A White Paper from
Emerson Network Power™
Embedded Computing

Demonstrating Software Reliability: A White Paper

Presented By
Jim Darroch (jimd@artesyncp.com)

Acknowledgements
Norman McWhirter (norman@artesyncp.com)

EMERSON™
Network Power

Demonstrating Software Reliability: A White Paper

Contents

Demonstrating Software Reliability: A White Paper

1 Document Overview

1.1 Introduction

The terms "Reliability", "High Availability" and "Carrier Grade" have become common in the context of communications equipment in particular, and the broader context of Embedded Systems as a whole. In many cases, the terms are used synonymously, which is confusing – as they are quite distinct.

In addition, systems engineers tend to be very comfortable in dealing with the reliability of Hardware (in terms of MTBF and MTTR). Software is not so mature in the field of measured reliability. This is a cause for concern, as Software is the cause of at least as many system failures as Hardware!

This paper attempts to define the terms associated with the overall reliability of a system (comprising multiple hardware and software components), and illustrates how Software Reliability is factored in. The methods used to determine Software Reliability for High Availability solutions and the techniques used to model System Reliability and Availability are explored

As a result, recommendations are made for software engineering practice for High Availability (or "Carrier Grade") applications.

1.2 Definitions

Availability In general terms, availability is the fraction of time a component or system is able to perform its intended function. A system with "5 Nines" availability is capable of performing its mission more than 99.999% of the time, roughly equivalent to missing five minutes of operation per year.

In strict mathematical terms:

$$Availability = \frac{MTBF}{MTBF + MTTR}$$

And is usually expressed on a scale of 0 (always unavailable) to 1 (always available). Thus, "5 nines" is expressed as 0.99999

MTBF The "mean time between failures" of a repairable component or system, measured in hours. The MTBF is generally estimated from a model. An example for hardware is calculation according to the Bellcore Reliability Method. For software components, an example is the model developed by Hughes Aircraft Co., the application of which forms the majority of this White Paper.

N.B. "Reliability" can be viewed as a synonym for MTBF

MTTR The "mean time to repair" of a repairable component or system, measured in hours. The MTTR spans the period from the point the fault occurred, through the discovery of the fault, the mobilization of service personnel and the replacement of the failed component, to the reintroduction to service.

Failure Rate, λ(t) The failure rate function gives the expected number of failures per hour at time t. For software, which doesn't become worn out, λ(t) is usually expressed as the constant λ. Mathematically,

$$\lambda = \frac{1}{MTBF}$$

Reliability calculations are generally simpler using μ rather than MTBF.

Repair Rate, μ Similarly, expressing the MTTR as a rate simplifies the modelling and calculations:

$$\mu = \frac{1}{MTTR}$$

Reliability, R(t) The reliability function expresses quantatively the reliability of a component or system. $R(t)$ is the probability that the component has failed by time t. For constant failure rate systems:

$$R(t) = e^{-\lambda t}$$

R(t) is generally only used for non-repairable components or systems.

Carrier Grade A collection of attributes which define the suitability of a system for deployment in core telecom applications. The attributes are:

Availability – Generally defined as "5 nines" (i.e. providing service 99.999% of the time) calculated by statistical means and validated by field data

Performance – Has performance which meets industry demands. Performance may be characterised by capacity, throughput, latency and jitter

Scalability – Is scalable to account for traffic and subscriber growth

Serviceability – Provides in-built diagnostic tools to allow rapid fault detection and repair, automatically isolate faulty components and allow upgrade and reconfiguration (all without loss of service)

Security – Provides sufficient security to prevent loss of service from malicious attack

1.3 References

[1] Bellcore Reliability Method

[2] Rome Laboratories Technical Report RL-TR-92-15, February 1992.

2 System Reliability

Reliability is an attribute of systems and their components. An application may have specific reliability requirements, expressed as MTBF or as Availability, and the challenge for the system architects and designers is to devise an architecture capable of meeting those requirements. Although reliability is less obvious than some of the other attributes (performance, power consumption etc), it is amenable to design and management, and the goal of this White Paper is to give a basic understanding of how reliability should be managed in High Availability software projects.

