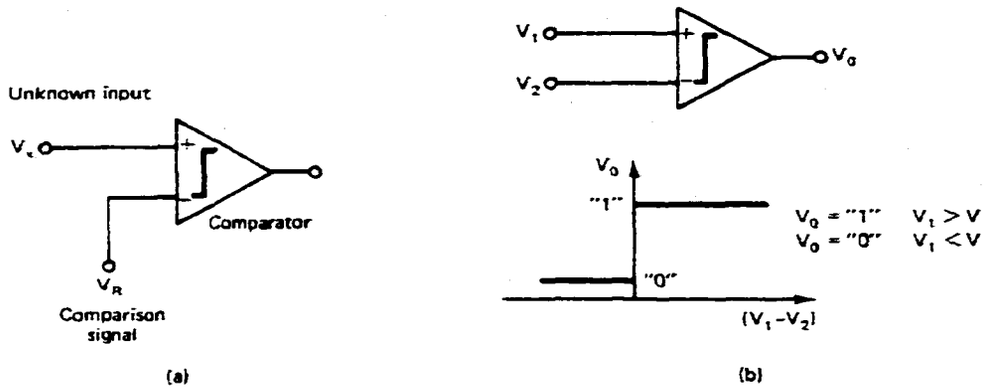


Figure 2.2 Analog-to-digital conversion: (a) general scheme; (b) Comparator [10]

$$V_R = V_0 \sum_{i=1}^{n} A_i 2^{-i} \qquad (2\text{-}1)$$

where V_0 is a dc reference voltage and Ai are binary coefficients. The logic of the A/D converter attempts to choose the coefficients Ai so that the difference between the unknown input V_x and the set of possible discrete representations of V_R is a minimum:

$$error = |V_x - V_R| = \left| V_x - V_0 \sum_{i=1}^{n} A_i 2^{-i} \right| \qquad (2\text{-}2)$$

2.1.3 Digital Bandpass Filter

Filters are used to pass or attenuate (block) a certain frequency range of a signal and can be implemented by either analog or digital approaches. Due to advantages such as software programmability, stability and predictability, no drift with temperature or humidity and superior performance-to- cost ratio, digital filters are gaining popularity.

Depending on the frequency range that they either pass or attenuate, filters can be classified into four types: lowpass filters, high pass filters, bandpass filters(to pass a certain band of frequencies) and bandstop filters (to attenuate a certain band of frequencies). Digital filters can also be classified as FIR (finite impulse response) filter and IIR (infinite impulse response) based on their impulse response, the response of a filter to an input that is an impulse. The impulse response of FIR filters falls to zero after a finite amount of time while the response of IIR filters exists indefinitely.

In this work, although the original analog signals are sampled at one rate for display purposes, the samples are decimated to a lower sampling rate before being applied to the relay algorithm. This results in a fast execution time, but may cause an aliasing effect. In addition, sub-harmonics may exist in the signals. To eliminate the above problems, a bandpass digital filter was designed and employed.

2.1.4 DFT& FFT

The Discrete Fourier Transform (DFT) is used to obtain the representation of the finite-length signal (sampled version of the original signal) in the frequency domain, while the FFT is one of the fast computation methods of implementing multiple DFT. For a finite length signal x[n] defined over the range 0<n<N , the DFT of such a signal is given by

$$X[k] = \sum_{n=0}^{k-1} x[n] * \exp(\frac{-j2\pi kn}{N})$$
$$= \sum_{n=0}^{k-1} x[n] * [\cos(\frac{2\pi kn}{N}) - j\sin(\frac{2\pi kn}{N})] \qquad (2\text{-}3)$$

For periodic and real signals in a power system, the DFT coefficients X[k] actually provide the amplitude and phase information of the harmonic components of the

original signals. Therefore, the real part $R_e(X[k])$ and imaginary part $I_m(X[k])$ of the DFT coefficients can be shown as the following

$$R_e(X[k]) = \sum_{n=0}^{k-1} x[n] * \cos(\frac{2\pi kn}{N})$$

$$I_m(X[k]) = -\sum_{n=0}^{k-1} x[n] * j\sin(\frac{2\pi kn}{N}) \qquad (2\text{-}4)$$

In terms of the amplitude and phase, these equations can be recast as

$$|X[k]| = [R_e^{\,2}(X[k]) + I_m^{\,2}(X[k])]^{\frac{1}{2}}$$

$$\theta[k] = tg^{-1}[\frac{I_m(X[k])}{R_e(X[k])}] \qquad (2\text{-}5)$$

where k is the harmonic number.

The FFT/DFT is a basic for digital relay technique, and the relay algorithms are achieved after the fundamental components of signals are extracted using DFT/FFT.

2.1.5 Relay Algorithm

The developed relay algorithm is based on the PT relay and includes the following main functions.

- To calculate the phase-phase/phase-ground impedance
- To calculate Direction Impedance (Incremental Positive Sequence Impedance) and pre-fault Impedance (Load Impedance)
- To determine the fault type and fault direction
- To adapt the trip zone for the faulted phase
- To send a trip signal if a fault is detected.

2.2 Organization of the Block Diagram for the GDDR relay

To develop a graphical user interface for the GDDR relay using Hypersignal RIDE, the relay algorithm was divided into individual function blocks to obtain the most flexible and independent utilization of each block. In this way, the relay algorithm can be easily built, configured, modified and executed. According to their functions, the blocks were assigned into different libraries and groups as listed in Table 2.1.

Table 2.1 Description of Block Function for GDDR

Note: RT.SI and UC represent Real-time, Simulation and User Control block respectively

Group List	Function (block) List	Block Function Description
A/D Functions	Daqu	Anti_aliasing filter and A/D sampling, RT
Elements	DE	Direction Element, determine the fault direction. RT
	DirLmp	Calculate direction impedance, RT
	LoadImp	Calculate load impedance, RT
	PS	Determine faulted phase and fault type, RT
Region	AdapZone	Set parameter for the T_Line protected, adjust the trip zone if there is a fault, RT
	TripZone	Determine whether the calculated impedance run into the trip zone, RT
	ZoneDis	Draw the trip zone and impedance, SI
Relay Type	Ground Relay	Calculate phase to ground impedance, RT
	Phase Relay	Calculate phase to phase impedance, RT
Sequence	Sequence 120	Calculate sequence components, RT
Arithmetic	Incremental Caculator	Calculate the incremental value of input signals,RT
Hierarchy	DEC&FFT	Composed of Decimate and FFT blocks, RT
Fault Element	FltDetect	Send a trip signal to serial port if there is a fault, RT
Display	2-channel X Display	Display phase current and voltage, SI
	Digital Display	Display Parameters, SI
	XY Display	Display trip zone and calculated impedance, SI

RT data Transfer	DSP to PC	Upload data to PC
	PC to DSP	Download data to DSP
User Control	Horizontal Slider	Slider to visually control the parameters, UC

2.3 Creation of custom function blocks using Hypersignal Block Wizard

RIDE provides both the simulation and real-time function libraries, which include some general function blocks such as Arithmetic blocks etc. These functions are ready to use and can be easily connected to implement some applications. However, the custom function blocks of the GDDR relay as described in Section 2.2 have to be created in the RIDE to implement the relay algorithms. Hypersignal RIDE includes the Block Wizard which can facilitate the creation of user defined blocks (DLLs) for use within Hypersignal Block Diagram/RIDE. The following steps briefly illustrate the procedure of creating such a custom function block.

1. Define the block characteristics using prompted dialog boxes of the Block Wizard. The main characteristics include the name of the custom block, the name of the library storing the block, block pattern (Simulation, Real-time or User control), block type (Input, Output or Process), input/output number and some user defined parameters. Figure 2.3 shows a typical dialog box of Block Wizard to set the library, group and menu names for the custom block.
2. Generate Source Code. After all the block characteristics have been defined, the Block Wizard will create 14 files for a simulation block or 16 files for a real-time block. These source files are used to set-up the basic code frame of the custom block.

3. Produce the DLL file using Microsoft Visual C ++(Version 6.0). This step is the core part of the process, and it includes editing the source file (Block.C) to add code that implements the custom block function, compiling and linking the source code to create the block DLL. For a real-time block, the code to perform a custom function will be added to the DSP source file (*_Block*.C). Then, the DSP source file will be compiled using the TI compiler and the DSP object file with .OBJ extension will be generated. (*Block* is the name of the custom block)

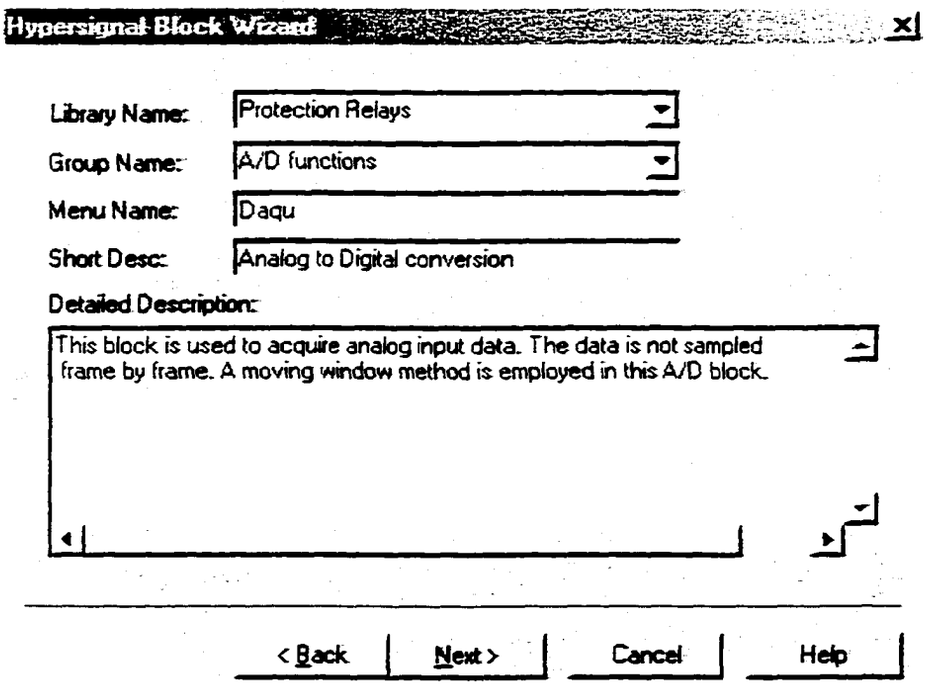


Figure 2.3 Dialog box of Block Wizard

4. Add the custom block to Hypersignal RIDE. Copy the generated file *Block*.DLL (*block* is name of new block) and Object file (Real-time block) to the corresponding directory, then the custom block can be created in RIDE by using the "Add New

Block" function. The custom block will be located in the library defined in Block Wizard Dialog Box.

5. Use the custom block. The custom block can be selected using the RIDE Block Function Selector and be added to the worksheet for use.

In this work, there are 13 custom function blocks developed for the GDDR relay to implement the basic relay algorithm. These blocks can be used independently and are stored into different groups in the library of "Protection Relays". Details of each block will be explained in the following section.

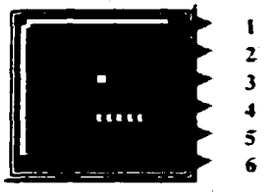
2.4 Developed library of graphical blocks for GDDR relay

2.4.1 A/D block function

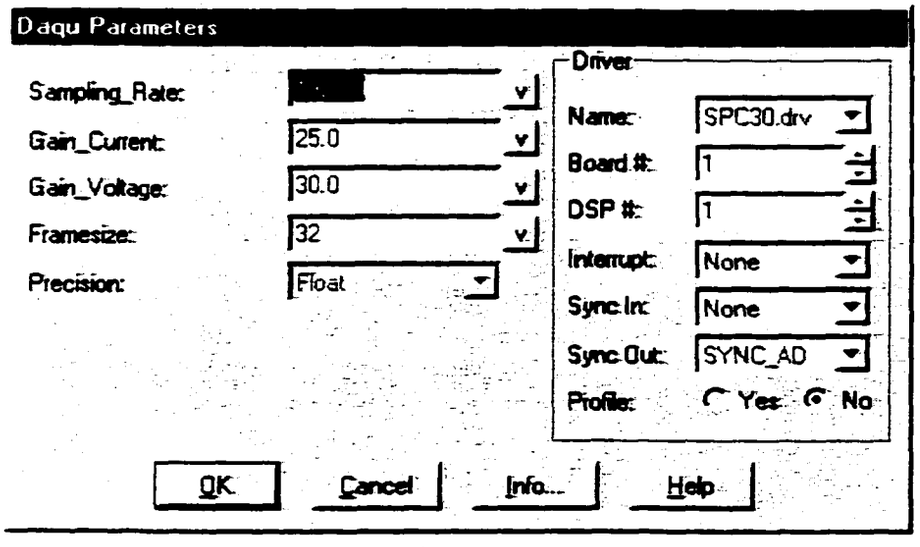
The employed A/D hardware is a custom 12-bit analog-to-digital converter with 9 simultaneous sampled independent input channels designed by Mr. E. Dirks, the technologist in this Department. For the sampling of the original signals, a "moving data window" approach is used. After the first sampling cycle, only one new sampled point is moved into the data window to replace the oldest sampled point. The "moving data window" is a power system cycle long and is a basic unit for calculation. The use of a moving window allows for faster detection of a signal change than the cycle by cycle sampling approach.

Figure 2.4 (a) shows the A/D sampling block interface which was built in Hypersignal RIDE. It has six outputs: outputs 1-3 are the three phase current signals, and outputs 4-6 output the three phase voltage signals. The parameter settings interface is shown in Figure 2.4 (b). Parameter Sampling_Rate is used to set the sampling rate of the

A/D board, while Framesize determines the number of samples to be acquired and output to the subsequent blocks. Parameters Gain_Current and Gain_Voltage can be set to adjust the gain of current and voltage signals sampled.



(a) A/D block Interface

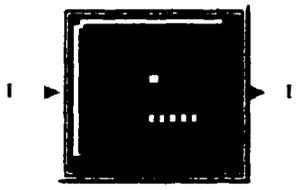


(b) Parameter Settings Interface

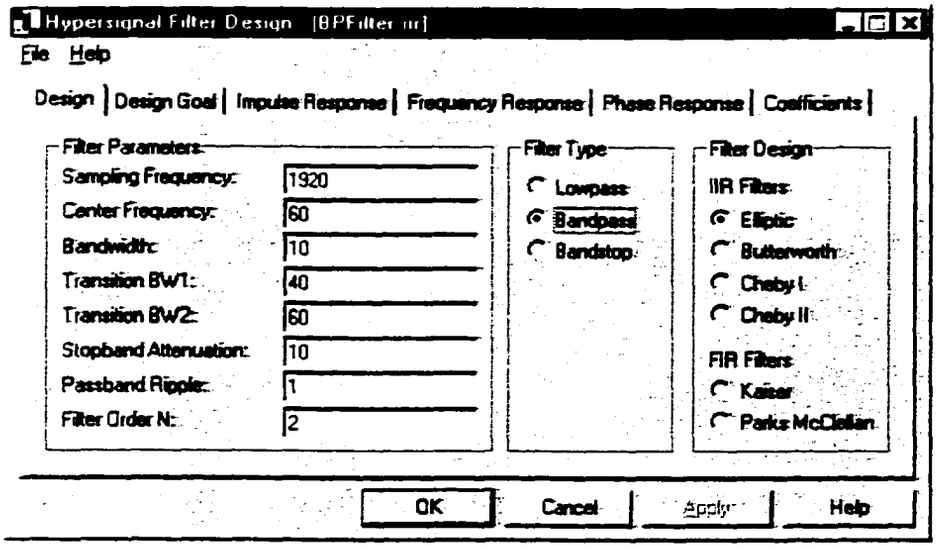
Figure 2.4 Interface of Analog to Digital Function

2.4.2 Digital filter function

The digital bandpass filter is employed to remove both low frequency and high frequency components from the sampled signals. Figure 2.5 (a) displays the block interface, while the interface for parameter settings supported by RIDE is shown in Figure 2.5 (b).



(a) Block Interface



(b) Parameter Settings Interface

Figure 2.5 Interface of Digital Bandpass Filter Block

To design a digital filter, select Filter Type and Filter Design (IIR or FIR filter) first. Then, the filter parameters shown in the left side of the box can be set according to the design requirements. RIDE also provides a testing function for the developed filter.

Once the design is done, the design goal, impulse response, frequency response and phase response can be used to check the design and effect of the filter. If necessary, adjustments can be applied to the filter. In this relay, the centre frequency is 60 Hz, and the other parameters are set according to the relay requirements.

2.4.3 Decimate/FFT block

The DEC&FFT block has two functions: signal decimation, and FFT calculation. The original signal is sampled at 64 or 32 points per power system cycle for a smooth waveform display. However, to reduce the execution time of the algorithms, the sampled signals (64 or 32 samples) will be decimated to 8 samples. The decimated data will then be processed with a FFT to obtain the fundamental frequency information of the filtered sampled signals. As shown in Figure 2.6, there are three inputs that can be either voltage or current signals. The six outputs represent both the real and imaginary parts of the fundamental components of the input signals respectively.

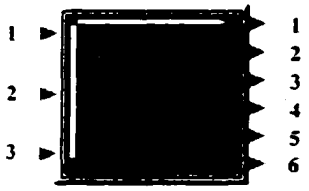
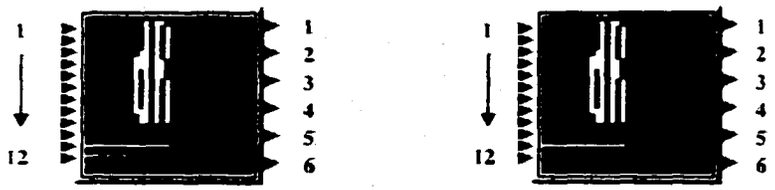


Figure 2.6 Interface of Decimate/FFT Block

2.4.4 Impedance Element



(a) Phase Distance Elements (b) Ground Distance Elements

Figure 2.7 Block Interface of Impedance Element

Function and principle:

The graphical interfaces of the impedance element block are shown in Figure 2.7. These blocks aim to measure the phase-ground and phase-phase impedance of the monitored power system. Generally, there exist six impedance measuring elements: three ground impedance elements for phase to ground faults and three phase impedance elements for phase to phase fault. The calculations are based on the following equations [13]:

Ground impedance elements:

$$Za = \frac{Va}{Ia + kIo}$$

$$Zb = \frac{Vb}{Ib + kIo}$$

$$Zc = \frac{Vc}{Ic + kIo}$$

Phase impedance elements:

$$Zab = \frac{Vab}{Ia - Ib}$$

$$Zbc = \frac{Vbc}{Ib - Ic}$$

$$Zca = \frac{Vca}{Ic - Ia} \qquad (2\text{-}6)$$

where Va, Vb, Vc, Vab, Vbc, Vca are the measured three phase- ground and phase-phase voltages at the relay location. Ia, Ib, Ic are the measured three phase currents, while Io is the zero sequence current.

The factor $k = \frac{Z_{l0} - Z_{l1}}{Z_{l1}}$ compensates for the difference between the positive and zero sequence impedance of the fault loop. Z_{l0}, Z_{l1} are the zero sequence and positive sequence impedance of the protected transmission line respectively.

In 3-pole tripping systems, no element should trip for a fault outside of its distance zone, while in single pole tripping systems, all elements should only pick up for their corresponding fault. In practice, more than one impedance distance element may

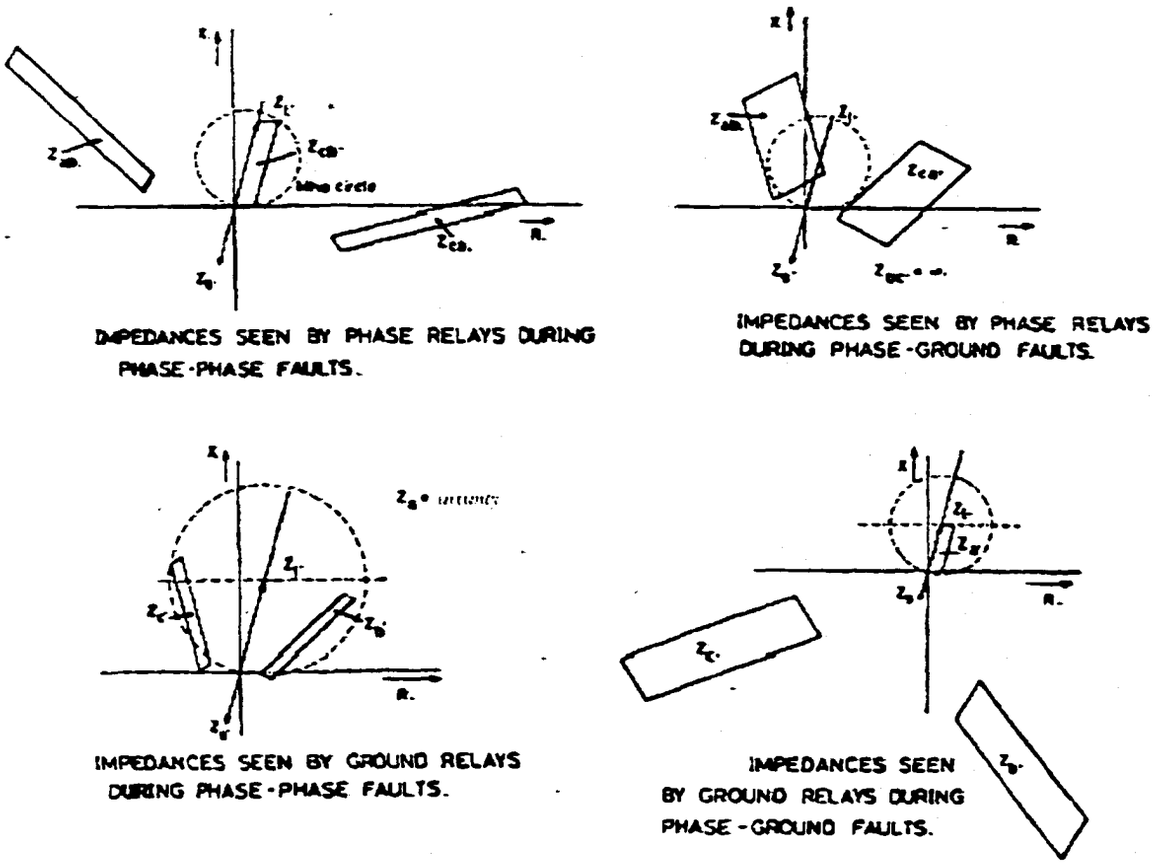


Figure 2.8 Typical zones of measurement for the 6 elements of an impedance relay [11]

pick up for the same fault (as shown in Figure 2.8 [11]), even though they were designed for different fault types. Therefore, when a single pole tripping scheme is employed, the phase selection associated with the impedance measurement is required to identify the fault type and avoid misoperation of the breaker.

Inputs and Outputs:

Inputs for both phase impedance element and ground impedance element are same.

Input1: The real part of Phase A voltage

Input2: The imaginary part of Phase A voltage.

Input3: The real part of Phase B voltage.

