6 Phase-locked-loop

The phase-locked-loop algorithm that will be used is shown in figure 5. It is derived directly from the discussion in section 5 of the Functional Specifications document, SDE 106. The phase and frequency controller determines whether fRes is within the predefined window and whether fRes and fInvActual are in phase. It outputs a command frequency signal, fInv, to the SVM timing algorithm. Changing fInv is the only way to affect the frequency or the phase of the SVM output wave. Compare this figure to figure 2 of SDE 105.

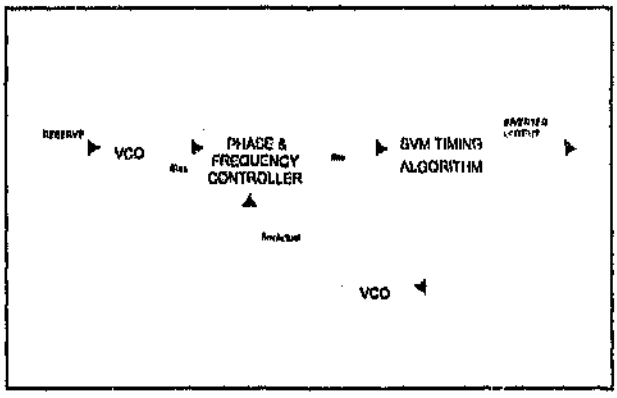


Figure 5: Phase-locked-loop operation interfacing with SVM

We now proceed to study in detail how to implement the phase detection and phase reduction mechanisms of the phase and frequency controller in a software algorithm.

6.1 Phase Difference Reduction Mechanism

Direct frequency tracking can easily be peformed by having fInv to simply be equal to fRes, the Reserve frequency signal. Phase tracking is more delicate and could be achieved by allowing fInv to jump between predefined limits. The limits will be so chosen as to increase the frequency difference between fRes and the actual inverter frequency, fInvActual. In other words a frequency difference is used to implement a phase reduction mechanism. The idea being that the faster waveform will eventually catch up on the slower one. In an out of phase condition, if fRes is lower than the nominal value, fNominal, then fInv should be changed to the higher frequency limit, fMax. If instead fRes is higher than fNominal, then fInv should be changed to the lower limit fMin. In the first case fInvActual is increased to a

established in section 4.3 of the Functional Specifications, where the mapping between the sectors and the switching vectors was listed. As far as the design is concerned, this will mean that at each sampling interval the sector location of Us*, is to be established prior to deciding upon the switching vectors. Note that states U0 and U7 always occur, irrespective of the sector.

Also, just like in the case of the switching times, the switching vectors order are to be reversed after each sampling interval, except at a sector transition.

A switching vector in that instance will simply be mapped to the actual physical state of the output port of the microcontroller. As an example, if port 1 is chosen as the output port, the voltage levels at port 1 for a switching state U1 (100 011) will be as shown in table 1.

Table 1: Mapping of Switching states onto the output port

PIN	P1.5	P1.4	P1.3	P1.2	P1.1	P1.0
LOGICAL STATE	1	0	0	0	1	1
VOLTAGE LEVEL	+5V	GND	GND	GND	+5V	+5V

5.7.2 Identified Classes

The responsibility of issuing the right switching vectors in the right order can be assigned to a new class, switchingVectors, with the following features:

switchingVectors

Responsibilities:

- selects the correct switching vectors according to the sector location of Us*
- choose their correct order of occurrence according to sequence

Collaborators:

- As established previously, switchingVectors collaborates with sequenceMonitor and sectorMonitor
- Like switchingTimes it needs to make its results accessible to dataRelay

switching times, the switching times too should follow the same pattern. The following sequences are identified for the switching times.

tOHalf - ta - tb - tOHalf	:Forward Switching Sequence
tOHalf - tb - ta - tOHalf	:Reverse Switching Sequence

5.6.3 Identified Classes

As shown above, the determination of the switching times involves several activities and as such is best assigned to a class, switchingTimes, which will have the following CRC representation:

switchingTimes

Responsibilities:

- retrieves values from the look-up-table
- computes the switching times ta and tb
- assign them to variables tjTemp and tkTemp in the correct order
- computes the time spent, tOHalfTemp, in the state uZero

Collaborators:

- As established before, class switchingTimes collaborates with classes angleDisplacement, sequenceMonitor, lookUpTable and vectorMagnitude
- In addition, it should make its computed values available to the class responsible of linking the main cycle to the interrupt cycle. We shall name this class, dataRelay

5.7 Switching States

5.7.1 Description

At any sampling interval, the switching states will depend upon the particular sector in which the desired switching vector Us* is located. This was already

working with floating type variables. Simple floating point operations can take as much as ten times more execution time than the integer type equivalent operations. A similar prolonged microprocessor time is required to perform any sophisticated operation like "floor" or "fmod".

From those observations the following decisions are made:

a. The switching times are to be precalculated, using a usMag value of 1, for a predetermined number of values of alpha, covering a full sector. Those values are to be calculated in the initialisation period and are to be stored in an array which can be accessed by the main cycle. In the main cycle, those values will be scaled according to the value of usMag calculated at each sampling interval. Note that the RAM size of the microcontroller will set a limit to the possible number of subdivisions of a sector. We name the number of subdivisions of a sector, maxSectorDiv.

b. Use of floating point numbers are to be avoided.

c. Only the simplest C functions are to be used.

d. Segmentation of the program into functions is to be kept to a minimum.

e. Compilation is to be performed using the speed option, to reduce the latency accompanied by function calls.

f. Parameter passing within time critical loops should be discouraged. Instead global variables are to be used in those instances.

Many of the points identified above, conflict directly with good programming practice. However, as this exercise showed, this is a regretful but unavoidable consequence of real-time applications and the limitations imposed by the microcontroller and its compiler. In order to compensate for the inevitable lack of clarity that will result from the above decisions, additional effort is to be spent on the documentation support.

5.6.2 The Switching Time Order

As shown in section 4.8 of the Functional Specifications, SDE 106, the switching sequence reverses after each sampling time, except at sector transitions. In order to ensure that the alternating switching vectors are assigned to the right

following structure:

sequenceMonitor
Responsibilities: - updates sequence Collaborators: - To know when a sector change occurs, sequenceMonitor will have to collaborate with class sectorMonitor. sequenceMonitor will be used by both classes switchingTimes and switchingVectors in the determination of the order to output their respective parameters.

5.6 Switching Times

The switching times are computed from the equations (1), (2) and (3) of section 4.7 of the functional specifications. Those equations are reproduced here for convenience.

$$ta = 9/\Pi^2 * usMag * tSampling(Cos\alpha - 1/\sqrt{3} * Sin\alpha) \quad (1)$$

$$tb = 6\sqrt{3}/\Pi^2 * usMag * tSampling * Sin\alpha \quad (2)$$

$$t0 = tSampling - ta - tb \quad (3)$$

5.6.1 Practical Observations About Process Time

Before deciding on the best method of performing the calculations, a simple exercise was set up. A small program (SDE0201.C) was written to perform the calculations above. The program was run on the simulator to obtain an idea of the ability of the compiler and the microcontroller to handle the calculations. The code listings are available in appendix A. The exercise quickly reveals the impracticality of doing those calculations on-line. It takes the microcontroller several seconds to perform the three equations above, while the allowable time, as set by the sampling time, is of the order of hundreds of microseconds! Furthermore the exercise shows the poor performance of the microcontroller when it comes to

angleDisplacement

Responsibilities:

- determines the angular displacement phata
- determines the angular displacement alpha

Collaborators:

- will need flnv and counter for the calculation of angular displacement. flnv can only be provided by the class responsible fo. the phase lock-loop. counter will be provided by class counterMonitor

- on the other side, angleDisplacement will be used by class switchingTimes which needs alpha to retrieve the correct values from the look-up-table. angleDisplacement is also expected to be used by counterMonitor when determining whether counter will be reset to 0 or 1.

5.6 Sequence Update

5.5.1 Description

From the discussion in section 4.8 of the Functional Specifications, SDE 106, we conclude that the sequence can take only two values, +1 for a positive sequence and -1 for a negative sequence.

Also from section 4.8 of SDE 106, emerges the importance of ensuring that sequence toggles between those two values after each sampling interval, except at a sector transition when it should remain unchanged.

5.5.2 Identified Classes

The above task can be assigned to a class sequenceMonitor that will have the

Us*. Closely coupled to those activities is the process of updating counter. Those activities are captured in three classes: angleDisplacement, sectorMonitor and counterMonitor respectively.

sectorMonitor

Responsibilities:

- Monitors and returns the sector location of Us*
- Indicates a change of sector

Collaborators:

- will require photo from angleDisplacement and will be used by counter and the class responsible for determining the switching vectors, which we will name class switchingVectors

counterMonitor

Responsibilities:

- updates counter
- resets counter

Collaborators:

- collaborates with angleDisplacement

- uses sectorMonitor to know when to reset counter

plane. At this point counter has to be reset in order to limit its value which, if not checked, will cause overflow errors. By resetting counter, the time tActual and the angle pheta will automatically also be reset. The reset value of counter will depend upon the location Us* lands after the sector 6 to sector 1 transition. One solution is to assume that it will always land on the U1 switching vector, in which case counter will take the value zero. However in order to improve the accuracy of the algorithm, we decide to include another variable, gamma, which corresponds to half the minimum angular displacement, lamda , taken at the nominal Inverter output frequency.

lamda = (360*fNominal*tSampling) (2)

gamma = lamda/2 (3)

If the desired vector, Us*, lands anywhere between an angle 0 and gamma, then counter is reset to zero, else counter takes the value 1. We assume that the position after a complete revolution will be less than twice alphaMin. This assumption is reasonable for the kind of small deviations in flnv that are allowed in the present case.

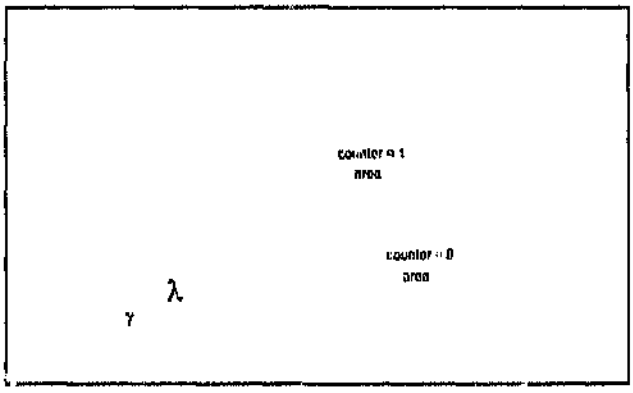


Figure 4: Reset values of counter

5.4.2 Identified Classes

Two major activities are identified in this section, both related to the location of the desired vector Us*. The first one is about the monitoring of the angular displacement of Us*, the second is about the monitoring of the sector location of

vectorMagnitude

Responsibilities:

- calibrates usMag
- computes and returns the current value of usMag

Collaborators:

- Since usMag has a direct bearing upon the switching times, like lookUpTable, it will collaborate with class switchingTimes.
- It will require a knowledge of vInvActual to perform the calculations

5.4 Sector Location

5.4.1 Description

sector is derived directly from a knowledge of the angular displacement, phata, measured with respect to the first switching vector position, U1. Phata itself depends on the the command inverter frequency, fInv, and on the time, tActual elapsed from when the desired vector was last at the first switching vector position, U1.

$$phata = 2\Pi * fInv * tActual \qquad (1)$$

Time tActual is incremented by the value of the sampling time, tSampling, after each sampling interval or each run around the main cycle loop. This can easily be monitored by a counter, counter, that will keep track of the number of runs through the loop.

A sector corresponds to a 60 degrees interval, therefore every 60 degree increment of phata, starting at vector U1, will cause a sector change.

The sector 6 to sector 1 transition requires special attention. This transition occurs when the desired vector, Us*, has done a complete revolution around the complex

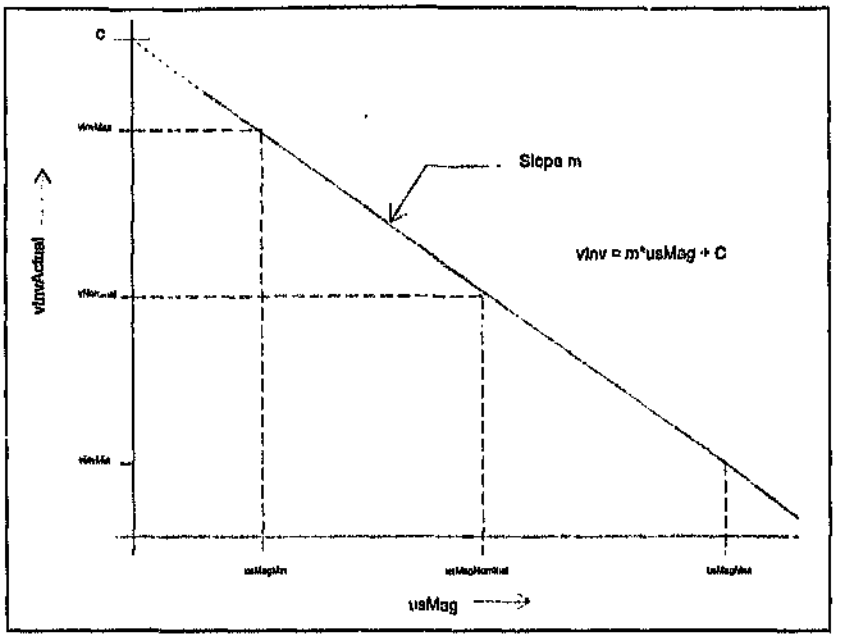


Figure 3: vInvActual v/s usMag

The range of vInvActual is vNominal +/- 10%. At a switching frequency of 6.25kHz, taking vNominal as 230V gives vInvMin = 207V and vInvMax = 253V

Substituting those values, together with usMagMin = 0.5 and usMagMax = 0.866604, into the equations above yields usMagNominal as 0.685898

5.3.4 Identified Classes

The process of determining the value of usMag can be captured by a single class, vectorMagnitude, which will have the following features:

5.3.2 Choosing A Lower Limit for usMag

Theoretically usMag could be allowed to drop to zero, at which point all IGBTs will naturally be turned completely off. However for an efficient system we do not want the IGBTs to spend too much time in the idle state (off states) at the nominal inverter output voltage, vNominal. The switching times are dependent upon the value of usMag as shown in equations (1), (2) and (3) of section 4.7 of the Functional Specifications, SDE 106. The higher the value of usMagNominal, which is the corresponding value of usMag at vNominal, the less time will be spent in the off states. We observe that since usMagNominal lies between usMagMin and usMagMax, the higher the value of usMagMin the higher usMagNominal will be. Taking this factor into consideration we choose usMagMin to be 0.5.

5.3.3 Algorithm to Determine the Correct Value for usMag

We have established that usMag is proportional to vInvActual. We also need to ensure that the on-times decreases as the vInvActual increases and increases when vInvActual decreases. This brings us to the relationship depicted in figure 3.

The relationship between vInvActual and usMag is given by,

vInvActual = m*usMag + c (1)

or usMag = (vInvActual-c)/m (2)

where the slope, m, is given by,

m = (vInvMax - vInvMin) / (usMagMin - usMagMax) (3)

and the intersection with the vInvActual axis, c, is given by,

c = vInvMax - m*usMagMin (4)

5.3 The Magnitude of the Desired Vector, usMag

The magnitude of the desired vector, usMag, allows control over the magnitude of the Inverter output voltage. It should be related to the actual Inverter output voltage, vInvActual itself, in order to correct for any deviation of the latter from the nominal value, vNominal. Those deviations might be caused by fluctuations in the DC supply, or changes in the load applied to the Inverter.

Theoretically usMag can vary between 0 and 1, but as we shall see other factors force an adjustment of this range.

5.3.1 Choosing A Higher Limit for usMag

As usMag increases, so does tj and tk. That means that t0Half, the time spent in the zero switching states U1 and U7, will decrease. As will be discussed later in section 7.1, t0Half itself cannot be allowed to decrease below tUpdate, the time required to implement the last assignment operations in the *main cycle*. This minimum value for t0Half, t0HalfMin, will set a higher value for usMag. A quick exercise on the simulator reveals that tUpdate is equal to 6 microseconds. Including a safety margin of 1 microsecond gives t0MinHalf as 7 microseconds.

Using equations (1), (2) and (3) of section 4.7 of the Functional Specifications, SDE 106, to solve for usMag gives:

$$usMag = (tSampling - 2*t0Half)/(tSampling(Ka(Cos\alpha - 1/\sqrt{3}Sin\alpha) + KbSin\alpha)) \qquad (1)$$

where $ka = 9/\Pi^2$

and $kb = 6\sqrt{3}\,\Pi^2$

We observe that t0Half is a minumum at the middle of a sector, i.e at alpha = 30 degrees. From this consideration we can derive the higher limit for usMag, usMagMax, as follows:

$$usMagMax = (tSampling - 2*t0HalfMin)/(tSampling(Ka(Cos30 - 1/\sqrt{3}Sin30) + KbSin30)) \qquad (2)$$

Equation (2) shows that usMagMax depends on t0HalfMin and tSampling. Both those parameters will be constant for a given design and therefore so will be usMagMax. For example taking t0HalfMin as 7μs and tSampling as 160μs (equivalent to a sampling frequency of 6.25kHz) gives a value of 0.866604 for usMagMax.

Appendix B: Code Listings For SDE0601.M

```
% phaselI.m
% Author: H.B
% Creation Date: 2/11/95
% Revision 0.01

% program to illustrate an active phase lock mechanism.

% Here yInv is the inverter output and yRes is the Reserve supply.
% The frequency window is taken to be 49-51Hz, with 50Hz as the nominal frequency.
% The program considers the case where the Reserve frequency has gone out of limits
% (causing fInv to be set to the nominal value) and now is back within the limits, but
% is 180 degrees out of phase with the inverter output.
% The aim is to reduce the phase difference to a minimum.
% The program shows that this can be achieved by increasing the frequency difference
% (by changing fInv) between yInv and yRes to a maximum without going outside the frequency
% limits of yInv. The idea is that the difference in frequency will cause the faster
% wave to slowly catch up on the slower wave.
% The convention used is the following: if fRes is less or equal to 50Hz, then fInv is set to
% 49Hz (lower limit) else fInv is set to 51Hz (higher limit)



% The worst case will be when the two waves are 180 degrees out of phase, i.e the
% maximum phase difference to be reduced, and with fRes being 50 Hz, i.e the less frequency
% difference margin available to perform the task.


fInv=49;            % fInv set to the lower limit because fRes =< 50
wInv=2*pi*fInv;
phiInv=90/180*pi;


fRes=50;
wRes=2*pi*fRes;
phiRes=270/180*pi; % phiInv and phiRes ensure a 180 deg phase diff. at the start

t=[0:1E-3:600E-3];

yInv=sin(wInv*t)+phiInv);
yRes=sin(wRes*t)+phiRes);

plot(t,yInv,'y',t,yRes,'m');
grid;
xlabel('time in seconds');
ylabel('magnitude')
```

Appendix A: Code Listings For SDE0201.C

```
/*                          SDEC0201.C                          */

/* Program to calculate ta, tb and t0 of the SVM equations  */

/* Author: Hansraj Bapoo */
/* Creation Date: 12/09/95 */
/* Revision Number 0.01 */

#pragma language = extended

#include <math.h>

void main (void)
{

#define tSampling 1E-3                  /* fSampling = 1 kHz */
#define pi 3.14159265359
#define ka 8.66366515E-4
#define kb 9.9E-4

float  alpha, ta, tb, t0;


ta= ka*(cos(alpha)-(1/sqrt(3))*sin(alpha));
tb= kb*sin(alpha);
t0= tSampling-ta-tb;

}
```

time of the IGBTs, lead to the same state transitions at the IGBTs. As such, implementing the one function, by necessity, implies doing the other, although each of them is targetted at a different aspect of the Inverter.

Table 2: Effect of the hardware-built delay on the switching state transitions

State of the switches before the next command signal	1	0	0	0	1	1
Next command signal	1	1	0	0	0	1
State of switches during tDelay	1	0	0	0	0	1
State of switches after tDelay	1	1	0	0	0	1

8 Transitory States and Igbt Turn-Off Time Considerations

As observed in section 4.8.3 of the Functional Specifications document, SDE 106, transitory states are needed in order to ensure a Gray code switching sequence. Another important concern brought forward in section 3.1 above, was about the turn-off time of the IGBT and the precaution to take to avoid this time delay to cause short circuit of the DC bus.

A close look at the transitory states in section 4.8.3 of the Functional Specifications reveals that by their very existence the transitory states exclude the possibility of having a short circuit on the DC bus due to the switching actions. That is, if the transitory states are implemented correctly, in any leg of the Inverter, there will always be one switch that will be turned off completely before the other is switched on.

Those transitory states can either be implemented by simply including instructions in the timer subroutines to switch the output port in the corresponding states or else through a hardware arrangement. Owing to the critical aspect of those two issues, especially that of the turn-off time, we choose the hardware solution for this design attempt.

The hardware arrangement consists of a simple time delay circuitry that only operates on 'switch on' command signals. The 'on' signals are always delayed by tDelay which is slightly more than the turn-off time of the IGBT, while the 'off' signals are unaffected by the circuitry. A delay in an 'on' signal means that the actual state of the switch the signal is directed to, will remain unchanged during the period tDelay. This arrangement is common to most drive circuits and is a very effective way of allowing 'off' signals to execute before 'on' signals, thus cancelling the turn-off time effect. Table 2 shows the effect of delaying the 'on' signals for the case of signal command 110 001 (U2), assuming the inverter is in the state 100 011 (U1) when the signal command is issued.

Comparing table 2 below with table 3 of section 4.8.3 of the Functional Specifications document reveals that the state of the switches during tDelay is identical to the transitional state, U1-2, attached to the same state transition (U1-U2).

What this whole study reveals is that implementation of the transitional states and implementation of a delay mechanism to counteract the effect of the turn-off

dataRelay

Responsibilities:

- assigns correct values to the switching times parameters tj, tk and t0Half
- do the same for the switching states parameters uj, uk and uZero

Collaborators:

- will obtain the temporary switching times parameters tjTemp, tkTemp and t0HalfTemp from class switchingTimes
- the temporary switching states parameters ujtemp, uktemp and uZeroTemp will be obtained from class switchingVectors
- dataRelay will have to interface with the interrupt classes in some way

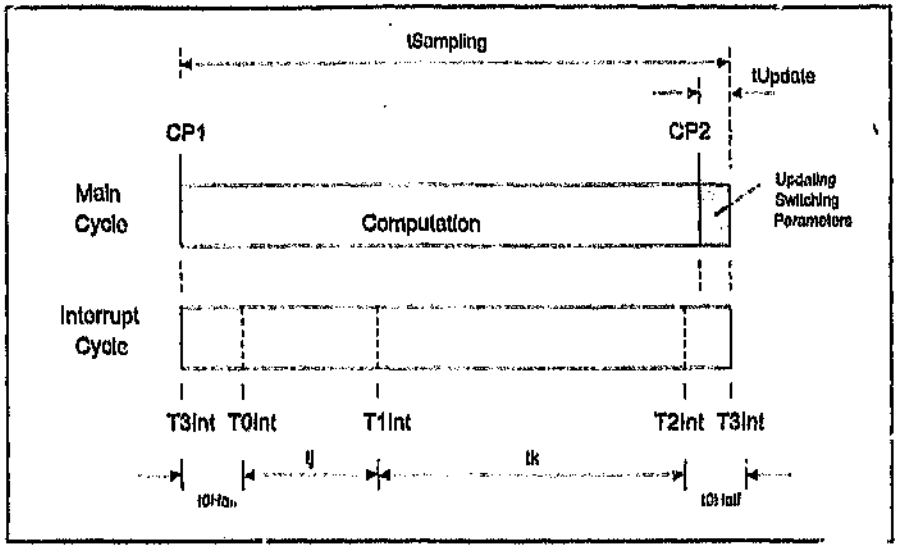


Figure 8: Synchronising the *main cycle* to the *interrupt cycle*

This approach however puts a restriction on the smallest value t0Half, can take. t0Half cannot be allowed to take any value below tUpdate, the minimum time needed for the assignment of new values to the switching parameters to occur. This condition is necessary to ensure that by the end of a sampling interval all the activities - in the *main cycle* - related to this interval, have been completed. The lower limit on t0Half is ensured by usMagMax as explained in section 5.3.1 above.

7.2 Identified Classes

The task of passing switching data (times and vectors) to the interrupt cycle can be assigned to the class dataRelay, whose specifications are summarised below.

7 Synchronising the Main Cycle to the Interrupt Cycle

7.1 Description

In order to ensure meaningful results, i.e. a genuine SVM output, the main cycle should be synchronised to the interrupt cycle. It is essential that the interrupt cycle be fed the correct values at the correct time.

Synchronisation is achieved by using two check points in the *main cycle*. The first one, CP1, is at the entry point of the main cycle itself, i.e. it controls access to the main cycle. The condition for access being that the timer T0 must have been started. Prior to that, access to the main cycle is denied. In this way the start of the *main cycle* is synchronised to the timerT3InterruptRoutine.

The necessity of the second check point arises purely from a logical consideration. It has been agreed that at the beginning of each sampling interval a new set of values for the switching parameters will be available to the interrupt cycle. However during this same sampling interval other switching values will be calculated. It is essential that the interrupt cycle uses the values provided right at the beginning of the sampling interval and not those calculated during the interval itself. Failure to respect this condition will result in an output containing mixed values, those for the actual interval and those for the next interval. In other words, we will fail to produce a PWM output.

A second check point and a dummy set of variables for the switching parameters can solve this problem. This is commonly known as the double-buffering method. The dummy switching vectors variables, ujtemp, uktemp and uZeroTemp and the dummy switching times variables, tjTemp, tkTemp and t0HalfTemp hold temporary values for their real counterparts. They are the only ones that appear during the whole computation section of the main cycle. Only in the last section of the main cycle are those temporary values assigned to the real switching parameters conterparts, uj, uk, uZero, tj, tk, and t0Half, all used by the interrupt cycle. The second check point, CP2, appears at the beginning of this last assignment section. Access to this last section is only permitted once the last subroutine of the interrupt cycle, timerT2InterruptCycle, has been executed. In this way there is no chance of the interrupt cycle using parameter values calculated for the next sampling interval. A typical scenario is portrayed in figure 8.

6.4 Identified Classes

Two important abstractions are identified in the phase-locked-loop process. The first is about phase difference detection, to which the class phaseDetector will be assigned. The second abstraction is about phase difference reduction upon an out of phase condition, to which the class phaseDiffReducer will be assigned.

phaseDetector

Responsibilities:

- evaluates the voltage difference and compares the wave directions, between Reserve and Inverter output, in order to determine whether an out of phase condition exists or not.

Collaborators:

- will have to collaborate with classes vSensor and fSensor to obtain the voltage and frequency values of Reserve and Inverter. It can as well pass the value of vInvActual to vectorMagnitude

phaseDiffReducer

Responsibilities:

- sets finv in such a way as to cause a reduction in phase between Inverter and Reserve.

Collaborators:

- None at this stage, but it is evident though that finv as set by phaseDiffReducer, should be available to angleDisplacement in some way

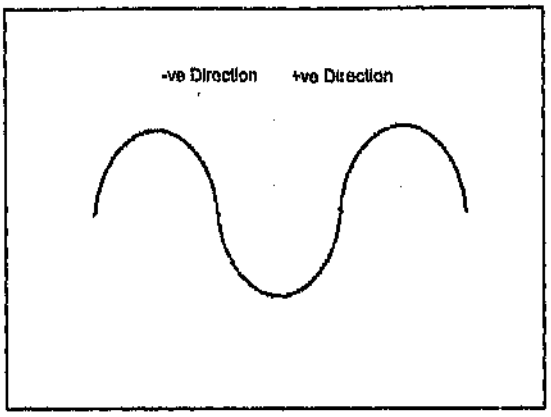


Figure 7(a): Wave direction convention

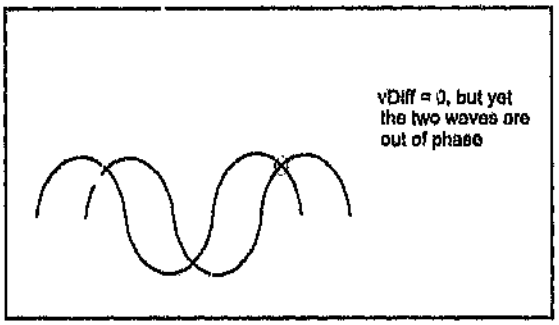


Figure 7(b): Danger of using vDiff only as an indication of phase difference

6.3 The Zero Crossing Point

It was established in the functional specifications that transfer from Inverter to mains, under an out of phase condition, can only be allowed at the next zero crossing of the Reserve supply (see section 5.3, SDE 106).

It is important to determine, for the software, what constitutes a zero crossing point. Strictly speaking the zero crossing point is the point when vRes is equal to zero. However since vRes is sampled at a fixed rate, this point could be missed by the microcontroller. To increase the chance of the sofware of determing when the zero crossing point is reached, we give this point a range from zero to vResCross, where vResCross is a very small value, say 0.01 of vNominal. Any value of vRes within this range will be intepreted by the software as a zero crossing point.

This exercise demonstrates the frequency difference technique as an excellent method of reducing phase difference in a minimum of time.

6.2 Phase Detection Mechanism

The proposed way of determining phase difference between Reserve and Inverter output is by working out the voltage magnitude, vDiff between the two and by comparing their wave directions, vResDirection and vInvDirection respectively. Wave direction itself can be determined by the difference between two consecutively sampled values of the wave, vInv1 and vInv2 for the Inverter output and vRes1 and vRes2 for the Reserve. In other words,

vDiff = vInvActual - vRes (1)
vResDiff = vRes2 - vRes1 (2)
vInvDiff = vInv2 - vInv1 (3)

In each of the cases (2) and (3), a positive difference implies a positive direction and a negative difference means a negative direction. The convention used for the direction is shown in figure 7(a).

vDiff and the direction parameters are complemetary in determining phase difference. To use only vDiff as a representation of the phase difference is not enough and might lead to erronous assumptions. This is obvious in figure 7 (b) where the two waves are out of phase but yet vDiff is zero at each intersection points. If the waves are sampled at any of those intersection points, they will incorrectly be assumed to be in phase.

In terms of those new parameters, an in-phase condition is a state where:

a. both Reserve and Inverter have the same wave directions
b. vDiff is smaller than a limiting value vDiffMax

vDiffMax is taken as 5% of vNominal. Under those conditions, on an Inverter to Reserve transition, the worse step change that can be observed at the output will be only about 12V.

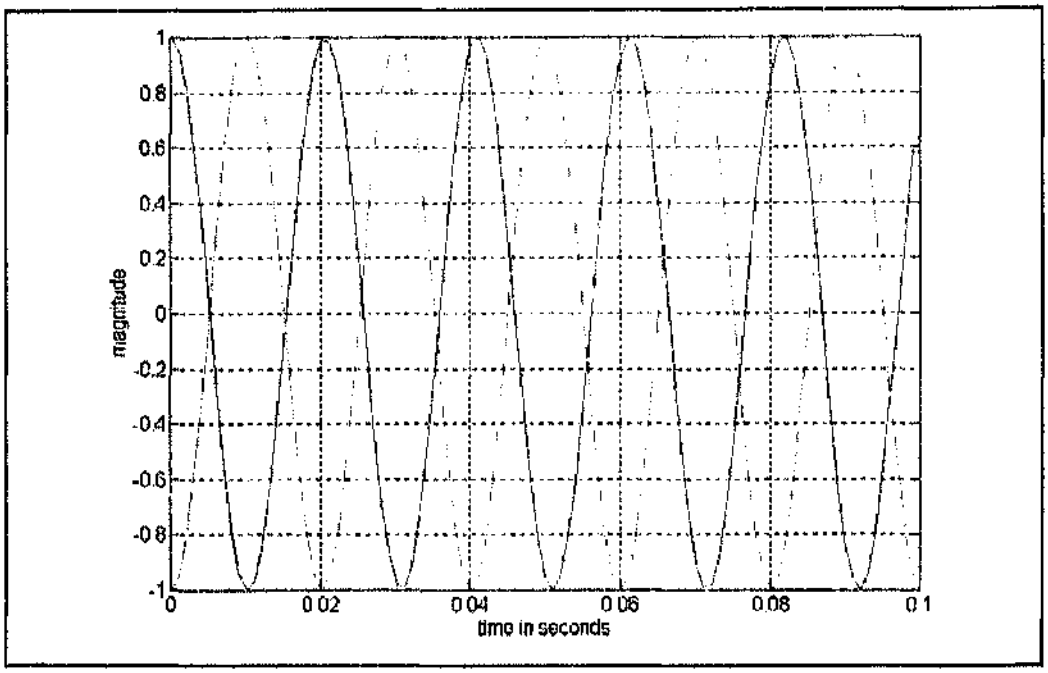


Figure 6(b): Illustrating the phase difference reduction mechanism - selected section: out of phase condition (Inverter Voltage - Blue, Reserve Voltage - Green)

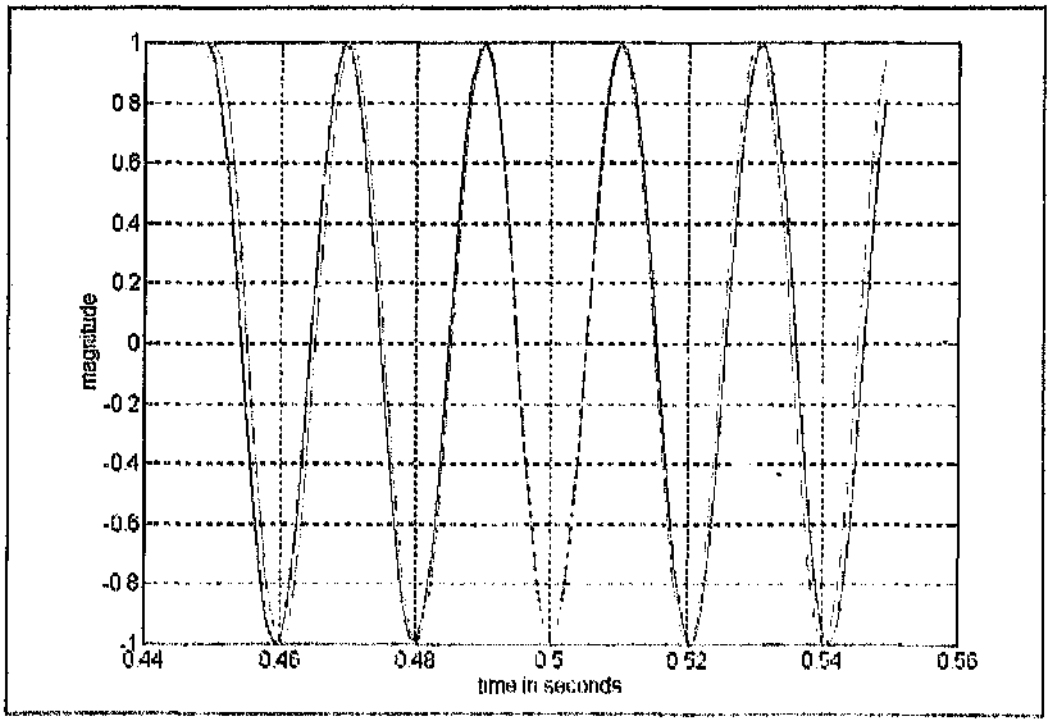


Figure 6(c): Illustrating the phase difference reduction mechanism - selected section: in-phase condition (Inverter Voltage - Blue, Reserve Voltage - Green)

below in figure 6 and the code listings are available in appendix B. The program considers the case where the Reserve frequency has gone out of limits (causing fInv to be set to the nominal value) and is now back within the frequency limits but is 180 degrees out of phase (worst case) with the Inverter output. fRes is considered to be 50Hz. In order to create the maximun difference in frequency between Reserve and Inverter, fInv is set to the lower frequency limit, fMin = 49Hz. Figure 6(a) shows the result over a time span of 0.6 seconds. As depicted closely in figure 6(b) a phase difference of 180 degrees exists between the two waves at start. This phase difference is quickly reduced as a result of the frequency difference created, and after only 0.5 seconds the waves are exactly in phase. The fact that the two waves are seen to drift apart after the 0.5 seconds point is only because the simulation only looks at the phase catching action and not at the whole phase-locked-loop mechanism. In the real algorithm the phase reduction mechanism will be discontinued as soon as phase-lock is reached.

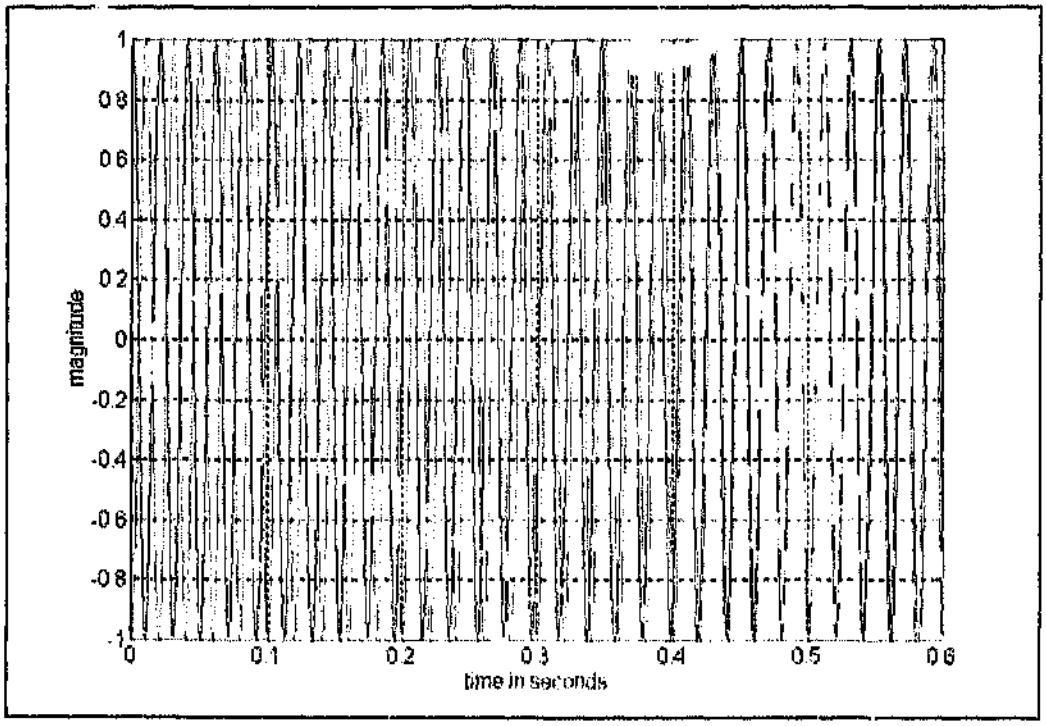


Figure 6(a): Illustrating the phase difference reduction mechanism (Inverter Voltage - Blue, Reserve Voltage - Green)

startUp

Responsibilities:

Contains a few subprograms responsible for setting the system at start up and for generating an NMI routine upon an overload condition.

Operations:

- void setRegs ( ) [M]
- void computeKTerms ( ) [M]
- void startIntCycle ( ) [M]
- void overloadInterruptRoutine ( ) [M]

Attributes:

None.

Listing

```
void  startUp::setRegs ( )
{
// H83048 Registers initialisation
ITU_TCR0 = 32;
..... etc.
}

void  startUp:: computeKTerms ( )
{
// Compute the constant terms used in the program
ka = 9/(pi*pi);
..... etc.
}

void  startUp:: startIntCycle ( )
{
ITU_TSTR = 8;     // start timer T3
}

Interrupt [NMI] void startUp::
overloadInterruptRoutine ( )
{
for (; ;)
P1DR = 0;
}
```

that it contains a number of common classes.

2.2.2 Initialisation Class Decomposition

Referring to section 4 of part 1 of the Software Design Specifications, SDE 107, two main classes are identified:

- startUp
- lookUpTable

To co-ordinate the activities of those two classes a third class will be included, systemConfig, which will use both of the above classes.

The structure and the activities of those classes are defined below. Notice that the class startUp is a class utility, in the sense that it contains free subprogra , grouped together because of their common involvement in the start up activities.

identified during the analysis phase. A format similar to the CRC cards will be used, except that this time each class will be accompanied with C++ listing to provide elementary executable releases. Those fragments of codes will help to bridge the gap between the conception of the object structure and the actual coding which is finally performed in C. Each operation will be adorned with an identifier, [M], [S], or [I], standing for modifier, selector or iterator.

2.2.1 Top-Level Class Decomposition

Following the decomposition performed in part 1 of the Software Design Specifications, SDE 107, four main clusters of abstractions emerge:

- The *Interrupt cycle*
- The Initialisation Procedure
- Management of the phase-locked-loop
- The *main cycle*

Each of the above group can be represented as a class category, related as in figure 2 below:

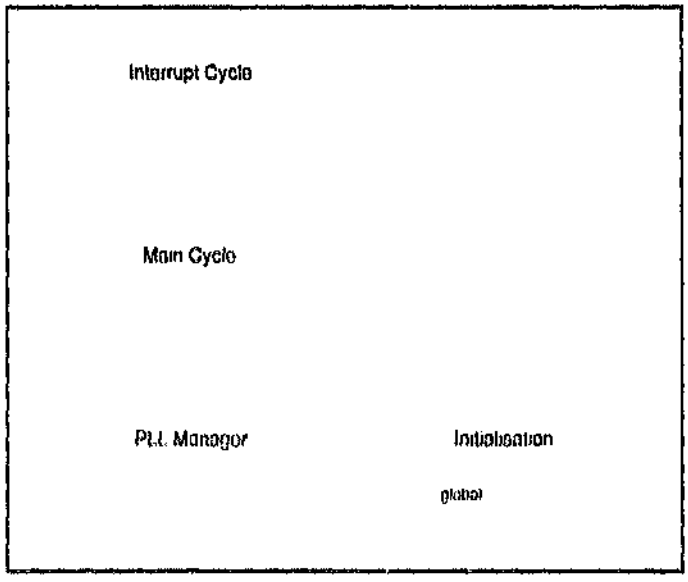


Figure 2: Class Diagram - Top Level

Note that class category initialisation is adorned with the keyword "global", meaning

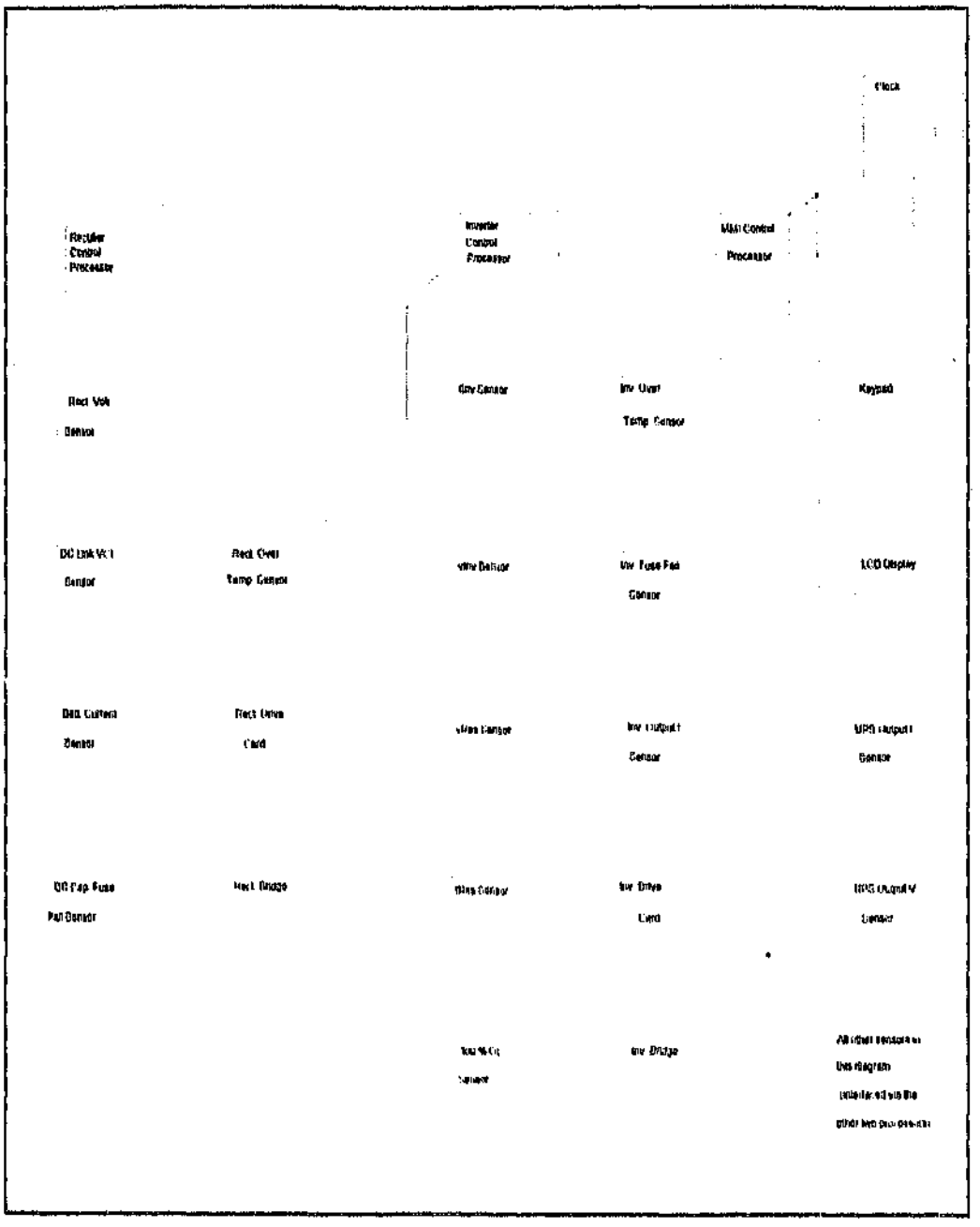


Figure 1: Simplified Process Diagram - UPS level

2.2 Class Structure

The class decomposition will follow from the abstractions and the scenario

2 Static Model

2.1 Process Architecture

The first step is to define the hardware on which the software will execute. In this exercise, the system shall be captured at a level of abstraction that exceeds the boundaries of the actual design. This offers a better perception of the system as a whole, as it locates the present design within the broader context of its environment.

The general hardware aspects of the UPS (sensing and control features) were discussed in the Functional Specifications, SDE 106. This exercise serves to link all those components into a framework. The proposed hardware platform can be summarised as follows:

a. The control of the UPS shall be carried out by three control boards, each with an H8/3048-class processor. One processor will be assigned to the rectifier control, the second one to the inverter control, and the last one will be responsible for the LCD display and the MMI.

b. Sensing shall be carried out as in appendix 1 of the Requirements Analysis document, SDE 104.

c. Selection of either Reserve or Inverter at the UPS output, shall be performed by the Static Switch. This function does not show the kind of complexity inherent in the other sections discussed and as such it does not require the power of a microcontroller.

Figure 1 depicts the distribution of the hardware elements in the system. Each of those elements were described in SDE 104. As is typical in process diagrams for production systems, we do not (and cannot) show every device and connection; e.g. the Static Switch and its connections were omitted. The diagram focuses on the microcontrollers and in particular on the inverter control aspect, which is the primary objective of this project.

1.5.5 Other Documents

a. SDE 105, Requirements Analysis
b. SDE 106, Functional Specifications
c. SDE 107, Design Specifications - Part 1

1.6 Assumptions

It is assumed that the reader is familiar with:

- Inverter systems and the concept of Space Vector Modulation
- Object Oriented Design and with Booch notation

1.7 ISO 9001 and ISO 9000-3 Requirements Traceability

- ISO 9001 (1994) Clause 4.4 - Design Control
- ISO 9000-3 (1991) Clause 5.6.2 - Design

1.4 Audience

The audience for this document comprise the various stakeholders of the SEAL, including:

- Full-time staff members of SEAL
- All full-time and part-time post-graduate students associated with the SEAL
- Members of the SEAL Management Board
- Head of the Department, Electrical Engineering
- Individuals who will perform internal and external surveillance audits of the SEAL Quality Management System
- Engineering Manager, Meissner
- Product Developer
- Product Manager

1.5 Applicable Documents

1.5.1 Specifications

None

1.5.2 Standards

a. SEAL QMS Document Creation Template, QS 002, Revision 0.01, 5 April 1994

b. SEAL QMS Document Layout, Presentation and Typesetting Guide, QS 003, Revision 0.01, 1 April 1994

1.5.3 Procedures

None

1.5.4 Guidelines

None

1 Scope

1.1 Introduction

Building on the analysis exercise performed in part 1 of the Software Design Specifications, SDE 107, part 2 proceeds to develop fully the models of Object Oriented Analysis and Design. The design completes the process of object-oriented decomposition and provides a logical, physical, static and dynamic models of the system under consideration. In addition, the design phase provides the skeleton of the C++ listing of each mechanism and scenario captured.

1.2 Purpose

The purpose of the design phase is to:

- Create an architecture for the implementation phase.
- Establish the common tactical policies that must be used by disparate elements of the systems.
- Identify the risk of each key architectural interface so that resources can be meaningfully allocated as evolution commences.

1.3 Definitions

sameDirection: direction comparison between Reserve and Inverter

temp: variable used to assist the swapping of ujTemp and ukTemp when there is a change of in the switching direction

vResBig: a positive number bigger than the highest value vRes can take, i.e. higher than vResMax

vResSmall: a negative number smaller than the smallest value vRes can take, i.e. smaller than vResMin

Change History

Configuration Control

Project:	SDES
Title:	Software Design Specifications - Part 2
Doc. Reference:	SDE 108
Created by:	H. Bapoo
Creation Date:	20 March 1996

Document History

Version	Date	Status	Who	Saved as:
0.01	96/03/20	Draft	H.Bapoo	\1995-13\TP\SDE108WP.001
1.00	96/07/18	Approved	H.Bapoo	\1995-13\TP\SDE108WP.100

Revision History

Version	Date	Changes
0.01	96/03/20	New Document
1.00	96/07/18	Update to revision number and status following Board approval and change of document format to comply with SEAL document presentation guide, QS 003

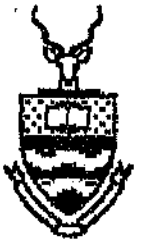
Management Authorisation

Version	Date	Status	Management Board Minute Reference
1.00	96/07/18	Approved	Section 2.4 of SDE 566

Change Forecast

This document will be updated each time there is a change in the design structure, especially when a change is made to the Object Analysis and Design archetypes of the project.

Table of Contents



SDES QMS

Software Design Specifications Part 2

Technical Product

Version 1.00

Document Status: Approved

but we choose to include all three of them, because the operations currentVoltage and currentFrequency are slightly more type-safe.

The class diagram for the three classes is shown in figure 6.

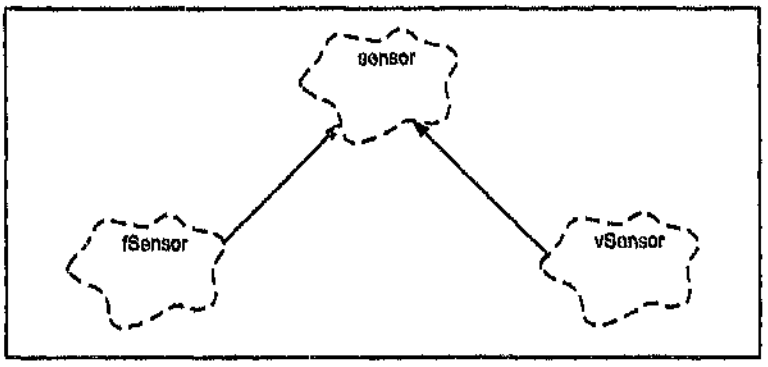


Figure 6: Class Diagram - Sensor Class Hierarchy

2.2.5.2 Evaluation of the Attributes Of the Desired Vector Us*

Four classes are involve in this activity:

- angleDisplacement
- counterMonitor
- sectorMonitor
- vectorMagnitude

Before we proceed into developing each one in detail, let us see how they relate to each other in figure 7.

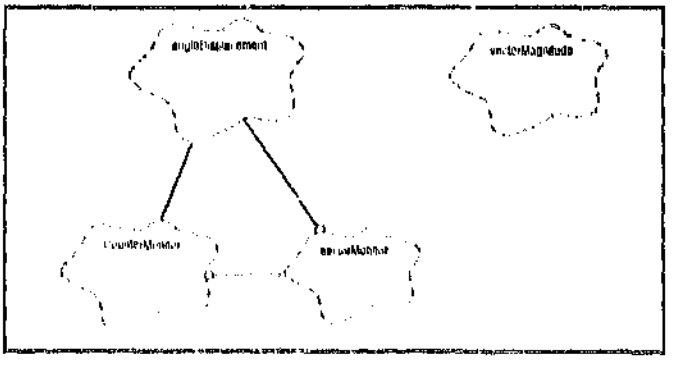


Figure 7: Class Diagram - Derived Attributes (preliminary configuration)

vSensor	Constructor:

Responsibilities:
 	
keeps track of the current voltage.
 	
Operations:
 	
- float currentVoltage ( ) [S]
 	
Attributes:
 	
float voltage;
	vSensor:: vSensor (unsigned int *vPort) : sensor (unsigned int *vPort)
	{ };
	
	Listing:
	
	float vSensor:: currentVoltage ( )
	{
	voltage = calculatedValue ( );
	return voltage;
	};

fSensor	Constructor:

Responsibilities:
 	
keeps track of the current frequency.
 	
Operations:
 	
- float currentFrequency ( ) [S]
 	
Attributes:
 	
float frequency;
	fSensor:: fSensor (unsigned int *fPort) : sensor (unsigned int *fPort)
	{ };
	
	Listing:
	
	float fSensor:: currentFrequency ( )
	{
	frequency = calculatedValue ( );
	return frequency;
	};

Although the calibration operation has been included in class sensor, this operation will not be carried out at this stage. Typically such an operation will be carried out via the keypad and display unit, where a value of actualValue will be entered for each sensor; actualValue being the value of the sensed parameter at the time of testing. To avoid possibilities of unpredictable behaviour while the unit has not yet been calibrated, actualValue and sensorSlope are initialised to zero. In other terms the inverter cannot run prior to the sensors being calibrated.

Notice that the operations currentVoltage and currentFrequency have been introduced. Those operations are semantically similar to the polymorphic function currentValue.

Three main areas of concern are recognised:

- sensing activities
- evaluating the attributes of the desired vector Us* (vector displacement and magnitude)
- Working the switching parameters (switching states and switching times)

Each segment will be developed separately.

2.2.5.1 Sensing Activities

Section 5.1.2 of part 1 of the Software Design Specifications, SDE 107, partitions the sensing activities into three classes, with the class sensor as an abstract superclass to the other two classes, vSensor and fSensor. Having separate classes to handle different sensing parameters, improves the typing and reduces the chances that the objects of those classes are interchanged.

Class sensor will have two operations, calibrateSensor to calibrate the sensor and currentValue to scale and to return the sensed parameter.

The structure for the three classes are developed below.

sensor	Constructor:

Responsibilities:
 	
attaches the sensor to an input port register, calibrates the sensor and returns the calculated value when prompted.
 	
Operations:
 	
- virtual void calibrateSensor (float initActualValue) [M]
- virtual float currentValue ( ) [M] & [S]
 	
Attributes:
 	
float actualValue = 0, sensorSlope = 0;
unsigned int *portRegister;
float currentValue;
	sensor:: sensor (unsigned int *portRegId)
	{
	portRegister = portRegId;
	};
	
	Listing:
	
	void sensor:: calibrateSensor (float initActualValue)
	{
	actualValue = initActualValue;
	sensorSlope = actualValue / *portRegister;
	};
	
	float sensor:: currentValue ( )
	{
	calculatedValue = *portRegister * sensorSlope;
	return calculatedValue;
	};

Notice that setTimerAndPort is declared as pure virtual in timerInterrupt. This is because, as pointed above, each of the classes will have a different setTimerAndPort operation and therefore it is best to leave the declaration of this operation to each of those classes.

timerT3InterruptRoutine, timerT0InterruptRoutine, timerT1InterruptRoutine and timerT2InterruptRoutine are iterators that permit the operations of their respective objects to be accessed in the right order.

Through class timerInterrupt it can also be observed how the switching parameters are passed to the *interrupt cycle* via the class dataRelay. This aspect will be discussed in more details shortly.

The class diagram for the *interrupt cycle* is shown below.

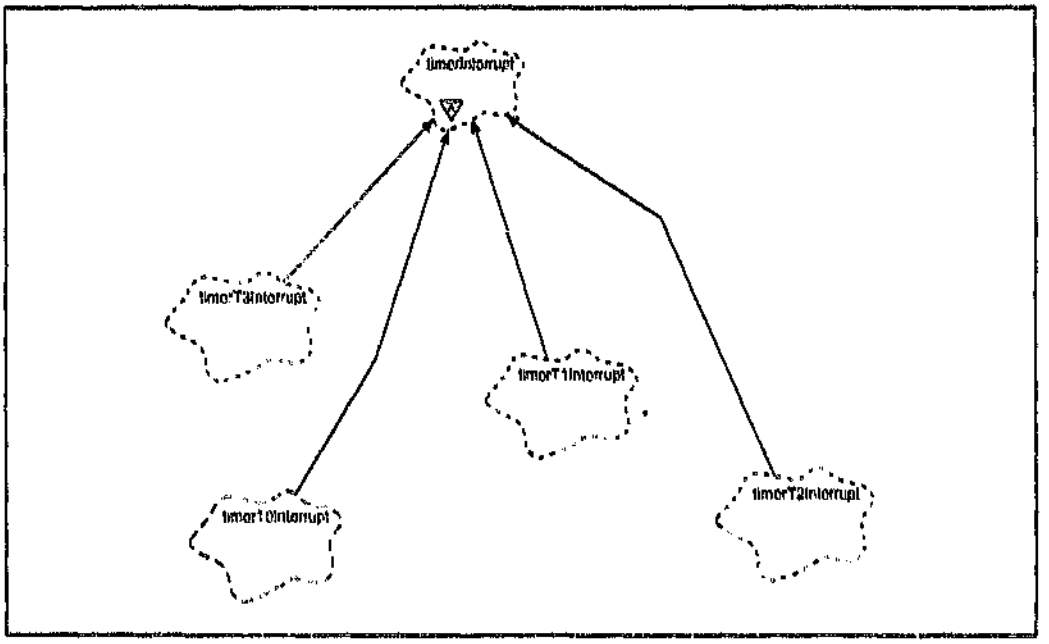


Figure 5: Class Diagram - The *Interrupt cycle*

2.2.5 The Main Cycle Class Decomposition

The *main cycle* contains more abstractions than the other categories. In order to simplify the representation of this category, we shall first proceed by subdividing its elements into loosely coupled and cohesive layers of abstractions or modules.

timerT1Interrupt

Responsibilities:

stops timer T1 and configures the output port for the next time interval.

Operations:

- timerT1InterruptRoutine ( ) [I]
- void setTimerAndPort ( ) [M]

Attributes:
None

Constructor:

```
timerT1Interrupt:: timerT1Interrupt ( ) : timerInterrupt ( )
{ };
```

Listing

```
void timerT1Interrupt:: setTimerAndPort ( )
{
ITU_GRA2= getTk ( );
P1DR = getUk ( );
ITU_TSTR = 4;
};

interrupt (ITU_IMIA1) void timerT0Interrupt::
timerT0InterruptRoutine ( )
{
stopTimer ( );
setTimerAndPort ( );
};
```

timerT2Interrupt

Responsibilities:

stops timer T2 and configures the output port for the next time interval.

Operations:

- timerT2InterruptRoutine ( ) [I]
- void setTimerAndPort ( ) [M]

Attributes:
None

Constructor:

```
timerT2Interrupt:: timerT2Interrupt ( ) : timerInterrupt ( )
{ };
```

Listing

```
void timerT2Interrupt:: setTimerAndPort ( )
{
ITU_GRA3= gett0Half ( );
P1DR = getUzero ( );
ITU_TSTR = 8;
};

interrupt (ITU_IMIA2) void timerT2Interrupt::
timerT2InterruptRoutine ( )
{
stopTimer ( );
setTimerAndPort ( );
};
```

timerT3Interrupt

Responsibilities:

stops timer T3 and configures the output port for the next time interval.

Operations:

- timerT3InterruptRoutine ( ) [I]
- void setTimerAndPort ( ) [M]

Attributes:
None

Constructor:

```
timerT3Interrupt:: timerT3Interrupt ( ) : timerInterrupt ( )
{ };
```

Listing

```
void timerT3Interrupt:: setTimerAndPort ( )
{
ITU_GRA0= getT0Half ( );
ITU_TSTR = 1;
};

interrupt [ITU_IMIA3] void timerT3Interrupt::
timerT3InterruptRoutine ( )
{
stopTimer ( );
setTimerAndPort ( );
};
```

timerT0Interrupt

Responsibilities:

stops timer T0 and configures the output port for the next time interval.

