

**Basic Business Operation**  
**Company Operation**  
**Project Development Methods**  
**Documentation Methods**  
**C & C++ Coding Standards**  
**Principles and Practices**

Document # : 000001  
Document Type : SD – Standards Document  
Version : 1.0.0  
Date : 13-03-12  
Time : 11:01  
Written by :- Mr Kim Lyon  
Managing Director  
World Computers Ltd. ( U.K. )  
Web Site :- wrldcomp.com .

Copyright © : World Computers Ltd. ( U.K. ) . 2000 - 2012 . All rights reserved !

This material is subject to the proprietary rights of World Computers Ltd. ( U.K. ) . It may be copied and altered without permission however any such copying and altering must not alter the proprietary rights of World Computers Ltd. ( U.K. ) or it's successors . It may not be used in publications – ie. direct commercial use may not be made of it . It may , however , be used privately and commercially within companies under the following conditions :-

- 1) a reference to the original author and the original copyright holder must be made within the resulting document .
- 2) a reference to the web site address must be made within the resulting document .
- 3) an acknowledgement of the original copyrights must be made within the resulting document .
- 4) the resulting document must not be copyrighted .
- 5) an acknowledgement that the use of the material does not imply any intellectual property rights including copyrights by the company and that these always remain , as primary rights , with World Computers Ltd. ( U.K. ) . Further , that the company will ensure that any usage of the material will not effect any intellectual property rights including copyrights of World Computers Ltd. ( U.K. ) .
- 6) if the resulting document spawns further resulting documents these conditions must be carried forward .

Usage of this material implies acceptance of these conditions .

Basic Business Operation.....	5
1. Business Basics.....	6
1.1 The Economic Cycle.....	6
1.2 Embracing the Inevitable.....	7
1.3 The Current State of the Software Industry.....	7
1.4 Basic Business Operation.....	8
1.5 Truths – Theory of Knowledge – Set Theory.....	9
1.6 Money.....	9
2. Company Value.....	10
1.1 Assets and Commodities.....	10
Drill Down of Company Operation from Basic Business Requirements.....	11
1. Drill Down of Company Operation from Basic Business Requirements.....	12
2. Rules of Thumb.....	14
Documentation Methods.....	15
1. Documentation.....	16
2. Documentation Methods.....	17
2.1 The Specifications in General.....	17
2.2 Rules of Thumb.....	17
2.2.1 Between Documents.....	17
2.2.2 Within Documents.....	17
2.2.2.1 URL's.....	17
3. The Project Register.....	18
4. The Document Register.....	18
5. Document Names , Numbering and Organisation.....	19
5.1 Names.....	19
5.2 Data Base.....	19
5.3 Principals.....	19
5.4 Organisation.....	20
6. The Requirements Specification.....	21
7. The Project Specifications.....	22
7.1 Hardware.....	22
7.1.1 Top Level.....	22
7.1.2 Bottom Level.....	22
7.2 Software.....	22
8. The Project Implementation.....	23
8.1 Hardware.....	23
8.1.1 Hardware Standards.....	23
8.2 Software.....	23
8.2.1 Software Standards.....	23
9. The Test Specifications.....	24
10. The Test Results.....	24
11. Version Control.....	24
12. Issues and Bugs.....	25
13. The Key to the Documentation.....	25
14. Testing.....	26
14.1 Asserts.....	26
14.2 Test Harnesses.....	26
15. Code Auditing.....	26
Project Management.....	27
1. Ownership and Responsibility.....	28
2. Complexity.....	28
3. Version Control.....	28
4. Metrics.....	29
Coding Standards.....	30
1. Purpose.....	31
1.1 Methodology.....	31
1.2 Design Philosophy – in Brief.....	31
1.3 Project Development.....	32
1.4 Explicit Software Development – Formal Methods.....	34
1.5 Layout.....	34

1.6 Structured Approach.....	34
1.7 Code Architecture.....	35
1.8 Hermetic – Self Contained Design.....	36
1.9 Sensibility of Code.....	36
1.10 Design Method.....	37
1.11 Asserts.....	37
1.12 Test Harnesses.....	37
1.13 Operating Systems.....	38
1.14 Re-usability.....	38
1.15 Substantial Coding.....	39
1.16 Noddy Code.....	39
1.17 Big Blob Code.....	39
1.18 Spaghetti Code – Big Complicated Messes.....	39
1.19 Mucky Coding & Code Freakery.....	39
1.20 Fire fighting.....	39
1.21 Differentiation.....	40
1.22 Consistency.....	40
1.23 Use of Macros.....	40
1.24 Use of Typedefs and Pointers.....	40
1.25 Use of Types and Casting.....	40
1.26 General Construction.....	41
1.26.1 Parameters.....	41
1.26.2 Code Sections.....	41
2. Componentisation.....	42
1.1 Co-Operative Operating Systems.....	42
1.2 Principles of Components.....	42
1.3 Layering.....	43
1.4 Device Drivers.....	43
1.5 Operating System Interfaces.....	43
3. General.....	44
1.1 Localising Data and Code.....	45
1.2 Separating of Code and Data.....	45
1.3 Structure and Organisation of Code.....	45
1.4 Common Code.....	45
4. Headers and Section Separators.....	46
1.1 File Headers.....	46
1.2 File Sections.....	47
1.3 Code , Header and Interface Files.....	48
1.4 Header Files.....	49
1.5 Function Headers.....	50
5. Functions.....	51
6. Nesting.....	52
7. Conditional Compiles.....	53
8. Spreading Out.....	54
1.1 Multiple Lines.....	55
1.2 Placement of Variable Definitions.....	55
1.3 Lining Up.....	55
9. Naming Conventions.....	56
1.1 Function and Variable Names.....	56
1.2 Constants and Enumerations.....	57
1.3 Macros.....	57
1.4 Structures.....	57
1.5 Unions.....	57
1.6 Types.....	57
10. Typedefs.....	58
1.1 Enums.....	58
11. Public Interfaces.....	59
12. Use of Brackets.....	60
1.1 ( & ) - Round Brackets.....	60
1.2 [ & ] - Square Brackets.....	60
1.3 { & } - Curly Brackets - Braces.....	60
13. Statements and Operators.....	61

1.1 If Statements.....	61
1.2 Else If Statements.....	61
1.3 ? Operator.....	61
1.4 Switch Statements.....	62
1.5 For Statements.....	62
1.6 Loops.....	62
1.7 Return Statements.....	63
1.8 Structure Field Alignment.....	63
1.9 Continue and Goto Statements.....	63
1.10 Endian.....	63
1.11 Pragmas.....	63
14. Program Architecture.....	64
1.1 Task Interconnection.....	64
1.2 C++ - Object Orientation - Encapsulation & Relationships - Componentisation.....	64
1.3 General Issues.....	64
15. Comments.....	65
1.7 The Keys to Commenting.....	66
1.8 Temporary Code.....	66

## **Basic Business Operation**

## 1. Business Basics

### 1.1 The Economic Cycle

The economic cycle oscillates between Top Down Corporate Capitalism and Bottom Up Democratic Capitalism .

During the Great Depression through after WWII legislation was introduced around the world to enable , facilitate , balance with and protect the Democratic Capitalism side of the economy . This was essential to the post WWII recovery .

This included legislation :-

- ✦ ensuring sufficient competition between corporations – giving customers choice .
- ✦ media legislation – ensuring that the media companies provided accurate information and giving customers sufficient choice .
- ✦ separation of retail banking and investment banking – ensuring created money did not pollute the investment side – ie. ensuring that there was no undue monetisation of essentially valueless moneys .
- ✦ regulation and protection of the retail banking industry – to ensure it's integrity and to protect the public .
- ✦ regulation of the investment industry – to ensure that it did not operate as a very unstable casino environment and that it served the public and society at large .

This ensured that there was sufficient diversity and it ensured that the economy was very vibrant , that it was highly creative , that it adapted to new business opportunities very quickly and that it progressively evolved .

This resulted in a massive jump in new businesses developing and marketing new technologies and associated products .

During the mid 70's the mergers and acquisitions boom started . This was the period that the economic cycle swung around from Democratic Capitalism to Corporate Capitalism . The corporates provided economies of scale and very large market access . It was also an era where there was an excess of investment moneys looking at a shortage of safe investment placements , this resulted in the creation of the debt culture and , hence , a return to the 20's and the cycle rolled on .

A small or medium sized company is very close to the market place , is very much intouch with what their customers want and is able to immediately see the results of any actions that they take . They are able to very quickly make any course adjustments and they do so often and at low cost . As such they are very fault tolerant . They try things out and they progressively evolve – adopting what works , dumping what doesn't . They are very relaxed and very professional in their manner , even if thing are very frenetic which they often are . This all means that the basic question 'why bother ?' is almost always answered in the positive . The staff are prepared to stick their necks out .

Corporations are often very not fault tolerant . They play it very safe because mistakes can be very costly . They also often have an environment that greatly remotes the results of their staff's actions from those staff themselves . They are often dysfunctional and have a large disjuncture from reality . They often make mountains out of molehills – making big issues out of situations rather than dealing with them very quickly and getting on with the job . They can also often not think things through and not have good methods and standards . They are often not places where things can be tried out and evolved .

In some corporations they can have a lot of politics , personalities and egos at work . This all means that the basic question 'why bother ?' is often answered in the negative . The staff keep their heads below the parapets .

Monoliths – corporate and political – can often be very intolerant of the very things they need for their survival . They can often be very immature and very dysfunctional environments . Their staff often end up like kangaroos staring into the headlights of an oncoming road train .

Corporations often take a commodities view of the world – they manufacture commodities and they employ people as commodities . Small and medium sized companies often take an assets view of the world – they develop assets and they employ people as assets .

Corporations often build into themselves the seeds of their own destruction . They build in the bad culture , the immaturity , the bad morality , the inflexibility , the lack of thought , of discussion and of perspective .

They are often subject to mass extinction events because they often don't see the reality of the changing times and they are often incapable of adapting to the changing times . This we are seeing many times during the 00's and the 10's .

During the 90's and 00's many corporations reversed the relationship with the customer . Instead of trying to find out what the customer wanted they dictated to the customer what they were to buy . They aggressively marketed , they used loyalty schemes , advertising and public relations to convince the customer what to buy and where to buy it . The corporates made the mistake of trying to alter the externals rather than the internals . Meanwhile what the customer wanted diverged from what the corporation was supplying . So the customer went elsewhere or nowhere . This being clearly visible where purchase and consumption methods made possible by the Internet diverged from High St. chains methods of operation and where the customer simply stopped buying .

This approach of trying to adjust externals rather than the internals has not only effected individual corporations , it also has effected large parts of society producing massive economic and moral problems . The cheating has been practiced widely and has had broad effects .

## **1.2 Embracing the Inevitable**

When corporations start trying to change the externals -their customers – that's when things are starting to go wrong for them .The big problem is that corporations often become dysfunctional and , as such , they either don't take any action or they take the wrong action . Many corporations spend vast sums of money trying to stave off the inevitable and along the way make massive losses .

A good example of this is the replacement of purchasing from the High St. shops with purchasing from Internet shops . Often the High St. chains have tried aggressively marketing – loyalty schemes , one or more reductions of product range to core products , then 'pile them high and sell them cheap' . During this process they change from a 'go to' place to a 'don't go to' place . The Internet becomes the 'go to' place . The customers know that if they go to the High St. shop they are unlikely to find what they want and they will spend much time and cost ( petrol and parking ) and be in an unpleasant environment and have bad service in the process . They know that if they use the Internet they can quickly , easily and cheaply find what they want and in the comfort of their own homes .