Input4: The imaginary part of Phase B voltage.

Input5: The real part of Phase C voltage.

Input6: The imaginary part of Phase C voltage.

Input7: The real part of Phase A current.

Input8: The imaginary part of Phase A current.

Input9: The real part of Phase B current.

Input10: The imaginary part of Phase B current.

Input11: The real part of Phase C current.

Input12: The imaginary part of Phase C current.

Outputs for Ground Impedance Element:

Output1: Resistance of Za

Ouput2: Reactance of Za

Output3: Resistance of Zb

Ouput4: Reactance of Zb

Output5: Resistance of Zc

Ouput6: Reactance of Zc

Outputs for Phase Impedance Element:

Output1: Resistance of Zab.

Ouput2: Reactance of Zab.

Output3: Resistance of Zbc.

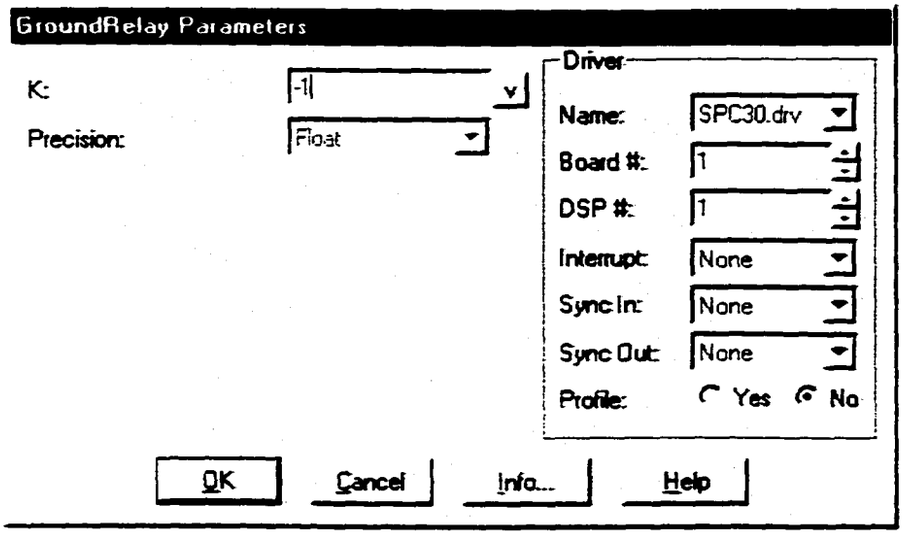
Ouput4: Reactance of Zbc.

Output5: Resistance of Zca.

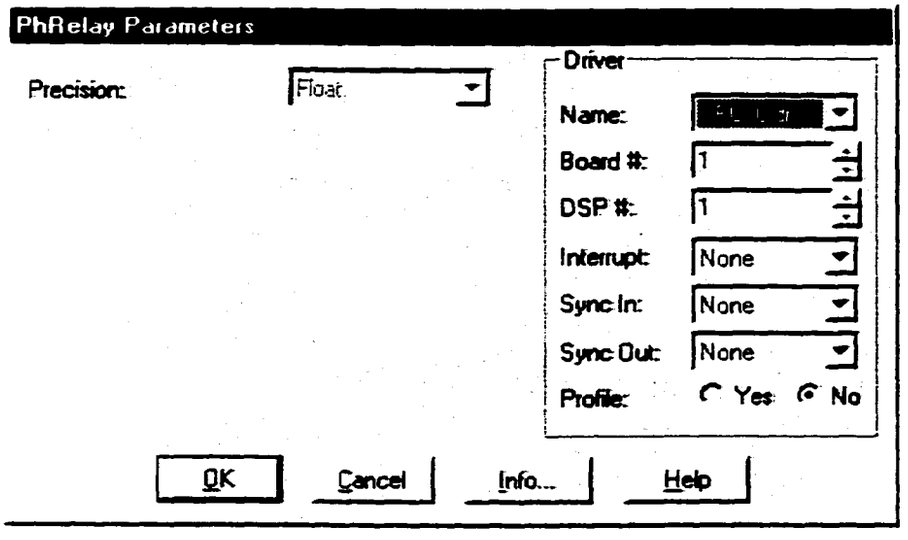
Ouput6: Reactance of Zca.

Parameters Setting:

Figure 2.9 shows the interfaces for the parameter settings. For ground impedance elements, parameter k is used to compensate for the difference between the positive and zero sequence impedance of the fault loop. The value of k can be selected and adjusted according to the properties of the protected line. There is no parameter setting for phase impedance elements.



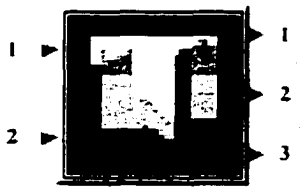
(a) Parameter Settings for Ground Element



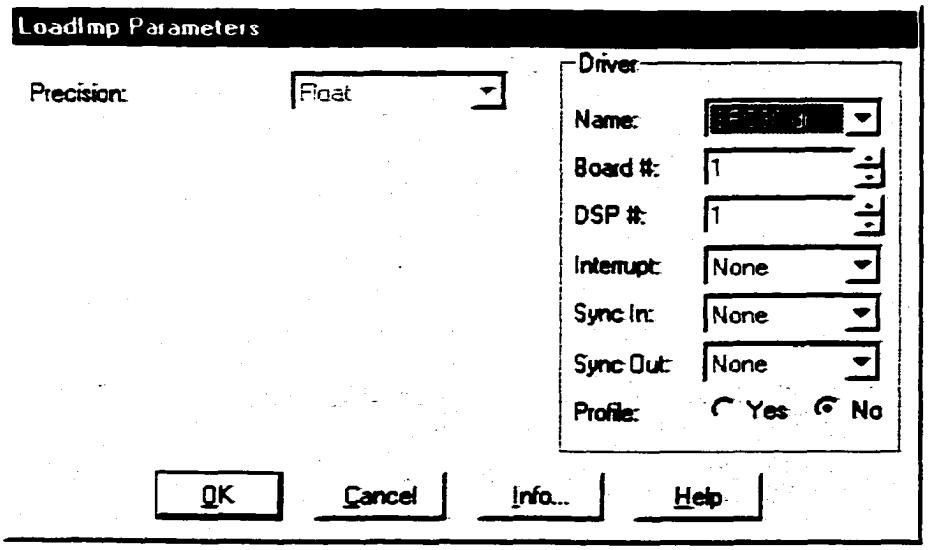
(b) Parameter Settings for Phase Element

Figure 2.9 Interface of parameter settings for Impedance Element Block

2.4.5 Load Impedance (LI) Element



(a) Block Interface



(b) Parameter Settings Interface

Figure 2.10 Interface of Load Impedance Element

Function:

The Load Impedance Element provides the pre-fault load impedance that will be used by the AdapZone block (section 2.4.11) to do the horizontal expansion (Figure 2.10). During a fault, it will provide the relay with the pre-fault load impedance within a certain period after the fault.

Inputs and Outputs:

Input1: The real part of measured impedance (resistance).

Input2: The imaginary part of measured impedance (reactance).

Output1: The real part of pre-fault impedance.

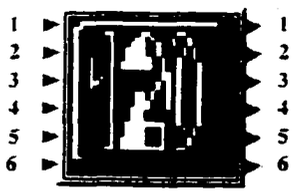
Output2: The imaginary part of pre-fault impedance.

Output3: A flag L_{ld_ld}, high level means the validation of the Output1 and 2.

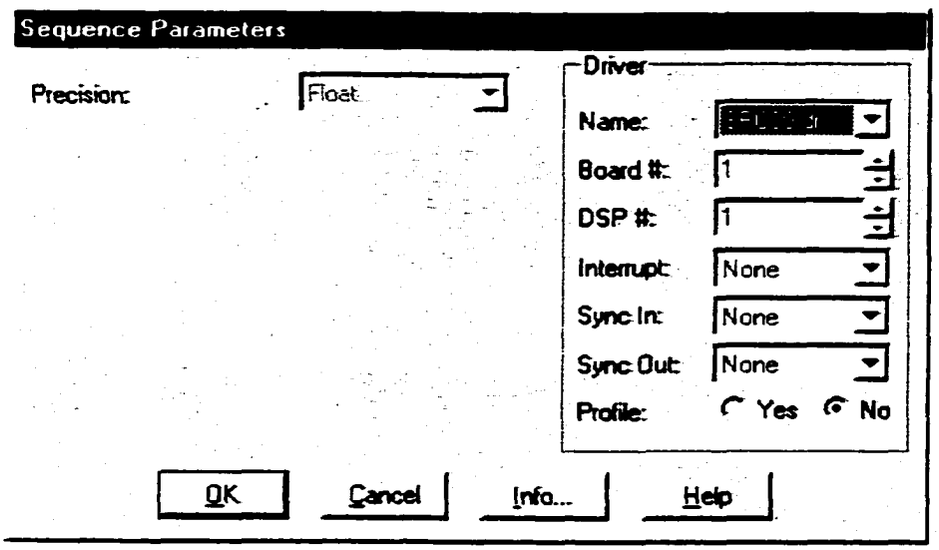
Parameter Setting:

There is no parameter setting for this block.

2.4.6 Sequence element



(a) Block Interface



(b) Parameter Settings Interface

Figure 2.11 Interface of Sequence Element

Function and Principle:

The Sequence block shown in Figure 2.11 calculates the sequence components of the three phase currents or voltages. Normally, a three-phase power system is balanced. When a fault (except for symmetrical fault) develops, the symmetry of the power system is broken and unbalanced currents and voltages appear.

Symmetrical component theory provides a method to analyze the fault conditions. By applying the 'Principle of Superposition', any general three-phase system of vectors may be replaced by three sets of balanced (symmetrical) vectors; two sets are three-phase but having opposite phase rotation and one set is co-phasal. These vector sets are labeled the 012 sets and also described as positive, negative and zero sequence sets respectively.

The relationship between phase and sequence quantities of voltage are given below:

$$\begin{bmatrix} E_a \\ E_b \\ E_c \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & a^2 & a \\ 1 & a & a^2 \end{bmatrix} \begin{bmatrix} E_0 \\ E_1 \\ E_2 \end{bmatrix}$$

$$\begin{bmatrix} E_0 \\ E_1 \\ E_2 \end{bmatrix} = \frac{1}{3} \begin{bmatrix} 1 & 1 & 1 \\ 1 & a & a^2 \\ 1 & a^2 & a \end{bmatrix} \begin{bmatrix} E_a \\ E_b \\ E_c \end{bmatrix} \qquad (2\text{-}7)$$

Where $a = -\frac{1}{2} + j\frac{\sqrt{3}}{2}$

Similarly, the relationship between phase and sequence current can be obtained as following

$$\begin{bmatrix} I_a \\ I_b \\ I_c \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & a^2 & a \\ 1 & a & a^2 \end{bmatrix} \begin{bmatrix} I_0 \\ I_1 \\ I_2 \end{bmatrix} \tag{2-8}$$

$$\begin{bmatrix} I_0 \\ I_1 \\ I_2 \end{bmatrix} = \frac{1}{3} \begin{bmatrix} 1 & 1 & 1 \\ 1 & a & a^2 \\ 1 & a^2 & a \end{bmatrix} \begin{bmatrix} I_a \\ I_b \\ I_c \end{bmatrix}$$

The positive sequence components of voltage and current are used by the DI block (section 2.4.7) to calculate the incremental positive sequence impedance, and the zero sequence current will be used by the PS block (section 2.4.9).

Inputs and Outputs:

Input1, 2: Phase A voltage or current (Real and Imaginary part)

Input3, 4: Phase B voltage or current (Real and Imaginary part)

Input5, 6: Phase C voltage or current (Real and Imaginary part)

Output1, 2: Positive sequence voltage (Real and Imaginary part).

Output3, 4: Negative sequence voltage (Real and Imaginary part).

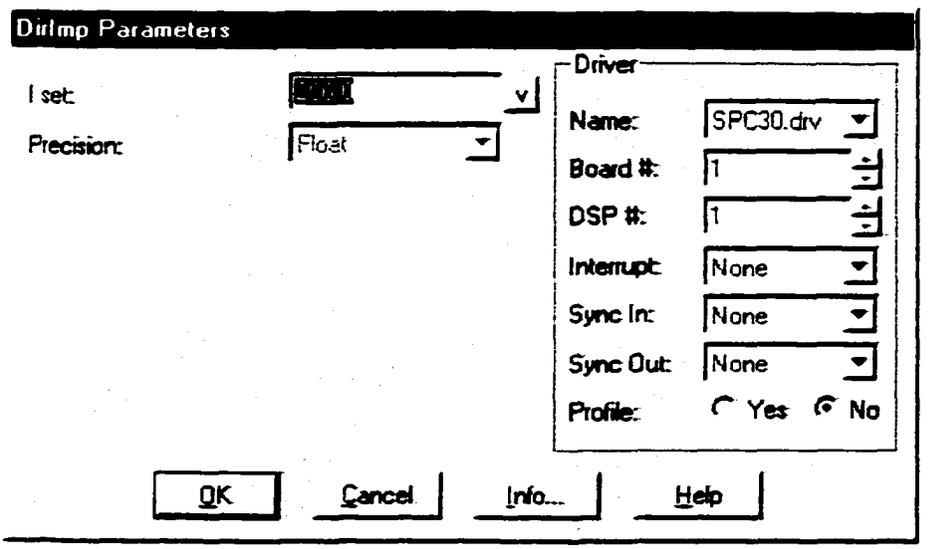
Output5, 6: Zero sequence voltage (Real and Imaginary part).

Parameters Setting:

There is no parameter setting in this block.

2.4.7 Directional Impedance (DI) Element

(a) Block Interface



(b) Parameter Setting Interface

Figure 2.12 Interface of Direction Impedance Element

Function and principle:

The Direction Impedance block as shown in Figure 2.12 is programmed to calculate the directional impedance, which will be used by the block DE (directional element, section 2.4.8) to determine whether the fault is a forward or reverse fault. The Positive Sequence Incremental Impedance (PSII) ΔZ_1 is selected to be the target directional impedance because of both convergence and its availability in various fault situations [12].

For a forward fault, in a single transmission line system, $\Delta Z_l = -Z_s$, where Z_s is the positive sequence impedance from the relay busbar to the source E_s. For a reverse fault, on the same line, $\Delta Z_l = Z_s'$, where Z_s' is the positive sequence impedance from the relay point to the remote source E_s' [13].

The inputs to this block are the incremental positive sequence voltage and current, which represent the difference between the positive sequence voltage and current of the present cycle and that of several cycles ago. When a fault occurs in the system, incremental positive sequence current ΔI_l is detected and compared to the pre-set value. If $|\Delta I_l|$ exceeds the pre-set value of the parameter (I-set), the L_{Dir} will be set to logic 1 and ΔZ_l will be calculated. Otherwise, no calculation will be done.

Inputs and Outputs:

Input1: The real part of incremental positive sequence voltage.

Input2: The imaginary part of incremental positive sequence voltage.

Input3: The real part of incremental positive sequence current.

Input4: The imaginary part of incremental positive sequence current.

Output1: The real part of incremental positive impedance ΔZ_l

Output2: The imaginary part of incremental positive impedance ΔZ_l

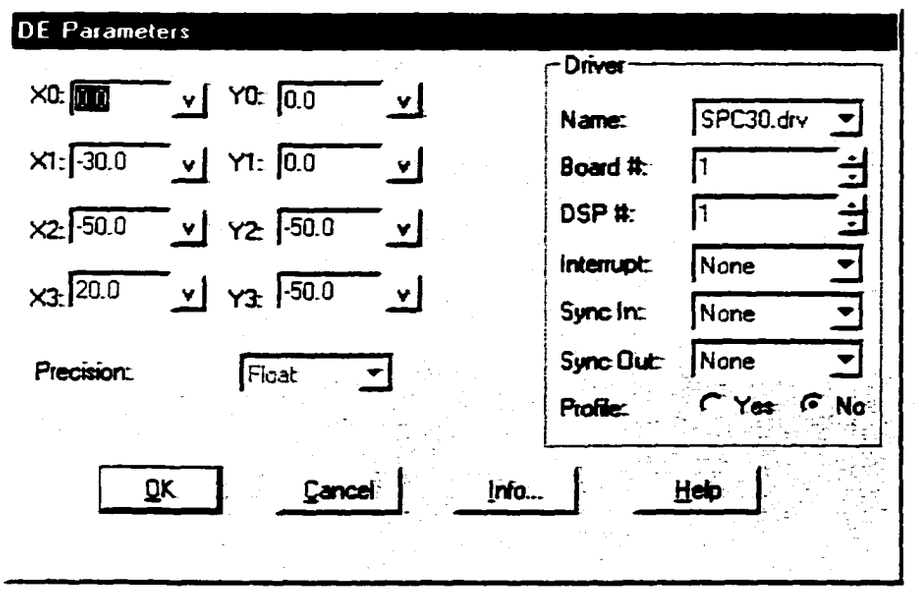
Output3: logic output L_{Dir} that determines whether ΔZ_l is valid. If L_{Dir} is high, ΔZ_l is true.

Parameters Setting:

I set: The pre-setting value of incremental positive current based on the protected system. The incremental positive sequence impedance will be calculated only if the measured incremental positive sequence current exceeds this pre-setting value.

2.4.8 Direction Element (DE)

(a) Block Interface



(b) Parameter Settings Interface

Figure 2.13 Interface of Direction Element

Function and Principle:

The Direction Element shown in Figure 2.13 determines the direction of the fault, and its inputs are the three outputs of the DI element (section 2.4.7). The DE block will function based on the logic level of the input L_{Dir}. A Logic 1 (high) of L_{Dir} will enable the DE block, and the ΔZ_1 will be compared with a pre-set zone as shown in Figure 2.14. If

the ΔZ_1 is inside the zone, the fault is a forward fault and a logic 1 will be output. On the contrary, if ΔZ_1 is outside the zone, a reverse fault is indicated and the output will be 0.

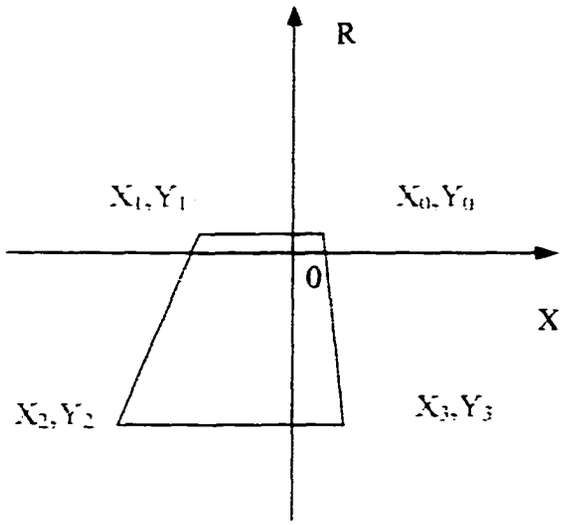


Figure 2.14 Typical Pre-set Zone of DE Element

Inputs and Outputs:

Input1: L_{Dir}, the output1 of DI element.

Input2: Real part of the Incremental positive impedance ΔZ_1 (the output2 of DI element).

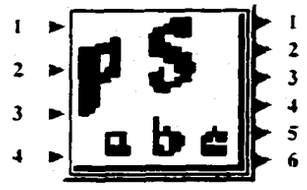
Input3: Imaginary part of the Incremental positive impedance ΔZ_1 (the output3 of DI element).

Output: Logic signal **Dir**, high level (1) indicates a forward fault and Low Level (0) indicates a reverse fault.

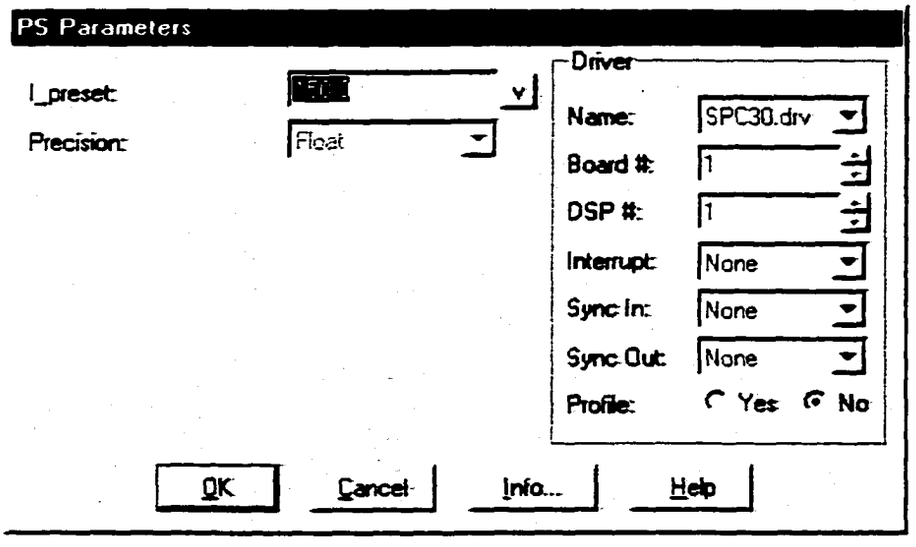
Parameter Setting:

Parameters X_0, Y_0, X_1, Y_1, X_2, Y_2, X_3, Y_3 are the coordinates of the pre-set zone, which are determined by the protected power system.

2.4.9 Phase Selection (PS) Element



(a) Block Interface



(b) Parameter Settings Interface

Figure 2.15 Interface of Phase Selection Block

Function and principle:

The Phase Selection element shown in Figure 2.15 aims to decide which measured impedance represents the impedance in the fault loop, and therefore to determine the fault type. The ratios of the phase to phase current with the combination of zero sequence current under different fault conditions are shown in Table 2.2. The algorithm of the Phase Selection element is derived from these ratios [9].

Table 2.2 The Ratio of the Incremental Phase to Phase Current [9]

Fault	I_0	Φ_{ab1}	Φ_{ab2}	Φ_{bc1}	Φ_{bc2}	Φ_{ca1}	Φ_{ca2}
AG	>0	1	0	∞	∞	0	1
BG	>0	0	1	1	0	∞	∞
CG	>0	∞	∞	0	1	1	0
AB	0	0.5	0.5	2	1	1	2
BC	0	1	2	0.5	0.5	2	1
CA	0	2	1	1	2	0.5	0.5
ABG	>0	(0.5,1)	(0.5,1)	(1,2)	1	1	(1,2)
BCG	>0	1	(1,2)	(0.5,1)	(0.5,1)	(1,2)	1
CAG	>0	(1,2)	1	1	(1,2)	(0.5,1)	(0.5,1)
ABC	0	1	1	1	1	1	1

Where $\Phi_{bc1}=\left|\frac{\Delta Iab}{\Delta Ibc}\right|$, $\Phi_{bc2}=\left|\frac{\Delta Ica}{\Delta Ibc}\right|$, $\Phi_{ab1}=\left|\frac{\Delta Ica}{\Delta Iab}\right|$, $\Phi_{ab2}=\left|\frac{\Delta Ibc}{\Delta Iab}\right|$, $\Phi_{ca1}=\left|\frac{\Delta Ibc}{\Delta Ica}\right|$, $\Phi_{ca2}=\left|\frac{\Delta Iab}{\Delta Ica}\right|$,

and ΔI_{ab}, ΔI_{bc}, ΔI_{ca} are incremental phase to phase currents. The PS element has four current related inputs and six outputs representing six phase selection flags: S_a, S_b, S_c, S_{ab}, S_{bc}, S_{ca}. These flags will be set according to the fault type, which is tabulated in Table 2.3.