Operations:

- timerT0InterruptRoutine ( ) [I]
- void setTimerAndPort ( ) [M]

Attributes:
None

Constructor:

```
timerT0Interrupt:: timerT0Interrupt ( ) : timerInterrupt ( )
{ };
```

Listing

```
void timerT0Interrupt:: setTimerAndPort ( )
{
ITU_GRA1= getTj ( );
P1DR = getUj ( );
ITU_TSTR = 2;
};

interrupt [ITU_IMIA0] void timerT0Interrupt::
timerT0InterruptRoutine ( )
{
stopTimer ( );
setTimerAndPort ( );
};
```

2.2.4 The Interrupt Cycle Class Decomposition

The *interrupt cycle* as described in section 3 of part 1 of the Design Specifications, SDE 107, consists of four classes, each one controlling a different timer interrupt. From the analysis study it is manifest that the task of each timer interrupt routine can be grouped under two operations:

- stopTimer, responsible for stopping the timer. This function will be exactly the same for all the timer interrupt routines.

- setTimerAndPort, responsible for setting the timer and port for the next interval. Each subroutine will have a different setTimerAndPort operation, since each one will be setting a different timer and a different port, using different parameters.

The commonality among the classes suggests that they are unified under an abstract class, timerInterrupt. The classes of the *interrupt cycle* can now be described as follows:

timerInterrupt	Constructor:

Responsibilities:
 	
superclass to all "timer interrupt" classes.
 	
Operations:
 	
- void stopTimer ( ) {M}
- virtual void setTimerAndPort ( ) = 0; [M]
 	
Attributes:
 	
None
	timerInterrupt:: timerInterrupt ( ): dataRelay ( )
	{
	};
	
	Listing
	
	void timerInterrupt:: stopTimer ( )
	{
	ITU TSTR = 0;
	ITU TSRX &= ~1;
	};

In class phaseDetector, voltageDiff evaluates the difference between Reserve and Inverter output while direction compares the direction of the two. outOfPhase uses the results of those two operations to return the phasing status. Note the aggregation relationship between phaseDetector and the classes vSensor and fSensor. This relationship allows phaseDetector to check the voltage and frequency characteristics of Reserve and Inverter. Since the instances of vRes and vInvActual are declared in this class, those parameters - also needed by class vectorMagnitude - are made available outside the scope of phaseDetector via the selectors getvRes and getvInvActual. The structure of direction follows directly from the phase detection mechanism described in section 6.2 of part 1 of the Software Design Specifications, SDE 107.

reducePhaseDiff of class phaseDiffReducer merely sets finv such as to force a reduction in phase between Reserve and Inverter output.

reducePhaseDiff should only be called on an out of phase condition. This is the responsibility of the iterator phaseLockLoop of class pllSupervisor. pllSupervisor inherits from phaseDiffReducer. This grants it access to all the public operations of the two other classes, including the attribute finv declared as protected in phaseDiffReducer. fResOutOfRange is called upon an outOfPhaseCondition to set finv to fNominal and to verify the status of the inverter. Finally getfInv makes finv, the ultimate output of the phase-locked-loop mechanism, accessible outside the scope of its parent class. The relationship among those three classes is depicted in the figure 4 below.

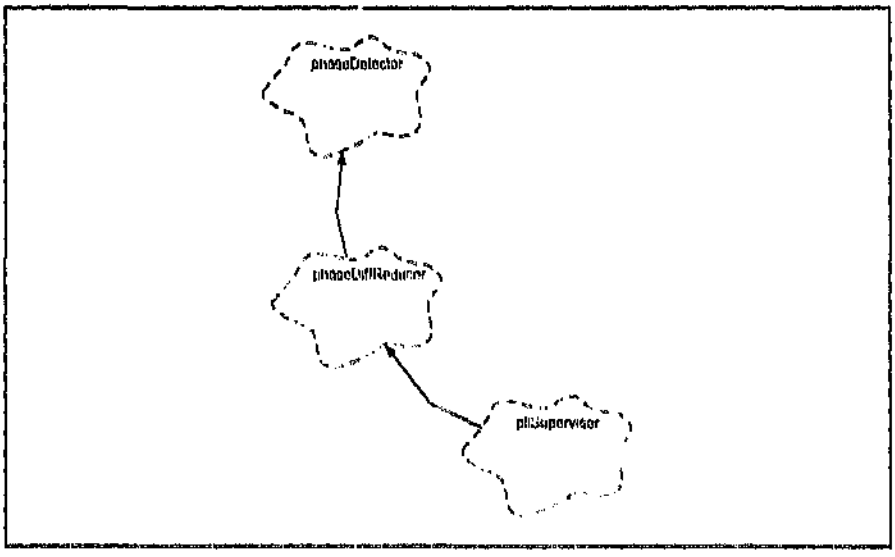


Figure 4: Class Diagram - Managing the Phase-locked-loop

phaseDiffReducer

Responsibilities:

reduces the phase difference between Reserve and Inverter output.

Operations:

- void reducePhaseDiff ( ) {M}

Attributes:

protected fInv

Constructor

```
phaseDiffReducer:: phaseDiffReducer (fSensor&,
vSensor&): phaseDetector (fSensor&, vSensor&)
{ };
```

Listing

```
void  phaseDiffReducer:: reducePhaseDiff ( )
{
sSwitchInhibit = true;
if (fRes .urrentVoltage ( ) <= fMin)
fInv = fMax;
else
fInv = fMin;
};
```

pllSupervisor

Responsibilities:

decides upon the course of action following feedback on the phases and frequencies of Reserve and Inverter output.

Operations:

- void fResOutOfRange ( ) {M}
- void phaseLockLoop ( ) {I}
- float getfInv ( ) {S}

Attributes:

None

Constructor:

```
pllSupervisor:: pllSupervisor (fSensor&, vSensor&):
phaseDiffReducer (fSensor&, vSensor&)
// pllSupervisor inherits from phaseDiffReducer
{
};
```

Listing

```
void  pllSupervisor:: fResOutOfRange ( )
{
sSwitchInhibit = true;
fInv = fNominal;
if (((fInv.currentFrequency ( ) < fMin ) | | (
fInv.currentFrequency ( ) > fMax)) && ((
vInv.currentVoltage ( ) >= vMax ))  // i.e if inv not OK
P1DR = 0;
while (vRes.currentVoltage ( ) > vResCross)
{ };
sSwitchInhibit = false;
};};

void pllSupervisor:: phaseLockLoop ( )
{
if (((fRes.currentFrequency ( ) < fMin) | |
(fRes.currentFrequency ( ) >= fMax))
fResOutOfRange;
else fInv = fRes;
if (outOfPhase ( ) == true)
reducePhaseDiff ( );
else sSwitchInhibit = false;
};

float pllSupervisor:: getfInv ( )
{ return fInv;
};
```

phaseDetector

Responsibilities:

keeps track of the phase difference between Reserve and Inverter output.

Operations:

- void voltageDiff ( ) [M]
- void direction ( ) [M]
- Boolean outOfPhase ( ) [M] & [S]

Attributes:

float vRes1, vRes2= vBig, vInv1, vInv2 = vSmall,
vDiff
int vResDirection, vInvDirection
Boolean sameDirection

In Header file:

```
class phaseDetector{
public:
.....

protected:
  fSensor& fRes (ADDRA), fInvActual (ADDRB);
  vSensor& vRes(ADDRC), vInvActual(ADDRD);
};
```

Listing

```
const int posDirect 1;
const int negDirect -1;

void  phaseDetector:: voltageDiff ( )
{
vDiff = vInvActual.currentVoltage ( ) -
vRes.CurrentVoltage ( );
}

void phaseDetector :: direction ( )
{
vRes1 = vRes2;
vInv1 = vInv2;
vRes2 = vRes.currentVoltage ( );
vInv2 = vInvActual.currentVoltage  ( );
if ((vRes - vRes1) > 0) vResDirection = posDirect;
else vRes.Direction = negDirect;
if ((vInv2  vInv1) >= 0) vInvDirection = posDirect;
else vInvDirection = negDirect;
if ( vResDirection = vInvDirection )
sameDirection = true;
else sameDirection = false;
};

 void phaseDetector:: outOfPhase ( )
{
voltageDiff ( );
direction ( );
if (( vDiff > vDiffMax) || (sameDirection = false))
return true;
else return false;
};
```

Operation setRegs initialises the microcontroller registers. startIntCycle initiates the *interrupt cycle*. By capturing overloadInterruptRoutine, the interrupt routine upon a 300% overload condition, as a Non-Maskable Interrupt (NMI), we ensure that this routine will have priority upon all other interrupts. This guarantees the prompt response specified in section 4.9 of the Functional Specifications SDE 106. genTable generates a look-up-table of switching times. Those values are accessible via getTa and getTb.

The class diagram for the class category initialisation is shown in figure 3, below.

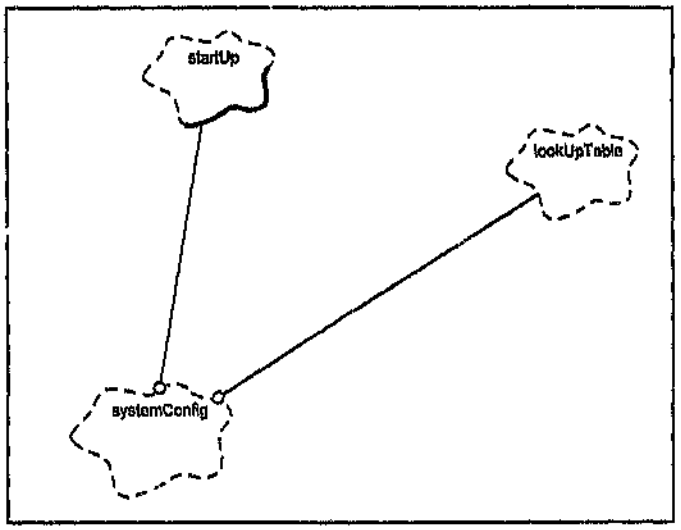


Figure 3: Class diagram - initialisation

2.2.3 Phase-locked-loop Class Decomposition

As described in section 6.4 of part 1 of the Software Design Specifications, SDE 107, two main abstractions emerge from the management of the phase-locked-loop:

- phaseDetector
- phaseDiffReducer

In order to co-ordinate the activities of those two classes a third class, pllSupervisor, will be introduced. The structure of the three classes are shown below.

lookUpTable

Responsibilities:

provides a look-up-table of switching times (at a usMag value of 1)

Operations:

- void genTable ( ) [M]
- void getTa ( ) [S]
- void getTb ( ) [S]

Attributes:

float phi, phiRad, ka1, kb1

int i, j

Listing

```
void lookUpTable:: genTable ( )
{
ka1 = ka * tSampling / tClock;
kb1 = kb * tSampling / tClock;
for ( j=1, phi=0; j <= (MaxSectorDiv - 1), phi<60;
j++, phi+= alphaMin)
{
phiRad = pi * phi / 180;
t[j][0] = floor ( ka1 * cos (phiRad) - 1/sqrt(3) * sin
(phiRad));
t[j][1] = floor ( kb1 * sin (phiRad));
}

void lookUpTable:: getTa (int j )
{
return t[j][0];
}

void lookUpTable:: getTb (int j )
{
return t [j][1];
}
```

systemConfig

Responsibilities:

configures the system, i.e the microcontroller and the software, appropriately.

Operations:

- void configSystem ( ) [I]

Attributes:
None

Listing

```
void systemConfig:: configSystem ( )
{
startUp repStartUp ( );
lookUpTable repLookUpTable ( );

repStartUp.setRegs ( );
repStartUp.computeKTerms( );
repLookUpTable.genTable ( );
repStartUp.startIntCycle ( );
}
```

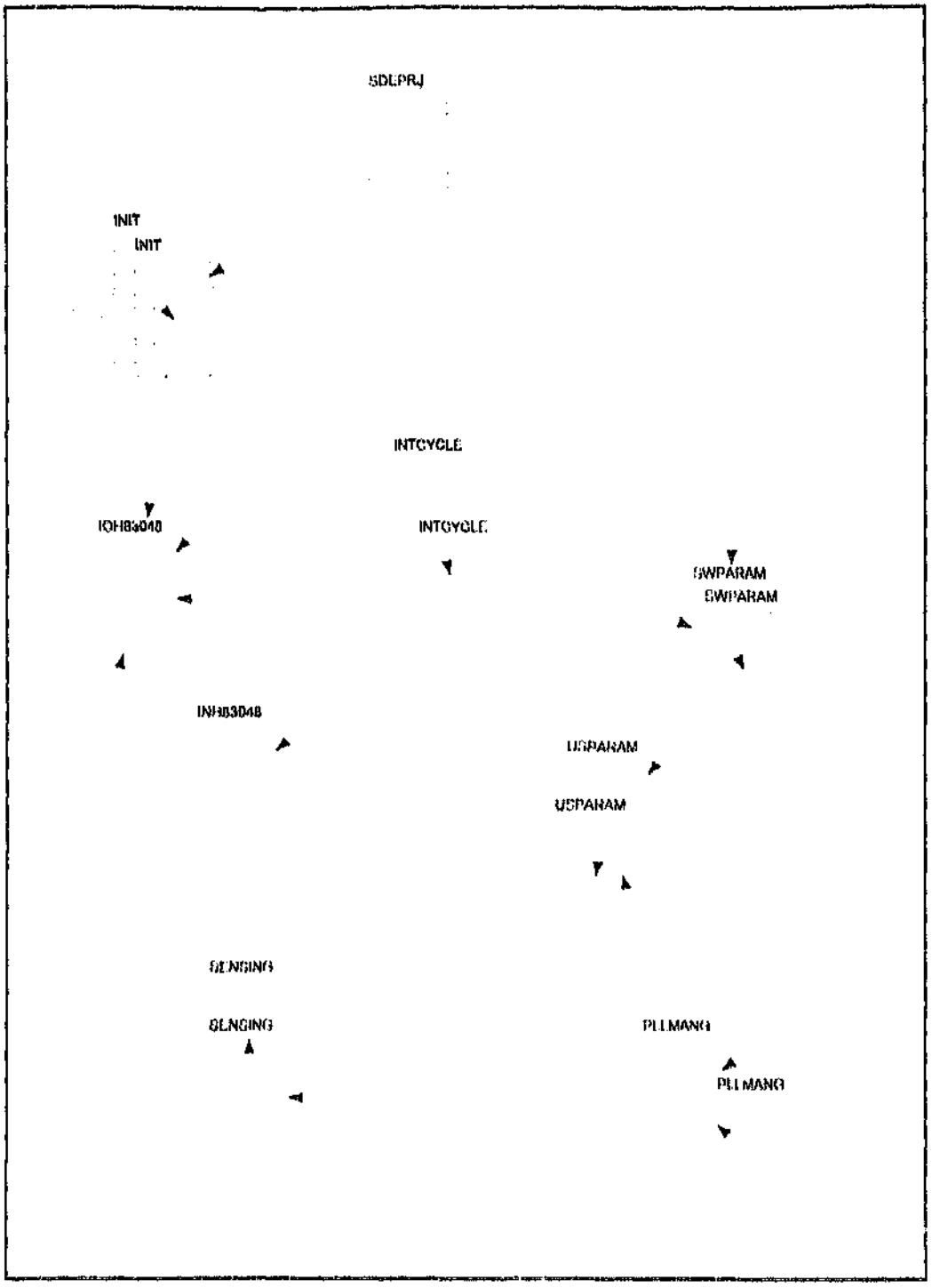


Figure 13: Module Architecture Of The Program

2.3 Module Architecture

Part of the architectural decisions must involve the physical packaging of the classes described in the previous section. In figure 13 the module architecture of these classes is illustrated. The particular partitioning chosen is again based upon loosely coupled and strong cohesion factors. A good module architecture is particularly useful when making changes, as it can significantly reduce the number of physical files upon which there will be intervention. The less files are modified, the less are the risks of making errors and the less time is wasted in compilation.

The current program is fairly small and as such physical partitioning follows much the same pattern as that captured in the class category division. Only the class category *main cycle* has been further divided into three layers:

- sensing - classes involved in sensing activities
- usparam - classes involved in determining the attributes of the desired vector Us*
- swparam - classes involved directly with the switching parameters, on the *main cycle* side

This partitioning was already discussed in section 2.2.5 above.

The grouping of classes sensor, vSensor and fSensor into a separate module points out another advantage of module partitioning. Grouped into a module those classes encourage re-use as they can easily be exported into other applications, such as the control of the rectifier.

Following C++ conventions, each module interface is placed in a separate header file, with a suffix *.h*, while the module implementations are carried out in *.cpp* files.

Notice the presence of two header files, ioh83048.h and inh83048.h. Those files act like interfaces between the microcontroller hardware and the software. They merely contain variable declarations and initialisations and have no implementation codes. The file ioh83048.h contains the definitions of the internal registers of the H8/3048 microcontroller. Variables of this file have already been encountered when dealing with operations that read or write values from or to the microcontroller registers; e.g. the use of the ITU TSTR register to stop the timers. The other header file, inh83048.h contains the vector addresses of all the interrupts supported by the Hitachi H8/3048 microcontroller.

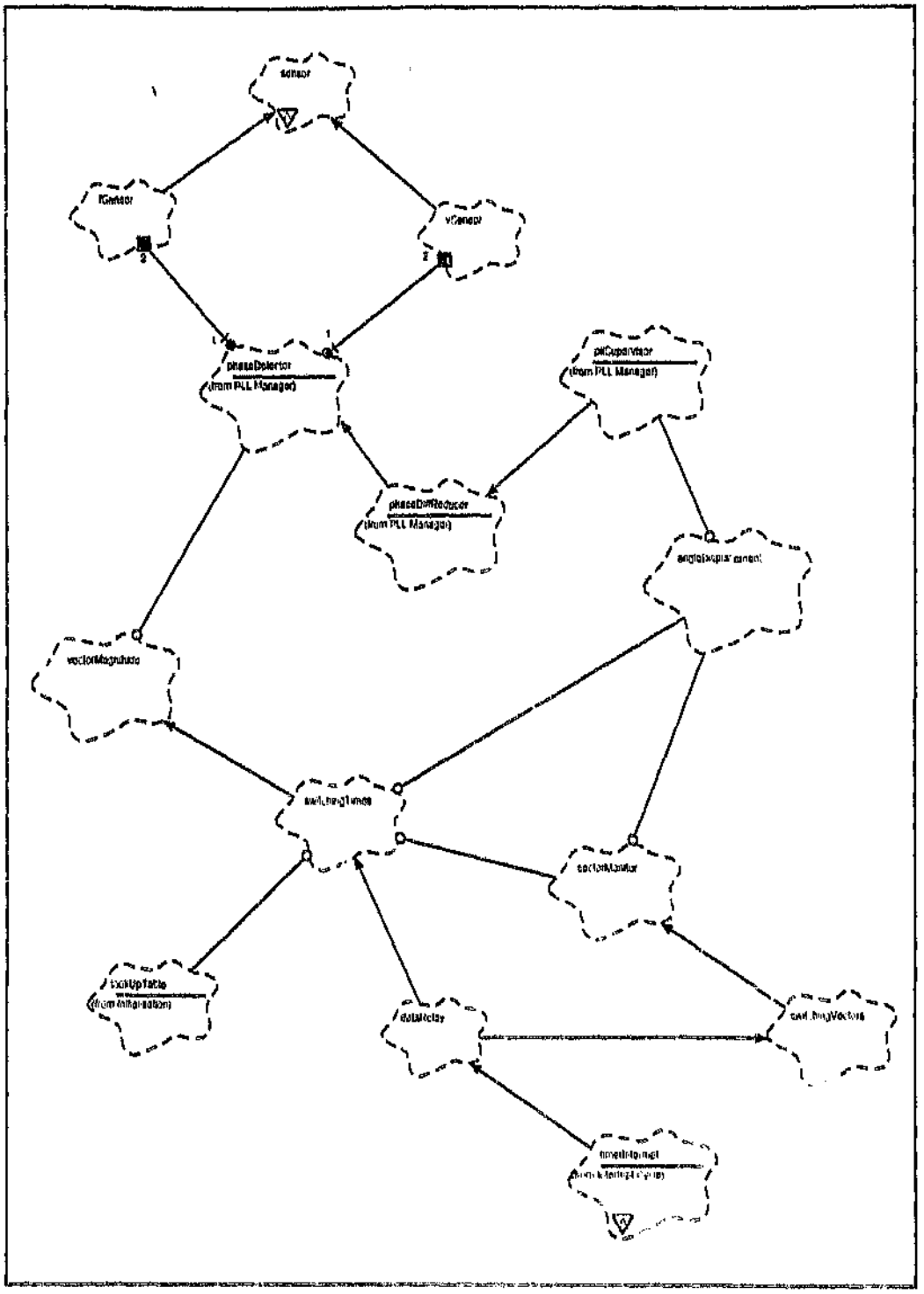


Figure 12: Class Diagram - Overall Architecture of the Class Category *main cycle*

2.2.5.4 Overall Class Architecture of the Main Cycle

The elements of the three modules developed above can be displayed in a single diagram as shown in figure 12. Notice that a few classes from other class categories have been included in this diagram in order to show the boundaries of the *main cycle*.

calcSwitchingTimes of class switchingTimes calculates the switching times. Let us consider the invariants associated with this operation: its preconditions include the assumption that the values of alpha, sequence and usMag have been updated. Its postconditions include the assumption that the values returned are in the correct order.

Similarly the preconditions of vectorUpdate of class switchingVectors include the assumption that the values of sequence, sector and sectorChange reflect the current status. Its postconditions include the assumption that the vectors returned matches the actual sector location and that they follow a Gray code order.

assignValues of class dataRelay merely assigns the temporary or dummy switching variables to their genius counterparts. The dummy variables are declared as protected attributes in their respective classes so that they can be accessed by dataRelay via the inheritance links. A further remark about dataRelay is that through its constructor, the switching variables can be initialised. This enables the *interrupt cycle* itself to be configured at the initialisation stage, when an instance of dataRelay is declared.

The rest of the member functions are quite straightforward and their characteristics have already been discussed in sections 5.6, 5.7 and 7.1 of part 1 of the Design Specifications, SDE 107. The corresponding class diagram for the three classes discussed in this section is shown below in figure 11.

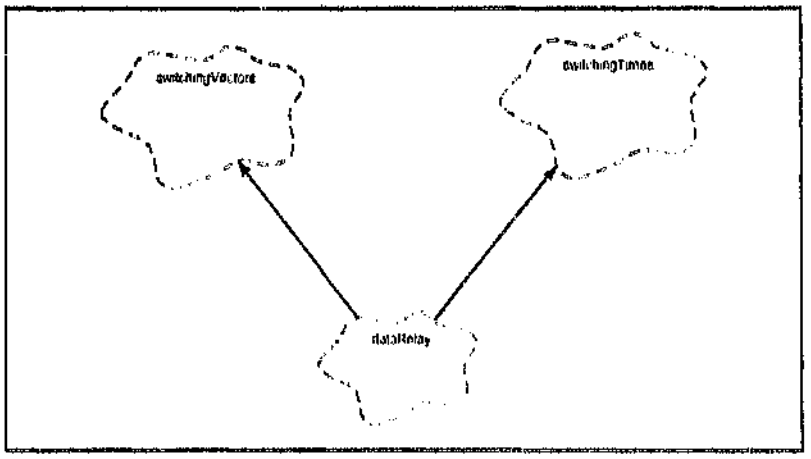


Figure 11: Class Diagram - Output Stage of the *main cycle*

dataRelay

Responsibilities:
assigns correct values to the switching parameters used by the *interrupt cycle*.

Operations:
- void assignValues (M)
- int getTj ( ) (S)
- int getTk ( ) (S)
- int getT0Half ( ) (S)
- int getUj ( ) (S)
- int getUk ( ) (S)
- int getUzero ( ) (S)

Attributes:

int tj, tk, t0Half, uj, uk, uZero

Constructor:

```
dataRelay:: dataRelay ( int tjInit, int tkInit, int t0HalfInit,
int ujInit, int ukI      uZeroInit ): switchingTimes ( ),
switchingVectors (
{
tj = tjInit;
tk = tkInit;
t0Half = t0HalfInit;
uj = ujInit;
uk = ukInit;
uZero = uZeroInit;
};
```

Listing:

```
void dataRelay:: assignValues ( )
{
uj = ujTemp;
tj = tjTemp;
uk = ukTemp;
tk = tkTemp;
uZero = uZeroTemp;
t0Half = t0HalfTemp;
};

int dataRelay:: getTj ( )
{
return tj;
};

int dataRelay:: getTk ( )
{
return tk;
};

int dataRelay:: getT0Half ( )
{
return t0Half;
};

int dataRelay:: getUj ( )
{
return uj;
};

int dataRelay:: getUk ( )
{
return uk;
};

int dataRelay:: getUzero ( )
{
return uZero;
};
```

switchingVectors

Responsibilities:

selects the correct switching sequence and their correct order of occurrence.

Operations:

- void vectorSwapping ( ) [M]
- void vectorSelect ( ) [M]
- void vectorUpdate ( ) [I]

Attributes:

```
private:
temp, actualSector, actualSeq;

protected:
unsigned int ujTemp, ukTemp, uZeroTemp;
```

Constructor:

```
switchingVectors:: switchingVectors ( ): sectorMonitor ( )
{ };
```

Listing:

```
void switchingVectors:: vectorSwapping ( )
{temp = ujTemp;
ujTemp = ukTemp;
ukTemp = temp;
};

void switchingVectors:: vectorSelect ( )
{
if ( actualSeq = 1)
{
/* select vectors ( ujTemp and ukTemp ) according to a
forward sequence */
.....
};
else
{
/* select vectors ( ujTemp and ukTemp ) according to a
reverse sequence */
.....
};

void switchingVectors:: vectorUpdate ( )
{
actualSector = getSector ( );
actualSeq = getSequence ( );
if ( getSector Change ( ) = true )
vectorSelect ( );
else
vectorSwapping ( );
};
```

2.2.5.3 Manipulation of the Switching Parameters

Manipulation of the switching parameters involves three distinct processes:

- calculation of switching times
- determination of switching vectors
- assigning the dummy switching variables to their genius counterparts, used in the *interrupt cycle*

Those processes are characterised by the following abstractions, in respective order:

- switchingTimes
- switchingVectors
- dataRelay

The structures of those classes are shown below:

switchingTimes

Responsibilities:

calculates the switching times and their order of occurrence.

Operations:

- void checkTable (lookUpTable &replookUpTable, angleDisplacement rop2AngleDisplacement) [M] & [S]

- void calcSwitchingTimes (sectorMonitor &repSectorMonitor)

Attributes:

private:
int i, ta, tb

protected:
int tjTemp, tkTemp, t0HalfTemp

Constructor:
switchingTimes:: switchingTimes ( ) : vectorMagnitude ( )
{ };

Listing:

```
void switchingTimes:: checkTable ( lookUpTable
&repLookUpTable, angleDisplacement
&rop2AngleDisplacement)
{
i = (rop2AngleDisplacement.currentAlpha ( ) / alphaMin );
ta = repLookUpTable.getTa (i);
tb = repLookUpTable.getTb (i);
};

void switchingTimes:: calcSwitchingTimes ( )
{
sectorMonitor repSectorMonitor;
currentusMag ( );
if ( repsectorMonitor.getSequence ( ) = 1 )
{
tjTemp = ta * usMag;
tkTemp = tb * usMag;
};
else
{
tjTemp = tb * usMag;
tkTemp = ta * usMag;
};
t0HalfTemp = ( tSampling - tjTemp  tkTemp) / 2;
};
```

sectorMonitor

Responsibilities:

keeps track of the sector location of the desired vector Us*.

Operations:

- Int getSector ( ) [M] & [S]
- Boolean getsectorChange ( ) [S]
- int getSequence ( ) [S]

Attributes:

unsigned int sector;
Boolean sectorChange;
int sequence;

Listing:

```
int sectorMonitor:: getSector ( )
{
angleDisplacement rep1AngleDisplacement ;
sectorUpdate = fmod (
rep1AngleDisplacement.currentPhata ( ), 60 );
if ( sectorUpdate > sector )
sectorChange = true;
else
{
sectorChange = false;
sequence = -sequence;
};
if (sectorUpdate > 6) sector =1;
else sector = sectorUpdate;
return sector;
};

Boolean sectorMonitor:: getSectorChange ( )
{ return sectorChange;
};

Int sectorMonitor:: getSequence ( )
{return sequence;
};
```

The two operations of vectorMagnitude are straightforward, one calibrates the operation while the other computes usMag. Note that usMag is declared as protected. This is because vectorMagnitude is only used by class switchingTimes and hence it is more effective to make usMag directly accessible to switchingTimes via the inheritance channel.

The operations currentAlpha and currentPhata follow from the discussion in section 5.4 of part 1 of the Design Specifications, SDE 107. In addition, following the discussion earlier in this section, currentAlpha now calls resetCounter upon a change of revolution. Operation resetCounter, which is now simply another member function angleDisplacement, in turn checks for the value of alpha first.

Operation getSector is the only modifier of the class sectorMonitor. It updates both sector and sequence. It further determines a change of sector condition. getSectorChange and getSequence are two selectors returning the values of sectorChange and sequence respectively.

- angleDisplacement
- vectorMagnitude
- sectorMonitor

The structure of those three classes are developed below:

angleDisplacement

Responsibilities:

keeps track of the angular displacement of the desired vector Us*.

Operations:

- float currentPhata ( ) [M] & [S]
- float currentAlpha ( ) [M] & [S]
- void resetCounter ( ) [M]

Attributes:

float tActual, phata, alpha;
unsigned int counter;

Listing:

```
float angleDisplacement:: currentPhata ( )
{
pllSupervisor repPllSupervisor ;
tActual = + + counter * tSampling;
phata=( 2*pi*tActual*repPllSupervisor.getFinvl ( ) );
if ( phata > = 360)
resetCounter;
return phata;
 };

void  angleDisplacement:: curron Alpha ( )
{
alpha = fmod (ph  ia, PI);
return alpha;
};

void angleDisplacement:: resetCounter ( )
{
if (alpha < gamma )
counter = 0;
else counter = 1;
};
```

vectorMagnitude

Responsibilities.
determines usMag

Operations:
- void currentusMag ( ) [M]
- void calibrateusMag ( ) [M]

Attributes:
private: float m, c;
protected: float usMag;

Listing:

```
void vectorMagnitude:: calibrateusMag ( )
{
m = ( vInvMax - vInvMin) / (usMagMin - usMagMax);
c = vInvMax - m * usMagMin;
 };

void  vectorMagnitude:: currentusMag ( )
{
phaseDetector repPhaseDetector;
usMag = (repPhaseDetector.getvInvActual ( ) - c) / m;
};
```

switchingTimes and switchingVectors.

Figure 9 shows that the class switchingVectors collaborates with both sectorMonitor and sequenceMonitor while the latter itself needs sectorMonitor to determine whether there has been a sector change. This diagram is too strongly coupled. A better separation of concern can be reached by integrating sequenceMonitor into sectorMonitor. This will lead to the structure shown in figure 10.

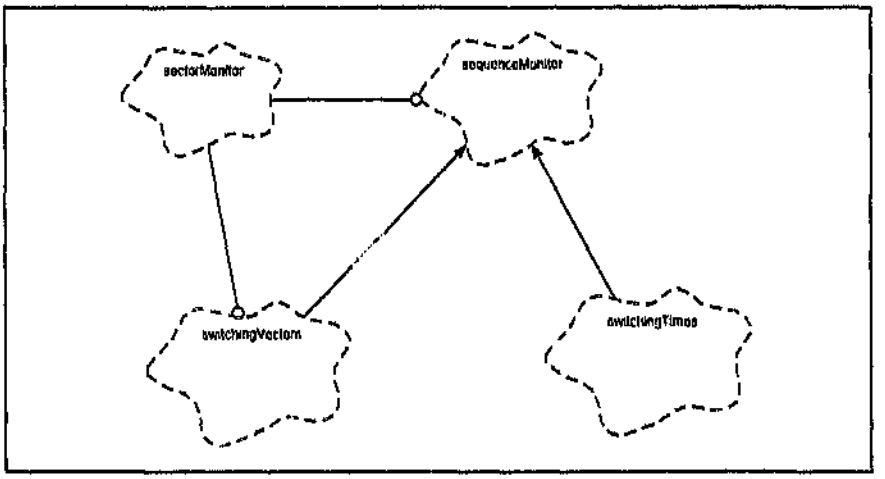


Figure 9: Class Diagram - Derived parameters (preliminary configuration)

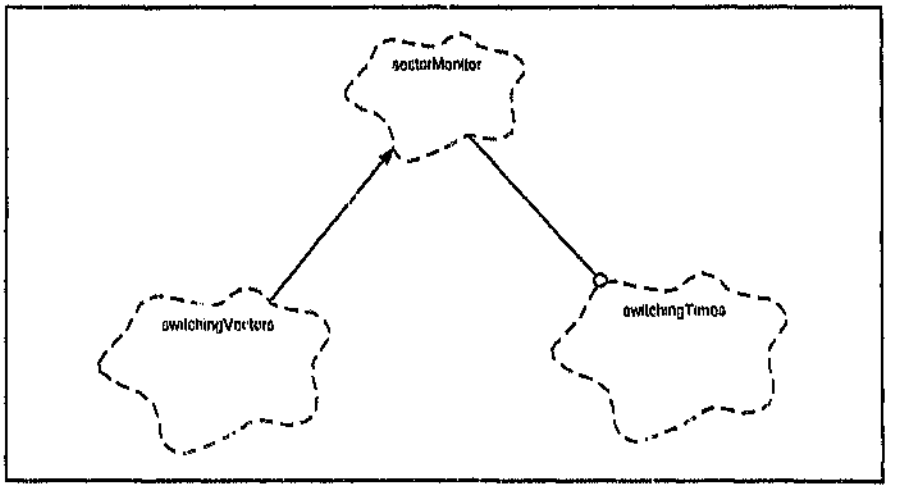


Figure 10: Class Diagram - Derived Parameters (final configuration)

We are now left with only three classes for this module:

This exercise reveals two factors:

a. class vectorMagnitude is not directly related to the other classes. This would tend to cast it out of this module. However, due to the commonality of the nature of its function with the rest of the members, it shall be left in this module.

b. The relationship among the other classes seems clumsy. First we note a two-way association relationship between angleDisplacement and counterMonitor. This is because angleDisplacement needs counter to determine pheta, while counterMonitor needs angleDisplacement to determine counter on a change of revolution. Secondly counterMonitor uses sectorMonitor to determine when a change of revolution has occurred. However this function could have easily been carried out by angleDisplacement itself from a manipulation of pheta which is its own attribute. This study suggests that a better simplification would result if counterMonitor is in fact integrated in angleDisplacement. This is depicted in figure 8.

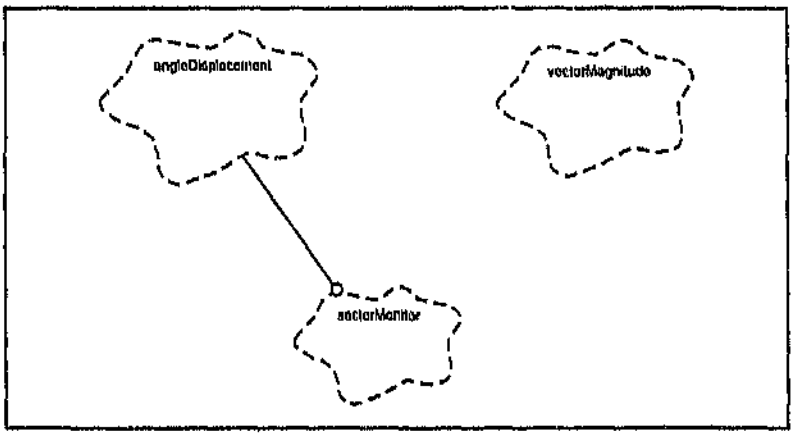


Figure 8: Class Diagram - Derived Attributes (final configuration)

This exercise further demonstrates the power of OOD and in particular that of the Booch methodology in revealing redundancy and awkwardness in the design structure. Even if one is not familiar with the particular problem domain, it is fairly easy to spot those inconsistencies and to take corrective actions early in the design process, simply by analysing the distribution of classes.

Moving a little forward in the design we can find yet another possibility for similar simplifications; this time between the classes sectorMonitor, sequenceMonitor,

determination of the attributes of Us* and the calculation of the switching parameters are shown in two separate diagrams, figure 19 and figure 20.

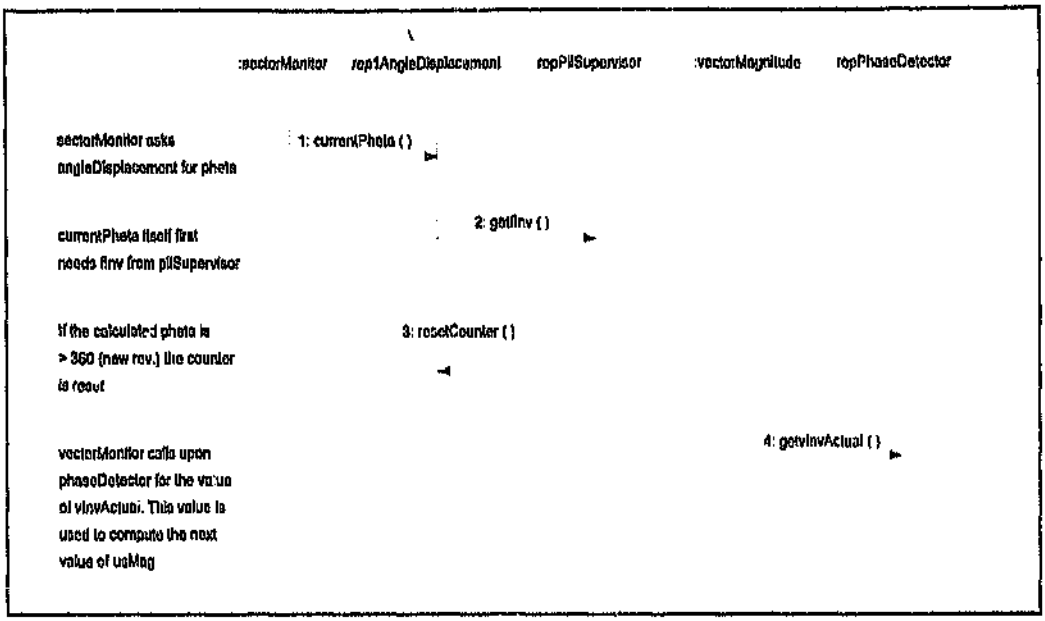


Figure 19 (a): Interaction Diagram - Updating The Attributes Of Us*

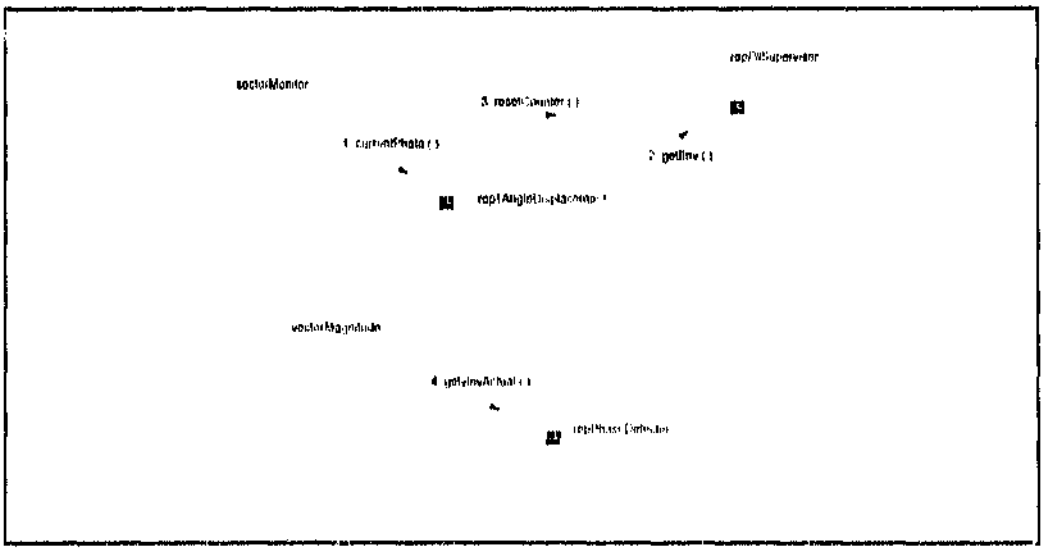


Figure 19 (b): Object Diagram - Updating The Attributes Of Us*

repSwitchingVectors.vectorUpdate ( ):

This operation, like the next one determines the values of the switching parameters. They both trigger a whole set of activities involved in first determining the values of the attributes of the desired vector Us*. Because of their commonality their interaction and object diagrams will be developed together.

vectorUpdate provides a new set of switching vectors. It calls getSector from class sectorMonitor. getSector in turns calls currentPhata from class angleDisplacement. currentPhata first increments counter and then calculates and returns the new value of phata, based on the incremented counter.

After computing the new value of phata, currentPhata checks for a change of revolution condition. If that is the case, counter is reset. But the reset value of phata itself is only effective on the next sampling interval. Also note that getSector updates the switching sequence as well.

repSwitchingTimes.calcSwitchingTimes ( ):

calcSwitchingTimes computes the switching times. It calls currentAlpha from angleDisplacement and compares it with alphaMin to obtain the value of the running counter i which allows it to retrieve the correct switching times, using getTa and getTb, both from class lookUpTable.

The previous operation of main, vectorUpdate( ), updated phata and by the same instance enabled alpha to be updated when currentAlpha is called. This is an important consideration and it explains why vectorUpdate is invoked before calcSwitchingTimes.

calcSwitchingTimes also calls currentusMag from class vectorMagnitude. The latter calls getInvActual from class phaseDetector, which provides the up-to-date value of vInvActual previously calculated on the call to vInvActual.currentVoltage - see repPllSupervisor.phaseLockLoop( ) above.

This value of usMag (from currentusMag) is multiplied by the switching times obtained from the look-up-table to derive the actual switching times values.

The diagram for the last two operations can now be drawn. As mentioned earlier, both vectorUpdate and calcSwitchingTimes first call upon other operations to determine the attributes of the desired vector Us*. For a simpler representation the

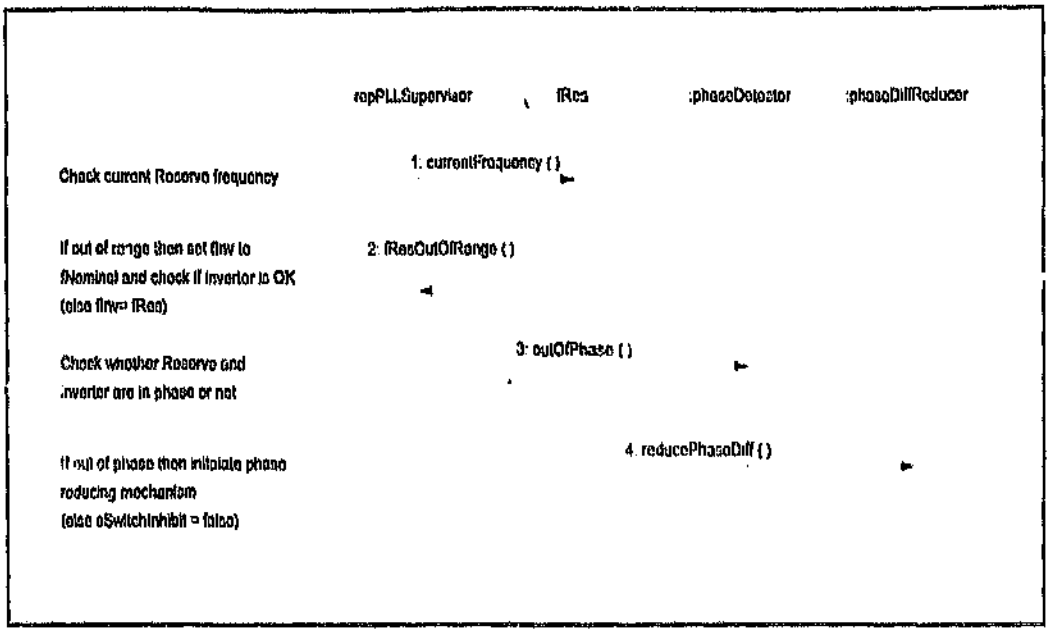


Figure 18 (a): Interaction Diagram - Phase-locked-loop Process

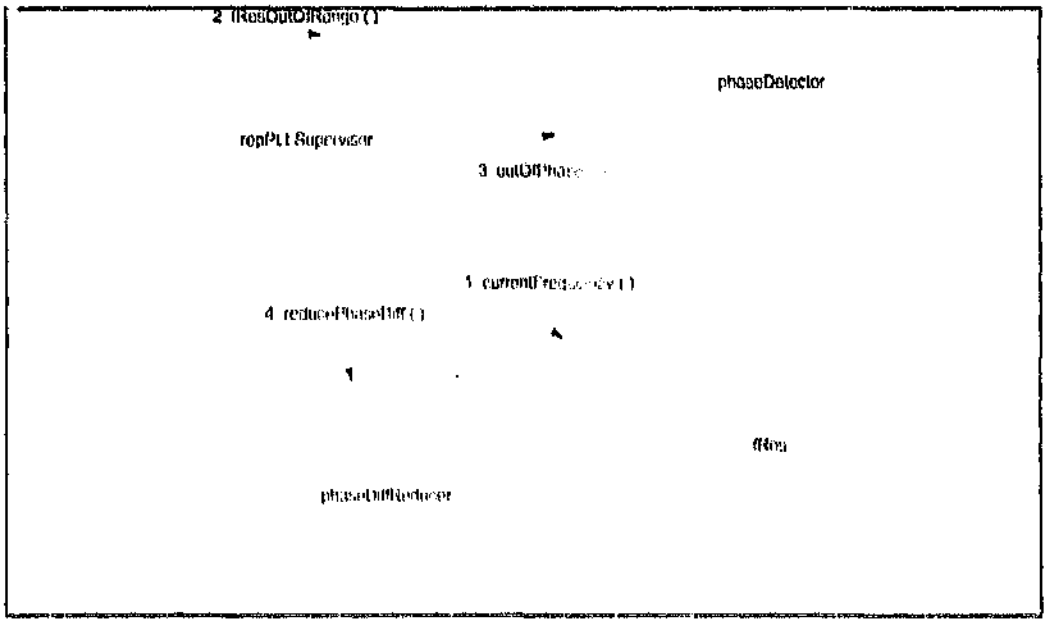


Figure 18 (b): Object Diagram - Phase-locked-loop Process

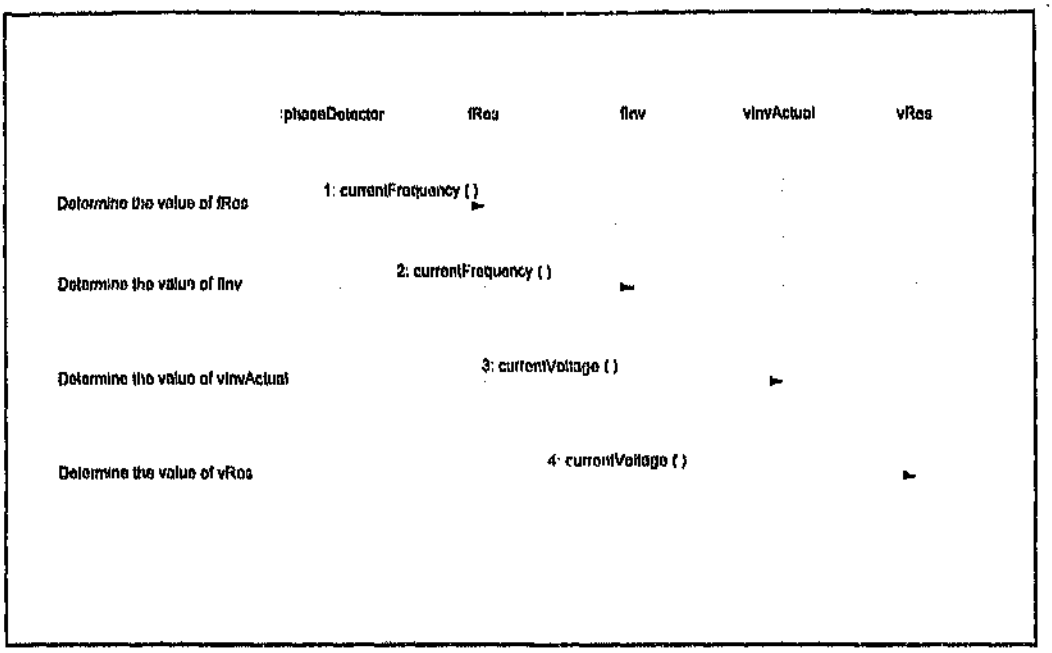


Figure 17 (a): Interaction Diagram - Sensing Activity

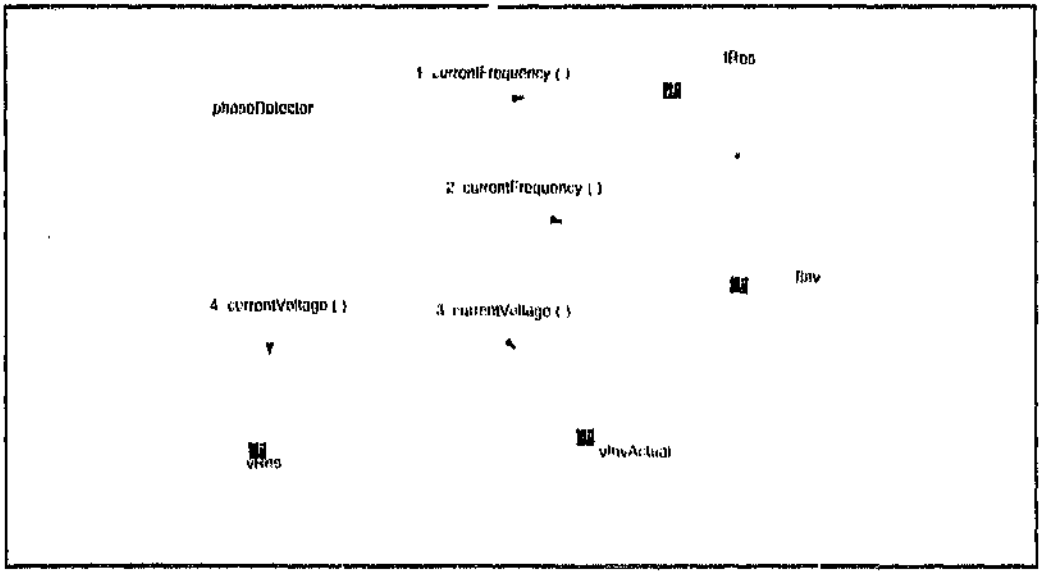


Figure 17 (b): Object Diagram - Sensing Activity

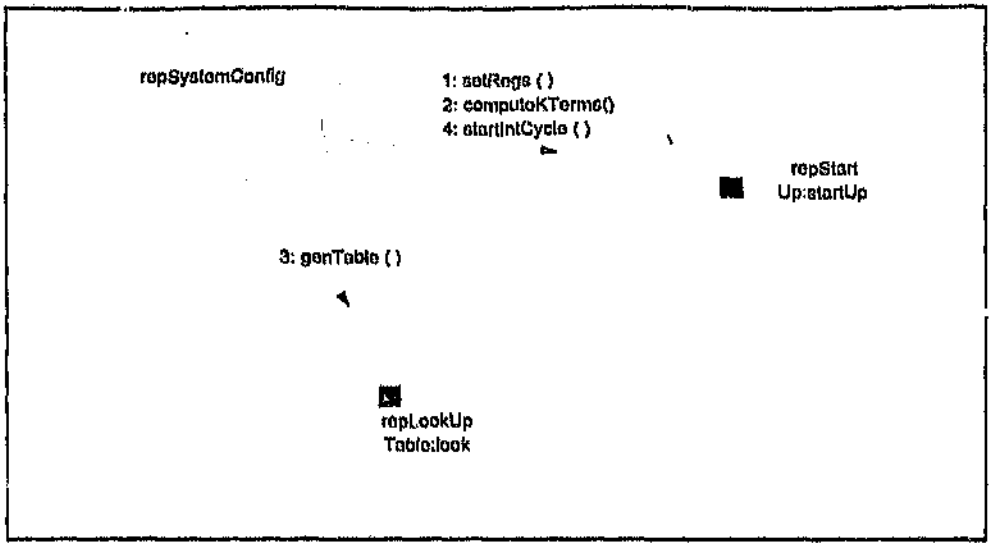


Figure 16 (b): Object Diagram - Initialisation Process

repPllSupervisor.phaseLockLoop ( ):

phaseLockLoop from class pllSupervisor performs the phase-locked-loop and generates a value of finv. It calls fRes.currentFrequency from class fSensor which provides the actual Reserve frequency by calling currentValue from class sensor. If at this stage the frequency is out of limits, the necessary steps are taken by fResOutOfRange of class pllSupervisor itself. phaseLockLoop also calls outOfPhase from class phaseDetector which calls voltageDiff and direction. voltageDiff calls currentVoltage of both the instances of class vSensor, i.e. vRes and vInvActual. Like fSensor, vSensor calls currentValue from class sensor to obtain the sensed values. voltageDiff uses those two values to determine the voltage difference between Reserve and Inverter output. direction compares the direction of Reserve and Inverter. direction too, invokes currentVoltage.

The dynamic semantics here are best captured using two sets of diagram. The first set in figure 17 (a) and (b) describes the sensing activity while the second set in figure 18 (a) and (b) depicts the phase-locked-loop process.

3.2 Dynamic Semantics of the Individual Elements

In this section we go down into the ramifications of the objects associated with each of the process illustrated in the interaction diagram in figure 14 (a), in order to capture the dynamic behaviour of each of the objects. The same logical order of operations, as presented in the function main will be followed.

repSystemConfig.configSystem ( ):

configSystem from class systemConfig sets up the microcontroller registers, then generates the look-up-table and finally initiates the *interrupt cycle*. It calls:

- setRegs from class startUp
- genTable from class lookUpTable
- and startIntCycle from class startUp

The corresponding interaction and object diagram for this set of activities are shown in figure 16 (a) and (b).

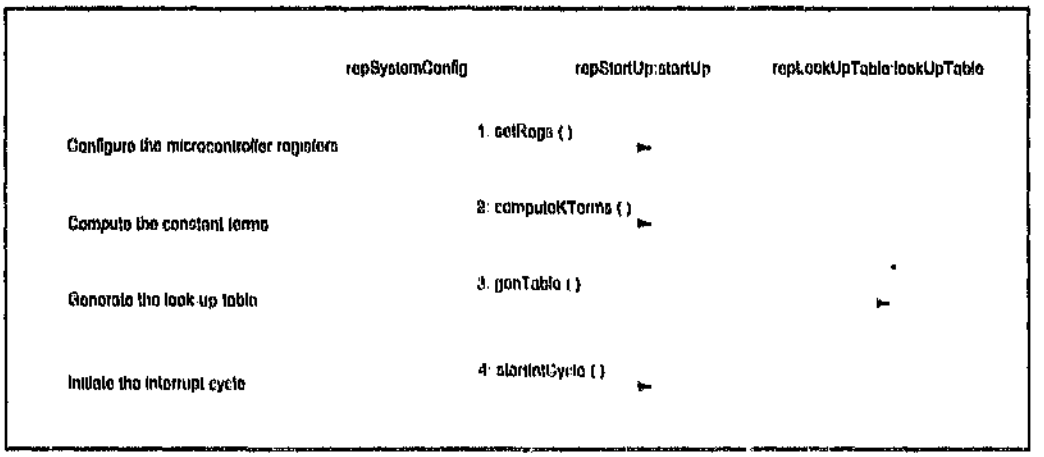


Figure 16 (a): Interaction Diagram - Initialisation Process

All the sensing, checking and computation activities have one thing in common in the sense that they are all processing data. Not surprisingly they constitute the core activities of the application. We shall name the state when the microcontroller is involved into one of the processing activities the *processing state*. Similarly every time values are being assigned to the *interrupt cycle*, the microcontroller will be in an *assigning state*. This state differs from the *processing state* in that here data is merely being passed along, without being manipulated. A third state will be required to designate the status of the *main cycle* at the check points, where its activities are temporarily halted. This last condition will be described as a *waiting state*. This state is only relative to the *main cycle* activities and should not be confused with an idle state of the microcontroller. In this application, due to the continuous activities of the *interrupt cycle*, the microcontroller, under normal conditions, is never idle. Figure 15 captures the state transitions in the *main cycle*. Notice that the *main cycle* goes into a *waiting state* both before and at the end of a *processing state*.