In the same way the actions of the corporations over issues of copyright and anti piracy are also attempts to alter the externals rather than adapting to the changing nature of the market place .

So the fact is if the corporation doesn't quickly re-invent itself it is going down at great cost to itself and it's shareholders . This re-invention will change everything but it has to be addressed and adopted if the corporation is to survive . If the corporation goes down and destroys itself , as often happens , others , that have adopted the new paradigm , will rise in it's place .

## **1.3 The Current State of the Software Industry**

The software industry is very much like the automotive industry of a hundred years ago . Then if you wanted to drive from London to Brighton you had to shove a mechanic in the back . These days we call them IT support personal .

Cars were , in those days , very customised affairs . These days everything is modular . This means that design and manufacturing costs are greatly reduced and market reach is greatly increased .

Software is often a re-inventing of the wheel . Part of the problem is the rapid pace at which the industry has developed – it is yet to mature . Another part of the problem is the issue of copyrights – there is no system in place for contractors to develop code independently and then to license clients with it . It's a case of the clients preferences – they would rather the code be redeveloped and then for them to have the copyright . This means that development time is often not utilised to further the field .

## 1.4 **Basic Business Operation**

As such , within the company , it is very important to have an environment that enables , facilitates and promotes progressive evolution :-

- 1) value creativity and independent thought . It is out of these that new products are created and understandings of new business directions comes from .
- 2) be thinking in approach . Always look at all the various different ways of doing things . Think things through and choose the best ways forward .
- 3) support free speech and discussion . The bouncing around of ideas is very important to progressive evolution .
- 4) keep everything professional . Keep out personalities and egos . Don't attack people personally . Don't pull people down into the gutter . Respect people .
- 5) always challenge oneself . Always look for areas where one can improve , always look better ways of doing things . Always be moving forward , anyone who isn't is stagnating and is losing the race .
- 6) use failure as part of the learning process . Try things out – nothing ventured , nothing gained .

Life is not black and white . Successes can always be improved on and they should be improved on , not to do so is to fall behind and let the competition steal a lead . Failures are usually not complete failures , the positive aspects must be retained for use in the future .

It is very important to identify failures – wrong decisions – early and to correct them very quickly and cheaply . This should be a natural and normal part of the everyday process .

- 7) localise responsibilities and control – the ownership principle . Allow people to see the results of their actions directly and to make the necessary decisions to correct their course .
- 8) managers should keep in frequent – eg. daily – contact with their staff and keep a good eye on the course and progress of the project development and get any necessary corrections made . This should be the main form of peer review .
- 9) keep very much in contact with what the customer wants . Be proactive and be strategic . Always be fully in touch with the reality of the situation .
- 10) adapt to the changing times . Embrace the future . Even if this means destroying all that you currently are .
- 11) be very strategic business wise with the engineering design . Always look for ways of reducing downstream development costs and turn around time . And always look at maximising the reach of all designs – where they can be used .

Life is very much a balancing exercise . It is important to remember this . It is very important to keep in mind the whole picture .

## 1.5 **Truths – Theory of Knowledge – Set Theory**

Truths have locus' of validity . A greater truth has a larger locus of validity . A superior truth has a larger locus of validity and takes into account – contains – inferior truths .

When a truth is originated , stands up off itself , and is seen as valid it is adopted . It is supplanted by a superior truth . In this way the people and the companies adopting the truth maintain their validity and , hence , their relevancy .

## 1.6 **Money**

The economy is the environment under which people exchange hours worked . Money is the means under which they do this . Money forms the means under which an equivalence is made between one persons labour – producing goods and services and another persons labour .

In the carrying out of work the total money associated has the following breakdown :-

- 1) the actual value of the work to the person performing the work – this is what the person can exchange for other people's goods and services .
- 2) taxation paid by the person – a combination of overheads , debt interest payments and advance payment for goods and services from the government .
- 3) overheads and payment for work not directly related to the production of goods and services within the company .
- 4) taxation paid by the company .
- 5) profit to the company . This includes dividends paid by the company .

Overheads and debt interest payments produce a discrepancy between the cost of goods and services and the amount of money available for payment for goods and services . This is why it is important to minimise non productive costs thus maximising general wealth .

## **2. Company Value**

### **1.1 Assets and Commodities**

An asset is something – a project , an object , a piece of equipment , even a person – whose repeated use produces profits for the company .

A commodity is an object that is manufactured once and then is sold .

It is very important to understand the differences and to understand where their places are found in the company .

If a company sets itself up as a software applications developer and develops for each customer basically the same package , but develops it largely from scratch , then that software is largely being developed as a commodity – ie. it has no asset value . Any assets that they company has are largely with their staff rather than the work produced by their staff . This often occurs where the company is trying to start a business on minimal cost – quick , cheap and nasty .

If the development staff put in the extra effort upfront ( requires extra expenditure and development time ofcourse ) they are able to design the software to be easily reconfigurable , ie. to be reusable . As such the software becomes an asset and the cost of using the software in subsequent projects is greatly reduced thus greatly increasing the profits of the company . The software development becomes one of developing as assets and marketing as commodities .

The same principle can be applied to the staff . A staff member can be an asset or can be employed as a commodity . If the staff member is engineering the software – producing structures , frameworks and software components - they adding to the asset value of the company . If they are just churning out code , essentially re-inventing the wheel , then they are just being used as a commodity .

Within any engineering company it is very important that the engineers operate as assets and that they are a part of the senior management . Very often the business side of the senior management have no idea of how to translate their business aims into how the engineering is implemented . This disjuncture greatly effects business performance . Bringing in senior engineers into senior management and training them up in the basics of business goes a long way to solving this problem .

## **Drill Down of Company Operation from Basic Business Requirements**

## 1. Drill Down of Company Operation from Basic Business Requirements

As with any company there must be a relationship between the basic business requirements and the operation of the business .

The most basic requirement of any business is ' capital productivity ' . Capital productivity is the maintenance and increase of the capital value of a company . When capital is invested in a company it is transformed into assets . These assets can be of many forms . Conceptually they are divided into tangible assets - buildings , equipment etc. - and intangible - people etc. . Within a company that develops software it is very important that the software be treated as an asset . Often it isn't .

Assets must have ' asset calculations ' applied to them . These determine

- 1) the cost of the asset - how much it costs to develop the code - the initial investment .
- 2) the loss of value of the asset - the devaluation as expressed as a loss of value over a period of time - usually over a year .
- 3) the expenditure required to maintain and increase the value of the asset . In software terms this is maintenance and update costs .

Assets are very curious things . They are items that cost up front , can generate income and can be sold on and in some forms they can walk out of the door .

What minimises the costs in the industry as a whole - minimises the amount of unnecessary code development - minimises the cost of using the components - maximises the value of those components that are developed . The best designed code is code that requires the least amount effort expended overall . The software industry is like the motor car industry of the 1920's or earlier . Then there were many car parts that had to be fitted by the use of extra tooling - such as the use of lathes . Subsequently the car industry manufactured parts such that they could be put directly in without further work . IE. the amount of overall effort was minimised .

Components should be developed as assets but marketed as commodities . Not the other way around .

The fact is that a lot of business people do not translate the business side into the engineering side - they treat the engineering side as an unknown . Likewise on the engineering - programming - side - they often don't fully understand the business side . As a result of this disjuncture software development is often undertaken , within the business , without full consideration of the business requirements . Good business translated into good engineering produces good engineering which translates into good business .

As with anything in life code development is a matter of balance - of optimising - rather than minimising . As such :-

- 1) it is very important that the initial development produce optimised code . Minimal development cost - ' quick and nasty ' - code actually has large down stream - maintenance and update - costs . Consequentially it has low asset value .
- 2) it is very important that the life time of the code be maximised . If code has to be constantly replaced it has minimal lifetime and hence minimal value . The best bad example of this is customised application engineering companies . These usually redevelop large amounts of their code for each end user contract . As such their code has very limited lifetime and , hence , very limited asset value .
- 3) it is very important that the market reach of the code be maximised . This is carried out by generalising the design of the code - maximising the locus of validity of the code .
- 4) it is very important that the expenditure to maintain , support and update the code be minimised . This is carried out by designing the code such that bugs are minimised , contained and easily found . By designing the code such that it is easy and quick to use . By designing the code such that it can be easily expanded and evolved .

A project is designed well if good design principles - good design philosophy - is applied to it .

It is very important to engineer the project - not to just slap down the code without any thought as to how the structure of the project is going to be designed .

The structure defines how the components are to be connected together and defines how they are to communicate with each other .

Componentisation is very important but components have to be designed on the following principles :-

- 1) the total amount of code developed has to be minimised . Actions which work against this are :-
  - i) simplistic - noddy - code . Many layers of simplistic code does not optimise - minimise overall - the amount of code - it just leads to overly complicated code .
  - ii) big blob - directly overly complicated code - requires a lot of code to deal with the complexity .
  - iii) code which requires much support code . This often occurs where parent components - such as a multi media play engine - require child components - such as the codecs - to each have large amounts of common code - such as buffer management - duplicated in each of the components . At it's worst components can be designed such that they are not self contained and , as such , require a lot of support code - a lot of ' write arounds ' - in the components that they are connected to . This is a situation that is half way from a well compontised system to a big blob .

The ways to optimise - minimise overall - the amount of code are :-

- i) to choose the best structure for the code . The structure has to be thought out . It is not a matter of just bunging in a solution . There are different ways of solving a problem , some are better than others . It's a matter of examining all the ways and choosing the best way . Often , though , peoples personalities - egos - and politics get in the way . A common way that this goes astray is where RTOS's are used - often there is far greater use of mutex's and far too little use of mailboxes - event messaging - than there should be . Principles of self containment must be applied :-
  - 1) a component must be totally responsible for all data and operations within it's domain - there must be no visibility of the component's domain provided to other components .
  - 2) all actions within the component's domain must be performed within the component's task .
- ii) to divide the areas of the program into components . It is very important that the areas - hence components - are properly identified and demarcated - that there is no fussiness .
- iii) the structure - and hence the components - must have a clear 1 to 1 relationship to the physicality - the reality - of what is trying to be achieved . The fact is that software is actually a very physical thing - to try to portray it as anything other than that is actually self deluding and just adds to the complexity of the code and makes it harder to deal with .
- iv) to common the code . This is undertaken by :-
  - 1) within components identifying common operations and commoning this code into functions .
  - 2) in between components identifying common operations and commoning these into parent components .
  - 3) putting general functionality into utility functions within utility components .

The common area where projects go wrong is the lack of rationalising of the project . Projects have to be thought out rationally , thoroughly and much commoning of code must be undertaken .

  - 2) to minimise maintenance costs . This is undertaken by :-
    - i. minimising the likelihood of bugs occurring . This is done by sensible code design - maximising the locus of validity of the code . If the code is designed on the basis of rules - if exceptions are not created - then opportunities for bugs are not created .
    - ii. designing the code such that it can be easily used without causing bugging conditions .
    - iii. designing the code such that bugs are localised in their effect .
    - iv. designing the code such that bugs are easily identified .
    - v. sufficiently testing the code .
- 3) to minimise update costs . This is undertaken by :-
  - i. ensuring that any expansions require minimum alterations to the existing code .
  - ii. ensuring that the code can be easily evolved .

Common to all aspects of good code design is good methodology . This includes good coding standards which includes using good naming conventions . Code has to be easy to copy , paste and change using the replace operation . It must be explicit in it's construction and layout and be well commented .