The overall system reliability can be estimated from the individual component reliabilities by modelling the interactions between the components using one or more of the building blocks described in sections 2.1 to 2.4.

Throughout this document, it is assumed that all failures are statistically independent of each other.

2.1 Series Components

In this scheme, all of the components must operate for the system to operate. For example, a computer and its operating system must both be operating for the system to be functional. The aggregate failure rate of a system consisting of k components is then:

$$\Lambda = \sum_{i=1}^{k} \lambda_i$$

i.e. the simple arithmetic sum of the individual failure rates.

A more general scheme may be created if not all of the components are required to be operating continuously. Let $a_i(t)$ be a function whose value is 1 if at time t component i is required to be operational, and 0 otherwise. The aggregate failure rate is given by:

$$\Lambda(t) = \sum_{i=1}^{k} \lambda_i a_i(t)$$

2.2 Sequential Components

Here, components 1 through to k are active consecutively: once component i finishes, component i+1 starts. The aggregate failure rate is

$$\Lambda(t) = \lambda_{i+1}; \quad where \quad t \in [t_i, t_{i+1}]$$

2.3 k-out-of-n Components

In this scheme, k out of n components must be operational for the system to be operational. If all n components have the same reliability function $r(t)$, the aggregate reliability is given by:

$$R(t) = \sum_{i=k}^{n} \binom{n}{i} [r(t)][1 - r(t)]^{n-1}$$

Once $R(t)$ is available, the aggregate failure rate is given by

$$\Lambda(t) = \frac{\lambda \dfrac{n!}{(n-k)!(k-1)!} e^{-k\lambda t}(1-e^{-\lambda t})^{n-k}}{R(t)}$$

Alternatively, an approximate failure rate based on time to first failure is:

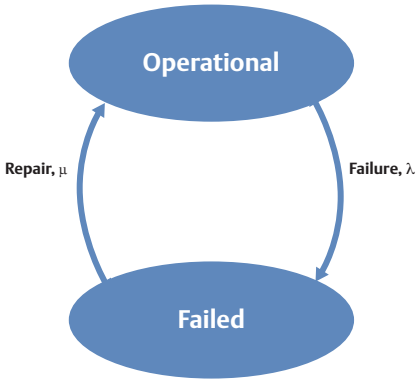
$$\Lambda = \frac{\lambda}{\sum_{i=k}^{n} \frac{1}{i}}$$

2.4 Markov State Diagrams

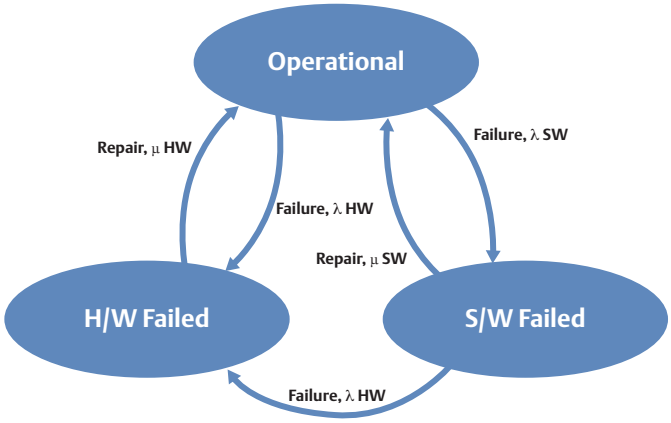
Markov State Diagrams provide a method to combine the simple structures above into more complicated systems, including repairable systems.

A Markov process is a stochastic (i.e. random) process in which the future state or condition is completely determined by the current state, regardless of how the current state was achieved. A Markov State Diagram is a set of discrete states, each of which has transitions to neighbouring states that are taken according to a table of probabilities.