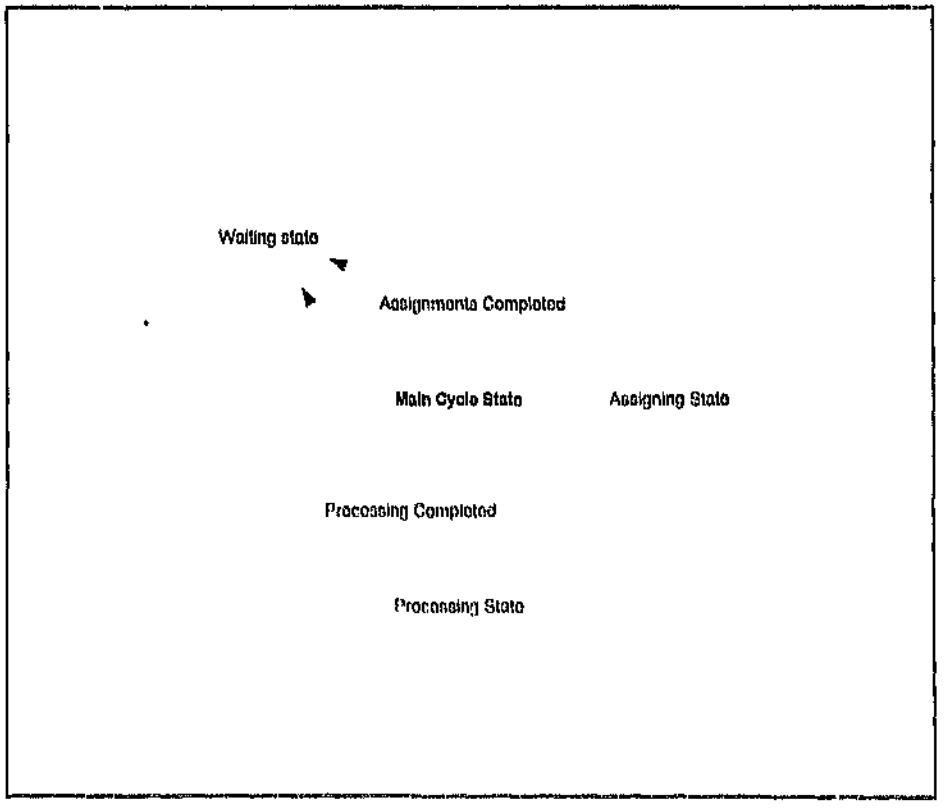


Figure 15: State Diagram - *main cycle* process

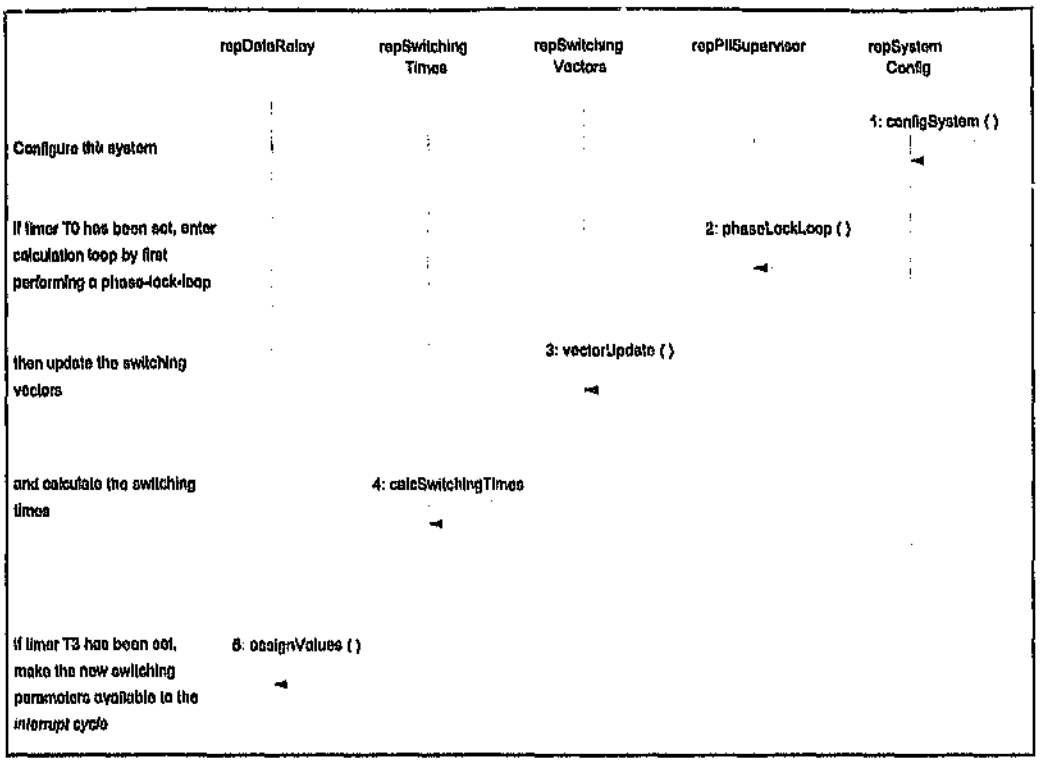


Figure 14(a): Interaction Diagram - The *main cycle* as run in the function main

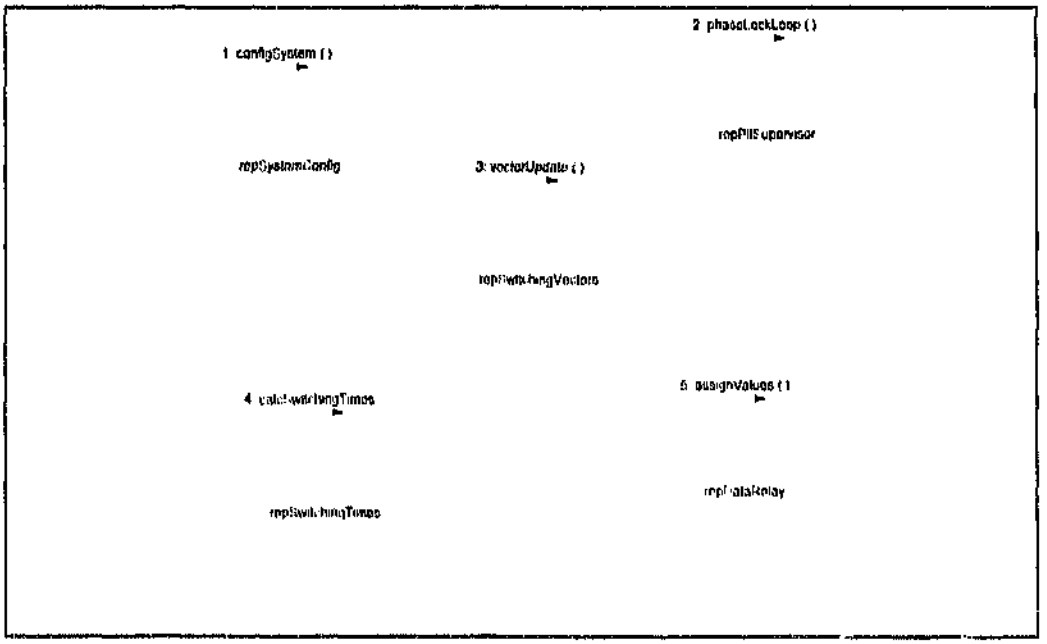


Figure 14(b): Object Diagram - The *main cycle*, as run in the function main

```
// preprocessor directives
# define pi 3.14159
# define MaxSectorDiv 240

# define tClock  62.5E-9
# define tSampling 1.6E-4
# define tInitial  tSampling / 4
#define vBig 400
# define vSmall 0


// include files
#include <inh83048.h>
#include <inh83048.h>
.....
```

```
int  main ( )
{

// creating objects
systemConfig repSystemConfig;
dataRelay repDataRelay ( tInitial,  tInitial, tInitial, 48,35,
7);
pllSupervisor repPllSupervisor;
switchingVectors repSwitchingVectors;
switchingTimes repSwitchingTimes;

// initialisation
repSystemConfig.configSystem ( );

// main cycle
for ; ;
{
while ( ITU_TSTR  != 225 )  // first check point
{ };
rep PllSupervisor.phaseLockLoop ( );
repswitchingVectors.vectorUpdate ( );
repSwitchingTimes.calSwitchingTimes ( );
while ( ITU_TSTR != 232 )   // second check point
{ };
repDataRelay.assignValues ( );
};

return 0;
}
```

3.1.5 Schematic Representation

The corresponding interaction diagram and object diagram for the *main cycle* process are shown in figure 14 (a) and (b) respectively. Notice the simplicity that OOD generates at the main function level. Although each of the operations shown triggers a series of complex activities, it is possible, by viewing the system at a high level of abstraction, to ignore those complexities while still maintaining insight into the process.

- performing the phase-locked-loop
- calculating the values of the attributes of the desired vector Us*
- updating the sequence polarity
- updating the switching vectors and their order of occurrence
- calculating the switching times and their order of occurrence
- passing the calculated values to the *interrupt cycle*

This set of activities will have to be repeated at every sampling interval. In addition the software will also have to handle out of limits conditions like out of frequency range and overload.

3.1.3 Synchronising the Main Cycle with the Interrupt Cycle

In our last treatment of the synchronisation problem, in section 7 of part 1 of the Design Specifications, SDE 107, it was concluded that two check points will be necessary. The first one, located at the entry point of the *main cycle*, ensures that a new run through the *main cycle* and the *interrupt cycle* always coincide. This condition can be maintained by allowing a new run through the *main cycle* only when the first timer interrupt routine of a new sampling interval has been completed, i.e. when the timer T0 has been set (ITU_TSTR = 225).

The second check point maintains the integrity of the data used by the *interrupt cycle*. It is located at the end of the *main cycle* and it checks for the execution of the last timer interrupt routine of the sampling interval, i.e it checks if timer T3 has been set (ITU_TSTR = 232). After this last subroutine has been completed, the switching values for the next sampling interval can safely be passed to the *interrupt cycle* without the risk of mixing of data occurring.

3.1.4 The Function main

The ideas expressed above are summarised in the structure of the function main below. All the elements of the function have been treated separetely above. The initialisation value of repDataRelay follows from the discussion in section 5.3 of part 1 of the Design Specifications, SDE 107.

3 Dynamic Model

We now proceed to develop the dynamic semantics of the problem. Following Booch method, this task will be performed through two additional diagrams:

- state diagrams
- interaction diagrams

Due to the strong relationship between interaction diagrams and object diagrams, the latter will also be developed in the process, as the dynamic model is unfolded. It should be clear though, that object diagrams represent snapshots in time of transitory stream of events over a certain configuration of objects and as such are very much part of the static model and not of the dynamic model.

3.1 The Main Cycle: An Overview

At this stage the outline of the function main can be sketched. It will have the following important activities:

- running the initialisation procedure, including the initiation of the *interrupt cycle*
- co-ordinating the activities of the *main cycle*
- synchronising the *main cycle* with the *interrupt cycle*

3.1.1 Initialisation Procedure

The first task can be easily accomplished by running configSystem of systemConfig. We recall that configSystem is an iterator operation that manages the whole initialisation procedure.

3.1.2 Co-Ordinating the Activities of the Main Cycle

Co-ordinating the activities of the *main cycle* implies running the operations in such a way as to ensure that both the preconditions and the postconditions of all the objects are satisfied. This will become clearer as the model is developed, but already it is evident that the function main will have to prompt the following activities (other than the initialisation) in the following order:

- sensing the voltage and frequency parameters

microcontroller. In the present application it is used by the timer interrupts and by the NMI.

The module sdeprj.cpp contains the privileged non-member function main and is the root of the program. Description of the function main is postponed to section 3.1.4 where its role in co-ordinating and synchronising the operations is considered.

2.4 Object Architecture

An object diagram shows the interaction among the class instances. It is used to trace the execution of a scenario just like an interaction diagram. Object diagrams and interaction diagrams are alternate representations of a scenario. For the purpose of this case study, the object diagrams and the interaction diagrams will be developed simultaneously in the next section.

- Members of the SEAL Management Board
- Head of the Department, Electrical Engineering
- Individuals who will perform internal and external surveillance audits of the SEAL Quality Management System
- Engineering Manager, Meissner
- Product Developer
- Product Manager

1.5 Applicable Documents

1.5.1 Specifications
None

1.5.2 Standards
a. SEAL QMS Document Creation Template, QS 002, Revision 0.01, 5 April 1994
b. SEAL QMS Document Layout, Presentation and Typesetting Guide, QS 003, Revision 0.01, 1 April 1994

1.5.3 Procedures
None

1.5.4 Guidelines
None

1.5.5 Other Documents

a. SDE 108, Software Design Specifications - Part 2

1.6 Assumptions

It is assumed that the reader is familiar with inverter systems and the concept of Space Vector Modulation. In addition the reader should be acquainted with the design developed in the Software Design Specifications, document SDE 108.

1 Scope

1.1 Introduction

Coding follows directly from the design specifications. It builds on the structures and the executable releases derived in the design exercise to produce an executable version of the program. The task is especially concerned with the integration of the different modules into a working software version that will meet all the specifications developed so far.

1.2 Purpose

This document aims at providing a traceability between the design exercise and the source code. By capturing the major software versions developed so far, it provides more insight into the problem and explains why certain design decisions were made.

1.3 Definitions

For the purposes of this document, the definitions given in document SDE 108 apply, together with the following:

IGBT : Insulated Gate Bipolar Transistor

OOD : Object-Oriented Design

OOL : Object-Oriented Language

PWM : Pulse-Width Modulation

1.4 Audience

The audience for this document comprise the various stakeholders of the SEAL, including:

- Full-time staff members of SEAL
- All full-time and part-time post-graduate students associated with SEAL

Change History

Configuration Control

Project:	SDES
Title:	Software and Hardware Implementation Documentation
Doc. Reference:	SDE 110
Created by:	H. Bapoo
Creation Date:	24 July 1996

Document History

Version	Date	Status	Who	Saved as:
0.01	96/07/24	Draft	H.Bapoo	\1995-13\TP\SDE110WP.001
1.00	96/08/25	Approved	H.Bapoo	\1995-13\TP\SDE110WP.100

Revision History

Version	Date	Changes
0.01	96/07/24	New Document
1.00	96/08/25	Update to revision number and status following Board Approval

Management Authorisation

Version	Date	Status	Management Board Minute Reference
1.00	96/08/25	Approved	Section 2.3 of SDE 692

Change Forecast

This document will be changed every time there is a modification to the source code or to the hardware design.

Table of Contents

Appendices

SDES QMS

Software and Hardware Implementation Documentation

Technical Product

Version 1.00

Document Status: Approved

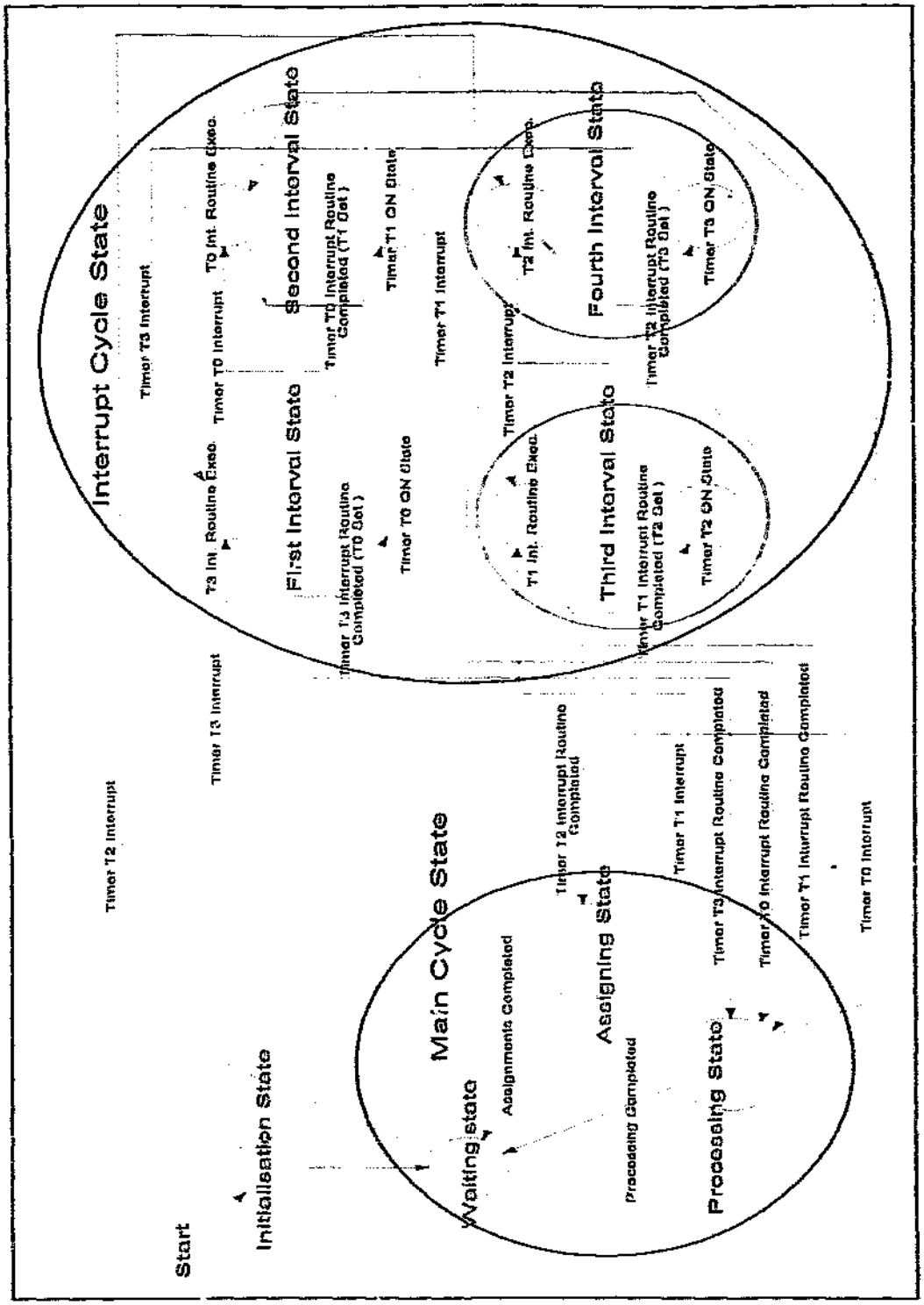


Figure 23: State Diagram - The Overall Process

3.3 The Overall Process

Finally, the initialisation process, the *main cycle* and the *interrupt cycle* can all be captured in a single state diagram as in figure 23. There, the event-ordered behaviour of the whole process can be followed.

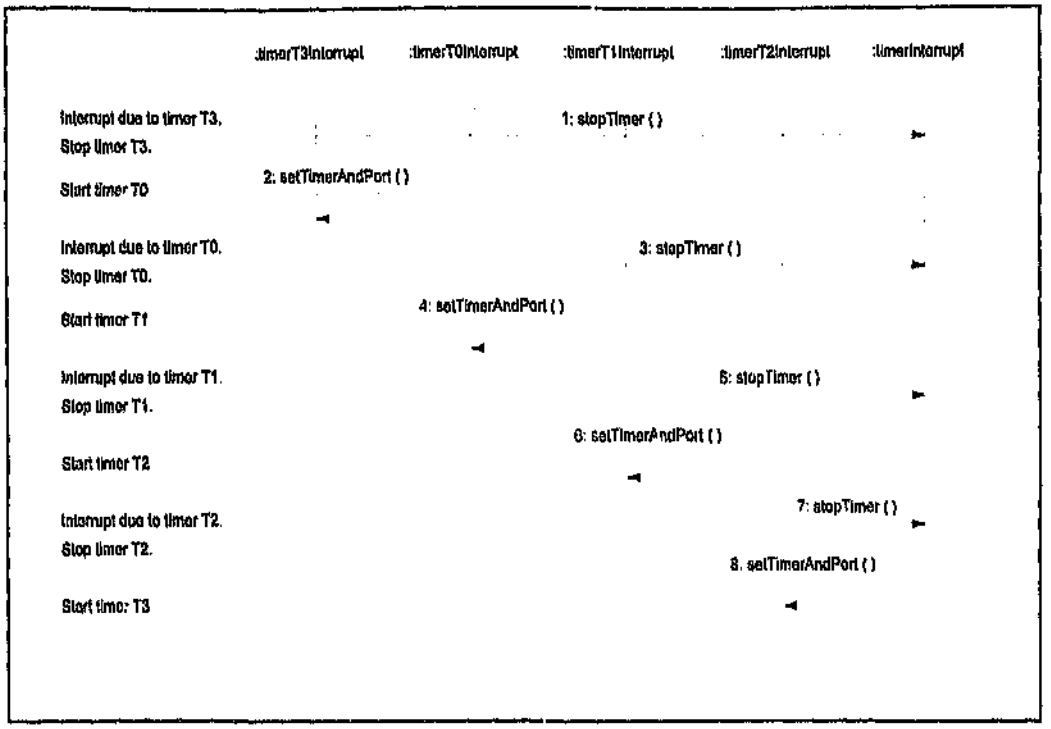


Figure 22 (a): Interaction Diagram - The *Interrupt cycle*

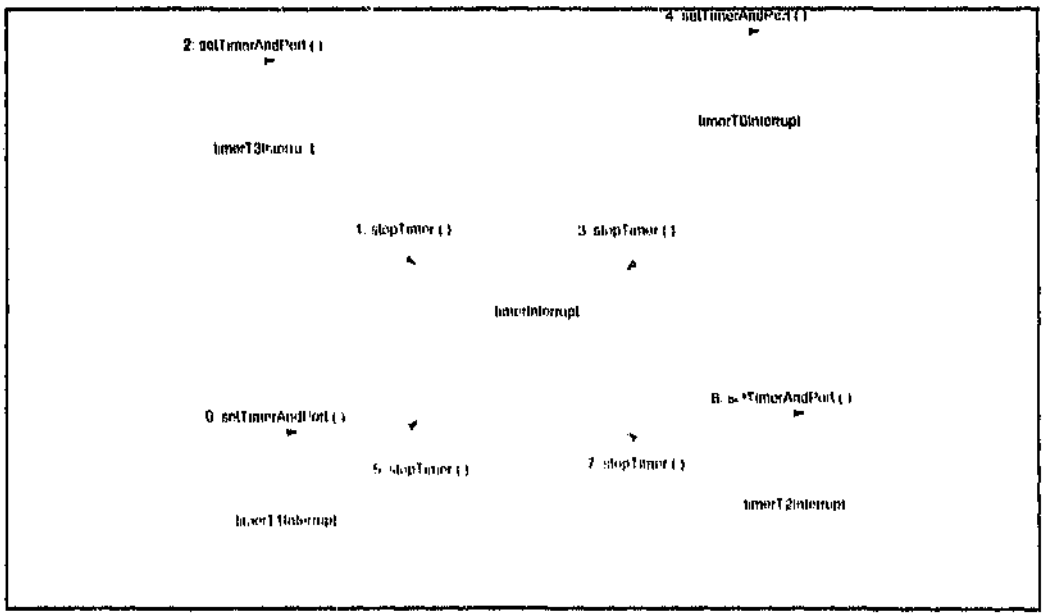


Figure 22 (b): Object Diagram - The *Interrupt cycle*

The dynamic representation developed in section 3.1 of part 1 of the Design Specifications, SDE 107, can now be refined and adapted into a state diagram as shown in figure 21. In addition, we can also draw the interaction and object diagrams as in figure 22.

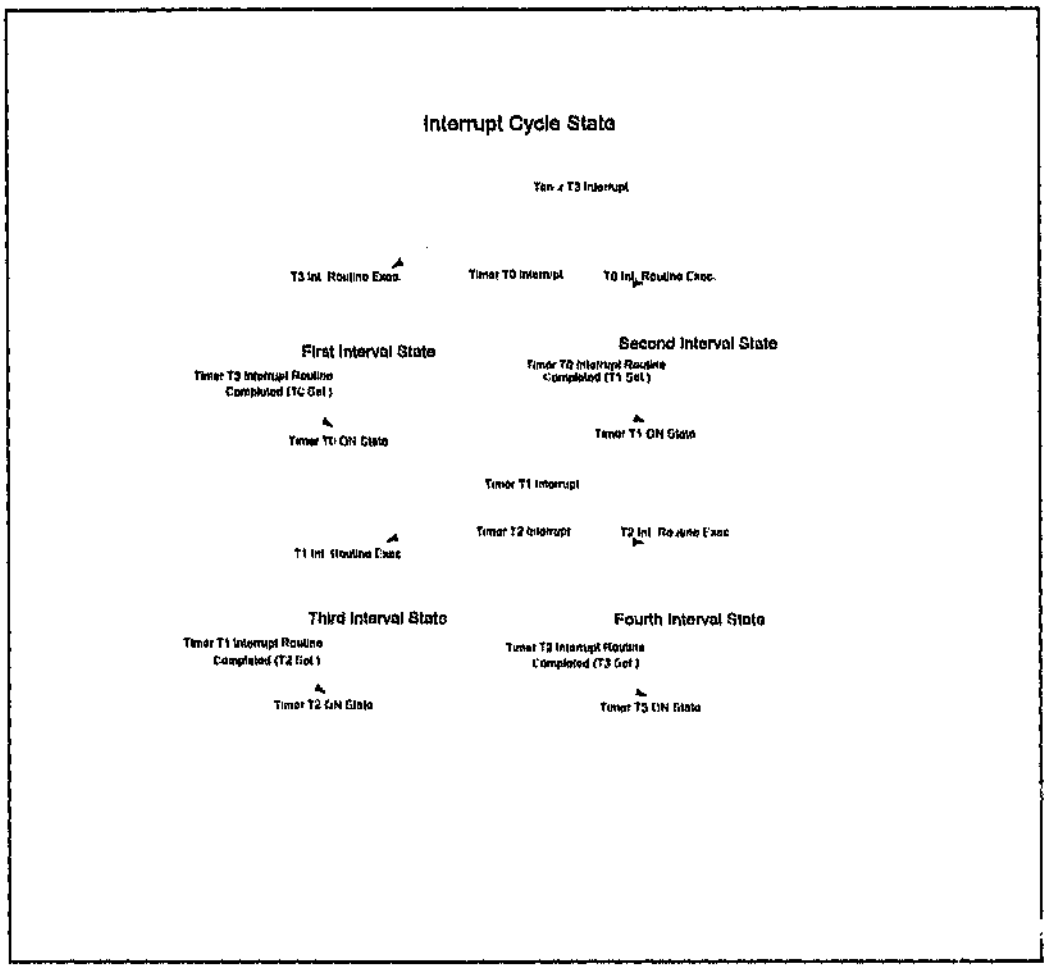


Figure 21: State Diagram - The *interrupt cycle*

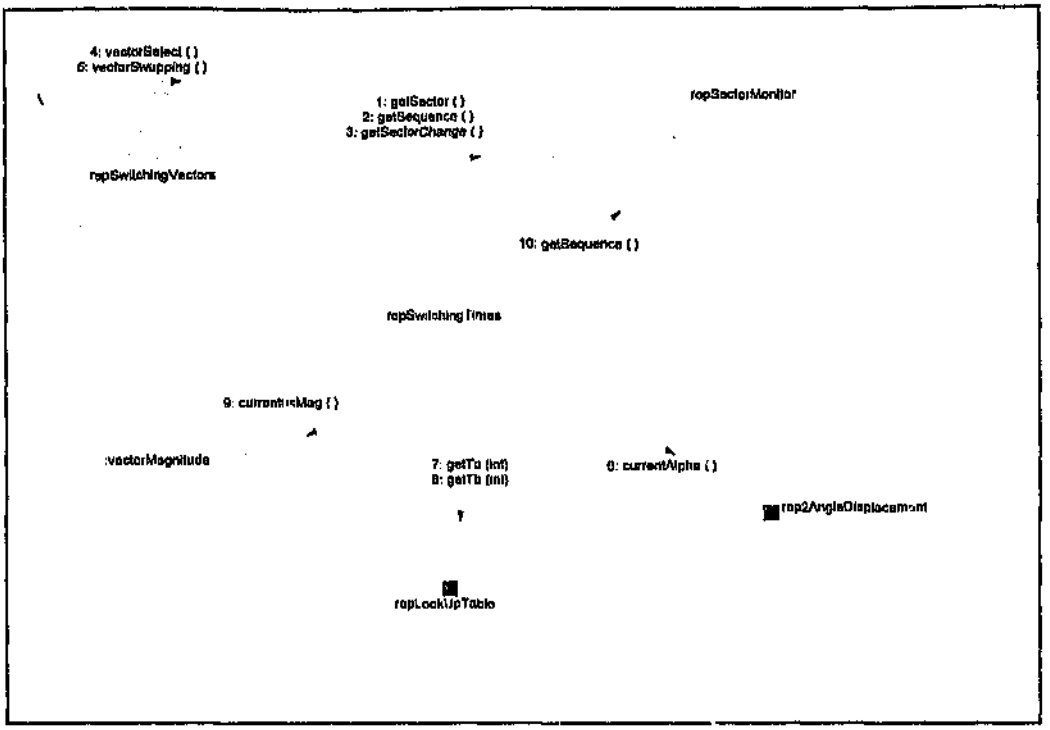


Figure 20 (b): Object Diagram - Calculating The Switching Parameters

repDataRelay.assignValues( );

assignValues assigns the temporary values (ujTemp, ukTemp, uZeroTemp, tjTemp, tkTemp and t0HalfTemp) calculated after operations vectorUpdate and calSwitchingTimes were called, to their genius counterparts (uj, uk, uZero, tj, tk and t0Half).

The *interrupt cycle*

On top of all the operations mentioned above, running in the background is the *interrupt cycle*. Its evolution is governed by timer interrupts that follow each other in a repetitive loop. This cycle was initiated in the initialisation procedure - see repSystemConfig.configSystem ( ).

Each of the four timer interrupt classes, timerT3Interrupt, timerT0Interrupt, timerT1Interrupt and timerT2Interrupt, calls the operation stopTimer of class timerInterrupt. They also obtain all the necessary parameter values (uj, tj, etc.) through their inheritance link with class timerInterrupt, itself inheriting from class dataRelay.

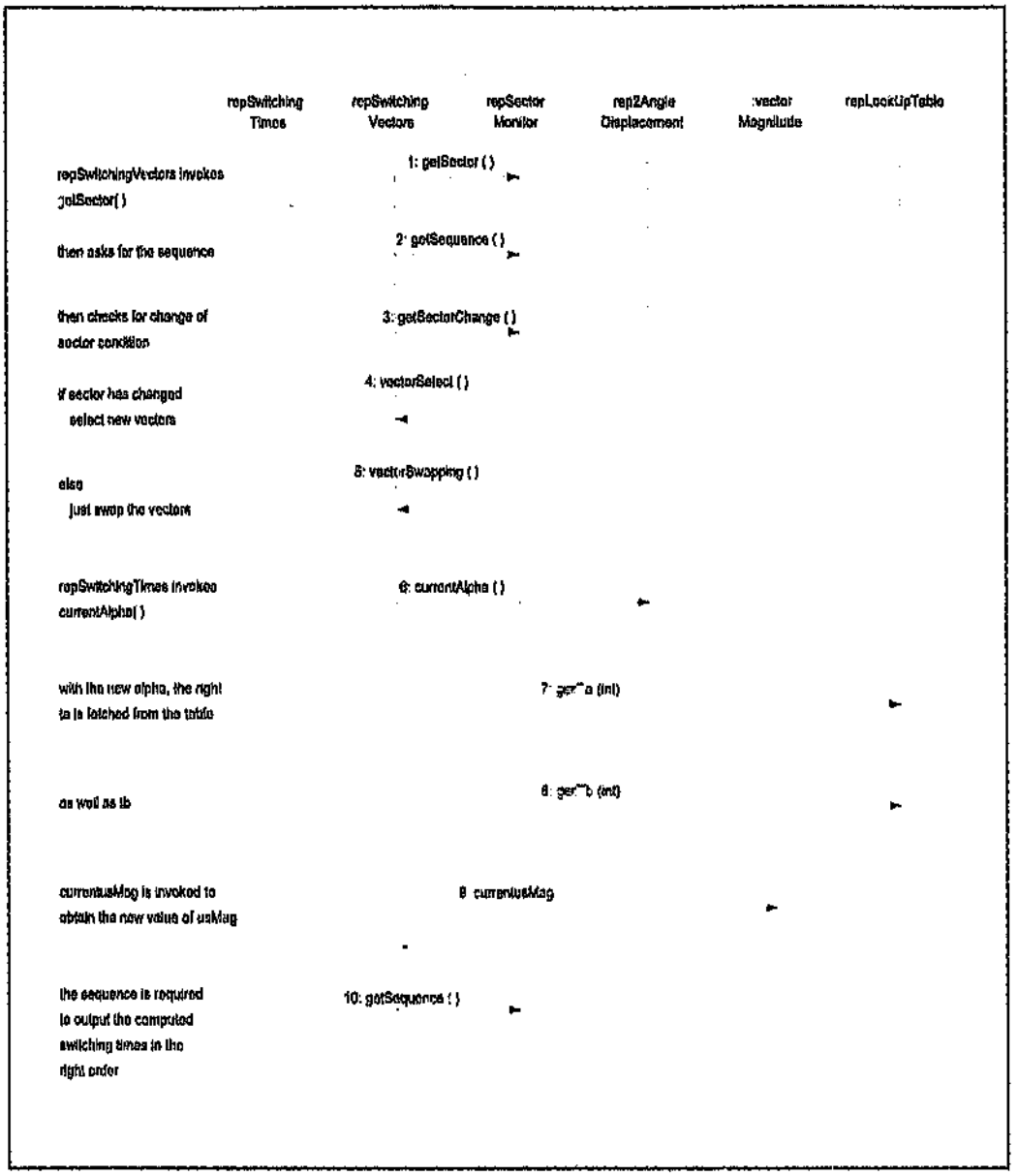


Figure 20 (a): Interaction Diagram - Calculating The Switching Parameters

Appendix C: Source Code for SDEC0401.C

```
/*                          SDEC0401.C                          */

/* Program to perform the 3x3 PWM control of the microprocessor  */

/* Version 0.1 */
/* Author  Maurice Grays  */
/* Date  18/10/95 */

/* Sampling frequency : 1 KHz */

/* Description
   This version performs the timing calculations fully on-line and does
   not implement the phase lock loop algorithm
*/

/* Results
   Results show that this method is not practical, as too much time is
   spent on doing the on-line calculations. This version failed to completed
   the main cycle on one sampling interval, even at such a low frequency. Out
   of this emerged the following aspects become apparent

              (1)  On-line calculation of the switching times is not feasible

              (2)  Float variables and their manipulations are to be avoided

              (3)  Computational C functions, like fmod and floor, are to be avoided
*/


#include <math.h>

#pragma language = extended       /* Enable the use of extended keywords */
#include <ch83905.h>              /* internal registers define  */
#include <ch03005.h>              /* interrupt vector addresses define  */

/*          Define Constant Terms          */

#define pi 3.141592654
#define two 2*pi
#define sqrt3by2 0.5*pi*sqrt(3)
#define tClock 12.5E-9                    /* At a clock frequency of 16MHz */
#define sqrt2by3by2by2and3 sqrt(3)
/*          Define Parameters          */

#define tSampling 1E-3                    /* At a Sampling frequency of 1kHz */
#define limit tSampling/4*tClock
#define tNominal 50
#define ta1 ton*tSampling/tClock
#define tb1 ton*tSampling/tClock



/*          Running Variables          */

int sequence, counter, rounds;

int uTemp, uTemp1, uTemp2, uTemp3, tATemp, tBTemp, tCTemp, tTemp

int ui, uA, uZero, ti, tA, tCount;

int tCount

float phase, angle

*****************************************************************
*                                                               MAIN
*****************************************************************

void main (void)
{                 /* opening bracket for main loop */

/*          Running variables initialisation          */

sequence = 1
counter = 1
rounds = 1
uATemp = uB1 = uTemp = 20; uZeroTemp = 1
ui = 47; uA = 75; uZero = 2
ti = tInitial = 0; tInitial = tInitial = tInitial

/*          Initial Registers Initialisation          */

ITCR1 = 0x20;                     /* Set ext INTA interrupt mode as Rising Edge
                                     Select Internal Clock CLOCK/4 */
ITCR2 = 0x01;
ITCR3 = 0x02;
ITCR4 = 0x03;
```

```
PPDR = 0.                       /* Set all unbonded possible latch to zero (low) */

P1DDR = 255.                    /* All Port 1 pins set for output */
P1DR = mZero.                   /* Set Port 1 initially to mZero */

ITU_CRA2 = 10000.               /* load new value of CRA2 */

P1DR = mZero.                   /* output appropriate signal */

ITU_TCT1 = 0.       /* start Timer T2 */

}


/* Interrupt Acknowledge */

only_expt [ITU_CMIRA2] void timerT2Interrupt(unsigned) {

{
ITU_TSTR = 0.     /* Stop timer T2 */

ITU_TSR2 &= ~1.  /* acknowledge timer 2 interrupt compare interrupt */

/* Note "error_exp (0)," the reset bit 1 bit of the CCR is not necessary
   here since at this stage we do not have to read any other higher
   priority interrupt that can interrupt this on going interrupt
   service routine. The 'dummy' will interrupt will be a Non Maskable Interrupt
   (NMI) and is not can cause an interrupt at any time, whether the 2 bit of
   the CCR is set or not. */

ITU_CRA2 = 10000;               /* load new value of CRA2 */

P1DR = mZero                    /* output appropriate signal */

ITU_TSTR = 1        /* start Timer T0 */
}


interrupt [ITU_IMIA0] void timerT0InterruptRoutine (void)

{
ITU_TSTR = 0     /* Stop timer T0 */

ITU_TSR0 &= ~1.  /* acknowledge timer 0 interrupt compare interrupt */

ITU_CRA1 = 1;    /* load new value of CRA1 */

P1DR = 1;                       /* output appropriate signal */

ITU_TSTR = 2        /* start Timer T1 */
}


interrupt [ITU_IMIA1] void timerT1InterruptRoutine (void)

{
ITU_TSTR = 0     /* Stop timer T1 */

ITU_TSR1 &= ~1;  /* acknowledge      interrupt compare interrupt */

ITU_CRA2 = 1;    /* load new value of CRA2 */

P1DR = 4;                       /* output appropriate signal */

ITU_TSTR = 4.       /* start Timer T2 */
}


interrupt [ITU_IMIA2] void timerT2InterruptRoutine (void)

{
ITU_TSTR = 0     /* Stop timer T2 */
                        * 0 hit 224. Need to check */

ITU_TSR2 &= ~1;  /* acknowledge timer 2 interrupt compare interrupt */
                        * Must also try ITU_TCR2 = 0; as a similar way */

ITU_CRA1 = mHigh                /* load new value of CRA1 */

P1DR = mZero                    /* output appropriate signal */

ITU_TSTR = 0        /* start Timer T0 */
}
```

Appendix B: Source Code for SDEC0301.C

```
/*                                  SDEC0301.C                                  */

/* Program to test the interrupt cycle of the SDES inverter control project */

/* Author  Nicholas Bagnall */
/* Date  27/08/98 */

#include <math.h>

#pragma language = extended     /* Enable the use of extended keywords */
#include <ina90c08.h>           /* internal registers define  */
#include <ina90c003.h>          /* interrupt vector addressed define  */

/*  ------  Define Constant Terms  ------  */

#define pi 3.141592654
#define Hz Hz/1e3
#define kHz Hz*1e3
#define MHz kHz*1e3
#define fClock 12.5E6                   /* At a clock frequency of 12.5MHz */
#define half fullClock
#define qtr fullClock

/*  ------  Define Parameters  ------  */

#define fSampling 12.2                  /* At a Sampling frequency of 12Hz */
#define fMains 50
/*#define pendingTimerDelay
#define off
#define red
#define grn */



/*          Running Variables          */

int sequence, motor, counter

int pTemp, mTemp, pZeroTemp, fTemp, fMotorTemp, temp

int pL, mL, pZero, fL, fR, fDiff.

float loopDelay, timerDelay

float time

float phase, alpha


/*****************************************************************
*                                MAIN                            *
*****************************************************************/

void main(void)
{

/*          Running Variables Initialisation          */

sequence = 1
motor = 1
counter = 1
pTemp = 41 mTemp = 39 pZeroTemp = 7
pL = 49 mL = 15 pZero = 7
fL = fSampling-4 fR = fSampling-8 fDiff = fSampling-4

/*          H3048 Registers Initialisation          */


P5IO  P5=0x7     /* Select P5IO register that is as following format
                    Select External Interrupt Edge Polarity */
P5IO  P4=0x7
P5IO  P3=0x7
P5IO  P2=0x7
P5IO  P1=0x7

P5IO  P5=0x7     /* Select P5IO as an output compare register
                    P5 is output of the T3OC1 A and B match compare register
                    P5IO compare as output */
P5IO  P4=0x7
P5IO  P3=0x7
P5IO  P2=0x7
P5IO  P1=0x7

P5IO  P5=0x7     /* Enable P5IO interrupt [illegible]
                    to permit the H3048 */
P5IO  P4=0x7
P5IO  P3=0x7
P5IO  P2=0x7
P5IO  P1=0x7     /* [illegible] as required by [illegible] */

P5IO  P5=0x7     /* [illegible] as input for P5IO flag
                    edge selected [illegible] P5IO register */

P5IO  P=0x7
```

Appendix A: Source Code for SDEC0201.C

```
/*                                  SDEC0201.C                          */

/* Program to calculate fa, fb and fd of the SVM equations (1&2) */

/* Author: Hisaaraj Bapoo */
/* Date: 12/09/95 */

#pragma language = extended

#include <C:\math.h>

void main (void)
{

#define fSampling 12.8                  /* fSampling = 3 kHz */
#define pi 3.14159265359
#define ka 0.63245553E-4
#define kb 0.9E-4

float alpha, ka, fb, fd;

unsigned i;  /* assume a value of unity of 1 */

ka= 2pi*(desired fourier coefficient)*(unity gain).
kb= kb*pi*alpha.
fSa= fSampling*ka
```

3 Hardware Implementation

A printed-circuit board (PCB), was designed for the purpose of this application. The main purpose of this Inverter Control Card[1] is to hold the microcontroller and its support elements. It is a two-layer card consisting essentially of:

- a power supply circuitry
- the microcontroller
- interfacing circuitry between the microcontroller and the sensed parameters
- interfacing circuitry between the microcontroller and the drive circuitry. The drive circuitry is located on a separate card.

Figure 2 shows the block diagram of the Inverter Control card. Sensing and output of the drive signals are performed via the connectors X1, X2, X3 and X4. The actual dimension of the card are 234mm x 160mm. The PCB dimensions were so chosen so as to fit into the card cage of the targeted UPS (MP 3000 MKII). The schematic file for this card, designed using the CAD package Tango Pro. is available in the file SDEI0501.SCH. In order to reduce the cost of this first prototype, the card was not fitted with a ground plane.

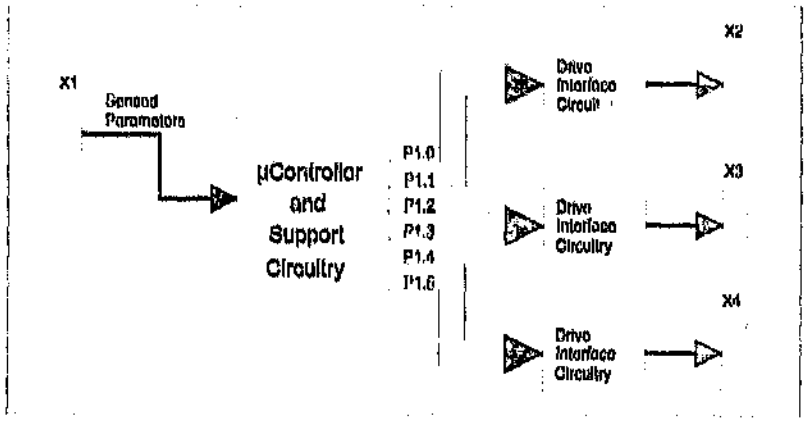


Figure 2: Block diagram of Inverter Control Card

[1] The interface with the drive circuit as well as several other design features on the card were borrowed from the Phase Control Card, PCB 905030, designed by Noel Hulley, Meisaner, in 1994

to pay for the processing speed limitations of the microcontroller and is common to real-time embedded systems. In fact it is not rare to see applications where hard real-time algorithms are implemented directly in assembly language; visibility in those cases is even worse.

SDEC0408.C	Same as version 1.1 but with the real sensing and A/D conversion as tested in SDEC0501.C integrated to the program. Sampling frequency: 6.25 kHz	The output waveforms were unstable. Frequent up and down "swaying" movements were observed, combined with frequent overshoots. This suggests that the switching frequency of 6.25 kHz is too high for this version. Actually the addition of the A/D conversion and the associated new calculations increases the execution time by about 40 microseconds.
SDEC0410.C	This version contains several changes to version 0.8. A hysteresis algorithm was included to avoid unwanted changes in the sensed parameter. In order to cater for the hysteresis algorithm and other minor changes, the sampling frequency was lowered to 4 kHz. This version also contains an inverter-off subroutine.	Program performs successfully. This version satisfies all the functional specifications of the project. It is the first approved version of the software.
SDEC0501.C	This program is to test the A/D conversion (scan mode) and sensing of the microcontroller. It also includes the immediate calculations to be performed on those parameters in each sampling interval. Those calculations are the computation of usMag and that of pheta. Variations of the sensing parameters was simulated using a voltage divider network and potentiometers.	Program ran successfully, with variation in input voltage being followed by the A/D data registers. The time taken to complete the A/D conversion and to perform the computation of usMag and pheta, was about to 40 microseconds. Also small variations in the register values were observed for fixed input voltages at different runs of the conversion loop.

Going from the earlier versions to the later ones, a general trend towards more simplicity is observed, both in the language and the structures used. This is achieved by breaking the high abstractions of the language into more basic algorithms. Although reduction in execution time, which is the prime objective of this exercise, is achieved, the costs in avoiding a higher level of abstraction, appear in the length and the clarity of the program. Going down the list, from one version to the other, the program becomes more obscure; high-level functions (like fmod) are translated into several lines of code. The mapping between the design and the coding becomes more and more difficult to follow. This is the price

SDEC0404.C	This version is similar to version 0.3, but with several changes to eliminate the few float operations left, especially those operations that involve phete and alpha1. In addition the number of divisions per sector, maxSectorDiv, was increased from 60 to 240, hence giving a higher accuracy of the switching times values. Sampling Frequency: 5 kHz	Program ran successfully, performing the on-line calculations well on time, at each iteration of the computation loop.
SDEC0405.C	Following the time saving achieved in version 0.4, this present version operates at a higher frequency, 6.25 KHz. In addition this version contains transitory states to ensure a full Gray code flow. usMag and vlav are now properly computed, instead of being merely assigned values as in the previous versions. Sampling Frequency: 6.25 kHz	Program ran successfully, but results show that it is performing at its limit in terms of available execution time per sampling interval. In fact when tested at 7142 Hz the software failed to produce the right output pattern.
SDEC0406.C	This version is si: ilar to version 0.5, but without the additional transitory states that ensure a full Gray code flow. This function as well as the time delay to cancel the effect of tOff of '' switches is to be taken care of by the hardware. In addition it contains a subroutine for the 300% OL case and instructions to send a signal on a Statjc Switch Inhibit condition. Sampling frequency: 6.25 kHz	Program ran successfully, as in version 0.5. When later tested on the emulator, typical PWM drive pulses switching between +/- 12V could be observed at the output of the IGBT drive cards. This result demonstrates clearly the succesfull implementation of SVM into producing PWM pulses for a three-phase system.
SDEC0407.C	Same as version 0.6 but with several changes to accommodate for the sensing of Reserve and Inverter parameters, and their A/D conversions. Note that the real sensing and A/D conversions are not yet performed at this stage. Sampling frequency: 6.25 kHz	Program performed successfully, suggesting that the sensing and A/D conversion could be integrated.

2.2 Summary of Software Revisions

Coding was performed in an iterative loop involving design and testing as well. The major developments in the design-coding-testing loop (those that justified a revision update) are summarised in table 1. Each revision is shown it its complete form in appendix A to L.

Table 1: Summary of intermediate software revision

Source Code	Description	Brief Performance Result
SDEC0201.C	To calculate ta, tb and t0Half of the SVM algorithm	Time taken for calculations was in the order of milliseconds suggesting that the idea of having full on-line computation is impractical.
SDEC0301.C	To test the interrupt cycle algorithm	Ran successfully, without any overflow errors or other interruptions.
SDEC0401.C	To perform the full SVM control - without the phase-locked-loop algorithm. All timing calculations are performed on-line. Sampling Frequency: 1 kHz	Program failed to complete the calculations in one sampling interval. This confirms the findings of version 0.1
SDEC0402.C	In this version the switching times are first generated and stored in a look-up-table in the initialisation stage. Also the use of sophisticated C functions, like fmod and floor, were eliminated in the main cycle loop. Sampling Frequency: 1 kHz	The program ran successfully with all calculations being performed within one sampling interval. The ITU registers and the output port register were updated appropriately.
SDEC0403.C	This version has fewer float manipulations in the main cycle loop. Furthermore many of the expressions defined as #define directives were eliminated. Sampling Frequency: 5 kHz	Program ran successfully - although the number of lines of code has now been increased considerably and more variables have been added.

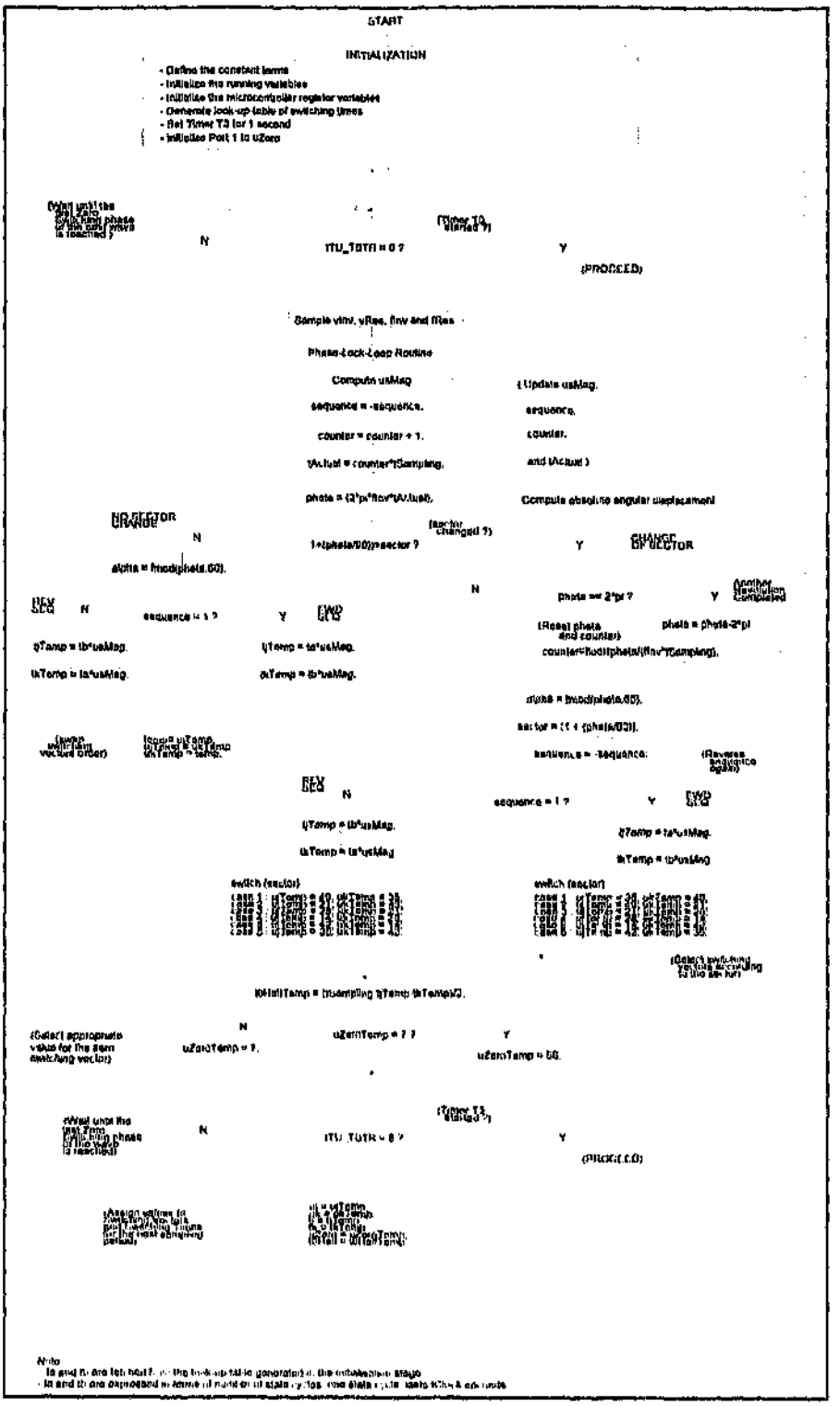


Figure 1: Flow Chart Diagram of the *main cycle*

2 Software Implementation

2.1 Flow Diagram Representation

The program was written in C instead of an OOL like C++. This is mainly a consequence of the compiler limitation, mentioned already in section 3.2 of the Functional Specifications, SDE 106.

As an intermediate abstraction between the object-oriented representation and the exercise of coding in a non-OOL (C), the flow diagram in figure 1 was developed. It is mainly derived from the state transition diagrams and the class diagrams presented in SDE 108. The flow diagram is a helpful tool for bridging the gap between OODs and non-OOLs implementation.

The flow chart in figure 1 relates to version 1.0 of the software. It uses pseudocode to show the different activities. Including codes in the chart, is a convenient way of depicting the activities but those codes are not necessarily a true image of the actual coding in the program. In the actual program, different codes might be used to implement the same functions to meet time and space constraints; e.g. In the actual program the function `fmod` is replaced by a set of instructions in order to reduce the execution time. The actual codes are displayed in appendix K.

1.7 ISO 9001 Requirements Traceability

- ISO 9001 Clause 4.4.5 - Design Output

```
ta=0;bit1,
tb=0;bit1;

if (sequence == 1)                  /* forward sequence */
{                                   /* opening bracket for forward sequence */
qTemp = (tq*qMSB) >> 16;
kTemp = (tk*qMSB) >> 16;

}                                   /* closing bracket for forward sequence */

else                                /* reverse sequence */
{                                   /* opening bracket for reverse switching */
qTemp = (tq*qLSB) >> 16;
kTemp = (tk*qLSB) >> 16;

}                                   /* closing bracket for reverse switching */

temp = qTemp;                       /* swap switching */
qTemp = qkTemp;                     /* vectors */
qkTemp = temp                                   /* order */

}                                   /* closing bracket for "else"  vector changes */


if (qTemp == 0) qTemp = 1Ma;
if (kTemp == 0) qTemp = 1Ma;        /* to prevent unpredictable results due to starting a timer for 0 seconds */

qSignalTemp = mSinc(qTemp,kTemp);                       /* Compute qSignal temp */

if (oldSeqTemp == 0)                                    /* change */
qTempTemp = 50;                     /* the zero */
else                                                    /* switching */
qTempTemp = 0;                                          /* vector */

while (qFU_TCR1 == 232)             /* wait until */
{                                                       /* last two sequence */
}                                                       /* of waveforms is reached */

qt = qTemp                          /* addressing the */
qk = qkTemp;                        /* parameter */
tq = tqTemp;                        /* values */
tk = tkTemp                         /* for the */
qTime = qTimeTemp;                  /* waveform */
qSignal = qSignalTemp;              /* algorithm */


}                                   /* closing bracket for the the forever loop */

}                                   /* closing bracket for main */

/* Interrupt Subroutines */

interrupt [4] void timer1Interrupt(void)
{
qFU_TCR1 = 0;   /* Stop timer 1 */

if(qFU_TCR1 & 1)  /* acknowledge timer 1 interrupt compare interrupt */
                  /* Must also try qFU_TCR1 = 0  as a simpler way */


/* Note "wrong" use (0) to reset the bits of the SCR1 as was necessary
   here since at this stage we do not have in mind any other higher
   priority interrupt that can interrupt this service interrupt
   service routine. The 300% cpu utilisation will be a non-desirable interrupt
   (MSB) and so it can cause an interrupt at any time whether the others of
   the SCR1 is set or not */


qFU_tMSB = qTime;                   /* load new value of tMSB1 */

PtpA = qVec;                        /* output appropriate signal */

qFU_TCR1 = 1;       /* start Timer 1() */
}

interrupt [5] void timer0Interrupt(void)
{
qFU_TCR0 = 0;   /* Stop timer 0 */

if(qFU_TCR0 & 1)  /* acknowledge timer 0 interrupt compare interrupt */

qFU_qMSB = 0;       /* Switch off all bits */

qFU_qLSB = 0;   /* load new value of tLSB1 */

PtpA = 0;                           /* output appropriate signal */

qFU_TCR0 = 2;       /* start Timer 1() */
}

interrupt [6] void timer2Interrupt(void)
{
qFU_TCR2 = 0;       /* Stop timer 2 */

if(qFU_TCR2 & 1)  /* acknowledge timer 2 interrupt compare interrupt */

PtpA = 0;           /* switch off all bits */

qFU_qMSB = 0;   /* load new value of tMSB1 */
```

```
}
}

/* Sample setting and filtering time */

wtReg = 63817,   /* Move values are assigned to setting and */
filtering = 50.                         /* filtering but more for are sampled these
                                           two code lines must be removed */

sequence = sequence.                    /* Change sequence */

counter = counter + 1.                  /* Go to */
start = counter * 15.           /* start sampling period */

if (difference< 481) {difference = 0;}  /* Phase lock loop  counter frequently */
filter = Natural.               /* equal to the transmit frequency of the  */
else                                    /* filtered frequency is outside the window. */
filter = filtering              /* else it is equal to the filtering frequency */

shift = (0.360*start*filter.    /* angle displacement w.r.t the horizontal */

if (phase < 60) {alpha = phase. sectorUpdate = 1.}
if (phase>= 120) && (phase.>= 600 {alpha = phase 60, sectorUpdate = 2 }
if (phase>= 180) && (phase >= 120) {alpha = phase 120. sectorUpdate = 3 }
if (phase>= 240) && (phase >= 180) {alpha = phase 180. sectorUpdate = 4 }
if (phase>= 300) && (phase >= 240) { alpha = phase 240. sectorUpdate = 5 }
if (phase>= 360) && (phase >= 300) {alpha = phase 300. sectorUpdate = 6 }
if (phase >= 360){
alpha = phase 360 sectorUpdate = 1.
if (alpha> =samval
counter=0.
else
counter=1.
}

 /* alpha = Angle we are have vector of the present sector */

if (sector != previousVector)           /* Sector change ? */
{               /* opening bracket for sector change* */

/*              change of sector                */

sector = previousSave.

/*      sector to and it from the look up table         */

if (alpha>alpha0[sec])
{sec=1;C0}
else{sec=2}

sequence = sequence     /* remember  sequence does not change when there is
                        a sector change */

if (sequence == 1)
{               /* opening bracket for "if sequence == 1"  change of sector loop */

yTemp = (int)yWeight * 10
xTemp = (int)xWeight * 10

switch (sector)
{               /* opening bracket for "switch(sector)"  sequence 1 */
case 1: {xTemp = 85; yTemp = 60; break }        /* select appropriate */
case 2: {xTemp = 40; yTemp = 75; break }        /* switch up */
case 3: {xTemp = 21; yTemp = 70  break }                /* up down */
case 4: {xTemp = 30; yTemp = 10; break }
case 5: {xTemp = 10; yTemp = 45; break }
case 6: {xTemp = 90; yTemp = 15  break }
}               /* closing bracket for "switch(sector)"  sequence 1 */

}               /* closing bracket for "if sequence == 1"  change of sector loop */


else                                    /* sequence = 2 */
{               /* opening bracket for "else"  sequence not equal to 1 */

yTemp = (int)yWeight * 10.
xTemp = (int)xWeight * 10

switch (sector)
{               /* opening bracket for "switch(sector)"  sequence not equal to 1 */

case 1: {xTemp = 80; yTemp = 55; break }        /* select appropriate */
case 2: {xTemp = 41; yTemp = 44; break }        /* switching */
case 3: {xTemp = 20; yTemp = 21; break }                        /* up down */
case 4: {xTemp = 10; yTemp = 70; break }
case 5: {xTemp = 9;  yTemp = 15; break }
case 6: {xTemp = 75; yTemp = 42; break}
}               /* closing bracket for "switch(sector)"  sequence not equal to 1 */

}               /* closing bracket for "else"  sequence not equal to 1 */

}               /* closing bracket for sector change */

else
{               /* opening bracket for "else" of sector change */

/*      output to the driver the look up table          */

}
```

```
C task and main level
{                       /* opening bracket for main loop */

/*          Running Variables Initialization ... ... ... ...*/

int n=0;  /* use scheduled task to be able to see the exact value displayed*/

iSampling = 3E-4,
i2 = RoundSampling*1000000;
k200=0.0010,
sequence = -1,
sector = 1,
counter =  1,
vTemp = 40, vTemp = 35, vZeroTemp = 2,
vj = 40, vk = 38, vZero = 2,

timeVar = PeriodSampling*4*MaxAdc;

N = decimal, de = decimal, dNull = decNull

da = Dir*sin,
db = d*cos(2*pi*ip
ka3 = ka*iSampling*dCheck,
kb1 = kb*iSampling*dCheck,
cosOneDegree = cos(pi/2),
iSamp = floor(iSampling*dCheck),
alphaMax = 60*degreesPerDay,
vsens = 1563*(Hysteresis*iSampling)2,
dAdc= 0.0,

/*          PIC16F84 Registers Initialization ... ... ... ...*/


MCU_TCON = 32,          /* Select GRA compare match by Clearing Routine
                           Select Internal Clock (FOSC/4) */
MCU_TCR1 = 32,
MCU_TCR2 = 32,
MCU_TCR3 = 32,
MCU_TCR4 = 32

MCU_TIOR0 = 2           /* Select GRA as an output compare register
                           0 is output at the TIOCA pin at GRA compare match
                           GR0 cleared, no output */
MCU_TIOR1 = 2,
MCU_TIOR2 = 2
MCU_TIOR3 = 2,
MCU_TIOR4 = 2,

MCU_TIER0 = 1           /* Enable GRA interrupt to occur when it is
                           requested by IMFA */
MCU_TIER1 = 1
MCU_TIER2 = 1
MCU_TIER3 = 1
MCU_TIER4 = 0           /* Disable all interrupt requests by timer TA */

SYSCR = 31              /* Select OI bit as CCR of p-ch bit (NMI falling
                           edge software  On-chip RAM enabled */

MPTA = 0,
IPRC = 0                /* Set all interrupts priority level to zero level */

P1DDR = 2F5             /* All Port 1 pins set for output */


MCU_GRA2 = 10000        /* Set an output delay at time before going into
                           The main loop */

P1DR = 0Fpa,            /* Set Port 1 initially to zero */

/*          CONSTANTS LOOK-UP TABLE          */

for i=0; alphaval <= MAX*sin(2*pi/n * i); alpha = alpha +alphaMin
{
alphaVal = pi*alpha 180
sinalphaVal = sin(alphaVal)
tab[i] = Main.lng1*sign.alphaMain.sin(OneDegree)*sinAlphaVal;
i+1 = floor(tab1*sinAlphaVal);
}

/* redefining the following because above long gives wrong values for
dcount, dTemp, and dzero.  the two are used using a MEASELTaMinV
of 60 degrees only.  Note these values only apply for a MEASELTaMinV of
60 only */

dDMax = 2048            /*45 pi */        /*dMin */
dTMax = 1638    /*36 */         /*2048 */
dZero = 1638    /*36 */         /*2048 */

alphaVal                /* reset alpha after the above calculation loop */

dAdc = 0                /* reset timer 1 */

/*          MAIN LOOP                   */

for ( ;
{                               /* Opening Bracket for the for loop */

while (TIFR1 & 0x01) == 0x01;           /* Loop here as long as TIFR1 bit 0 is zero
                                           wait */
```