## 2. **Rules of Thumb**

- 1) Design code as an asset and market it as a commodity .
- 2) Maximise the locus of validity - design by rules , not by exceptions .
- 3) The best designed code is code that requires the least amount of effort expended on it overall - ie. in development , maintaining , updating , supporting and usage of etc. and that has maximum market reach - maximum possible number of end user platforms . Optimal input for maximal output . Spend the time upfront to minimise the time required downstream .

## Documentation Methods

## 1. **Documentation**

A documentation chain must be established . Each document is based on it's preceding document and is accountable to it . The project is driven by the specifications . As the project is evolved each effected document is also updated and re-issued as a new internal version .

The specifications are divided up into 2 distinct areas . The first is the Requirements Specification . This is produced by the customer and \ or the Project Manager . The second is the project's Architecture and Component Specification that is prepared by the architect . The Development Engineers are supplied with Requirements Specifications . They then put together a specification of operation of the components . During the project development – where common component types are being developed – the specifications of the initial component of each of those groups – as drawn up by the Development Engineer working on the component – is used as the specification for all other components in that group .

EG. – where device drivers are being developed – the Project Manager , with input from the Project Architect , provides Requirement Specifications to the Development Engineer . The Development Engineer then develops the first driver . This would include a wrapper – supplied by the architect – and a template – developed by the Development Engineer along with a component specification – also developed by the Development Engineer . As such – these for the basis for all future driver development in that area .

The architect must have an Architecture and Componentisation document that describes how projects are put together ( the architecture ) and how the components are attached to the framework and how they communicate via the framework . This is a general document that is referred to by the Architecture and Component Specification title . This is passed along with the Requirement Specifications to the Development Engineers . It is the responsibility of the Development Engineers to design and develop the components – they need to be able to think for themselves and to think things through – they need to Engineer ! The thinking for the project , and , correspondingly , the coding and the documentation is spread throughout the project .

As much as possible the documentation must be placed within the actual source files . Separate documents must only be constructed where it is not appropriate – due to the nature of the document – that the document be separate from the code .

The individual components must have their own parameter and functional specifications .

Each file has a file header describing the file , functional headers describing the individual functions and comments describing the individual statements . Each statement is commented as to what it does with what .

The developer is held accountable to the component specifications . The component specifications are held accountable to the requirement specifications . The requirement specifications protect the customer and the supplier . The component specifications protect the client and the contractor .

If the developer sees any problems with the component implementation he \ she raises an issue - logged on the email . This is passed up to the Project Manager and if necessary to the customer . The Project Manager or customer then responds with a ruling or a project or component specification variation .

All documents must be placed within their respective project areas . Paths to the project areas and the individual documents must be placed within the Document Data Base . The Document Data Base must be set up such that the document can be accessed directly via the document number .

Within files standard European – Day – Month – Year – date format should be used - eg. 23-05-11 . Where used – times should be in 24H format - eg. 14:49 .

Where files are dated reverse date format – 2 digit Year – 2 digit Month – 2 digit Day – should be used - eg. 110523 .

## **2. Documentation Methods**

### **2.1 The Specifications in General**

The specifications are used to drive the project . Each document links upwards and downwards to other documents . At the bottom the project is implemented . Where major questions need to be answered a document is created and derived .

The specifications are a description of how the project is to be implemented or is being implemented . They are worded in exact phrasing and are concise , precise and pertinent . The documentation is tightly put together . The optimum amount of documentation is produced :-

- large enough to allow a person who has no knowledge of the project to quickly and easily gain a good understanding of the project .
- small enough not to waste unnecessary time .

In all aspects of the project the complexity must be driven down . Simple and sophisticated – not simplistic and complicated . The use of engineering approaches – looking at how things are put together , and engineering techniques – doing things in better ways – must be made .

Time must be spent upfront in order to save significantly more time downstream .

### **2.2 Rules of Thumb**

#### **2.2.1 Between Documents**

- 1) highly organise all data going into the documents :-
  - i) use tree and lateral relationships .
  - ii) use technical numbering .
  - iii) place everything into appropriate places .
- 2) put in matching sections – number for number .
- 3) do not duplicate – refer to .
- 4) make things logical , intuitive and easy to find .

#### **2.2.2 Within Documents**

- 1) be consistent in the use of names - same names for the same things everywhere .
- 2) use names that mean something specific .
- 3) use definite wording – not vague wording . If you don't know the definites work down the document tree to produce the definites . If it isn't in the Requirements Specification and it is an issue then raise it as an issue .

##### **2.2.2.1 URL's**

All URL's must end in a white space .

### **3. The Project Register**

The Project Register is a database that contains a list of all projects produced by the company along with their release dates and times . This is placed in the root directory of the project documentation area .

Along with this document a standards directory is created that contains the company's standards documents .

### **4. The Document Register**

The Document Register is a database that contains a list of all documents produced for the project along with their names , document numbers , versions and release dates and times . This is placed in the root directory of the individual project documentation area – one level below the Project Register directory .

## 5. Document Names , Numbering and Organisation

### 5.1 Names

- 1) The document number . Each document must be assigned a unique number .  
eg. 000001\_
- 2) The project name . This may be a short form name .  
eg. ABCProject\_
- 3) The particular part of the project that the document applies to .  
eg. XYZ\_Board\_
- 4) The document type – eg. Requirements Specification , . This may be a short form type . It may also consist of sub types . These type and sub type abbreviations must correspond to the standard abbreviations used by the company and detailed in the company's standards .

SD	Standards Document	
RS	Requirement Specification	
IS	Implementation Specification	
ID	Implementation Description	- often part of the Implementation Specification or just placed within the code . Hardware requires separate description documents – circuit diagrams etc.
TS	Test Specification	
TR	Test Results	

- 5) The document issue version number . This consists of the following format :-
  - i) the major release version number . 0 indicates not released .
  - ii) the minor release version number . starts from 0 .
  - iii) the internal update number . starts from 0 .
 major-minor-internal – eg. 1-01-363\_

### 5.2 Data Base

A data base must be maintained allowing the documents to be accessed via links on the following basis :-

- 1) document number .
- 2) document number and version .
- 3) document name .
- 4) document name and document type .

### 5.3 Principals

- 1) each document must organise into a directory in alphabetical order according to the areas that they are covering and according to the versions that they are covering .
- 2) each document must , by it's name , be localisable within the document tree .
- 3) when a document is modified internally it's internal number is then incremented .
- 4) when a project release occurs , if the release is a minor update , the minor number is then incremented and the internal number reset , if the release is a major release – eg. a completely new set of features – the major number is then incremented and the minor number and the internal number are reset . At the release point copies are made of all the corresponding documents with their numbers updated . The document register is then updated .
- 5) the document name must , by it's nature , give it's position in the document tree .

#### **5.4 Organisation**

- 1) the project must be organised into a tree directory structure
- 2) each area of the project must have it's own sub directory .

## 6. The Requirements Specification

"What are we going to achieve overall"

This is the initial document from which all other documents are derived , either directly or indirectly . It is what is used to drive the project in the initial instance . What it specifies are the overall functional and performance requirements of the project . It is the client facing document .

Projects can be internal projects – eg. components and modules developed for general release , or external projects – projects developed to specific client requirements .

The project must start with the External Requirements Specification . This is a document that is developed in direct consultation with the client . If necessary there is a corresponding Internal Requirements Specification also developed . The Requirements Specification consists of a set of statements that specify exactly what the project will consist of in all respects . This is done by using the key words such as :-

shall	have the feature
will	provide the interface to
must	have the capability
should	if possible have the feature \ capability
may \ can	feature \ operation is allowable
cannot	feature \ operation is not allowable

The Requirements Specification does not describe how the project is implemented .

The Requirements Specification is the primary document and is on top of the project document tree – the document from which all other documents are derived .

Each item in the Requirements Specification is a testable item and must have a corresponding test specification and a corresponding test constructed and , when passed , signed off . If that test can be automated it must .

It is important that the Requirements Specification contains all aspects of the project that the client expects to be in place – no assumptions must be made – everything , beyond the obvious , needs to be specified .

When the Requirements Specification \ External Requirements Specification is completed it must then be signed off by the client – this is their agreement as to the specifications of the project . At this point the project is quoted for . It may be a fixed price quote or it may have certain leeway built into it .

When the project is complete the project is tested against the Requirements Specification and if it passes delivery is then made , an invoice is issued and full payment is then made .

Should the client require a variation to the Requirements Specification a modification to the Requirements Specification is issued . This is quoted on and signed off by the client . Modification require changes in development under way .

Should the client require an addition to the Requirements Specification an update to the Requirements Specification is issued . This is quoted on and signed off by the client . Additions usually just require additional work to be carried out . As such if they can be dealt with as an upgrade to the initial release they should . This allows the initial release to occur as quickly as possible and , likewise , payment to be made as quickly as possible .

## **7. The Project Specifications**

"How we are going to achieve it"

This describes how the project is actually to be implemented . It includes text documents but may also include specifications placed within the code - such as data communications protocols . It is the internal engineering document – along with the Project Implementation document it contains the result of the engineering design of the project .

The Project Specifications are a set of documents that lead directly off the Requirements Specification . These specify the external technical aspects of the project . The specifications form branches of the tree . Each level splits down producing more specifications as necessary . The implementation then follows from these .

### **7.1 Hardware**

#### **7.1.1 Top Level**

- 1) overall functionality of the device – what it does .
- 2) what it consists of major components wise .
- 3) external interfaces .
- 4) size and shape of the box . Materials and colours .
- 5) placement of holes .

#### **7.1.2 Bottom Level**

- 1) hardware blocks required .
- 2) major components – processor , graphics , screen type , keyboard type etc. .
- 3) power supplies – number and types . Regulation used .

### **7.2 Software**

- 1) what OS is being used – in house or bought in .
- 2) how is the project componentised .
- 3) what are the major modules required .
- 4) what are their corresponding components .
- 5) how are the modules communicating . What are the messaging format specifications .

## **8. The Project Implementation**

"How we achieved it"

This describes how the project was implemented . It includes an overall description of the project and it includes giving reasons for design choices . It also includes the actual project design - circuit diagrams , source code , comments etc. .

### **8.1 Hardware**

- 1) hardware description .
  - 2) associated calculations .
  - 3) circuit diagrams .
  - 4) FPGA , VHDL designs etc. .
  - 5) Gerber files .
- etc.

#### **8.1.1 Hardware Standards**

A hardware standards document must be created and maintained .

### **8.2 Software**

- 1) software description .
  - 2) interface specifications .
  - 3) code files .
- etc.

#### **8.2.1 Software Standards**

A software standards document must be created and maintained . This follows in this document .

## **9. The Test Specifications**

"How we are proving that we achieved it"

This specifies each test that is to be performed on the project . As such it is used to verify the project specification and , in turn , it verifies the requirement specification . Often the test specification is just a set of tests within a test harness and , as such , contains the documentation associated with the tests . Each test is a simple Do A , Check for B operation .

The Test Specifications document must have a 1:1 correspondence to the Project Specifications document . As such nothing must be specified in the Test Specifications document if it is already specified in the Project Specifications document . Where tests are automated these must be coded with a test script file or a code file . As such they must not be duplicated within the Test Specifications document – only referred to by it in it's corresponding section .

## **10. The Test Results**

"Proof of how we achieved it"

These are just a list of results corresponding 1:1 to the items in the test specification . Basically Pass \ Fail and comments .