Table 2.3 Setting of Phase Selection Flags (for typical fault conditions)

Fault	S_a	S_b	S_c	S_{ab}	S_{bc}	S_{ca}
AG	1	0	0	0	0	0
BC	0	0	0	0	1	0
BCG	0	1	1	0	1	0
ABC	1	1	1	1	1	1

Inputs and Outputs:

Input1: Amplitude of incremental phase A to B current ($|\Delta I_{ab}|$)

Input2: Amplitude of incremental phase B to C current ($|\Delta I_{ab}|$)

Input3: Amplitude of incremental phase C to A current ($|\Delta I_{ab}|$)

Input4: Amplitude of zero sequence current ($|I_0|$)

Output1: Phase selection flag S_a

Output2: Phase selection flag S_b

Output3: Phase selection flag S_c

Output4: Phase selection flag S_{ab}

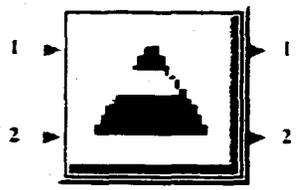
Output5: Phase selection flag S_{bc}

Output6: Phase selection flag S_{ca}

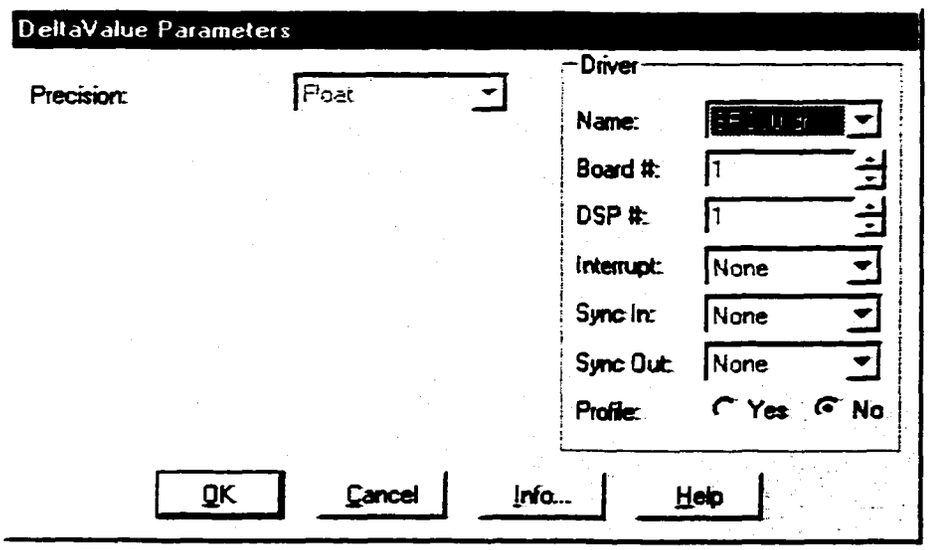
Parameters Setting:

The parameter 'I_preset' is a preset current value. The PS element will run only when the amplitude of the incremental phase to phase currents is greater than this preset current value. For different transmission line systems, this parameter should be chosen differently.

2.4.10 Incremental Calculator Block



(a) Block Interface



(b) Parameter Settings Interface

Figure 2. 16 Interface of Incremental Calculator

***Function*:**

The Incremental Calculator block (Figure 2.16) calculates the incremental value of the input signals which can be either a real value or a complex value. In the GDDR relay, it is used to calculate the incremental positive sequence voltages and currents.

***Inputs and Outputs*:**

Input1: Signal 1 to be calculated.

Input2: Signal 2 to be calculated.

Output1: Incremental Value of input signal 1.

Output2: Incremental Value of input signal 2.

***Parameters Setting*:**

There are no parameters setting for this block.

2.4.11 Adaptive Zone Block

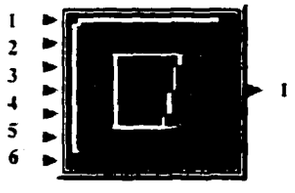


Figure 2.17 Interface of Adaptive Zone Block

Function and Principle:

The function of the Adaptive Zone block shown in Figure 2.17 is to adapt the protection zone to obtain a better setting for the present condition of the protected system during a fault. In this situation, the relay can operate faster and be attuned to the fault direction. The algorithm flowchart of the Adaptive Zone block is depicted by Figure 2.18.

To implement the adaptive function, horizontal and vertical expansion of the trip zone will be performed separately based on their own embedded logic conditions during a fault [9]. If the conditions for horizontal expansion are satisfied, the trip zone is expanded in both directions along the R axis shown in Figure 2.19. The amount of

expansion can also be controlled. Likewise, if the conditions for vertical expansion are met, the trip zone will be expanded vertically. The fault direction will determine the mode of the vertical expansion. Figure 2.20 illustrates the typical vertical expansions under both forward and reverse fault.

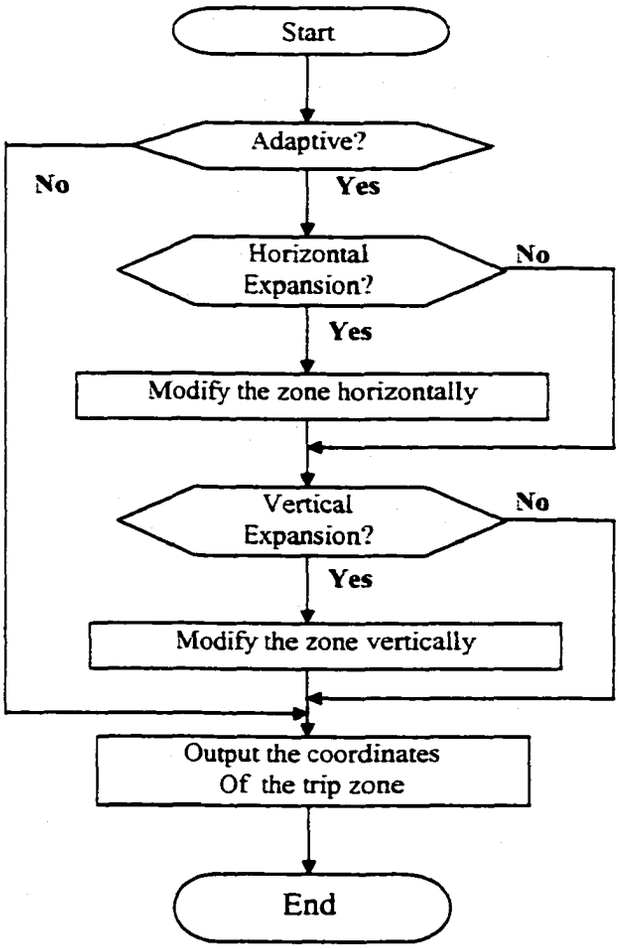
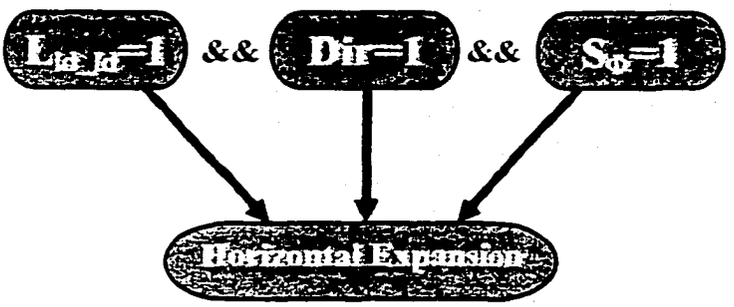
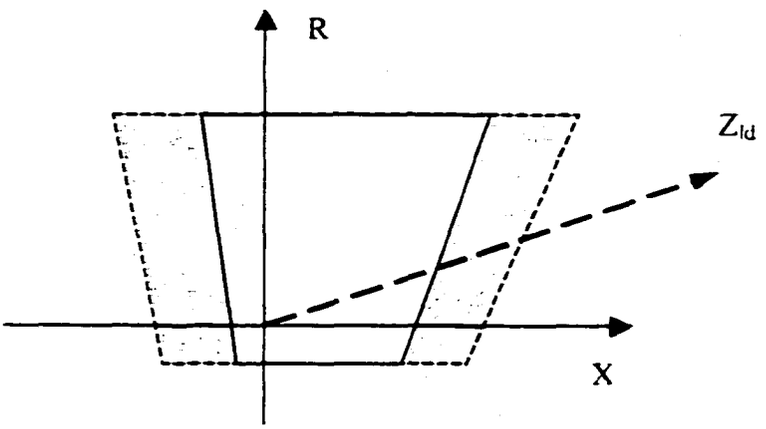


Figure 2.18 Flowchart of the Adaptive Function



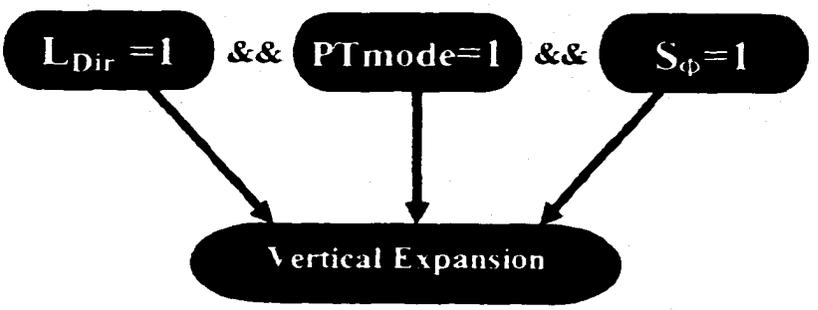
(a) Logic Selection for Horizontal Expansion



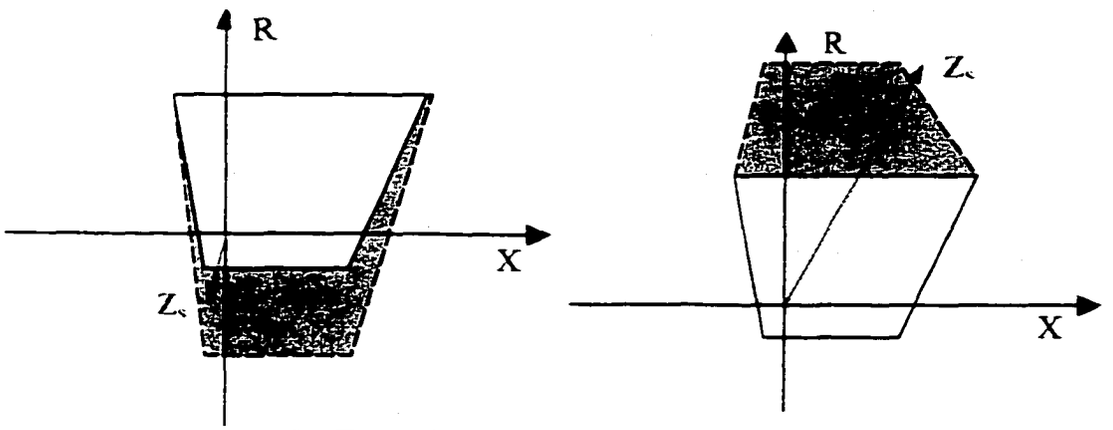
(b) Typical Horizontal Expansion of Trip Zone

Figure 2.19 Horizontal Expansion of Adaptive Function

When there is a forward fault, the relay will expand its protection in both the horizontal and vertical direction. The benefit of this is that the near zero impedance of the close-up forward fault is well inside the trip zone as opposed to being on the boundary of the prefault zone. However, during a reverse fault, only the vertical expansion will be employed. The benefit is that the near zero impedance of the close-up reverse fault is well outside the trip zone.



(a) Logic Selection for Vertical Expansion



(b) Typical Vertical Expansion of Trip Zone
Left: Forward Fault Condition; Right: Reverse Fault Condition

Figure 2.20 Vertical Expansion of Adaptive Function

The expansion of trip zones has several advantages. Firstly, it provides a larger trip zone for the forward fault which enhances its identifying ability during a forward fault and reduces its misoperation opportunities during a reverse fault. Secondly, it offers the possibility of quicker location of the fault, since the measured impedance can more quickly enter into the expanded trip zone than that of the prefault trip zone.

Inputs and Outputs:

Input1 — Zld_Re, the resistance of load impedance

Input2—Zdir_Im, the reactance of incremental positive impedance

Input3—Dir (output of DE element)

Input4 —L_{Dir} (Output3 of DI element)

Input5 —L_{Id_Id} (Output3 of LI element)

Input6 —S_{Φ} (Φ is a, b, c, ab, bc or ca)

Input7 —PTmode, which is used to disable or enable the vertical modification of the protection zone and its value. It is determined by the setting of the relay.

Output1 —coordinates of expanded trip zone: X_0, Y_0, X_1, Y_1, X_2, Y_2, X_3, Y_3

Parameter Settings: (Shown in Figure 2.21)

1. The protected transmission line impedance, PT &CT ratio and relay reach can be set in the 'system setting' group.
2. There are two relay types available: A PT_relay or a mho_relay. If 'mho_relay' is chosen, set its offset in the 'setting for mho relay' group. If 'PT_relay' is selected, its corresponding offsets need to be set in the 'setting for PT relay' group. These parameters will determine the prefault zone according to the protected line
3. If the 'adaptive' item is selected, the adaptive feature will be employed. Otherwise, the pre-set values (coordinates of four points which define the zone) will be fixed.

4. If the adaptive function is selected, set the preferable value of parameter "Adaptive K" The "Adaptive K" will determine how much the zone will be expanded.

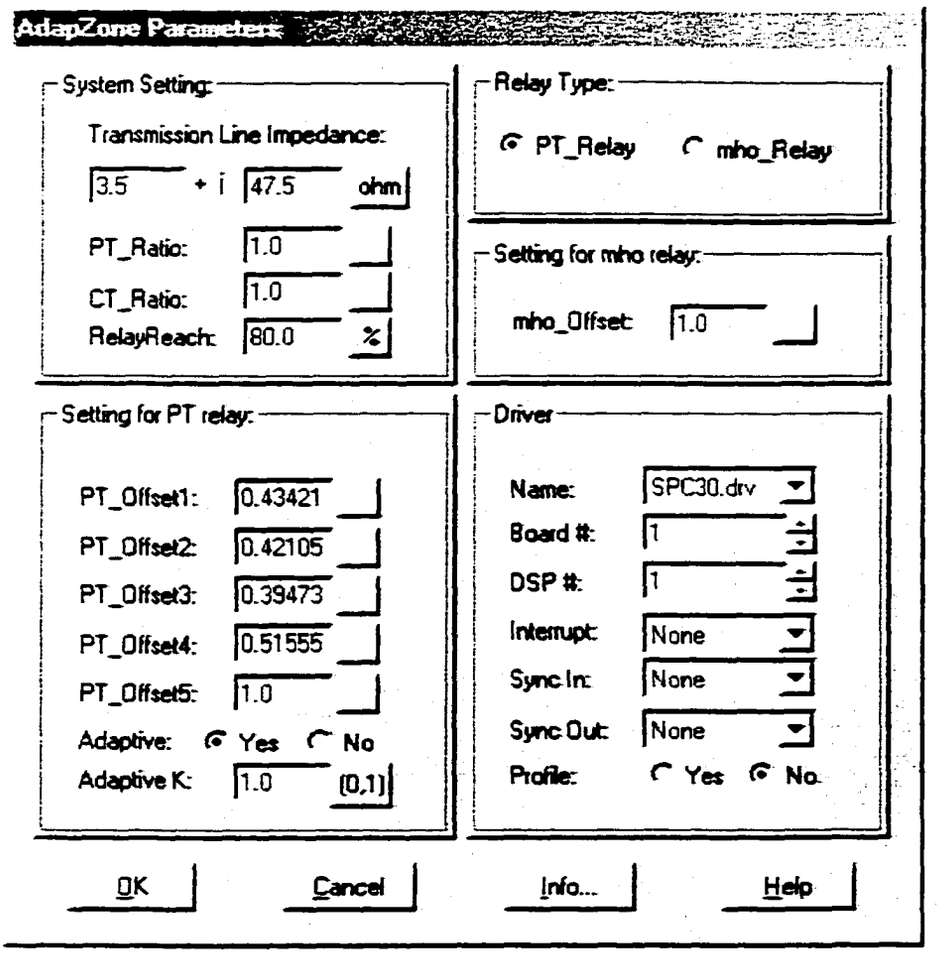
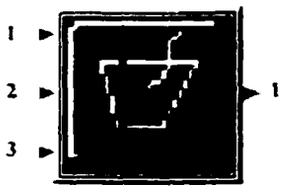
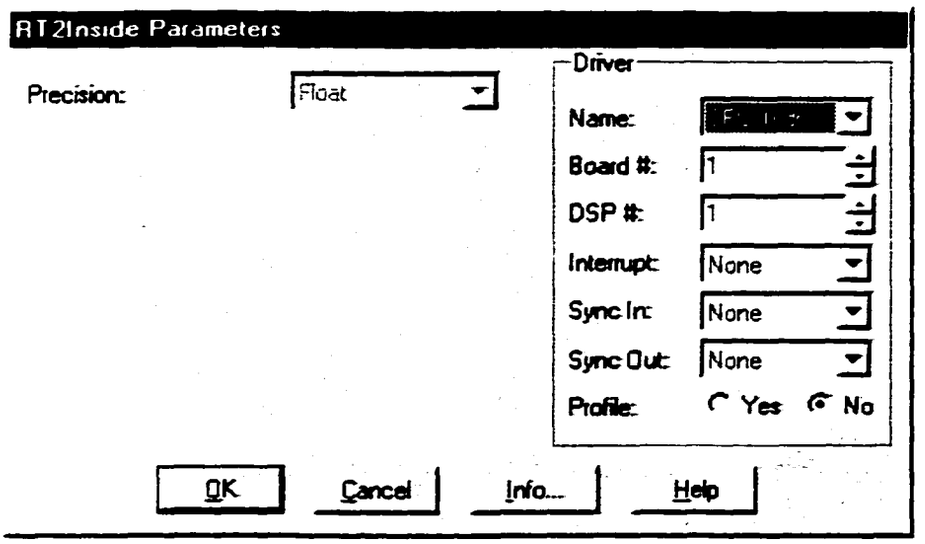


Figure 2.21 Interface for Parameter Settings of Adaptive Zone Block

2.4.12 TripZone Block



(a) Block Interface



(b) Parameter Settings Interface

Figure 2.22 Interface of TripZone Block

Function:

The TripZone block as shown in Figure 2.22 will determine whether the impedance measured by the Phase or Ground Distance Elements is within the adapted protection zone. If the measured impedance falls within the trip zone, this block will output a logic 1. Otherwise, a logic 0 will be output.

Input and output:

Input1: The coordinates of the modified protection zone.

Input2: The resistance of the measured impedance.

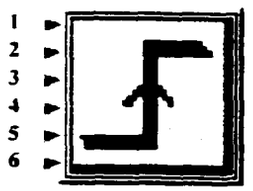
Input3: The reactance of the measured impedance.

Output: The logic trip signal, either a 0 or 1.

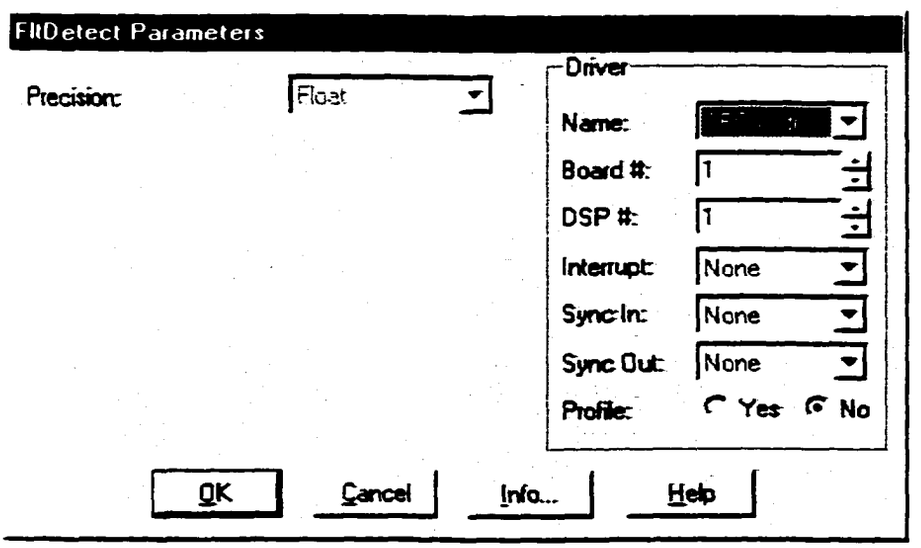
Setting of Parameters:

There is no parameter setting for this block

2.4.13 Fault Detection Block



(a) Block Interface



(b) Parameter Settings Interface

Figure 2.23 Interface of Fault Detection Block

***Function*:**

The 'Fault Detection' block as shown in Figure 2.23 will decide whether or not to generate a trip signal based on the outputs from both the TripZone blocks and the Phase Selection element. If the fault conditions are detected, the Fault Detection block will send a logic high signal to serial ports which will be used as the trip of the breaker. It can also be used for relay algorithm testing such as whether the relay algorithm can find the fault, how long it will take the algorithm to locate the fault and so forth.

***Inputs and Outputs*:**

Input1: Logic AND of TripZone Output with Phase Selection Output1 (Relay A).

Input2: Logic AND of TripZone Output with Phase Selection Output2 (Relay B).

Input3: Logic AND of TripZone Output with Phase Selection Output3 (Relay C).

Input4: Logic AND of TripZone Output with Phase Selection Output4 (Relay AB).

Input5: Logic AND of TripZone Output with Phase Selection Output5 (Relay BC).

Input6: Logic AND of TripZone Output with Phase Selection Output6 (Relay CA).

***Parameters Setting*:**

There is no parameter setting for this block.

2.5 Implementation of the GDDR relay

Based on the developed graphical block library and the libraries provided by Hypersignal RIDE, the following steps are adopted to design a typical Graphical DSP-based Distance Relay (GDDR) for the transmissions line protection.

- Setup of the GDDR relay Block Diagram

Three steps were carried out to create the block diagram of the GDDR relay in the Hypersignal RIDE environment.

Step1: Opened a new worksheet in Hypersignal RIDE

Step2: Selected the function blocks according to the relay algorithm and placed them on this worksheet.

Step3: Connected the function blocks. The block's input/output terminals are the bridges for the connection. "Connecting" blocks is referred to as a process of establishing the Data Flow Relationship among blocks by joining up their inputs/outputs.