Appendix E: Source Code for SDEC0403.C

```
/*                              SDEC0403.C                          */

/* Program to perform the full SVM control of the inverter */

/* Version 0.3 */
/* Author: [illegible] */
/* Date: 30/10/95 */

/* Sampling Frequency: 5 KHz */

/* Description
   In this version effort was made to remove the float manipulations in the
   main cycle loop. This involved techniques such as pre-multiplying the float
   variables by a large integer value (like 10000) or performing division and
   multiplication by bit shifting the variables. Furthermore many of the
   expressions defined as #define macros into global variables. The
   reason being that each time they are called, the expression they define is
   calculated. This results in a waste of calculation time.
*/

/* Results
   These changes effected a reduction in calculation time and hence permitted
   a [illegible] improvement of the sampling rate. The price to pay is that
   [illegible] of code; clarity [illegible] and C++ [illegible]
   [illegible] in [illegible] that C offers
*/


#include <math.h>

#pragma language = extended        /* Enable the use of extended keywords */
#include <io80C196.h>              /* internal registers defns */
#include <int80C196.h>             /* interrupt vector addresses defns */

/*              Define Constant Terms              */

#define pi 3.141592654
#define tClock 62.5E-9                          /* At a clock frequency of 16MHz */
#define MAXSECTORDIV 60

/*              Define Parameters                  */

/* #define tSampling 1E-3 */                   /* At a Sampling frequency of 1kHz */
#define tNominal 50


/*              Defining Variables                 */

int sequence, sector, counter

int [illegible] [illegible] [illegible] [illegible] temp

unsigned int [illegible] [illegible]

[illegible]

unsigned long [illegible]                       /* [illegible] changed to int */

float [illegible] [illegible]

float [illegible] [illegible] alpha             /* changed to int */

int [illegible]

float alpha[illegible] gamma

float tSampling

[illegible]

[illegible]

[illegible]

int [illegible]

[illegible]

[illegible]

[illegible]

[illegible]
[illegible]
[illegible]
[illegible]

/************************************************************************/
/*                                 MAIN                                 */
/************************************************************************/
```

```
}               /* closing bracket for "else" - never changed */

iDiffLoop = iClockTemp - iTempLoop;                     /* compute iDiffClock */

/* Interface */                                         /* change */
mZeroFlag = 0;                  /* the time */
else                                                    /* continuing */
mZeroFlag = 1;                                          /* vector */

while (P1.1 == 1)               /* wait until */
{                                                       /* last one expires */
}                                                       /* or interface is reached */

ui = uiTemp;                    /* updating the */
uk = ukTemp;                    /* parameter */
uj = ujTemp;                    /* values */
ui = uiTemp;                    /* for the */
mZero = mZeroFlag;              /* interrupt */
iDiff = iDiffTemp               /* algorithm */

}               /* closing bracket for the Do Forever loop */

}               /* closing bracket for main */


/* Interrupt Subroutines */

interrupt [IE0_VECTOR] void Int0Function(void)          /* Int0 */
{
TF0 = 0;    /* Clear timer TF0 */

IE0 = 0;    /* acknowledge since T interrupt complete interrupt */

/* Note: Since only the value of the CCFs is now assigned
   here since at this stage we do not know    what one other bytes
   currently exist that can interrupt the program
   service routine. The 8051 will interrupt will be a Non-Maskable Interrupt
   Right and so it can - only an interrupt of the same  maskable flag bit at
   the IE0 is use or not */

TH0 = CCF0;             /* load new value of CCAP */

P1.0 = mZero;           /* output appropriate signal */

TL0 = 0;    /* start timer T0 */
}

interrupt [IE1_VECTOR] void Int1Function(void)          /* Int1 */
{
TF0 = 0;    /* Clear timer T0 */

IE1 = 0;    /* acknowledge since 0 interrupt complete interrupt */

P1.0 = 0;   /* Switch off all bits */

TH0 = CCF1;             /* load new value of CCAP */

P1.0 = mZero;           /* output appropriate signal */

TL0 = 0;    /* start Timer T0 */
}


interrupt [TIMER2_VECTOR] void Timer2Function(void)
{
TF0 = 0;    /* Clear timer T0 */

TF2 = 0;    /* acknowledge since T interrupt complete interrupt */

P1.0 = 0;   /* Switch off all bits */

TH0 = CCF2;             /* load new value of CCAP */

P1.0 = mZero;           /* output appropriate signal */

TL0 = 0;    /* start Timer T0 */
}
```

```
}               /* closing bracket for "if(sequence == 1)  change of sector case "

else                                    /* sequence = 2 */
{               /* opening bracket for "else"  sequence not equal to 1 */
ujTemp = ub*subReg
ukTemp = ub*subReg;

switch (sector)
{               /* opening bracket for "switch(sector)"  sequence not equal to 1 */

case 1: {ujTemp = 49; ukTemp = 35; break;}          /* Select appropriate */
case 2: {ujTemp = 21; ukTemp = 49; break;}          /* switching */
case 3: {ujTemp = 28; ukTemp = 21; break;}                          /* sectors */
case 4: {ujTemp = 14; ukTemp = 28; break;}
case 5: {ujTemp = 42; ukTemp = 14; break;}
case 6: {ujTemp = 35; ukTemp = 42; break;}
}               /* closing bracket for "switch(sector)"  sequence not equal to 1 */

}               /* closing bracket for "else"  sequence not equal to 1 */

}               /* closing bracket for sector change" */

else
{               /*opening bracket for "else" of sector change" */


/*      select ra and rb from the look up table         */

sequence = alpha15;

getOutOfLoop = false;

switch (sequence)

case 0 {

for (alphaLow=alpha1.alphaHigh=alpha1.row=Start1; getOutOfLoop==false; alpha, row = =alphaMin  alphaHigh = =alphaMax; ++)
{
if (alpha >= alphaHigh && alpha <= =alphaLow)
{
ra=tb[0];
rb=tb[1];
getOutOfLoop = true;
break;
}}}

case 1 {

for (alphaLow=alpha2.alphaHigh=alpha2.row=Start2; getOutOfLoop==false  alpha, row = =alphaMin  alphaHigh = =alphaMax; ++)
{
if (alpha >= alphaHigh && alpha <= =alphaLow)
{
ra=tb[2];
rb=tb[3];
getOutOfLoop = true;
break;
}}}

case 2 {

for (alphaLow=alpha3.alphaHigh=alpha3.row=Start3; getOutOfLoop==false  alpha, row = =alphaMax  alphaHigh = =alphaMin; ++)
{
if (alpha >= alphaHigh && alpha <= =alphaLow)
{
ra=tb[0];
rb=tb[1];
getOutOfLoop = true;
break;
}}}

case 3 {

for (alphaLow=alpha4.alphaHigh=alpha4.row=Start4; getOutOfLoop==false  alpha, row = =alphaMax  alphaHigh = =alphaMin; ++)
{
if (alpha >= alphaHigh && alpha <= =alphaLow)
{
ra=tb[2];
rb=tb[3];
getOut.. = true
break;
}}}
}

if (sequence == 1)                  /* forward sequence */
{               /* opening bracket for forward sequence */
ujTemp = ub*subReg
ukTemp = ub*subReg
}               /* closing bracket for forward sequence */

else                                    * reverse sequence *
{               /* opening bracket for reverse switching *
ujTemp = ub*subReg
ukTemp = ub*subReg
}               * closing bracket for reverse switching *

temp = ujTemp                   * swap switching *
ujTemp = ukTemp                         * values *
ukTemp = temp                                   * also*
```

```
if (phase <  240) && (phase >= 180) {alpha = phase-180; sectorUpdate = 4;}
if (phase <  300) && (phase >= 240) {alpha = phase-240; sectorUpdate = 5;}
if (phase <= 360) && (phase >  300) {alpha = phase-300; sectorUpdate = 0;}
if (phase == 360){alpha = phase-360; sectorUpdate = 1;}
  /* alpha = Angle we i final vector of the present sector */

if (sectorUpdate  != previous)              /* Sector change ? */
{                   /* opening bracket for sector change? */

/*                  Change of sector - - - - - - - - - - */

previous = sectorUpdate;

if (phase >= 360)                                   /* Another rev completed ? */
{                   /* opening bracket for "if (phase >= 360)" */

/* Another rev completed */
if (alpha <= quadrant)
counter =0;
else
counter = 1;
}                   /* closing brackets for "if (phase >= 360)" */


/*      Select tb and tb from the look up table          */

quadrant = alpha/15;

getOutOfLoop = false;

switch (quadrant){

case 0 :{

for (alphaLow = alpha1.alphaHigh = alpha+1,++Step1, getOutOfLoop = false, alphaLow <= alphaMin alphaHigh = alphaMax;++ )
{
if (alpha <= alphaHigh && (alpha >= alphaLow))
{
tb=th[0]
tb = tb[1];
getOutOfLoop = true;
break;
}}}

case 1 :{

for (alphaLow = alpha2.alphaHigh = alpha-1,--Step2, getOutOfLoop = false, alphaLow <= alphaMin alphaHigh = alphaMax;++ )
{
if (alpha <= alphaHigh && (alpha >= alphaLow))
{
tb=th[0];
tb=th[1];
getOutOfLoop = true
break;
}}}

case 2 :{

for (alphaLow = alpha3.alphaHigh = alpha+1 ;++Step3, getOutOfLoop = false, alphaLow <= alphaMin alphaHigh = alphaMax;++ )
{
if (alpha <= alphaHigh && (alpha >= alphaLow))
{
tb = th[0]
tb=th[1];
getOutOfLoop = true
break
}}}

case 3 :{

for (alphaLow = alpha4,alphaHigh = alpha+0 ;--Step4, getOutOfLoop = false, alphaLow <= alphaMin alphaHigh = alphaMax;++ )
{
if (alpha <= alphaHigh && (alpha >= alphaLow))
{
tb=th[1];
tb=th[0];
getOutOfLoop = true;
break;
}}}
}


sequence = sequence     /* sequence sequen = does not change when there is
                        a vector change */

if (sequence == 1)
{                   /* opening bracket for "if sequence == 1" change of sector loop */
hiTemp = tb*walking
thTemp = th*walking;

switch (sector)
{                   /* opening bracket for "switch-case sector sequence" */
case 1 :{ loiTemp = 5; thTemp = 29; break;}          /* select approximate */
case 2 :{ loiTemp = 44; thTemp = 21; break;}         /* switching */
case 3 :{ loiTemp = 25; thTemp = 28; break;}             /* sectors */
case 4 :{ loiTemp = 28; thTemp = 10; break;}
case 5 :{ loiTemp = 10; thTemp = 47; break;}
case 6 :{ loiTemp = 47; thTemp = 29; break;}
}                   /* closing bracket for "switch-case sector sequence" */
```

```
alphaX1 = alphaL1 + alphaXm.
alphaY2 = alphaL2 + alphaYm.
alphaX3 = alphaL3 + alphaXm.
alphaY4 = alphaL4 + alphaYm.

iStart1 = 0.
iStart2 = iStart1 + MAXSECTORSPW4
iStart3 = iStart2 + MAXSECTORSPW4.
iStart4 = iStart3 + MAXSECTORSPW4.

psmax = 1000*(decmax*(Sampleper2.

/*      ... 11000 Registers Initialisation                    */

itU0 TCR3 = 32.                 /* Select GPA compare match as Timer#0 Source
                                   Select Internal Clock (16MHz) */
itU0 TCR1 = 32.
itU0 TCR2 = 32.
itU1 TCR3 = 32.
itU2 TCR4 = 32.

itU0 TIOR0 = 2.                 /* Select GPA as an output compare register
                                   1 = output at the TIOCA pin at GRA compare match
                                   GRB compare no output */
itU0 TIOR1 = 2.
itU0 TIOR2 = 2.
itU1 TIOR3 = 2.
itU0 TIOR4 = 2


itU0 TIER0 = 1.                 /* Enable GRA interrupt to occur when it is
                                   requested by TSTA */
itU0 TIER1 = 1
itU0 TIER2 = 1
itU0 TIER3 = 1.
itU0 TIER4 = 0.                 /* Disable all interrupt requests by timer T4 */

STRCR = 11                      /* Select all bits in STRR as a count bit  NAM falling
                                   edge selected  Un-stop GRA enabled */

IPRA = 0.
IPRB = 0                        /* Set all interrupts priority level to zero (off) */


P10DDR = 255                    /* All Port 1 pins set for output */


itU0 GRA3 = 10000.              /* Set an initial delay of 1ms before getting into
                                   the main loop  */

P10DR = mZero                   /* Set Port 1 outputs to 0Vdc */


/*              GENERATE LOOK-UP TABLE                        */

for i=0 alpha=0; i<=(MAXSECTORSPW4) i= alpha+ .60 i++ )  alpha4 = alpha4+i
{
alpha4a4 = p*alpha*180
tp30[i = (int)(sin(2*PI*(double)alpha4a4/psmax)*32767.0+alpha4a4+
tp31[i = (int)(cos(2*PI*(double)alpha4a4/psmax)*32767.0+alpha4a4+
}
tp0[0]=tp30          /*because above loop gives wrong value for tp0[0]  reason unknown */
alpha=0              /* reset alpha after the above calculation loop */


itU0 TSTR = 0        /* start Timer 11 */

/*              MAIN LOOP                                     */

for(;;)
{                               /* Opening Bracket for the Do Forever loop */

while(itU1.TSTR != 32){         /* Wait in loop to get end angle for logic 1a
                                   to pump */
}
}

/* Sample sinWag and filtered freq */

sinWag = (((double)ADC[15]/1023.0)*.   /* Here values are assigned to sinWag and */
filtered = (int)                       /* Filtered but only these are sampled these
                                          two variables must be converted */

sequence = sequence                    /* Change sequence */.

counter = counter + 1                  /* Inc by 1 */
this = counter - (int)sinWag           /* Inst sampling period */

if(filtered   ((double)sinWag*100.0)   /* Phase lag loop inactive if sequence is 1 */
freq = (double)                        /* equal to the Nominal Frequency of 60 */
else                                   /* Variable frequency is equal to the previous */
freq = previous                        /* step it is equal to the previous frequency */

phase = (int)(freq*this)               /* angle displacement in this increment */

if(phase < (int)(alpha + phase_correction)) i
if(phase >= (int)(alpha + phase_correction)) i
if(phase >= (int)(alpha + phase_correction)) i
```

Appendix D: Source Code for SDEC0402.C

```
/*                              SDEC0402.C                              */


/* Program to perform the full SVM control of the inverter  */
/* Version 0.2 */
/* Author  Hamza Bapoo */
/* Date  12/10/95 */

/* Sampling Frequency  5 KHz */

/* Description:
  In this version the switching times are first generated and stored in a
  look-up table in the initialization stage. Also the use of replacement
  C keywords, like forall and floor, were eliminated in the main cycle loop
*/

/* Results
  The version performs successfully with the motor cycle *  q cor. id wi* -
  ner sampling time
*/

#include <math.h>

#pragma language = extended          /* Enable the use of extended keywords */
#include <mk82048.h>                 /* internal registers define  */
#include <mk82003.h>                 /* interrupt vector addresses define  */

/*  -  -  Define Constant Terms  -  -  */

#define pi 3.14159265
#define k1 0.5*pi
#define k2 0.5*sqrt(3.0)*pi
#define fClock 12.5E 6                       /* At a clock frequency of 10MHz */
#define maxPwmDutyRegs1 (long)1375
#define MAXSECTION 20

/*  -  -  -  -  -  Define Parameters              */

#define fSampling 1E 3                       /* At a Sampling frequency of 1kHz */
#define fControl fSampling/(float)fClock
#define fNominal 50
#define kd1 kd*fSampling/fClock
#define kd2 kd*fSampling/fClock

/*          -  -  Returned Variables          */

int segment, sector  counter.

int vpTemp  vkTemp, uZeroTemp, tjTemp  vkTemp  iOffsetTemp  temp.

int vp  vk  uZero, p  vk  offset, vpTemp  iOut.

  uskMag.                            /* uskMag changed to int */

float fObserver  time

float fAct, theta  alpha              /* t changed to fAct */


float alphaRad  samples.

int a [MAXSECTION/6+1]

int  vectorUpdate  alphaLow, alphaHigh  to  vk  iTemp  alphaMin

enum {false, true}.

int getVoltRef(temp  vectorAlpha1  alpha2  alpha3  alpha4

int  alpha1  alpha2  alpha3  alpha4  devPwmTemp

int  iStart1, iStart2  vector3  iStart4


/*****************************************************************
*                                                          MAIN
*****************************************************************/

void main (void)
{                    /* Opening bracket for main loop */

/*          Initializing Variables Initialization          */

sequence = 1
vector = 1.
count  = 1
uTemp = 63  vkTemp = 75  uZeroTemp = 7
uj = 63  vk = 75  uZero = 7
t_ = floor(fAct  vk = (float) fOffset = floor(

fClock = floor(fSampling fClock)

alphaMin = (u) MAXSECTION/6
deltaTemp = 1k alphaMin
alpha1 = 0
alpha2 = alpha1+deltaTemp
alpha3 = 2*alpha2
alpha4 = 3*alpha2
```

```
P6DR = x;                       /* output appropriate signal */

ITU_TSTR = 2;       /* start Timer T2 */
}

interrupt [ITU_GRA1] void Timer1InterruptRoutine (void)
{
ITU_TSTR = 0;     /* Stop timer T1 */

P6DR = 0;         /* Switch off all IGBTs */

iDelay = 1*1;     /* Delay to prevent overlapping of switching */
                  /* sequences */

ITU_TSR1 &= ~1;  /* acknowledge timer 1 interrupt compare interrupt */

ITU_GRA2 = i;     /* load new value of GRA2 */

P6DR = x;                       /* output appropriate signal */

ITU_TSTR = 4;         /* start Timer T3 */
}

interrupt [ITU_GRA2] void Timer2InterruptRoutine (void)
{
ITU_TSTR = 0;     /* Stop timer T2 */

P6DR = 0;         /* Switch off all IGBTs */

iDelay = 1*1;     /* Delay to prevent overlapping of switching */
                  /* sequences */

ITU_TSR2 &= ~1;  /* acknowledge timer 2 interrupt compare interrupt */

ITU_GRA2 = iDelay;              /* load new value of GRA2 */

P6DR = xZero;                   /* output appropriate signal */

ITU_TSTR = 0;         /* start Timer T2 */
}
```

```
ujTemp = Rsqrt(b1*sqSeg*initAlpha1.                    /* to and th ee terms of */
ukTemp = Rsqrt(b1*qrSeg*(cos(alpha) sneSerdsed0yRoot2*initAlpha1)); /* states */

switch (motor)
{               /* opening bracket for "switch(motor)"  sequence not equal to 3 */

case 1: ujTemp = 48; ukTemp = 33; break;          /* Select appropriate */
case 2: ujTemp = 31; ukTemp = 49; break;          /* switching */
case 3: ujTemp = 28; ukTemp = 11; break;                    /* vectors */
case 4: ujTemp = 14; ukTemp = 28; break;
case 5: ujTemp = 42; ukTemp = 14; break;
case 6: ujTemp = 35; ukTemp = 42; break;
|               /* closing bracket for "switch(motor)" - sequence not equal to 1 */

}               /* closing bracket for "else" - sequence not equal to 1 */

}               /* closing bracket for motor change? */

else
{               /*opening bracket for "else" of motor change? */
alpha = InvSin(Ans,60).                           /* compute alpha */
if (sequence = = 1)             /* forward sequence */
{               /* opening bracket for forward sequence */
ujTemp = Rsqrt(b1*sqSeg*(cos(alpha) sneSubdedByRoot2*sin(alpha))).* to and */
ukTemp = Rsqrt(b1*qrSeg*(sin(alpha)).                       /* th in terms of states */
}               /* closing bracket for forward sequence */

else                                              /* reverse sequence */
{               /* opening bracket for reverse sequence */
ujTemp = Rsqrt(b1*sqSeg*initAlpha1.                         /* to and th ee terms of */
ukTemp = Rsqrt(b1*qrSeg*(cos(alpha) sneSerdsed0yRoot2*sin(alpha)).* states */
}               /* closing bracket for reverse switching */

temp = ujTemp.                  /* swap switching */
ujTemp = ukTemp.                          /* motion */
ukTemp = temp.                                      /* order */

}               /* closing brackes for "else"  motion change? */

inMagnitude = InSqrt(ujTemp*ukTemp).                        /* compute sQrt(ujTemp) */

if (oldDirTemp = = 1)                                       /* change */
dirVectTemp = 58.                         /* the line */
else                                                        /* switching */
dirVectTemp = 7.                                    /* vector */

while (((4) TSTA ^= 2)>                   /* wait until */
{                                                   /* last vect sequence */
}                                                   /* of movement reached */

uj = ujTemp.                              /* updating the */
uk = ukTemp.                              /* parameter */
tj = tjTemp.                              /* values */
tk = tkTemp.                              /* for the */
dirVect = dirVectTemp.                    /* interrupt */
oldDir1 = dirNew1Temp                     /* algorithm */


}               /* closing bracket for the Do Forever loop */

}               /* closing bracket for main */


/* Interrupt Subroutines */

interrupt [17:0  (ARM3)] void timer1_Interrupt1Routine (void)
{
T1.. 3STA = 0       /* Stop timer 1 */

IFG  TSTA3 &=       ;  /* acknowledge timer 3 interrupt compare interrupt */

/* Note  "pause  <<= 10" to reset the 1 bit of the CCR1 is not necessary
  here since at this stage we do not have to send any other higher
  priority interrupt that can interrupt the ongoing interrupt
  service routine. The 300ns out interrupt will be a Non Maskable interrupt
  signal and so it can reach an interrupt at any time  whether the 1 bit of
  the CCR is set or not  */

TIC  CCR1 = dirVect1                      /* load new value of CCR1 */

P1OUT = dirVect                           /* output appropriate signal */

TC0  TSTA = 1;          /* start Timer 3A */
}


interrupt [17:0  (ARM4)] void timer1_Interrupt2Routine (void)
{
TC0  TSTA = 0       /* Stop timer TA */

P1OUT = 0;          /* switch off all lights */

timer = 1 + 1       /* index to prevent overlapping of switching */
            /* sequences */

IFG  TSTA3 &=       /* acknowledge timer 3 interrupt compare interrupt */

TIC  CCR1 = 1;      /* load new value of CCR1 */
```

```
ITU_TCR4 = 32,

ITU_TMOD = 2,                   /* Select CRA as an output compare register
                                   for output on the TIOCA pin at CRA compare match.
                                   CRA compare, no output. */
ITU_TIOR1 = 2,
ITU_TIOR2 = 2,
ITU_TIOR3 = 2,
ITU_TIOR4 = 2,
\

ITU_TIER0 = 1,                  /* Enable IMIA interrupt to occur when it is
                                   requested by IMFA */
ITU_TIER1 = 1,
ITU_TIER2 = 1,
ITU_TIER3 = 1,
ITU_TIER4 = 0,                  /* Disable all interrupt requests by timer T4 */

SYSCR1 = 11;                    /* Select SP bit as CCR as a user bit. NMI falling
                                   edge selected. On chip RAM enabled */
IPRA = 0,
IPRB = 0,                       /* Set all interrupts priority level to zero (low) */

P1DDR = 255,                    /* All Port 1 pins set for output */

ITU_GRA3 = 10000,               /* Set an initial delay of time before getting into
                                   the main loop */

P1DR = 0xFF,                    /* Set Port 1 initially to all on */

ITU_TSTR = 8       /* start timer T3 */


/*  . . . . . MAIN LOOP                 */

for(;;)
{                               /* Opening Bracket for the C0 Forever loop */

while(ITU_TSR != 239)           /* Continue to stay in wait for first zero
                                   sequence */
{
}
}

/* Sample voltag and Hall data */

voltag = 0.82*1015523.          /* More values are assigned to voltag and */
Hall = 50.                      /* Hall but once they are sampled these
                                   over side lines must be removed */

sequence = sequence.            /* Change sequence */

counter = counter + 1;          /* Go to */
t = counter * tSampling.        /* next sampling period */

w_voltage = (Hall_Value - 512); /* Phase lock loop. Inverter frequency is */
time = thetaval.                /* equal to the Normal frequency of the  */
else                            /* Reverse Fedquency in opposite due direction */
time = -time.                   /* else it is equal to the Reverse Frequency */

phase = (2*pi*fnom*t).          /* angle displacement w.r.t the horizontal */

if(t%tSampling==0)  = sector;   /* Sector change */
{                   /* opening bracket for sector change*  */

/*              Change of sector                */

if (phase > =3*pi)                                      /* Another cycle completed ! */
{                   /* opening bracket for "if (phase > =2*pi);" */

phase = phase - 2*pi.                                   /* Another cycle completed */
counter = Rem(phase/(voltage*tSampling)).  /* Reset counter */
}                   /* closing bracket for "if (phase > =2*pi)" */


alpha = fmod(phase,60).  /* Angle w.r.t first sector of the present sector */
s  = (t - (phase.60)).                  /* find new sector */
sequence = sequence.   /* remember sequence does not change when there is
                          a sector change */

if (sequence == 0)
{                   /* opening bracket for "if (sequence == 0)  Change of sector loop */
ujTemp = (sin(2*pi*voltag*(cos(alpha-(sin(alpha)*cos(alpha)/2*sin(alpha)))  /* ta and */
ukTemp = (sin(2*pi*voltag*sin(alpha)                    /* tb of terms of states */

switch(sector)
{                   /* opening bracket for "switch(sector)  sequence 0 */
case 0: ujTemp = 39; ukTemp = 40; break;        /* select appropriate */
case 2: ujTemp = 49; ukTemp = 31; break;        /* switching */
case 3: ujTemp = 31; ukTemp = 24; break;        /* sectors */
case 4: ujTemp = 20; ukTemp = 14; break;
case 5: ujTemp = 14; ukTemp = 42; break;
case 6: ujTemp = 27; ukTemp = 35; break;
}                   /* closing bracket for  "switch(sector)  sequence 0 */

}                   /* closing bracket for "if (sequence == 0)  Change of sector loop */


else                                            /* sequence == 1 */
{                   /* opening bracket for "else"  sequence not equal to 0 */
```

```
P1DIR = sbaTxmsk; /* Subsidiary Data between sh and ap */

ITU_GRA2 = tUint2;                  /* load new value of GRA2 */

P1OUT = mTxm;                       /* output appropriate signal */

ITU_TSTR = 0;           /* start Timer T5 */
}


void phaseLockLoopcrol()
{

vBus1 = vRes2;
vVar1 = vVar2
pVar2 = vDes;
vVar2 = vVar;
/* if = vVar vRes;
if (vBit = 0)
vDiff = vBit;


vResDiff = vRes2 vRes1;
if (vDiffDiff >=0) vResDirection = posDirect;
if (vResDiff < 0) vResDirection = negDirect;
vVar1.Y = vVar2.vVar1;
if (vVarDiff >=0) vVarDirection = posDirect;
if (vVarDiff <=0) vVarDirection = negDirect;

/* tRes = tMint; tRes = tMint; /* Phase lock loop inverter frequency */
1                   /*open bracket ** if tRes out of limit*/
bSwitchInhibit = true;
tRes = tUpcount;
if (tUnderActual <= tMint || tResActual >= tMaxInf (tVar1 = vVarMint) (tVar1 =tVarMax))
{}
else
{ /* open bracket for inverter tRes*/
invInhibit =true;
while tRes >=tResCount
|
}
vQuickInhibit = false
} /* close bracket for inverter tUpcr*/
} /* close bracket for tRes out of limit */

else
{ /* open bracket for tRes within limit */
tRes = tRes;
if (vDiff >= vDiffMax)
{ /* open bracket for vDiff >=vDiffMax   out of phase*/
vDoubleInhibit = true.
if tRes<= tMint
Rev =tMax;
else
Rev=tRes;
} /* closing bracket for vDiff >=vDiffMax   out of phase */

else
{ /* open bracket for vDiff in */
/* inPhaseDuration = =vInvDuty; out
vQuickInhibit =false;                   /* in phase */
else
{ /* open brackes for opposite direction */
bSwitchInhibit = true
if tRes<  =tMint
Rev =tMax;
else
Rev= tRes;
} /* closing bracket for opposite direction */
} /* closing bracket for vDiff in*/
} /* closing bracket for tRes within limit */

/* end of phase lock loop */
}
```

```
}               /* closing bracket for "else"  vector change */


if cyTemp<. (Uint) cyTemp=Uint.
if (Int)Temp>. (Uint)(Int)Temp=Uint.     /* to prevent negative value results due to starting a timer for 0 seconds */

(Uint)Temp = (Uint)(cyTemp (Int)Temp) + 1                    /* compute (Uint)Temp */

if (bZeroTemp == 1)                                          /* change */
  uZeroTemp = 1.
else                                                         /* particular */
  uZeroTemp = 0.                                             /* uZero */

uy = uyTemp.
uyTemp = uyZeroTemp.
Uy = uyTemp.

while (uTU  TBIR >= 2321)                                    /* wait until */
{                                                            /* last time expired */
}                                                            /* or overflow is reached, we want T_interrupt to be implemented */

                                                             /* updating the */
ux = uxTemp.                                                 /* parameters */
uyTemp = uyTemp1Temp
Ux = UxTemp.

uxTemp = uxTempTemp          /* for the */
uPos = uZeroTemp                                             /* interrupt */
(Uint)R = (Uint)Temp                                         /* alignment */


}               /* closing bracket for the timer1 temp */

}               /* closing bracket for main */


interrupt [TU  (Uint)] void Timer1InterruptRoutine (void)

{
TU  TBIR = 0     /* Stop timer TU */

TU  TFRQ &=    1;  /* acknowledge Timer 1 interrupt = compare interrupt */


/* Note "uZero  Error()" the result is a bit of the CCR, so not necessarily
   have done at this stage and we do not have an error until higher
   priority interrupt that can interrupt this one by nesting interrupt
   service routine. The 100% error interrupt will be a Non Maskable interrupt
   (NMI) and so it can never be interrupted at one time. Whether the I bit of
   the CCR is set or not. */

TU  TBAR = (Uint)R                   /* load new value of (Uint) */

TU  TSTR = 1          /* start timer TU */
}


interrupt [TU  (Uint)] void Timer2InterruptRoutine (void)

{
TU  TBIR = 0     /* stop timer T2 */

TU  TFRQ &=    1;  /* acknowledge Timer 2 interrupt = compare interrupt */

PTUOR = uyTemp1;  /* Temporary store for Uint */

TU  TBAR = 0     /* load new value of (Uint) */

PTUOR = uy                           /* output appropriate signal */

TU  TSTR = 1          /* start Timer T2 */
}


interrupt [TU  (Uint)] void Timer3InterruptRoutine (void)

{
TU  TBIR = 0     /* Stop timer T3 */

TU  TFRQ &=    1;  /* acknowledge Timer 3 interrupt = compare interrupt */
```

```
else{ = qbate EQUIVALENT, sectorUpdate = 1,
if (elphel < =guesses)
counter=0,
else
counter=1,
}
}


 /* xtemp = Angle of 1 first vector of the present sector * 100 000*/

if (sector != sectorUpdate)             /* Sector change ? */
{               /* opening bracket for sector change? */

/*              Change of sector                */

xtemp = sectorUpdate,


/*      return xx and xb from the look up table     */

<= (elphel1/elphel2)*
ixx=(int)X,
iy=(int)Y;

sequence = sequence     /* remember sequence does not change when there is
                        a sector change */

if (sequence == 1)
{               /* opening bracket for "if sequence == 1:  change of sector loop */

xyTemp = (int)*xxMagn + 0.5,
xxTemp = (int)*yyMagn + 0.5,

switch (sector)
{               /* opening bracket for "switch(sector)  sequence 1 */
case 1  {xyTemp=37, xxTemp=43, xyyTransTemp=3, xyxTransTemp=35, xxyTransTemp=40 break;}          /* offset appropriate */
case 2  {xyTemp=43, xxTemp=21, xyxTransTemp=48, xyyTransTemp=12, xxxTransTemp=5 break;}         /* switching */
case 3  {xyTemp=21, xxTemp=26, xxyTransTemp=6, xxxTransTemp=30, xxyTransTemp=24 break;}            /* vectors */
case 4  {xyTemp=29, xxTemp=14, xyyTransTemp=24, xyxTransTemp=19, xxyTransTemp=8 break;}
case 5  {xyTemp=14, xxTemp=42, xxyTransTemp=5, xxxTransTemp=10, xxxTransTemp=44 break;}
case 6  {xyTemp=42, xxTemp=29, xxyTransTemp=48, xxxTransTemp=24, xxxTransTemp=3 break;}
}               * closing bracket for "switch(sector)  sequence 1 */

}               /* closing brace on for "if sequence == 1:  change of sector loop */


else                                    /* sequence =  1 */
{               /* opening bracket for "else"  sequence not equal to 1 */

xyTemp = (int)*xxMagn + 0.5,
xxTemp = (int)*yyMagn + 0.5,

switch (sector)
{               /* opening bracket for "switch(sector)"  sequence not equal to 1 */

case 1  {xyTemp=40, xxTemp=34, xxyTransTemp=46, xxxTransTemp=23, xxxTransTemp=3 break;}          /* Select appropriate */
case 2  {xyTemp=31, xxTemp=45, xxyTransTemp=6, xxxTransTemp=13, xxxTransTemp=48 break;}          /* switching */
case 3  {xyTemp=28, xxTemp=21, xxyTransTemp=24, xxxTransTemp=29, xxxTransTemp=5 break;}                  /* vectors */
case 4  {xyTemp=16, xxTemp=28, xxyTransTemp=8, xxxTransTemp=12, xxxTransTemp=26 break;}
case 5  {xyTemp=42, xxTemp=18, xxyTransTemp=40, xxxTransTemp=10, xxxTransTemp=8 break;}
case 6  {xyTemp=25, xxTemp=42, xxyTransTemp=3, xxxTransTemp=34, xxxTransTemp=40 break;}
}               /* closing bracket for "switch(sector)"  sequence are equal to 1 */

}               * closing bracket for "else"  sequence not equal to 1 */

}               * closing bracket for sector change* */

else
{               *opening brackets for "else" of sector change? */


/*      output xx and xb from the look up table         */

<= (int)xxt_sign*xb+1;
iy=(int)X;
ix=(int)Y;

if (sequence == 1)                      /* forward sequence */
{               * opening bracket for forward sequence */
xyTemp = (int)*yyMagn + 0.5,
xxTemp = (int)*xxMagn + 0.5,

}               * closing bracket for forward sequence */

else                                    * reverse sequence */
{               * opening bracket for reverse switching */
xyTemp = (int)*xxMagn + 0.5,
xxTemp = (int)*yyMagn + 0.5,

}               * closing bracket for reverse switching */

temp = xyTemp                   /* swap switching */
xyTemp = xxTemp                         /* switch */
xxTemp = temp                                   * vectors */
temp = xyTransTemp
xyyTransTemp = xxyTransTemp
xxyTransTemp = temp
```

```
                                        edge selected  On-chip BAUD enabled */
TIRA = 0;
IPRB = 0                        /* Set all interrupts priority level to zero (lowest) */

P1DDR = 255;                    /* All Port 1 pins set for output */

ltoa_CRAS = 1 * 1000;           /* Give an initial delay at 1ms before entering into
                                   the main loop  */

P1DR = 0x5a;                    /* Set Port 1 initially as 0x5a */


/*              GENERATE LOOK-UP TABLE          */

for a=0, alpha=0, a<= (MAXSELECTORDIV 1), alpha+= 90, a++, alpha = alpha&0x3f)
{
alphaRad = pi*alpha/180;
sinAlphaRad = sin(alphaRad);
t1[a] = (int)(sinAlphaRad*sinDivByRad2*sinAlphaRad);
t2[a] = (int)(cosAlphaRad);
}

/* replacing the following, because above loop gives wrong values for
t1[90], t1[270], and t1[360] - the last two when using a MAXSELECTORDIV
of 90  Sample calculations given there values only apply for a MAXSELECTORDIV of
90 only  */

t1[0] = 2534;                   /*34560 */        /*4096 */
t1[1000] = 1347;  /*8423.4*/    /*8096.*/
t1[2000] = 1347;  /*8423.4*/    /*8096.*/

alpha=0;        /* reset alpha after the above initialization loop */


TIR |= 0x10;    /* start Timer T2 */

/*              MAIN LOOP               */

for(;;)
{                               /* Opening Bracket for the for(;;) loop */

while (ITL & 0x10 != 0x10)              /* Condition to prevent wait for two zero
                                           sequence  i.e wait for completion of T1interrupt routine */
{
}

sequence = sequence                     /* Change sequence */

counter = counter + 1                   /* Go to */
sRet = counter * 15;            /* next sample period */


/* Sample voltage, IRef, alpha and offset here */

/*voltage = 0x0FF0;*/           /* these values are assigned to voltage, iRef, offset and alpha */
IRef = 0x0; Alpha = 0x0                 /* but more they are required these */
alpha=230; offset=230;                  /* each time must be */
offset2=235  alpha2=230;                /* removed */

p[sequence] = port  & 0x0
voltage = (offset<<8)  alpha           /*  assigning voltage */

/*voltage = 0x0700;  /*0x0700 */     /* forcing voltage to a      val  for testing purposes only */

generate_sine();

phase = /*  alpha*0x01          /* angle displacement w.r.t the horizontal */



/*for (a = 0 *                  /*************;
/*phase = 0x0 as */                     /*************;

/* an alternative way to finding alpha1 and secondalpha */
/*
secondQuadrant = (phase>=FIRSTQUAD)
alpha1 = phase-secondQuadrant*FIRSTQUAD
secondQuadrant=secondQuadrant*2;
if(phase >= secondQuadrant)
{
if(alpha1<= quadrant)
alpha1=0
else
alpha1=1
}
*/

if(phase <= FIRSTQUAD)
{
if(phase <= quadrant) {alpha1 = phase; secondQuadrant = 0;}
if(phase <= SECONDQUAD && phase >= quadrant) {alpha1 = phase-(quadrant); secondQuadrant = 1;}
if(phase <= THIRDQUAD && phase >= SECONDQUAD) {alpha1 = phase-(SECONDQUAD); secondQuadrant = 2;}
}
else
{
if(phase <= FIRSTQUAD*3 && phase >= THIRDQUAD) {alpha1 = phase-(THIRDQUAD); secondQuadrant = 3;}
if(phase <= SECONDQUAD*4 && phase >= FIRSTQUAD*3) {alpha1 = phase-(FIRSTQUAD*3); secondQuadrant = 0;}
if(phase <= THIRDQUAD*4 && phase >= SECONDQUAD*4) {alpha1 = phase-(SECONDQUAD*4); secondQuadrant = 1;}
if(phase >= THIRDQUAD*4) {alpha1 = phase-(THIRDQUAD*4); secondQuadrant = 2;}
```

```
float ba, kgt, kb, kbt, sampDurBySec=2, secAlpha=4,
float interSectionDur=240,
int i0, m,
unsigned long pixie, alpha3, alphaMin1, tAcc, gamma,
long clockTemp, c,
float val, c1,

/* global test loop variables */
int clockSampleAddr, breakAddr, clockDirection,
int xPos, xPos1, xPos2, xDec, yDec1, yDec2, xDecCross, posDirect, negDirect,
int lineAccept, xDecMin, xDecMax, tMin, tMax,
int xDiff, xPos0Diff, xDec0Diff, xDecDirection, xDiffAbs,

/*                                                                          */
/*                                MAIN                                      */
/*                                                                          */

void phaseLockLoop (void)

{  start wait main (void)
{                    /* spacing brackets for main loop */

/*              Scanning Variables Initialization                           */

int a=0;  /* use unlocked just to be able to see the stack when debugging*/

xSampling = 1.0E-6,
yA = (long)xSampling*1000000,
sequence = 1
sector = 1
center = 1,
xpTemp = 45, xbTemp = 35, xZeroTemp = 7,
yA = 40, yB = 35, xZero = 7,
xpZeroTemp = 40, xpTraceTemp = 33, xbZeroTemp = 3
xpTrans = 48 xpTrans = 33, xbZTrans = 3,

xTotal = (long)(xSampling/(4*xClock)),

ky = TimInt, kk = (int)kA, xDiff = xTotal,

ka = 9 (a*b)
kb = 5*sqrt(3)(a*a)
ka1 = ka*(xSampling/xClock,
kb1 = kb*(xSampling/xClock,
xmDivByTwo3 = (int)xT);

xSamp = Accel(xSampAngle/xClock),

alphaAbs = 60/xmisSectorDiv,
alphaMin = alphaMin*1000000,

gamma = 1/60*(Pi*xmin*(xSampling)2,

xTestAbs = .01, xTestDec=252,      /*  +/- 10% Offset Filter */
int = (to pulse xmxMinimumAlphaMin-xAlphaMin)
m = int,
c = (int)(Dir m)*xAlphaMin*65536

xMin= 40;

xSwitchInhibit = true
xSpinDir = false,
xPos2= xDec
xDec2 = xSmall
xPosCross = 0.01*xDecmax,
posDirect = 1 negDirect = 1,
tAcc=200, tMax=510
xDiffMax = 0.01*xDecmax,

/*            I/O648 Registers Initialization                               */


xPort  PortA = 92                  /* Select 8255 Control which is Missing "Mode"
                                      Select external clock (+5Volts) */

xPort  PortB = 92
xPort  PortC = 92
xPort  PortD = 92
xPort  PortE = 92


xPort  PortA = 2                   /* Select Clock as an output control register
                                      Turn output of the Time Base on with sampling rate/A
                                      with a square wave output */
xPort  PortB = 2,
xPort  PortC = 2
xPort  PortD = 2
xPort  PortE = 2


xPort  PortA = 1                   /* Enable 8254 control pulse to turn off Motor
                                      as controlled by the A */

xPort  PortB = 1
xPort  PortC = 1
xPort  PortD = 1
xPort  PortE = 0                   /* Disable all other port registers by Port 3 */

xPort  PortA = 14                  /* Select Phase Lock Loop and Main Clock Signal
```

Appendix G: Source Code for SDEC0405.C

```
/*                              SDEC0405                                */

/* Program to perform the full DVM control of the inverter (DE)  */

/* Sampling frequency  8.26 KHz */

/* Version 0.0 */
/* Author  Thomas Ropes */
/* Date  03/07/95 */

/* Description
   Following the goal mainly achieved in version 0.4, this program version
   contains an higher frequency  8.26 KHz. In addition this version contains
   timestamp checks to ensure a full loop code (DVM  control and slow  and more
   properly computed  method of being exactly sampled values as in the
   previous versions
*/

/* Results
   Program runs correctly  except at extreme values as shown within a section (
   where tdiff is longer!  At these points Ij and mjk1sink are not yet
   updated when timer interrupt F2 occurs  However they do not detrimental to
   the overall performance of the software, since Ij and mjk1sink were both
   observed to be updated by the time the next timer interrupt (F3) has
   occured  i.e the whole calculations loop is still being completed within a
   sampling period  The requirement does however indicate the present sampling
   frequency  8.26 KHz, as a limiting case  When run at 7163 Hz the software
   failed to produce the correct PWM output pattern
*/



#include < math.h >

#pragma language = extended        /* Enable the use of extended keywords */
#include < iom3048.h >             /* interrupt vector addresses define  */
#include < ina90903.h >            /* interrupt vector addresses define  */

/*          Define Constant Terms                */

#define pi 3.141592654
/* #define tdiv  0.242 E-6 */                    /* At a clock frequency of 16MHz */
#define MAXJ  10000/2*3.240

/*          Define Parameters                    */

/*#define tSampling 1E-3*/                      /* At a Sampling Frequency of 1kHz */
#define tNominal 850
#define tBig 400
#define tSmall 0
#define tMinimal 230
#define mMaxMin 0.5
#define mMaxMax 0.8056914

/*          Runway Variables                     */

float mjk1sink = 0.5E-6

int sequence_counter,

int ujTemp, ukTemp, uJcalTemp, temp, mjkTempTemp, mkTempTemp, mkcTempTemp

unsigned int tjTemp, tkTemp, tDiffTemp

int mj, mk, uJcal, ujTemp, mkTemp, mkcTemp, tj, tk, tDiff,

long mjkSq                         /* mjkSq changed to int */

int tLast, time

float mjk1                         /* changed to int */

float alphaTan

float tSampling

int * portTwoDirection;

unsigned long tp, tq

pragma (interrupt_fast)

int getTimerValue ()

int * p = (int *) 0x0;

int ASMK(int function);

float mjk1Sq;
```

```
P1DR = 0;                       /* output appropriate signal */

ITU_T3IR = 2;           /* start Timer T1 */
}

interrupt [ITU_GRA1] void timerT1interruptRoutine (void)
{
ITU_T3IR = 0;    /* Stop timer T1 */

ITU_T3SR &= ~1; /* acknowledge timer 1 interrupt compare interrupt */

P1DR = 0;       /* Switch off all IGBTs */

ITU_GRA2 = 1A;  /* load new value of GRA2 */

P1DR = 1A;                      /* output appropriate signal */

ITU_T3TR = 4;           /* start Timer T2 */
}

interrupt [ITU_GRA2] void timerT2interruptRoutine (void)
{
ITU_T3TR = 0;    /* Stop timer T2 */

ITU_T3SR &= ~1; /* acknowledge timer 2 interrupt compare interrupt */

P1DR = 0;       /* Switch off all IGBTs */

ITU_GRA3 = 1DA1A1;              /* load new value of GRA3 */

P1DR = 0Fxx;                    /* output appropriate signal */

ITU_T3TR = 8;           /* start Timer T3 */
}
```

```
case 6: [u]temp = 4; /* change to step 4 */ break;
case 8: [u]temp = 5; /* change to step 5 */ break;
}                    /* closing bracket for "switch(sector)" sequence not equal to 1 */

}                    /* closing bracket for "else" sequence not equal to 5 */

}                    /* closing bracket for sector change? */

else
{                    /* opening bracket for "else" of sector change? */


/*    - point to end up from the look up table -    */

(= [alpha beta][delta]);
w = [0][0];
w = [1][1];


if (sequence == 0)                  /* forward sequence */
{                    /* opening bracket for forward sequence */
u[temp = u[w*sub[up] > > 10;
v[temp = v[w*sub[up] > > 10;

}                    /* closing bracket for forward sequence */

else                                /* reverse sequence */
{                    /* opening bracket for reverse switching */
u[temp = u[w*sub[up] > > 10;
v[temp = v[w*sub[up] > > 10;

}                    /* closing bracket for reverse switching */

temp = w[temp;                      /* swap switching */
w[temp = w[temp;                    /* reverse */
w[temp = temp;                                  /* order */

}                    /* closing brackets for "else" sector change? */

if (u[temp == v[v[temp == v[w
if (v[temp == v[v[temp == v[w       /* to prevent unpredictable results due to storing a value out of range */

v[w[v[temp = (u[temp u[temp v[temp > > 4;        /* compute W[AvTemp */

if (u[temp[temp == 0)                           /* change */
w[temp[temp = 65;                   /* the case */
else                                            /* wantarg */
w[temp[temp = 0;                                /* pause */

while (PIR_TSIR != 232)             /* wait until */
{                                               /* last rate sequence */
}                                               /* all waveforms reached */

w = w[temp;                         /* updating the */
vk = v[temp;                        /* parameter */
u = u[temp;                         /* values */
vk = v[temp;                        /* for the */
v[temp = w[temp[temp;               /* sequence */
v[w[v] = v[w[temp;                  /* algorithm */

}                    /* closing bracket for the Do Forever loop */

}                    /* closing brackets for main */


/* Interrupt Subroutine */

interrupt [TIM_OVR] void timer1_ISR(void);
{
PIR_TSIR = 0;    /* Step base 12 */
                                    /* if OVR 224 Need to check below */

[TIM_TSIR] &= 1;  /* acknowledge timer 1 interrupt resume interrupt */
                                    /* Must give the [TIM_TSIR] = 0 as a simpler way */


/* Note "wait = set (0)" to show the time of the ISR is not necessary
   here since a  this stage we do not have to read any other higher
   priority interrupts that can interrupt this on going interrupt
   service routine. The [TIM_OVR] interrupt will be a Non-Maskable Interrupt
   (NMI) and so it can cause an interrupt at any time, whether the CPU is
   the ISR is on or not. */

[TIM_OVR] = [TIM_OVR                * read new value of [TIM_OVR */

PIR_TSIR = w[temp                   * output appropriate signal */

[TIM_TSIR = 1                       * reset timer TIR */
}

interrupt [PIR_TSIR] void timer1_output_compare_ISR(void)
{
[TIM_TSIR = 0;    * stop timer TIR */

[TIM_TSIR] &=     * acknowledge timer 1 interrupt resume interrupt */

PIR_TSIR = 0;     * switch off all TSIRs */

[TIM_TSIR = 1;    * read new value of TSIR */
```

```
ushtag = 92471.   /* these values are assigned to ushtag and */
Rtoserve = 900.                         /* rtoserve but once they are sampled there
                                           new code lines must be measured */

sequence = sequence.                    /* change sequence */

counter = counter + 1.                  /* Go to */
tAst = counter * 10,            /* next sampling period */

if (RReserve < 4901) | (RReserve > 5101)  /* Phase lock loop: Inverse frequency in */
Rno = RNominal;         /* equal to the Nominal Frequency of the */
else                                    /* Returns Frequency is outside the window. */
Rer = RReserve;         /* else it is equal to the Reserve Frequency */

phase = 136*Rer*Rtst;           /* angle displacement w.r.t the landmarks */

/*
sectorUpdate = (phase/60000000.
alpha1 = phase sectorUpdate*60000000.
sectorUpdate=sectorUpdate+1;
if (phase >=360000000)
{
if (alpha1<=gamma)
counter=0
else
counter=1.
}
*/

if (phase<=10000000)
{
if (phase < 60000000) {alpha1 = phase. sectorUpdate = 1.}
if (phase<= 120000000) && (phase >= 60000000) {alpha1 = phase-60000000. sectorUpdate = 2.}
if (phase<=.180000000) && (phase >= 120000000) {alpha1 = phase-120000000. sectorUpdate = 3.}
}
else
{
if (phase<=240000000) && (phase >= 180000000) {alpha1 = phase-180000000. sectorUpdate = 4.}
if (phase<= 300000000) && (phase >= 240000000) {alpha1 = phase-240000000. sectorUpdate = 5.}
if (phase<=.360000000) && (phase >= 300000000) {alpha1 = phase-300000000. sectorUpdate = 6.}
if (phase >= 360000000)
alpha1 = phase-360000000. sectorUpdate = 1
if (alpha1 <= gamma)
counter=0.
else
counter = 1
}
}

 /* alpha1 = Angle w.r.t last vector of the present sector * 100 000*/

if (sector != sectorUpdate)             /* Sector change ? */
{               /* opening bracket for sector change? */

/*              Change of sector                        */

sector = sectorUpdate.

/*      select Ts and Tr from the look up table         */

x= (alpha1-alpha1%1e3)
Ts=Ts[x]
Tr=Tr[x]

sequence = test_seq.    /* sometimes sequence does not change when there is
                        a sector change */

if (sequence == 1)
{               /* opening bracket for "if (sequence == 1)  change      in loop */

tyTemp = (Ts*usWtg) >> 16.
txTemp = (Tr*usWtg) >> 16

switch (sector)
{               /* opening bracket for "switch(sector)  sequence 1 */
case 1: {uyTemp = 35; uxTemp = 49; break;}              /* select appropriate */
case 2: {uyTemp = 49; uxTemp = 21; break;}              /* switching */
case 3: {uyTemp = 21; uxTemp = 78; break;}                      /* vectors */
case 4: {uyTemp = 28; uxTemp = 14; break;}
case 5: {uyTemp = 14; uxTemp = 42; break;}
case 6: {uyTemp = 42; uxTemp = 35; break;}
}               /* closing bracket for "switch(sector)  sequence 1 */

}               /* closing bracket for "if (sequence == 1)  change of sector loop */


else                                            /* sequence != 1 */
{               /* opening bracket for "else"  sequence not equal to 1 */

tyTemp = (Ts*usWtg) >> 16
txTemp = (Tr*usWtg) >> 16

switch (sector)
{               /* opening bracket for "switch(sector)  sequence not equal to 1 */

case 1: {uyTemp = 49; uxTemp = 35; break;}              /* select appropriate */
case 2: {uyTemp = 21; uxTemp = 49; break;}              /* switching */
case 3: {uyTemp = 78; uxTemp = 21; break;}                      /* vectors */
case 4: {uyTemp = 14; uxTemp = 28; break;}
```

```
cb = NewtSampleq*100000;
sequence = 1;
tester = 1;
counter = 1;
qf4orb = 45, wf1eeq = 35, wZeo1emp = 7,
xy = 40, yb = 35, wZero = 7,

Normal = Reset(&SampleqH*4[Check],

N = Shift(N), Nb = Shift(N, 10Mqtf = Normal.

ba = Dipa*pa,
kb = 8*sqrt(3)Sqrt*pa,
ba3 = ba*3SampleqH[Check,
bb1 = kb*3SampleqH[Check,
aaaDivDipNeed3 = Vperc(420;  ,

tSamp = Reset(&SampleqH[Check),

alphaMid = 90/maxSpeedDiv,
alphaMax1 = alphaMin*100000,

gamma = (360*Rtnormal*1SampLeqH2.

tMin= 50.

/*          HC548 Registers Initialization          */

HTU  T2IO = 32.                 /* Select GRA compare match at Cleaning Clocks
                                   Select Internal Clock (16MHz) */
HTU  T2R1 = 32.
HTU  T2R2 = 32.
HTU  T2R3 = 32;
HTU  T2R4 = 32.

HTU  T2OR0 = 2.                 /* Select GRA as an output compare register
                                   1 is output at the TIOCA pin at GRA compare match
                                   GR0 compare, as output */
HTU  T2OR1 = 2.
HTU  T2OR2 = 2.
HTU  T2OR3 = 2.
HTU  T2OR4 = 2

HTU  T2ER0 = 2,                 /* Enable ITA3 interrupt to occur when it is
                                   requested by IMFA */
HTU  T2ER1 = 1.
HTU  T2ER2 = 1.
HTU  T2ER3 = 1.
HTU  T2ER4 = 0.                 /* Disable all interrupt requests by table T4 */

SYSCR = 11.                     /* Select bit bit in CCR as a user bit  NMI falling
                                   edge selected  On chip RAM enabled */

IPRA = 0.
IPRB = 0.                       /* Set all interrupts priority level to zero (low) */

P1DDR = 255.                    /* All Port 1 pins set for output */

HTU  GRA2 = 16000.              /* Set up initial delay of 1ms before going into
                                   the main loop  */

P1DR = mSpeed.                  /* Set Port 1 initially as zeros */

/*              GENERATE LOOK UP TABLE              */

for x=0, alpha=0, >=  =(MAXSELECTION/2) alpha+ CO. (= +, alpha = alphaMin)
{
alphaRad = pi*alpha/180,
sineAlphaVal = sin(alphaRad)
NUM1 = (gamma*sin(alphaRad+asin(Rtf*sin(alphaRad)*sineAlphaRad)/2
tbl[x] = (long int)*sineAlphaRad.
}

/* redefining the following because above has given wrong values for
tbl[90] tbl[2050] and tbl[2051]  the last one when array = MAXSELECTION/2
at 90 degrees centigrade. Note these values only apply for a MAXSELECTION of
60 only  */

tbl[0] = 4976                   /*4459 */          /*4526 */
tbl[2050] = 5088  /*5422 */     /*5685 */
tbl[2051] = 5088  /*5422 */     /*5685 */

mSpeed=0          /* reset alpha after the above calculation is p */


HTU  T1STU = 0          /* start Timer 1 */

/*              MAIN LOOP                   */

for(;;)
{                               /* Optional Register for the 0/- Increment loop */

while(HTU  T1STR != 2)                  /* Conditions to prevent speed for first zero
                                           sequence */
{
}

/* various settings as all sequences have */
```