The Test Results document must have a 1:1 correspondence to the Test Specifications document . As such nothing must be specified in the Test Results document if it is already specified in the Test Specifications document . Where tests are automated these must result in a test results file being outputted . The results file must be referred to in the Test Results document in the document's corresponding section .

## **11. Version Control**

A version control system allows a version history to be easily maintained . However it should not be relied on – files can become corrupted thus throwing out a large number of the contained versions . As as full image backups must also be made .

Within the versioning a clear distinction must be made between internal development , minor releases and major releases .

When a project is released a full copy of all files must be made . This must include output files such as object files and executables . This allows all aspects of the release to be referred to . Within the version control system a label point must be established .

## 12. Issues and Bugs

During project development issues often occur that need to be raised and dealt with . These are aspects discovered during the development of the project that require input from the client . A database needs to be constructed and maintained listing the issues and bugs \ failures . The client needs to have access to the external items of the database . The fields that the database should have are :-

- 1) the issue number – automatically assigned .
- 2) the name of the issue .
- 3) a description of the issue .
- 4) what type of issue it is :-
  - i) design vagueness . Needs clarification from the client .
  - ii) design conflict . Needs resolution of the conflict – deciding which side needs fixing .
  - iii) bug \ failure . Needs fixing .
- 5) who gets visibility of the issue :-
  - i) internal .
  - ii) internal & client .
- 6) who the issue is assigned to .
- 7) whether the issue is solved .
- 8) comments .

## 13. The Key to the Documentation

The key to the documentation is :-

- 1) that there is a documentation chain - proof of project .
- 2) that each document drives the next document and drives the project development . And , as such , each document or project part is derived off it's preceding document .
- 3) that the project can be signed off .
- 4) that the project can be easily , quickly and cheaply expanded and updated as required .

## 14. Testing

### 14.1 Asserts

The code must be instrumented throughout with asserts . The assert code must have the following characteristics :-

- 1) an assert handler component must be constructed .
- 2) within the assert handler component's header file a basic series of assert macros must be defined .
- 3) a set of assert macros must then be defined to handle the specific areas in the code being checked and must use the basic assert macros in their definitions .
- 4) the asserts must be conditionally compiled into or out of the build .
- 5) switches for the conditional compilation must be placed in the general conditional compilation switches area .

### 14.2 Test Harnesses

All projects must be verified at the binary level by using test harnesses . Wherever possible - test harnesses must be constructed – and these must be , where possible , fully automated and display and record the result of the tests . The test harnesses must connect via a single port and their events be propagated down to the individual frameworks and to the individual components and the responses propagated back to the originating test harness . It must be possible to :-

- 1) test individual components separate – isolated - from the rest of the build .
- 2) test groups of components .
- 3) test the overall build .
- 4) monitor - intercept and pass on - messages from components operating within the build .
- 5) monitor any output from the asserts .

The test harnesses must undertake the following :-

- 1) display an identifying syndrome of the binary build – such as a checksum .
- 2) display the size of the binary build .
- 3) for released builds - display the date and time of the build compile .
- 4) display the date and time of the test .
- 5) display details of the individual tests .
- 6) display the results of the individual tests .
- 7) display a pass or fail on the individual tests .
- 8) display an overall pass or fail .
- 9) store all the information displayed in a file .

## 15. Code Auditing

All code must be audited by the respective project manager . Checks must be made for the following :-

- 1) code integrity – that the code will function fully as required and under all conditions – this must be verified by using the test harnesses and by checking the source code .
- 2) code security – that the code does not expose the system it any security problems .
- 3) intellectual property issues – that the code does not violate any external intellectual property rights .

## **Project Management**

## 1. Ownership and Responsibility

It is very important that the project is run on an ownership and responsibility basis . This ensures that all elements of the design and of the components are fully managed and fully handled . There must be full accountability and full feedback . Full support must be provided to ensure that the project functions as desired . The customer must be fully satisfied .

## 2. Complexity

It is very important that the project be designed with the least amount of complexity possible . The complexity must be driven down - if it can be done in a simpler way then do it in a simpler way . This is very much a case of optimal design – keeping things simple but sophisticated , not simplistic but complicated .

## 3. Version Control

A good VCS system must be used . The VCS data base must be located on a secure server in a separate location from the development . The VCS must be used separately for development versioning and release versioning . The components must be checked in at least at the middle of the work day and at the end of the work day ( preferably in a working form ) - thus providing for project backups . When the module is released all of the files must be labelled with the release version .

Once the project is released it must be maintained on a separate released project data base and logs – automated and manual - of update dates , authors and updates details must be maintained within the code .

The Version Control must be operated on the following basis :-

- 1) the VCS must indicate – via a file manager – explorer – type interface which files are checked out and which local files have a new version on the VCS data base .
- 2) the VCS must allow previous versions of files to be opened – via a version history list .
- 3) each person has ownership of their own specific area . Merging is not permitted – the project must not be developed such that more than one person is modifying the same file . At any one time only one person is to be responsible for checking out , modifying and checking in a file .
- 4) wherever possible the code from any specific code development area must only be supplied to other members of the project development group in the form of libraries or executables . Access to source code not owned by an individual must not be provided to that individual . This ensures that fundamental security of the project is maintained – no one person can walk off with all the code . This does require clear demarcation between components and between modules and clear definitions of the interfaces and operations of the components and modules – but this is fundamental to good project management and development . Architects and Project Managers would , of course , be given further visibility .
- 5) for the project - public branches only are to be used . The public branch is used to provide access to the resulting libraries or executables developed . Each area of ownership must have it's own private area on the VCS data base with no visibility to the areas owned by other members of the project development group . Where common files – such as project wide headers and interface headers – are used these must be placed within the public area .
- 6) when a person is modifying a file he or she checks out that file . The checking out of the file must prevent anyone else modifying the file .
- 7) when a person has finished modifying a file and has proved his or her build he or she must first get the latest copy of all other modified files within the project area and ensure that the build is not broken . He or she must not check in his or her files such that the build – on the VCS data base – is broken . When the files are ready to be checked in the person must :-
  - i) send an email to all project group members indicating a check in is about to occur – listing a summary of the changes made and the names of the files being checked in . No other check ins are to occur at this time .
  - ii) check the files in .
  - iii) send an email to all project group members indicating that the check in is complete . Other check ins may now commence .
- 8) all project files – including object files , libraries and executables and documentation files must be checked in .

#### 4. Metrics

As a general rule of thumb the following metrics are to be worked to :-

##### Machine Code Bytes Produced per Person per Week

C	2.5KB
C++	2.8KB

##### Ratio of Source Code and Documentation Size to Machine Code Size – Byte to Byte

C	32:1
C++	36:01:00

##### Cost of Code Production – per Byte of Code

C	€ 0.92
C++	€ 0.80

This includes the production of test harnesses .

## **Coding Standards**

## 1. **Purpose**

The purpose of coding standards is multi fold :-

- 1) to ensure that a systemised methodology is applied to the coding development in order to :-
  - i) minimise the chances of bugging occurring .
  - ii) to minimise the damage that those bugs may cause .
  - iii) to maximise the chances of identifying , localising and fixing bugs .
  - iv) to maximise the chances of successful development of the project .
- 2) to ensure that the code can be easily , quickly and accurately reviewed .
- 3) to ensure that the code can be easily , quickly and accurately debugged .
- 4) to ensure that the code can be easily and quickly maintained , updated and evolved .

### 1.1 **Methodology**

When writing code many different approaches can be made .

Some of these approaches will produce code that will not work in all circumstances .

Some of these approaches will produce code that will work in all expected circumstances but not in all circumstances .

The rest will produce code that will work in all circumstances .

One of the purposes of coding standards is to ensure that the code does not fall into the first category – ie. that it operates within the third category . Coding standards pay a very large part in raising the code from the first category to the second . However , to raise the code from the second category to the third also requires a heavy dose of design philosophy .

A highly methodical approach is very important , irrespective of the size of the program . It is especially important , critical even , with large projects .

Good methodology results in :-

- 1) ensuring that the program does what it is supposed to do .
- 2) ensuring that bugs are not designed into the project .
- 3) ensuring that complexity is minimised .
- 4) ensuring that if it is the same that it is written the same and constructed the same .
- 5) ensuring that the structure is clearly visible – that the program can be clearly seen and understood in its parts and in its whole .

### 1.2 **Design Philosophy – in Brief**

Design philosophy requires a principles and concepts approach . It requires that the generalities be understood – general effects and their causes – directions and results .

Design Philosophy is largely theory of knowledge - set theory , however , it goes a lot further than that . It is also concerned with understanding structures and relationships .

One of the key things that Design Philosophy is concerned with is ‘ locus’ of validity ‘ . What this is concerned with is maximising area of truth ( coverage ) of the design . As such it looks for commonalities and then deals with these in a common manner . It uses rationalisation to bring methods into common methods and , as such , to form supersets .

It's only at the upper levels where choices are made as to the best – or most appropriate - approach .

It is very important to keep in mind the principles being used and the relationships present . Consistency of approach must be maintained . The design must be approached logically and objectively . Each design decision must be made purely on technical engineering grounds and must correspond to the superior or greater truth . General approaches must be used .

The design must be by rules – not by exceptions .

### 1.3 Project Development

Project development requires the following personal :-

- 1) the Project Manager – who is responsible for :-
  - i) the overall technical management of the project .
  - ii) the people management of the project .
- 2) the Project Architect\ s – who is \ are responsible for :-
  - i) the design of the architecture .
  - ii) the development of the frameworks .
  - iii) the handling of issues raised by the Development Engineers .
- 3) the Development Engineers – who are responsible for the development of the individual components that are bound to the frameworks .

The Project Manager provides the requirement specifications to the Project Architect and the Development Engineers . As issues are raised and addressed and as the design of the project progresses the Project Manager provides updated requirement specifications – differences and complete specifications . It is very much a collaborative effort .

The project is not designed in a top down fully specified manner but is designed in a combination of top down and bottom up . What this means is :-

- 1) the project is always working , demonstrate-able and test-able .
- 2) the project is evolved and expanded . The architecture must lend itself to being easily evolved and expanded .
- 3) the communication path is bi-directional with requirement and architecture specifications going down and issues going up . These issues are addressed by the Project Manager and Project Architect . In such a manner the project is fleshed out .

The Project Architect and the Development Engineers :- are expected to act as engineers – to design and develop . They must think out their design thoroughly and implement it properly .

It is not the Project Architect's job to know everything about everything in the project . It is the Project Architect's job to know everything about connecting everything together in the project and to manage the operation of the project . It is the responsibility of the Development Engineers to inform the Project Architect of what their requirements are . It is then the responsibility of the Project Architect to provide the Development Engineers with the facilities that they require . It is the responsibility of the Project Architect to ensure that the Development Engineer's components can function within the build . It is the responsibility of the Development Engineers to ensure that their components are constructed to the requirements dictated by the framework .

Every member of the group is expected to contribute – within their respective areas – to the design – ie. the design process is one of collaboration . Given that each part of the group has a specific area of responsibility they also have specific people to whom they are accountable . The Project Manager manages the overall project – what he or she says goes . The Project Architect is accountable to the Project Manager and he or she dictates to the Development Engineers how their components are going to link into the framework . The Development Engineers design and develop the individual components on the basis of the Requirement Specifications and the Architectural documents . The Development Engineers raise issues , as they occur , with the Project Manager – this is the collaboration . The Development Engineers do not design the architecture and they do not manage the project – this is not their area . The Development Engineers are accountable to the Project Manager and the Project Architect . The development of the project is very much one of developing a new technology – as such each engineer working on the project is expected to bring their own expertise to the development group . A project manager's job is to manage the project – not to be an expert on the technology . A project architect's job is to connect the components together – not to be an expert on the technology . Granted – they may each have a lot of knowledge about the technology but they are not expected to have the level of expertise of the development engineers . The development engineers are expected to have a good amount of knowledge of their respective technology areas and to ensure that the issues associated with their technological areas are raised within the group .