The developed block diagram of the GDDR relay is composed of the Block Core and Display/Control interface as shown in Figure 2.24 and Figure 2.25. The Block Core includes the function blocks employed in the GDDR relay and their connection relationships. The Display/Control interface consists of both display and user control sections. The display section shows the instantaneous values of three-phase voltage and current waveforms, fault direction selection as well as the six graphical protection zones of the impedance elements. The PT ratio, CT ratio and other pre-set parameters used for calculation can be set in the control section.

- Configuration

  The configuration of the GDDR relay includes:

  1. Set the parameters in AdapZone Block. I.E. input the Transmission Line protected impedance, and select the trip zone and other parameters as described in Section 2.4.11.
  2. Set Zone parameters in DE block as described in Section2.4.8
  3. Set parameters in DirImp and PS as illustrated in Section2.4.7 and Section2.4.8.
  4. Set sampling rate, framesize, Gain_Current and Gain_Voltage in A/D sampling block as illustrated in Section 2.4.1.

- Compilation and Execution of GDDR relay

  The GDDR relay can be executed in the Hypersignal RIDE environment after it is compiled by selecting the Compile Command of Hypersignal RIDE.

- Generation of stand-alone application

  After the relay algorithm is verified, the GDDR relay can be exported to a standard COFF file for use with an embedded DSP through Hypersignal RIDE's export capability and Hyperception Application Interface (HAPPI) product.

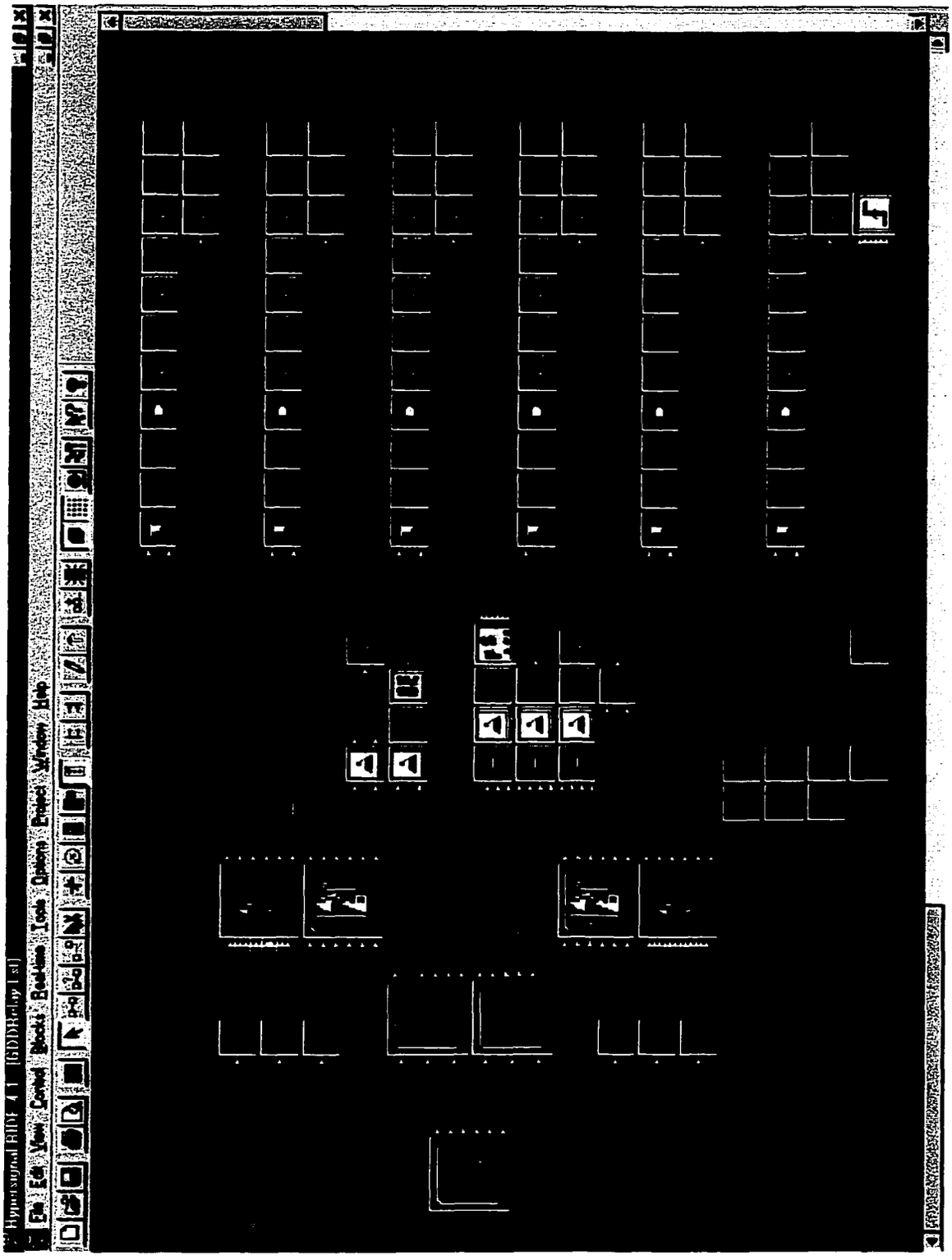


Figure 2.24 Block Core of the Developed GDDR relay Block Diagram

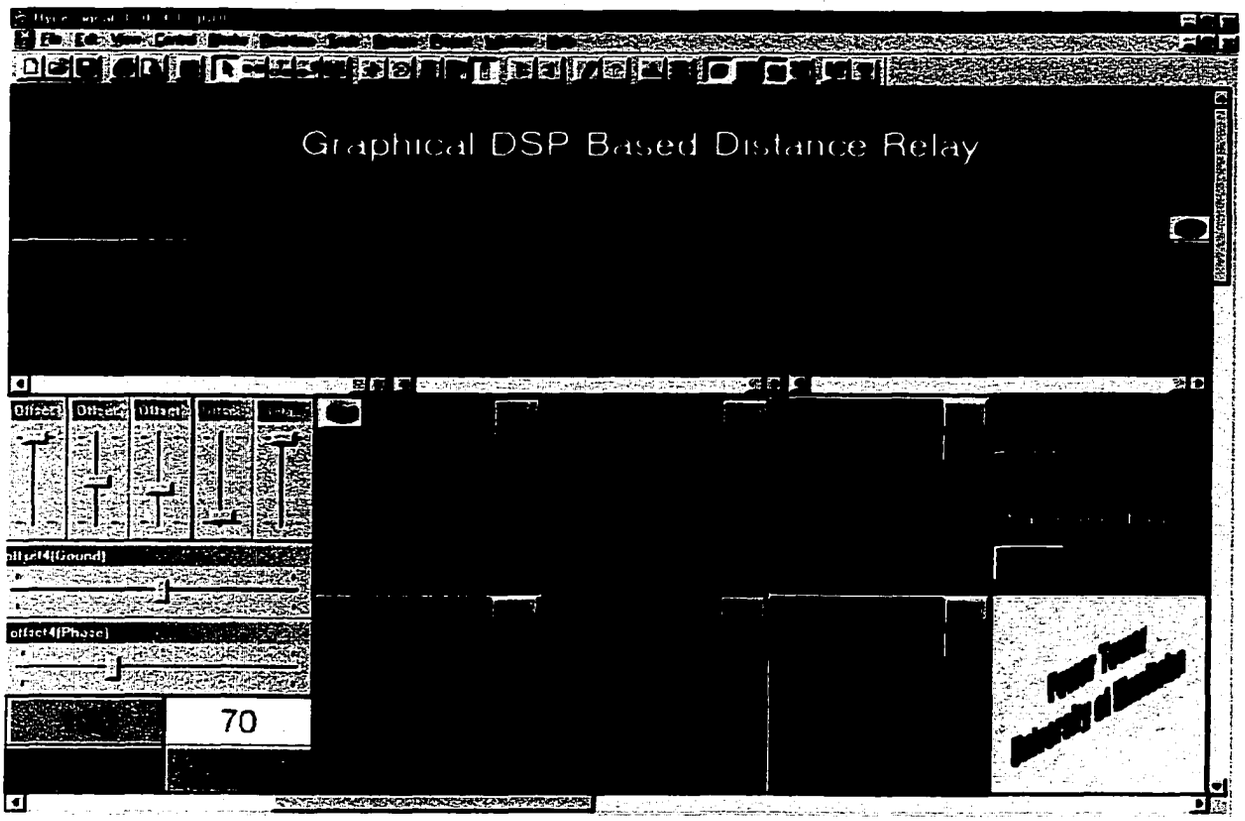


Figure 2.25 Display/Control Interface of the Developed GDDR relay

Chapter 3 Testing of Graphical DSP-based Distance Relay

3.1 Lab Set-up for GDDR Testing

To evaluate the performance of the developed GDDR relay, a lab set-up was displayed in Figure3.1. Its schematic diagram is shown in Figure 3.2.

Figure 3.1 Lab Set-up of GDDR Testing

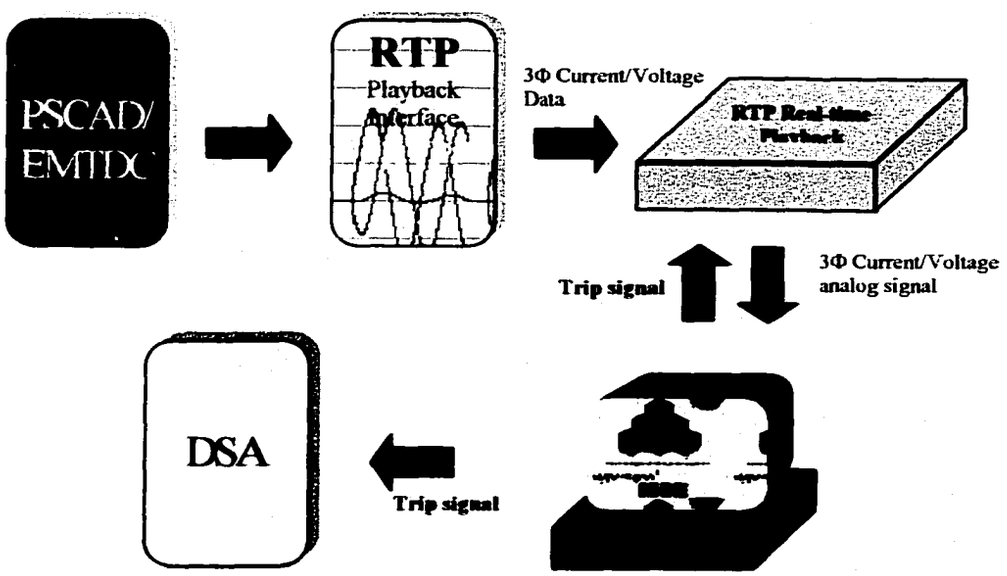


Figure 3.2 Schematic Diagram of Lab Set-up

3.1.1 PSCAD/EMTDC software package.

PSCAD/EMTDC V3.0 is a MS windows operating system based Application package developed by the Manitoba HVDC Research Center. It is a simulator for electric circuits used in low voltage power electronics systems, high voltage DC transmission (HVDC) and flexible AC transmission systems (FACTS), as well as distribution systems and complex controllers. In this work, it was used extensively to simulate a power system and generate the fault waveforms used by Playback (RTP).

3.1.2 Real time playback simulator

The Real Time Playback (RTP) developed by the Power group and Manitoba HVDC Research Centre is a computer based waveform playback system with 12 analog outputs and 16 logic inputs/outputs. Best described as an arbitrary waveform generator, the RTP can replay any waveform generated by PSCAD/EMTDC for testing protection, control or measurement system. In addition, the RTP can generate practical analog signals from on-line recorded data files or created by the RTP STATE program. The RTP has an advanced graphical interface for displaying and controlling waveforms and a Batch playback mode for automated testing. In this work, the RTP was employed to feed the analog signals into the developed GDDR relay.

3.1.3 Testing procedure

The procedure for testing the GDDR relay is described as follows:

1. A practical single transmission line system case is simulated in PSCAD/EMTDC. The waveforms generated by PSCAD/EMTDC under different operation conditions are stored in data files.

2. The data files obtained from PSCAD/EMTDC will be loaded by the RTP (Real-time Playback) software. The PC running RTP will output the simulated current and voltage signals to the developed GDDR relay. In addition, the logic signals set by the customer can be output at the same time.
3. The three phase voltage and current signals are then sampled and processed by the developed GDDR relay running in both the PC and DSP. The PC displays the instantaneous values of the three phase current and voltage signals, calculated apparent impedance and their locations in the impedance planes, while the DSP runs the relay algorithms and monitors the system status. If there is a fault in the system, the fault type will be determined and a logic signal representing the trip signal will be generated.
4. This generated logic output signal is sent to either the DSA (Digitizing Signal Analyzer) or the RTP to monitor the interval between the fault start instant and the trip instant. (i.e. operating time offered by the developed GDDR relay).

3.2 Simulation case and fault waveforms

A typical power system with one transmission line connecting two sources is simulated in PSCAD/EMTDC as shown in Figure 3.3.

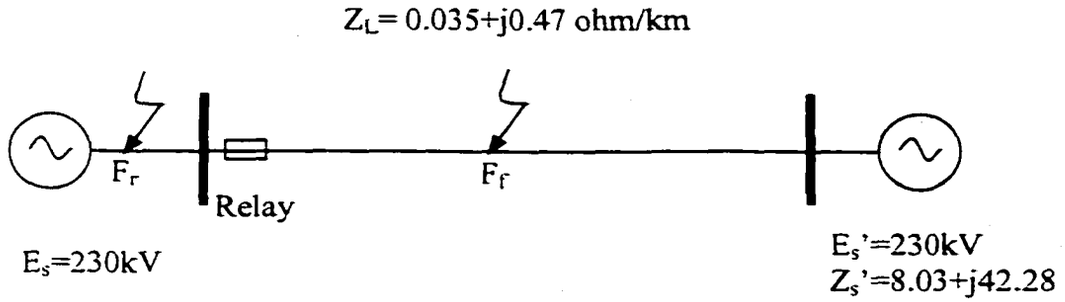
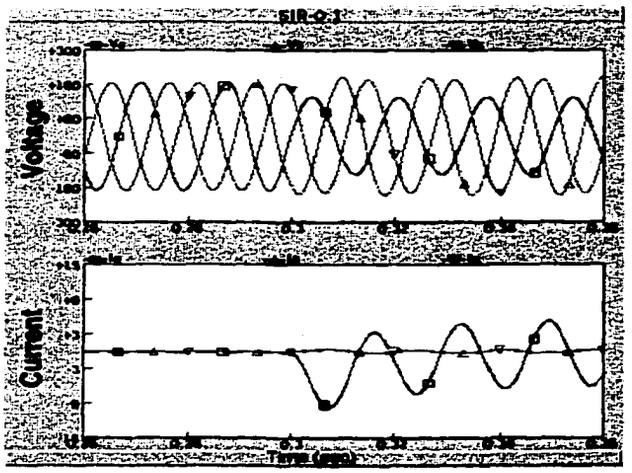
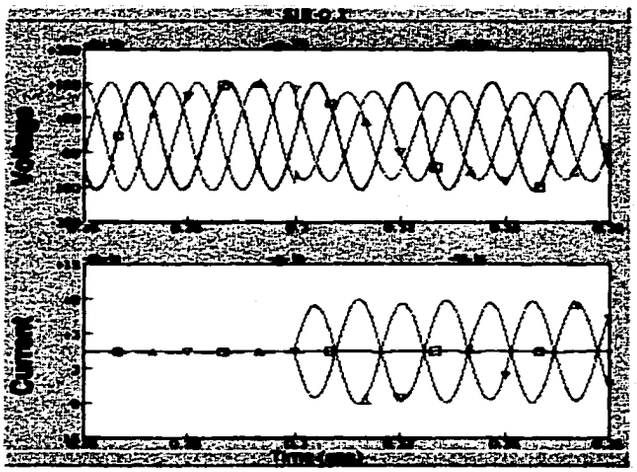


Figure 3.3 PSCAD/EMTDC Simulation System

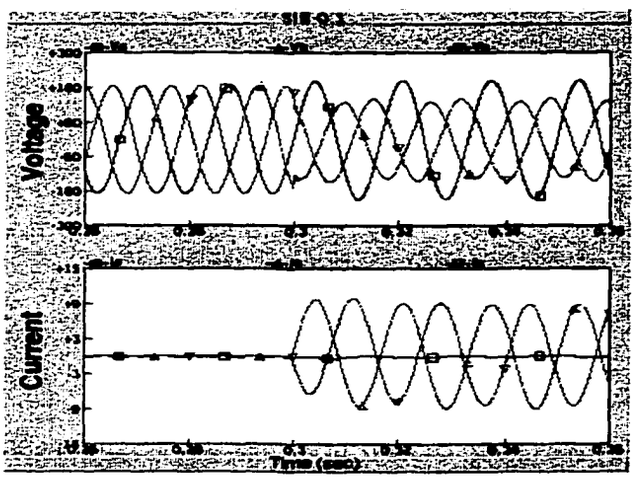
All typical fault types were simulated and run in PSCAD/EMTDC. Figures 3.4 (a), (b), (c) and (d) show the typical voltage and current waveforms under AG, BC, BCG and ABCG fault conditions.



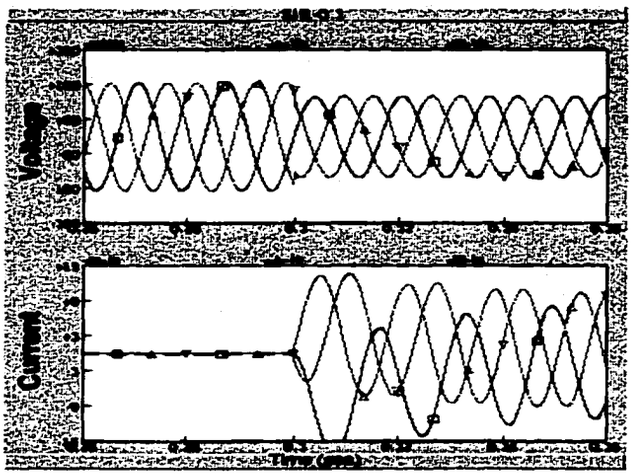
(a) AG fault



(b) BC fault



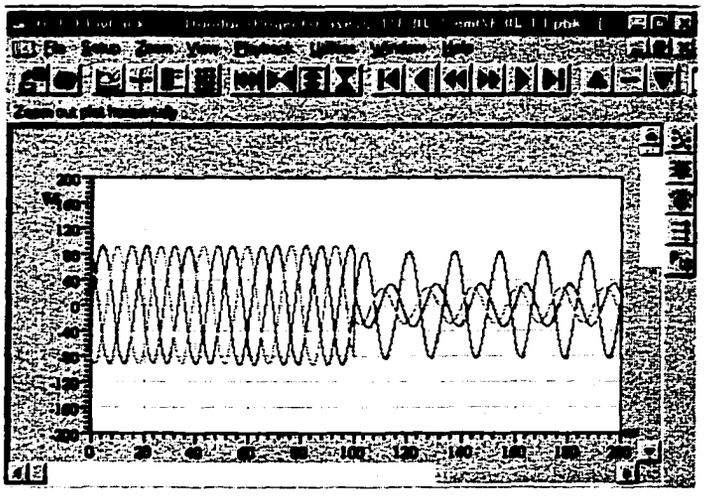
(c) BCG fault



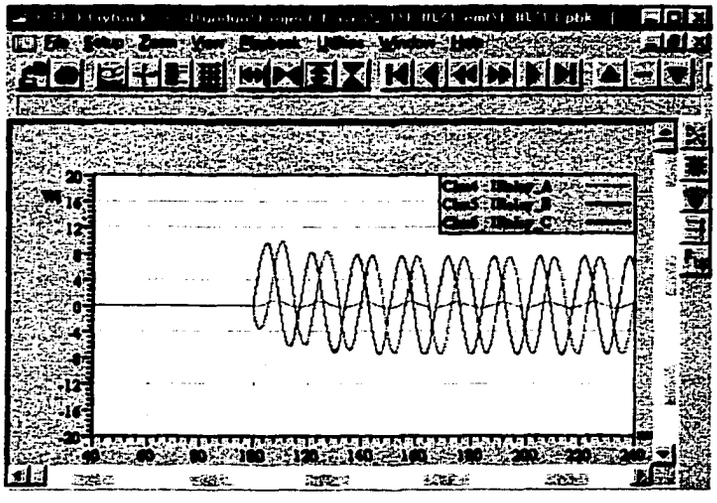
(d) ABCG fault

Figure 3.4 AG, BC, BCG, ABCG fault current and voltage waveforms

The RTP reads the data files generated from the PSCAD/EMTDC software and outputs the corresponding analog voltage and current signals. The typical interface of the RTP with waveform to be displayed is shown in Figure 3.5.



(a) 3Φ voltage waveform



(b) 3Φ current waveform

Figure 3.5 Real Time Playback interface

3.3 Testing Results

All the real-time and simulation blocks (section 2.3) developed for the GDDR relay have been tested individually in the Hypersignal RIDE environment using software methods. The following testing was performed to evaluate the GDDR relay as an integrated unit.

3.3.1 Fault Phase selection test

Different fault conditions were simulated in PSCAD/EMTDC, and the waveforms generated were applied to the GDDR using the RTP simulator. The comparison between applied faults and the fault types determined by the GDDR relay is shown in Table 3.1. These results show that the phase selection element is capable of picking up the faulted phases and can identify the fault type under all fault condition tested.

Table 3.1 Verification results of phase selection

Fault Applied	Logic Output of GDDR Relay Phase Selection Element					
	S_a	S_b	S_c	S_{ab}	S_{bc}	S_{ca}
AG	1	0	0	0	0	0
BG	0	1	0	0	0	0
CG	0	0	1	0	0	0
AB	0	0	0	1	0	0
BC	0	0	0	0	1	0
CA	0	0	0	0	0	1
ABG	1	1	0	1	0	0
BCG	0	1	1	0	1	0
CAG	1	0	1	0	0	1
ABCG	1	1	1	1	1	1

Note: S_a, S_b, S_c, S_{ab}, S_{bc}, S_{ca} (see section 2.3 for explanation)

3.3.2 Determination of fault direction test

Both forward faults (fault at location F_f in Figure 3.3) and reverse faults (fault at location F_r in Figure 3.3) were simulated and tested. The output of the DE (Direction Element) of the GDDR was programmed to monitor these conditions: a logic "1" for a forward fault, and a logic "0" for a reverse fault. In the test, the DE element will always give the proper logic output value as shown in Figure 3.6. This verifies that the GDDR relay can determine the fault direction accurately.