Appendix F: Source Code for SDEC0404.C

```
/*                                  SDEC0404                              */

/* Program to perform the full SVM control of the inverter */

/* Version 0.4 */
/* Author: Hassan Bassa */
/* Date: 08/11/95 */

/* Description:
   This version is similar to version 0.3, but with several changes to
   eliminate the few floating operations left, especially those operations that
   involve phase and alpha1. In addition the number of divisions per sector,
   MAXSECTORDIV, was increased from 60 to 240, hence giving a higher accuracy
   of the switching timer values
*/

/* Results
   The program ran successfully and showed a significant decrease in
   computational time / SE microseconds /, hence giving possibility for
   an even higher operating sampling frequency
*/


#include <math.h>

#pragma language = extended      /* Enable the use of extended keywords */
#include <intr3046.h>            /* internal register define  */
#include <intr3003.h>            /* interrupt vector addresses define */

/*      - - - Define Constant Terms          - - - */

#define pi 3.141592654
#define tClock 32.0E-9                   /* At a clock frequency of 16MHz */
#define MAXSECTORDIV 240

/*      - - - Define Parameters - - -        */

/*#define tSampling 1E-3*/                /* At a Sampling frequency of 1kHz */
#define tNominal 800


/*      - - - - Naming Variables     - - -  */

int sector, center, register,

int u[To...3  uTemp, vZeroTemp, tOfallTemp, temp,

unsigned int tTemp, ti , zero,

int op  uk, uZero, a, tk, tOfall,

unsigned long calMag                     /* calMag changed to int */

int tAssume  flag ,

float alpha.        /* t changed to tAlt */

flags symmetry

flags tSampling

int a  secondUpdate  tStamp,

unsigned long ts, tk.

enum {false, true}

int getOutOfLoop.

int k , timing  tMax.

int tMAXSECTORDIV=240.

float alphaMin

float ka  kal  tb  kb?. cosDivByTen3  sineAlphaRad
float maxSectorDiv=240.
int n;
unsigned long phase, alpha1, alphaMin, tAlt, phase

/*****************************************************************************
*                                                                    MAIN
*****************************************************************************/

[ task ma  main (void)
{                    /* opening brac tet for main loop */

  *          Naming Variables Initialization          *.

int pwm  /* pwm included just to be able to use the wa  h when debugging*.

tSampling = 2E-6
```

```
P1DR = x0;                      /* output appropriate signal */

ITU_TSTR = 4;       /* start Timer T2 */
}

interrupt [ITU_ISR2] void timer2(void)
{
ITU_TSTR = 0;    /* Stop timer T2 */

ITU_ISR2 &= ~1;  /* acknowledge timer A interrupt causing interrupt */

P1DR = 0;        /* Switch off all LEDs */

ITU_GRA3 = 1000;                /* load new value of GRA3 */

P1DR = x2000;                   /* output appropriate signal */

ITU_TSTR = 8;       /* start Timer T3 */
}
```

```
pinmap = 1280' Maximal 12mapping*1000000UL2.


setmicro = 1.
unsigned = 1;
complex = 1,
ii = 48, ok = 22, nZero = 7.

q = (double) ik - (float) (Uint)i = (Uint)ik                    \


/* values for the phase lock loop */
qSwitchAbsNet = new.
P1DDR = 1;                      /* Set qSwitch(pin4) : equal to "1"  on pin 8 of port 5 */
MaxAbsNet = false.
qNet2= xOne;
Vec2 = xSmall,
qNet^reset = 0.01*qRound(*512.
consumer = 1, negOutput = 1.
qQuiNet2 = 0.01*qRound,



/*  ..... H8340 Registers Initialization .....  */


ITU_TCRD = 32.                  /* Select GRA compare match as Clearing Source
                                   Select Internal Clock (IEAllTyp */

ITU_TCR1 = 32,
ITU_TCR2 = 32.
ITU_TCR3 = 32.
ITU_TCR4 = 32.


ITU_TIOR0 = 2.                  /* Select GRA as an output compare register
                                   1 in output as the ENDEA pin of GRA  comp on match
                                   GRB compare  no output */
ITU_TIOR1 = 2,
ITU_TIOR2 = 2.
ITU_TIOR3 = 2.
ITU_TIOR4 = 2.


ITU_TIER0 = 1.                  /* Enable IMIA interrupt to enter when it is
                                   requested by IMFA */
ITU_TIER1 = 1
ITU_TIER2 = 1.
ITU_TIER3 = 1.
ITU_TIER4 = 0.                  /* Disable all interrupt requests by timer 14 */

STCSR = 11.                     /* Select bit in CLR as a every bit  IMIA falling
                                   edge selected  On chip RAM enabled */

IPRA = 0.
IPRB   3                        /* Set all interrupts priority level to zero down */


P1DDR = 255.                    /* All Port 1 pins set for output */


ITU_GRA1 = 10000                /* Use an interval delay of 1sec before getting into
                                   the same loop  */

P1DR = xZero.                   /* Set Port 1 initially to xZero */


/*              GENERATE LOOK UP TABLE                  */

for (i=0; qSphereD >> =MAXSECTIONS) && qSphere < qD <= + qt && < pi, hpMain
{
qSphereD = qSphere-qD;
negAlpha2Red = (unsigned)qSphereD;
qSin = (float)sin((double)qSphereD) qnSinDivDeltaInc /*sec*(pi/qRad)* (Count/qMag)
qCos = (float)cos((double)qSphereD)*(2Encode/qMag)
}

/* redefining the following  because above loop gives wrong values for
qSin[0] q120[0] and q120[1]  the last four when using a MAXSECTIONS
of 512  Replace subroutine  Note these values only apply for a MAXSECTIONS of
60 only */

qSin[0] = 74884  /* 2000 */;
q120[0] = 0298  /* 1275 */;
q120[1] = 0298  /* 1275 */;

qSin++                  /* reset alpha after the above calculation loop */


ITU TSTR = 0            /* start Timer 14 */

}


/*  Engine cycle loop subroutine  */

void qSpeedInit(void)
{
```

```
{
ITU_T0TR = 0.    /* Stop Timer T0 */

ITU_T0IR &= ~1.  /* acknowledge timer 0 interrupt compare interrupt */

ITU_GRA0 = q;    /* load new value of GRA0 */

P1DR = q0;                    /* output appropriate signal */

ITU_T0TR = 2.    /* start Timer T1 */
}


interrupt(ITU_IMIA1) void timerT1(unsigned floating iomb)

{
ITU_T1TR = 0     /* Stop timer T1 */

ITU_T1IR &= ~1.  /* acknowledge timer 1 interrupt compare interrupt */

ITU_GRA2 = tk.   /* load new value of GRA2 */

P1DR = uk.                    /* output appropriate signal */

ITU_T1TR = 4.    /* start Timer T2 */
}


interrupt(ITU_IMIA2) void timerT2(unsigned floating iomb)

{
ITU_T2TR = 0.    /* Stop timer T2 */

ITU_T2IR &= ~1.  /* acknowledge timer 2 interrupt compare interrupt */

ITU_GRA3 = (Word.             /* load new value of GRA3 */

P1DR = q2ero.                 /* output appropriate signal */

ITU_T2TR = 0     /* start Timer T3 */
}


void initialize(void)
{

/*          Running Variables Initialization          */


/*          Running Variables Initialization          */

/*    Calculate Constant terms . . .  */


/* lookUpTable terms */
ka = 2*pi*pi;
kb = 8*sqrt(2)*pi*pi;
kb1 = kb*tSampling*tSqrd;
kb2 = kb*tSampling*tCurd;
oneOverSecond = 1/sqrt(2);

/* voltageMagnitude terms */
mVoltMag = (sqrt(2)/(sqrt(3)*pi*tSampling*maxMagVolt*2^15)
oVoltMag = (sqrt(2)*(sqrt(3)*pi*tSampling*maxMagVolt*2^15)          /* kV = 2^0 */



/* voltage sensing terms */
mVSense = (sqrt(2)/(sqrt(3)*pi*2^0 R).
oVSense = (sqrt(2)/(sqrt(3)*pi*2^0 R*2^12.          /* kV = 2^0 */
VSense = vRdf*mVSense*kV/2^1023

/* frequency sensing terms */
mFSense = (Max_Value/2.0 R)
oFSense = (Max_Value*4.0.
kf1 = 2Q*tSampling*vRdf*mFSense*tF/1023.
kf2 = 2Q*tSampling*oFSense*tF
oFSensing = (kf1*kf2)/2^2.0*oVRdf*kf2


/* piSupervisor  (MaxMinAdap  terms */
                                                                    /* 1024 = 2^10  bit resolution A/D converter */

MMaxFactor = MaxAD*2^1023*vRdf;
MMinFactor = MinAD*2^1023*oRdf;

kMMaxFactor = (kf*2^1023*4.0*vRdf)*kf2
kMMinFactor = (kf1*2^1023*2.0*oRdf)*kf2


/* kInstantaneous terms */
kSampl = MaximumSampling*4*tFactor;

kSampl = (kSampl/2)*tFactor;

sqrt2Kf = 50.*sqrt(2)*tFactor;
sqrt2Kf2 = sqrt(2)*2*pi*tFactor;
```

```
case 3: {qTemp=20; nbTemp=25;break;}                    /* vectors */
case 4: {qTemp=10; nbTemp=20;break;}
case 5: {qTemp=42; nbTemp=10;break;}
case 6: {qTemp=26; nbTemp=42;break;}
}               /* closing bracket for "switchinterval"  sequence not equal to 1 */

}               /* closing bracket for "else" - sequence not equal to 1 */

}               /* closing bracket for cooler change? */

else
{               /*opening bracket for "else" of power change? */

/* . . . select to and fr from the look up table  . . . */

<< brightness/alphaNumLamp1.
%i=0100;
%o=0100;


if (sequence = = 1)             /* forward sequence */
{               /* opening bracket for forward sequence */
qTemp = (%i*qMagn) >> 16;
nbTemp = (nb*nbMagn) >> 16;

}               /* closing bracket for forward sequence */

else                                    /* reverse sequence */
{               /* opening bracket for reverse switching */
qTemp = (nb*nbMagn) >> 16;
nbTemp = (%i*qMagn) >> 16;

}               /* closing bracket for reverse switching */

temp = qTemp;                   /* swap switching */
qTemp = nbTemp;                         /* vectors */
nbTemp = temp;                          /* ends */


}               /* closing bracket for "else" - power change? */

if (qTemp < nbTemp) qTemp=nbTemp;
if (nbTemp < nbTemp) nbTemp = qTemp;    /* to prevent unpredictable results due to skipping a loop for 0 seconds */

nbHighTemp = (nbTemp qTemp nbTemp) >> 1;                /* compare nbHighTemp */

if (qZeroTemp < > 0)                                    /* change */
qZeroTemp = 0;                  /* the zero */
else                                                    /* switching */
nbZeroTemp = 0;                                         /* vector */


while (qTemp  TSTR != 233)              /* wait until */
{                                       /* test zero sequence */
}                                       /* of waveform is reached & until T2interrupt is completed */

qi = qTemp;
ni = nbTemp;
                                        * updating the */
qi = qTemp;                             /* parameters */
ni = nbTemp;
                                        /* for the */
qZero = qZeroTemp;                      /* sequence */
nbHigh = nbHighTemp;                    /* algorithm */


}               /* closing bracket for the Do Forever loop */

}               /* closing bracket for main */


interrupt [T2] void timer2InterruptRoutine (void)

{
LED_TSTR = 0;   /* Stop timer 2 */


[3] TSTR.0 = 1; /* acknowledge timer 2 interrupt compare interrupt */


/* Note "pause 1 us [0]" is equal the 1 bit of the LED[3] on microcontroller
   board pulse at five steps we do not know in exact why other higher
   priority interrupts that can interrupt this are going enterrupt
   service routine. The ADJ% on I interrupt will be a Non Maskable Interrupt
   which will not be cause an interrupt at any time  whether the I bit of
   the CCR is set or not */


TCR_nMAX = nbHigh;              * load new value at t=Max *

TCR_TSTR = 1;           /* start Timer1 bit */
}


interrupt [T1] void timer1InterruptRoutine (void)
```

```
unsigned int temp,temp1,flag1;                    /* temporary variable */


phase1=phase;

phase = (unsigned)(phase*0.703125);     /* angle displacement w.r.t the horizontal */


/* an alternative way to finding alpha1 and sector1update */
/*
sector1update = phase/60000000;
alpha1 = phase-sector1update*60000000
sector1update= sector1update + 1;
if (phase / 60000000)
{
if (alpha1 <= 60000)
counter=0;
else
counter = 1;
}
*/



if (phase<=  1800000000)
{
if (phase<=  600000000) {alpha1deg = phase  sector1update = 1;}
if (phase>  1200000000) && (phase <= 1800000000) {alpha1deg = phase-1200000000  sector1update = 2;}
if (phase>  600000000) && (phase <= 1200000000) {alpha1deg = phase -600000000  sector1update = 3;}
}
else
{
if (phase>= 2400000000) && (phase <= 3000000000) {alpha1deg = phase -2400000000  sector1update = 4;}
if (phase>  3000000000) && (phase <= 3600000000) {alpha1deg = phase -3000000000  sector1update = 5;}
if (phase>  1800000000) && (phase <= 2400000000) {alpha1deg = phase -1800000000  sector1update = 6;}
if (phase <= 3600000000)
alpha1deg = phase -3000000000  sector1update = 1;
if (alpha1deg <= 60000000)
counter=0
else
counter = 1
}
}



 /* alpha1 = Angle w.r.t first vector of the present sector * 100.000*/

if (sector != sector1update)          /* Sector change ? */
{               /* on      istrupt for sector change? */

/*          ) change of sector          */

sector = sector1update;


/*      select ts and th from the look up table     */

ts = tslook[alpha1deg];
th = thlook[alpha1deg];

sequence = sequence     /* so number sequence does not change when there is
                a sector change  */

if(sequence == 1)
{               /* opening bracket for "if (sequence == 1)  change of sector loop */

temp = (th*ts1flag) + ts;
flag1 = (th*ts1flag) + th;

switch(sector)
{               /* opening bracket for "switch(sector)   sequence 1 */
case 1: {ts=temp+0; ts1flag=40;break;}          /* select appropriate
case 2: {ts=temp+21;ts1flag=21;break;}          /* switching */
case 3: {ts=temp+21;ts1flag=21;break;}              /* vectors */
case 4: {ts=temp+21;ts1flag=21;break;}
case 5: {ts=temp+14;ts1flag=14;break;}
case 6: {ts=temp+7; ts1flag=7; break;}
}               /* closing bracket for  switch(sector)   sequence 1 */

}               /* closing bracket for "if (sequence == 1)  change of sector loop */


else                                    "
{               /* opening bracket for "else"  sequence = not equal to 1 */

flag1=temp = (th*ts1flag) + ts;
th1flag = (th*ts1flag) + th;

switch(sector)
{               /* opening bracket for "switch(sector)   sequence not equal to 1 */

case 1: {ts=temp+0; ts1flag=40;break;}          /* select appropriate
case 2: {ts=temp+21;ts1flag=21;break;}          /* switching */
```

```
int error, controlUpdate, requeues;

/*  - FrontEndUpdate variables --- */

long ulMag,
float ulUtilMag,
long ulUtilMag,

/*  - pendingTimes variables --- */

unsigned int ulTemp, thTemp, ulUtilTemp, tUtil,
unsigned long tu, th;

/*  - interMagFactors variables  - */

unsigned int ulTemp, utTotal, ulZeroTemp, temp,

/*    dataMag parameters  - */

int uj, ut, uZero, tj, th, rUtil,

/*  phaseDirector  phaseLockLoop variables  - */

unsigned int vMagChan, vInpChan, vInp1, vInp2, vInv1, vInv2, vU tUtil,
int zUtil, vMagDuration, vInpDuration, vInpUtil, vInvUtil, postTimer, negTimer,

/*  pllCurrents  phaseLockLoop variables - */

unsigned int vMagCount, UtilFactor, tUtilFactor,
unsigned long tUtilFactor, tPUtilFactor,
int zCorrectUtil, tmpUtil,
unsigned long limitFactor, tPUtilized,

/*  - frequency scaling variables  */

unsigned int tMagADDR  tmpADDR
float utTemp, ptTemp
unsigned long tPT, tPF

/*  - voltage compare variables  */

long vVTable  tVTemp  vInvTemp, vMagTemp,
float mVTemp

main (void, char);

float maxDeviation = 340

int overload = false

/*****************************************************************
*                                                               *
*****************************************************************/

void configureLoop()
void phaseLockLoop()

{  task and clock loops
{                /* opening bracket for main loop */


initialize()

/*              MAIN LOOP              */

for ( )
{                              /* Opening Brace for the Do Forever loop */

while (TASK = 270)             /* Condition to perform main task and
                                   acquire a id used for completion of Interrupt routine */
{
}


tmpADDR = tmpADDR              /* Assign pointer */

tmpADDR = counter + 1          /* Task 1 */


timeLong = 0xFFFF
vMagLong = 0x1234
vMagUtil = 0xh
timeUtil = 0xs
```

Appendix I: Source Code for SDEC0407.C

```
/*                                  SDEC0407                              */

/* Program to perform the full DSP control of the session */

/* Sampling frequency  8.26 KHz */

/* Version 0.2 */
/* Author  Mamadou Bayoo */
/* Date  08/05/96 */


/* Copyright

   This program is similar to session 0.6, but with the necessary modifications
   to use data that will be output by the A/D converter. Notice that A/D
   conversion is not yet performed here, only the different variables are
   loaded accordingly, so that when the A/D conversion is to be performed, then
   the values delivered by the A/D interrupt routine is properly interpreted
   and accepted by the rest of the program. There revision or three of the
   program affected by these changes are

            a. The look up Tytes
      b. The calculation of analog (vectorMagnitude)
      c. The phase lock loop

   As before, in place of the desired parameters, inverter voltage and frequency
   reverse voltage and frequency  dummy values are used  for testing purposes

   The values that are eventually output via the PWM algorithm itself remains
   unaltered
*/

/* Result
   Program runs successfully  suggesting that the A D  conversion can now
   be implemented
*/



#include <c:\work\h>

#pragma language = extended          /* Enable the use of extended keywords */
#include <dsp32c40.h>                /* internal registers define  */
#include <dsp32000.h>                /* interrupt vector addresses define  */

/*              Global Constant Terms                */

#define pi 3.14159265354
#define clock 50.0E6                         /* As a clock frequency of 10MHz */
#define CLKFREQ_FOR_1MS 250
#define fSampling 16.3                       /* As a sampling frequency of thirty */
#define vDig 4095
#define vDigHalf 2047
#define vDigQuart 1023
#define fMin 45                              /* define lowest limit of frequency variable */
#define fMax 55                              /* define upper limit of frequency variable  */
#define fSecond 50
#define vInstMax 2047                        /* define inverter high */
#define vInstMin 2043                        /* define inverter low */
#define clear 10                             /* Minimum value after the zero crossing times are allowed to hold */
#define vRef 4.61  /* A D Reference Voltage */
#define vMagMax 0.9
#define vMagMin 0.5e-10



/*              Running Variables                */

/*     Interrupt variables     */

unsigned int flag1,
flag2, alphaAcc

/*     Toggling variables     */

float alpha, oldAlpha, newAlpha;
int i;
int flag1, flag2, flag3, flag4, newAlphaFlag;

/*     Acquiring input data variables     */

int i, count;
unsigned long phaseA, alphaAcc, oldAlpha;
int flag1, flag2;

/*     vectorMagnitude variables     */
```

```
}

void pAnalock(void)
{

vInst = vInst2;
vInst1 = vInst2;
vInst2 = vPos;
vInst2 = vInst;
vDelV = vInst - vInst1;
if (vDelV < 0)
vDelV = -vDelV;


vNextDelV = vInst2 - vInst1;
if (vNextDelV >= 0) vNextDirection = positive;
else vNextDirection = negative;
vInstDelV = vInst2 - vInst1;
if (vInstDelV >= 0) vInstDirection = positive;
else vInstDirection = negative;

if (vPos < vMin) || (vPos > vMax)  /* Phase lock loop  inverter frequency ok */
{                    /*open bracket for ,, vPos out of limit*/
vSwitchInhibit = true;
P6DR = 1;                          /* Send vSwitchInhibit signal on pin 0 of port 6 */
line = thermal;

if (vNewAction < -5Hz) || (vNewAction > (vMax)) || (vInst < vInvMin) || (vInst > vInvMax))
||                   /* inverter ok */

else
{ /* open bracket for inverter failure*/
invInhibit = true;
while (vInst > vNextCount)
{
}
vSwitchInhibit = false;
P6DR = 0;                          /* Cancel vSwitchInhibit signal on pin 0 of port 6. */
} /* close bracket for inverter failure*/
} /* close bracket for vPos out of limit*/

else
{ /* open bracket for vPos within limit */
vInv = vPos;
if (vDelV >= vDelVMax)
{ /* open bracket for vDelV >= vDelVMax   not in phase*/
vSwitchInhibit = true;
P6DR = 1;                          /* Send vSwitchInhibit signal on pin 0 of port 6 */
if (vPos < vMin)
line = vMax;
else
line = vMin;
} /* closing bracket for vDelV >= vDelVMax   not in phase */

else
{ /* open bracket for vDelV ok */
if (vNextDirection = vInstDirection)
{
 SwitchInhibit = false;            /* in phase */
P6DR = 0;                          /* Cancel vSwitchInhibit signal on pin 0 of port 6 */
}
else
{ /* open bracket for opposite direction */
vSwitchInhibit = true;
P6DR = 1;                          /* Send vSwitchInhibit signal on pin 0 of port 6 */
if (vPos < vMin)
line = vMax;
else
line = vMin;
} /* closing bracket for opposite direction */
} /* closing bracket for vDelV ok*/
} /* closing brackets for Vpos within limit */

/* end of phase lock loop */
}


interrupt [NMI] void overloadInterrupt (void)
{
for ( )
{
P1DR=0;                /* The only way to get out of the loop is to reset the microcontroller */
}
}
```

```
}                       /* closing bracket for "else" statement */

if (qTemp<=1Wval) qTemp=1Wval;
if (qTemp<=1Min) qTemp=1Min;      /* to prevent unpredictable results due to starting a timer for 0 seconds */

qWinitTemp = qTemp qTemp qTemp >> 1;                    /* compare qWinitTemp */

qrZeroTemp += 2;                                        /* change */
mZeroTemp = 55;                     /* the new */
skip                                                    /* new change */
mZeroTemp = 3.                                          /* vector */


= - mIO_1STR != 232)                /* wait until */
{                                                       /* first zero sequence */
                                                        /* of which return is reached, i.e until T3interrupt is completed */

qj = qjTemp.
tj = tjTemp.
                                                        /* equating it to */
vk = vkTemp.                        /* parameters */
tk = tkTemp.
                                                        /* for the */
mZero = mZeroTemp.                  /* interrupt */
qWinit = qWinitTemp.                /* algorithm */


}                       /* closing bracket for the Do Forever loop */

}                       /* closing bracket for main */


interrupt [0x10  0x0A3] void timer3Interrupt(void) /*interrupt level 3 routine*/

{
IRQ_1STR = 0.     /* Stop timer T3 */

IRQ_1STR &= ~1.  /* acknowledge timer 3 interrupt compare a interrupt */


/* Note, "write exx XOR" instruction is used the 1 bit of the CCR is not necessary
   here since at this stage we do not have to worry any other higher
   priority interrupts that can interrupt this as going interrupt
   service routine. The SCC% int interrupt will be a Non Maskable interrupt
   (NMI) and so it can cause an interrupt at any time, we do not care the I bit of
   the CCR is set or not  */


IRQ_GRA3 = qWinit;                  /* load new value of GRA3 */

IRQ_1STR = 1.          /* start timer T3 */
}


interrupt [0x10  0x0A4] void timer4Interrupt(void) /*interrupt level 4 routine*/

{
IRQ_1STR = 0.     /* Stop timer T4 */

IRQ_1STR &= ~8.  /* acknowledge timer 4 interrupt compare interrupt */

IRQ_GRA4 = tj.   /* load new value of GRA4 */

P1DR = vj                           /* output appropriate signal */

IRQ_1STR = 2.          /* start timer T4 */
}


interrupt [0x10  0x0A5] void timer5Interrupt(void) /*interrupt level 5 routine*/

{
IRQ_1STR = 0     /* Stop timer T5 */

IRQ_1STR &= ~16.  /* acknowledge timer 5 interrupt compare interrupt */

IRQ_GRA5 = tk.   /* load new value of GRA5 */

P1DR = vk.                          /* output appropriate signal */

IRQ_1STR = 4          /* start timer T5 */
}


interrupt [0x10  0x0A6] void timer6Interrupt(void) /*interrupt level 6 routine*/

{
IRQ_1STR = 0     /* Stop timer T6 */

IRQ_1STR &= ~32;  /* acknowledge timer 6 interrupt compare interrupt */

IRQ_GRA6 = mZero;                   /* load new value of GRA6 */

P1DR = mZero                        /* output appropriate signal */

IRQ_1STR = 8          /* start timer T6 */
```

```
if sector := 360000000{
alpha1 = phase 360000000. vectorUpdate = 1.
if (alpha1 <= gamma)
counter=0.
else
counter=1.
}
}


/* alpha1 = Angle of t i first vector of the present sector * 100 000*/

if (vector != vectorUpdate)          /* Sector change ? */
|                    /* opening bracket for sector change?  */

/*...  .. .. .. Change of sector  .. . . . .. . . . . ..*/

vector = vectorUpdate.


/*   - Select ta and tb from the look up table   -   */

i= (alpha1/alphaMax1).
ta=tb ..i.
tb=tb[i].

Sequence =  sequence.   /* resultant  sequence does not change when there is
                a sector change */

if (sequence == 1)
|                    /* opening bracket for "if (sequence == 1)  change of sector loop */

xyTemp = (ta*xaMag) >> 16.
xbTemp = (tb*xaMag) >> 16.

switch (vector)
{                    /* opening bracket for "switch(vector)  sequence 1 */
case 1  {xyTemp=26. xbTemp=48.break.}          /* select appropriate */
case 2  {xyTemp=48. xbTemp=21.break.}          /* switching */
case 3  {xyTemp=21. xbTemp=29.break.}              /* sector1 */
case 4  {xyTemp=29. xbTemp=14.break.}
case 5  {xyTemp=14  xbTemp=42.break.}
case 6  {xyTemp=42. xbTemp=25.break.}
}                    /* closing bracket for "switch(vector)  sequence 1 */

|                    /* closing bracket for "if (sequence == 1)  change of vector loop */


else                                    /* sequence =  1 */
{                    /* opening bracket for "else" - sequence not equal to 1 */

xyTemp = (tb*xaMag) >> 16.
xbTemp = (ta*xaMag) >> 16.

switch (vector)
{                    /* opening bracket for "switch(vector)  sequence not equal to 1 */

case 1  {xyTemp=48. xbTemp=25.break.}          /* Select appropriate */
case 2  {xyTemp=21. xbTemp=48.break.}          /* switching */
case 3  {xyTemp=29. xbTemp=21.break.}                                  /* vector */
case 4  {xyTemp=14. xbTemp=29.break.}
case 5  {xyTemp=42. xbTemp=14.break.}
case 6  {xyTemp=25. xbTemp=42.break.}
}                    /* closing bracket for "switch(vector)"  sequence not equal to 1 */

}                    /* closing bracket for "else"  sequence not equal to 1 */

}                    /* closing bracket for sector change? */

else
{                    /*opening bracket for "else" of sector change? */


/*      select ta and tb from the look up table          */

i= (alpha1/alphaMax1)
ta=ta[i]
tb=tb[i];


if (sequence == 1)                  /* forward sequence */
{                    /* opening bracket for forward sequence */
xyTemp = (ta*xaMag) >> 16.
xbTemp = (tb*xaMag) >> 16.

}                    /* closing bracket for forward sequence */

else                                    /* reverse sequence */
{                    /* opening bracket for reverse switching */
xyTemp = (tb*xaMag) >> 16
xbTemp = (ta*xaMag) >> 16.

}                    /* closing bracket for reverse switching */

temp = xyTemp                   /* swap switching */
xyTemp = xbTemp                                 " vectors "
xbTemp = temp                                                   " order "
```

```
P1DDR = 255.                    /* All Port 1 pins set for output */

P5DDR = 1.                      /* Configure pin 0 of port 5 as an o/p port */


TIU_CNT3 = 10000.               /* Set an initial delay of 1ms before getting into
                                   the main loop  */

P1DR = nZero.                   /* Set Port 1 initially to nZero */


/*..... GENERATE LOOK-UP TABLE ..... */

for (n=0, alpha=0; n <= (MAXSECTORROW-1); alpha += DB; n++, alpha++) alpha1[n]
{
alpha1[n] = pi*alpha/180;
sinAlpha[n] = sin(alpha1[n]);
tk30[n] = (long int)(sin(alpha1[n])*centreSyllen)*pixelAlpha[n]);
tk31[n] = (long int)(cos(alpha1[n])*pixelAlpha[n]);
}

/* redefining the following, because above loop gives wrong values for
tk3[30], tk120[30], and tk120[15] - the last two when using a MAXSECTORROW
of 60. Hence redefine HERE these values only apply for a MAXSECTORROW of
60 only. */

tk30[0] = 2234;                 /*24960 */      /*4668.*/
tk120[0] = 1343.  /*8433.*/     /*2605.*/
tk120[1] = 1343.  /*8433.*/     /*8596.*/

alpha=0,        /* reset alpha after the above calculation loop */


TIU_TSTR = 0.       /* start Timer T3 */

/* ..... MAIN LOOP ..... */

for (;;)
{                               /* Opening Bracket for the De Forest loop */

while (TIU_T3TR != 225)                 /* Condition to proceed: wait for last pwm
                                           sequence + 9 to wait for completion of T3interrupt routine */
{
}


sequence = sequence;                    /* Change sequence */

counter = counter + 1;                  /* Up cnt */
tAcc = counter * 16.            /* next sampling period */


/* Sample xyMag. (Red, blue and yMag here */

/*xyMag = 52[7].*/          /* Need values not assigned to xyMag. (Red xMag and yMag
Mag = 600. Iter = 600.                  /* but once they are sampled these */
xInc=230  yMag=230.                     /* code lines must be */
yMag2=227. yInc2=75.                    /* removed */

xIncLong = xInc<<.16.
xyMag = (xIncLong>>16);                 /* computing xyMag */

/*xyMag = 37208.*/ /*58794.*/     /* forcing xyMag to achieve/plot  for testing purposes only */

photoClockInput[1.

phase = (30*iSec*tAcc),         /* angle displacement w.r.t the horizontal */


/*sector = 0.*/                         photoSensorInput[0];
/*phase = 359.99.*/                     photoSensorInput[1];

/* an alternative way to finding alpha1 and sectorUpdate */
/*
sectorUpdate = (phase/60000).
alpha1 = phase-sectorUpdate*60000.
sectorUpdate=sectorUpdate+1.
if (phase >= 360000)
{
if (alpha1 >= maximum)
counter=0
else
counter=1
}
*


if (phase <  60000)
{
if (phase <  60000) {alpha1 = phase; sectorUpdate = 1;}
if (phase >  120000) && (phase <= 60000)  {alpha1 = phase-60000; sectorUpdate = 2;}
if (phase >  180000) && (phase <= 120000) {alpha1 = phase-120000; sectorUpdate = 3;}
}
else
{
if (phase >  240000) && (phase <= 180000) {alpha1 = phase-180000; sectorUpdate = 4;}
if (phase >  300000) && (phase <= 240000) {alpha1 = phase-240000; sectorUpdate = 5;}
if (phase >  360000) && (phase <= 300000) {alpha1 = phase-300000; sectorUpdate = 6;}
```

```
/* phase-lock loop variables */
int sSwitchInhibit, tmrInhibit, vDesQuantum;
int vMax, vRef1, vRef2, vIerr, vIerr1, vIerr2, vErrorCheck, prevErrorC, negErrorC,
int IIerrAbsol, vSysMin, vIerrMax, iMax, iMin;
int vDiff, vPosDiff, vIerrDiff, vIerrDerivSum, vDiffMax;

/*****************************************************************
*                          MAIN                                  *
*****************************************************************/

void phaseLockLoop (void)

{  task void main (void)
{                    /* opening bracket for main loop */

/*-------- Running Variables Initialization --------*/

int i=0;  /* var included just to be able to see the stack when debugging*/

iSampling = 1.8E-6;
iS = (TimerPeriod*iSampling*4000000);
sysmath = -1;
index = 1;
counter = -1;
cpTemp = 45, mTemp = 35, zeroTemp = 7,
ej = 45, ek = 35, mZero = 7,

timeVal = AbsoluteSampling(0*iClock);

ti = timeVal, t1 = (int)t1 (0*diff = time-0)

ia = 0;
ib = 0*sqrt(2)/pi*pi;
ia1 = ia*iSampling*iClock;
ib1 = ib*iSampling*iClock;
absDivByReal2 = (int)iSi;

iSamp = AbsoluteSampling(iClock);

alphaVal = (0xaaaSecondDiv,
alphaVal2 = alphaVal*100000;

gamma = (360*(int)sin)*iSampling*7.

vSysMin=207; vIerrMax=253.          /*  <7  10% about 230?  */
val = (int)vIerr vSysMin<alphaVal.
ej = ek,
e = (vIerrMax and 0xA5A5)*0x55E,
f*e = e1*65535.*/


i1... = 40.

sSwitchInhibit = true.
P5DR = 1,                    /* Used sSwitchInhibit equal as pin 0 of port 5 */
tmrInhibit = false.
vRef2= vDig.
vIerr2 = vSmall.
vDesQuan = 0.01*vIerrMax5.
prevErrorC = 2. negErrorC = 3.
iMax=480. iMin=340.
vDiffMax = 0.01*vIerrMax.

/*  -  -  H2048 Registers Initialization - - - -  */


TIU_TCRD = 32.               /* Select CRA compare match as Clearing Source
                                Select Internal Clock (1E6Hz) */
TIU_TCR1 = 32.
TIU_TCR2 = 32.
TIU_TCR3 = 32.
TIU_TCR4 = 32.


TIU_TIORD = 2.               /* Select CRA as an output compare register
                                1 at output on the TIOLA pin at CRA compare match
                                CRB compare  no output */
TIU_TIOR1 = 2
TIU_TIOR2 = 2.
TIU_TIOR3 = 2.
TIU_TIOR4 = 2


TIU_TIERD = 1                /* Enable IMIA interrupt to occur when the
                                requested by IMFA */
TIU_TIER1 = 1
TIU_TIER2 = 1
TIU_TIER3 = 1
TIU_TIER4 = 0                /* Disable all interrupt requests by Timer T4 */

bVSUR = 11                   * Select IIRbit in ICR as a user bit  IMAI taking
                                edge selected  (by chg IMAI enable) */


IPRA = 0
IPRB = 0                     /* Set all existing interrupts priority level to zero  level */
```

Appendix H: Source Code for SDEC0406.C

```
/*                                          SDEC0406                                    */

/* Program to perform the A/D SVM control of the inverter */

/* Sampling frequency, 8.55 KHz */

/* Version 1.0 */
/* Author: Henning Hopen */
/* Date: 23/02/96 */
/* Status: Approved */

/* Description:
   Same as version 0.5, but includes the implementation of the 200KHz interrupt
   and as well as a signal sent to an output pin as a "static Switch inhibit
   condition. As before: the double Dataset latches flag is set as an out of
   phase condition. In addition the transitory states and the time delay to
   prevent the short of the turn off time of the IGBTs have been removed. These
   are to be taken care of by the hardware

 */

/* Results
  The program runs successfully both on the simulator and the emulator!
*/



#include <math.h>

#pragma interrupt = extended        /* Enable the use of extended keywords */
#include <m68332dh.h>               /* internal registers define  */
#include <m68332intv.h>             /* interrupt vector addresses define  */

/* ... ... ... Define Constant Terms ... ... */

#define pi 3.141592654
/* #define fClock 32.5E 6 */                    /* At a clock frequency of 16MHz */
#define MAXSECTORNUM 240

/*  ... ... ... Define Parameters ... ... ... ... */

/*#define tSampling 16.3*/                      /* At a Sampling frequency of 7kHz */
#define tNominal 600
#define vDig 400
#define vSmall 0
#define vSaturat 250
#define vdiffpMin 0.0
#define vdiffpMax 0.866604

/*              ... ... Running Variables ... ... ... */

float fClock = 32.5E 6.

int sequence, sector, counter,

int qTemp, vTemp, vZeroTemp, temp,

unsigned int tTemp, tSTemp, tDiffTemp,

int Ui, vA, vZero, Ia, Ib, tDiff1,

long vAlDig                         /* vAlDig changed to int */

int tDir, time,

flag alpha        /* 1 changed to (Act */

float alphaRad

float tSampling,

int i, sectorUpdate, tStamp,

unsigned long ta, tb

enum {false, true},

int calTestOfLoop

int k , limited tAdv,

int tMAXSECTORNUM1

float alphaMin

float ta, tb1, tb, tb2, oneMinDyRatio3, oneAlphaRad
float oneLSectorLen = 240.
int i, m
unsigned long phase, alpha1, alphaMin1, tAct, gaining
long elevating, z
float int, r1
```

```
{ tlineAddr1 = tlineADDR1; }

if (yTemp >= (tlineAddr1 + (tvHeight * tlineWidth)) || (yTemp < tlineAddr1))
{ tlineADDR = tlineADDR1;
tlineLong = tlineADDR*tVScale + tVShort;
}

if (tvHeight < tlineAddr1) || (tvHeight >= tlineAddr2))
{ tlineADDR = tlineADDR2;
tlineLong = tlineADDR*tVScale + tVShort;
}

tshLong = tvLong*tLong;                         /* computing tshLong */

pPointLongX;

pPoint = tlineFactor*counter;       /* angle displacement w r t the horizontal */

if (pPoint <= 180000000)
{
if (pPoint <= 90000000) {alphaLong = pPoint; sectorUpdate = 1;}
if (pPoint <= 120000000 && pPoint >= 90000001) {alphaLong = pPoint-90000000; sectorUpdate = 2;}
if (pPoint <= 150000000 && pPoint >= 120000001) {alphaLong = pPoint-120000000; sectorUpdate = 3;}
if (pPoint <= 180000000 && pPoint >= 150000001) {alphaLong = pPoint-150000000; sectorUpdate = 4;}
}
else
{
if (pPoint <= 240000000 && pPoint >= 180000001) {alphaLong = pPoint-180000000; sectorUpdate = 5;}
if (pPoint <= 270000000 && pPoint >= 240000001) {alphaLong = pPoint-240000000; sectorUpdate = 6;}
if (pPoint <= 300000000 && pPoint >= 270000001) {alphaLong = pPoint-270000000; sectorUpdate = 7;}
if (pPoint <= 360000000 && pPoint >= 300000001) {alphaLong = pPoint-300000000; sectorUpdate = 8;}
if (pPoint == 360000000);
alphaLong = pPoint-300000000; sectorUpdate = 1;
/* tanAlphaLong = -pPoint;
counter = 0
else
pPoint = 1.
{
}
```

```
  /* alphaLong = &  le = > first vector of the present sector * 100 000*/

if (sector != sectorUpdate)             /* Sector change = */
{               /* opening bracket for sector change* */

/*              Change of arrays  -->       .       .       .  */

sector = sectorUpdate;


/*      select fx and fy from the look up table  --  .  */

x = (alphaLong-alphaAdjLong)
fx = fx[i];
fy=fy[i];

tfxSquare =  sqSquare;   /* remember sequence does not change when there is
                        a sector change */

if (sequence == 1)
{               /* opening bracket for "if sequence == 1)  change of sector loop */

xyTemp = (fx*xyMagX) >> 16;
yxTemp = (fy*xyMagY) >> 16;

switch (sector)
{               /* opening bracket for "switch(sector)"  sequence 1 */
case 1  {xyTemp=35; yxTemp=43;break;}           /* select appropriate */
case 2  {xyTemp=43; yxTemp=21;break;}           /* switching */
case 3  {xyTemp=21; yxTemp=20;break;}               /* vectors */
case 4  {xyTemp=20; yxTemp=14;break;}
case 5  {xyTemp=14; yxTemp=42;break;}
case 6  {xyTemp=42; yxTemp=35;break;}
}               /* closing bracket for "switch(sector)"  sequence 1 */

}               * closing bracket for "if (sequence == 1)  change of sector loop */


else                                    /* sequence = 1 */
{               /* opening bracket for "else"  sequence not equal to 1 */

yxTemp = (fy*xyMagX) >> 16;
xyTemp = (fx*xyMagY) >> 16;

switch (sector)
{               * opening bracket for "switch(sector)"  sequence not equal to 1 */

case 1  {xyTemp=49; yxTemp=35;break;}           /* Select appropriate */
case 2  {xyTemp=21; yxTemp=49;break;}           /* switching */
case 3  {xyTemp=20; yxTemp=21;break;}                                   /* vectors */
case 4  {xyTemp=14; yxTemp=20;break;}
case 5  {xyTemp=42; yxTemp=14;break;}
case 6  {xyTemp=35; yxTemp=42;break;}
}               * closing bracket for "switch(sector)"  sequence not equal to 1 */

}               * closing bracket for "else"  sequence not equal to 1 */


}               * closing bracket for if x > change */
```

```
/*  - assembly/data variables -  */

int vector, vectorUpdate, sequence,


/*    Vectorblaqnibeb variables      */

long vVblaq,
float vflqVblaq,
long vBlaVblaq,


/*  - switchingTimes variables      */

unsigned int tjTemp, tkTemp, tDiffTemp, tBlaTemp,
unsigned long tk, tBi;


/* -- switchingVectors variables     */

unsigned int vjTemp, vkTemp, vZeroTemp, temp,


/*    - detailMdy variables -  */

int vj, vk, vZero, tj, tk, tDiff,


/*  - phaseDetector - phaseLockLoop variables -    */

unsigned int vBeat, vBeat2, vDet1, vDet2, vDetVec,
int vDet, vNewDirection, vDetDirection, vBeatDiff, vn1Det, vDet0Diff, vDiff, posDirect, negDirect,


/*  - pllSequencer - phaseLockLoop variables -    */

unsigned int vDetCount1, vDetCount2, tDiffFactor, tMaxFactor, tMinimumFactor, vMaxFactor, vMaxFactor
unsigned long tPhaseFactor, tPhaseFactor, tPMaximum, tMinFactor,
int aSpeedUnchDet, tPeakDet,


/*  - - Loopmonkey sensing variables      */

unsigned int tMaxADDR1, tMaxADDR = 0, tMinADDR1, tMinADDR = 0,
int tMaxDet, tMinDet,
float mFSense, dFSense,
unsigned long bF1, bF2,

/*      voltage sensing variables      */

long  vSense, vVSense, vSenSens, vPeakSens,
unsigned int vMaxADDR1, vMaxADDR = 0, vMinADDR1, vMinADDR = 0,
float aVSens,

enum {false, true},

float maxCounterDiv = 240

int sysClock = false,


/*****************************************************************************
*                                                                    MAIN
*****************************************************************************/

void commutation Peak(),
void phaseLockLoop (void)

{ /* Mark end main event*/
|                    /* opening bracket for main loop */

initialization()

/*                    START LOOP                    */

for (;;)
{                                  /* Opening Brackets for the for forever loop */

ADCSR = ADC_SETUP; /* start A/D conversion - A D conversion is stopped at the end of each A D conversion routine */
while (ADCSR & 0x80 != 0x00)                    /* Condition to proceed - wait for high zero
                                                    Equivalent to D'cent for completion of A/D conversion routine */
{
|


vSysClock = sequence;                          /* Change sequence */

counter = counter + 1                          /* Go to */


tMaxDet = PhaseDet[tMaxADDR];
tMinDet = MaxADDR[tMinADDR];
vMaxDet = vMaxADDR[vMaxADDR];
vMinDet = vMaxADDR[vMinADDR];

if (tMaxDet - tMinDet) > (tMaxDet - tMinDet)
{ tMaxADDR = tMaxADDR; }

if (tMaxDet - tMinDet) > (tMaxDet - tMinDet)
```

Appendix K: Source Code for SDEC0410.C - First Approved Version

```
/*                                  SDEC0410                              */

/* Program to perform the full SVM control of the inverter */

/* Sampling frequency: 4 kHz */

/* Version 1.0 */
/* Author: Hasanuj Gupta */
/* Date: 28/05/29 */


/* Description
   In this version the instability observed in the output waveform (current
   & V) was solved by:

   a. adding an hysteresis procedure to prevent unwanted fluctuations of the
      control parameters

   b. reducing the sampling frequency to 4 kHz

   The program also include an inverter OFF interrupt routine

*/

/* Result:
   The output waveforms obtained considerable
   The feedback loop works properly. The control algorithm & status switch
   exhibit hysteris, inverter off, etc ) also performed correctly
*/


#include <math.h>

#pragma language = extended          /* Enable the use of extended keywords */
#include <ioh32401b.h>               /* external registers define  */
#include <inh83003.h>                /* interrupt vector addresses define  */

/*          Define Constant Values  . . . . .  */

#define pi 3.142857143
#define fClock 12.5E6                        /* At a clock frequency of 10MHz */
#define MAXSECTORDRV 240
#define fSampling 2.5E4              /* At a sampling frequency of 5kHz */
#define vBig 400
#define vSmall 0
#define vNominal 230
#define fMin 48                      /* define lower limit of frequency window */
#define fMax 51                      /* define upper limit of frequency window */
#define fNominal 50
#define vInvOn 203                   /* define inverter high */
#define vInvOff 203                  /* define inverter low */
#define iMax 40                      /* maximum value that the switching mosfet are allowed to take */
#define vRef 4.01  /* 5.0 Reference Voltage */
#define mskRegMsk 0 6
#define mskRegMsk1 0 660504

#define mcr0411 50
#define mcrDiff1 50

#define CLR_ASOF 0x01
#define DIR_EXT 0x7F
#define ADC_RESUP 0xF3
#define CNT_ACT 0x7F

/*              Declaring Variables                 */


/*      Initialization variables      */

unsigned int counter;
Byte  alphaMax;


/*      mskRegTable variables     */

float alpha  deltaRad  newAlphaRad
int
enum eMAINSELECTSWINDOW{
float ka  kaT  kb  kbT  newDutyRatio1;


/*      regMskTable argument variables      */

int i, number
unsigned long phase  alphaDeg  alphaMod deg
unsigned int delmax
```

```
aswitchbit = true,
/DU_TIME = 0.  /* Stop all timer counters. */
while (/DuLong > offsetTime)
{
}
aSwitchbitstat =false,
PB0D = 0,                       /* Cancel aSwitchbitbit signal on pin 0 of port 0 */
Set.1
{}              /* the only way of re-starting the inverter after an inverter failure is by restarting the microcontroller itself  */
} /* close bracket for inverter failure*/
} /* close bracket for three out of times*/


/* ---- CLOSE IF VOLTAGE WITHIN LIMITS, PERFORM PHASE CHECK VOLTAGE DIFFERENCE CHECK -- */

else
{ /* open bracket for this within limit */


if (vDiff  > vDiffMax)

                /* -  IF OUT OF PHASE (VOLTAGE DIFF), PERFORM PHASE DIFF REDUCTION   */


{ /* open bracket for vDiff > vDiffMax  - not in phase*/
aSwitchbitbit =true,
PB0D = 1,                       /* Send aSwitchbitbit signal on pin 0 of port 0. */
if (RunADC1<> =(NominalFactor)
theFactor =offMaxFactor;
else
theFactor =diffMinFactor.
} /* closing bracket for vDiff > vDiffMax  - not in phase */


  /*  - CLOSE IF FIRST PHASE OK. PERFORM EXTRA PHASE CHECK (VOLTAGE DIRECTION)   */

else
{ /* open bracket for vDiff ok */
if (theVDirection = =lowDirection)

                /*  - IF IN PHASE AFTER SECOND CHECK MAKE INVERTER VOLTAGE POSITIVE   */

{ /* Remove and reverse options are in phase */
theFactor = diffMaxADC0 + 0F2,
aSwitchbitbit =false,
PB0D = 0                        /* Cancel aSwitchbitbit signal on pin 0 of port 0 */
}

                /* ELSE IF FAILED SECOND PHASE CHECK, PERFORM PHASE DIFF REDUCTION AGAIN   */

else
{ /* open bracket for opposite direction */
aSwitchbitbit = true
PB0D = 1,                       /* Send aSwitchbitbit signal on pin 0 of port 0 */
if (theADC0 <, =0) -  diffZero)
theFactor =diffMax;
else
theFactor =diffMinFactor;
} /* closing bracket for opposite direction */

} /* closing bracket for vDiff ok*/
} /* closing bracket for three minutes */

/* end of phase lock loop */
}


/*    ADC interrupt Subroutine    */

interrupt [0x11] void sampleADCinterrupt (void)
{

/*for (;;) /* the loop is put as a comment only for testing purposes */
{
PB0D=0;          /* the only way to get out of the loop is to reset the microcontroller */
}
*/

sampleOK = true

}


/*    A/D Interrupt Subroutine    */

interrupt [AD] ADC sampling interrupt (void)
{

digitADC0 = ADCDTA;  /* A
hexADC0 = *ADCDA;  /* B
difference = (ADCDTA - (0.5*hexADC0 + 0.5)*Vref
sampleADC0 = (ADCDTA + (0.5*hexADC0 + 0.5)*Vref
ADCSR = 0       /* Clear ADC end flag (ADCSR.7) to stop data conversion */

}
```

```
ITU_TCR0 = 32.                  /* Select CRA compare match as Clearing Source
                                   Select internal Clock (1/64kHz) */
ITU_TCR1 = 32.
ITU_TCR2 = 32.
ITU_TCR3 = 32.
ITU_TCR4 = 32.

ITU_TIOR0 = 2.                  /* Select 0/1 as an output compare register
                                   1 as output at the TIOCA pin at CRA compare match
                                   CRB compare: no output */
ITU_TIOR1 = 2.
ITU_TIOR2 = 2.
ITU_TIOR3 = 2.
ITU_TIOR4 = 2.

ITU_TIER0 = 1.                  /* Enable IMFA interrupt to occur when it is
                                   requested by IMFA */
ITU_TIER1 = 1.
ITU_TIER2 = 1.
ITU_TIER3 = 1.
ITU_TIER4 = 0.                  /* Disable all interrupt requests by timer 1A */

SYSCR = 19.                     /* Select 01 bit as CCR as user bit RAM backup
                                   edge reference  On chip RAM enabled */
IPRA = 0.
IPRB = 0.                       /* Set all interrupts priority level to zero (low) */

P1DDR = 255.                    /* All Port 1 pins set for output */

ITU_GRA3 = 18000.               /* Set an initial delay of 1sec before getting into
                                   the main loop  */

P1DR = 0xaa                     /* Set Port 1 initially as 0Xaa */


/*              SINE/COSINE LOOK UP TABLE                   */

for (i=0; alpha<0 ++; alpha<MAXSECTORDIV 1); alpha <= 90; i++, alpha = alpha+delta)
{
alpharad = pi*alpha/180.
sin[alpharad] = sin(alpharad);
sin[90] = (int)(sin(alpharad)*cos(90-alpharad)*128+sin(alpharad)*128+0.5);
cos[90] = (int)(cos(alpharad)*cos(alpharad)*128+sin(alpharad)*128+0.5);
}

/* Redefining the following 3 arrays above loop to get wrong values for
sine, sin120[90] and sin240[90]  for last two arrays using a MAXSECTORDIV
of 60. Reason unknown. These steps values only apply for a MAXSECTORDIV of
60 only  */

sin0[0] = 14484.    /* 113d, for 0 phase */
sin120[0] = 8564.   /* 133h, for 0 phase */
sin240[0] = 3564    /* 13Fh, for 0 phase */

alpha=0.            /* resets alpha after the above calculation loop */


ITU_TSTR = 8.       /* start Timer #3 */

}


/*   Phase Lock Loop Subroutine   */

void phaselockloop()
{

/*              PHASE DETECTION INITIALIZATION              */

sine2last = sine2reg = 0;
sine1last = sine1reg = 0
sine1 = sine2
sine2 = sine1.
sine2 = sine1new.
sDiff = sine1last-sine1last
if (sDiff < 0)
sDiff= sDiff.

sine1new = sin[ sine1reg ];
if (sine1last  <0) sine1direction = positive;
else sine1direction = negative.
sine2[0] = sine2 sine1
if (sine1[0]  <0) sine2direction = positive;
else sine2direction = negative;

/*      If SineA(1st) not at Zero, wait until it is at Zero to synchronize phase      */

if (sine1direction!= sine2direction) /* Phase is wrong, increase frequency of */
{               /*sine and set flag if sine out of sync*/
phaseshiftbit=true
P5DR = 1                        * Used to check phase signal on port 5 *
sine2=sine1                     * Synchronize sine2 to sine1 *


                * If Synchronized but not in Phase, then adjust Sine2 to sine1 *

if (sine1last!=0  && sine1new!=0 && sine1direction == sine2direction)
{ /* again check for incorrect phase*/
```

```
ITU_T5TR = 4.          /* start Timer T2 */
}


interrupt [ITU_GMA2] void timerT2InterruptRoutine (void)

{
ITU_T3TR = 0.     /* Stop timer T2 */

ITU_T3R2 &= ~ 2.  /* acknowledge Comp 2 interrupt compare x  except */

ITU_GMA2 = (UINT)t.                    /* load new value at GMA1 */

P10D1 = ~Zero.                         /* output appropriate signal */

ITU_T3TR = 0.          /* start Timer T3 */
}


void controlLoop (void)
{

/*  - - - - Running Variables Initialization                */

/*    Calculate Control terms  -  */


/* backUpTx2b terms */
b0 = 0(pu*pu).
b1 = 0*pu*2(pu)*pu.
b2 = b0*tSamplingClock.
b01 = b0*tSamplingClock.
antiDroBvReqd = (Unsigned)0.

/* controlMagnitude terms */
mfcMag = (pu)Max - fmMachineMagNum.mfcMag(Max).
mfcMag = fmormalInvMax.mfcMag*mfcMag(Min)*0.42;                    /* 512 = 2^9 */

/* Voltage steering terms */
mfcSpace = (pu)Max vbyMin)(4.5 0 5).
mfcSpace = (pu)Max   mfcSpace*0.5)* 512.                           /* 512 = 2^9 */
byStore = qRef*mfcStore*512/1023

/* Frequency steering terms */
mfcSpace = (Pu)(4.5MaxVal 5 0 5).
mfcStore = (Max   mfcSpace*4.5.
kF2 = 300*xSampling*vRef*mfCmax*1024/1023
kF1 = 300*tSampling*mfStore*512.

/* pICoprocessor  phaseCorrLoop  terms */ (* 1023 =  2^10   9) *10 bit resolution A-D converter */
fMaxFactor = (fmax)0.5*1023*pMax.
fMinFactor = (fmin)4.5 *1023*pMin.
fMaxMinFactor = fmMax)4.5 *1023*pMin.
kfMaxFactor = (kf)*1023*4.5   mfc+kF2
kfMaxFactor = (kf)*10_23*0.5*pMax)+kF2.
kfMaxMin = (kf)*1023*2.5*pMin)+kF2.

/* kIntegralTerm terms */
kIntInf = (fSamplingTime)*(kf*tSample)

fSamp = (fSampleSampling)*(pu).

alphaMin = 80-maxSectorDe
alphaMaxLong = alphaMax*0.95*0.90.

gamma = 1.00*(fNominal*(Sampling*10mm000)).


/*      Allocate values to variables      */

/* values for the A-D Conversion */
ADCSR = 0          /* clear ADST flag */
ADCR = 0           /* disable external trigger */
ADCSR = 1+7.       /* select          enable A-D interrupt
                                                    Start A-D conversion By putting this step delay as the initialization routine the first sample conversion was negatively active (the result of the first A-D conversion
are available before the main cycle is required)
                                                    select fixed conversion
                                                    scan of the channels of group 0 *
sequence = 1
vector = 0
sequNUM = 1.
ui = 40 uA = 30 uData = 7
uiTemp = 40  uiTemp = 0  uiTemp = 0
ig = (Unsigned) ib = (unsigned) uiHalf = (Sunsig)


/* values for the sampling loop */
bSpot-bitAlign = true
P1bit = 1                          * bit 0 (zero) bitwise signal for "1"  on pin 0 of port 1 *
alphaMode = false
alfa_2 = alfa
alfa2 = alfaMax
alfaMaxPass = (Unit)*alfa.m ref*0.5)
pulseCount = 1  angleNum = 1
alfaMax = (Unit)*alfa.max


/*              variable keyword assignments                */
```

```
|               /* opening bracket for forward sequence */
qTemp = (ra*xaddtag) >> 10,
rkTemp = (rk*qrMagt) >> 10,

}               /* closing bracket for forward sequence */


qtse                                    /* reverse sequence */
|               /* opening bracket for reverse switching */
qTemp = (rk*xaddtag) >> 10,
rkTemp = (ra*qrMagt) >> 10,

}               /* closing bracket for reverse switching */

temp = qTemp.               /* swap switching */
qTemp = rkTemp,             /* restore */
rkTemp = temp,                          /* order */


}               /* closing bracket for "else" - sector change? */


if (qTemp < tMin) qTemp = tMin,
if (rkTemp < tMin) rkTemp = tMin.       /* to prevent unpredictable results due to entering a timer for 0 seconds */

cOffsetTemp = (qSamp-qTemp-rkTemp)/2.                   /* compute cOffsetTemp */

if (qZeroTemp == 2)                                     /* change */
qZeroTemp = 66,                         /* the zero */
else                                                    /* switching */
qZeroTemp = 2,                          /* vector */


while (TU_TSTR1 == 332)                 /* wait until */
{                                       /* first zero sequence */
}                                       /* all waveforms are matched, + until T2interrupt is completed */

qq = qTemp
q = qTemp.
                                        /* updating the */
qk = qkTemp.                            /* parameters */
rk = rkTemp
                                        /* for the */
qZero = qZeroTemp.                      /* interrupt */
cOffset = cOffsetTemp.                  /* algorithm */


}               /* closing bracket for the Do-Forever loop */

}               /* closing bracket for main */


interrupt (TU_INUM2) void timer1Interrupt(void event)

{
(TU_T1TR = 0.   /* Stop timer T1 */


(TU_T1IS &= ~1. /* acknowledge timer 1 interrupt compare interrupt */


/* note "write .and.(IOL" the reset the first pt the CCR it not necessary
  here since at this stage we do not have to wait for other higher
  priority interrupt that can interrupt this ac pump interrupt
  service routine. The 3U0% and interrupt will be a Non Maskable Interrupt
  (NMI) and as it can always be interrupt at any time. neither the 1 bit of
  the CCR is set or not */


(TU_QRAD = qTimM                /* load new value of QRAD */

(TU_T2TR = 1.           /* start timer T2 */
}


interrupt (TU_INUM3) void timer2Interrupt(void event)

{
(TU_T2TR = 0.   /* Stop timer T2 */

(TU_T2IS &= ~1. /* acknowledge timer 2 interrupt compare interrupt */

(TU_QRAD = q.   /* load new value of QRAD */

PTOB = qq                       /* output appropriate signal */

(TU_T1TR = 2.           /* start timer T1 */
}


interrupt (TU_INUM4) void timer3Interrupt(void event)

{
(TU_T1TR = 0.   /* Stop timer T3 */

(TU_T3IS &= ~1. /* acknowledge timer 3 interrupt compare interrupt */

(TU_QRAD = qk.  /* load new value of QRAD */

PTOB = qk                       /* output appropriate signal */
```

```
counter=0;
else
counter=1;
}
*/

if (phase < 1800000000)
{
if (phase<600000000) {alphaLong = phase counterUpdate = 1;}
if (phase<1200000000) && (phase >=600000001) {alphaLong = phase 600000000, counterUpdate = 1;}
if (phase<1800000000) && (phase>=1200000001) {alphaLong = phase 1200000000, counterUpdate = 1;}
}
else
{
if (phase<2400000000) && (phase >=1800000001) {alphaLong = phase-1800000000, counterUpdate = 1;}
if (phase<3000000000) && (phase >=2400000001) {alphaLong = phase 2400000000, counterUpdate = 1;}
if (phase<3600000000) && (phase >=3000000001) {alphaLong = phase 3000000000, counterUpdate = 1;}
if (phase >=3600000001)
alphaLong = phase 3600000000, counterUpdate = 1;
if (alphaLong <= sunset
counter=0;
else
counter=1;
}
}


/* alpha1 = Angle we are first vector of the present sector * 100 000*/

if (sector != vectorUpdate)          /* Sector change ? */
{                                    /* opening bracket for outer change*  */

/*  ... ... ... Change of sector  ...                  */

sector = vectorUpdate;


/*  ... select ss and ch from the look up table ...     */

ss = (alphaLong/phase600000000);
ss=ch[ss];
ch=ch[ss];

sequence =  sequence;    /* remember a quance does not change when there is
                            a sector change */

if (sequence == 1)
{                    /* opening bracket for "if (sequence == 1)  change of sector loop */

qyTemp = (ss*qyMag) >> 16;
qxTemp = (ch*qxMag) >> 16;

switch (sector)
{                    /* opening bracket for "switch(sector)   sequence 1 */
case 1  {qyTemp=35, qxTemp=48;break;}          /* values appropriate */
case 2  {qyTemp=48, qxTemp=31 break;}          /* switching */
case 3  {qyTemp=31  qxTemp=28;break;}              /* vectors */
case 4  {qyTemp=28, qxTemp=14;break;}
case 5  {qyTemp=14, qxTemp=42;break;}
case 6  {qyTemp=42, qxTemp=35;break;}
}                    /* closing bracket for "switch(sector)   sequence 1 */

}                    /* closing bracket for "if (sequence == 1)  change of sector loop */


else                                 /* sequence = 1 */
{                    /* opening bracket for "else"  sequence not equal to 1 */

qyTemp = (ch*qyMag) >> 16;
qxTemp = (ss*qxMag) >> 16;

switch (sector)
{                    /* opening bracket for "switch(sector)   sequence not equal to 1 */

case 1  {qyTemp=48, qxTemp=35;break;}          /* Values appropriate */
case 2  {qyTemp=31, qxTemp=48;break;}          /* switching */
case 3  {qyTemp=28, qxTemp=31;break;}                              /* vectors */
case 4  {qyTemp=14  qxTemp=28;break;}
case 5  {qyTemp=42  qxTemp=14;break;}
case 6  {qyTemp=35  qxTemp=42;break;}
}                    /* closing bracket for "switch(sector)   sequence not equal to 1 */

}                    /* closing bracket for "else"  sequence not equal to 1 */

}                    /* closing bracket for sector change* */

else
{                    /*opening bracket for "else" of sector change* */


/*      select ss and ch from the look up table      */

ss = (alphaLong/phase600000000);
ch=ss+90;
ch=ch[ch];


if (sequence == 1)                   /* forward sequence */
```

```
/*------ Vectoblequnude variables ------*/

long xsblng,
float xvlsblng,
long ysblng;


/*------ twistblngTimes var  -tus -------*/

unsigned int qTemp, /*  zero, xDistFactor, zSamp.
unsigned long ta, tb.


/* -- -- xwistblngVectors variables ----*/

unsigned int xpTemp, xaTemp, xZeroTemp  temp.


/*  -  - datasistay variables -----*/

int xg, xa, xZero, tp, fx, tDist;


/* -  - phaseDirector  phaseLockLoop variables      -*/

unsigned int xTestStart, xZeroStart, xMag1, yPlus2, xPlus2, xZero2, xDistMax,
int xOut, xPastDirection, xZeroDirection, xReadOut, xZeroDist, pastDirect, zeroDirect


/* -- -- xDSuperwriter - phaseLockLoop variables - -  */

unsigned int xMagCount, fMaxFactor, fMinFactor, fZeroCrossFactor
unsigned long xFMinFactor, xFMaxFactor, xFMaximal, *xmrFactor,
int xSwitchStart, fmaxStart.


/*     - frequency sweep variables   - -*/

unsigned int fMaxADDR, fZeroADDR,
float xFStore, xFStore,
unsigned long xF1, xF2;

/* -  - voltage sensing variables - -  */

long xVCount, xYStore, xIncLong, xFactLong,
float xvfStore.

enum {false, true}.

float maxDeltaFDiv = 240;

int overland = false.


/*****************************************************************
*                                                    MAIN
*****************************************************************/

void initialDesign (void);
void phaseLockLoop (void)

{ int xpid main (void)
{                    /* opening function for main loop */

initialisations().


/*                    MAIN LOOP                    */

for ( ;;)
{                                   /* Opening bracket for do - the Forever loop */

ADCSR = 123. /* start A/D conversion   A/D conversion is stopped at the end of each A/D interrupt routine */
while (ITD  TSTR != 32);                              /* Condition to prompt wait for next conv
                                                         sequence to wait for completion of ITUinterrupt routine */
{
}

sequence = sequence                                   /* Change sequence */

counter = counter + 1                                 /* Go to */

xsblng = /xinvLoopCalblng();                          /* computing xsblng */


phaseLockLoop();

phase = MaxFactor*counter          /* angle displacement xy = this horizontal */


/* an alternative way to finding alpha1 and secondAngle */
/*
secondAngle = sqrt(a*a+b*b);
alpha1 = sqrt(a*a+b*b)*sin(alpha);
secondAngle = sqrt(a*a+b*b);
if sqrt(a*a+b*b) < 0.0000001)
{
alpha1 =  gamma;
```

Appendix J: Source Code for SDEC0408.C

```
/*                                  SDEC0408                                */

/* Program to perform the full SVM control of the inverter */

/* Sampling frequency @ 25 KHz */

/* Version 0.8 */
/* Author: Mourad Creese */
/* Date: 08/05/21 */


/* Description
   This version is similar to version 0.7, except that this time the A/D
   conversion is fully implemented. This is the first working version of the
   program where feedback of the inverter voltage and frequency, and reverse
   voltage and frequency has been implemented and integrated with the rest of
   the program
*/

/* Result
   The program compiled and ran properly on the in-circuit emulator, i.e.
   without generating any errors. However the output signals were unstable.
   Same like effect was not observed on the previous version. It can be
   postulated that this is a result of adding the A/D conversion routine
   Adding the A/D conversion can disturb the proper running of the program
   in two ways

   a. By causing small but frequent changes of the stored parameters. These
      small changes are typically only due to the high sensitivity of the
      sensing circuit or because of slight instability in the supply. This
      would suggest that an hysteresis algorithm (software or hardware
      implemented) be added

   b. By adding to the execution time of the program. It was already observed
      in version 0.6 that the program was then operating at its limit, within
      the time constraints. This would suggest that the sampling frequency be
      lowered
*/


#include <math.h>

#pragma language = extended          /* Enable the use of extended keywords */
#include <sfr320F243.h>              /* internal registers define */
#include <intr320F243.h>             /* interrupt vector addresses define */

/*            Define Constant Terms                                         */

#define pi 3.14159265
#define sClock 40.0E6                        /* At a clock frequency of 16MHz */
#define MAXSECTORDIV 240
#define fSampling 25.0                       /* At a sampling frequency of 8.25kH */
#define vReg 400
#define vSmall 0
#define vNominal 230
#define fMin 45                              /* define lower limit of frequency window */
#define fMax 55                              /* define upper limit of frequency window */
#define fNominal 50
#define vInvMin 200                          /* define inverter high */
#define vInvMax 260                          /* define inverter Low */
#define iMin 40                              /* minimum value that the switching times are allowed to take */
#define vRef 4.05                            /* A/D Reference Voltage */
#define vMagMin 0.5
#define vMagMax 0.9E6804

/*              Running Variables                                           */


/*     initialization variables      */

unsigned int theta;
float  alphaMin


/*     floating point variables      */

float alpha, alpha2nd  sinAlpha;
int i;
int SAMPLESECTORDIV;
float ka  ke  kb  kc  sectorByRad3


/*     assembly-language variables   */

int i, newSet;
unsigned long phase  phase2, alpha1, alpha2, long
unsigned int period


/*     variables in C variables      */

int sector  adjustedSector  sectorNo
```

```
/* -- HSI Interrupt Subroutine --- */

interrupt [HSI] void onboardInterrupt (void)
{

/*for (;;) /* this loop is put as a command only for testing purposes */
{
P1DR=0;          /* The only way to get out of this loop is to reset the microcontroller */
}
*/

control = true;

}
```

```
/*          PHASE DETECTION INITIALISATION                              */

vNewShort = vNewLong >> 8;
vInvShort = vInvLong >> 8;
vNet1 = vNet2;
vInv1 = vInv2;
vNet2 = vNetShort;
vInv2 = vInvShort;
vDiff = vInv2-vNet1-vNetShort;
if (vDiff < 0)
vDiff = -vDiff;

vNetDiff = vNet2-vNet1;
if (vNetDiff > 0) vNetDirection = posDirect;
else vNetDirection = negDirect;
vInvDiff = vInv2-vInv1;
if (vInvDiff > 0) vInvDirection = posDirect;
else vInvDirection = negDirect;

/* ------ IF vNetADDR OUT OF LIMITS, SET INVERTER FREQUENCY TO NOMINAL VALUE  -- */

if ((vNetADDR < vMinFactor)) ((vNetADDR > vMaxFactor)) /* Phase lock loop: Inverter Frequency ok */
{               /*open bracket for if files out of limit*/
aSwitchInhibit =true;
P5DR = 1;                       /* Send aSwitchInhibit signal on pin 0 of port 5 */
vInvFactor = vFNominal;


                /*  IF INVERTER NOT OK, TRANSFER TO RESERVE AFTER A BREAK -- */

if (((vNetADDR < vMinFactor+1) ((vNetADDR > vMaxFactor0) ((vInvShort<vInvMin)) ((vInvShort > vInvMax))
{ /* open bracket for inverter failure*/
inverterFail =true;
(TW_TEST) = 0;  /* Skip all timer routines */
while (vInvLong >> vResCount)
{
}
aSwitchInhibit =false;
P5DR = 0;                       /* Cancel aSwitchInhibit signal on pin 0 of port 5 */
test..1
{}              /* the only way of restarting the inverter after an inverter failure is by resetting the microcontroller itself */
} /* close bracket for inverter failure*/
} /* close bracket for if(Not out of limit*/


/*  --  ELSE IF vNetADDR WITHIN LIMITS, PERFORM PHASE CHECK (VOLTAGE DIFFERANCE CHECK    */

else
{ /* open bracket for if(not inverter fault */


if (vDiff > vDiffMax)

                /*  -- IF OUT OF PHASE (VOLTAGE DIFF ), PERFORM PHASE DIFF  REDUCTION   */


{ /* open bracket for vDiff > vDiffMax - out of phase*/
aSwitchInhibit =true;
P5DR = 1;                       /* Send aSwitchInhibit signal on pin 0 of port 5 */
if (vNetADDR <= vMinFactor)
vInvFactor =vFMaxFactor;
else
vInvFactor=vNetFactor;
} /* closing bracket for vDiff > vDiffMax - not in phase */


  /*    ELSE IF FIRST PHASE OK  PERFORM EXTRA PHASE CHECK (VOLTAGE DIRECTION)   */

else
{ /* open bracket for vDiff ok. */
if (vNetDirection == vInvDirection)

                /*  IF IN PHASE AFTER SECOND CHECK MAKE INVERTER FOLLOW REFERN   */

{ /* However send inverter output pos as phase */
vInvFactor = (5/4)*(vNetADDR + 512);
aSwitchInhibit =false;
P5DR = 0                        /* Cancel aSwitchInhibit signal on pin 0 of port 5 */
}

                /* ELSE IF FAILED SECOND PHASE CHECK, PERFORM PHASE DIFF  REDUCTION AGAIN    */

else
{ /* open bracket for opposite direction */
aSwitchInhibit =true;
P5DR = 1;                       /* Send aSwitchInhibit signal on pin 0 of port 5 */
if (vNetADDR <  vMinFactor)
vInvFactor =vFMaxFactor
else
vInvFactor=vNetFactor;
} /* closing bracket for opposite direction */

} /* closing brackets for vDiff ok*/
} /* closing brackets for if(Not within limit */

/* end of phase lock loop */
}
```

1 Scope

1.1 Introduction

This document lists the steps and procedures taken to verify the correctness of the software and the system in general. The software is tested on its own as well as when integrated to the hardware platform. Testing here is performed according to the strategy laid down in section 5.4 of the Literature Survey on Embedded Systems, SDE 103. Except when otherwise stated, the test results in this document refer to the latest revision of the software, version 1.0.