For example – within a multi media framework running MP3 , MPEG2 & MPEG4 components – the Project Architect may decide to run a asynchronous push system – this is a perfectly valid design choice and gives distinct advantages over a pull system . MP3 & MPEG2 are designed to handle data streams and can work perfectly ok. with a push system . However MPEG4 is file based and requires random file access facilities – ie. a pull system . The engineer , in this example , would ensure that these issues are raised with the Project Manager and Project Architect . The Project Architect would then provide a random access facility within the framework with an asynchronous push response .

## 1.4 Explicit Software Development – Formal Methods

Explicit Software Development ( Formal Methods ) involves the following basic principles :-

- 1) the project is developed in a directly related tree manner in that the basic aims - both business and technical - are directly related to the project development via a documentation tree all the way down through the requirement documents , the project module and component specifications and down to the documentation of the individual functions , function sections and even the individual lines . The project is documented ( within the source code ) before and whilst the project is being developed - not when it is finished . IE. if a comment appears above a line ( relating to the purpose of the line ) it is placed before the code line is developed . The code line must then correspond to the comment line above it .
- 2) the code is laid out in such a manner that it's layout describes the relationships between the various parts of the code - eg. nesting relationships and if statement argument nesting are clearly indicated .
- 3) the code is constructed in a structured and object orientated manner .
- 4) the code uses names that state their purpose - eg. index is used rather than i for an index variable .
- 5) nothing is hidden in the code - whether it be in unnecessary layers , behind macros or anything operating in the background etc. - everything is clearly visible - either via directly associated documentation or via the code itself . Macros are used but they are clear in their use .
- 6) everything in the code and documentation has a clear purpose and a clear place - nothing is left out and nothing is added unnecessarily .

## 1.5 Layout

The code must be neat and tidy – it must have shape and form – formalised interconnections and relationships – self containment and hierarchies – this corresponds to a well organised mind which is vital in good project development . If the code is pleasant to the eye it is easy to read and to follow – hence quick and low in cost to manage – and it's reliability , evolve-ability and expandability is maximised and side effect situations are removed . Spread the code out , make full use of white spaces .

## 1.6 Structured Approach

Imposing a structured and disciplined approach means that the knowns are maximised and the unknowns minimised . What this means is that code is formed into common forms . These forms being 100% effective and being within a known form – thus reliable , easily reviewed and understood .

Software has to be handled as a complex system as such it is vitally important to keep it simple but sophisticated – not simplistic but a big complicated mess .

The software should consist of one or more modules each of which are made up of components . Each of the components having a set purpose and being re-useable . Re-usability requires maximising the locus of validity of the component .

The code should be hierarchical . Lateral connections must not occur – these are one source of side effects . The componentisation and layering must reflect the physical World . Each layer must handle a specific encapsulation or representation of the handling of the data – from the physical at the bottom , through the logical to the abstract or virtual at the top . Each layer should not have visibility of any other layer . Each component must be self contained .

The form of the code must correspond to the structural relationships of the code . IE. the directory layout and the layout within the files must clearly correspond to the structural relationships of the code . The look – layout etc. – of the code must clearly say – in it's form – how the various parts of the code relate to each other .

## 1.7 Code Architecture

Every module must have 1 or more frameworks . Each framework must provide the following facilities :-

- 1) a method of dynamically binding the components .
- 2) management of the communication to and from the components – via events .
- 3) management of the messaging and buffering of the data communicated between the components – via a buffer manager .
- 4) management of the linkage and operation of the components .

The components being bound into the individual frameworks must have the following features :-

- 1) a dynamic binding structure that provides the linkage from and to the external environment . Typically this would consist of pointers to a characteristics table and a function table .
- 2) a component identity section that provides – via macros – a component specific identity .
- 3) a wrapper that provides :-
  - i) a functional interface from the external environment – common starts and ends of functions are provided in between which component specific code is placed .
  - ii) common wrapped functionality placed within the task – ie. the task provides an asynchronous interface to the external environment and a synchronous interface to the components .
  - iii) framework interface functions for the component to communicate – via events – with the framework .
  - iv) an interface to framework based utility functions .
  - v) individual buffer – messaging – input and queues .

It is very important that all approaches must be examined and the best – most appropriate – one be adopted . This requires a very clear set of values and priorities . The code must be engineered – it must be thoroughly and fully thought through – not just plonked down ! The skill of the engineering is directly related to the quality of the thought that goes into the project .

One of the key parts of the design of the code is that problems must be designed out of the code – not into it . Further the design of the architecture must take into account how the framework is going to be used and , as such , it must minimise the amount of code that has to be developed within the dynamically bound components . Generally speaking – the intelligence within the system should largely reside within the framework and the layers above it – the dynamically bound components should just carry out simpler tasks such as receiving or reading , extracting , converting and sending or writing data .

## **1.8 Hermetic – Self Contained Design**

The architecture and code must be designed on hermetic principles . All designs must be localised . Componentised principles must be used . No side connections out of the components – either by relationship or actual – must be used . External connections may only be hierarchical or loop ( open or closed ) . Any design philosophy must be taken to it's full conclusion . No 10% or mixed design philosophies , or construction may , be used . The design must be consistent in it's philosophy .

All code must be self contained – irrespective of where it is in the program . This includes all aspects of the code – scope , definitions , functions , data etc. .

For example a screen design , written in source code , must be specified at a common area – a single ( localised ) area where the whole specification is placed – not a specification that is spread over a number of files . Localising the specification ensures that problems due to side effects are minimised and , ultimately , removed . Spreading the specification over multiple files introduces massive side effect problems . This means that each screen specification has multiple specification points and these each have to be demarcated between each of the screens in each of the files thus greatly complicating the design process – resulting in greatly increased development time and cost and greatly increased chance of bugging .

Each component must compile and act of itself . Each header file must be able to compile of itself – it must not require other headers to be pre-included in order to compile – these must be included from within the header – not from outside it . Likewise with functionality – eg. each component must have start up code that ensures that all components that the component uses are also started up . As such , as much as possible , it should not matter in which order components are explicitly started up – they will always start up successfully . This is , of course , also dependent on the operating system – facilities such as script based start ups along with dynamically linked tasks and components executed from the scripts and automatic execution of components being linked to that haven't been executed - go a long way to ensuring that the start up order is correct .

Each component must contain all of the data and functions that it needs to operate and that is directly concerned with it's operation . All data and functions must be used for it's own individual purpose – ie. data and functionality must not have mixed usage . The components must be organised into explicit layers – separate files for each layer . All the code for each individual layer must be kept together – ie. in the file corresponding layer . Different files may only be used for area parts that are distinctly different – eg. stack up and down directions may be in separate files . Also utility functions – which are used by multiple components – must be placed in a separate file .

## **1.9 Sensibility of Code**

Code must always be sensibly designed . Rules , rather than exceptions , must be used . Where an exception has to occur – there must be a very specific , and unavoidable , reason .

### **1.10 Design Method**

The design must be a combination of top down and bottom up . As such the branches must be constructed to produce a working project – the top down . The branches must then be evolved to add all the required features – the bottom up :-

- 1) Work out the overall functionality required .
- 2) Develop the individual branches ( top down ) .
- 3) Fill out the branches - add the features ( bottom up ) .

This ensures that the project is always working and can be used as part of the development of other modules and can be demonstrated to clients . It also ensures that the project can be evolved and added to with ease .

Always apply common sense and wise-ness to the project design . Don't do anything without reason - always have a reason for choosing the design course .

The code must be easy to read and easy to follow . The architecture and operation should be self evident . Sufficient documentation should be present to allow a person who is not familiar with the project to gain an understanding of the project quickly and easily . Good code design saves massive amounts of time in maintenance and updating .

The code must be designed such that it is always fully working and such that it can handle all conditions . If an error occurs – it must not detect the error and refuse to work – it must detect and handle the error . Where it can work it must work .

The code must be designed to handle all variances of the external conditions and to automatically recover to correspond to changes in the external state . Asynchronous connections and loose coupling must be used .

### **1.11 Asserts**

The code must be instrumented with asserts . The asserts must be set up to check for expected conditions .

The asserts should be compile switch enable-able ( macro based ) and the error conditions be reported via the debugger or via a console output and via log files .

### **1.12 Test Harnesses**

Wherever possible - test harnesses must be constructed . The assert operation must be integrated in with the test harness operation - the test harness operation manager must be able to pick up what the asserts are outputting .

### 1.13 Operating Systems

RTOS's must be used such that resource take up ( eg. malloc's and new's ) and resource releasing ( eg. free's and delete's ) are placed into separate tasks such that the task that takes up resources has a lower priority than the task that releases resources . There must be no resource contention problems . mutex'es must only be used for resource allocation – not for inter-task communication .

Synchronous methods - such as where the use of mutexes , semaphores and event flags occurs - produce the following problems :-

- 1) Lockups . Tasks waiting for inputs .
- 2) Priority inversion - task locks up completely .
- 3) Task blindness - ignoring of inputs other than what are expected - task does not see what is happening externally .
- 4) Going to sleep - tasks just not executing - occurs due to prioritisation problems .
- 5) Task synchronisation problems - tasks operating out of sequence - order of operation of tasks not maintained .
- 6) Task operations out of task context - data not initially protected etc. .

These can be designed out of a project by using asynchronous methods - event queues - mailboxes .

It is often appropriate to use COS's within a RTOS framework . Where a COS can be used it is to be used . A COS will allow the usage of a single stack for the code area . Prioritisation is carried out on the basis of the event message priorities – each COS Task has a priority which determines which task's event messages are processed in priority over other task's event messages .

The projects should use multiple tasks within a RTOS environment . Co-operative Operating Systems – COS's - must be used where appropriate – usually within a RTOS task . COS is basically execute to completion RTOS - ie. a set of tasks operating with the same priority .

Communication between the tasks should be via mailboxes - event queues - and should use events in the following manner :-

- 1) a combined Data and Event Manager should be constructed as a base class .
- 2) the existence of data – new or changed – is notified by an event .
- 3) the event structure should consist of the event number ( 32 bits ) and a parameter ( 32 bits ) .
- 4) the parameter should consist of the identity number of the data being updated – set by the task owning the data . This can be used as an index into a table to directly access the associated handling function for the data – the pointer to the function being contained in a field in the derived structure .
- 5) the data is updated ( new value loaded ) via a member function .

alternatively :-

- 1) the event structure should consist of the event number ( 32 bits ) and a parameter ( 32 bits ) .
- 2) the parameter should be either a number or a pointer to a structure .
- 3) where the parameter points to a structure on the heap ownership is handed over to the receiving task . Ownership of the structure must be clearly managed .

The mailboxes should be connected up to the Event Manager ( contained within the Data Manager ) which will broadcast all events to all tasks that have registered to monitor for the specific event group . Each individual task should process the event detection as quickly as possible . Vectoring – via dynamic binding tables – should be used in preference to conditional checking . As such the structure uses an integrated dynamic binding and event messaging address system .