(a) Forward fault applied (b) Reverse fault applied

Figure 3.6 Test results of fault direction

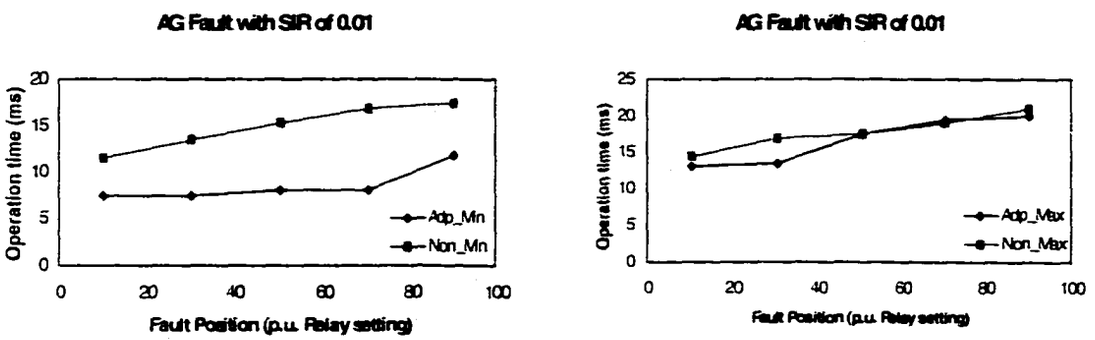
3.3.3 Effects of fault location, fault type, SIR (System impedance ratio) and adaptive relaying

Fault pick up time, (i.e. operating time of the relay) is a big concern for any relay system developed. In this work, typical fault types, fault locations as well as the SIR ratio are employed to simulate various power system conditions. All of the voltage and current waveforms under these conditions were applied to the developed GDDR relay to study their effects on the operation time. Since the execution time of the relay algorithm with real-time display is approximately 4ms, for faults which occur at the same point of wave, there is a time differential (min/max operating time) depending on the point within the

algorithm when the fault occurs. In addition, the effect of adaptive relaying on the operation time was investigated.

- **SIR=0.01**

Figures 3.7, 3.8, 3.9 and 3.10 illustrate the relationship between operation time and fault position (p.u. relay setting) with a SIR (System impedance ratio) of 0.01. In Figure 3.7 (a), it can be seen that minimum operating time becomes longer when the distance to the fault location increases and the adaptive function is not active; However, with the adaptive function enabled, the minimum operating time is less sensitive to the change of fault locations. Also, the operation time with the adaptive function enabled is less than that without the adaptive function and the time difference ranges from 4 to 8.75 ms. Similar results are illustrated in figures 3.8 (a), 3.9(a), 3.10 (a) which represent the operating time for BCG, BC, ABCG faults respectively.

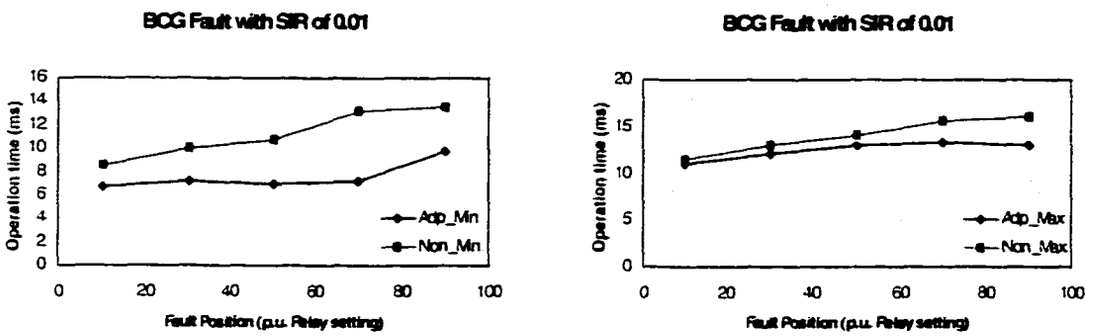


(a) Minimum Operating time

(b) Maximum Operating time

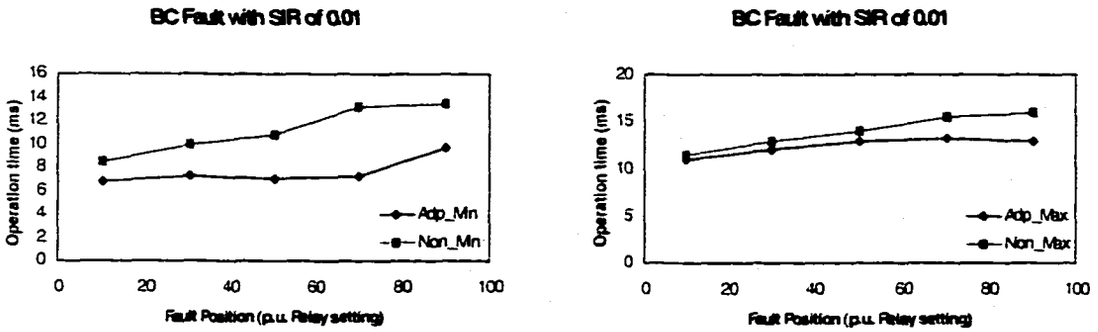
Figure 3.7 Operating time vs. Fault location of AG fault (SIR = 0.01)

Figures 3.7(b), 3.8(b), 3.9(b) and 3.10 (b) show the maximum operating times vs. fault location with adaptive function enabled or disabled under AG, BC, BCG and ABCG fault conditions. A slightly different result is illustrated in these results and the time difference with or without the adaptive function is less than that of the minimum operating time. However, the adaptive function still shows the effect of decreasing the operating time under most circumstances.



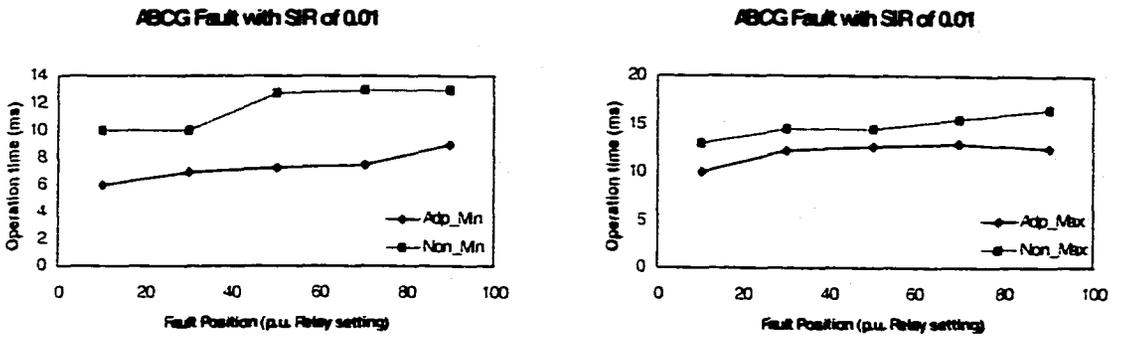
(a) Minimum Operating time (b) Maximum Operating time

Figure 3.8 Operating time vs. Fault location of BCG fault (SIR=0.01)



(a) Minimum Operating time (b) Maximum Operating time

Figure 3.9 Operating time vs. Fault location of BC fault (SIR=0.01)



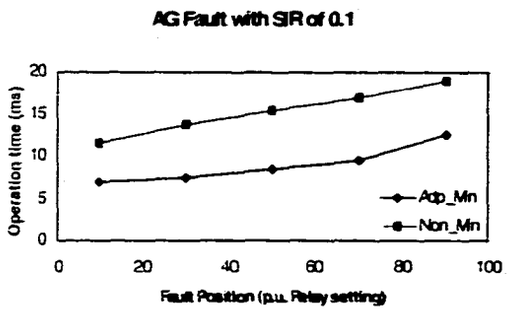
(a) Minimum Operating time

(b) Maximum Operating time

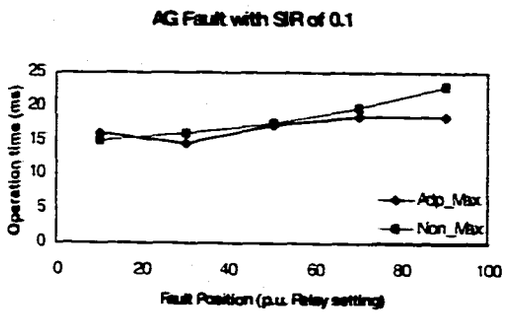
Figure 3.10 Operating time vs. Fault location of ABCG fault (SIR=0.01)

- **SIR=0.1**

The results for operating time versus fault location with a SIR of 0.1 are shown in figures 3.11-3.14. As before, the curves representing the minimum operation time shows that the adaptive function can reduce the operation time effectively, and the fault location has less effect on operation time with adaptive function active than that without it. According to the fault type, the curve tendency differs slightly from the others. The differences of maximum operating time between adaptive and non adaptive function is smaller compared with that of the minimum operating time.

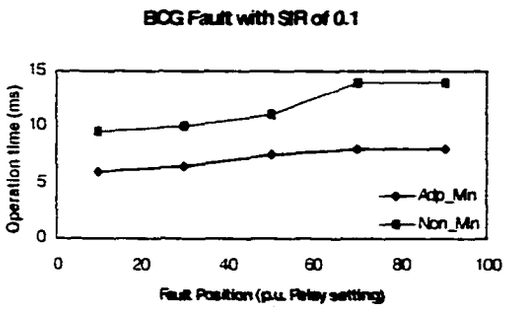


(a) Minimum Operating time

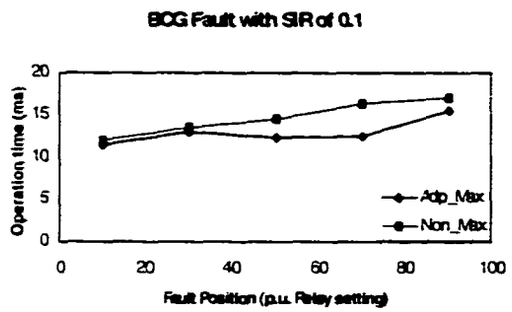


(b) Maximum Operating time

Figure 3.11 Operating time vs. Fault location of AG fault (SIR=0.1)

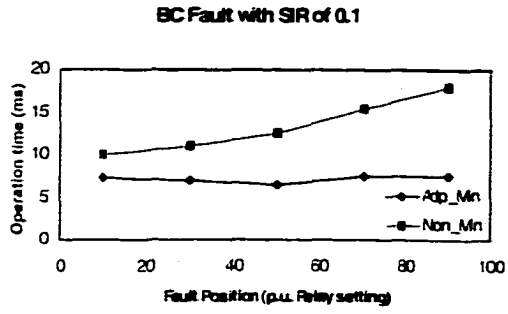


(a) Minimum Operating time

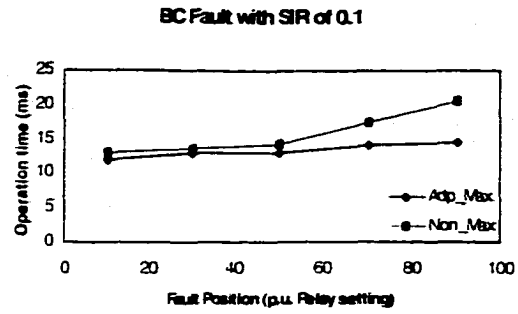


(b) Maximum Operating time

Figure 3.12 Operating time vs. Fault location of BCG fault (SIR=0.1)

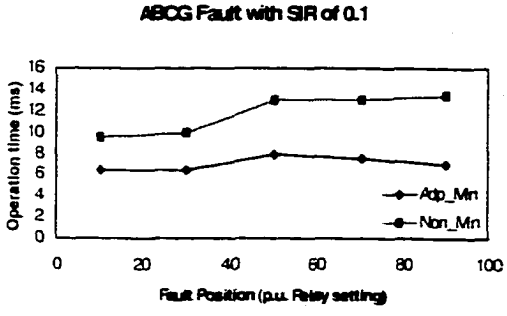


(a) Minimum Operating time

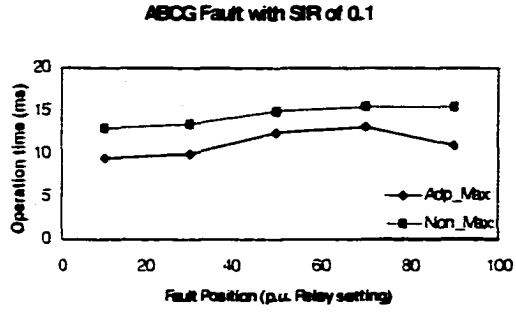


(b) Maximum Operating time

Figure 3.13 Operating time vs. Fault location of BC fault (SIR=0.1)



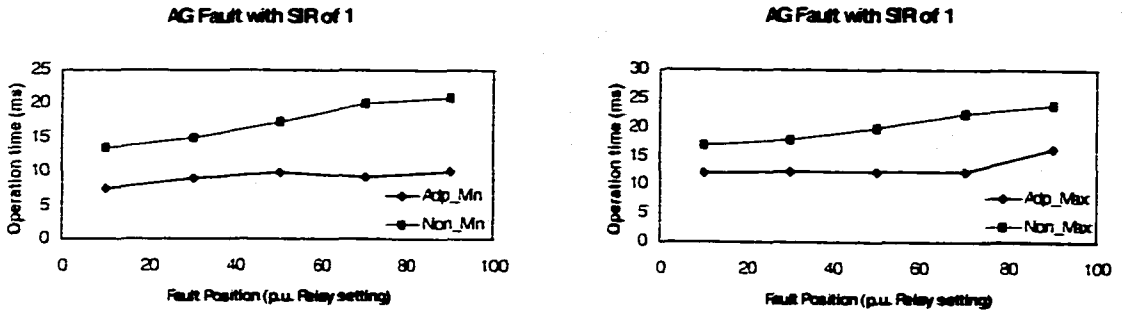
(a) Minimum Operating time



(b) Maximum Operating time

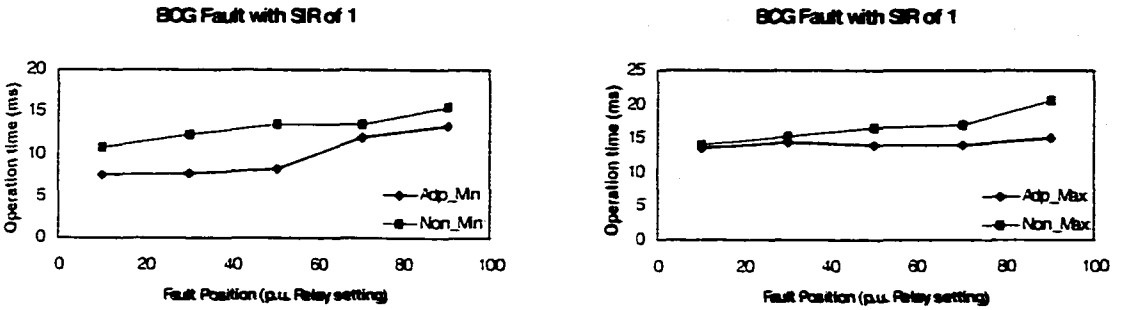
Figure 3.14 Operating time vs. Fault location of ABCG fault (SIR=0.1)

- **SIR=1**



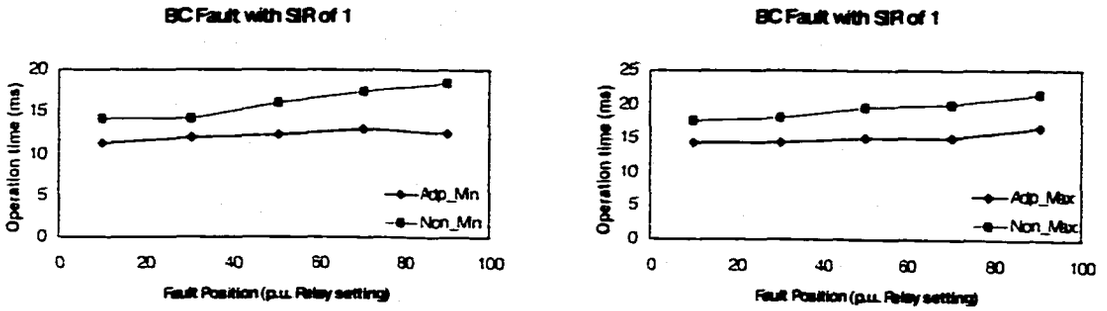
(a) Minimum Operating time (b) Maximum Operating time

Figure 3.15 Operating time vs. Fault location of AG fault (SIR=1)



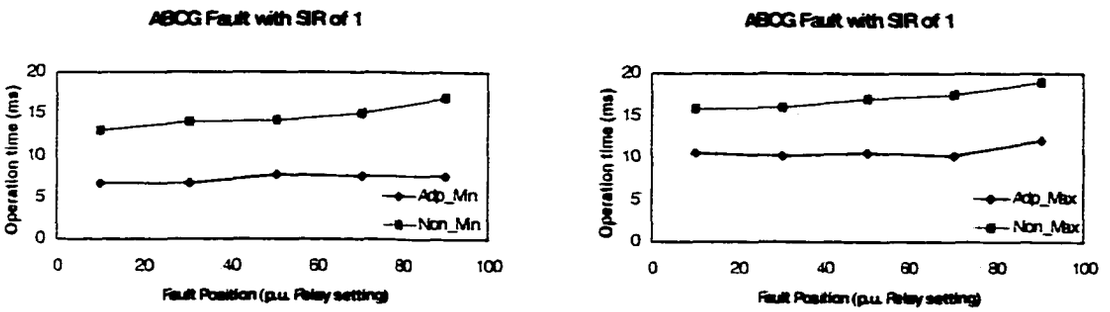
(a) Minimum Operating time (b) Maximum Operating time

Figure 3.16 Operating time vs. Fault location of BCG fault (SIR=1)



(a) Minimum Operating time (b) Maximum Operating time

Figure 3.17 Operating time vs. Fault location of BC fault (SIR=1)

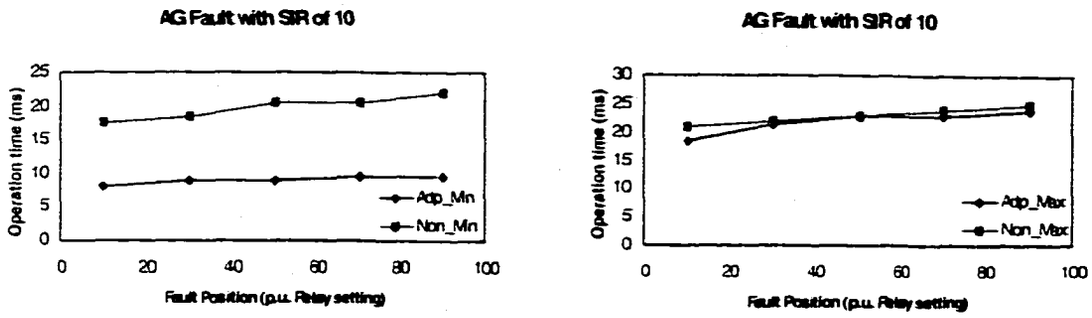


(a) Minimum Operating time (b) Maximum Operating time

Figure 3.18 Operating time vs. Fault location of ABCG fault (SIR=1)

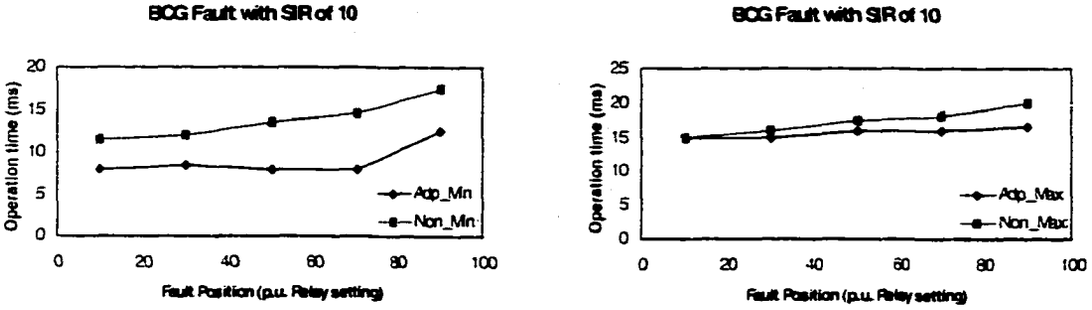
Figures 3.15, 3.16, 3.17 and 3.18 display the results of the relationship between operating time and fault location with a SIR of 1. When adaptive function is enabled, both the minimum and maximum operating times increase with the fault location. If adaptive function is enabled, the operating time became less sensitive to the fault location and is much faster than that without using the adaptive function. Apparent gaps of 4 to 8 ms were observed for both maximum and minimum operating times.

- SIR=10



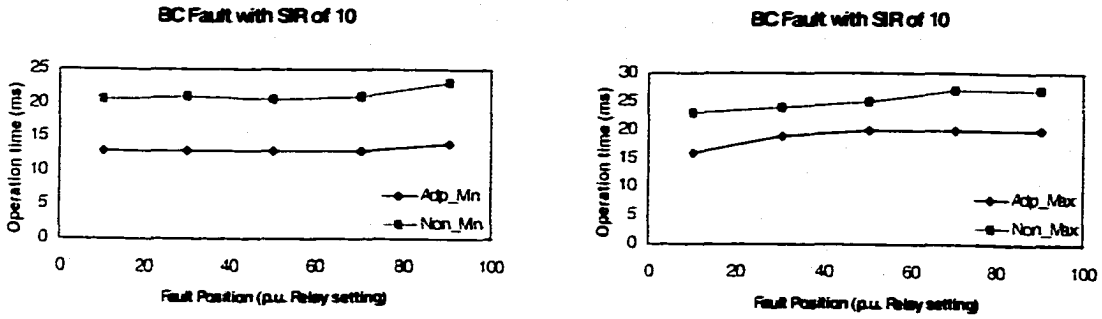
(a) Minimum Operating time (b) Maximum Operating time

Figure 3.19 Operating time vs. Fault location of AG fault (SIR=10)



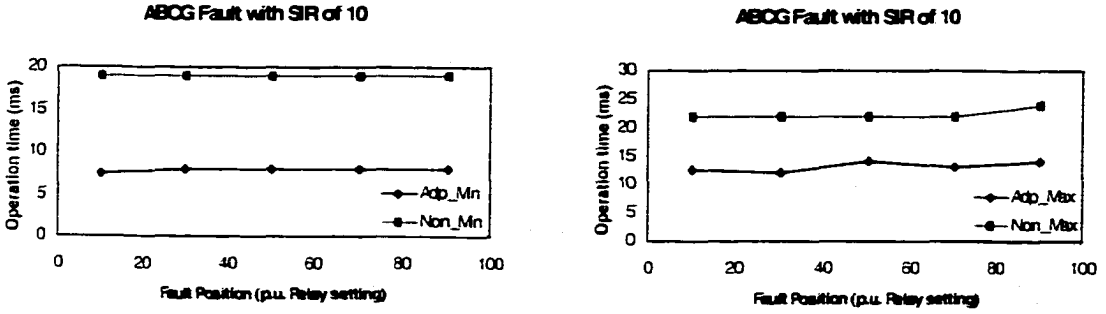
(a) Minimum Operating time (b) Maximum Operating time

Figure 3.20 Operating time vs. Fault location of BCG fault (SIR=10)



(a) Minimum Operating time (b) Maximum Operating time

Figure 3.21 Operating time vs. Fault location of BC fault (SIR=10)



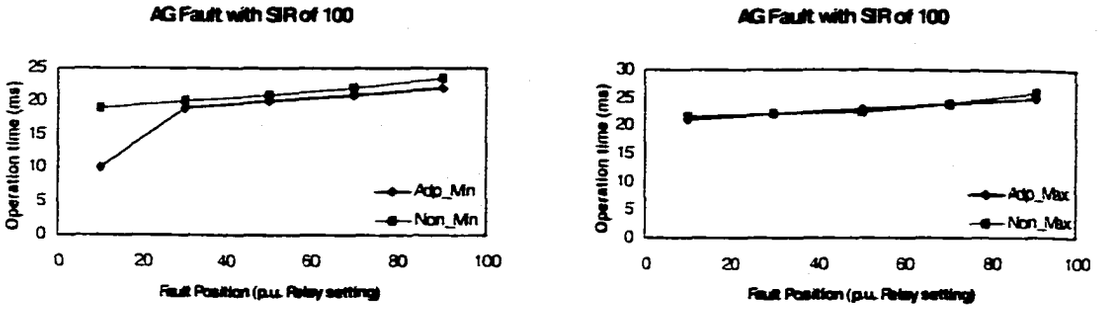
(a) Minimum Operating time (b) Maximum Operating time

Figure 3.22 Operating time vs. Fault location of ABCG fault (SIR=10)

Results shown in figures 3.19, 3.20, 3.21, 3.22 represent the relationship for the operating time versus fault location with a SIR of 10. For AG and BCG faults, similar results were found. The minimum operating time without the adaptive function is more sensitive to the fault location and longer than that with the adaptive function. The maximum operating time for both adaptive and non-adaptive functions is quite close even though the former is still shorter than the latter. For BC and ABCG faults, a different results are displayed. Both the minimum and maximum operating times without the

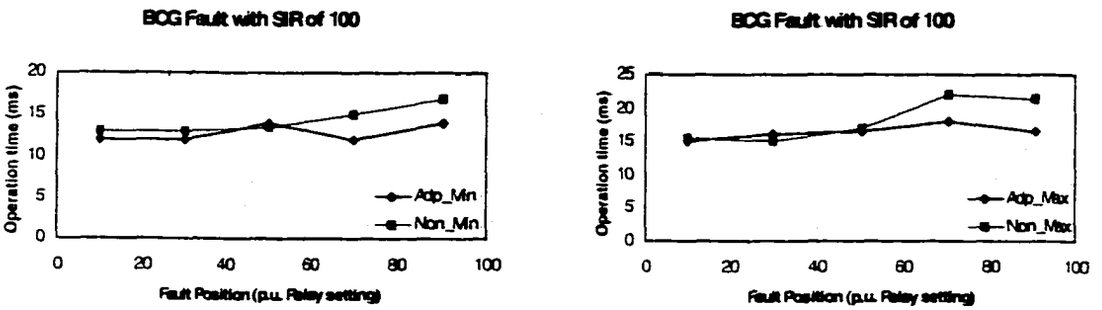
adaptive function became less sensitive to the fault location. However, the operating time is still longer than that with the adaptive function.