1.2 Purpose

The purpose of the testing activity is to determine whether the software is behaving according to the specifications. In particular the system is verified against the requirements analysis specifications and the functional specifications.

1.3 Definitions

PWM : Pulse Width Modulation
PLL : Phase-locked loop
SVM : Space Vector Modulation

1.4 Audience

The audience for this document comprise the various stakeholders of the SEAL, including:

- Full-time staff members of SEAL
- All full-time and part-time post-graduate students associated with SEAL
- Members of the SEAL Management Board
- Head of the Department, Electrical Engineering
- Individuals who will perform internal and external surveillance audits of

Change History

Configuration Control

Project:	SDES
Title:	Testing Documentation
Doc. Reference:	SDE 112
Created by:	H. Bapoo
Creation Date:	24 July 1996

Document History

Version	Date	Status	Who	Saved as:
0.01	96/07/24	Draft	H.Bapoo	\1995-13\TP\SDE112WP.001
1.00	96/08/25	Approved	H.Bapoo	\1995-13\TP\SDE112WP.100

Revision History

Version	Date	Changes
0.01	96/07/24	New Document
1.00	96/08/25	Update to revision number and status following Board Approval

Management Authorisation

Version	Date	Status	Management Board Minute Reference
1.00	96/08/26	Approved	Section 2.3 of SDE 692

Change Forecast

This document is to be revised whenever changes are made to the method of testing the product and when additional test results are available.

Appendices

Table of Contents

SDES QMS

Testing Documentation

Technical Product

Version: 1.00

Document Status: Approved

```
{                               /* Opening Bracket for the Do Forever loop */

/*
while (proceed == false)        /* Condition to proceed: wait for first A/D conversion to complete */
{}.
*/

ADCSR = 123; /* start A/D conversion.  A/D conversion is stopped at the end of each A/D interrupt routine */ .

wsMag = viewMag.vsMag.
flowFactor = bPI*(RadADC) < 172.
pVolt = flowFactor * counter.
proceed = false

}               /* closing bracket for the Do Forever loop */

}               /* closing bracket for main */.



void initialisation (void)
{

float bPIe.


/*          Runaway Variables Initialisation          */

/*      Arithmetic Values to variables      */

/* values for the A/D Conversion */
ADCSR = 0;      /* clear ADST flag */
ATCR = 0;       /* disable external trigger */
ADCSR = 123;    /* select          enable A/D interrupt
                                                        start A/D conversion. By putting this step early in the initialisation program (before the loop up table operation) we practically ensure that results of the first A/D conversion
wq available before the main cycle is started
                                                        select scan mode
                                                        select fast conversion
                                                        select all the channels of group 0 */


/*      Calculate Constant terms      */

/* wsVsMagnitude terms */
wsMagMag = (wsVsMax - wsVsMin)/(wsMagMax - wsMagMin);
vsMagMag = (wsVsMax - wsVsMin)*vsMagMag*(1/512);


/* Voltage sensing terms */
mVSense = (wsVsMax - wsVsMin)/4.0;
vVSense = (vsVsMax - mVSense*4.0)* 0.0125;                      /* 0.0125 = 2.5/200 */
LVSense = (vsVsMax*mVSense*512/1023.0);

/* Frequency sensing terms */

mFSense = (fMax - fMin)/1023.0;
vFSense = fMax - mFSense*1.0;
kP1 = (200.0*iSampling*vFsd*mFSense*127.0)/pi;
bP1 = (200.0*iSampling*2.0);
kP2 = mFSense*iSampling;

SWSCR = 1;              /* Select INT bit in CSCR as a mask for IRQ0 falling
                                edge selected  interrupt RQA enabled */

IPRA = 0;
IPRB = 0;               /* Set all interrupt priority level to zero (turn */


}


interrupt [AD  dtc=none] void ad_interrupt (void)
{

iRawADDR = ADDR0;  - 0
iRawADBR = ADDR0;  - 0
vMagSamp = (ADDR0 -  0)*LVSense + LVSense;
vImgSamp = (ADDR0 -  0)*LVSense + LVSense;
ADCSR = 0;      /* clear ADIF and ADST bits, ready for next conversion */
interrupt = true */

}
```

Appendix L: Source Code for SDEC0501.C

```
/*                                          sdec0501.c                          */
                                                    \
/* Program to test the A/D feature - Cross Mode */


/* Author  Richard Beppe */
/* Date:  10/05/98 */
/* Status  Draft */

/* Description
   This program tests the manual sensing and A/D conversion of Pots. Rev.
   vbw and vPot. It also includes the immediate calculations involved with
   each of these parameters
*/

/* Results
   Successful. Time taken to perform A/D conversion and the calculations on
   these parameters  including the acquisition of vbatRef and pbatt  equals
   approximately 40 microseconds
*/

#include <math.h>

#pragma language = extended       /* Enable the use of extended keywords */
#include <ioh16481h.h>            /* personal registers define  */
#include <ioh83923.h>             /* interrupt vector addresses define  */

/*          Define Constant Terms                */

#define tSampling 1.01.4
#define pResonant 230
#define tMax 40                   /* defines lower limit of frequency window */
#define tMin 0                    /* defines upper limit of frequency window */
#define vmaxMax 200               /* defines inverter high */
#define vminMin 200               /* defines inverter Low */
#define vRef 4.51                 /* A/D Reference Voltage */
#define voltageMax 0.8805804
#define voltageMin 0.8

/*          -- Scanning Variables          */


/*    calibration variables --   */

/* ---- vectorinterpolate variables  -- */
long calSig,
float vbtMag,
long vbtMag,


/*     frequency sensing variables     */

unsigned int tStartA1CR, tStopA1CR,
float dfSense, dfSense
unsigned long tF1, tF2,

/*     voltage sensing variables     */

long vPSense  vbtSense  vbatLong, vRestMag
float vbtSense

struct { bool, bool}
unsigned int pressed,

/*    pulOutputs  phaselockLoop variables    */

unsigned long dutyFactor

/*    main implementation variables    */

int counter
unsigned long phase




/*****************************************************************************
*                                                                            Main
*****************************************************************************/
void initialisation (void)

int main (void)
{                    /* opening bracket for main loop */

                                                                    */


/*                  Main Loop                  */

                                                                    */

}
```

```
else
{ /* opto bracket for opposite direction */
aSwitchInhibit =true;
P1OR 0 = 0;                    /* Send aSwitchInhibit signal on pin 0 of port 1 */
if (MeetQBR<= aClearanceFactor)
ThisFactor =aMidleFactor;
else
theFactor=aMaxFactor;
} /* closing bracket for opposite direction */

} /* closing bracket for rQuit ok*/
} /* closing bracket for XRec notRec test */

/* end of photo fetch loop */
}


/*     NMI Interrupt Subroutine     */

interrupt (NMI) void overtureNMIinterrupt (void)
{
overload = true
P1DR=0;              /* the only way to get out of this loop is to reset the microcontroller */
for (;;) /* this loop is put in a comment only for testing purposes */
{
}
}


/*     Receiver Off Interrupt Subroutine     */

interrupt [IRQ_3] void receiverOffInterrupt (void)
{
IEN_1BIT = 0
P1DR = 0;
PTR 3=0;             /* disable IRQ 3 interrupt */
for (;;) /* endless loop, must reset microcontroller to get out of loop */
{
}

}


/*     A/D Interrupt Subroutine     */

interrupt [AD] void A_to_D_Interrupt (void)
{

tRecADDR = ADDRA ;> 6;
tRecADDR1 = ADDRB ;> 0;
tRecADDR1 = ADDRC ;> 0;
tRecADDR1 = ADDRD ;> 0;
ADCSR &=CLR_ADF;             /* clear ADF and ADCE bits, to stop data ...conv */
ADCSR &=CLR_ADST;

}
```

```
                    *
ITU_T81R = 0,       /* start Timer T3 */

}


/*  ... Phase Lock Loop Subroutine ....*/

void phaseLockLoop(void)
{

/* ..... ..... PHASE DETECTION INITIALISATION ..... ....*/

vDiff = vInvADDR-vRmsADDR;
if (vDiff<0)
vDiff = -vDiff;

vMag1 = vMag2;
vInv1 = vInv2;
vMag2 = vMagADDR;
vInv2 = vInvADDR;

vMDiff = vMag2 - vMag1;
vIDiff = vInv2 - vInv1;

if (vIDiff >=0) vMagDirection = posDirect;
else vMagDirection = negDirect;

if (vIDiff >= 0) vInvDirection = posDirect;
else vInvDirection = negDirect;

/*  ... IF vRmsADDR OUT OF LIMITS, SET INVERTER FREQUENCY TO NOMINAL VALUE ...  */

if ((vRmsADDR<(vNomFactor*0.9)) || (vRmsADDR>(vNomFactor*1.1))) /* Phase lock loop inverter frequency */
{                   /*open bracket for if (the out of limit*/
vSwitchInhibit =true;
PSCR.0 = 0;                 /* Send vSwitchInhibit signal on pin 0 of port 5 */
DimFactor = vFMaximum;


            /*  IF INVERTER NOT OK, TRANSFER TO RESERVE AFTER A BREAK  */

if ((vInvADDR<(vNomFactor*0.9)) || (vInvADDR>(vNomFactor*1.1)) || (vInvADDR==vMaxFactor)) /*open bracket for inverter failure*/
{ /* open bracket for inverter failure*/
vInvAbort = true;
ITU_T81R = 0;  /* Stop t3 timer counting */
counter = 20;
while ((vInvADDR1<(vNomFactor*0.9)) || (vInvADDR1>(vNomFactor*1.1)))
{
ADCSR &=CLR_ADS;            /* clear ADI and ADST bits, to stop auto conversion */
ADCSR &=CLR_ADSF;
ADCSR = A2D_DEXOP; /* start A/D conversion   A/D conversion is stopped at the end of each A/D interrupt routine */
while (counter>0)
counter--;
}

vSwitchInhibit =false;
PSCR.0 = 1;                 /* Cancel vSwitchInhibit signal on pin 0 of port 5 */
test;}
{}                  /* the only way of re-starting the routine after an inverter failure is by restarting the microcontroller itself */
} /* close bracket for inverter failure*/
} /* close bracket for if(not out of limit*/


/*    ELSE IF vRmsADDR WITHIN LIMITS, PERFORM PHASE CHECK VOLTAGE DIFFERENCE CHECK    */

else
{ /* open bracket for if(in within limit */


if (vDiff  > vDiffMax)

            /*  IF OUT OF PHASE VOLTAGE DIFF, PERFORM PHASE DIFF REDUCTION   */


{ /* open bracket for vDiff > vDiffMax  out of phase*/
vSwitchInhibit =true;
PSCR.0 = 0;                 /* Send vSwitchInhibit signal on pin 0 of port 5 */
if (vRmsADDR< vNominalFactor)
DimFactor = vFMaxFactor;
else
DimFactor =vMinFactor;
} /* closing bracket for vDiff > vDiffMax  out of phase */


  /*    ELSE IF INPHASE, PERFORM EXTRA PHASE CHECK VOLTAGE DIRECTION    */

else
{ /* open bracket for vDiff ok */
if (vInvDirection = vInvDirection)

            /*    IF IN PHASE AFTER EXTRA CHECK, MAKE INVERTER FOLLOW MESMER    */

{ /* Inverter and mesmer outputs are in phase */
timeFactor = get(*vRmsADDR) * 8F2
vSwitchInhibit =false
PSCR.0 = 1;                 /* Cancel vSwitchInhibit sig. on pin 0 of port 5 */
}

            /* ELSE IF PHASE ERROR, PERFORM EXTRA PHASE DIFF CORRECTION ALGO    */
```

```
ADCSR &= CLR_ADCF;          /* clear ADCF flag */
ADCR = CLR_EXT;             /* disable external trigger */
ADCSR = ADC_SETUP;          /* select      - enable A/D interrupt
                                           start A/D conversion. By putting this step only in the initialization process (before the look up table generation) we practically assure that results of the first A/D conversion
are available before the main cycle is reached
                                           select scan mode
                                           select 2 conversion
                                           select all the channels of group B */


Increase = 1,
Factor = 3,
Counter = 1,
uj = 40, ui = 20, uZero = 0,
ujTemp = 40, uiTemp = 20, uZeroTemp = 0,
tj = (int)uj, ti = (int)ui, tDiff = (int)uk;


/* values for the phase-lock loop */
lockAlert = false,
uTest2 = vTrig,
vTest2 = vCount,
resetCount = 1, negCount = 1,
vDutMax = 0.05*(float)uTest;


/* ..... M3040 Registers Initialization */


ITU_TCR0 = 32,              /* Select CRA compare match as Clearing Source
                               Select External Clock (EGAMPer) */
ITU_TCR1 = 32,
ITU_TCR2 = 32,
ITU_TCR2 = 32,
ITU_TCRA = 32,


ITU_TIOR0 = 2,              /* Select GRA as an output compare register
                               1 as output at the TIOCA pin at GRA compare match
                               GRB compare, no output */
ITU_TIOR1 = 2,
ITU_TIOR2 = 2,
ITU_TIOR2 = 2,
ITU_TIORA = 2,


ITU_TIERO = 1,              /* Enable GRA/A interrupt to occur when it is
                               requested by DMTA */
ITU_TIER1 = 1,
ITU_TIER2 = 1,
ITU_TIER2 = 1,
ITU_TIERA = 0,              /* Disable all interrupt requests by timer T4 */

SYSCR = 11,                 /* Select 01 bit in SCR as area bit RAM Mapping
                               area selected. On chip RAM enabled */

IPRA = 0,
IPRB = 0,                   /* Set all interrupts priorities level to zero level */


P1DDR = 255,                /* All Port 1 pins set for output */

P4DDR = 1,                  /* Port 4 are for output */

P8DDR = 0                   /* Port 8 set for input */

ITU_GRA3 = 10000,           /* Set an initial delay of two before getting into
                               the main loop */

P1DR = uZero                /* Set Port 1 initially to uZero */


pSwitchAlert = true,
P8DR.0 = 0                  /* Set switch/shutdown signal to "0" - on port 8 of port 8 */

iSCR.J = 0,
SCR 3 = 1,

/*              GENERATE LOOK UP TABLE                  */

for (i=0; alpha<=8 ; alpha=MAXRES*(float)i/1000 ); i++, alpha = alpha+deltaAlpha)
{
alphaRad = pi*alpha/180
sinAlph  Rad = sin(alphaRad);
ibh[i] = (int)(0.5+(sinAlphaRad*(float)uZero+(float)uZero));
ibl[i] = (int)(0.5+(sinAlphaRad*(float)uTest+(float)uZero));
}

* calculating the full ... ing the values above loop once using values for
ujTemp, uiTemp, and tDiff ... the last two values using a MAXRES/STEPSIZE p*
of the Master phase-a- field these values only apply for a MAXRES/STEPSIZE p*
full cycle

ujTemp = 0,
uiTemp = 2048
tDiff = 2048

while(1)          * reset again after the pause interruption loop */
```

```
ITU_T0TR = 2.          /* start Timer T1 */
}


interrupt [ITU_IMIA1] void interT1Interrupt(void)

{
ITU_T0TR = 0,    /* Stop timer T1 */

ITU_T0SR &= ~1;  /* acknowledge timer 1 interrupt compare interrupt */

ITU_GRA2 = rt.   /* load new value of GRA2 */

PIDR = ok;                    /* output appropriate signal */

ITU_T0TR = 4,          /* start Timer T2 */
}


interrupt [ITU_IMIA2] void interT2Interrupt(void)

{
ITU_T0TR = 0,    /* Stop timer T2 */

ITU_T0SR &= ~1;  /* acknowledge timer 2 interrupt compare interrupt */

ITU_GRA2 = oldval;            /* load new value of GRA2 */

PIDR = ok2ms;                 /* output appropriate signal */

ITU_T0TR = 0,          /* start Timer T2 */
}


void initialisation (void)
{

/*  ---  Running Variables Initialisation  - - - - - - - */

/*  - Calculate Constant terms - - */


/* feedUpTable terms */
ka = 91/p*mi,
kb = 8*sqrt2/pi*pi,
ka1 = ka*tSampling*tClock,
kb5 = kb*tSampling*tClock,
sqrt2By8Pi = 1/sqrt(2*8),


/* vectorMagnitude terms */
minMag = (vInMax-vInMin)*(vInMax-vInMin)*vInMagMax1,
vInMag = (long)(vInMax-mVInMag*vInMagMin)*512,          /* 512 = 2^9 */


/* Voltage sensing terms */
mVSense = (unsigned short)(4.8 * 0.9),
vVSense = (vInMax - mVSense*0.5)* 512,                  /* 512 = 2^9 */
vVSense = vVal*mVSense*512/1023,


/* frequency sensing terms */
mFSense = (1/(2 * tMin))*(4.8 * 0.9),
vFSense = (1/(2 * mFSense))*4.5,
kF1 = 360*tSampling*vRef*mFSense*1E3/1023,
kF2 = 360*tSampling*vFSense*1E3,


/* pllSupervisor  phaseLockLoop  terms */                                  * 1023 = (2^10  1) * 10 bit resolution A/D converter */

vMaxFactor = (long)0.6*1023/vRef,
vMinFactor = (long)4.6*1023/vRef,

(MaxFactor - MinFactor)/(vMax - vMin),
(MaxFactor - (long)(4.6*1023/vRef)),

kFMaxFactor = (kF2*1023*4.6/vRef)+kF2,
kFMinFactor = (kF1*1023*0.6/vRef)+kF2,
kFNominal = (kF1*1023*2.6/vRef)+kF2,

vMaxCount1 = (long)(4.6*1023/vRef),
vMaxCount2 = (long)(0.6*1023/vRef),


/* Miscellaneous terms */
cmMax = (count/tSampling)*(4.8/tClock),

tSamp = (long)(tSampling/tClock),

sqrt8Min = (unsigned)(0.707*sqrt8Max),
sqrt8Mid = sqrt8Min + (sqrt8Max - sqrt8Min)/2,

gamma = (3.0*(float)(sqrt8Max)*(tSampling)*1000/1000)/2


/*  Attribute values to variables  */

* values for the A/D Conversion */
```

```
else
{               /*opening bracket for "else" of motor change? */


/* ..... power on and to check the hand-up table      */


x= (right_rear|left_rear|left_front|right_front);
to = (x[0]);
th=(x[1]);


if (sequence == 1)              /* forward sequence */
{               /* opening bracket for forward sequence */
tyTemp = (tx*multing1) >> 16;
thTemp = (th*multing1) >> 16;

}               /* closing bracket for the word sequence */

else                                    /* reverse sequence */
{               /* opening bracket for rever'n switching */
tyTemp = (ty*multing1) >> 16;
thTemp = (th*multing1) >> 16;

}               /* closing bracket for reverse switching */

temp = tyTemp;                  /* swap switching */
tyTemp = thTemp;                        /* motion */
thTemp = temp                           /* order */


}               /* closing bracket for "else"  motor change? */


if (tyTemp <= (-Max)) tyTemp=-Max;
if (thTemp <= (Max)) thTemp=+Max,       /* to prevent unpredictable results due to starting a timer for 0 seconds */

tOffsetTemp = (tStore-(tyTemp >> 1));                   /* compute tOffsetTemp */

if (tZeroTemp == 2)                                     /* change */
tStoreTemp = 65.                        /* the zero */
else                                                    /* switching */
tZeroTemp = 2.                          /* motor */


while (TIM_TSR1 == 2321                 /* word omit */
{                                                       /* last zero sequence */
}                                                       /* all waveforms is reached, i.e until T2interrupt is completed */

ty = tyTemp;
th = thTemp;
                                                        /* updating the */
tx = txTemp;                            /* parameters */
tx = txTemp;
                                                        /* for the */
tZero = tZeroTemp                       /* interrupt */
tOffset = tOffsetTemp                   /* algorithm */


}               /* closing bracket for the Do Forever loop */

}               /* closing bracket for main */


interrupt [TU_ 0x1A3] void timerT2interruptRoutine (void)

{
TIM_TSR1 = 0.   /* flags timer T2 */

TIM_TSR2 &=     1  /* acknowledge timer 2 interrupt compare interrupt */


/* Note: "write one (0)" to reset the flag of the CSR's is not necessary
   here since at this stage we do not have in mind any other higher
   priority interrupt that can interrupt this on going interrupt
   service routine. The 20MHz ref interrupt will be a non-maskable interrupt
   (NMI) and so it can cause an interrupt at any time, whether the flag of
   the CSR is set or not  */


TIM_tTIAD = tOffset;            /* load new value of tTIAD */

TIM_TSTA = 1            /* start Timer T2 */
}


interrupt [TIM_TSR1 & (1 << 2)] void timerT2interruptRoutine (void)

{
TIM_TSTA = 0;           /* stop timer T2 */

TIM_TSR1 &=     1;      /* acknowledge timer T2 interrupt compare interrupt */

TIM_tTIAD = tx;         /* load new value of tTIAD */

PTIM = tx;                      /* output square-wave signal */
```

The components on the board were verified against the bill of materials listed in appendix A.

Table 7: Component check

Have the right components been used	Yes
Is the clearance between the components OK	No, the clearance between IC27 and C60 and between IC26 and C57 is too small
Are the components in the right orientation (where it matters)	No , the base and the emitter of all the transistors are in the wrong position. The problem lies in the file specification of the component in the schematic-pcb program.

In addition, further verification of the tracks and the components revealed the following discrepancies:

- The pins of DL1 should be swapped and the logic on pin 53 of the microcontroller should be reversed. This is because port 5 can only provide current sinking, i.e. it can only switch on an LED when configured as an input pin.

- IC6, the hex non-inverting buffer is supplied by the wrong power supply, 12V instead of 5V. For the purpose of the prototype this can be solved by severing the tracks and making the proper connections with a piece of link wire.

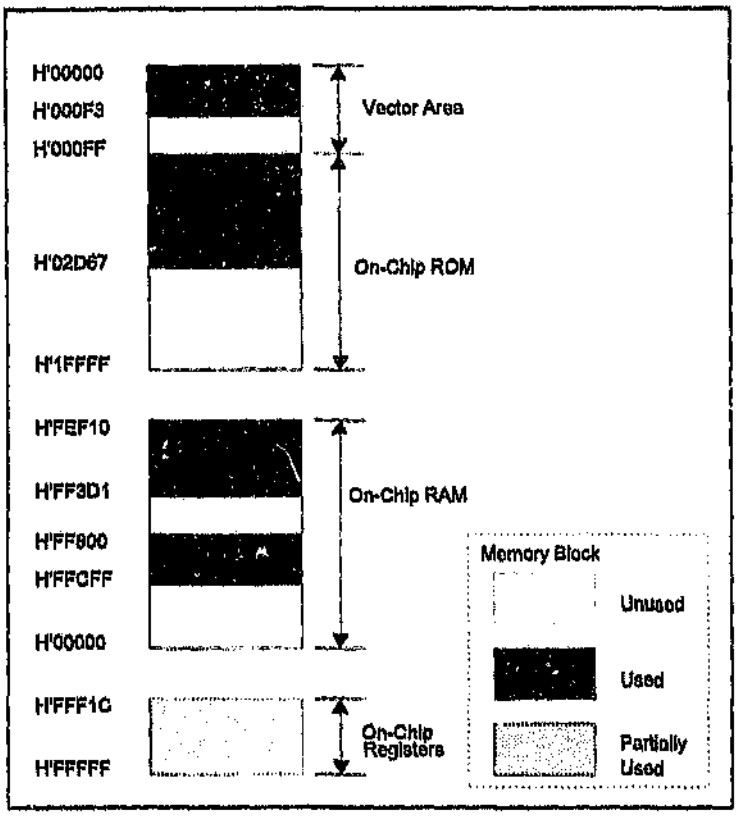


Figure 4: Memory Map for SDEC0410.C

Further details of the compilation and linking results can be obtained from the generated map file, sdec0410.map.

4.2 Hardware testing

Static analysis as applied to the hardware is limited to a basic check on the component and the track layout. This is typically performed with a meter and an external DC power supply unit. Table 6 shows the result of a power supply check. Table 7 shows the result of a component-check on the board.

Table 6: Power supply check

+12VA (+/-1V) power supply OK	✓
+12VB (+/-1V) power supply OK	✓
+5VA (+/-0.5V) power supply OK	✓
+5VB (+/-0.5V) power supply OK	✓

4 Static Analysis

4.1 Software Testing

Static analysis of the software is performed by the compiler. Accordingly the assessment below is the result of the compilation activity.

The specifications of the compiler are as follows:

Micro Series H8/300 C Compiler V 3.11 A/386
Front End V 1.14 A
Global Optimiser V 1.01 D

The result of the compilation is shown in table 4 below.

Table 4: Compilation Results

Errors	none
Warnings	none
Code Size	5180
Constant Size	252
Static Variable Size	data: 1218 IRam: 0

All the sizes are in bytes. This shows that the microcontroller is operated well within its space limitations.

The linker used was the Micro Series Universal Linker V 4.46 L/DXT. Table 5 shows the result of the linking activity. Figure 4 depicts the memory map of the microcontroller with the software downloaded. It shows that all the code and data segments are well contained within the memory boundaries of the processor.

Table 5: Linking Results

Errors	none
Warnings	none

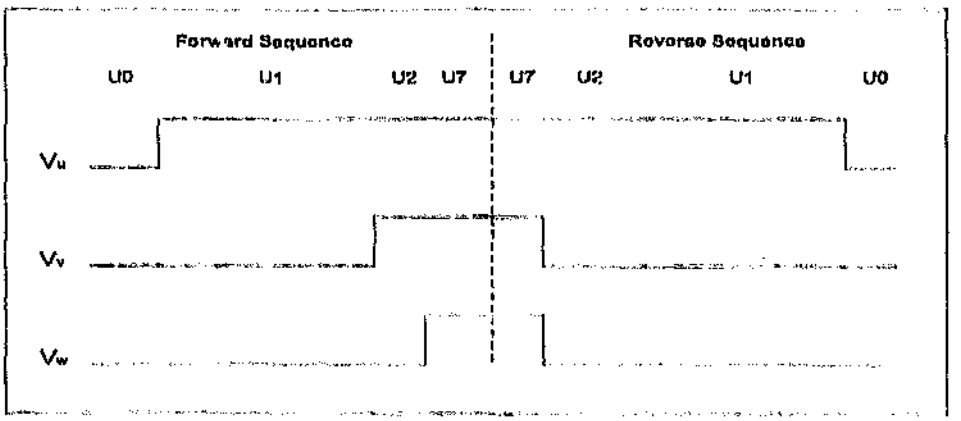


Figure 3(a): Typical Pulse Train in Sector 1 as Derived From Figure 2

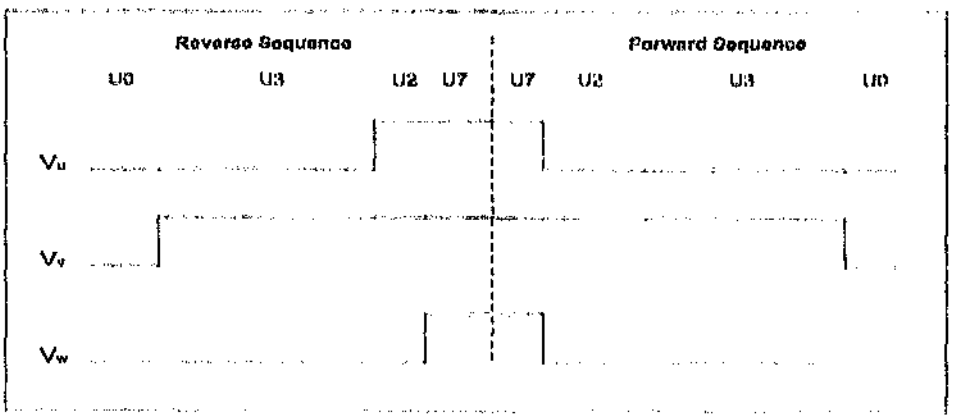


Figure 3(b): Typical Pulse Train in sector 2 as Derived from Figure 2

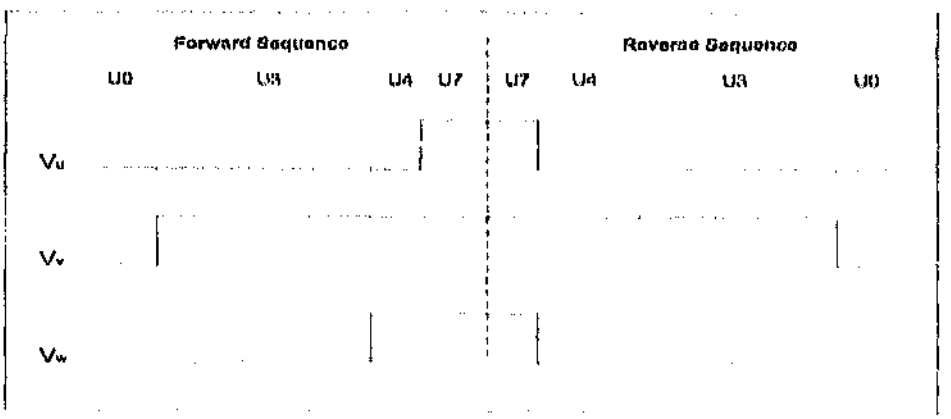


Figure 3(c): Typical Pulse Train in Sector 3 as Derived From Figure 2

Table 3: Checklist to verify the correctness of the algorithm in synthesising SVM

Does the output start on a forward sequence	✓
Does the output follow a Gray code pattern	✓
Is the Gray code pattern maintained on a sector change	✓
Does the switching time values vary according to the SVM equations	✓
Are the variables reset appropriately on a change of revolution	✓

Another perspective of the same pattern flow, as obtained when going through the software algorithm, is projected in figure 3. Figure 3 depicts the voltage levels in each of the three legs of the inverter, taken with respect to the negative DC bus. It captures the general pattern in the first three sectors. From this picture the PWM effect of applying SVM is clearly observed. This exercise serves to further validate the software algorithm. It is also worth noting that the pattern shown in figure 3 should actually correspond to the physical drive pulses applied to the IGBTs. Checking if the actual drive pulses correspond to the expected PWM train of pulses will constitute an important precautionary step before applying DC to the bus when performing dynamic tests.

- In the first sampling interval the pattern at the output is not consistent with SVM. This is as expected since this first interval is only used to configure the variables properly (see section 4.3, SDE 107).

- The second sampling interval, which should be the first real SVM output, produces a forward sequence output in accordance with the specifications of SVM.

- The switching vectors appear in the right order, with uZero remaining unchanged when moving to the next sampling interval.

- The sequence toggles after each sampling interval.

- The value of ta decreases while that of tb increases until the middle of a sector is reached, at which stage the trend reverses.

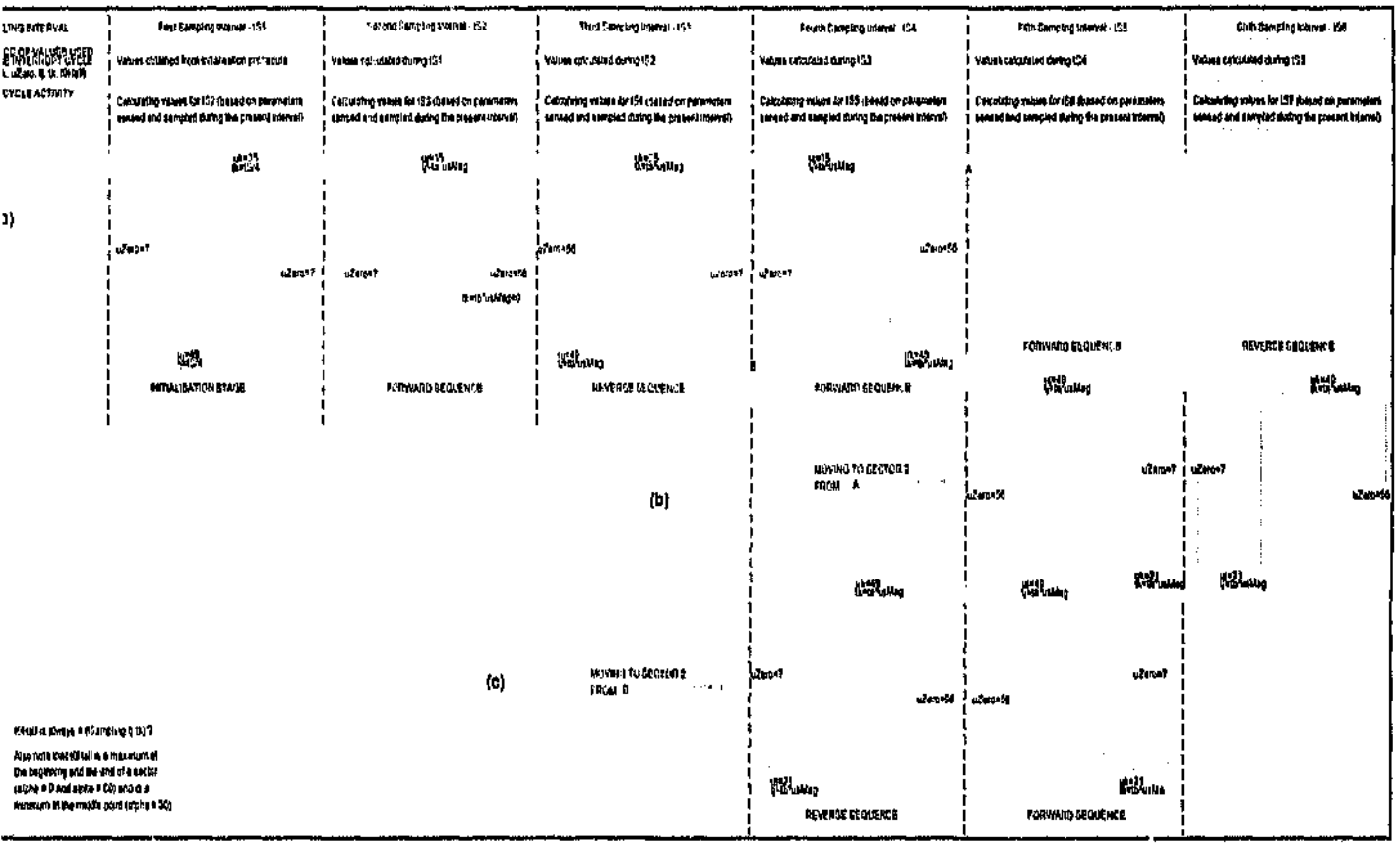
Figures 3 (b) and (c) investigate the behaviour of the algorithm under a change of sector. The following points are observed:

- The sequence remains unchanged on a sector transition

- The sequence toggles again after the first sampling interval in the new sector and the same variation as observed in (a) is noted, except that the initialisation stage is not repeated.

In addition, by following the sequence of events (from the flow chart or the state diagram) it is easy to verify that the counter is reset on a change of revolution. As the counter is reset, so will phota, on the next sampling interval.

It is left to the reader, as an exercise, to verify the above observations. Following this exercise we can fill in a checklist to confirm the correctness of the algorithm in synthesising SVM. The checklist is shown below in table 3 and is derived directly from the functional specifications of SVM as laid in document SDE 106.


First Sampling Interval - t51
Second Sampling Interval - t52
Third Sampling Interval - t53
Fourth Sampling Interval - t54
Fifth Sampling Interval - t55
Sixth Sampling Interval - t56
Values obtained from calibration procedure
Values calculated during t51
Values calculated during t52
Values calculated during t53
Values calculated during t54
Values calculated during t55
Calculating values for t52 (based on parameters sensed and sampled during the present interval)
Calculating values for t53 (based on parameters sensed and sampled during the present interval)
Calculating values for t54 (based on parameters sensed and sampled during the present interval)
Calculating values for t55 (based on parameters sensed and sampled during the present interval)
Calculating values for t56 (based on parameters sensed and sampled during the present interval)
Calculating values for t57 (based on parameters sensed and sampled during the present interval)
INITIALISATION STAGE
FORWARD SEQUENCE
REVERSE SEQUENCE
FORWARD SEQUENCE
REVERSE SEQUENCE
FORWARD SEQUENCE
REVERSE SEQUENCE
MOVING TO SECTOR 2 FROM A
MOVING TO SECTOR 2 FROM B
(b)
(c)

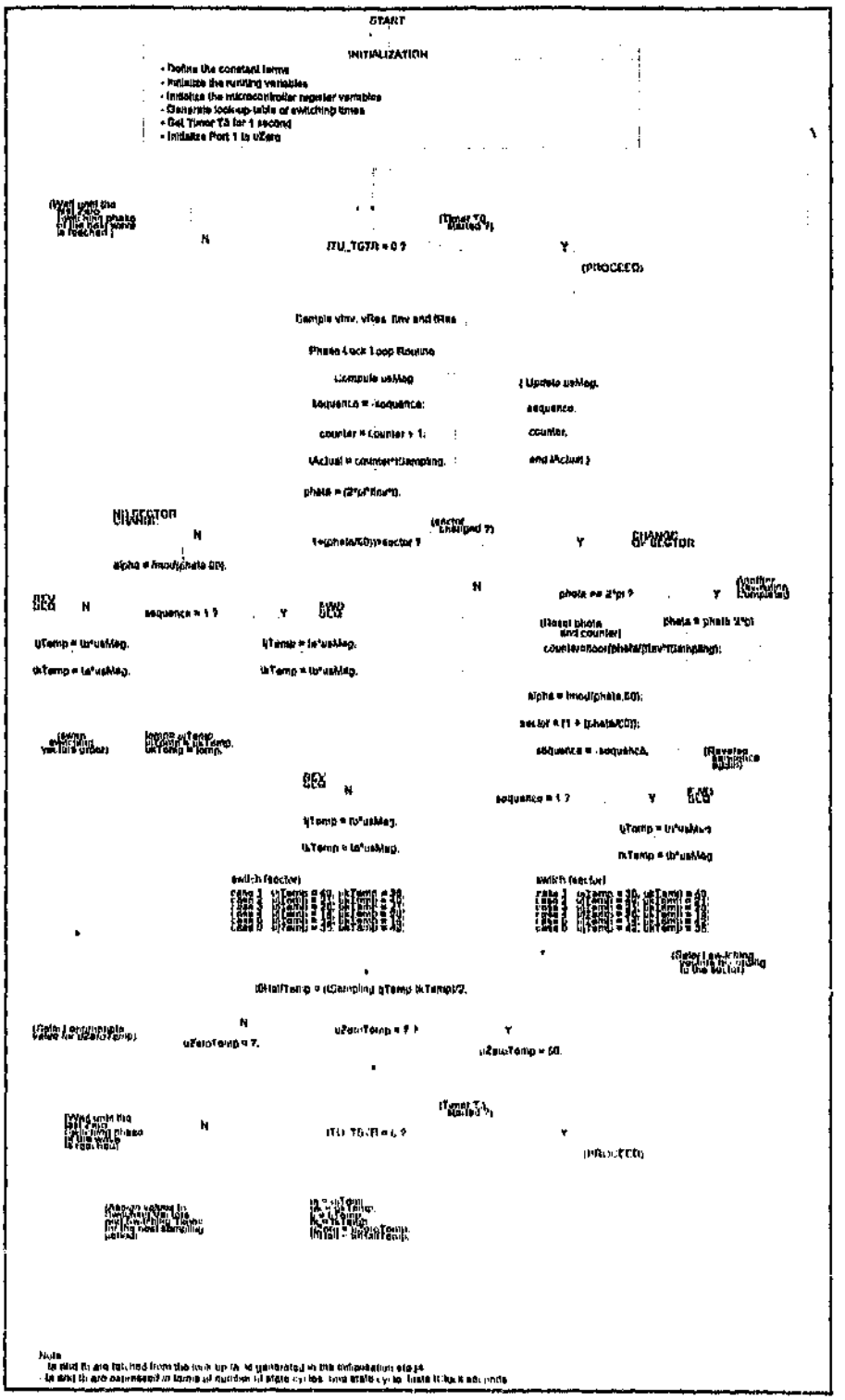


Figure 1: Flow Chart Diagram of the *main cycle*

3 Symbolic Evaluation

Like the code inspection and walkthrough testing, this test involves analysing the program at a purely theoretical level. In the present context this test is used to establish the suitability of the chosen algorithm at synthesising an SVM pattern. Although this test is not concerned about timing measurements, it is possible and quite useful to make general observations about the synchronisation and the time-frame of the events.

The overall state transitic . diagram (see section 3.3, SDE 108) or the flow chart diagram of the program (reproduced in figure 1.0 below) can be used to verify the correctness of the algorithm. This test is tedious but provides deep insight into the mechanism of the algorithm. It is particularly helpful in early versions of the software, prior to the integration stage. Once the viability of the algorithm has been established with this test, the test can be omitted for higher versions of the software to reduce the time spent on testing. Typically, on higher versions, testing is limited to the static and dynamic types. In the present case symbolic evaluation tests were performed up to version 0.6. The rest of this section exposes the result of the symbolic evaluation as performed on version 0.6. Note that the only major difference between version 0.6 and 1.0 is that in version 1.0 the sensed parameters are physically sampled, whereas version 0.6 only uses dummy values for those parameters. Therefore, the flow diagram in figure 1.0 which defines version 1.0, could also be used for version 0.6 for the purpose of the symbolic evaluation test.

By going through the different transitions of the algorithm as laid in figure 1, the pattern in figure 2 is obtained. In figure 2 successive switching states are represented by voltage levels just to depict the transitional change. This should not be confused with the logical level of the output pins. uZero, uj and uk are all switching vectors representing the state of six different switches. Each of those switches can take two logical levels, on or off (0V or +5V). To project a simplified view of the actual situation, figure 2 displays uZero as a zero voltage level while uj and uk are shown as voltage levels that swing between positive and negative values. The different polarity is used to indicate the difference in state and also to capture the sequence movement. The convention used here is that a high level followed by a low level represents a *forward sequence*, while a low level followed by a high level represents a *reverse sequence*.

From figure 2 (a) the following observations can be drawn:

2 Code Inspection and Walkthrough

2.1 Software Inspection

This exercise constitutes the weakest level of testing (see section 5.4.1, SDE 103). In terms of the software it is limited to the programming style at a general level. The checklist in table 1 was applied for this level of software testing.

Table 1: Checklist for testing the general programming style

Is there consistency in the naming convention	✓
Is there a consistent indentation	✓
Is the language and its structures used effectively	✓
Is there efficient use of comments	✓

2.2 Hardware Inspection

Hardware inspection at this stage is limited to the card on which the microcontroller is mounted.

A basic inspection test of the inverter control card follows in table 2 below.

Table 2: Basic Inspection of the Inverter Control Card

General appearance of the card	Good
Does the card fit into the card cage	Yes
Are the pads and drill holes sizes right	Yes
Is the silkscreen right	No - Some of the component legends have not been printed and some others overlap
Are the track sizes right	Yes

The artwork files were amended to make the necessary corrections to the silkscreen.

b. SDE 104, Requirements Analysis
c. SDE 106, Functional Specifications
d. SDE 110, Coding Documentation

1.6 Assumptions

It is assumed that the reader is familiar to the requirements and functional specifications of the project.

1.7 ISO 9001 Requirements Traceability

- ISO 9001 Clause 4.4.7 - Design Verification
- ISO 9001 Clause 4.4.8 - Design Validation
- ISO 9001 Clause 4.10 - Inspection and Testing

the SEAL Quality Management System

- Engineering Manager, Meissner
- Product Developer
- Product Manager

1.5 Applicable Documents

1.5.1 Specifications

None

1.5.2 Standards

a. SEAL QMS Document Creation Template, QS 002, Revision 0.01, 5 April 1994

b. SEAL QMS Document Layout, Presentation and Typesetting Guide, QS 003, Revision 0.01, 1 April 1994

1.5.3 Procedures

None

1.5.4 Guidelines

[1] IGBT User Guide, Toshiba Electronics Limited, Version 1.0, Ref. No. X35/03, June 1992

[2] Diana G., :"Query about SVM Implementation - e-mail Message", SDE 586, 1 July 1996

[3] Fukuda S., Iwaji Y., Hasegawa H., "PWM Technique for Inverter with Sinusoidal Output Current", IEEE Transactions on Power Electronics, Vol. 5, No.1, January 1990

[4] Super-H, SH 7700 Hitachi Series Microcontrollers, Semiconductor and IC Division, Hitachi Ltd., Japan, 1995

1.5.5 Other Documents

a. SDE 103, Literature Survey on Embedded Systems

In order to verify the phase-locked-loop algorithm, changes in the Reserve frequency were simulated by varying the potentiometer fRes. Figure H-15 and H-16 pictures the filtered output at limiting values of fRes. We can make the following observations:

- The inverter frequency follows the fRes command within the frequency window Figures H-15 pictures the higher end (49.5Hz) and figure H-16 pictures the lower end (46.7Hz). Compare those results with the waveform in figure H-6 (48 Hz) which is taken at nominal frequency. Although (because of the low-order harmonics present) the observed frequency limits do not correspond exactly to the command values of 51Hz and 49 Hz respectively, the trend in the frequency adjustment is clearly observed.

- Throughout the adjustment of fRes the magnitude of the waveform remains unaffected.

- As fRes is taken outside the frequency window, the static switch inhibit LED comes on and the inverter waveform jumps back to that in figure H-6, which is taken at nominal values of frequency and voltage. Tracking is restored and the static switch inhibit LED is cancelled as fRes is brought back within the acceptable frequency range.

6.3.4 Magnitude Adjustment Test

The voltage feedback loop was tested similarly to the frequency feedback loop., except that this time vInv was adjusted while all other parameters were kept at their nominal values. Figures H-17 and H-18 pictures the filtered waveform at limiting values of vInv. We can make the following observations:

- The magnitude of the waveform varies inversely as the voltage simulated by vInv. At the low-end of vInv the observed magnitude is 11.20V while at the high end of vInv the observed magnitude is 6.88V. Compare those values to the magnitude of 9.36V, taken at nominal conditions in figure H-6.

- When vInv is driven out of the above limits, the filtered waveform disappears completely, indicating that the software has recognised an inverter failure condition and has cancelled all the driver signals.

- The frequency of the filtered output remains unchanged throughout the test.

6.3.2 Three-Phase Output

The waveforms as observed from the filtered output of the drive circuitry are shown in figures H-4, H-5, H-6, H-7 and H-8. FFT plots of the waveforms are shown in figures H-9, H-10 and H-11. The following remarks are applicable:

- The fundamental sinusoidal content is clearly observed in all the time plots. The exact frequency is shown to be 48 Hz instead of 50 Hz. This is only a result of the rudimentary filtering used. On average the frequency is observed to be much closer to 50 Hz.

- Figure H-4 and H-5 show the three-phase voltages as being displaced by 120^0 as in characteristic in a three-phase system.

- Changing the nominal frequency parameter fNominal to 60 Hz causes a corresponding change in the filtered output of the pulses as shown in figure H-7.

- The waveforms were all obtained from the bottom switches of the bridge. A check on the ordering of the waveforms allows us to confirm the correctness of the software and the interface circuitry in delivering the waveforms in the right order at the right gate inputs. This is further confirmed in figure H-8 where drive pulses for IGBTs in the same leg are seen to be displaced by 180^0 as would be expected.

- The same general trend as observed in the FFT of the unfiltered pulses is observed in the case of the filtered output. In addition, in figure H-9 the fundamental and the third harmonic are clearly depicted. The dramatic change in the magnitude of the higher-order harmonics, as observed in figure H-11, demonstrates the considerable improvement that a simple low-pass filter can bring.

- The effect of higher switching frequency can be observed in figures H-12, H-13 and H-14, which corresponds to version 0.6 of the software. Revision 0.6 operates at 3.25 kHz, instead of 4 kHz, but does not support any feedback. The waveforms in this case are observed to be a lot smoother, with smaller high-order harmonics, for the same filtering arrangement.

6.3.3 Frequency Adjustment Test

high frequency contents that obscure the fundamental component. Although still containing low-order harmonics, which explains the distortion of the filtered waveforms, those filtered pulses are centred more on the fundamental component than on the switching frequency component, and hence are more adequate for testing some of the control features. They are not to be confused though with the output of the inverter itself, which is conventionally subjected to a more sophisticated filter system that yields a much smoother output.

6.3 Analysing the Results

In this section the output results of the test, as compiled in appendix H, are analysed and compared with the expected results. Unless otherwise stated all the results in appendix refer to version 1.0 of the software which operates at a switching frequency of 4 kHz. The results below are either taken at the microcontroller output pins or across the gate (G) and the source (S) of the IGBT switches (SW).

6.3.1 The PWM Drives

The pulses as observed straight from the output of the microcontroller are shown in figure H-1 and H-2. Figure H-3 provides an FFT plot of the waveform. We can make the following observations:

- The PWM pattern is clearly apparent in the first two pictures.
- The transitions between the supply voltage (5V) and ground are sharp as required for IGBT switching.
- Figure H-3 reveals a high 50 Hz content and harmonics contents at multiple values of half of the switching frequency and its side-bands

Usually the harmonics for a PWM waveform occur at integral multiples of the switching frequency. The presence of intermediate harmonics is one of the characteristics of SVM. It can be explained by the nature of SVM itself. SVM is not a fixed switching pattern method and is not synchronous, i.e. the carrier and the fundamental component are not at a fixed or predetermined ratio [2]. As a result SVM tends to generate the sub-harmonics observed in figure H-3. In the literature [3], SVM is often referred to as a sub-harmonic modulation method (SHM).

fRes: Reserve frequency
fInv: Inverter output frequency
vRes: Reserve instantaneous voltage
vInv: Inverter output instantaneous voltage

Each of the above parameter can be adjusted between ground (0V) and the supply voltage (5V). In the treatment that follows, for frequency monitoring 0.5V corresponds to 49Hz and 4.5V corresponds to 51Hz, and for voltage monitoring 0.5V corresponds to 207V and 4.5V corresponds to 253V.

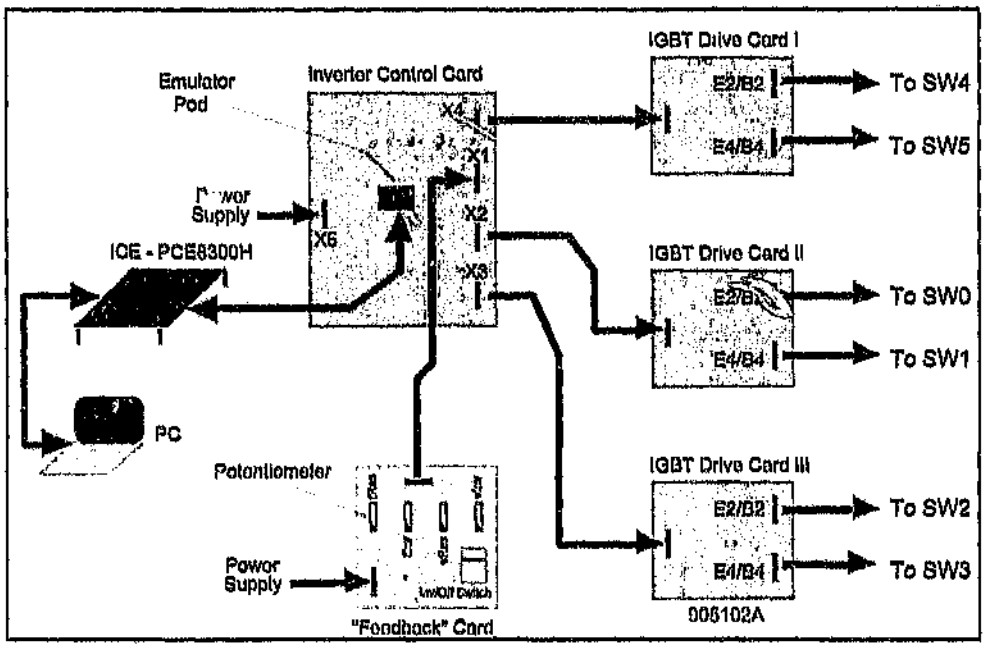


Figure 5: Set-up for the dynamic test (with the in-circuit emulator)

The pulses are generated by the microcontroller at port 1, pins 0 to 5. Those pulses are fed to the IGBTs via the drive circuitry. The separate drive circuitry provides isolation between the control circuitry and the high power on the inverter bridge. It also converts the 0V/+5V pulses from the microcontroller card to standard -12V/+12V pulses required to switch the IGBTs [1]. The pulses generated by the microcontroller are rich in harmonics, a characteristic of PWM pulses. In order to facilitate the examination, some of the tests were performed on filtered forms of the pulses generated by the drive cards. Those waveforms are referred to as filtered waveforms. The filter in this case was a simple first-order low-pass filter (corner frequency: 213 Hz) aimed only at removing some of the

6 Dynamic Analysis - Live Testing

6.1 Introduction

In this part of the dynamic test, 'live' data is applied to the program and the software system is subjected to stress tests and environmental tests. All the sensing are physically implemented and no dummy values are used. In order to reduce the complexity and the cost of the testing exercise, the tests were not performed at high DC voltage. This does not affect the verification and validation of the software which is uniquely concerned with the generation and control of PWM pulses for a three-phase inverter system. The present exercise aims at making sure that the designed system (software and hardware) is fully fitted for its purpose. Generation of high voltages and the design of the UPS system fall into the power electronics province and are left as future development exercises (see SDE 104, section 2).

Preceding the final version 1.0 is version 0.8. Revision 0.8 is an intermediate revision between version 0.7 and version 1.0. It is a complete version of the software, and runs at 6.25kHz, with feedback. Experiment shows that the output of this version was unstable. This prompted a reduction in the sampling frequency and the implementation of a hysteresis algorithm to reduce unwanted fluctuations in the sensed parameters values. Version 1.0 which operates at a sampling frequency of 4kHz, runs satisfactorily and satisfies all the requirements imposed on this project. It is the first approved version and it constitutes the final revision for the software of this case study. The rest of this document lists the results of the dynamic tests performed on this revision of the software.

6.2 Testing Procedure

The tests are performed with the in-circuit emulator connected to the inverter control card, itself connected to the IGBT drive cards. For the purpose of this test, feedback of the frequency and voltage parameters are simulated by a potentiometer network, fed by the analogue supply voltage of the microcontroller. Variation in the parameter values are then simulated by adjustment of the potentiometers. The set-up is shown in figure 5 below. Relate this diagram to the schematic of the inverter bridge in figure 2 of SDE 104. Referring to the legends on the "feedback" card, each of the potentiometer, labelled fRes, fInv, vRes and vInv respectively, simulate a different parameter as follows:

Following the decision to have transitory states and the time delay to cancel the effect of the turn off time (tOff) of the IGBTs, to be implemented by the hardware and not by the software, this version operates without the transitory states. In addition, an NMI interrupt routine was included to cater for the case of a 300% overload and provision was made to send a signal on pin 0 of port 5 under a Static Switch Inhibit condition. Otherwise it is exactly the same as version 0.5. A routine check was performed on it by letting it run on the simulator. No abnormal behaviour was observed.