### 1.14 Re-usability

The code must be designed to be re-usable , evolvable and expandable . The evolution and expansion of the code must not be limited by the structure of the code .

### **1.15 Substantial Coding**

The code must be substantial . Coding is not a question of rearranging the furniture . It is a question of developing the best long term solution .

### **1.16 Noddy Code**

Noddy code - unsubstantial code - must be avoided - it unnecessarily complicates the code .

### **1.17 Big Blob Code**

Big Blob code - must be avoided - it unnecessarily complicates the code .

### **1.18 Spaghetti Code – Big Complicated Messes**

The code must be side effect free . The code must be designed on the basis of rules – not exceptions . Spaghetti code – big complicated messes are not allowed .

### **1.19 Mucky Coding & Code Freakery**

Mucky Coding & Code Freakery are not permitted .

Mucky Coding leads to :-

- 1) Nasty bugs .
- 2) Poor architecture .
- 3) Poor methodology .
- 4) High maintenance costs and long maintenance times .

The ' Keep it Simple ' rule must be used . The code must be easy to understand and update . It must also have long life and be easily expanded . It must be KISS !

### **1.20 Fire fighting**

Fire fighting is not permitted . Bugs may not be hidden within the program . The faulty code must be identified and fixed .

### 1.21 Differentiation

One of the vital aspects of coding standards is to provide a means of differentiating the statement parts . As such the individual parts can be easily recognised . In this manner , for example , function names –

FunctionName( ... )

are differentiated from variable names -

VariableName

which are differentiated from enum names -

ENUM\_NAME

which are differentiated from macro names –

MACRO\_NAME(

and all of these are differentiated from plain text .

As such the individual parts can be easily recognised and easily searched for .

Differentiation also means that the \* - pointer – type modifier is placed next to the type – not the variable or function name . This ensures that the pointer typing is not confused with loading data via a pointer .

Also – file names must not be duplicated – every file must have a unique name .

### 1.22 Consistency

It is vitally important that the code is fully consistent in it's construction – in structure , principles of operation and usage of naming conventions and names and in all other respects .

### 1.23 Use of Macros

Macros may not be used where Enums can be used .

Macros must be used where common methods are being used – where the method is changed or updated the macro is changed and the change is propagated throughout the program . However care should be taken to code debug-ability – ie. macros should not be used such that they hide bugs . Bugs must be exposed , found and fixed . Macros can then be used .

### 1.24 Use of Typedefs and Pointers

Pointers may not be hidden within typedefs – the basic type must be used with the pointer symbol being used to modify it at the variable definition point .

### 1.25 Use of Types and Casting

Variables and fields etc. , where they are being used as part of the same operation – eg. as part of an assignment of one variable with another variable and wherever possible , must be of the same type . Mixing of types – eg. UINT32 & INT32 – with casting to solve the matching problem – is not permitted . Casting may only be used in exceptional circumstances – such as handling compiler typing deficiencies , assigning bit fields and packing and unpacking data .

## **1.26 General Construction**

### **1.26.1 Parameters**

The following placements of parameters must be used :-

- 1) Destinations must be placed before sources .
- 2) Sizes must be placed before locations .
- 3) Parameters must be laid out horizontally and then , if necessary , wrapped around to the next line and aligned vertically .

### **1.26.2 Code Sections**

Variables must be placed in the order that they are used . The return variable must be the last variable declared . Within C++ encoding variables must be localised to their area of use .

Constant tables must be placed directly above the main function that access them .

## 2. **Componentisation**

RTOS Tasks must have the following characteristics :-

- 1) a single mailbox based entry point which is placed within the task loop .
- 2) communication must be via an event based mechanism – via the mailbox .
- 3) the task must convert the event numbers to calls to the associated object data load function - contained within the originating task or within separate event object functions - to load the data from the originating task to the receiving task . This provides a simple mechanism for multiple sources and multiple destinations with the originating sources not requiring any knowledge of where the data is going to .
- 4) full self containment – minimal or no external visibility from outside the task .
- 5) clear and consistently applied management of dynamic data passed out of the task .

### 1.1 **Co-Operative Operating Systems**

In a full statically or dynamically loadable componentisation system – where appropriate , a Co-operative Operating System ( COS ) - asynchronous and synchronous connection based call \ event handler - backbone can be provided to connect up the components . The component and function numbers should be defined within enum tables . A table should be provided to link up the component numbers to the components and within the components function numbers to the functions . Co-operative Operating Systems operate on the basis of Execute to Completion ( of the components ) . Programs \ tasks have their own event queues . Task switching occurs by switching from one queue to another .

Co-operative Operating Systems provides the following advantages :-

- 1) Provides for full modularity by removing all external data and operational dependencies .
- 2) Ensures all modules execute to completion within their time requirements - ie. are not switched out of operation at crucial moments by the task switcher .
- 3) Eliminates re-entrancy problems .
- 4) Eliminates resource contention problems .
- 5) Minimises stack usage .
- 6) Allows for an easy interface to the external environment - external operations are converted to and passed as events .
- 7) Provides for full virtualisation of operations - event message describes operation and passes associated data .
- 8) Localises testing to the components .
- 9) Allows component updates to occur without producing side effects on the rest of the system .
- 10) Provides for full re-useability and full re-configurability .

Co-operative Operating Systems offer a simple solution to multi tasking requirements and should be used where appropriate .

### 1.2 **Principles of Components**

- 1) must be fully self contained - must be able to determine all the parameters for correct operation . EG. an IP component must be able to load it's default IP address from a configuration file .
- 2) must use common interfaces to allow full interchangeability of components .
- 3) must have external visibility of internally used components minimised .
- 4) ideally must use static ( operating system based installation ) or dynamic binding to allow system level specification of facilities .

### **1.3 Layering**

It is very important to divide the project up into layers . The basic layers are :-

- i) virtual .
- ii) abstract .
- iii) logical .
- iv) physical .

### **1.4 Device Drivers**

Device drivers should be constructed such that they have a wrapper and the separate hardware specific code . The principle being that the development time for new device drivers – same device , difcferent hardware – should be minimised .

### **1.5 Operating System Interfaces**

All operating systems should be interfaced via an operating system wrapper . This is to allow for ease of portability .

The operating system should be relied on as least as possible . This is to ensure that code can be quickly ported onto other operating systems .

### 3. **General**

Always write the code such that it will compile successfully in any environment . Ensure that data sizes are explicitly determined and hence assured . Ensure that the byte order is not contrary to any external expectations .

Always match the destination and source types - avoid the use of casting .

Do not hard code the code - use enum's and #define's . Wherever possible enums must be used in preference to #define's .

Where data sizes are being used use the sizeof operator to determine the data size .

When using ASCII characters use the actual characters - eg. 'A' - not their values - 0x41 . This ensures full portability .

Do not use the same variable names at multiple levels within the component . IE. do not use the same variable name for globals , member variables , parameters and local variables . A variable with the name should only occur at one of these levels .

All variable , function and file names must be descriptive – full length – names .

Overloaded conditional operators must have operators for all the conditions concerned - eg. < must have => and > operators defined . Likewise == must have a != operator defined .

Only standard U.S. ASCII characters may be used - no accented characters such as é etc. or other non standard characters such as ↑ etc. may be used .

Ensure that statements display within the screen area . Use the full screen . If statements are longer than the screen line wrap them around – at appropriate points .

### 1.1 Localising Data and Code

- 1) Data and code must be localised as much as possible . This ensures that areas of responsibility are maintained and hence minimises bugs . Hence localise the scope of functions and variables such that they are only visible to those functions and function areas that require to use them .
- 2) If at all possible responsibility for changing individual data must rest with a single component ( owned by that component ) . This avoids co-responsibility problems where differing assumptions produce incompatible changes on the data .
- 3) Single component or class entry points must be provided to the components so that the internal data and functionality can be localised and managed .

### 1.2 Separating of Code and Data

Code and data must be separated . For example - don't use printf statements for displaying each set of text and data . Place the text and data in a scripted form in separate tables and then use a set of display functions operating in a threaded code ( interpretative ) manner . This will save a lot of code space and is very easy to modify.

A display base class should be constructed that interprets the display script . Data and graphics should be displayed using derived classes that use the base class to access display interface functions . The derived classes should be referred to within the display script by object instantiation identification numbers . When the derived display objects instantiate they should register their respective identification numbers ( defined within an enum table ) with the base class .

Key operations - data and control codes - should be passed to a specific function of the derived classes .

General parameters such as fonts , styles , sizes etc. should be stored in the base class .

### 1.3 Structure and Organisation of Code

Separate the functionality out into individual functions . Each function should only carry out one task .

Keep it simple . Look for simple but sophisticated . Avoid simplistic and complex .

Don't use casting unless absolutely necessary . Casting can be used for explicit size conversion but it's usage should be carefully monitored and , if possible , avoided . Common typedef's , instead , must be used . The code must be consistent in it's typing .

Be very careful with the use of indexes and pointers . Always ensure that functions can not operate out of bounds .

Localise all responsibility . If a component constructs an area of data \ object it should have the responsibility of destructing it .

### 1.4 Common Code

Where common code is being used either :-

- 1) replace the code with a macro – care must be taken to ensure that the code is fully debugged so that the macro does not hide the operation .
- or
- 2) use a function .

Never duplicate definitions – enums , macros and types etc. . Never duplicate header files .

## 4. Headers and Section Separators

### 1.1 File Headers

Each file must have a file header of the following format :-

```

/*****
*
* File           : FileName.Extension
*
* Component      : Component Name - the name of this component .
*
* Module         : Module Name - the name of the module which contains the
*                  component .
*
* Description    : File ( component ) Purpose
*
* Copyright (C) : Owner "Date" All Rights Reserved
*
* History        : Date By Addition \ Alteration
*                  Created - baseline date . Version 1.00 .
*
*****/

```

The file name and extension is used to identify the file when it is printed .

The component name identifies the actual component that the file contains . The file name should reflect the component name .

The module name identifies the module to which the component is dynamically bound or to which other components are dynamically bound to - ie this component .

The description describes the purpose of the file or component . Main files must contain the requirement specifications for the project .

The copyright message establishes the copyright ownership of the component .

The history details the creation and changes carried out on the component .

Each file header must be followed by a blank line .

## 1.2 File Sections

Within each file the following sections , if present , must have the following headers :-

```

/* ===== INCLUDES ===== */
/* ===== DEFINES ===== */
/* ===== TYPEDEFS ===== */
/* ===== ENUMS ===== */

/* ===== STRUCTURES ===== */
/* ===== MACROS ===== */
/* ===== GLOBALS ===== */
/* ===== OBJECTS ===== */
/* ===== FUNCTIONS ===== */

```

Each section header must be followed by a blank line .

Each section must be ended by 2 blank lines .

Where functions are grouped into functional areas they must be preceded by a title . Eg. :-

```

/*****
 *
 *   Component Functions
 *
 *****/

```

Where functions are separated by their nesting arrangement the respective function groups must be ended by :-

```

/*****
 *****/

```

thus clearly separating them from subsequent functions that are higher in the nesting arrangement .

### 1.3 **Code , Header and Interface Files**

- 1) The code file must contain only declarative elements .
- 2) All locally only used descriptive elements must be placed within the header file .
- 3) All externally used descriptive elements must be placed within the header file or within a separate interface file .
- 4) Use the following order for the code file :-
  - i) File descriptor .
  - ii) Local function prototypes .
  - iii) Globals .
  - iv) Constants .
  - v) Functions .