- SIR=100



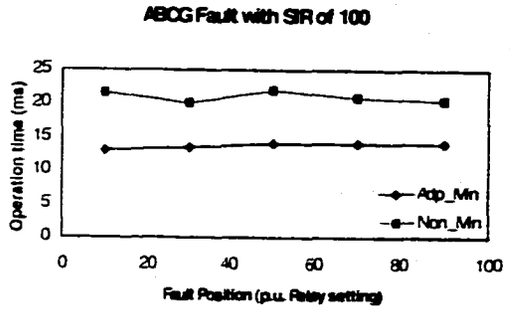
(a) Minimum Operating time (b) Maximum Operating time

Figure 3.23 Operating time vs. Fault location of AG fault (SIR=100)

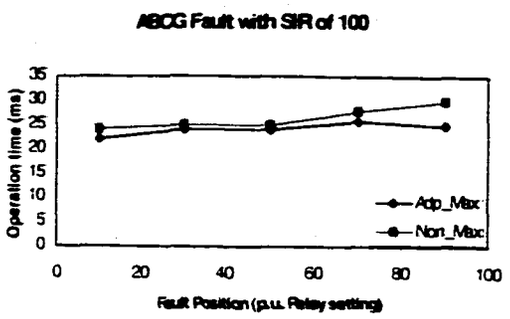


(a) Minimum Operating time (b) Maximum Operating time

Figure 3.24 Operating time vs. Fault location of BCG fault (SIR=100)

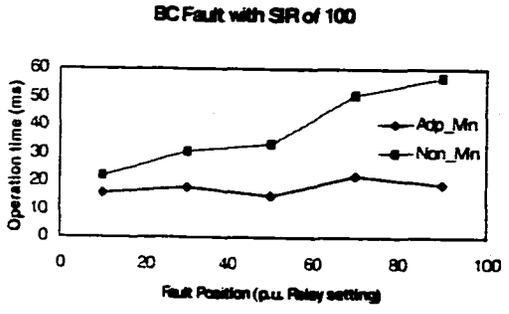


(a) Minimum Operating time

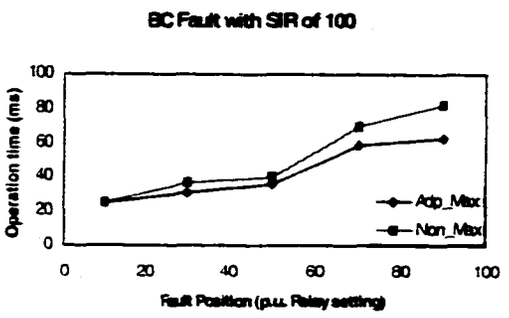


(b) Maximum Operating time

Figure 3.26 Operating time vs. Fault location of ABCG fault (SIR=100)



(a) Minimum Operating time



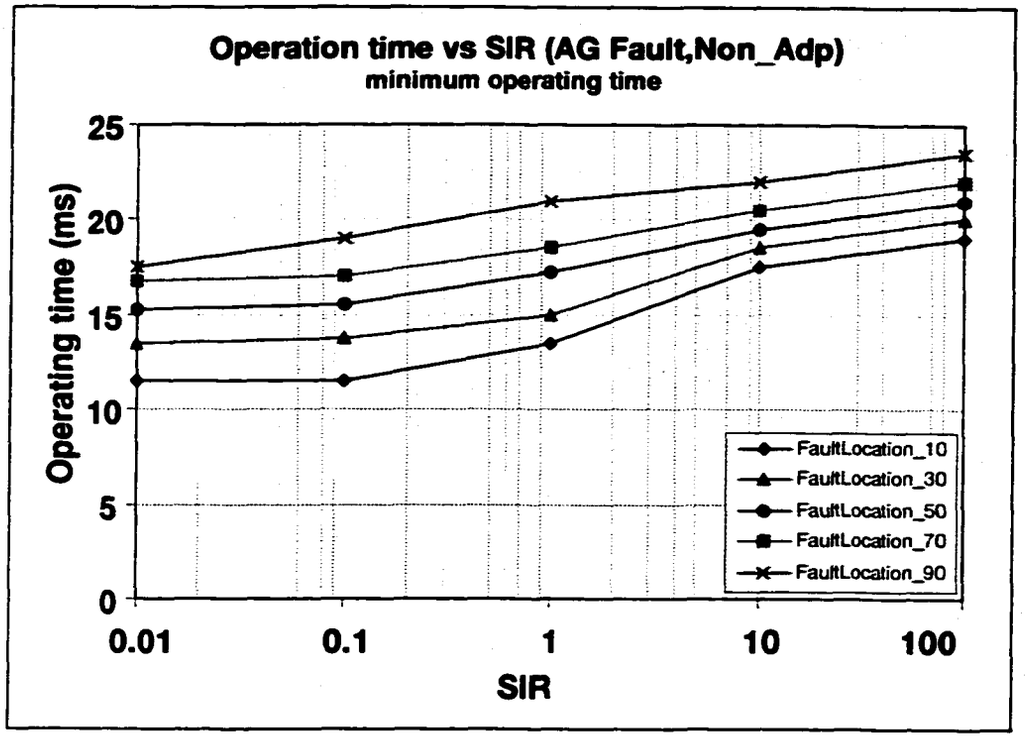
(b) Maximum Operating time

Figure 3.25 Operating time vs. Fault location of BC fault (SIR=100)

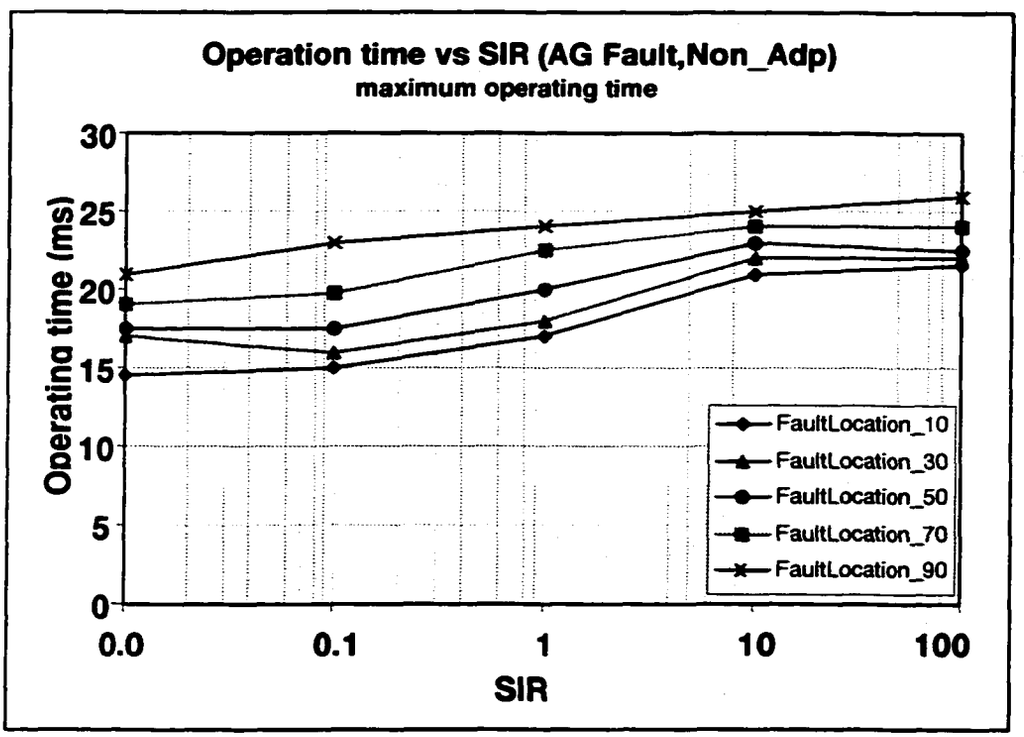
With the SIR increased to 100, the adaptive function is no longer effective for AG and BCG fault as shown in figures 3.23 and 3.24. The two curves are very close for both the minimum and maximum operating time. The adaptive function is, however, still valid for BC and ABCG faults as shown in figure 3.25 and 3.26. Further investigation is needed to explain the above results.

- Effect of SIR

The System Impedance Ratio (SIR), the ratio of source impedance behind the distance relay to the line impedance the relay is protecting, could be high resulting from a low line impedance (short lines), weak source behind the line, or a combination of both. A higher SIR results in lower voltages and currents during faults if the higher ratio was due to a higher source impedance rather than lower line impedance at the relay location. The effects of SIR are summarized in a set of curves shown in Figure 3.27. These curves describe the relationship between operating time and SIR with the fault location ranging from 10% to 90% of the relay setting. Minimum and maximum operating times versus SIR for an AG fault are drawn in (a) and (b) separately. For the same fault location, it can be seen that the operating time increases with the increase of SIR under most circumstances when the adaptive function is not activated. That is, a higher SIR slows the operating speed of the relay. Slower response could make distance relay application unacceptable when a fast speed relay is required by a transmission system, such as an EHV transmission system.



(a) Minimum Operating time



(b) Maximum Operating time

Figure 3.27 Operating time vs. SIR (system impedance ratio)

3.4 Summary

Based on the above results, a summation of the effect of fault location, SIR and adaptive relaying can be outlined as following:

1. If the adaptive function is not activated, the operating time for the GDDR relay to detect a fault increases with the distance to the fault location, whereas, with the adaptive function active, the operating time becomes less affected by the fault location.
2. The System Impedance Ratio (SIR) affects the operating time of the GDDR relay when the adaptive function is disabled. A higher SIR leads to a slower response time.
3. The adaptive relay has the advantage of shortening the operating time to locate the fault. It is able to adapt to fault direction, type and fault location.

Chapter 4 Conclusions

In this thesis, the development of a Graphical DSP-based Distance Relay (GDDR) is presented.

Chapter 1 briefly introduces the concept of protective relaying, especially the digital computer relays and adaptive relaying. In Chapter 2, the developed graphical block library of basic functions for distance relaying is described in detail. In addition, a Graphical DSP-based Distance Relay (GDDR) was implemented based on the developed block library. Laboratory tests were carried out to evaluate system performance, and find the effects of System Impedance Ratio (SIR), fault types, and fault location in Chapter 3. The impact of the adaptive features on fault pick-up time is also discussed.

The following is a summary of the conclusion:

1. The developed low-level hardware window's driver for TMS320C30 DSP board can effectively support the communication between a DSP and a host PC.
2. A graphical block diagram library of the basic functions for a distance relay was successfully developed using commercial software: Visual C++, TI compiler and Hypersignal RIDE.
3. A Graphical DSP-based Distance Relay (GDDR) for transmission line protection was implemented based on the developed block diagram library.
4. The lab tests using PSCAD/EMTDC and the RTP verify that the developed GDDR relay can perform promptly under fault conditions.

5. The effects of fault location, fault type and the System Impedance Ratio (SIR) on the tripping time using the developed GDDR relay were investigated.
6. The developed GDDR relay with adaptive relaying features can effectively reduce the operation time in many fault conditions.
7. The relay engineers can configure a generic DSP-based relay hardware as a distance relay with the developed library and GUI.

Future work

Some work is still required to extend the development of Graphical User Interface in future. Additional blocks representing other relay features such as differential blocks, harmonic restraint, fault location calculation etc. could be added to the developed library. An interface for alternative DSP's could also be developed.

References

[1] The Electricity Council, "Power System Protection 1", Peter Peregrinus Ltd, 1981.

[2] Protection Relays Application Guide, Third Edition, ALSTOM, 1987.

[3] IEEE Guide for Protective Relay Applications to Transmission Lines, 1999.

[4] T. S. Madhava Rao, "Power System Protection Static Relays", Tata McGraw-Hill Publishing Company Limited, 1981.

[5] Arun G. Phadke, James S. Thorp, "Computer Relaying for Power System", Research Studies Press Ltd, 1988.

[6] S. H. Horowitz, A. G. Phadke, "Adaptive Transmission System Relaying", IEEE Transactions on Power Delivery, Vol. 3, No. 4, October 1988.

[7] A. K. Jampala, S. S. Venkata, "Adaptive Transmission Protection: concepts and Computational Issues", IEEE Transactions on Power Delivery, Vol. 4, No. 1, January 1989.

[8] P.G. McLaren, G.W. Swift, "Open system relaying", IEEE Transaction on Power Delivery, Vol. 9, No. 3, July 1994.

[9] H. Li, "A New Adaptive Distance Relay", Ph.D thesis, University of Manitoba, 1999.

[10] Charles L. Phillips, H. Troy Nagle, "Digital Control System Analysis and Design", Prentice-Hall, 1984.

[11] Miles A. Redfern, "Signal Processing Technique in Distance Protection Applications", Ph.D thesis, University of Cambridge, 1976.

[12] P.G. McLaren, G.W. Swift, Z. Zhang, E. Dirks, " A New Directional Element for Numerical Distance Relays", IEEE Transactions on Power Delivery, Vol 10, No. 2, April 1995.

[13] Ioni T. Fernando, " Protection of Transmission Lines Sharing The Same Right-of-Way", Ph.D thesis, University of Manitoba, 1997.

[14] Hypersignal RIDE User's Manual Version 4.1 for windows 95/98, Hypersignal, 1998.

[15] RIDE Function Reference, Hypersignal, 1998.

[16] TMS320C3X User's Guide, Texas Instruments, 1991.

[17] TMS320C3X/4X Assembly Language Tools: User's Guide, Texas Instruments, 1991.

Appendix A: Tools used in this thesis

For the development of theGDDR relay, different commercial software and hardware have to be employed together. In this work, Visual C++, Hypersignal RIDE, DSP and its code compiler are extensively used. The features of some tools are introduced in the following.

Hypersignal RIDE software package

Hypersignal RIDE is a complete visual design environment for real-time systems development [14-15]. This tool can be used for a variety of exciting applications which range from low-level DSP systems design & implementation to application specific projects such as real-time instrumentation, data acquisition, control systems, and more. Hypersignal RIDE has the following features:

- It is a superset of the Hypersignal Block Diagram visual environment which adds support for the design, implementation and analysis of real-time DSP algorithms and systems. Figure 1 shows a typical interface for Hypersignal RIDE.
- Both real-time and simulation functions are available for graphically designing custom DSP applications, and the custom blocks can be built, programmed and added to the block library.
- The user interface is same for both simualted and real-time DSP block functions. Both real-time (work done by DSP board) and simulated (work done by PC) executables function within the same design, which allows for convenient conversion between the PC simulations and real-time implementations.

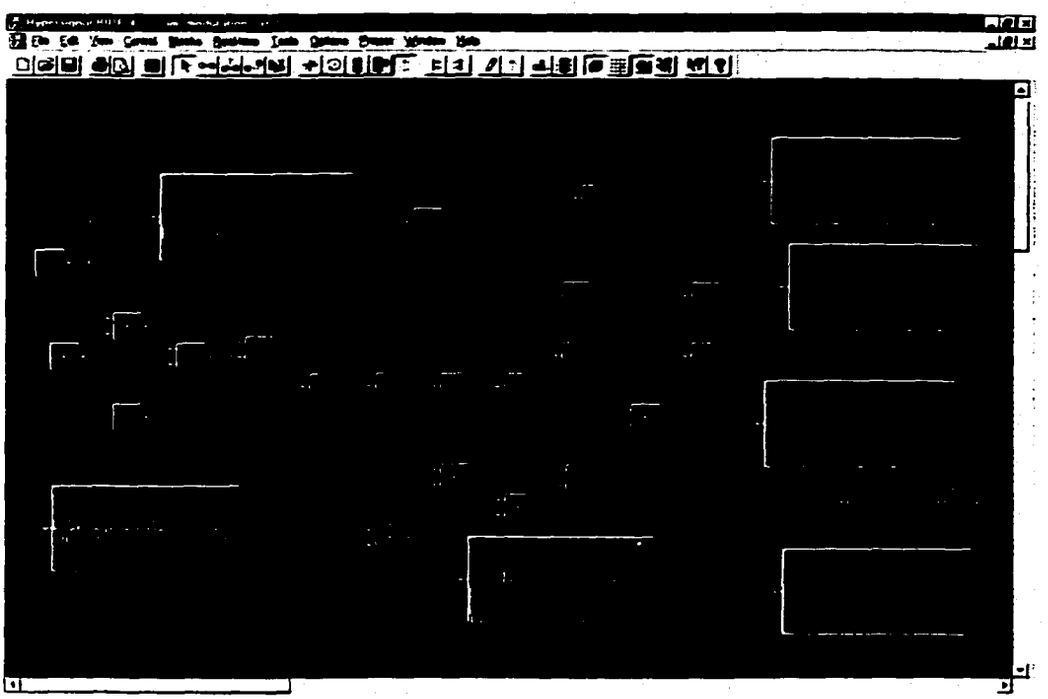


Figure 1 Typical Block Interface of RIDE

- It supports a wide range of industry standard DSP/acquisition boards directly.
- A digital filter can be designed with the support of Hypersignal Filter Design tools.
- The execution mode adapted by RIDE is data flow control.

Hardware Driver Wizard

A window's driver program is needed to handle the communication between target DSP board and host PC. Hypersignal RIDE not only directly provides drivers for some DSP boards, but also supports the development of a custom DSP driver with its supplementary Driver Wizard, which facilitates many of the tasks associated with creating a hardware driver. In this work, a custom DSP driver was built using the Hypersignal Driver Wizard. This driver links DSP COFF object files, downloads code,

data and parameters to the DSP memory, controls the execution of the DSP, and monitors activity on the DSP, and uploads data to host PC.

TMS320C30 DSP board

The TMS320C30 is a 32-bit floating-point processor which is a member of the TMS320C3x generation of DSPs from Texas Instruments [16-17]. The TMS320C30 optimizes speed by implementing signal processing functions in hardware, which provides performance previously unavailable on a single chip; Typically, the TMS320C30 can perform parallel multiply and ALU operations on integer or floating-point data in a single cycle. A wide variety of system functions from host processor to dedicated coprocessor are supported by the TMS320C30 due to its large address space, multiprocessor interface, internally and externally generated wait states, two external interface ports, two timers, serial ports, and multiple interrupt structure.