This version demonstrates unquestionably the ability of the software to implement SVM. When later tested on the ICE and the controller card, the physical PWM output pulses of the microcontroller could be clearly observed (see section 6). This last test serves to confirm the ability of SVM in producing PWM output patterns.

5.9 Testing SDEC0407.C

This version is similar to version 0.6, but with the necessary modifications to use data that will be sensed by the A/D converter. Notice that A/D conversion is not yet performed here, only the different variables are scaled accordingly, so that when the A/D conversion is to be performed, then the values delivered by the A/D interrupt routine is properly recognised and accepted by the rest of the program. Those routines or areas of the program affected by the changes are:

a. The look-up-Table
b. The calculation of usMag (vectorMagnitude)
c. The phase-lock-loop

As before, in place of the sensed parameters, inverter voltage and frequency, reserve voltage and frequency, dummy values are used for testing purposes. The values that are eventually output, i.e. the SVM pattern itself remains unaffected.

The program runs successfully, suggesting that the A/D conversion can now be implemented. At this stage, enough insight and confidence has been amassed in the system. The higher versions of the software can now be tested on the ICE connected to the target hardware. The tedious exercise of gathering variables values at breakpoints is not pertinent anymore. The ability of the program to generate PWM drive pulses has been established. Any irregularities that can occur in the higher versions can be monitored directly from the physical output of the circuitry with logic probes, LEDs, oscilloscopes, etc.

observations are made:

- The timing values are correctly adjusted according to this new value of usMag

- At extreme values of alpha, i.e. when the desired vector Us* is close to a sector boundary and when more time is spent in the uZero state, the last timer interrupt (T2) occurs before the program has reached the second checkpoint in the main cycle. This can easily be verified from the table in appendix F, where it can be observed that tj is not yet updated on three occasions - recall that the tj assignment has been shifted before the second checkpoint.

- The previous observation does not affect the switching pattern since within the last timing interval all the remaining instructions in the main cycle are executed, at all values of alpha. This can easily be proved from a verification of the actual switching parameter values. Although the algorithm still passes the test at 6.25 kHz, the results show that the software is performing at its upper limit, in terms of the switching frequency.

- The time elapsed before the first breakpoint is now different from that observed in the previous test, although its position has not changed and the sampling frequency and the value of maxSectorDiv - which are parameters that might affect the calculations in the initialisation stage - have not changed. This again would tend to point at the lack of strict accuracy of the timing of the in-circuit emulator.

Version 0.5 was further tested at a switching frequency of 7142Hz. The results are tabulated in appendix G.

As expected, under this condition, the software fails to produce the right output pattern. First of all the last timer interrupt (T2) catches the program even earlier in the main cycle, before tjTemp is computed. Secondly, unlike the previous version when it was run at usMag equals 0.5, the last time interval is not sufficient for the microcontroller to cover the remaining instructions left in the main cycle. That is, the next sampling interval starts without the calculations of the previous sampling interval being completed. Synchronisation is lost and the software fails to synthesise an SVM pattern at the output port.

5.8 Testing SDEC0406.C

operations. maxSectorDiv was increased from 60 to 240, thus providing a higher accuracy in the timing values. At this new value of maxSectorDiv, generation of the look-up-table takes about 830 ms. Like version 0.3 it also operates at 5 kHz.

This present version serves to demonstrate that further improvement in accuracy and the sampling frequency are still possible.

5.7 Testing SDEC0405.C

Following the encouraging results in version 0.4, the sampling frequency was increased to 6.25kHz. Further processing activities were added to version 0.5. Those include the transitory states to ensure a full Gray code - see section 4.8.3, SDE106 - and the on-line computation of usMag and vInv. In addition this version includes the full phase-locked-loop algorithm. Appendix E shows the simulation results at usMag set to usMagMax (0.866604).

The above exercise found no irregularities in the behaviour of version 0.5. The sequence is updated properly and sector changes and changes of revolution are handled according to specifications. No timing problems are observed. In addition we note that:

- uj and uk have the same values. This is because, in order to make room for the assignments of the transitory states, some of the assignments performed in the last section of the main cycle were shifted just before the second checkpoint. Among the assignments shifted up, is the instruction "uj = ujTemp ". This should not affect the timing pattern since by the time this assignment instruction occurs, timer interrupt T0, which uses uj, has long since occurred and been completed. The other assignments that occur before the checkpoint are that of uzjTans and tj.

- The switching time values are correct. This can easily be verified from a look-up-table, similar to the one in appendix D, but which is generated for a sampling frequency of 6.25kHz and a value of maxSectorDiv of 240.

- The in-circuit emulator again shows a time offset of about 12 µs.

As a further test on this version, the software is run at usMag set to usMagMin (0.5). This value of usMag corresponds to the maximum allowable output voltage, when usMag is adjusted to a minimum, forcing the output port to spend more time into the state uZero. The results are displayed in appendix F. The following

In an attempt to bring down the processing time, float manipulations were avoided and the use of the #define macro directives was reduced to simple value assignments (i.e. all expression assignments were removed), giving the third version of the program.

This version operates at a switching frequency of 5kHz, with a value of maxSectorUpdate of 60. Like its predecessor it was run with breakpoints (BP's) applied at regular intervals. The table in appendix C shows the result of the simulation test at different sampling intervals.

The following observations can be made:

- The sequence is toggled after each sampling interval, except at a sector change where it remains unchanged.
- Values of ta and tb are chosen properly. This can easily be verified from the contents of the look-up-table for SDEC0403.C, listed in appendix D.
- The counter is reset on a change of revolution.
- No timing problems are observed, i.e. all the processing are performed within the expected and allocated time intervals.
- The time taken to determine alpha and sectorUpdate has been considerably reduced. It is now about 20µs. Similar reduction in execution time were observed in computation where float manipulations were removed. As a further example, the computation of tActual now only takes about 3 µs (as compared to 58µs before)
- Computation of phata takes about 59µs.
- On average the time measured between the sampling intervals is 212µs, instead of 200µs as would be expected at a sampling frequency of 5 kHz. It seems that the emulator has an offset of 10µs in its timing. In-circuit emulators inaccuracies are treated in section 9 of document SDE 103.

5.6 Testing SDEC0404.C

This program version is very similar to the previous version, version 0.3. The only difference occurs in the value of maxSectorDiv and in further elimination of float

points where the values are read and the timing checked.

Breakpoints are included both at each sampling interval and within a sampling interval. The results in appendix B were taken at the beginning of different timer interrupts within the first sampling interval, i.e. within the first pass round the loop (pass =1). The timing results are all normalised according to the clock speed, which is 62.5 ns for a 16 MHz crystal. Observe that the time interval between each timer interrupt is equal to a quarter of the sampling interval as is expected for the first sampling interval.

Following this exercise it was found that this second version runs satisfactorily, produces the expected results, and meets the time constraints. In addition, the following observations were made:

- After the timer T3 interrupt routine, the program jumped back to the " while (ITU_TSTR != 225) " loop and then proceeded out of that loop successfully.

- After the timer T0 interrupt routine the program jumped back in the main cycle, at " tjTemp = ta*usMag; " (no sector change branch).

- After the T1 interrupt the program jumpec back into the the second checkpoint loop, " while (ITU_TSTR !=252) " and remained in it for a while, since the last quarter of the sampling interval has not been reached yet. This indicates that there is room for increasing the switching frequency.

- It takes about 50µs to execute the simple instruction " tActual = counter *tSampling; " Similar float manipulations were observed to be very time consuming.

- When pheta equals to 359 degrees (worst case) it takes about 120µs to go through the algorithm that determines alpha and the value of sectorUpdate.

- The generation of the look-up table takes about 450ms.

- The assignment of switching parameter values to the interrupt cycle takes about 7 µs.

5.5 Testing SDEC0403.C

5.2 Testing SDEC0301.C

This test was mainly used to verify if the algorithm used to update the switching parameter values is a feasible approach. When run, the program performed without any break or overflow errors. The subroutines follow each other in the right order, in an endless loop. The timer registers and the output port register were updated accordingly. However, notice that the same set of switching parameter values are used each time. This is because this version is not concerned with the computation of the switching parameters, and so new values are not calculated during sampling intervals.

5.3 Testing SDEC0401.C

This program was written to verify the conclusion reached in section 5.1.1 above. It blends the first two programs together, in a first attempt to synthesise SVM using full on-line calculations. The switching frequency is set to 1 kHz. As expected this first version failed to produce an SVM pattern. The following observations were made:

- The program is still in the section where the switching times are calculated even after a complete sampling period has elapsed and the interrupt cycle is ready for new switching parameter values.

- Some C functions like fmod and floor use a lot of execution time. Some of the C functions themselves contain calls to other subroutines; e.g.: floor calls fmod.

5.4 Testing SDEC0402.C

In version 0.2 the switching time values are first generated in a look-up-table. Those pre-calculations assume a value of usMag of 1. In addition in this version, some of the most time consuming C functions were removed and replaced by simpler (though longer) routines. This version operates at 1 kHz with a value of maxSectorDiv of 30 (see section 2, SDE110).

In order to check the validity of the software at different stages, the checklist in appendix B was used. The table in appendix B offers a quick way of ensuring whether the values taken by the variables are correct and whether they are updated properly at the right time. In order to obtain running conditions closer to the real case, the software is run in real-time mode and then stopped at critical

5 Dynamic Analysis - Simulator Testing

During dynamic testing the program behaviour is observed while the program is running. The dynamic testing has been divided into two stages. The first one is restricted to simulator testing, while the second one runs on the emulator interfaced with the hardware platform.

A dedicated simulator for the microcontroller was not available for this project. Instead, simulation was performed by using the in-circuit emulator without connecting it to the target hardware. This test has for main objective to test timing issues. Through this test one can easily observe the effect of changes in the algorithm on the execution time. It was a major contributor to the improvement of the execution time of the processing operations. The test goes in detail, right at the variable values, for the first few revisions of the program. Once this iteration process has yielded a satisfactory working revision, the level of detail sought can be reduced to a routine check.

5.1 Testing SDEC0201.C

This small program was only written to have an idea of the time it takes for the microcontroller to perform the timing calculations. The program was run at different values of alpha. The results are tabulated below.

Table 8: Time taken to compute the switching times

Instructions	Execution time, measured at a value of alpha of:		
	0	Π/6	3Π/10
ta = ka*cos(alpha)-(1/sqrt(3))*sin(alpha);	2ms 923µs	4ms 378µs	4ms 387µs
tb = kb*sin(alpha);	1ms 366µs	1ms 366µs	1ms 366µs
t0 = tSampling-(ta-tb;	100µs	102µs	104µs

The above exercise rules out the possibility of performing those calculations on-line. To compute the switching times takes on average 5 to 6 ms. Even at a very low switching frequency, like 1 kHz (=1ms period), on-line computation will not be possible. This test strongly suggests alternative ways of obtaining the switching times, like for instance by using a look-up-table.

R14	8K2	SMD	External
R15	10K	SMD	External
R16	47E	SMD	External
R17	0E	SMD	MPS-RES0768
R18	1K	SMD	External
R19	2K2	SMD	External
R20	10K	SMD	External
R21	0E	SMD	MPS-RES0768
R22	1K	SMD	External
R23	10K	SMD	External
R24	47E	SMD	External
R25	0E	SMD	MPS-RES0768
R26	1K	SMD	External
R27	10K	SMD	External
R28	2K2	SMD	External
R29	0E	SMD	MPS-RES0768
R30	1K	SMD	External
R31	10K	SMD	External
R32	47E	SMD	External
R33	0E	SMD	MPS-RES0768
R34	1K	SMD	External
R35	2K2	SMD	External
R36	10K	SMD	External
R37	10K	SMD	External
R38	0E	SMD	MPS-RES0768
R39	1K	SMD	External
R40	4K7	SMD	External
R41	100E	SMD	External
R42	1K	SMD	External
R43	220K	SMD	External
R44	4K7	SMD	External
R45	4K7	SMD	External
R46	4K7	SMD	External
R47	4K7	SMD	External
R48	4K7	SMD	External
TR1	2N2222A	NPN Transistor	MPS-TRS0000
TR2	2N2222A	NPN Transistor	MPS-TRS0000
TR3	2N2907	PNP Transistor	MPS-TRS0000
TR4	2N2907	PNP Transistor	MPS-TRS0000
TR5	2N2222A	NPN Transistor	MPS-TRS0000
TR6	2N2222A	NPN Transistor	MPS-TRS0000
TR7	2N2907	PNP Transistor	MPS-TRS0000

IC4	4098	Dual Monostable Multivibrator	MPS-ICC0057
IC5	4013	Dual D-type Flip-Flop	MPS-ICC0023
IC6	4050B	Hex Non-Inverting Buffers	External
IC7	LM311	Voltage Comparator	MPS-ICC0154
IC8	40106B	Hex Inverting Schmitt Trigger	MPS-ICC0150
IC9	4071	Quad 2-Input OR Gate	MPS-ICC0052
IC10	LM311	Voltage Comparator	MPS-ICC0154
IC11	4098	Dual Monostable Multivibrator	MPS-ICC0057
IC12	4013	Dual D-type Flip-Flop	MPS-ICC0023
IC13	LM311	Voltage Comparator	MPS-ICC0154
IC14	LM311	Voltage Comparator	MPS-ICC0154
IC15	40106B	Hex Inverting Schmitt Trigger	MPS-ICC0150
IC16	4098	Dual Monostable Multivibrator	MPS-ICC0057
IC17	4013	Dual D-type Flip-Flop	MPS-ICC0023
IC18	LM311	Voltage Comparator	MPS-ICC0154
IC19	40106B	Hex Inverting Schmitt Trigger	MPS-ICC0150
IC20	4071	Quad 2-Input OR Gate	MPS-ICC0052
IC21	LM311	Voltage Comparator	MPS-ICC0154
IC22	4093	Quad 2 Input Nand Schmitt Trigger	MPS-ICC0076
IC23	4018	Presettable Divide by-N Counter	MPS-ICC0028
IC24	7812	+12V Regulator	MPS-REG0004
IC25	7812	+12V Regulator	MPS-REG0004
IC26	7805	+5V Regulator	MPS-REG0002
IC27	7805	+5V Regulator	MPS-REG0002
J1	-	Jumper3	External
J2	-	Jumper3	External
J3	-	Jumper3	External
J4		Jumper2	External
R1	10K/10T	4X10K Resistor Array	External
R2	1M	SMD	External
R3	10K	SMD	External
R4	10K	SMD	External
R5	1R	SMD	External
R6	22K	SMD	External
R7	4K7	SMD	External
R8	10R	SMD	External
R9	10K	SMD	External
R10	10K	SMD	External
R11	3K3	SMD	External
R12	10K	SMD	External
R13	4K7	SMD	External

C39	0.1uF-SMD	-	External
C40	0.1uF-SMD	-	External
C41	0.1uF-SMD	-	External
C42	0.1uF-SMD	-	External
C43	0.1uF-SMD	-	External
C44	0.1uF	-	MPS-CAP0052
C45	22uF	Electrolytic-Radial	MPS-CAP0258
C46	22uF	Electrolytic-Radial	MPS-CAP0258
C47	0.1uF-SMD	-	External
C48	0.1uF-SMD	-	External
C49	0.1uF-SMD	-	External
C50	0.1uF-SMD	-	External
C51	0.1uF-SMD	-	External
C52	0.1uF-SMD	-	External
C53	0.1uF-SMD	-	External
C54	0.1uF-SMD	-	External
C55	0.1uF	-	MPS-CAP0052
C56	22uF	Electrolytic-Radial	MPS-CAP0258
C57	22uF	Electrolytic-Radial	MPS-CAP0258
C58	0.1uF	-	MPS-CAP0052
C59	22uF	Electrolytic-Radial	MPS-CAP0258
C60	22uF	Electrolytic-Radial	MPS-CAP0258
D1	BAV99	Diode (SOT-23 Pack)	External
D2	BAV99	Diode (SOT-23 Pack)	External
D3	BAV99	Diode (SOT-23 Pack)	External
D4	BAV99	Diode (SOT-23 Pack)	External
D5	BAV99	Diode (SOT-23 Pack)	External
D6	BAV99	Diode (SOT-23 Pack)	External
D7	1N4148	Diode	MPS-DIO0021
D8	1N4148	Diode	MPS-DIO0021
DL1	LED	RED	MPS-DIL0004
DL2	LED	RED	MPS-DIL0004
DL3	LED	GREEN	MPS-DIL0001
DL4	LED	GREEN	MPS-DIL0001
DL5	LED	GREEN	MPS-DIL0001
DL6	LED	GREEN	MPS-DIL0001
-	Clip for each LED	Bi-coloured LED Holder	MPS-ACC0042
IC1	LM324	Low Power Quad Op-amp	External
IC2	LM385-2.5	Voltage Reference Diode - (SOT Package)	External
IC3	LM311	Voltage Comparator	MPS-IC0154

Appendix A: Bill Of Materials of the Inverter Control Card

Ref. Des.	COMPONENT	DESCRIPTION	SUPPLIER
C1	0.1uF-SMD	-	External
C2	0.1uF-SMD	-	External
C3	0.1uF-SMD	-	External
C4	0.01uF-SMD	-	External
C5	10uf-SMD	Electrolytic	External
C6	0.01uF-SMD	-	External
C7	10uF-SMD	Electrolytic	External
C8	0.1uF-SMD	-	External
C9	10uF-SMD	Electrolytic	External
C10	0.01uF-SMD	-	External
C11	10uF-SMD	Electrolytic	External
C12	0.1uF-SMD	-	External
C13	0.1uF-SMD	-	External
C14	10pF-SMD	-	External
C15	15pF-SMD	-	External
C16	0.1uF-SMD	-	External
C17	22uF	Electrolytic-Radial	MPS-CAP0250
C18	0.1uF-SMD	-	External
C19	22uF	Electrolytic-Radial	MPS-CAP0250
C20	1nF-SMD	-	External
C21	0.1uF-SMD	-	External
C22	22uF	Electrolytic-Radial	MPS-CAP0250
C23	0.1uF-SMD	-	External
C24	22uF	Electrolytic-Radial	MPS-CAP0250
C25	1nF-SMD	-	External
C26	0.1uF-SMD	-	External
C27	22uf	Electrolytic-Radial	MPS-CAP0250
C28	0.1uF-SMD	-	External
C29	22uf	Electrolytic-Radial	MPS-CAP0250
C30	1nF-SMD	-	External
C31	0.1uF-SMD	-	External
C32	1nF-SMD	-	External
C33	10uF-SMD	Electrolytic	External
C34	680pF-SMD	-	External
C35	0.1u	-	MPS-CAP0050
C36	22uf	Electrolytic-Radial	MPS-CAP0250
C37	22uf	Electrolytic-Radial	MPS-CAP0250
C38	0.1uF-SMD	-	External

that the integration of the software into its working environment does not work although they have been successfully tested separately. It is thus essential that a methodical and progressive method, such as the one adopted, be used when testing the system.

Machine Interface (MMI) are all controlled by microcontrollers.
Following this practical exercise, embedded systems as a whole is reviewed in SDE 113. For the time being, we can note the following observations in terms of the objectives of the case study towards the assessment of embedded systems at large:

a. The case study identified short design-coding-testing loops to address the risk elements associated with embedded systems developments. Those loops proved to be effective in providing insight in the problem and in probing unsure regions quickly.

b. The benefits of a disciplined approach. It supported the progression of the development towards the final goal of meeting the requirements within the prescribed time. It provided a stable framework and a set of intermediate milestones and guidelines to efficiently support the management of time and other resources.

c. The contribution of a quality management systems in supporting the development process was paramount. Through this study it became apparent that the development of embedded systems is characterised by a series of design changes which evolve as new constraints are experienced and as the system is better understood. Such changes have to be properly documented if the integrity of the development is to maintained.

d. Object-orientation adapts well to the problem presented by embedded systems. Unlike other design methodologies its framework encompasses naturally the software and as well as the hardware elements of the problem. It handles well the complexities of embedded systems, such as dependent events and concurrency. The four models (logical and physical, static and dynamic) of object-orientation covers all the possible design angles of the embedded system. At the moment, implementation of object-oriented design in an object-oriented language is still a problem. Either there is no OOL compiler for the microcontroller or the time constraints imposed on the design excludes the use of added complexity of an OOL.

e. The final integration and testing exercise for an embedded systems consumes a large proportion of the project time (about 20%). This is mainly due to the strong hardware interaction present in embedded systems. This increases considerably the number of unknowns and increases the chances

7 Closing the Chapter on the Case Study

All the objectives of the case study were achieved. In terms of the requirements specific to the case study itself, we note that:

a. PWM control pulses were generated by the microcontroller

b. The PWM pulses correspond to a balanced set of three-phase voltages, displaced by 120^0

c. The system provides automatic adjustment of the output voltage and is regulated by a feedback loop

d. The output is phase-locked to the Reserve within a pre-defined frequency window.

e. The system generates a "Static Switch Inhibit" signal upon an out of phase condition

f. The PWM output can be immediately inhibited upon an overload condition or upon an "Inverter Off" command.

In terms of the case study alone those results encourage further development in the microcontroller-control of the inverter. The relatively low switching frequency of 4 kHz will be adequate for power transistors but might not be sufficient to take the full advantage of fast switching devices like IGBTs and power MOSFETs. For those devices a different scheme, like the in-built PWM feature of the microcontroller might be more adequate. Another solution, if generation of the PWM pulses by SVM is to be maintained, is to use a more powerful controller. The introduction of new microcontrollers, like the Super-H SH7700 Hitachi microcontroller, which boasts a clock frequency of 60 MHz [4] (as compared to the 16 MHz clock frequency of the H8/3048) could improve the switching frequency by more than three times[2]. This will bring the switching frequency easily in the range of the fast-switching power devices. As sketched in section 2 of SDE 108, the application of microcontrollers can also be extended to capture most of the control processes of the UPS. This would promote the idea of an "intelligent" UPS, whereby the setting, data management, decision making and the Man-

[2] Note that the next generation of Hitachi Super H microcontrollers, the SH-4 is aiming at a clock frequency of 200 Mhz!

Table 13: Completed Functional Specifications Matrix on Control Features - Scenario III: Inverter Shutdown Condition

Requirements Traceability: Section 2.f, SDE 104			
Step	**Action**	**Expected Observation**	**Requirement Met: ✓/ ✗**
1	Trigger the Inverter-Off signal	All drive pulses should be off	✓
2	Cancel the Inverter-Off signal	All drive pulses should remain off	✓
3	Reset the microcontroller	-	-
4	Trigger the 300% Overload signal	All drive pulses should be off	✓
5	Cancel the 300% Overload signal	All drive pulses should remain off	✓

Table 12: Completed Functional Specifications Matrix on Control Features - Scenario II: Out of Phase Condition

Requirements Traceability: Section 2.d, 2.e, SDE 104			
Step	**Action**	**Expected Observation**	**Requirement Met: ✓/ ✗**
1	Set fRes below the Nominal value	-	-
2	Adjust vRes or vInv so as to create about 1 V difference between the two; i.e. simulate a large voltage difference between Reserve and Inverter	Static Switch Inhibit Signal must be Set (low signal) - Inhibit	✓
		Inverter output frequency should be at the higher limit of the frequency window	✓
3	Reduce the difference between vRes and vInv; i.e. return to Normal conditions	Static Switch Inhibit Signal must be Off (high signal) - Inhibit lifted	✓
		Inverter output frequency should follow fRes command input again	✓
4	Repeat steps 2 and 3 with fRes set above the Nominal value	The same behaviour should be observed, except that this time the inverter output frequency should shift at the lower limit of the frequency window	✓
5	Move vRes and vInv slightly on opposite side of the Nominal value; e.g. 2.45 and 2.55 - i.e. simulate an opposite direction condition	The same behaviour as in steps 2, 3 and 4 should be observed; i.e. the inverter frequency moves to the limiting value opposite to the side fRes is of the Nominal value. Again, the inverter should be as according to fRes command, when the parameters are adjusted back to the Normal Conditions	✓

Table 11: Completed Functional Specifications Matrix on Control Features - Scenario I: Inverter Failure Condition

Requirements Traceability: Section 2.c, 2.f, SDE 104			
Step	**Action**	**Expected Observation**	**Requirement Met: ✓/ ✗**
1	Move f_{Res} out of limits (test both limiting cases)	Static Switch Inhibit Signal must be Set (low signal) - Inhibit	✓
		Inverter output frequency should be at the nominal value, on all three phases	✓
2	Vary v_{Res}	Waveform pattern should not change	✓
3	Adjust v_{Res} outside the 2.4V - 2.6V range	-	-
4	Move v_{Inv} out of limits; i.e. simulate an Inverter Failure	Static Switch Inhibit Signal must be still be Set (low signal) - Inhibit	✓
		Drive pulses must be off - check at microcontroller output	✓
5	Move v_{Inv} back within limits	Drive pulses should NOT be restored	✓
		Static Switch Inhibit Signal remains Set (low signal)	✓
6	Move v_{Res} within 2.4V and 2.6V range - to simulate a zero-crossing	Static Switch Inhibit Signal must be Off (high signal) - Inhibit Lifted	✓
		Drives pulses should remain off	✓
7	Reset the microcontroller and repeat steps 1, 3, 4, 5 and 6, but instead of adjusting v_{Inv}, adjust f_{Inv}	The same behaviour as when v_{Inv} was adjusted should be observed	✓

Table 10: Completed Functional Specifications Matrix on General Feedback Features

Requirements Traceability: Section 2.c, 2.d, SDE 104			
Step	**Action**	**Expected Observation**	**Requirement Met: ✓/ ✗**
1	Vary f_{Res} within the frequency window	Inverter output[1] frequency should follow that of f_{Res} command, on all three phases	✓
		No change in output voltage magnitude should be observed	✓
2	Vary v_{inv} within the voltage window	As v_{inv} is adjusted below the nominal value the inverter output magnitude should increase	✓
		As v_{inv} is adjusted above the nominal value the inverter output magnitude should decrease	✓

[1] Since no DC voltage is applied, inverter output here refers to the filtered output of the drive pulses

6.4 Summary of Results obtained

The test results can now be compared with the original functional specifications for the project, as laid in section 6 of the Functional Specifications document, SDE106. The completed Functional Specifications matrix below, shows that all the requirements were successfully met. For more details about specific results, refer to section 6.1 above.

Table 9: Completed Functional Specifications Matrix on General Output Features

Requirements Traceability: Section 2.a, 2.b, SDE 104			
Step	**Action**	**Expected Observation**	**Requirement Met: ✓/ ✗**
1	Reset the microcontroller	PWM pulses should appear across the gate and source of each of the six IGBTs of the inverter bridge	✓
		A 50 Hz sinusoidal waveform should appear at each output phase of the inverter	✓
		The PWM pulses should correspond to a balanced set of three-phase waveforms displaced by 120°	✓
2	Change the nominal frequency to 60Hz and run the program - this test can be limited to the ICE testing	The same output as in step 1 should be observed, except that the output frequency should now be at 60 Hz	✓

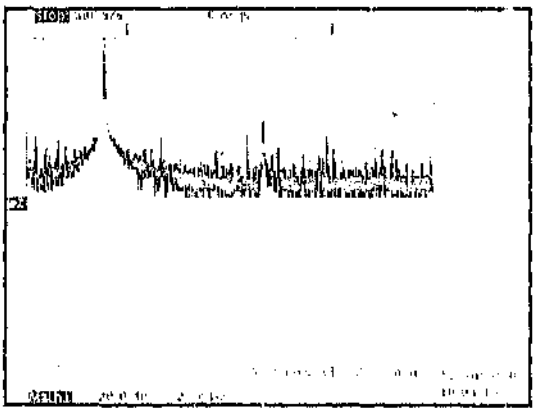


Figure H-9: FFT Plot (up to 250 Hz). SW1 (G1-S1) - filtered output

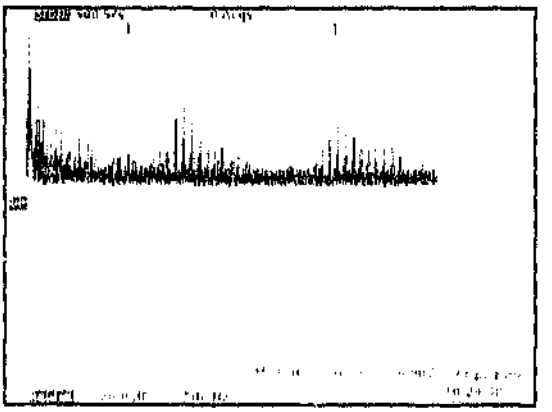


Figure H-10: FFT Plot (up to 5 kHz). SW1 (G1-S1) - filtered output

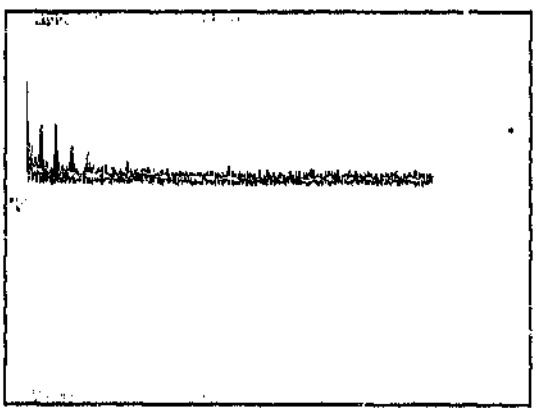


Figure H-11: FFT Plot (up to 50 kHz). SW1 (G1-S1) - filtered output

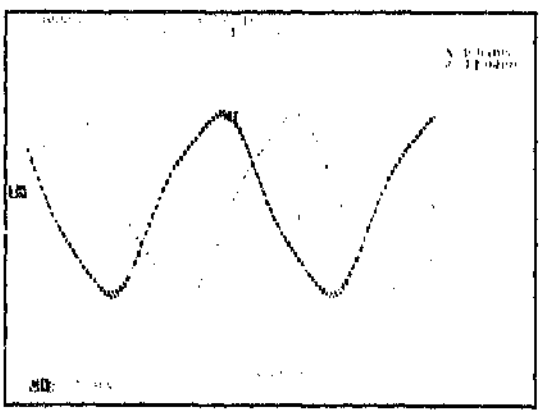


Figure H-12: Time Plot -version 0.6 (6.25kHz, no feedback). CH1: SW2 (G2-S2); CH2: SW1 (G1-S1); CH3: SW0 (G0-S0) - filtered output

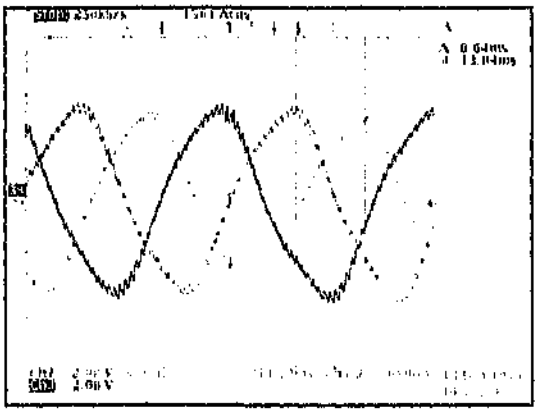


Figure H-5: Time Plot . CH1: SW2 (G2-S2); CH2: SW1 (G1-S1); CH3: SW0 (G0-S0) - filtered output

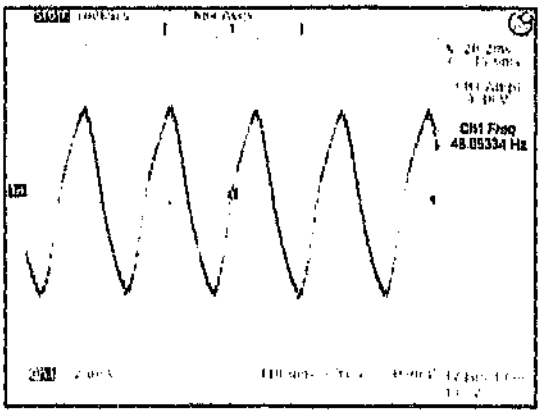


Figure H-6: Time Plot. SW1 (G1-S1) - filtered output

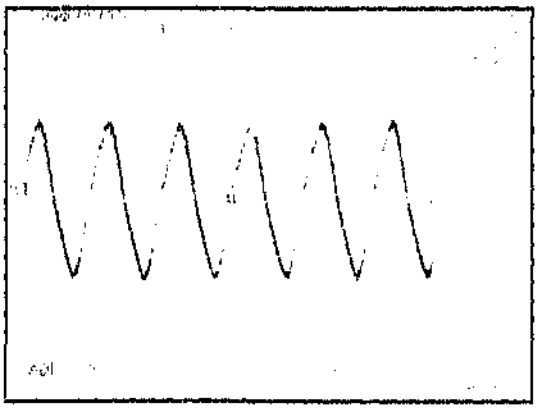


Figure H-7: Time Plot . CH1: SW5 (G5-S5); fNominal=60 Hz - filtered output

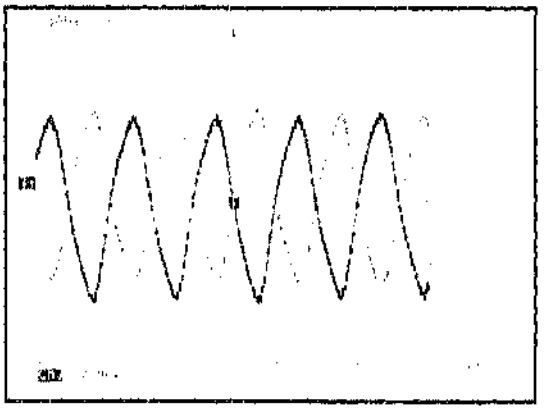


Figure H-8: Time Plot. CH1: SW5 (G5-S5); CH3: SW2 (G2-S2) - filtered output

Appendix H: Dynamic Live Testing Results

The pictures below were captured using the Tektronix Oscilloscope: TDS 544A (Serial Number: HV 110)

Date: 96/06/12 - 96/06/13

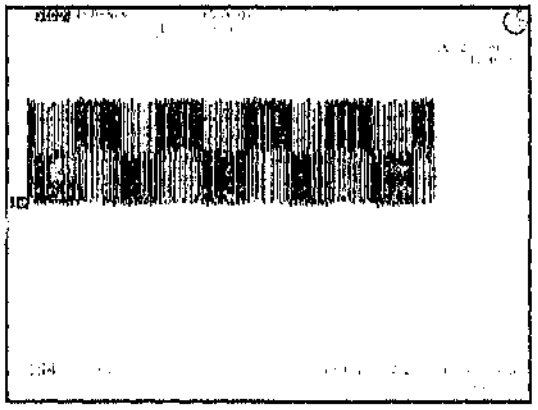


Figure H-1: Time Plot. Microcontrollor output pin P1.3-Ground - unfiltered output (I)

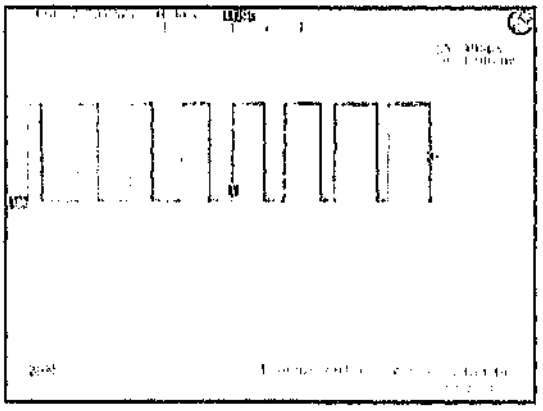


Figure H-2: Time Plot. Microcontroller output pin P1.3-Ground - unfiltered output (II)

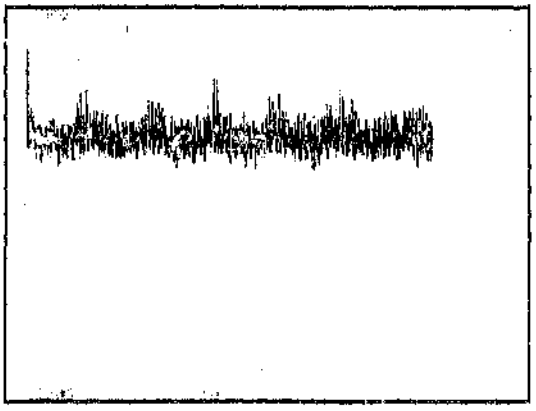


Figure H-3: FFT Plot. Microcontroller output pin P1.3 -Ground - unfiltered output

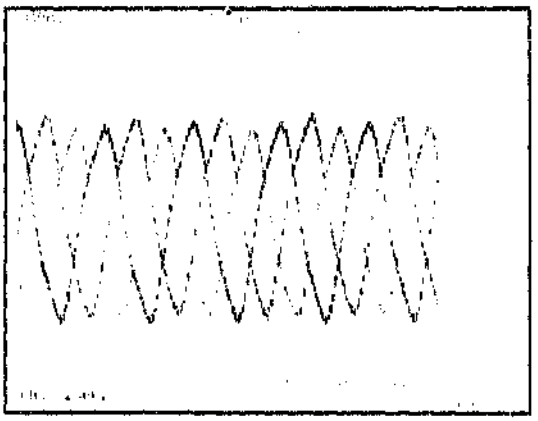


Figure H-4: Time Plot . CH1: SW6 (G6-S6); CH2: SW4 (G4-S4); CH3: SW3 (G3-S3) - filtered output

Appendix G: Simulation Test Results for SDEC0406.C (at usMag = usMagMin)

Variable	Value Held at the beginning of timer T2 Interrupt at pass:			
	1	2	3	4
Time between BPs (µs)	836 ms (first BP)	151	152	152
sequence	1	-1	1	-1
sector	1	1	1	1
counter	0	1	2	3
tActual	0	13	26	39
phsta	0	234000	468000	702000
alpha	0	234000	468000	702000
finv	500	500	500	500
ujTemp	35	35	49	35
ukTemp	49	49	35	49
uZeroTemp	7	56	7	56
tjTemp	1021	46	971	143
tkTemp	0*	40*	997*	92*
t0HalfTemp	0	589	598	588
uj	49	35	49	35
uk	35	49	35	49
uZero	7	56	7	56
tj	560	1021	46	971
tk	560	40	997	92
t0Half	560	589	598	588
ta	2042	1974	1943	1883
tb	0	92	185	287
i	0	9	18	28
uzjTrans	48	3	48	3
ujkTrans	33	33	33	33
ukzTrans	3	48	3	48

* Parameters not updated yet

Appendix F: Simulation Test Results for SDEC0405.C (at usMag = usMagMin)

Variable	Value Held at the beginning of timer T2 interrupt at pass:					
	1	2	3	4	5	6
Time between BPs (µs)	851 ms	170	173	174	173	174
sequence	1	-1	1	-1	1	-1
sector	1	1	1	1	1	1
counter	0	1	2	3	4	5
tActual	0	16	32	48	64	80
phase	0	288000	576000	864000	1152000	1440000
alpha	0	288000	576000	864000	1152000	1440000
finv	500	500	500	500	500	500
ujTemp	35	49	35	49	35	49
ukTemp	49	35	49	35	49	35
uZeroTemp	56	7	56	7	56	7
tjTemp	1167	64	1093	199	1009	331
tkTemp	40	1133	135	1054	268	965
t0HalfTemp	676	681	666	65.	641	632
uj	35	49	35	49	35	49
uk	35	49	35	49	35	49
uZero	7	56	7	56	7	56
tj	1167	1167*	64*	1093*	1009	331
tk	640	40	1133	135	1054	268
t0Half	640	676	681	666	693	641
ta	2334	2267	2187	2109	2018	1930
tb	0	129	270	398	537	663
i	0	11	23	34	46	57
ujTrans	3	48	48*	48	3	48
ujkTrans	33	33	33	33	33	33
ukzTrans	3	48	3	48	3	48
ujTransTemp	-	-	3	48	3	48
ujkTransTemp	-	-	33	33	33	33
ukzTransTemp	-	-	48	3	48	3

* Parameters not updated yet Compare with table in appendix 5

tk	640	40	1964	233	1827	465	1672	223	1969	1889	122	2022
t0Half	640	240	242	216	194	173	157	218	246	214	240	249
ta	2334	2267	2187	2109	2018	1930	1829	117	2328	141	2334	2267
tb	0	129	270	398	537	663	799	2273	11	2260	0	129
i	0	11	23	34	46	57	69	230	1	228		11
uzjTrans	3	48	3	48	3	48	3	3	48	3	3	3
ujkTrans	33	33	33	33	33	33	33	33	33	34	34	33
ukzTrans	3	48	3	48	3	48	3	3	48	3	40	3

Appendix E: Simulation Test Results for SDEC0405.C (at usMag = usMagMax)

	Value Held at the beginning of timer T2 interrupt at pass:											
Variable	1	2	3	4	5	6	7	21	22	125	126	127
Time between BPs (µs)	636 ms	197	173	174	174	174	174	-	171	-	171	172
sequence	1	-1	1	-1	1	-1	1	1	1	-1	-1	1
sector	1	1	1	1	1	1	1	1	2	6	1	1
counter	0	1	2	3	4	5	6	20	21	124	0	1
tActual	0	16	32	48	64	80	96	320	336	1984	2000	16
pheta	0	288000	576000	864000	1152000	1440000	1728000	6760000	6048000	36712000	36000000	288000
alpha1	0	288000	576000	864000	1152000	1440000	1728000	6760000	6048000	6712000	0	288000
finv	500	500	500	500	500	500	500	500	500	500	500	500
ujTemp	35	49	35	49	35	49	35	35	49	35	49	35
ukTemp	49	35	49	35	49	35	49	49	21	42	35	49
uZeroTemp	56	7	56	7	56	7	56	56	7	56	7	56
tjTemp	2022	111	1895	344	1748	574	1585	101	2017	1968	40	1964
tkTemp	40	1964	233	1827	485	1672	692	1969	40	122	2022	111
t0HalfTemp	249	242	216	194	173	157	141	245	251	240	249	242
uj	35	49	35	49	35	49	35	35	49	35	49	35
uk	35	49	35	49	35	49	35	35	49	35	42	35
uZero	7	56	7	56	7	56	7	7	56	7	56	7
tj	2022	111	1895	344	1748	574	1585	101	2107	1958	40	1964

t[34][0] = 1477,	t[34][1] = 1954
t[35][0] = 1424,	t[35][1] = 1932
t[36][0] = 1370,	t[36][1] = 1980
t[37][0] = 1316,	t[37][1] = 2027
t[38][0] = 1262,	t[38][1] = 2074
t[39][0] = 1207,	t[39][1] = 2120
t[40][0] = 1152,	t[40][1] = 2165
t[41][0] = 1096,	t[41][1] = 2210
t[42][0] = 1041,	t[42][1] = 2254
t[43][0] = 985,	t[43][1] = 2297
t[44][0] = 928,	t[44][1] = 2340
t[45][0] = 872,	t[45][1] = 2382
t[46][0] = 815,	t[46][1] = 2423
t[47][0] = 757,	t[47][1] = 2464
t[48][0] = 700,	t[48][1] = 2504
t[49][0] = 642,	t[49][1] = 2543
t[50][0] = 585,	t[50][1] = 2581
t[51][0] = 527,	t[51][1] = 2618
t[52][0] = 468,	t[52][1] = 2655
t[53][0] = 410,	t[53][1] = 2690
t[54][0] = 352,	t[54][1] = 2726
t[55][0] = 293,	t[55][1] = 2760
t[56][0] = 235,	t[56][1] = 2793
t[57][0] = 176,	t[57][1] = 2825
t[58][0] = 117,	t[58][1] = 2857
t[59][0] = 58	t[59][1] = 2888

Appendix D: Look-Up-Table of Switching Times at a Sampling Frequency of 5 kHz and a Value of maxSectorDiv of 60 (Assuming usMag =1)

t[0][0] = 2918.	t[0][1] = 0
t[1][0] = 2980.	t[1][1] = 58
t[2][0] = 2857.	t[2][1] = 117
t[3][0] = 2925.	t[3][1] = 176
t[4][0] = 2793.	t[4][1] = 235
t[5][0] = 2760.	t[5][1] = 293
t[6][0] = 2726.	t[6][1] = 352
t[7][0] = 2690.	t[7][1] = 410
t[8][0] = 2656.	t[8][1] = 468
t[9][0] = 2618.	t[9][1] = 527
t[10][0] = 2581.	t[10][1] = 585
t[11][0] = 2542.	t[11][1] = 642
t[12][0] = 2504.	t[12][1] = 700
t[13][0] = 2464.	t[13][1] = 757
t[14][0] = 2423.	t[14][1] = 815
t[15][0] = 2382.	t[15][1] = 872
t[16][0] = 2340.	t[16][1] = 928
t[17][0] = 2297.	t[17][1] = 985
t[18][0] = 2254.	t[18][1] = 1041
t[19][0] = 2210.	t[19][1] = 1096
t[20][0] = 2165.	t[20][1] = 1152
t[21][0] = 2120.	t[21][1] = 1207
t[22][0] = 2074.	t[22][1] = 1262
t[23][0] = 2027.	t[23][1] = 1316
t[24][0] = 1980.	t[24][1] = 1370
t[25][0] = 1932.	t[25][1] = 1424
t[26][0] = 1884.	t[26][1] = 1477
t[27][0] = 1835.	t[27][1] = 1529
t[28][0] = 1785.	t[28][1] = 1581
t[29][0] = 1735.	t[29][1] = 1633
t[30][0] = 1684.	t[30][1] = 1684
t[31][0] = 1633.	t[31][1] = 1734
t[32][0] = 1581.	t[32][1] = 1785
t[33][0] = 1529.	t[33][1] = 1835

Appendix C: Simulation Test Results for SDEC0403.C

Variable	Value Held at the beginning of timer T3 interrupt at pass:								
	1	2	3	4	5	6	7	19*	100**
Time between SPs (µs)	215 ms (first SP)	218	212	211	212	212	212	-	-
sequence	-1	1	-1	1	-1	1	-1	1	1
sector	1	1	1	1	1	1	1	2	1
counter	-1	0	1	2	3	4	5	17	0
tActual	0	0	20	40	60	80	100	340	2000
pheta	0	0	3.6	7.2	10.8	14.4	18	61.2	360
alpha	0	0	3.6	7.2	10.8	14.4	18	1.2	0
finv	0	50	50	50	50	50	50	50	50
ujTemp	49	35	49	35	49	35	49	49	35
ukTemp	35	49	35	49	35	49	35	21	49
uZeroTemp	7	56	7	56	7	56	7	7	56
tjTemp	0	2398	144	2211	480	1991	855	2373	2398
tkTemp	0	50	2322	337	2121	669	1852	47	50
t0HalfTemp	0	375	366	325	299	269	246	369	375
uj	49	35	49	35	49	35	49	49	35
uk	35	49	35	49	35	49	35	21	49
uZero	7	56	7	56	7	56	7	7	56
tj	799	2398	144	2211	480	1991	855	2373	2398
tk	799	50	2322	337	2121	669	1852	47	50
t0Half	799	375	366	325	299	269	246	369	375
ta	0	2918	2825	2690	2581	2423	2254	2688	2918
tb	0	0	176	410	585	815	1041	58	0
P1DR	7	56	7	56	7	56	7	7	56
ITU_GRA0	799	375	366	325	299	269	246	369	375
i	60	0	3	7	10	14	18	1	0

* Sector Change
** Revolution Change

Appendix B: Simulation Test Results for SDEC0402.C

Variable	Value Held at:		
	Beginning of timer T3 interrupt: ITU_TSTR = 1; (pass =1)	Beginning of timer T0 interrupt: ITU_TSTR = 2; (pass =1)	Beginning of timer T1 interrupt: ITU_TSTR = 4; (pass =1)
Time elapsed since last breakpoint	460 ms (First breakpoint)	250 µs	250µs
sequence	-1	1	1
sector	1	1	1
counter	-1	0	0
tActual	0	0	0
pheta	0	0	0
alpha	0	0	0
flnv	0	50	50
ujTemp	49	49	35
ukTemp	35	35	49
uZeroTemp	7	7	56
tjTemp	0	0	11890
tkTemp	0	0	0
t0HalfTemp	0	0	2055
uj	49	49	49
uk	35	35	35
uZero	7	7	7
tj	4000	4000	4000
tk	4000	4000	4000
t0Half	4000	4000	4000
ta	0	145	145
tb	0	0	0
ITU_TSTR	H'E0 (→ H'E1)	H'E0 (→ H'E2)	H'E0 (→ H'E4)
P1DR	7	49	35
ITU_TSR3	H'F8	-	-
ITU_TSR0	-	H'F8	-
ITU_TSR1	-	-	H'F8
ITU_GRA0	H'FA0	-	-
ITU_GRA1	H'FFFF	H'FA0	
ITU_GRA2	-	-	H'FA0

TR8	2N2907	PNP-Transistor	MPS-TRS0066
TR9	2N2222A	NPN-Transistor	MPS-TRS0006
TR10	2N2222A	NPN-Transistor	MPS-TRS0006
TR11	2N2907	PNP-Transistor	MPS-TRS0066
TR12	2N2907	PNP-Transistor	MPS-TRS0066
TR13	2N2222A	NPN-Transistor	MPS-TRS0006
TR14	2N2222A	NPN-Transistor	MPS-TRS0006
TR15	2N2907	PNP-Transistor	MPS-TRS0066
TR16	2N2907	PNP-Transistor	MPS-TRS0066
TR17	2N2222A	NPN-Transistor	MPS-TRS0006
TR18	2N2222A	NPN-Transistor	MPS-TRS0006
TR19	2N2907	PNP-Transistor	MPS-TRS0066
TR20	2N2907	PNP-Transistor	MPS-TRS0066
TR21	2N2222A	NPN-Transistor	MPS-TRS0006
TR22	2N2222A	NPN-Transistor	MPS-TRS0006
TR23	2N2907	PNP-Transistor	MPS-TRS0066
TR24	2N2907	PNP-Transistor	MPS-TRS0066
U1	H3046	Microcontroller	External
-	-	Socket for H3046	External
X1	-	Header 10P 90 Degree	External
X2	-	Header 16P 90 Degree	MPS-PLG0045
X3	-	Header 16P 90 Degree	MPS-PLG0045
X4	-	Header 16P 90 Degree	MPS-PLG0045
X5		6P Connector	MPS-PLG0047
XT1	16MHz	Crystal	External

2 Software Development Processes Examined in the Context of Embedded Systems

2.1 An Assessment

One aspect of small embedded systems that this study revealed is that they have traditionally been developed in an empirical way. Because of the relatively small size of the code, developers have been deceived into thinking that small embedded systems could be developed without any formal development process. The mistake here is to judge the complexity of a software system uniquely by its size and to fail to recognise such factors as the role the software is to play in the system, complexity of the control function, safety and reliability issues, and maintainability issues. Embedded systems, by their often critical role in a system and the frequent upgrades and modifications they are subjected to, can considerably benefit from a disciplined approach of development.

The question is: is there a specific development process that is appropriate for all embedded systems? The nature of embedded systems and the characteristics of their development as we saw it, would tend to show that the formulation of such a process is not feasible. The reasons are:

a. The large diversity in the applications of embedded systems

b. The maturity of the organisations developing embedded systems varies considerably, making it difficult to develop a process suitable to all.

c. Although embedded systems are generally characterised by constraints on space, time, power consumption and by concerns such as reliability, safety and cost, the actual degree of importance of each of those factors can vary largely between projects, making it necessary to approach each project differently.

d. The importance of the role of the microcontroller in the overall system can itself vary largely. Consequently the allocated time and budget for embedded systems projects vary accordingly.

However despite those difficulties we can attempt to sketch some general patterns that an embedded systems development should follow. In particular, a development process for embedded systems HAS to accommodate the following:

c. SDE 104, Requirements Analysis
d. SDE 105, Inverter Control Techniques and Phase-Locked-Loop
e. SDE 106, Functional Specifications
f. SDE 107, Design Specifications - Part 1
g. SDE 108, Design Specifications - Part 2
h. SDE 110, Software and Hardware Implementation Documentation
i. SDE 112, Testing Documentation

1.6 Assumptions

It is assumed that the reader is familiar with the contents of the literature survey as laid out in documents SDE 102 and SDE 103, and with the development of the case study associated with this project.

1.7 ISO 9001 Requirements Traceability

- ISO 9001, Clause 4.4.5 - Design Output

[18] Shen, Chia, Ramamritham K., Stankovic J.A., Resource Reclaiming in Real-Time. Proceedings of the 11th Real-Time Systems Synposium, Lake Buena Vista, Florida, December 1990, pg 41-50

[19] Washabaugh, Douglas M., Kafura D., Incremental Garbage Collection of Concurrent Objects for Real-Time Applications. Proceedings of the 11th Real-Time Systems Symposium. Lake Buena Vista, Florida, December 1990, pg 21-30

[20] Allard J.R., Hawkinson L.B., Real-Time Programming in common LISP, Communications of the ACM 35(9), September 1991, pg 64-69

[21] Allard J.R., Hawkinson L.B., Real-Time Programming in common LISP, Communications of the ACM 35(9), September 1991, pg 64-69].

[22] Laplante P.A., "Real-Time Systems Design and Analysis, An Engineer's Handbook", IEEE Press, USA, 1993

[23] Brown B., "Overlapped I/O", http://www.cit.ac.nz/smac/cbt/hwsys/embedded/swparta.htm

[24] Hersch R., "Embedded Systems Development Information", Orion Instruments, http://www.oritools.com/info/

[29] Budde R., Sylla K.," Horizontal Integration of Paradigms", http://www.

[30] Garnett J., "The Compositional C++ Language", http://pblo.caltech.edu/ccpp/lang_def/node1.html#SECTION00010000000000000000

[31] Schneidewind N.F., "Methodology for Validating Software Metrics", IEEE Transactions on Software Engineering, Vol. 18, No. 5, May 1992, pp. 410-421

[32] Crockett H.D., What's the Proper Role for CASE Tools? -Counterpoint", IEEE Software, March 1995, pp 19

1.5.5 Other Documents

a. SDE 102, Software Development: A General Survey of the Literature
b. SDE 103, Embedded Systems: A Literature Survey

[5] Zave P., "An Operational Approach to Requirements Specification for Embedded Systems", IEEE Transactions on Software Engineering, May 1982, pp 250-269

[6] Sharp A., "Software Quality and Productivity", Van Nostrad Reinhold, USA, 1993

[7] Adamson M., "Small Real-Time Design", Sigma Press, John Wiley and Sons, U.K, 1990

[8] Humphrey W.S., "Managing the Software Process", Addison-Wesley Publishing Co., USA, 1990

[9] Auer A., Levanto M., Okkonen A., Okkonen J., "Solution In Software Crisis", Microprocessor and Microprogramming, Vol 30, Part 1-5, pp 273-280

[10] Williams T., "Object-Oriented Methods Transform Real-Time Programming", Computer Design, September 1992, pp 101-118

[11] Booch G., "Object-Oriented Analysis and Design", Second Edition, The Benjamin/ Cummings Publishing Co., Inc., USA, 1994

[12] Rumbaugh J., Blaha M., Premerlani W., Eddy F., Lorensen W., Object-Oriented Modelling and Design, Prentice-Hall, USA

[13] Brooks F. P., No Silver Bullet- Essence and Accidents of Software Engineering, IEEE Computer, April 1987, pg 10-19

[14] Cox B. Object-oriented Programming: An Evolutionary Approach. New York: Addison Wesley. 1991

[15] Ingalls D., Object-oriented programming. Video tape, Apple Computer, Inc., Part of the University Video Communications collection, Distinguished Lecture Series, Volume II, 1989

[16] "Professional Language Development Toolkit", Abraxas Software, Inc. D2205@applelink.apple.com

[17] Wulf W., Shaw M., Global Variables Considered Harmful, Sigplan Notices 8(2), 1973

- Individuals who will perform internal and external surveillance audits of SEAL Quality Management System
- Engineering Manager, Meissner
- Product Developer
- Product Manager

1.5 Applicable Documents

1.5.1 Specifications

None

1.5.2 Standards

a. SEAL QMS Document Creation Template, QS 002, Revision 0.01, 5 April 1994

b. SEAL QMS Document Layout, Presentation and Typesetting Guide, QS 003, Revision 0.01, 1 April 1994

1.5.3 Procedures

None

1.5.4 Guidelines

[1] Honka H., Kattilakoski M., "A Simulation-Based System for Testing Real-Time Embedded Software in the Host Environment", Microprocessor and Microprogramming, Vol. 32, Part 1-5, pp 127-134

[2] Stoyenko A.D., Welch L.R., Chao Cheng B., "Response Time Prediction in Object-Based, Parallel Embedded Systems", Microprocessor and Microprogramming, Vol 40, Part 2-3, April 1994, pp 135-150

[3] Chiodo M., Giusto P., Jurecska A., Mareill M., Hsieh H.C., Sangiovanni-Vincentelli A., Lavagno L., "Hardware-Software Codesign of Embedded Systems", IEEE Micro, Vol 14, Part 26-36, August 1994, pp 26-36

[4] Wolf W. H., "Hardware-Software Co-Design of Embedded Systems", Proceedings of the IEEE, Vol. 82, No. 7, July 1994, pp 967-989

1 Scope

1.1 Introduction

Following the practical development project, the subject of embedded systems as framed in the literature survey documents (SDE 102, SDE 103), is reviewed. In particular, the development process and the quality management system are re-examined. In addition, drawing from the experience gained from the case study, some practical features of embedded systems development in general are captured. The document ends with some reflections on the future developments around embedded systems.

1.2 Purpose

The purpose of this document is to lay out the essential conclusions and observations regarding the whole study performed on software development for embedded systems.

1.3 Definitions

- OO : Object-Orientation
- OOA : Object-Orientation Anlaysis
- OOD : Object-Orientation Design
- PWM : Pulse-Width Modulation
- ICE : In-Circuit Emulator
- ASIC : Application Specific Integrated Circuit

1.4 Audience

The audience for this document comprise the various stakeholders of SEAL, including:

- Full-time staff members of SEAL
- All full-time and part-time post-graduate students associated with SEAL
- Members of the SEAL Management Board
- Head of the Department, Electrical Engineering

Change History

Configuration Control

Project:	SDES
Title:	Embedded Systems: A Review
Doc. Reference:	SDE 113
Created by:	H. Bapoo
Creation Date:	8 August 1996

Document History

Version	Date	Status	Who	Saved as:
0.01	96/08/08	Draft	H.Bapoo	:\1995-13\TP\SDE113WP.100
1.00	96/08/25	Approved	H.Bapoo	:\1995-13\TP\SDE113WP.100

Revision History

Version	Date	Changes
0.01	96/08/08	New Document

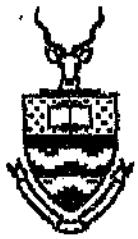
Management Authorisation

Version	Date	Status	Management Board Minute Reference
1.00	96/08/25	Approved	Section 2.3 of SDE 592

Change Forecast

None

Table of Contents



SDES QMS

Embedded Systems: A Review

Technical Product

Version 1.00

Document Status: Approved

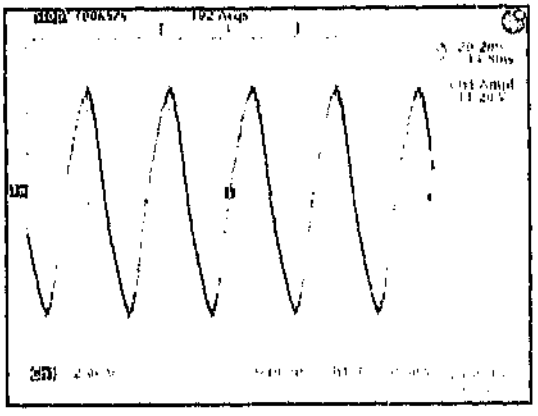


Figure H-17: Time Plot. SW5(G5-S5); vinv = 0.5V - filtered output

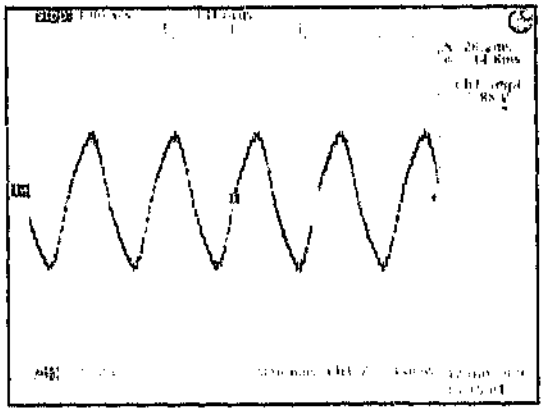


Figure H-18: Time Plot. SW5(G5-S5); vinv = 4.5V - filtered output

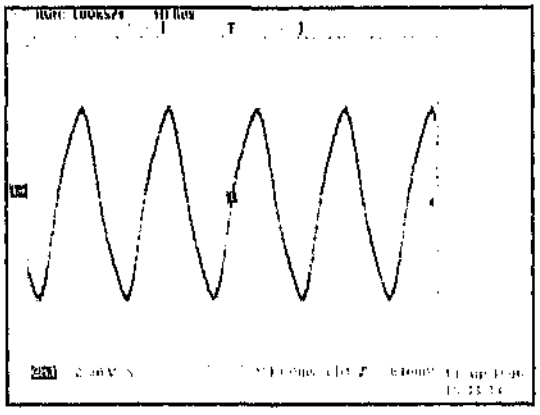


Figure H-13: Time Plot -version 0.6 (6.25 kHz, no feedback). SW2 (G2-S2) - filtered output

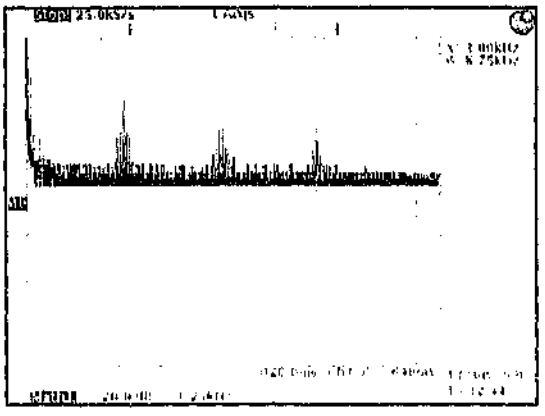


Figure H-14: FFT Plot -version 0.6 (6.25 kHz, no feedback). SW2 (G2-S2) - filtered output

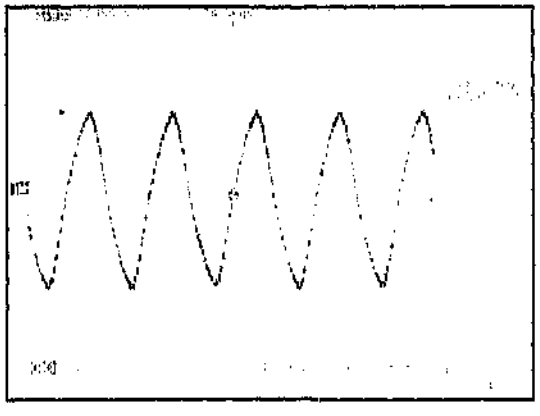


Figure H-15: Time Plot . SW5 (G5-S5); fRes= 4.5V - filtered output

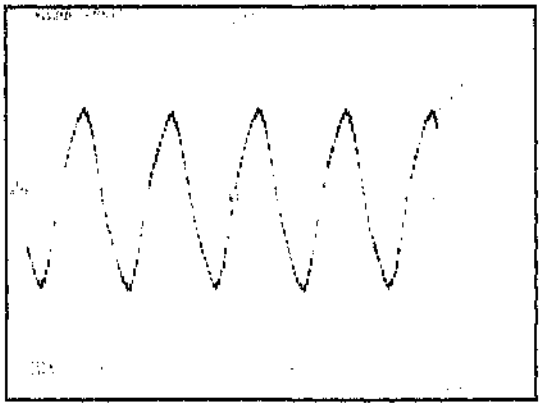


Figure H-16: Time Plot. SW5(G5-S5); fRes= 0.5V; No DC Applied

5 Object-Orientation and Embedded Systems Projects

5.1 An Assessment

Object-oriented real-time embedded applications were rare cases years ago and still tend to be scarce. But since more and more software is put into controllers, the demand for software that is easy to :

- understand
- maintain
- reuse

is growing, like the commercial applications, irrespective of the size of the project. Booch [11] himself explains that even simple applications can lend themselves well to object-oriented architecture.