This ensures that the sections of the code area are placed in known positions .
- 5) All files must be able to compile independently . This means that they must explicitly include any header files that they require . IE. header files must not rely on prior inclusions of other header files within the code file . The exception of this is system level header files that may have a single level of inclusions - eg. for definition files .

## 1.4 Header Files

- 1) The header file contains only descriptive elements . All physical elements - code and data - are contained within the code ( .c , .cpp or .cxx ) file . The header ( .h , .hpp or .hxx ) file contains #defines , enums , structs , typedefs class defs and prototypes .
- 2) Place a
 

```
#ifndef HeaderFileName_extension
#define HeaderFileName_extension
```

 at the start of the header file with a corresponding
 

```
#endif // HeaderFileName_extension
```

 at the end to prevent multiple and recursive inclusion problems occurring .
- 3) Use the following order for the header file :-
  - i) Includes - run time libraries followed by the system level definition file and then the component header files .
  - ii) Public Typedefs - corresponding to the public member function return and data types .
  - iii) #define and enum constant definitions . Where possible enum's must be used for all constant definitions .
  - iv) Typedefs and structs .
  - v) The class definition .
  - vi) Global externs .
  - vii) Extern function prototypes - must be listed in the order which they occur .
  - viii) Macros - place where appropriate .

This order can be varied – with sub sections ( usually typedef's ) – where appropriate .
- 4) Unless there are clear organisational reasons , such as system level definition files , avoid nesting of header files . Where nesting occurs , if at all possible , keep this down to a single level . Documentation must be maintained as to the specific nesting used .
- 5) All header files must be able to compile independently . This means that they must include any header files that they require .
- 6) System level definition header files must be used . These either include component header files – eg. typedefs , basic system enums , platform specific enums etc. or contain all system level definitions within themselves .
- 7) Module level definition files must be used . These define the interface to the module .

## 1.5 Function Headers

Each function must have a function header of the following format :-

```

/*****
*
* Function      : Function_Name
*
* Description   : Function purpose and operation - including what it alters
*
* Parameters    : Type      Name      Description - including values and
                  direction\s .
*
* Returns       : Type      Name      Description - including values .
*
* History       : Date      By        Addition \ Alteration
                  Created - baseline date .
*
*****/

```

## 5. Functions

- 1) Functions must have a void return type if not returning any data and a void parameter type if no data is to be passed in . This ensures that the return type and parameter type are explicitly stated
- 2) The function return type and function name must be placed on the same line and must be followed by the function parameters .
- 3) The left round bracket of the parameters must be placed next to the function name . This ensures that the function name is differentiated as a function name .
- 4) Parameters must be placed left to right and top to bottom and be vertically aligned . If the function parameters go beyond the line end the succeeding parameters must be placed on the subsequent lines and the types and names must be left justified and lined up in columns .
- 5) The parameter types , names and commas must be separated by tabs and spaces .
- 6) The function must have a single exit point . This ensures that the entry and exit conditions are clearly visible .
- 7) All functions must have prototypes .
- 8) The static specifier , if used , must only be placed in front of the prototype . Likewise with extern's .
- 9) A comment using the function name must be placed on the trailing brace . Parameters ( including void ) indicated by ( ... ) . This ensures that the end of the function is clearly identifiable with the function .
- 10) Functions must be placed in the order in which they are called with the top of the tree occurring first . Internal prototypes must be placed within the code file - at the start - and external prototypes in the header file .
- 11) Functions must be separated into sections with each section having a section header – a title with an optional ( if required ) description .

## 6. Nesting

- 1) All nested sections must be explicitly nested through the use of braces – { & } – and round brackets – ( & ) .
- 2) Braces and bracketed sections must be placed on separate lines . This ensures that all nested sections are clearly identified as such .
- 3) The braces must be placed at the same indentation position as the statement that precedes the nest . Brackets must be placed under their corresponding sections with the linking operator above and to the right of the opening bracket of the next section . This ensures that the nesting relationships are clearly visible . It also ensures consistency in the usage of braces and brackets .

Comments above bracketed sections must be on multiple lines with each line corresponding to each section and ended with the plain language word for the operator linking up the section .

Switch statements must be handled as a dual nest with the first set of braces placed , in vertical alignment , at the switch start and the next set of braces placed , in vertical alignment , at the case start .

The braces for if , for , while and do statements must be placed directly under the start's of the if , for , while and do .

If statements should have a line preceding the statement stating the condition being tested – “ determine whether ... ” . Each nest should have a line stating the result of the test in logical terms - eg. - “ if so - “ - followed by a description of the next statement . If there is a negative condition there should be a corresponding “ if not - “ - followed by a description of the next statement . There should be 2 spaces following the ‘if so’ such that the ‘-’ of the “ if so - “ is directly above the ‘-’ of the “ if not – “ .

- 4) The nested code must be indented 1 indentation . This ensures that the nest stands out . It also ensures that the end of the nest is clearly identified and allows the end to be matched up with the start of the nest .
- 5) Tabs must be used with 1 tab per indentation . This allows the indentation amount to be adjusted . The indentation width must be set to 4 spaces .
- 6) Argument components - such as operators , conditionals and brackets – must be vertically aligned . Connecting parts – such as conditionals – must be placed at the end of the line – to indicate a continuation .
- 7) A comment must be placed on the trailing brace that matches the nesting argument to indicate the section that the brace belongs to and the end of the nest .

Conditionals - the if brace must have the word if followed by the bracketed argument that follows the if key word .

the else brace must have the word else followed by a space followed by the bracketed argument that follows the if key word .

the comments may be truncated and continuation indicated by ‘ ... ’ .

Loop - the first line of the loop statement ( except do statements ) must be copied as a comment after the trailing brace .

Functions - the function name must be placed , as a comment , at the function end brace . No parameters are replaced with ( ) . Parameters are replaced with ( ... ) .

- 8) The number of nesting layers must be minimised and , if possible , should be no more than 3 per function . This minimises the complexity of the function . However this is not a hard and fast requirement

## 7. Conditional Compiles

Conditional compile statements should be spaced out with 1 blank line above and below .

The initial conditional compile statements in a nest must be left aligned .

Where conditional compile statements are nested the conditional compile statements should be indented by 2 spaces .

The alternate ( #else ) and trailing ( #endif ) statements must contain a comment matching the initial ( #ifdef or #ifndef ) argument . EG :-

```
#ifndef      BUILD_GRAB_BAR
#else       // BUILD_GRAB_BAR
#endif      // BUILD_GRAB_BAR
```

## 8. Spreading Out

The code must be spread out . Spaces must be placed between brackets , commas and arguments . This allows the statement and argument parts to be clearly identified . In most circumstances comments must be placed above the code . Only round function brackets are to be immediately followed by a semicolon .

For example :-

```
int FunctionName( void )
{
    UINT            index ;           // general index
    BYTE            packetByte ;      // packet byte - working data
    CMessageBuffer* messageBuffer ;   // pointer to the message buffer

    // section description

    // comment
    for( index = 0 ; index < MAXIMUM_NUMBER ; index++ )
    {
        // comment
        Array[ index ] = index ;
    }// for( index = 0 ; index < MAXIMUM_NUMBER ; index++ )

    // determine whether ~
    if( index == MAXIMUM_NUMBER )
    {
        // if so - ~
    }// if( index == MAXIMUM_NUMBER )
    else
    {
        // if not - ~
    }// else ( index != MAXIMUM_NUMBER )

    switch ( bPacketByte )
    {
        case STX :
        {
            MessageBuffer = new MessageBuffer ;
            break ;
        }// case STX :

        case ETX :
        {
            MessageBuffer->DespatchPacket();
            break ;
        }// case ETX :

        default :
        {
            MessageBuffer->PutByte( bPacketByte );
            break ;
        }// default :

    }// switch ( bPacketByte )

} // FunctionName()
```

A balance must be maintained between spreading the code out and maintaining as much of the code as possible within a single window area . As such unnecessary spreading out , such as spreading out vertically when horizontal spreading out can be used , must be avoided . Emphasis must be placed on making the code as readable as possible . The placement of the code must correspond to the relationships between the parts of the code .

### 1.1 Multiple Lines

Where an expression is split up into multiple lines the operator must be placed at the line end . This ensures that the continuation is flagged and that the subsequent section can be vertically aligned . Eg.:-

```
if( ( a < b ) &&
    ( a > c )      )
{
    ...
} // If( ( a < b ) && ...
```

Where part of an expression is split up indent the part by 2 characters . Eg.:-

```
for( index = 0 ;
    ( index < MAX_BUFFER_SIZE ) &&
    ( ArrayName[ index ].
      FileName           )      ;
    index++              )
{
    ...
} // for( index = 0 ; ...
```

Where an expression consists of conditional or arithmetic parts and these parts are placed on multiple lines these parts must be layed out in a nested format - with the argument sections , operators and brackets lined up vertically to show their relationship . EG. :-

```
if( ( InputEvent.Position_Y           >=
    ( Field->Position_Y   + Field->TEdgeWidth   ) ) &&
    ( InputEvent.Position_Y           <
    ( ( Field->Position_Y   + Field->Height     ) -
      Field->BEdgeWidth   ) ) ) &&
    ( InputEvent.Position_X           >=
    ( Field->Position_X   + Field->LEdgeWidth   ) ) &&
    ( InputEvent.Position_X           <
    ( ( Field->Position_X   + Field->Width     ) -
      Field->REdgeWidth   ) ) ) )
{
}
```

### 1.2 Placement of Variable Definitions

Variable definitions must be placed at the start of the functions . However where the compiler allows variables to be localised they must be placed within the nest where they are only used . This ensures that visibility is minimised .

### 1.3 Lining Up

Vertically line up variables , arguments , operators , nests and comments etc. so that they can be easily distinguished and followed . This maximises the readability of the code .

## 9. Naming Conventions

### 1.1 Function and Variable Names

Function names must be in title case with the first character capitalised .

Static and dynamic ( part of the object ) variables must be in title case with the first character capitalised . Parameters and local variables must be in title case with the first character in lower case .

Names must be chosen that are meaningful and accurate in relation to what they are used for . IE. for an index don't use 'i' use 'index' . If possible use a more descriptive word such as characterIndex . The name must be sufficiently unique that it can't be confused , in a search , with comments .

Within the variable and function names there must be no underscores separating the individual words . Public function and variable names must be prefixed by the component name followed by either an underscore - \_ - or a double colon - :: . This ensures that the name is as compact as possible and ensures that all name parts are easily distinguished .

As such -

All public global and function names must be in the form of :-

ComponentName\_GlobalOrFunctionName

or

ComponentName::GlobalOrFunctionName

and

All private global and function names must be in either the public form or the form of :-

GlobalOrFunctionName

The component name must uniquely identify the component within the system .

## 1.2 Constants and Enumerations

All constants ( enums ) , in general , must be in upper case with the words separated by an underscore .

CONSTANT\_NAME

Constants must be defined using enum's - not #define or const unless absolutely necessary .

Enum names , where used , must have the \_EN suffix .

Where ever possible code the enumeration as a sequence where the value of the current is equal to the previous value by implication . This ensures that if a new entry is put in all subsequent values will be renumbered accordingly .

Enumeration names ( including typedef'ed – C style ) must have the \_EN suffix .

EnumerationName\_EN

## 1.3 Macros

All macros , in general , must be in upper case with the words separated by an underscore . Macros must be followed by round brackets to differentiate them from constants . However this is not a hard fast rule as there are circumstances where macros are used where it is not appropriate that they are indicated in this way – such as framework function names . In these circumstances that macro usage must be explicitly indicated within the comment – eg. ... MACRO'ed .