Appendix B: Driver code for TMS320C30

```
/*=============================================================

FILE:     DSPMEM.C

PURPOSE:  Provides routines that access the DSP board's memory.
          These routines provide single-address memory read/write,
          block memory read/write, memory fill, and memory search.

          This file provides routines for the Hypersignal for Windows
          DSP board driver.

GLOBAL ROUTINES:
-------------------------------------------------------------
Function  : DSPWriteShort()
Purpose   : This function writes an short value to DSP memory
-------------------------------------------------------------
Function  : DSPWriteLong()
Purpose   : This function writes a long value to DSP memory
-------------------------------------------------------------
Function  : DSPWriteFloat()
Purpose   : This function writes a float value to DSP memory
-------------------------------------------------------------
Function  : DSPWriteInst() // same as long
Purpose   : This function writes an instruction value to DSP memory
-------------------------------------------------------------
Function  : DSPReadShort()
Purpose   : This function reads an short value from DSP memory
-------------------------------------------------------------
Function  : DSPReadLong()
Purpose   : This function reads a long value from DSP memory
-------------------------------------------------------------
Function  : DSPReadFloat()
Purpose   : This function reads a float value from DSP memory
-------------------------------------------------------------
Function  : DSPReadInst()
Purpose   : This function reads an instruction value from DSP memory
-------------------------------------------------------------
Function  : DSPReadFlash()
Purpose   : This function reads a single FLASH memory address.
-------------------------------------------------------------
Function  : DSPWriteShortBuf()
Purpose   : This function writes a buffer of short values to DSP memory
-------------------------------------------------------------
Function  : DSPWriteLongBuf()
Purpose   : This function writes a buffer of long values to DSP memory
-------------------------------------------------------------
Function  : DSPWriteFloatBuf()
Purpose   : This function writes a buffer of float values to DSP memory
-------------------------------------------------------------
Function  : DSPWriteInstBuf()
Purpose   : This function writes a buffer of instruction values to DSP memory
-------------------------------------------------------------
```

```
   Function  : DSPWriteDataBuf()
   Purpose   : This function writes a buffer of data values to DSP memory
   ---------------------------------------------------------------------
   Function  : DSPReadShortBuf()
   Purpose   : This function reads a buffer of short values from DSP memory
   ---------------------------------------------------------------------
   Function  : DSPReadLongBuf()
   Purpose   : This function reads a buffer of long values from DSP memory
   ---------------------------------------------------------------------
   Function  : DSPReadFloatBuf()
   Purpose   : This function reads a buffer of float values from DSP memory
   ---------------------------------------------------------------------
   Function  : DSPReadInstBuf()
   Purpose   : This function reads a buffer of instruction values from DSP memory
   ---------------------------------------------------------------------
   Function  : TestMem()
   Purpose   : Performs a test of DSP memory
   ---------------------------------------------------------------------

   Copyright (C) 1994,1998 Hyperception, All rights reserved

=====================================================================
=*/


// Include Files
#include <windows.h>
#include <memory.h>
#include <conio.h>
#include "driver32.h"
#include "bddrvdsp.h"
#include "bddrv.h"
#include "board.h"
#include "entry.h"
#include "dsp.h"
#include "resource.h"
#include "drvutil.h"
#include "pcmem.h"
#include "address.h"

// External Data
extern BOOL   bAbortMemTest;        // set true when user aborts memory test

// Memory Test Definitions
#define TOTAL_DATA_PATTERNS     3
#define DATA_UNIQUE             0
#define DATA_CHECKERBOARD       1
#define DATA_ZERO               2


//  Define some constants used by lib functions //
#define ALL 1


static LPSTR DataPatternStr[TOTAL_DATA_PATTERNS] = {
     "unique memory patterns...",
```

```
    "checkerboard memory patterns...",
    "zero memory patterns..."
};


// Static Function Prototypes
static void NEAR PASCAL TestMemWrite(DSP_PARAM far *lpDSP, DWORD far *lpBuffer,
                        DWORD dwLen, DWORD dwAddr, int MemType,
                        int DataPattern);

static BOOL NEAR PASCAL TestMemRead(HANDLE hDlg, DSP_PARAM far *lpDSP, DWORD far
*lpBuffer,
                        DWORD dwLen, DWORD dwAddr, int MemType,
                        int DataPattern);

// Pragmas
#pragma code_seg("dspmem")


//=============================================================================
// Function   : DSPWriteShort()
// Purpose    : This function writes a short value to DSP memory
// Parameters : lpDSP   - Pointer to real-time block's specific DSP
//              dwAddr  - destination DSP memory address
//              ShortVal  - short to write
//              MemInfo - memory type
// Returns    : Nothing.
//=============================================================================
int FAR PASCAL DSPWriteShort(DSP_PARAM far *lpDSP, DWORD dwAddr, short ShortVal,
                int MemInfo)
{
 // TODO: add code to download short value


        if((MemInfo==PROG_MEM)||(MemInfo==INV_MEM)){

                if(lpDSP->bDSPRunStatus != DSPSTAT_HOLD)
                return(DRV_FUNC_FAIL);
        }
        else

                dwAddr|=0x30000;


        SetAddr(lpDSP,dwAddr);//puts an address into the interface area address ports


        CounterDis(lpDSP);                  //Disable interface port address counter.

        _outpw(lpDSP->wDataRegAddr,(ShortVal));//put the low 16bit of shortval in data register


}

//=============================================================================
// Function   : DSPWriteLong()
```

```
// Purpose   : This function writes a long value to DSP memory
// Parameters : lpDSP  - Pointer to real-time block's specific DSP
//              dwAddr - destination DSP memory address
//              LongVal - long to write
//              MemInfo - memory type
// Returns   : Nothing.
//=============================================================================
int FAR PASCAL DSPWriteLong(DSP_PARAM far *lpDSP, DWORD dwAddr, long LongVal,
                    int MemInfo)
{
 // TODO: add code to download long value
        if((MemInfo==PROG_MEM)||(MemInfo==INV_MEM)){

                if(lpDSP->bDSPRunStatus != DSPSTAT_HOLD)
                return(DRV_FUNC_FAIL);
        }
        else

                dwAddr|=0x30000;


        SetAddr(lpDSP,dwAddr);//puts an address into the interface area address ports

        CounterDis(lpDSP);              //Disable interface port address counter.

        _outpw(lpDSP->wDataRegAddr,(LongVal&0xffff));
        _outpw(lpDSP->wHiDataRegAddr,((LongVal&0xffff0000)>>16));

}

//=============================================================================
// Function  : DSPWriteFloat()
// Purpose   : This function writes a float value to DSP memory
// Parameters : lpDSP  - Pointer to real-time block's specific DSP
//              dwAddr - destination DSP memory address
//              FloatVal- float to write
//              MemInfo - memory type
// Returns   : Nothing.
//=============================================================================
int FAR PASCAL DSPWriteFloat(DSP_PARAM far *lpDSP, DWORD dwAddr,
         float FloatVal, int MemInfo)
{
        long      DSPValue;

        // convert IEEE float to DSP float
        DSPValue = IEEEtoDSP(FloatVal);

        // TODO: add code to download float value
        if((MemInfo==PROG_MEM)||(MemInfo==INV_MEM)){

                if(lpDSP->bDSPRunStatus != DSPSTAT_HOLD)
                return(DRV_FUNC_FAIL);
        }
        else

                dwAddr|=0x30000;
```

```
        SetAddr(lpDSP,dwAddr);//puts an address into the interface area address ports

        CounterDis(lpDSP);              //Disable interface port address counter.


        _outpw(lpDSP->wDataRegAddr,(DSPValue&0xffff));
        _outpw(lpDSP->wHiDataRegAddr,((DSPValue&0xffff0000)>>16));
}

//=============================================================================
// Function   : DSPWriteInst()
// Purpose    : This function writes an instruction value to DSP memory
// Parameters : lpDSP   - Pointer to real-time block's specific DSP
//              dwAddr  - destination DSP memory address
//              LongVal - long to write
//              MemInfo - memory type
// Returns    : Nothing.
//=============================================================================
void FAR PASCAL DSPWriteInst(DSP_PARAM far *lpDSP, DWORD dwAddr, DWORD dwMSW,
                DWORD dwLSW, int MemInfo)
{
 // TODO: add code to download instruction

        if((MemInfo==PROG_MEM)||(MemInfo==INV_MEM)){

                if(lpDSP->bDSPRunStatus != DSPSTAT_HOLD)
                return(DRV_FUNC_FAIL);
        }
        else

                dwAddr|=0x30000;


        SetAddr(lpDSP,dwAddr);//puts an address into the interface area address ports
        CounterEnb(lpDSP);              //Enable interface port address counter.
        AutoInc(lpDSP);


        _outpw(lpDSP->wDataRegAddr,(dwMSW&0xffff));
        _outpw(lpDSP->wHiDataRegAddr,((dwMSW&0xffff0000)>>16));

        _outpw(lpDSP->wDataRegAddr,(dwLSW&0xffff));
        _outpw(lpDSP->wHiDataRegAddr,((dwLSW&0xffff0000)>>16));


}

//=============================================================================
// Function   : DSPReadShort()
// Purpose    : This function reads a short value from DSP memory
// Parameters : lpDSP   - Pointer to real-time block's specific DSP
//              dwAddr  - source DSP memory address
//              MemInfo - memory type
// Returns    : Returns short value
```

```
//=============================================================================
short FAR PASCAL DSPReadShort(DSP_PARAM far *lpDSP, DWORD dwAddr, int MemInfo)
{
 // TODO: add code to upload short value
        short     ShortVal;

        ShortVal = 0;  // uploaded value

        if((MemInfo==PROG_MEM)||(MemInfo==INV_MEM)){

                if(lpDSP->bDSPRunStatus != DSPSTAT_HOLD)
                return(DRV_FUNC_FAIL);
        }
        else

                dwAddr|=0x30000;


        SetAddr(lpDSP,dwAddr);//puts an address into the interface area address ports

        CounterDis(lpDSP);                  //Disable interface port address counter.


        ShortVal=(unsigned short)_inpw(lpDSP->wDataRegAddr);

 return ((short)0);
}

//=============================================================================
// Function   : DSPReadLong()
// Purpose    : This function reads a long value from DSP memory
// Parameters : lpDSP   - Pointer to real-time block's specific DSP
//              dwAddr  - source DSP memory address
//              MemInfo - memory type
// Returns    : Returns long value
//=============================================================================
long FAR PASCAL DSPReadLong(DSP_PARAM far *lpDSP, DWORD dwAddr, int MemInfo)
{
 // TODO: add code to upload long value

        long      LongVal,c30exp;


        LongVal = 0;  // uploaded value

        if((MemInfo==PROG_MEM)||(MemInfo==INV_MEM)){

                if(lpDSP->bDSPRunStatus != DSPSTAT_HOLD)
                return(DRV_FUNC_FAIL);
        }
        else

                dwAddr|=0x30000;

        SetAddr(lpDSP,dwAddr);//puts an address into the interface area address ports
```

```
        CounterDis(lpDSP);              //Disable interface port address counter.

        c30exp=(unsigned long)_inpw(lpDSP->wDataRegAddr);
        LongVal=(((long)_inpw(lpDSP->wHiDataRegAddr))<<16)|c30exp;

 return(0L);
}

//=============================================================================
// Function  : DSPReadFloat()
// Purpose   : This function reads a float value from DSP memory
// Parameters : lpDSP  - Pointer to real-time block's specific DSP
//          dwAddr  - source DSP memory address
//          MemInfo - memory type
// Returns   : Returns float value
//=============================================================================
float FAR PASCAL DSPReadFloat(DSP_PARAM far *lpDSP, DWORD dwAddr, int MemInfo)
{
        float    FloatVal;
        long     LongVal,c30exp;

        // TODO: add code to upload float value
        LongVal = 0;  // uploaded value

        if((MemInfo==PROG_MEM)||(MemInfo==INV_MEM)){

                if(lpDSP->bDSPRunStatus != DSPSTAT_HOLD)
                return(DRV_FUNC_FAIL);
        }
        else

                dwAddr|=0x30000;

        SetAddr(lpDSP,dwAddr);//puts an address into the interface area address ports

        CounterDis(lpDSP);              //Disable interface port address counter.

        c30exp=(unsigned short)_inpw(lpDSP->wDataRegAddr);
        LongVal=(((long)_inpw(lpDSP->wHiDataRegAddr))<<16)|c30exp;


        // convert to IEEE float
        FloatVal = (float)DSPtoIEEE(LongVal);

        return (FloatVal);
}

//=============================================================================
// Function  : DSPReadInst()
// Purpose   : This function reads a instruction value from DSP memory
// Parameters : lpDSP  - Pointer to real-time block's specific DSP
//          dwAddr  - source DSP memory address
//          MemInfo - memory type
// Returns   : Returns long value
//=============================================================================
```

```
DWORD FAR PASCAL DSPReadInst(DSP_PARAM far *lpDSP, DWORD dwAddr,
                DWORD far *dwMSW, int MemInfo)
{
 // clear MSW

        long    Low;

        *dwMSW = 0UL;

        if((MemInfo==PROG_MEM)||(MemInfo==INV_MEM)){

                if(lpDSP->bDSPRunStatus != DSPSTAT_HOLD)
                return(DRV_FUNC_FAIL);
        }
        else

                dwAddr|=0x30000;

        SetAddr(lpDSP,dwAddr);//puts an address into the interface area address ports

        CounterDis(lpDSP);              //Disable interface port address counter.


        Low=(unsigned long)_inpw(lpDSP->wDataRegAddr);
        *dwMSW=(((long)_inpw(lpDSP->wHiDataRegAddr))<<16)|Low;


 return(0L);
}


//=============================================================================
// Function  : DSPReadFlash()
// Purpose   : This function reads a single FLASH memory address.
// Parameters : lpDSP   - Pointer to real-time block's specific DSP
//          dwAddr  - source DSP memory address
// Returns   : Nothing.
//=============================================================================
DWORD FAR PASCAL DSPReadFlash (DSP_PARAM far *lpDSP, DWORD dwAddr)
{
 // TODO: add code to read flash memory
        DWORD           dwLowAddr, dwFlashMemAddr;


        SetAddr(lpDSP,dwAddr);//puts an address into the interface area address ports

        CounterDis(lpDSP);              //Disable interface port address counter.

        dwLowAddr=(unsigned short)_inpw(lpDSP->wDataRegAddr);
        dwFlashMemAddr=(((long)_inpw(lpDSP->wHiDataRegAddr))<<16)|dwLowAddr;


 return((DWORD)0);
}
```

```
//=============================================================================
// Function   : DSPWriteShortBuf()
// Purpose    : This function writes a buffer of short values to DSP memory
// Parameters : lpDSP   - Pointer to real-time block's specific DSP
//              dwAddr  - destination DSP memory address
//              dwLen   - length of data buffer
//              lpShortBuf- pointer to short data buffer
//              MemInfo - memory type
// Returns    : Nothing.
//=============================================================================
int FAR PASCAL DSPWriteShortBuf(DSP_PARAM far *lpDSP, DWORD dwAddr,
                    DWORD dwLen, short huge *lpShortBuf, int MemInfo)
{
 // TODO: add code to download short buffer
        DWORD           count=0;

        if((MemInfo==PROG_MEM)||(MemInfo==INV_MEM)){

                if(lpDSP->bDSPRunStatus != DSPSTAT_HOLD)
                return(DRV_FUNC_FAIL);
        }
        else

                dwAddr|=0x30000;

        SetAddr(lpDSP,dwAddr);//puts an address into the interface area address ports
        CounterEnb(lpDSP);              //Enable interface port address counter.
        AutoInc(lpDSP);

        while(count<dwLen){
        _outpw(lpDSP->wDataRegAddr,(*(lpShortBuf+count)));
        count++;
        }
}


//=============================================================================
// Function   : DSPWriteLongBuf()
// Purpose    : This function writes a buffer of long values to DSP memory
// Parameters : lpDSP    - Pointer to real-time block's specific DSP
//              dwAddr   - destination DSP memory address
//              dwLen    - length of data buffer
//              lpLongBuf- pointer to long data buffer
//              MemInfo  - memory type
// Returns    : Nothing.
//=============================================================================
int FAR PASCAL DSPWriteLongBuf(DSP_PARAM far *lpDSP, DWORD dwAddr,
                    DWORD dwLen, long huge *lpLongBuf,
                    int MemInfo)
{
 // TODO: add code to download long buffer
        DWORD           count=0;

/*      if((MemInfo==PROG_MEM)||(MemInfo==INV_MEM)){

                if(lpDSP->bDSPRunStatus != DSPSTAT_HOLD)
```

```
            return(DRV_FUNC_FAIL);
        }
        else */

            dwAddr|=0x30000;


        SetAddr(lpDSP,dwAddr);//puts an address into the interface area address ports
        CounterEnb(lpDSP);              //Enable interface port address counter.
        AutoInc(lpDSP);

        while(count<dwLen){
        _outpw(lpDSP->wDataRegAddr,(*(lpLongBuf+count)&0xffff));
        _outpw(lpDSP->wHiDataRegAddr,((*(lpLongBuf+count)&0xffff0000)>>16));
        count++;
        }

}

//=============================================================================
// Function   : DSPWriteFloatBuf()
// Purpose    : This function writes a buffer of float values to DSP memory
// Parameters : lpDSP    - Pointer to real-time block's specific DSP
//              dwAddr   - destination DSP memory address
//              dwLen    - length of data buffer
//              lpFloatBuf- pointer to float data buffer
//              MemInfo  - memory type
// Returns    : Nothing.
//=============================================================================
int FAR PASCAL DSPWriteFloatBuf(DSP_PARAM far *lpDSP, DWORD dwAddr,
                    DWORD dwLen, float huge *lpFloatBuf,
                    int MemInfo)
{
 // TODO: add code to download float buffer
        long      *lpDSPBuf;
        DWORD           count=0;

        if((MemInfo==PROG_MEM)||(MemInfo==INV_MEM)){

            if(lpDSP->bDSPRunStatus != DSPSTAT_HOLD)
            return(DRV_FUNC_FAIL);
        }
        else

            dwAddr|=0x30000;


        SetAddr(lpDSP,dwAddr);//puts an address into the interface area address ports
        CounterEnb(lpDSP);              //Enable interface port address counter.
        AutoInc(lpDSP);

        while(count<dwLen){
        *(lpDSPBuf+count) = IEEEtoDSP(*(lpFloatBuf+count));
        _outpw(lpDSP->wDataRegAddr,(*(lpDSPBuf+count)&0xffff));
        _outpw(lpDSP->wHiDataRegAddr,((*(lpDSPBuf+count)&0xffff0000)>>16));
        count++;
```

```
        }
}

//=============================================================================
// Function   : DSPWriteInstBuf()
// Purpose    : This function writes a buffer of instruction values to DSP memory
// Parameters : lpDSP    - Pointer to real-time block's specific DSP
//              dwAddr   - destination DSP memory address
//              dwLen    - length of data buffer
//              lpLongBuf- pointer to long data buffer
//              MemInfo  - memory type
// Returns    : Nothing.
//=============================================================================
void FAR PASCAL DSPWriteInstBuf(DSP_PARAM far *lpDSP, DWORD dwAddr,
                    DWORD dwLen, void huge *lpLongBuf,
                    int MemInfo)
{
 // TODO: add code to download instruction buffer
        DWORD           count=0;

        long      *lpDspBuf;

        lpDspBuf=(long *)lpLongBuf;

        if((MemInfo==PROG_MEM)||(MemInfo==INV_MEM)){

                if(lpDSP->bDSPRunStatus != DSPSTAT_HOLD)
                return(DRV_FUNC_FAIL);
        }
        else

                dwAddr|=0x30000;


        SetAddr(lpDSP,dwAddr);//puts an address into the interface area address ports
        CounterEnb(lpDSP);                  //Enable interface port address counter.
        AutoInc(lpDSP);

        while(count<dwLen){

        _outpw(lpDSP->wDataRegAddr,(*(lpDspBuf+count)&0xffff));
        _outpw(lpDSP->wHiDataRegAddr,((*(lpDspBuf+count)&0xffff0000)>>16));
        count++;
        }
}

//=============================================================================
// Function   : DSPReadShortBuf()
// Purpose    : This function reads a buffer of short values from DSP memory
// Parameters : lpDSP    - Pointer to real-time block's specific DSP
//              dwAddr   - source DSP memory address
//              dwLen    - length of data buffer
//              lpShortBuf- pointer to short data buffer
//              MemInfo  - memory type
// Returns    : Nothing.
//=============================================================================
```

```
int FAR PASCAL DSPReadShortBuf(DSP_PARAM far *lpDSP, DWORD dwAddr,
                DWORD dwLen, short huge *lpShortBuf, int MemInfo)
{
 // TODO: add code to upload short buffer
        DWORD           count=0;


        if((MemInfo==PROG_MEM)||(MemInfo==INV_MEM)){

                if(lpDSP->bDSPRunStatus != DSPSTAT_HOLD)
                return(DRV_FUNC_FAIL);
        }
        else

                dwAddr|=0x30000;


        SetAddr(lpDSP,dwAddr);//puts an address into the interface area address ports

        CounterEnb(lpDSP);              //Enable interface port address counter.
        AutoInc(lpDSP);

        while(count<dwLen) {
  *(lpShortBuf+count)=(unsigned short)_inpw(lpDSP->wDataRegAddr);
  count++;
  }
return(0);
}

//=============================================================================
// Function   : DSPReadLongBuf()
// Purpose    : This function reads a buffer of long values from DSP memory
// Parameters : lpDSP    - Pointer to real-time block's specific DSP
//              dwAddr   - source DSP memory address
//              dwLen    - length of data buffer
//              lpLongBuf- pointer to long data buffer
//              MemInfo  - memory type
// Returns    : Nothing.
//=============================================================================
int FAR PASCAL DSPReadLongBuf(DSP_PARAM far *lpDSP, DWORD dwAddr,
                DWORD dwLen, long huge *lpLongBuf, int MemInfo)
{
 // TODO: add code to upload long buffer
        DWORD           count=0;
        long      low=0, high=0;


/*      if((MemInfo==PROG_MEM)||(MemInfo==INV_MEM)){

                if(lpDSP->bDSPRunStatus != DSPSTAT_HOLD)
                return(DRV_FUNC_FAIL);
        }
        else */

                dwAddr|=0x30000;
```

```
        SetAddr(lpDSP,dwAddr);//puts an address into the interface area address ports

        CounterEnb(lpDSP);              //Enable interface port address counter.
        AutoInc(lpDSP);

        while(count<dwLen) {
  low=(unsigned short)_inpw(lpDSP->wDataRegAddr);
  high=(((long)_inpw(lpDSP->wHiDataRegAddr))<<16)|low;
  *(lpLongBuf+count)=high;
  count++;
  }
return(0);
}

//=============================================================================
// Function   : DSPReadFloatBuf()
// Purpose    : This function reads a buffer of float values from DSP memory
// Parameters : lpDSP     - Pointer to real-time block's specific DSP
//              dwAddr    - source DSP memory address
//              dwLen     - length of data buffer
//              lpFloatBuf- pointer to float data buffer
//              MemInfo   - memory type
// Returns    : Nothing.
//=============================================================================
int FAR PASCAL DSPReadFloatBuf(DSP_PARAM far *lpDSP, DWORD dwAddr,
                    DWORD dwLen, float huge *lpFloatBuf, int MemInfo)
{
 // TODO: add code to upload float buffer
        DWORD           count=0;
        long      low=0, high=0;
        long       *lpLongBuf;


        if((MemInfo==PROG_MEM)||(MemInfo==INV_MEM)){

                if(lpDSP->bDSPRunStatus != DSPSTAT_HOLD)
                return(DRV_FUNC_FAIL);
        }
        else

                dwAddr|=0x30000;


        SetAddr(lpDSP,dwAddr);//puts an address into the interface area address ports
        CounterEnb(lpDSP);              //Enable interface port address counter.
        AutoInc(lpDSP);

        while(count<dwLen) {
  low=(unsigned short)_inpw(lpDSP->wDataRegAddr);
  high=(((long)_inpw(lpDSP->wHiDataRegAddr))<<16)|low;
  *(lpLongBuf+count)=high;
        *(lpFloatBuf+count)=(float)DSPtoIEEE(*(lpLongBuf+count));
  count++;
  }
return(0);
```

```
}

//=============================================================================
// Function   : DSPReadInstBuf()
// Purpose    : This function reads a buffer of long values from DSP memory
// Parameters : lpDSP    - Pointer to real-time block's specific DSP
//              dwAddr   - source DSP memory address
//              dwLen    - length of data buffer
//              lpLongBuf- pointer to long data buffer
//              MemInfo  - memory type
// Returns    : Nothing.
//=============================================================================
void FAR PASCAL DSPReadInstBuf(DSP_PARAM far *lpDSP, DWORD dwAddr,
                    DWORD dwLen, void huge *lpLongBuf, int MemInfo)
{
 // TODO: add code to upload instruction buffer
        long     *lpDspBuf;
        DWORD          count=0;
        long     low=0, high=0;


        if((MemInfo==PROG_MEM)||(MemInfo==INV_MEM)){

                if(lpDSP->bDSPRunStatus != DSPSTAT_HOLD)
                return(DRV_FUNC_FAIL);
        }
        else

                dwAddr|=0x30000;


        SetAddr(lpDSP,dwAddr);//puts an address into the interface area address ports

        CounterEnb(lpDSP);              //Enable interface port address counter.
        AutoInc(lpDSP);

        while(count<dwLen) {
  low=(unsigned short)_inpw(lpDSP->wDataRegAddr);
  high=(((long)_inpw(lpDSP->wHiDataRegAddr))<<16)|low;
  *(lpDspBuf+count)=high;

  count++;
  }

        lpLongBuf=(void*)lpDspBuf;
return(0);
}



//=============================================================================
// Function   : TestMemWrite()
// Purpose    : Writes test data pattern to target memory
// Parameters : lpDSP       - pointer to DSP parameter structure
//              lpBuffer    - pointer to data buffer
//              dwLen       - length of buffer
```

```
//          dwAddr      - current target memory address
//          MemType     - target memory type
//          DataPattern - data pattern code
// Returns  : Nothing.
//=============================================================================
static void NEAR PASCAL TestMemWrite(DSP_PARAM far *lpDSP, DWORD far *lpBuffer,
                        DWORD dwLen, DWORD dwAddr, int MemType,
                        int DataPattern)
{
   DWORD   dwIndex;
   DWORD   dwValue;


   // write data into buffer based on data pattern type

   switch (DataPattern) {

     case DATA_UNIQUE:
        // write address value into memory
        for (dwIndex=0; dwIndex<dwLen; dwIndex++) {
           lpBuffer[dwIndex] = dwAddr + dwIndex;
        }
        break;

     case DATA_CHECKERBOARD:
        // write checkerboard value into memory
        for (dwIndex=0; dwIndex<dwLen; dwIndex++) {
           dwValue = (DWORD)(0x55555555UL << ((dwAddr+dwIndex) & 1));
           lpBuffer[dwIndex] = dwValue;
        }
        break;

     case DATA_ZERO:
        // write 0's to buffer
        for (dwIndex=0; dwIndex<dwLen; dwIndex++) {
           lpBuffer[dwIndex] = 0UL;
        }
        break;
   }

   // write buffer to target memory
   lpDSP->TargetWriteLongBuf(lpDSP,dwAddr,dwLen,lpBuffer,MemType);
}


//=============================================================================
// Function  : TestMemRead()
// Purpose   : Reads test data pattern from target memory
// Parameters : hDlg        - handle of test status modeless dialog box
//          lpDSP       - pointer to DSP parameter structure
//          lpBuffer    - pointer to data buffer
//          dwLen       - length of buffer
//          dwAddr      - current target memory address
//          MemType     - target memory type
//          DataPattern - data pattern code
// Returns  : Memory read result (PASS/FAIL)
```

```
//=============================================================================

static BOOL NEAR PASCAL TestMemRead(HANDLE hDlg, DSP_PARAM far *lpDSP, DWORD far
*lpBuffer,
                    DWORD dwLen, DWORD dwAddr, int MemType,
                    int DataPattern)
{
   DWORD   dwIndex;
         DWORD   dwValue;
   DWORD   dwExpect;
   BOOL    TestStatus = TRUE;
   int     MBResponse;
   char    ErrStr[128];

   // read data from buffer
   lpDSP->TargetReadLongBuf(lpDSP,dwAddr,dwLen,lpBuffer,MemType);

   // scan through memory buffer validating data
   for (dwIndex=0; dwIndex<dwLen; dwIndex++) {

     // get expected data pattern based on data pattern type
     switch (DataPattern) {

       case DATA_UNIQUE:
          // expected value is memory address
          dwExpect = dwAddr + dwIndex;
          break;

       case DATA_CHECKERBOARD:
          // expected value is checkerboard based on memory address
          dwExpect = (DWORD)(0x55555555UL << ((dwAddr+dwIndex) & 1));
          break;

       case DATA_ZERO:
          // expected value is 0
          dwExpect = 0UL;
          break;
     }

     // check if memory value is correct
     if ((dwValue = lpBuffer[dwIndex]) != dwExpect) {
       // set fail status
       TestStatus = FALSE;

       // indicate failure to user
       wsprintf(ErrStr,"Memory read error at 0x%lx.  The value 0x%lx which was read should be 0x%lx.",
            (dwAddr+dwIndex),dwValue,dwExpect);
       MessageBox(hDlg,ErrStr,"Memory Test Error",MB_OK|MB_ICONEXCLAMATION);
       MBResponse = MessageBox(hDlg,"Do you want to continue the memory test?","Memory Test",
                    MB_ICONQUESTION|MB_YESNO);
       if (MBResponse==IDNO) {
         // set global abort flag
         bAbortMemTest = TRUE;
         return(TestStatus);
       }
     }
```

```
    }

    // return test status
    return (TestStatus);
}


//=============================================================================
// Function   : TestMem()
// Purpose    : Performs a test of the DSP memory
// Parameters : hDlg         - handle of configuration dialog box
//              hTestModeless - handle of test status modeless dialog box
//              lpDSP        - pointer to DSP parameter structure
// Returns    : Nothing.
//=============================================================================
BOOL FAR PASCAL TestMem(HANDLE hDlg, HANDLE hTestModeless, DSP_PARAM far *lpDSP)
{
    BOOL        bMemoryTest = TRUE;  // default to pass, if error change to FALSE          // memory
size in kWords
    BOOL        bCurrMemTest;
    DWORD       dwTotalNumWords;
    DWORD       dwNumWordsDone;
    DWORD       dwSectWords;
    DWORD       dwSectWordsRemain;
    DWORD       dwStartAddr;
    DWORD       dwCurrAddr;
    DWORD       dwTotalBuffSize;
    DWORD       dwCurrBuffSize;
    HANDLE      hBuffer;
    DWORD far    *lpBuffer;
    int         Percent;
        int         DataPattern;
    MSG         msg;
    char        StatusStr[80];
    WORD        wSectNum;       // section number counter
    SECTION_PARAM   *lpDSPSections;   // pointer to sections
    int         MemType;


    // get pointer to DSP sections
    if ((lpDSPSections = (SECTION_PARAM FAR *)GlobalLock(lpDSP->hDSPSections))==NULL)
return(FALSE);

    // determine total number of addresses to test
    dwTotalNumWords = 0UL;
    for (wSectNum=0; wSectNum<lpDSP->wNumSections; wSectNum++) {
      dwTotalNumWords += lpDSPSections[wSectNum].dwLength;
    }

    // check if no memory
    if (dwTotalNumWords == 0) return(bMemoryTest);

    // allocate memory buffer
    // determine buffer size (5% or 2048; whichever is smaller)
    if ((dwTotalBuffSize = (dwTotalNumWords / 20)) > 2048UL)  dwTotalBuffSize = 2048UL;
    if (dwTotalBuffSize == 0) dwTotalBuffSize = dwTotalNumWords;
```

```
    if ((hBuffer =
GlobalAlloc(GMEM_MOVEABLE|GMEM_ZEROINIT,(DWORD)sizeof(DWORD)*dwTotalBuffSize))
== NULL) {
      MessageBox(hDlg,"Memory test buffer memory could not be allocated.","Memory Test
Error",MB_OK|MB_ICONEXCLAMATION);
      return(FALSE);
    }
    if ((lpBuffer = (DWORD far *)GlobalLock(hBuffer)) == NULL) {
      MessageBox(hDlg,"Memory test buffer memory could not be locked.","Memory Test
Error",MB_OK|MB_ICONEXCLAMATION);
      return(FALSE);
    }

    // loop through all test data patterns
    for (DataPattern = 0; DataPattern < TOTAL_DATA_PATTERNS; DataPattern++)  {

      // write data from memory
      wsprintf(StatusStr,"Writing %s",DataPatternStr[DataPattern]);
      SetDlgItemText(hTestModeless,IDC_TESTSTATUS,StatusStr);

      // clear number of addresses done
      dwNumWordsDone = 0UL;

      // write data to all all sections
      for (wSectNum=0; wSectNum<lpDSP->wNumSections; wSectNum++) {

        // get start address and number of words in section
        dwStartAddr = lpDSPSections[wSectNum].dwFirstAddr;
        dwSectWords = lpDSPSections[wSectNum].dwLength;

      // don't corrupt the monitor program which is located at 0x000000
      if ((dwStartAddr >= 0x900000) &&
         (dwStartAddr <= 0x900040)) continue;


        // initialize current address
        dwCurrAddr  = dwStartAddr;

        // initialize section words remaining count
        dwSectWordsRemain = dwSectWords;

        // get memory type
        MemType = (int)lpDSPSections[wSectNum].bMemType;

        // set test title
        SetDlgItemText(hTestModeless,IDC_MEMTYPE,lpDSPSections[wSectNum].SectName);

        // loop until all of section is tested
        while (dwSectWordsRemain) {

          // determine buffer size for next write (limit to total buffer size)
          dwCurrBuffSize = dwSectWordsRemain;
          if (dwCurrBuffSize > dwTotalBuffSize) dwCurrBuffSize = dwTotalBuffSize;

          // write memory
```

```
        TestMemWrite(lpDSP,lpBuffer,dwCurrBuffSize,dwCurrAddr,MemType,DataPattern);

        // update current address
        dwCurrAddr += dwCurrBuffSize;

        // update number of section words remaining
        dwSectWordsRemain -= dwCurrBuffSize;

        // update total number of words done
        dwNumWordsDone += dwCurrBuffSize;

        // update/display percent done
        Percent = (int)(((float)dwNumWordsDone / (float)dwTotalNumWords) * 100.0f);
        SetDlgItemInt(hTestModeless,IDC_TESTPERCENT,Percent,FALSE);

        // check if user abort
        if (bAbortMemTest)  return (bMemoryTest);

        // allow other Window's messages to be processed
        if (PeekMessage(&msg,NULL,0,0,PM_REMOVE))  {
          TranslateMessage(&msg);
          DispatchMessage(&msg);
        }
      }
    }
    SetDlgItemInt(hTestModeless,IDC_TESTPERCENT,100,FALSE);

    // read data from memory
    wsprintf(StatusStr,"Reading %s",DataPatternStr[DataPattern]);
    SetDlgItemText(hTestModeless,IDC_TESTSTATUS,StatusStr);

    // clear number of addresses done
    dwNumWordsDone = 0UL;

    // read data from all sections
    for (wSectNum=0; wSectNum<lpDSP->wNumSections; wSectNum++) {

      // get start address and number of words in section
      dwStartAddr = lpDSPSections[wSectNum].dwFirstAddr;
      dwSectWords = lpDSPSections[wSectNum].dwLength;

      // don't corrupt the monitor program which is located at 0x000000
    if ((dwStartAddr >= 0x900000) &&
       (dwStartAddr <= 0x900040)) continue;


      // initialize current address
      dwCurrAddr  = dwStartAddr;

      // initialize section words remaining count
      dwSectWordsRemain = dwSectWords;

      // get memory type
      MemType = (int)lpDSPSections[wSectNum].bMemType;

      // set test title
```

```
        SetDlgItemText(hTestModeless,IDC_MEMTYPE,lpDSPSections[wSectNum].SectName);

          // loop until all of section is tested
        while (dwSectWordsRemain) {

          // determine buffer size for next write (limit to total buffer size)
          dwCurrBuffSize = dwSectWordsRemain;
          if (dwCurrBuffSize > dwTotalBuffSize) dwCurrBuffSize = dwTotalBuffSize;

          // read memory
          bCurrMemTest =
TestMemRead(hTestModeless,lpDSP,lpBuffer,dwCurrBuffSize,dwCurrAddr,MemType,DataPattern);

          // set memory test flag if it is true (won't allow FAIL to be
          // overwritten by a section memory test PASS)
          if (bMemoryTest) bMemoryTest = bCurrMemTest;

          // update current address
          dwCurrAddr += dwCurrBuffSize;

          // update number of section words remaining
          dwSectWordsRemain -= dwCurrBuffSize;

          // update total number of words done
          dwNumWordsDone += dwCurrBuffSize;

          // update/display percent done
          Percent = (int)(((float)dwNumWordsDone / (float)dwTotalNumWords) * 100.0f);
          SetDlgItemInt(hTestModeless,IDC_TESTPERCENT,Percent,FALSE);

          // check if user abort
          if (bAbortMemTest) {
            // unlock memory
            FreeGlobalMemory(hBuffer);
            GlobalUnlock(lpDSP->hDSPSections);

            // return memory test result
            return (bMemoryTest);
          }

          // allow other Window's messages to be processed
          if (PeekMessage(&msg,NULL,0,0,PM_REMOVE)) {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
          }
        }
      }
    }

    // unlock memory
    FreeGlobalMemory(hBuffer);
    GlobalUnlock(lpDSP->hDSPSections);

    // return memory test result
    return (bMemoryTest);
}
```

```
/* ---------------------------------------------------------------
* Function SetAddr()
*
* Purpose    : puts an address into the interface area address ports.
* Parameters : the 24 bit address.
* Returns    : none.
*/
void FAR PASCAL SetAddr(DSP_PARAM far *lpDSP,DWORD dwAddr)
{

        _outpw(lpDSP->wAddrRegAddr, (dwAddr&0xFFFF));
        _outpw(lpDSP->wHiAddrRegAddr, ((dwAddr&0xFF0000)>>16));
}


/* ---------------------------------------------------------------
* Function CounterDis()
*
* Purpose    : Disable interface port address counter.
* Parameters : none.
* Returns    : Value of Ctrl Reg variable for current board.
*/
unsigned short FAR PASCAL CounterDis(DSP_PARAM far *lpDSP)

{
  unsigned short ControlVal;

  ControlVal=lpDSP->wControlReg;

  ControlVal|=0x0004;  /* ensure bit 2 hi */
  _outpw(lpDSP->wControlRegAddr,ControlVal);  /* put word out to I/O port.  */
  return (ControlVal);

}


/* ---------------------------------------------------------------
* Function CounterEnb()
*
* Purpose    : Enable interface port address counter.
* Parameters : none.
* Returns    : Value of Ctrl Reg variable for current board.
*/
unsigned short FAR PASCAL CounterEnb(DSP_PARAM far *lpDSP)

{
        unsigned short ControlVal;

  ControlVal=lpDSP->wControlReg;

  ControlVal&=0xFFFB;  /* ensure bit 2 low */
  _outpw(lpDSP->wControlRegAddr,ControlVal);  /* put word out to I/O port.  */
  return (ControlVal);
}
```

```
/* ---------------------------------------------------------
* Function AutoInc()
*
* Purpose    : Put interface port address counter to increment mode.
* Parameters : none.
* Returns    : Value of Ctrl Reg variable for current board.
*/
unsigned short FAR PASCAL AutoInc(DSP_PARAM far *lpDSP)

{

  unsigned short ControlVal;

  ControlVal=lpDSP->wControlReg;
  ControlVal&=0xFFF7;  /* ensure bit 3 low */

  _outpw(lpDSP->wControlRegAddr,ControlVal); /* put word out to I/O port.  */
  return (ControlVal);

}

/* ---------------------------------------------------------
* Function AutoDec()
*
* Purpose    : Put interface port address counter to decrement.
* Parameters : none.
* Returns    : Value of Ctrl Reg variable for current board.
*/
unsigned short FAR PASCAL AutoDec(DSP_PARAM far *lpDSP)

{
        unsigned short ControlVal;

  ControlVal=lpDSP->wControlReg;
  ControlVal|=0x0008;  /* ensure bit 3 hi */

  _outpw(lpDSP->wControlRegAddr,ControlVal); /* put word out to I/O port.  */
  return (ControlVal);

}
```