The case study is a very good example of the benefits that object-orientation can bring to small embedded systems. At first glance, the case study might seem, fairly simple, encompassing only a handful of classes. One might be tempted to discard an object-oriented approach at all. This would be to misjudge the problem. Firstly when judging the size of the project one should look beyond the boundaries of the problem domain as specified in the functional specifications. The problem should be seen in the context of the broader perspective of its application. Doing so for the case study, immediately reveals, firstly, a much more complex system. Secondly, it also shows the benefits that an object-oriented approach could yield in facilitating the integration of the inverter control design within the whole UPS framework, while also assisting and promoting maintenance and re-use. The potential for re-use here, is considerable, especially when observing the similarity between the inverter and the rectifier units.

The programming language though, is still a problem for embedded systems, as far as object-oriented implementation is concerned. It has been postulated that the essence of object-oriented development is the identification and organisation of application-domain concepts, rather than their final representation in a programming language, object-oriented or not [12]. Brooks [13] said that the hard part of software development is the manipulation of its essence due to the inherent complexity of the problem, rather than the accidents of its mapping into a particular language which are due to temporary imperfections in our tools that are rapidly being corrected. Those statements are certainly true, in the sense that one should differentiate between object-orientation and object-oriented languages. Object-orientation is a concept that is translated into object-oriented analysis and design techniques. Object-oriented languages (together with compilers) are simply

without a continuous monitoring and mentorship by the management, CASE tools will almost always lead to reduced productivity and quality. Relating the subject to the CM model; only at level 3, after that the process has been defined and understood can advanced technology be usefully introduced. Another factor that plays against software tools is their lack of standards. Increasingly, tool vendors are setting up environments that address the way programmers develop code in the real world, which often does not conform to theories about how such development should be done. [10] . Often the tools are intended for large systems where the penalty of longer, slower code is accepted in the interests of faster development and easier project management when using a team of programmers. This may make a small system unworkable, although some of the principles can be adopted where appropriate.

Therefore, while introducing CASE tools can be advantageous, especially in view of expansion and growth of the organisation, it should be more important for the organisation to define the process that is best suited for its needs first.

4 Software Metrics and CASE Tools

In addition to what has already been mentioned about software metrics, we find that it is appropriate to add a further cautionary remark. One should be aware that tentative attempts of quantifying the development process might have harmful consequences upon the process and the development team itself [8]. Two important points to take into considerations when trying to quantify a particular aspect of the software process are:

- One must have good justifications for measuring that aspect. There is no point in measuring something that is not pertinent or that is of minor importance to the process and then to try to thrust those measurements forward as being crucial. Arbitrary measurements might be harmful, especially because human beings are very sensitise to the grading of their work or their own abilities.

- Any measurement tool or process needs to be thoroughly tested before being made available and being used as a determining factor.

Should those two criteria not be respected the results of applying measurements tools might be disappointing. We agree with Schneidewind [31] who believes that software metrics should be treated as part of an engineering discipline and that they should be evaluated (validated) to determine whether they measure what they purport to measure prior to using them. He adds that, if metrics are to be of greatest utility, the validation should be performed in terms of the quality functions (quality assessment, control, and prediction) that the metrics are to support.

On the subject of CASE tools, we find that many of the existing design methodologies are strongly dependent upon them. This trend is well depicted in Auer's et. al. [9] seven golden rules to escape from the software crisis, which strongly emphasise the importance of automating the development process.

As far as embedded systems are concerned, design methodologies based strongly on CASE tools, have one thing in common in that they fail to recognise the actual immaturity of the embedded systems community. Inducing automation development techniques into an environment that is still trying to define the process can do more harm than good. Adamson notes that computer-aided software design packages can add to the inefficiency of embedded systems projects [7]. This point of view is shared by Crockett [32], who believes that

set-up of the quality system, it should be viewed as an investment in the life-cycle of the software. This, however is an area where the development of embedded systems might differ slightly. In the previous section we presented a development approach that has an inner design-coding-testing iterative loops. To apply strict control over all those minor tentative loops will tend to defeat their very purpose which is to obtain quick feedback about specific aspects of the design. In that respect enforcement of a strict quality management system might have detrimental effects upon the design and the ardour of the team itself. In addition the first few outputs of those loops might be of minor importance to the later development of the software. Their inclusions in the documentation will have the further drawback of increasing the volume of documentation and of obscuring the more important features.

A loose change control is contrary to good QMS practice but it has the advantage of giving the developer more freedom, with reduced interruptions. In fact, each iteration through the loop can be seen almost like a brainstorming exercise. However to allow complete freedom without any documentation throughout the process will, after a number of iteration, start to generate negative returns on this process. This will usually happen as the developer starts to forget some of the factors that evolved during earlier iterations. Therefore it is necessary to discipline the approach, while still providing it with the freedom necessary for its success. A balance has to be struck between the two concerns. The solution is simply to document only selected revisions at the early stage of the design. Only when a certain level of predetermined performance is achieved or when major characteristics of the software are revealed is the design process halted and the revision updated. Adamson [7] suggested a similar approach when he pointed out that at first rough drafts may be produced and circulated. At some point though, when there is general agreement on the direction the design should take, the design should be "frozen" . After that any changes must pass through the formal procedures.

With a careful management of the quality system, standards should not reduce creativity. As Sharp [6] puts it, to say that standards reduces creativity is like saying that the use of sentences and grammar stifles the creativity of writers. Creativity in software engineering comes from recognising new classes of problems that can be solved by computers and by putting together building blocks to create new and elegant solutions. Creativity is not manifested by a refusal to follow standards.

between the advocates of different programming styles. When this happens much of the benefit of the walkthrough is lost. To be able to undertake walkthroughs in the presence of external parties - representatives of management and the quality assurance department - without personal animosity and with all the efforts concentrated on the overall design, requires some maturity from the organisation. The organisation should preferably be involved in a maturation process, such as the one described in section 4.3 of SDE 102.

A lot of people object to following standards [6]. They tend to believe that the standards are not helpful to them. The validity of this statement depends upon whether standards are being looked from a project viewpoint or from a single developer viewpoint. From the project viewpoint it is crucial. From the single developer viewpoint it might be of less value, especially if the latter will be moving onto completely new software types once the present project is over or if he (or she) plans to leave the company soon.

It is important to be aware of the overheads associated with the application of a quality management system. The cost of quality appears mainly in the time it consumes. The areas where time on QMS activities was mostly consumed on the case study are:

- In the initial set-up of the standards to the project
- In managing changes

In the early stages of the project, new documents have to created, templates have to be set up or adjusted, parameters have to be defined and all the aspects covered by the applicable standards in general have to be reviewed prior to even starting the technical development. If the standards are new to the developers, they all have to become familiar to the standards. It is essential that the whole team be conversant with the standards and not just one or two members whose are assigned the task of enforcing the standards. Ignorance or unfamiliarity of the standards among the team, can lead to misunderstanding, disagreements and time wasting. However all those preconditions of applying quality standards require a considerable amount of time and commitment from all parties concerned. This time and commitment should not be underestimated. Adapting the standards to a new project is certainly not a minor task and should be viewed as a phase of the development itself, instead of being seen as an extra factor that consumes development time.

Control on changes too, are time consuming. Like the time spent on the initial

3 Quality Management Systems Examined in the Context of Embedded Systems

The case study was developed within the framework of the ISO 9001 quality standard. This section is a compilation of important factors experienced on the application of standards and some general reflections on the issue of quality management systems, with respect to the development of software.

One of the main strengths of quality standards that was experienced during the project was that of providing an excellent communication tool. Good communication makes the maintenance task easier and improves reliability. A program is like a foreign language, and the more similar each programmer speaks, the easier it is to understand another's programs. Standards offer landmarks that can be easily recognised by an outsider. This reduces the chance of a minor change to one part having unforeseen consequences. It also makes the maintenance engineer's job less tedious if he (or she) can quickly grasp the structure and find the required section without wading through pages of muddled code. It is not uncommon for an organisation to spend twice as much on maintaining a piece of software as it cost to produce it, so anything that makes this task simpler and more reliable is valuable. A systematic, well documented approach to the design process, combined with a knowledge of the more common pitfalls, will help the designer to produce reliable code which performs its function well and can be enhanced to meet future needs.

By encouraging documentation, standards encourage the developers to explain how the final result was reached. Relating this to the CM model, we can say that this aspect of standards encourage the organisation to operate at least at level 3, the 'defined' level. This is particularly important for large projects where the development is carried out by a team and not by a single developer. Large projects are generally more prone to changes in the design team due to factors such as illness, resignations, transfers or career changes.

The case study was subjected to several reviews throughout its development. The design review activity was very helpful in identifying problems and was a major contributor in keeping the design focused on the primary target. An important observation made about design reviews is that they need to be thorough if they are to be useful. Often the management approach and the attitude of the team in general will determine how meaningful the results of this test are. If care is not taken, the atmosphere can easily develop into that of an examination or a battle

by the process would also be appreciated by management who will have a better visibility into the system. In practice, the design of accurate and stable requirements cannot be completed until users gain some experience with the proposed software system. This is contrary to traditional practices which relies on the assumption that designers can stabilise and freeze the requirements. Again we find that the model captures well the intrinsic approach to development, instead of opposing it.

f. As already mentioned, this approach promotes re-use of codes by building a library of tested classes and modules.

g. The process encompasses the hardware development while keeping the software development as much independent as possible. A parallel hardware/ software development (as compared to a sequential development) has less bottleneck stages.

Note that the process evolved during the case study and as such it has only been tested on a single application. The case study showed that the approach was suitable for a real-time embedded systems development, but for a confirmation of this, the approach will have to be tested on a number of different cases.

2.4 Contribution From Existing Development Processes and Strategies

This process has been influenced by a number of development processes and strategies. In particular:

- It borrows the idea of library of modules and the retry/redesign limit test from Stoyenko [2]

- Like the co-design approach of Chiodo [3] and Wolf [4], it processes the hardware and software elements simultaneously

- It borrows the idea of including the environment into the specifications from Zave [5]

- It accepts both uncertainty and change, like the prototyping approach

- Like Boehm's spiral model, it is risk-based and iterative in nature.

The benefits of this development are summarised below:

a. It fits the natural iterative problem-solving pattern. When reaching for a solution the human mind seldom holds to a strictly sequential process. Instead it often follows a similar iterative pattern. It is also a natural tendency of the developer of addressing those regions that are unclear or where he or she is not confident in, first. Being close to the intrinsic problem-solving pattern it means that it could be more easily adopted and applied by a wider range of practitioners, without necessitating a sustained training process.

b. It builds the confidence of the developer along the process. The high elements of risk associated with embedded systems means that the developer is usually under enormous pressures and has many reservations (even if not openly expressed). This can lead to an unhealthy environment. By giving the developer the freedom to jump straight to a particular problem area, the process relieves much of the psychological pressures and increases the chances of success. In other words the process is achieving its objective, which is to facilitate the development.

c. The design-coding-testing process can build the system according to increasing complexity, whereby separately developed and tested modules are integrated gradually. To the management, this means that the process naturally accommodate for partitioning of tasks to different teams. That is, several modules can be developed simultaneously. In addition, the separation of concern means that the programmer (or team) who has been allocated a particular task does not have to be conversant with all the details of the problem. For example, someone implementing the phase-locked-loop algorithm does not need to understand the principle of Space Vector Modulation. In terms of the object structure, he or she only needs to know the preconditions and the postconditions of the module.

d. Testing is performed early in the development process. According to Honka [1] this can largely contribute to the software quality and reliability.

e. A further advantage that the management could find in this process, is its ability to identify problems quickly. The customer, whose requirements are often unclear, would also become aware of the limitations earlier in the process. Compromises and alterations in the specifications could be affected at lower expenses. The small intermediate deliverables produced

the unit. Usually it will also contain power supply circuits, isolation circuits and additional supporting circuitry for the microcontroller (reset circuit, crystal oscillator, etc.). The hardware interface development will therefore typically involve the design of the circuitry, the design of the PCB and the testing of the PCB.

While the hardware platform is being developed, the modules developed on the software side can be tested on a simulator or an in-circuit emulator. The testing phase will reveal whether the designed module meets the space and time constraints, in addition to the functionality requirements. If the requirements are not met, the design has to be reviewed; i.e. the design-coding-testing loop is repeated. It is important to relate any change to the design in terms of the object-orientation models of the system (i.e. the logical/physical models and the static and dynamic models) so as to keep track of how the changes affect the other elements of the system. This phase of the process will also provide the time scheduling of the operations. This is particularly important when several functional partitions share one hardware unit. If after a pre-determined number of iterations through the design-coding-testing loop the requirements on the module are still not met then the problem has to reviewed at the functional specifications level. At this stage, if it is purely a space or timing problem, the hardware/software partitioning can be revised. Alternatively, the suitability of the technical strategies (e.g. SVM or sinusoidal PWM) has to be reviewed and if necessary a different approach has to be investigated. This approach is repeated for each module of the system, starting with the ones associated with the highest risk factors (and not with the ones sitting on top of a flow chart).

As the development of the hardware is completed, some of the modules can be tested directly on the hardware (via the ICE at first); especially those modules that cannot be tested without external physical input, like the A/D module. Finally the software is integrated to the hardware and the whole system is verified and validated. Success in meeting the requirements will mean the end of the development stage and will carry the system into the maintenance stage of its life-cycle. If however the requirements specifications are not met, then the problem is again referred to the functional specifications level, where the technical strategies and the hardware/software partitioning can be reviewed. This iterative process can be repeated a fixed number of times. After that, if the requirements are still not met, the problem has to be referred to the requirements analysis level. At this stage, the constraints on the design are declared to be too tight and have to be relaxed.

2.3 Characteristics of the Approach

That might involve a survey and some research before making the decision. One might argue that this would be the customer's responsibility, but to do so would be to assume that the customer knows which technique is best suited for a microcontroller system approach. Techniques used in digital electronics are not necessarily appropriate for software solutions. For example, sinusoidal PWM is a widely used technique for generating PWM pulses. It has been successfully implemented in digital electronics using op-amps circuits. However the same technique is not appropriate for a software implementation, because of the non-linear equations that have to be solved in the process; a constraint that is obviously not experienced in a hardware implementation. Therefore, ultimately the embedded systems developer will have to come from an engineering background, typically an electrical engineering background. He or she will also be expected to work closely with the customer and the technical experts in the particular domain. Following the decision about the technical strategies, always at the functional specifications level, the software/hardware partitioning of the tasks has to decided. This decision should be based on factors such as cost, component-count, time constraints and space limitations.

The next step is the analysis and design of the system. Those two phases are combined in figure 1 under the heading "object-orientation application". Strictly speaking the design can be carried out in any design methodology, structured or object-oriented. However in our case object-orientation appears to be particularly suitable. Object-orientation offers the ability to view the system at different levels of abstractions. Therefore we can start by viewing the system at a high level of abstraction and then once the abstractions have been defined, proceed to implement and test only those abstractions that contain the highest elements of risk. Being built in terms of abstractions and modules, such an approach will quickly generate a library of tested modules. Implementation will start by verifying if the module to be implemented has not been previously developed or if there is not a similar module that could be modified and adapted to the problem. If no similar module is found then the codes for it has to be written from scratch.

During the same time as the design and coding unfolds, the hardware interface with the rest of the system should be developed. The hardware interface is the part of the system (typically a printed-circuit board) that holds the programmable device and helps to integrate it with the rest of the system. For example, in the case of a three-phase inverter for a UPS, the hardware interface will have for tasks to ensure that the feedback signals from the inverter and the rest of the unit are compatible with what the microcontroller can accept and also it will ensure that the signals output from the microcontroller can communicate with the rest of

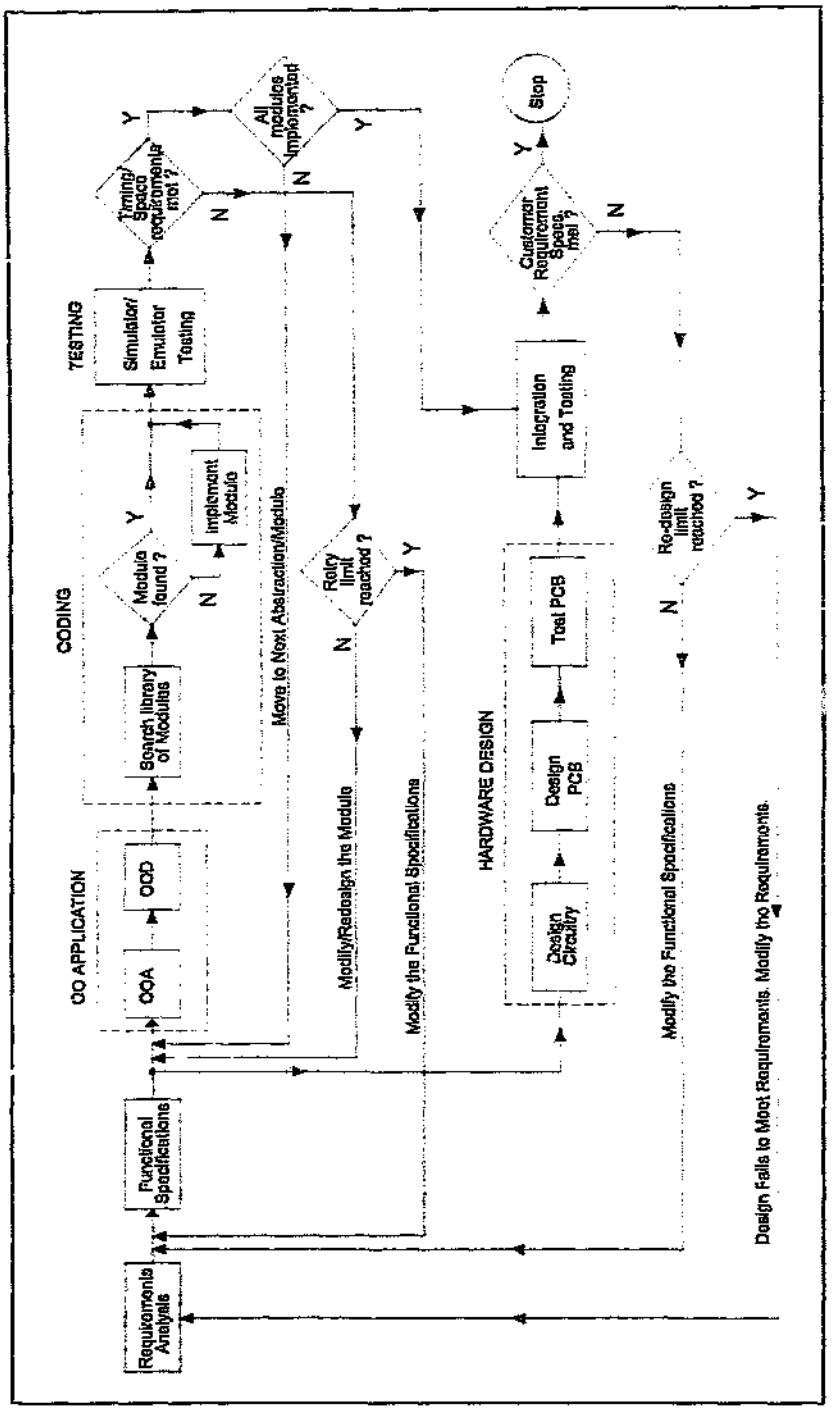


Figure 1: **The Risk-Based, Hardware-Software Co-design Approach for Embedded Systems Development**

- Risk factors
- Hardware interaction and the environment of the embedded system

Based on the work done on the case study, a general development approach to embedded systems became especially attractive. The process is described below.

2.2 The Development Approach

Having identified the main characteristics that are wanted of a development process for embedded systems, we can take the idea further to reveal more information about the process.

Risk factors, as we see it, have to be tackled early in the design in order to avoid wasting efforts on inappropriate solutions. It should also be remembered that not all the risk elements of a design are known prior to the development. Those considerations lead us to think that the process has to be iterative in nature instead of sequential. It has to offer the freedom to focus on specific aspects of the problem at any stage of the process, meaning it has to build on loosely coupled modules as in OOD.

The strong hardware interaction in embedded systems means that the hardware interface has to be developed along with the software. Building the hardware at the same time also helps in the decision on the hardware/software partitioning of the tasks and allows for possible adjustment of this partitioning as the system is developed.

The complete process is outlined in figure 1. It starts with a requirement analysis, which is typically a document drawn by the customer about what the system should do and how it should behave.

In order to relate the customer specifications to the software developers, a functional specifications document should be developed. This document should include a description of the environment in which the system is to operate. At this stage also, decisions should be made on the technical strategies to be adopted if applicable. For example in the development of a PWM inverter control system, one has to decide upon the method (SVM, sinusoidal PWM, optimal technique, etc.) of generating the PWM pulses right at the functional specifications stage.

language.

6.2.2 Use of Subroutines

Optimisation is not limited to the compilation activity or the way in which data is stored. The program itself can be written for speed or to reduce the amount of memory it occupies. As often experienced though, speed and space are conflicting objectives. The use of subroutines is a well-known way of reducing program size One section of code is called from several places in the program. On the other hand, code will run faster if it does not have to jump to a subroutine, pass parameters and jump back to the calling program.

6.2.3 Reducing Execution Time in Loops

An example where execution time reduction can be achieved - at the expense of lengthening the code - is the execution of loops. If an operation has to be performed several times, it will run faster if there is no need to update a loop counter and test the result on each pass. If the operation is fairly short and is not repeated too many times, it can be convenient to simply repeat the code. The proportion of time spent on loop control increases as the length of the loop decreases, so the potential savings are greater. With this approach though, the number of passes through the loop must be fixed when the program is written. This is often the case when, for example, reading a set of values or a multi byte number, one byte at a time from an external source. Obviously this approach will not suit every application.

6.2.4 Storing Pre-Calculated Values

Sometimes savings can be made by storing partial calculations which are used by other parts of the program. For example, if several routines in sequence each multiply the same numbers together as part of their individual calculations, the multiplications could be performed by one routine and the result made available to the rest. This would take less time than calling a subroutine to recalculate the value each time, especially if multiplication is a software function rather than part of the instruction set. Look-up-tables are based on this principle. They store pre-calculated intermediate results, which can then be adjusted or sized at need. Notice also that assembly programmers and those using some versions of C have the option of holding immediate results in registers.

As an example consider the following C program fragment:

technique is to build a heap or table of memory blocks along with an associated process ID for the owner of the memory block. This data structure is then periodically checked to see if memory has been allocated to a process which no longer exists. If this is the case, the memory can be released.

Because of the overhead involved, this method should not be implemented in high frequency cycles, and ideally garbage collection should be performed as a background activity or not performed at all [21].

6.2 Optimisation Strategies

One of the feature of the risk-based process presented in section 2 above, is the design-coding-testing iterative loop. Besides meeting the functional specifications, the modules generated by this loop has to meet the time and space constraints. If they do not meet those constraints the module or abstraction undergoes another turn round the loop. The purpose of going round the loop again, in most cases, would be to optimise the module performance according to the time and space constraints. As a last resort to meet the constraints, it may be necessary to write the program in assembly language to make optimum use of the resources available. However, experience has shown that the use of assembly language can often be avoided by making more efficient use of the high level language and by the use of skilful algorithms. Some of those techniques and observations are exposed below.

6.2.1 Code Generation

Of similar importance to the written code is the way in which the compiler maps the code into machine language. The output of the compiler can be affected by optimisation for speed, memory and register usage, jumps, etc. which can lead to inefficient code and timing problems. Thus it is imperative that real-time programmers be masters of their compilers. When using high level language, the compilation of the code should be optimised whenever possible. Some compilers, notably those for C, offer a choice of levels of optimisation. The programmer can specify the extent to which the compiler should rearrange the code to give the best overall performance. Moving from a lower to a higher optimisation level, the compiled codes become more compact and run faster. However the straightforward mapping from the high level language to assembler is lost, as the compiler eliminates some of the codes for the purpose of optimisation. The assembler code becomes more difficult to follow. Nonetheless, this need not be a disadvantage if the testing tools (simulator, ICE, etc.) support the high-level

than when all the parameters are passed using call-by-value, since indirect instructions are needed for any calculations involving the variables passed.

The decision to use one method of parameter passing or the other represents the tradeoffs between accepted software engineering methodology and speed. Using parameter lists is advantageous because the interfaces between the modules are clearly defined. Furthermore, in languages like Ada and Modula-2, whether the parameters are to be input, output, or both are specified. Unfortunately, parameter lists can get long, and often interrupts are disabled during parameter passing (to avoid mixing values of different sequences in time). This is effectively a series of LOAD and STORE instructions (see discussion about management of the stack above), thus potentially increasing interrupt latency in a place where it is least anticipated and in a way that is hard to spot. As was experienced in the case study, contrary to good software engineering practice [17], the realities of timing constraints in real-time embedded systems often leads to the frequent use of global variables. Global variables, whenever used should be clearly documented.

6.1.8 Dynamic Allocation

The ability to dynamically allocate memory is important in the construction and maintenance of stacks needed by the real-time operating system. While dynamic allocation is time-consuming, it is often necessary. Languages that do not allow for dynamic allocation of memory require a stack of fixed size. While it is faster to maintain a fixed size stack, flexibility is sacrificed, and the maximum size of the stack must be known a priori.

6.1.9 Real-Time Garbage Collection

In a memory-management context, garbage is memory that has been allocated but is r longer being used by a task (that is, the task has abandoned it). Garbage can accumulate due to tasks terminating abnormally without releasing memory resources [18], in object-oriented systems, and as a normal byproduct of nonprocedural languages [19][20].

In C, for example, if memory is allocated using the malloc procedure, and the pointer for that memory block is lost, then that block cannot be used or properly freed. The same situation can occur in Pascal, when records created with the new statement are not properly disposed of

The goal of garbage collection algorithms is to identify and free garbage. One

Languages pass parameters between procedures using either memory, registers or the stack. Languages like C often use stack based parameter passing. Each parameter consumes memory on available stack. Sometimes there is a trade-off between stack space/nested depth and whether or not to use a global variable.

In addition, high level languages also use the stack for temporary storage and expression evaluation, as well as saving register based variables between procedure calls. This impacts on the available stack size; thus in working out stack requirements, this should be added to the maximum nested procedure depth.

6.1.6 A/D Conversion

When performing A/D conversion one must keep in mind the Nyquist rate. In order to convert an analogue signal into a digital form without loss of information, samples of the analogue signal must be taken at twice the rate of the highest frequency component of the signal. Thus, a signal at 50 Hz must be sampled at 100 times per second. This implies that software tasks serving the A/D circuitry must run at least at the same rate, or risk losing information.

6.1.7 Parameter Passing

There are several methods of parameter passing, including the use of global variables, call-by-value and call-by-reference. When choosing a particular method one has to bear in mind the implications upon memory usage and execution time.

Global variables are those within the scope of all modules of the software system. This usually means that references to these variables can be made in direct mode, and thus are faster than references to variables passed via parameter lists. Global variables, although they introduce no subtle timing problems, can be dangerous in that access to such variables may be made by unauthorised modules, thus introducing bugs that are very hard to find.

Parameters which are passed by using *call-by-value* are typically copied onto a stack at run-time, at considerable execution time cost.

In *call-by-reference*, supported by C++, the address of the parameter is passed in the calling routine to the called procedure so that it can be altered there. Execution of a procedure using call-by-reference parameters usually takes longer

such as timers, counters and input/output ports should be decided at an early stage in the design. If they have to be shared between modules, the rules governing their use should be carefully defined. If two modules try to use the same counter simultaneously for different purposes the program will fail, even if the individual modules pass their tests. Resource conflicts are more likely to occur when programming in assembly language or when programming partly in assembly language and partly in a high-level language.

6.1.4 Interrupts

Interrupts can occasionally cause data corruption and condition flags. As a strong precaution it might be necessary to disable the interrupts when critical calculations are being performed. Interrupts should not be disabled for too long or else their very purpose is defeated. Especially when programming in assembly language, where built in subroutines do not exist for saving register contents and flag values on the stack , it is important that those aspects are "manually" implemented and checked to ensure that the interrupt routine restores the microcontroller to a tidy condition before they terminate.

6.1.5 Stack Space Requirements

Embedded systems generally have minimal RAM available. This means that special attention must be made to the factors that affect the free space on the stack. In particular the following aspects of the running software should be watched closely:

- nested procedure calls

  Each procedure call requires the placement of the program counter on the stack. The obvious implication of this is that there will be a limit on the procedure depth which can be implemented.

- Interrupt Service Routines, register saves, context switching

  The number of registers saved on interrupts differs between processors. Nested interrupts will rapidly consume large amounts of stack space. Those space requirements should be added to that of the nested procedure calls and temporary register variables.

- parameter passing between functions and procedures

Range checking should be introduced to improve the reliability of the software. Owing to time constraints imposed on real-time embedded systems, it is often not possible to provide range checking on all variables each time they are used. The solution in this case is to limit range checking only on data entering the system and on critical variables. Two main type of range checking are identified by Adamson [7], static check and dynamic check. Static check involves merely checking whether the value of a particular data falls between a predetermined range, e.g. check if the frequency is between 49 Hz and 51 Hz. Dynamic check often verifies the value of a variable against its previous value, e.g. check that the sensed speed of a car does not change by more that 15 Km per hour between samples, because of the inertia of the car, etc. The dynamic test is a more strenuous test than the static test but it requires more knowledge of the possible behaviour of the system being controlled and a little more processing time.

6.1.2 Hysteresis

The effects of noise can be considerably reduced through range checking and through the use of filters, discussed in section 7 below. However, not all data which cause undesirable fluctuations of the variables values are results of noise. In some cases we might want to suppress the occurrence of genuine and persistent values, especially those of small magnitude. As an example consider the case of a numerical display. If the value happens to be drifting around a major transitional level, such as between 99 and 100, all the digits could change. This is very inconvenient and distracting to the user, and has made some digital displays unpopular. In such a case, filtering will not be helpful because filtering is only concerned with data fluctuations occurring over short period of times; the allowable time interval being controlled by the time constant of the filter itself. The filter is however helpless again persistent signals, even if they are of small magnitudes. One way round this problem is to introduce hysteresis. The effect of this is that small changes are suppressed, but large changes cause an immediate response. It works by setting a band of values about the current level. Changes within this band do not change the output. As soon as the input moves outside the band, the output changes and the band is made to lie around the new value. This, however, leads to a reduction in the accuracy of the displayed figure, but this need only be small. The uncertainty caused by a flickering display can be of a similar magnitude.

6.1.3 Resource Conflicts

Resource conflicts should be checked for and avoided. The use of fixed resources

6 Some Practical Issues and Techniques About Programming for Embedded Systems

Many of the established techniques for writing effective and maintainable computer programs apply equally to large data processing designs and small embedded systems. Programs written in a clear, logical style will be easier to debug than those which are a random collection of code fragments, whatever the applications. In addition to these general considerations, the distinctive features of small real-time systems make special demands on the way in which the program is written. The choice of implementation language has already been discussed (see section 8, SDE 103 and section 5 above). Here we concentrate on some of the practical programming considerations of interest to the embedded systems developer. Many of them evolved during the case project.

6.1 General Considerations

General programming techniques and practices are discussed below. They address some of the practical issues that confront the embedded systems developer.

6.1.1 Preventing the Effects of Noise

The effects of interference can be reduced by careful hardware design, but software may also have to allow for this. Noise on an input line can give a sudden wrong value. If it affects a control or address line, the program may shoot off into some undefined state. As for push buttons and switches, the only safe rule is to assume that someone will use every combination in the wrong way at the wrong time. The software needs to perform rigorous range checking on all critical variables and must be robust enough to stand up to all these effects. The error recovery methods listed in section 4.2 of SDE 103 are particularly relevant in that instance.

Out of range values can arise from several sources, some of which are:

- faulty sensors
- noisy environment
- from the codes itself as a result of coding error
- unpredicted values, i.e. when the values are genuine but were not expected

when calling methods or passing arguments. If the class hierarchy changes then the programmer must manually reevaluate the traversals.

- Error Protection: The programmer must ensure that all methods are included in a dispatch method or dispatch structure. The programmer must initialise a new object with its class. the programmer must avoid accessing internal attributes of other classes.

- Maintainability: If changes are made to the object declarations, the programmer must determine their effects on the code and modify the code accordingly. An object-oriented language provides and enforces modularity of classes that prevents changes from propagating through the entire program. The non-object-oriented programmer requires discipline to achieve similar modularity without the help of the language.

c.	Allocating objects	Use a struct variable and initialise it at compile time (When an object is no longer needed it must be deallocated using the C free function)
d.	Inheritance structure	Embed the declaration for the superclass as the first part of each subclass declaration. The first field of each struct is a pointer to a class descriptor object shared by all direct instances of a class. The class descriptor object is a struct containing the class attributes, including the name of the class (optional) and the methods for the class. Note: Inheritance structures are best avoided if possible when programming in a non-OOL.
e.	Polymorphism	Again this feature is best avoided when programming in a non-OOL. The most general approach, however, is to define a class descriptor object for each class containing a pointer to the method function for each operation visible from the class, including inherited operations.
f.	Association Relationships	Map the associations into pointers or implement them directly as association container objects.
g.	Concurrency	Could be simulated by the use of interrupts and synchronisation techniques such as the one used in the case example. Note: Most languages do not explicitly support concurrency.
h.	Encapsulation	Can be implemented by using the following steps: - avoid the use of global variables - Package the methods for each class in a separate file. - Treat objects of other classes as type "void"

Although the above mapping plan does provide a path for implementing OO in non-OOLs, we find that the resulting codes are often clumsy and longer. A strict programming discipline could compensate for the loss of clarity and simplicity but would not make up for the considerable consumption of execution time and memory usage that such an implementation would require. For example, encapsulation, which is one of the major themes of object-oriented programming, requires separate file for the methods of each class. Most embedded systems would not be able to support such generous use of memory, not to mention the execution time overheads. In essence, the shortcomings of non-OOL appear in their inability or reduced ability to match the following OOL features [12]:

- Expressiveness: The non-object-oriented programmer must map object-oriented operations, such as method calling or subclass declarations, into explicit operations which are often clumsy.

- Convenience: The programmer must manually traverse the class hierarchy

Irrespective of the final implementation language chosen. A clear and explicit representation can be beneficial to software maintenance and reliability (both important factors in the development of embedded systems). During the lifetime of an embedded system it is often necessary to modify the program to include new features or change its response. This tasks is made much easier and less prone to fatal mistakes, if the original design is laid out clearly, with interaction between different parts of the program displayed in a well-defined way as with class diagrams.

Lastly, when applying OO techniques, it is important to realise that its application in embedded systems carries a risk. The main risk here is that of getting carried away in the Object-Orientation rules and structures and to make conformance to the methodology, rather than the product, the main goal. It does not make sense to struggle to implement a complex inheritance link for an 8-bit microcontroller just for the purpose of satisfying the methodology. The developer should keep in mind that the methodology is only a tool to help build the product.

5.2 Available Methods of Writing OO code for Microcontrollers that do not Support OOLs

Presently, only a few microcontrollers support OOLs. This kind of feature is mostly limited to the embedded PC type of microcontrollers, like the Intel 80386 EX. Two main avenues are available for circumventing the language limitation for the rest of the microcontrollers. One is by making use of software translators, e.g. C++ to C translators. However those products are not common and their use are not widespread. Two such language development tools are *ofront* [11], which runs on a UNIX platform and PCYACC from Abraxas, Inc [16].. The second alternative is to simply write the program in a non-OOL. This exercise is not straightforward and demands some special efforts and a strict programming discipline. One approach presented by Rumbaugh et. al. [12] is summarised in table 1 below. It indicates how to implement some of the OO functionalities using a non-OOL, taking C as an example.

Table 1: Mapping OO Features to a non-OOL

	OO Feature	Implementation in a non-OOL (C)
a.	Classes	Use a record structure (struct)
b	Passing arguments to methods	Use the pointer mechanism

- As a result of more lines of codes and more complicated data structures, OOLs tend to increase the execution time

- Testing the software is more difficult since many of the common development tools like the ICEs do not support OOLs

- A lack of experience in the programming language. OOLs are generally very complicated and their use are difficult to learn.

- A lack of documentation, examples, etc. on the applications of OO in real-time embedded systems. Existing documentation tends to limit the study to the analysis and design phases. The actual implementation and coding, together with the results are rarely produced.

- Changing to OO methods is not just moving to a different language. It is a completely new way of looking at the problem domain. The conceptual leap that is demanded certainly requires changes in old practices of the whole design process; a cost that not many organisations are ready to bear at this stage.

In terms of the available object-oriented languages, the C++ language seems to be the only OOL that is worth consideration for embedded implementation of embedded software, despite the fact that it is a difficult language to learn. The other OOLs do not seem to be suitable or mature enough to be widely used by the embedded systems industry.

Despite of all the obstacles listed above, the promise of object-oriented programming languages in real-time applications lies in clearer design and better maintainability. The shortcomings of object-oriented languages in real-time will be overcome. For example, hybrid languages that reduces the amount of late binding required are already being used [14], and better methods for automatic storage management (as well as faster microcontrollers) will make the use of interpreted object-oriented languages like Smalltalk viable for use in real-time. One thing is certain, we can look forward to the increased use of object-orientation and OOLs in real-time applications.

Still, even before those new developments take place, OO can be beneficial to the embedded systems developer. As was demonstrated in the case study, OOA and OOD can provide considerable insight and clarity in embedded systems,

convenient means of translating the specifications into microprocessor instructions. However in the light of the case study, it would appear that for real-time embedded systems, the language issue is too much important to be treated as lightly as in other software applications.

Object-oriented languages provide many of the features necessary to encourage good software engineering technique. For example, polymorphism allows the programmer to create a single function that operates on different objects depending on the type of object involved; e.g. the function "move" would act differently on a circle than on a rectangle, but the function of moving is the same in each case. Object inheritance allows the programmer to define new objects in terms of other objects so that the new objects can inherit the other's characteristics. For example, an object of type "dog" would have at least the same characteristics as an object of type "animals".

For these benefits though, an execution time penalty is often experienced. For example, in one study, code written in objective-C, another object-oriented variant of C, was 43% slower than code written in conventional C [14]. In addition, programs written in the (interpreted) language Smalltalk, are known to be approximately 5 to 10 times slower than those written in conventional C [15].

The execution time penalty in object oriented languages, may be due, in part, to their relative immaturity. But in compiled object-oriented languages, it seems clear that the requirements for late binding (run-time as opposed to link-time) necessitated by the polymorphic and inheritance structures are considerable delay factors.

In both interpreted and compiled object-oriented languages, additional delays result from automatic storage management (including garbage collection - see section 4 below) needed by objects over their life-time. For example, Ingalls [15] mentions that for programs written in Smalltalk, 10% of the execution time is spent in automatic storage management.

We can summarise the obstacles for the use of OOLs and OO in general in Embedded Systems as follows:

- No compiler for the OOL is available for the particular microcontroller; e.g. no C++ compiler is available for the H8/3048 microcontroller

- OOLs tend to make a larger use of memory

systems.

Automatic code generating programs will emerge, further encouraging a shift of emphasis from the coding stage to the design stage. With these, the software designer will practically not write code at all, but will specify the required system in a formal design language or set of diagrams. To some extent this move is already being observed with tools such as Rational Rose/C++. Although at this stage, the generated code is often of a rudimentary nature, the trend is clearly marked. Of course just like moving from assembly language programming to high-level programming means losing part of the in-depth control over the microcontroller functions, using automatic code generators will move the programmer further away from the hardware structure. In that respect such tools might take more time to become of interest to the embedded systems developer, but just like with object-orientation, they will eventually be adapted to the embedded systems
requirements.

As the need for clarity, maintainability and reuse imposes itself in the embedded province more and more developers will be looking at the possibility of making use of object-orientation. The trend will encourage more development in that field, which will lead object-orientation techniques, languages and tools specially architectured for the needs of the embedded systems development. Already there are examples of this trend. For example, the design and programming language Embedded Eiffel (ee) is developed for real-time applications. The language resembles the object-oriented language Eiffel as well as the synchronous languages Esterel and Argos [29]. Along the same line, there will be more development in concurrent systems, as the need for hard real-time systems increases. Here as well OO will play a role because of its ability to represent concurrent systems. Development has already started in this direction too. As an example we find new languages like Compositional C++ (CC++) [30], which is a parallel programming language based on the sequential language C++. The extensions in CC++ enable one to construct reliable parallel libraries that integrate a range of parallel programming paradigms, such as task parallel and data parallel programming styles.

encourage is a shift of the creative emphasis to an earlier stage of the development process.

The wrong conception that people have about embedded software development has encouraged one-man projects, with emphasis laid on the product and on short term targets. Indeed it is not rare to see whole embedded control systems being fully developed by a single individual who is as well responsible for the hardware platform development. Such approach puts enormous pressures on the developer, with the result that often documentation and good practices are ignored. For the project itself that also means that the result is limited to the capabilities of a single individual. This is typical of organisations operating at level 1 of the CMM (see section 4.3 of SDE 102), where failures and shortcomings of a project are associated with individuals rather than to the process itself.

As the embedded software industry matures one should expect to see more software being written by teams rather than individuals. Team development encourages information sharing and requires a strict discipline on the way documentation is handled.

Along the same line, and closely related to the decrease in emphasis on the coding process mentioned above, we can expect a move towards "designer software" where the skills and reputation of particular software engineers become selling points of products. The skills of software designers will focus more on specifying a system to meet the client's needs, and less on the mechanics of how the code will be implemented. In terms of maturity, this will move software development closer to other engineering disciplines. An architect is not expected to build the product he designs, the same for a car body designer. This allows them to concentrate their efforts on a limited aspect of the problem. Even though software is a less tangible output than the product of other engineering fields, there is no inherent reasons why a similar move should not be observe in the field of embedded systems and software in general.

There will be a continuing growth in the power and the functionalities of support systems for program development. More sophisticated tools will surface. In-circuit emulators will become more powerful and more accurate. Already some in-circuit emulators allow high-level debugging. This will encourage the move away from assembler language. A further development will be the introduction of tools that can emulate parallel processors or several interacting microcontrollers of different types. This will promote the use of multiple microcontrollers in embedded

Future developments in microcontroller hardware should allow more features to be added for the same price. Development along this line would also reduce the cost and size of a basic programmable controller, thus widening the range of applications of microcontroller systems. Already microcontrollers are finding their way into the home security systems, central heating controllers, video controllers and even cameras.

Embedded systems will also find their use in the growing concern for environmental care. Pollution level will have to be watched closely and more precisely, as strict quality standards are enforced. This might involve monitoring of several parameters and taking decisions based on the values observed. The control system might have to operate under severe environmental conditions, thus lending itself well for embedded systems techniques. As an example, the tighter environmental controls on car exhaust emissions will increase the need for closer control engines. This will almost certainly lead to an increase in the use of programmed engine management systems. Automobile systems, as mentioned in section 2.1 of SDE 103, already form one of the growing sectors of the electronic market, and an increasing proportion of them involve programmable devices.

Always in the field of programmable product development, many of the digital controllers will be incorporated into Application Specific Integrated Circuits, or (ASICs). Already several well-known microcontrollers have been included in the libraries of functional blocks available to integrated circuit designers. The software may be included in ROM on the same chip. Of course those systems will almost certainly be difficult to debug and maintain once they have gone in production, which is why their use should be limited to the most common applications which are not likely to change frequently.

In future less emphasis and effort is expected to be spent on coding. Not only will coding evolve more naturally from the design process, but less code will even be written individually. Instead there will be increasing use of predefined modules which will be combined to produce new systems. This is analogous to other areas of engineering where production is limited to the assembly of parts and where components that fit together, like a nut and a bolt, are produced together instead of being produced separately by different manufacturers. This trend in the software industry will promote a better compatibility and uniformity in the design of software, thus reducing much of its subjectiveness. To some extent this already happens. Some software houses build tailor made versions of their products by linking a subset of a large library of functions. Such development will not and cannot remove the creative aspect of software development, but what it will

8 Future Developments

Embedded systems is a growing field. More and more applications of microcontrollers are being investigated. Consequently it is a dynamic field where we can expect many developments in the future. Some of those potential changes are discussed below.

The computing power and additional features of the microcontroller is an area particularly prone to evolution. As technology develops, costs fall and expectations rise. In future we can expect more powerful controllers to be introduced, in parallel with the spread of simpler microcontrollers to a wider range of products. This in turn would affect the implementation platform. More applications would be developed on 32-bit devices for instance. This would not necessarily mean the end of the 16-bit or 8-bit devices (especially in household appliances) but it is just that newer applications would require the processing and data-handling speed, as well as the address space, of 32-bit engines. In some cases it is not merely raw power that will drive the choice of 32-bit devices, but also the need for the programmer-friendliness and documentability that high-level languages can provide. The trend to program in object-oriented languages will be another factor that will encourage the use of more powerful controllers.

As the size and price of microprocessors continue to fall, there is no reason why designs should be restricted to make economic use of the CPU or to share the processing power between several applications. If a system involves three distinct tasks (like rectification, inversion and display), there is no reason why they should not be run on three separate microcontrollers. The practice is already observed in some products - even the small printers used with many desk top computers sometimes contain three microcontrollers - and can be expected to become more common in the future. With the economic need to concentrate processing power in one place greatly reduced, control could be distributed around the system, thus improving the modularity of the overall design.

With multiple processors, there is no need for the software to be designed as a set of procedures following each other in a defined sequence. Instead several tasks can run at the same time, with the full facilities of their own controllers. This will usually change the design drastically for it would mean that instead of being restricted to the software model, methodologies like object-orientation, could be built into the distribution of controllers as well.

For example, if the program resides in locations 0 to 2038, the emulator or analyser can be configured to trigger if the microcontroller performs an opcode fetch from any address outside the range 0- 2038. If the program crashes and tries to fetch an instruction from an address outside of the normal range, the emulator will trigger and provide a trace of what happened before the crash occurred [28].

7.5 Real-Time Monitoring

Ultimately the developer is interested about the real-time performance of the system. This can easily be performed by the emulator. More difficult is to monitor changes in real-time. The variables values and registers contents are only updated when the emulator is stopped. One solution is to use the emulator to interrupt the program at a regular interval. At each of those interruption a small routine is to be executed that displays the variables values and the registers contents. While the emulator updates the screen with the latest sample, the target program continues to run. This method does have an impact on the execution time or processing speed of the software, although the overhead is small and can be negligible according to the application.

7.6 About Breakpoints

Breakpoint-based debugging is most useful when trying to debug the basic logic flow of the program. In effect it is a good means of crude validation. It is a very cost-effective way of obtaining confidence in the general algorithm of the software before making further investment in the project. However, since breakpoints interrupt the flow of the program and alter the system timing, they drastically change the real-time behaviour of the program. The fact that they offer a microscopic view of the program operation, can obscure the big picture. In other words their use should not be exaggerated and should preferably be limited to the initial stages of dynamic testing.

7.7 Preventing Interferences - Hardware Solutions

Inputs and outputs can be filtered to smooth out sudden changes. For example, the level of fuel in a car's petrol tank will vary as the car drives over bumps and around corners. The driver is not interested in this variation, but does need to know the overall change as the fuel is used up. Noise can also arise at the component-level. Modern chips are so fast they cause terrible undershoot and crosstalk problems. An emulator, in particular, tries to minimise propagation

want it to work. In other words, in the case of embedded systems the software engineer needs to have a sound understanding of the target application, the microcontroller architecture and the software development tools. This necessity to be very close to the hardware platform is characteristic to embedded systems projects.

Unlike other software developments, in embedded systems, the use of high-level languages does not exempt the developer from the need to have an in-depth knowledge of the architecture of the microcontroller. To some extent, even a knowledge of the assembly language is as well necessary. The necessity for such knowledge becomes obvious when having to develop for size and speed, and when debugging. On that subject, it is interesting to point out one advantage of assembly language that is often overlooked. When programming in assembler one is bound to master the underlying architecture of the chip along the way.

7.3 About In-Circuit Emulators

When choosing an emulator is often best to choose one that is developed by the producer of the microcontroller itself, if it is available. Since the producers are the ones that should know the architecture of the microcontroller best, they are often able to provide more accurate emulation of the device. One should be aware that not all in-circuit emulators give an exact emulation of the circuit they replace. Voltage and timing margins may not be identical to the integrated circuit. This can be a problem if the application's hardware design is marginal or if the nature of the application itself can only allow marginal variations (like the control of an inverter). Obviously timing is a more important consideration in real-time applications. If the emulator cannot use the same clock frequency as the final product, or variations are substantial, the execution time of all the instructions will be affected. A program which works perfectly in the emulator may run into timing problems in the real circuit or vice-versa.

7.4 Finding An Intermittent Bug

A very inconvenient bug to find is one where the program functions perfectly most of the time, but occasionally loses control and crashes. By defining a proper triggering specification for the ICE to detect the crash, it is possible to find out what led to the crash. Since program crashes often result in opcode fetch cycles from invalid locations, the bug can simply be found by triggering on a fetch cycle outside the normal program address range.

- How involved is the manufacturer in embedded systems; i.e. is he also a supplier of general peripherals (A/D, memory, voltage regulator, etc.)?. At times the manufacturer can also be a supplier of other products than are often associated with microcontrollers; e.g. Hitachi is also a supplier of LCDs and power electronics switching devices. When looking at microcontroller products in general, always enquire on the delivery lead-time as well. Because of the specialised nature of some of the products they are often not kept in stock.

- Does the manufacturer support OTPs (one-time programmable), windowed devices, mask parts?

One disadvantage of using single-chip devices is that the performance of algorithms implemented using the device is very seldom comparable to that of the similar in-built features; e.g. the in-built PWM feature in some of the micro-controllers can be driven at very high switching frequencies (20kHz and higher) but if one writes one's own program to generate the PWM pulses, one would hardly be able to go above 6kHz, using the very same microcontrollers. In other words the limitations of the microcontrollers (clock speed, bus sizes, register sizes, etc.) do not allow the possibility to emulate the same performance as the in-built features, by using different techniques. This disparity is seldom a problem, since one is often more concerned with what the device can give rather than the underlying design principles, but every now and then, one will come with new ideas and algorithms that will clash with the existing features of the controller. When this happens, one is often contrived to adapt to the existing features. In those instances the microcontroller acts as a limiting agent to the design and creativity process.

7.2 The Need for Understanding the Hardware Environment

Software designs for embedded systems is more closely related to the hardware than is the case in larger data processing applications. This was already established in SDE 103. Ultimately this factor affects the programmer as well.

Understanding the technology fundamentals from an implementation perspective is important. The engineering problems that need to be solved to deliver an embedded system are often unique to the application and may not have been anticipated by a tool developer. Ultimately, understanding why something works the way it does is the key to understanding how to make it work the way you

7 Some General Practical Considerations for Embedded Systems Developments

Apart form the programming issues, several other practical matters were faced during the development of the case study. Some of those concerns are discussed below.

7.1 Choosing the Microcontroller

The choice of the microcontroller is often critical. The associated development tools are so expensive that one cannot usually afford to come back on the choice. The same apply for the product range. Although most development tools are compatible with a wide range of devices they do have limitation and obviously an in-circuit emulator for an 8-bit family will not support the 16-bit family. Besides investigating who else is using these devices (and how many they are using) the following considerations are generally helpful:

- Is the microcontroller company represented locally and how competent are the local agent in the technical field? The use of e-mail and other fast communication means tend to reduce the importance of this factor, but still, a quick demonstration on the spot can save a lot of unproductive hours and frustrations. In any case, the close presence of competent support increases considerably one's confidence in embarking into complex projects with the device. The value of competent support is often only realised when things are not working properly.

- What development tools are available and how much do they cost? When investigating this issue one should ascertain who will be providing support for those products. This is particularly important when the development tools (including the software) are not produced by the microcontroller company.

- What sort of documentation is available, in particular reference manuals, application notes (both programming example and hardware implementation example) and books. One should also verify whether the examples provided have actually been tested and if so are the conditions under which this was done clearly laid (in particular are the compiler and linker options stipulated).

variables which are only updated each time the slower cycle is completed. When used in combinations with timer interrupts it can simulate a parallel processing system. The double-buffering method was demonstrated in the case study when the *main cycle* was synchronised to the *interrupt cycle*. Notice that this method makes use of global variables.

6.3 Summary

To summarise, the amount of processor time spent on calculations can often be reduced by tailoring the design of the application, while still programming in a high-level language. Redundant calculations can be removed, and others combined to produce the desired result without unnecessary intermediate steps. Pre-calculation of constants can save time during the program's execution, and a suitable choice of constants, where this possible, can replace slow division and multiplication routines with faster shifts. However, invariably, we note that those techniques, aimed at producing a more efficient system in terms of speed, have detrimental effects upon the program structure itself. As a result of keeping the high-level functions as basic as possible, the program becomes lengthy and more difficult to read, with the more serious effect of decreasing the ease of maintenance. Ease of maintenance, it will be remembered was one of the prime motivation of using high level languages in the first place. Such factors emphasise the need for good documentation and a quality management system.

and to optimise the software in general. Often those techniques will be enough to meet the time constraints imposed on embedded systems. However, depending upon the constraints those techniques might only bring negligible improvement to the system. Those are the cases where execution times need to be halved or reduced to a large proportion, otherwise the system will simply be of little or no value. In such instances one needs more powerful techniques that operate at the structural level of the program algorithm. Their applications might even necessitate changes in the design algorithm.

Once such technique consists in rearranging the sequence of events [23]. The waiting time for devices can at times be utilised more efficiently by rearranging the sequence of events in an algorithm. Consider a task which performs the following functions.

a. request a byte from a device
b. wait for the byte to be available
c. read the byte
d. yield to let another task run

The task spends most of its time waiting for the byte to become available. By re-arranging the sequence, it is possible to significantly reduce or eliminate the waiting time, thus improving the system performance, and giving more processor time for other tasks. The task is re-coded as,

a. wait for the byte to be available
b. read the byte
c. request next byte
d. yield to let another task run

Now, whilst the processor is running another task, the device is acting upon he request to get the next byte. By the time the task is re-activated, this byte will be available. It does require an initial priming for the first request.

A method that particularly proved to be efficient in the case study was that of double-buffering [22]. Double-buffering or ping-pong buffering is a technique used when time relative (also called synchronous or correlated) data needs to be transferred between cycles of different rates, or when a full set of data is needed by one process but can only be supplied slowly by another process. It is a simple method which consists of making use of buffered (or dummy) variables in the slower cycle. The faster cycle are fed with the genuine counterparts of the

with multi-byte numbers. Shifts can be continued across bytes boundaries by using the overflow or carry flag to hold the intermediate bit. The result of this is that multiplication or division by powers of two can be replaced by the appropriate number of shifts. There is no need to shift by more than eight places even when large numbers are used, since a shift of eight leaves each byte except the last with the contents of its predecessor. The same effect can be obtained by leaving the data in place but reassigning the mapping of bytes. The problem here is that this technique is limited to operations by powers of two. In addition, to take advantage of the byte structure (like when shifting by more than ten places) requires including more control into the program. This inevitably increases the complexity of the program.

6.2.8 Scaled Arithmetic

Due to their large consumption of time, floating point operations cannot be afforded in real-time embedded systems. Yet they are often necessary. On the other hand, integer operations are typically faster than float operations, for most microcontrollers. A solution then is to multiply integers by a scale factor to simulate floating point operations. This solution was one of the first methods for implementing real-number operations on computer systems [22]. Notice that to preserve as much of the accuracy of the float numbers, at times long integers have to be used instead of short integers. This increases the execution time but is still acceptable since operations on long variables are generally faster than operations on float variables. This technique was extensively used in the case study to improve the switching speed.

6.2.9 Typing Considerations

Strongly typed languages force the programmer to be precise about the way data is to be handled. They prohibit the mixing of different types in operations and assignments. With this in mind, one should be careful with languages such as C, that are typed but do not prohibit mixing of types in operations. They can cause rounding and truncation problems. In addition, C generally performs calculation at the type which has the highest storage complexity. That is, it "promotes" variables to the highest type necessary. This can generate "unseen" and time-consuming instructions to perform the promotion.

6.2.8 More Advanced Strategies

The above techniques were simple methods of cutting down the execution time

```
x = y + a + b ;
y = a + b + z ;
```

This could be replaced with:

```
t = a + b ;
x = y + t ;
y = t + z ;
```

As a result the additional addition has been eliminated. This can result in significant savings if a and b are floating point numbers and the code occurs in a tight loop. This technique was often used in the case study (see SDE 110) where constant terms which occur several times, like in a loop, where pre-evaluated before the loop.

6.2.5 Choosing Variable Types Appropriately

The amount of calculation can sometimes be reduced by considering the way in which the results will be passed. There is no point in working out an answer to eight decimal places if it is going to be rounded to a whole number at the next stage of the program. However on the other hand too little precision or too much truncation, too early in the program might considerably affect the reliability and integrity of later results.

6.2.6 Choosing Constant Values Appropriately

When using a timer which counts clock pulses, the value read from the timer often has to be converted into fractions of a second. Even if the conversion is combined with another calculation, it can still involve an awkward division routine. If other design considerations allow, the clock frequency can sometimes be chosen to make calculations easier. For example if the design calls for a clock frequency of approximately 4 MHz, the choice of 4.194 304 MHz will reduce calculations involving the counter/timers. This is because 4 194 304 is equal to 2^{22} and so division or multiplication by the clock frequency can be replaced by shifts.

6.2.7 Making Use of Binary Arithmetics

Another way in which the time spent on calculations can be reduced takes advantage of the binary nature of microprocessors. Most microprocessors can perform shift operations more quickly than multiplication or division, especially