MACRO\_NAME()

Where enums can be used they must be used , in preference , to defines .

The variables and the macro definitions must be surrounded with brackets to prevent any compile confusion . Macro name portions should be separated with underscores . Macro definitions should be ended with round brackets irrespective of whether there are any arguments . If at all possible macros should not be ended in a ';' to allow them to be placed within lists such as for statement sections . When used there must be a space between the trailing ')' and the ';' . Where function names are macro'ed and not in upper case these should be indicated in the associated comment as being a macro .

## 1.4 Structures

Structure names ( including typedef'ed – C style ) must have the \_ST suffix .

StructureName\_ST

## 1.5 Unions

Union names must have the \_UN suffix .

UnionName\_UN

## 1.6 Types

Type names must have the \_TP suffix .

TypeName\_TP

## 10. Typedefs

Typedefs must be used extensively . Types must not be mixed .

Do not define types that are pointers . Use the base type and place a pointer \* in front of it . When defining a pointer place the \* symbol directly after the variable type .

### 1.1 Enums

Enumerations need not be typedef'd to the enumeration base type . Enum's are often used with masked values – flags , bit group based indexes , etc. – and , as such , some compilers can find problems with some values . Also setting enumerated variables to the enum base size can , by causing filed sizes to be changed , prevent expansion . As such the enum variables should be explicitly set to specific sizes .

## 11. **Public Interfaces**

All public component and class fields , functions and data must be categorised and typedef'd or macro'ed . This allows the types and access methods to be changed .

## 12. Use of Brackets

### 1.1 ( & ) - Round Brackets

The opening bracket – ( – must directly follow what precedes – irrespective of whether this is a function name , a macro name or an if statement etc. . In this situation the keyword etc. – eg. the if – is operating on the argument – on what is in between the brackets – and , as such , must be associated with the argument . Further the statement preserves the ability to be searched for – eg. as if( – as the keyword is distinguished from plain text .

A space must be placed after the opening bracket – ( – and before the closing bracket – ) . Where nested round brackets are used , internal to the argument , use the bracket placements to show the nesting . All variables , operators etc. within the round brackets , as with all areas of the program , must be spaced out .

### 1.2 [ & ] - Square Brackets

The opening bracket – [ – must directly follow what precedes . A space must be placed after the opening bracket – [ – and before the closing bracket – ] . Again – this is to distinguish the array name and to allow for the fact that it is being indexed to be searched for .

### 1.3 { & } - Curly Brackets - Braces

Braces must always be placed on a separate line and at their indent position . Indent positions are left justified – ie. they are directly , vertically , in line with the start of the preceding statement . The code in between the braces must be indented 1 indentation with nests indented correspondingly further . The trailing brace must have as a comment a copy of the first line of the statement preceding the nest . EG. :-

```
first line of statement preceding the nest
{
    first line of statement preceding the nest
    {
        first line of statement preceding the nest
        {
            // copy of first line of statement preceding the nest
        } // copy of first line of statement preceding the nest
    } // copy of first line of statement preceding the nest
}
```

where the trailing comment is only a part of the leading statement it must be suffixed by a space followed by ... ( 3 dots ) . Similarly with trailing function name comments where the function parameters must always be replaced by a space ... a space – eg. // **FunctionName( ... )**

## 13. Statements and Operators

### 1.1 If Statements

If statements must be preceded by a 'determine whether ...' comment. This must state the condition on which the first nest is executed. The first nest must be preceded (within the nest) by a 'if so - ...' comment. The second nest must be preceded (within the nest) by an 'if not - ...' comment. EG. :-

```
// determine whether ...
if( index == MAXIMUM_NUMBER )
{
    // if so - ...
} // if( index == MAXIMUM_NUMBER )
else
{
    // if not - ...
} // else ( index == MAXIMUM_NUMBER )
```

Where nested if statements are being used and are checking the same argument using == or != use, instead, a switch statement.

The 'if so' is followed by 2 spaces and the 'if not' by 1 space. This ensures that the '-'s line up.

### 1.2 Else If Statements

If Else statements essentially form a case statement structure and, as such, do not require nesting. EG. :-

```
// determine whether ...
if( value == ABC )
{
    // if so - ...
} // if( value == ABC )
else if( value == DEF )
{
    // if so - ...
} // else ( value == DEF )
else if( value == GHI )
{
    // if so - ...
} // else ( value == GHI )
```

### 1.3 ? Operator

Avoid the use of the ? operator – use the if and switch statements instead. This provides consistency which makes the code easier (ie. quicker) to follow.

## 1.4 Switch Statements

Ensure that the nesting brackets are placed in left vertical alignment with the statement preceding the nest .

The switch statement must have a default case .

EG. :-

```
switch( variable )
{
    case condition_1 :
    {
        ...
        // exit
        break ;
    } // case condition_1 :
    case condition_2 :
    case condition_3 :
    {
        ...
        // exit
        break ;
    } // case condition_2 : condition_3 :
    ...
    default :
    {
        ...
        // exit
        break ;
    } // default :
} // switch( variable )
```

The first line of the switch and individual case and default statements must be placed as a comment on their respective trailing braces .

## 1.5 For Statements

Except for simple statements for statements should be separated into separate lines . EG. :-

```
for( initial ;
    while ;
    next )
{
    ...
} // for( initial ; ...
```

The first line of the for statement must be placed as a comment on the trailing brace .

The while conditions must be placed such that the individual conditions can be clearly seen . Often this involves placing the individual conditions on separate lines .

## 1.6 Loops

Unless absolutely necessary a single break – exit – point should be used . continue and goto statements must not be used .

## 1.7 Return Statements

A return statement is not a function . As such brackets must not be used - unless absolutely necessary .

```
return returnValue ;
```

A single return - exit - point must be used .

## 1.8 Structure Field Alignment

Care must be taken for field placement . Where necessary structures should have their fields automatically aligned such that their fields can be accessed irrespective of the processor bus size and bus access restrictions - ie. chars padded to shorts and shorts padded to longs such that chars are accessed on 8 bit borders , shorts on 16 bit borders and longs on 32 bit borders . Assumptions must not be made about the byte placement of the data . Unions should be avoided .

## 1.9 Continue and Goto Statements

continue and goto statements may not be used . They are a hang over from assembly language and they violate the basic philosophy of C & C++ based program construction - that being the use of nests and loops .

## 1.10 Endian

The code must be endian independent . As such – if loading big endian data – for a data communication stream , for example , the data must be loaded a byte at a time into the individual fields of the header and frame structure . The data must then be accessed via this structure .

## 1.11 Pragmas

Pragmas should be avoided as they are compiler and processor specific and hence produce non portable code .

## 14. Program Architecture

### 1.1 Task Interconnection

All tasks should be constructed in the following manner :-

- 1) all communication should be asynchronous and via events sent via mailboxes - queues .
  - 2) mutexes , semaphores and event flags must not be used . The only exception to this is that mutexes can be used in memory - heap - management . They can also be used in ISR's .
- Basically :-
- i) there should not be any synchronous blocking points .
  - ii) there should not be any points where priority inversion can occur .
  - iii) there should not be any points where the task execution order can become out of sequence .
  - iv) there should not be any points where a task goes to sleep - where it is blind to external communications and events etc. .

### 1.2 C++ - Object Orientation - Encapsulation & Relationships - Componentisation

All modules should be constructed in the following manner :-

- 1) full object orientation including inheritance of characteristics must be used .
- 2) there must be a static root pointer to the main object . This being instantiated on start up .
- 3) there must not be any other static variables . Any variable that would normally be considered static must be placed in the tree .
- 4) all objects must be linked into the object tree - in a linked list - branching from the root . The linked list structure must be able to accommodate parents , siblings and children .
- 5) all objects must have a object type identifier .
- 6) when a branch is destructed the object type identifier must be used , via the registry , to call the destructor functions in the objects in the child linked lists so that the whole branch is destructed in the background .
- 7) all objects must have a unique object instantiation identifier . This can be used for searching for specific objects in the tree .
- 8) all objects must be self relative - if any object requires to access a parent object it must traverse the tree upwards until it finds the parent object it requires .
- 9) objects must not be instantiated statically or locally . Where an object is just being used locally - it must be instantiated off the heap and be destructed at the end of the function .

### 1.3 General Issues

- 1) all heap operations should be handled via a memory manager . The Memory Manager should manage fragmentation by using a recycler - by providing the following :-
  - i) it should allocate memory out of open ended pools - pools not restricted in size .
  - ii) the pools should be organised on a linked list basis .
  - iii) the allocation basis should be on the basis of the block sizes .
  - iv) an index table should be used to select the pool starts .
  - v) when a block is de-allocated - and where the pool does not have the number of de-allocations at it's high water mark - the block's area in memory should be retained for future use .

## 15. Comments

The drill down approach must be used for all documentation including comments . All documentation should be contained within and associated with the code being developed .

Comments must be associated with the following :-

- 1) Modules .
- 2) Components .
- 3) Enumerations .
- 4) Typedefs .
- 5) Defines \ Macros .
- 6) Variables .
- 7) Prototypes .
- 8) Functions .
- 9) Sections .
- 10) Lines .

Every line in a enumeration or a structure definition should have a comment to the right of it .

Every variable should have a comment above or to the right of it .

Every section should have a comment preceding it describing the purpose of the section .

Every line of active code should have a comment line preceding it and vertically aligned with it . If at all possible use the `//` comment prefix . Keep the number of lines for the comment as small as possible .

Comments above bracketed sections must be on multiple lines with each line corresponding to each section and ended with the plain language word for the operator linking up the section – eg. :-

```
// if a equals b or
```

```
// c equals d
```

If statements should have a line preceding the statement stating the condition being tested – “ determine whether ... “ . Each nest should have a line stating the result of the test in logical terms - eg. - “ if so - “ - followed by a description of the next statement . If there is a negative condition there should be a corresponding “ if not - “ - followed by a description of the next statement . There should be 2 spaces following the ‘if so’ such that the ‘-’ of the “ if so - “ is directly above the ‘-’ of the “ if not – “ .

Every nest end must have , as a comment , a copy of the line at the start of the nest . Else statements should have a `// else (` comment with the argument following . If the line is too long ( to fit into a single line on the screen ) it may be truncated . The ‘...’ sequence is to be used to indicate continuation .

## 1.7 The Keys to Commenting

The keys to commenting are :-

- 1) comment the individual lines to specify what it is intended that they do . The purposes of this are :-
  - i) to aid in the development of the individual statement - often not required for experienced software engineers but often useful for recent graduates .
  - ii) to aid in remembering the purpose of the statement - not only for code maintenance purposes - updates etc. - but also if alterations are made to the line and the original purpose of the line is lost the comment can be referred to to recover the original purpose of the line .
  - iii) to provide a methodological approach in program design .
  - iv) to provide a check - if the statement doesn't match the comment then one of them is wrong - find out which one and fix it .
- 2) comment sections and functions to describe their purposes .
- 3) comment files and modules to describe their architecture , relationships and purposes .
- 4) always remember that the ultimate purpose of commenting is to minimise the time that a person , who is not familiar with the software , can come on stream and be working with the software .

## 1.8 Temporary Code

Where code is put in or commented out for purposes of debugging place the @@@ tag , followed by your initials , next to the code . Before the code is released check for the presence of this pattern and deal with and remove the code and associated commenting .