Appendix C: Data acquisition code

```
//=============================================================================
=
//
//   FILE NAME : _DAQU.c
//
//   BLOCK NAME: Daqu
//
//   GROUP NAME: A/D functions
//
//   PURPOSE   : Provides the real-time block's DSP C source code.
//
//   Hypersignal Block Wizard Version 4.00.14 Auto-Generated Block
//
//   Number of Inputs : 0
//   Number of Outputs: 6
//
//   Creation Date: Wed - 14 February 2001
//   Creation Time: 01:27 PM
//
//=============================================================================
=
*/

#include "_daqu.h"


/*-------------------------------------------------------------------*/
/*            Optional real-time block interrupt routine             */
/*-------------------------------------------------------------------*/
/* If this routine is activated, it will be called in response to a selected  */
/* DSP interrupt. If this routine in not activated, the main block routine   */
/* will be called in response to a selected DSP interrupt.                   */
/*-------------------------------------------------------------------*/
/*
void DAQU_INT(PARAMS *pPtr)
{

}
*/
void c_int01(PARAMS *pPtr)

{
        unsigned int      index;
        float                     t;


        Dis_Interrupts();

  Buffer_Put(A2D(0)*0.0024414*C_Gain);
        Buffer_Put(A2D(2)*0.0024414*C_Gain);
```

```
        Buffer_Put(A2D(4)*0.0024414*C_Gain);
        Buffer_Put(A2D(6)*0.0024414*V_Gain);
        Buffer_Put(A2D(7)*0.0024414*V_Gain);
        Buffer_Put(A2D(8)*0.0024414*V_Gain);



        flag=1;
        if(HjxHead==HjxTail) Buffer_Full=1;

        Clear_Intr_Flag();
        En_Interrupts();


}

/*-------------------------------------------------------------------*/
/*          Optional real-time block initialization routine          */
/*-------------------------------------------------------------------*/
/* If this routine is activated, it will be called one time before the main  */
/* application begins. This allows for any required software or hardware     */
/* initialization to be performed before the block executes.                 */
/*-------------------------------------------------------------------*/

void DAQU_INIT(PARAMS *pPtr)
{
        Dis_Interrupts();

        Enable_Int1();

        Set_Sample_Rate(pPtr->_Sampling_Rate);
  Set_Filter_Cutoff(1920);
        Clear_Buffer(pPtr->FramesizeOut0*6);
        C_Gain=pPtr->_Gain_Current;
        V_Gain=pPtr->_Gain_Voltage;

/*      FltCounter=0;*/

        En_Interrupts();

}



/*-------------------------------------------------------------------*/
/*               Optional real-time block stop routine               */
/*-------------------------------------------------------------------*/
/* If this routine is activated, it will be called whenever the block diagram */
/* worksheet's execution is stopped. Blocks that deal with hardware may need */
/* this routine to stop the hardware's execution.                            */
/*-------------------------------------------------------------------*/

void DAQU_STOP(PARAMS *pPtr)
{
        Dis_Interrupts();
```

```
}


/*-------------------------------------------------------------*/
/*              Optional real-time block restart routine       */
/*-------------------------------------------------------------*/
/* If this routine is activated, it will be called whenever a block diagram  */
/* worksheet is executed after being stopped. Blocks that deal with hardware */
/* may need this routine to restart the hardware's execution.                */
/*-------------------------------------------------------------*/

void DAQU_RESTART(PARAMS *pPtr)
{
        En_Interrupts();
}


/*-------------------------------------------------------------*/
/*                  Real-time block routine                    */
/*-------------------------------------------------------------*/
/* This is the main block routine.  It is called during each loop of the main */
/* application.  If an interrupt is selected, and the interrupt routine above */
/* is not activated, this routine will be called in response to the selected  */
/* interrupt instead of during the main application loop.                     */
/*-------------------------------------------------------------*/
void DAQU(PARAMS *pPtr)
{
 unsigned int index;     /* index for frame processing loop */
 unsigned int HjxStart;
 unsigned int HjxEnd;


        Dis_Interrupts();

        if(pPtr->SyncOut) *(pPtr->SyncOut)=FALSE;


        if((Buffer_Full==1)&&(flag==1))
        {
                HjxStart=(HjxTail-HjxHead)/6;
                HjxEnd=HjxHead/6;

                for(index=0;index<HjxStart;index++)
                {
                        *(pPtr->PtrOut0+index) =HjxBuffer[HjxHead+index*6];
                        *(pPtr->PtrOut1+index) =HjxBuffer[HjxHead+index*6+1];
                        *(pPtr->PtrOut2+index) =HjxBuffer[HjxHead+index*6+2];
                        *(pPtr->PtrOut3+index) =HjxBuffer[HjxHead+index*6+3];
                        *(pPtr->PtrOut4+index) =HjxBuffer[HjxHead+index*6+4];
                        *(pPtr->PtrOut5+index) =HjxBuffer[HjxHead+index*6+5];
                }
                for(index=0;index<HjxEnd;index++)
```

```
                }
                        *(pPtr->PtrOut0+HjxStart+index)=HjxBuffer[index*6];
                        *(pPtr->PtrOut1+HjxStart+index)=HjxBuffer[index*6+1];
                        *(pPtr->PtrOut2+HjxStart+index)=HjxBuffer[index*6+2];
                        *(pPtr->PtrOut3+HjxStart+index)=HjxBuffer[index*6+3];
                        *(pPtr->PtrOut4+HjxStart+index)=HjxBuffer[index*6+4];
                        *(pPtr->PtrOut5+HjxStart+index)=HjxBuffer[index*6+5];
                }
                if(pPtr->SyncOut) *(pPtr->SyncOut)=TRUE;
                flag=0;
        }

        En_Interrupts();


}

float Set_Timer(int n, float freq, long mode)
{
        long period;
        long *timer = n ? Timer1 : Timer0;

        if (mode & CLK_MODE)
                freq *= 2;

        period =  Timer_Base_Freq/freq+0.5;

        timer[TimerPeriod] = period;
        timer[TimerGlobalCtrl]  = SRC_INT | mode | FUNC_TIMER | RST_START;

        return Timer_Base_Freq/period;
}



/*****************************************************************************\
\*****************************************************************************/

/* float      Base_Frequency; */

void Set_Sample_Rate(float sample_rate)
{
        float Sample_Rate;

        Sample_Rate = Set_Timer(1, sample_rate, PLS_MODE | INV);
        /*using timer1 for A/D conversion because Timer0 doesn't
        work for PLS_MODE*/

}


void Set_Filter_Cutoff(float filter_cutoff)
{
        float   Filter_Cutoff;
        float   filter_freq = 100*filter_cutoff;
```

```
        Filter_Cutoff = Set_Timer(0, filter_freq, CLK_MODE);
        /*It was Timer0 before*/
}


/* Read the converted data from the address of ADC converter*/
long A2D(int i)
{
        return ADC[i]>>20;
}

/* Functions used to enable or disable interrupt */

void Enable_Int1(void)
{
        asm("   OR 2h,IE");
}

void Disable_Int1(void)
{
        asm("    AND ~2h, IE");
}

void En_Interrupts(void)
{
        asm("    OR 2000h, ST");
}

void Dis_Interrupts(void)
{
        asm("    AND ~2000h, ST");
}

void Clear_Intr_Flag(void)
{
        asm("    AND    ~2h, IF");
}

/*Buffer control functions*/


void Clear_Buffer(long buf_len)
{
        HjxHead = 0;
        HjxTail = buf_len;
        Buffer_Full=0;
        flag=0;
}


void Buffer_Put(float z)
{

        if(HjxHead!=HjxTail)
```

```
        {
                HjxBuffer[HjxHead]=z;
                HjxHead++;

        }
        else
        {
                HjxHead=0;
                HjxBuffer[HjxHead]=z;
                HjxHead=1;
        }

}


int     Buffer_Empty(void)
{

        return HjxHead=HjxTail;
}
```