For example, a simple repairable system could be modelled as:



A more complicated example would be a combined hardware and software system. Note that the "repair" time here is the time required to bring the software back into service, and not the time required to find and eliminate the bug.



The Markov model allows us to differentiate between different failure modes and model their effects. For example, the hardware repair rate μHW, requiring the mobilization of repair personnel, may be very much greater than the software repair rate, μSW, which may be as simple as an automatic reboot. Conversely, the software failure rate, λSW, may be very much higher than the hardware failure rate, λHW.

Note the additional transition between the S/W Failed and H/W Failed states. This additional, non-obvious transition, hints at one of the main problems with Markov models: the number of states and transitions tends to increase exponentially as the complexity of the problem increases, making it difficult to check the model for correctness.

The failure rate for a Markov model can be calculated either analytically or by Monte Carlo simulation. For the analytical method, if p_{ij} is the probability of state j following the operation of state i, the "steady state" probabilities (i.e. the long term probability of the model being in a particular state) is found by solving the system of equations:

$$\pi_i = \sum_{j \in \Omega} \pi_j p_{ji} \qquad i \in \Omega$$
$$\sum \pi_i = 1$$

(Think of this as follows: the probability of being in state i is the sum of the probability of entering state i from state j given the probability that you were in state j to start with. Keep guessing values of π until the set of equations is true).

The failure rate is then given by

$$\Lambda = \sum_{i \in \Omega_f} \sum_{j \in \Omega} \pi_j \lambda_{ji}$$

where Ω is the set of all states in the model, and Ω_f is the set of all failure states.

3 Allocation of Reliability Requirements

We can use the equations and tools from Section 2 to determine the reliability requirements of a software element of a system, given that

1. The reliability of the hardware is known (or has been estimated/calculated)
2. The desired reliability of the system as a whole has been quantified.

This section serves as an example of how this may be performed

3.1 System Description

The example is based on a High Availability system using redundant "blades". Each "blade" comprises of two single board computers (physically connected) and software running on both, as follows:

1. The carrier board, with a calculated MTBF of 305,929 hours, and associated failure rate of 3.268 x 10^{-6}.
2. A mezzanine module, with a calculated MTBF of 757,336 hours, and associated failure rate of 1.320 x 10^{-6}.
3. A software component, whose reliability must be determined.

3.2 Determining Desired System Failure Rate

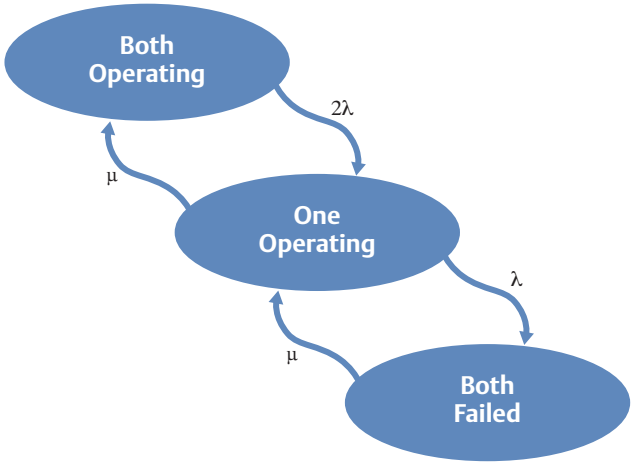
If we assume that the system is required to be "5 nines" (0.99999) available, and that the average repair time is four hours, we get the following requirement for system MTBF, and hence failure rate:

$$Availability = \frac{MTBF}{MTBF + MTTR}$$
$$MTBF = \frac{Availability}{1 - Availability} MTTR$$
$$MTBF = 399996$$
$$\Lambda = \frac{1}{MTBF}$$
$$\Lambda = 2.5 \times 10^{-6}$$

Thus, we have a failure rate for the system as a whole of 2.5 x 10^{-6} failures per hour.

3.3 Dual Blade Topology

If we further assume that the High Availability system architecture is based on two identical blades of which at least one must be operating for the system to be available, we can estimate the failure rate requirement for each blade. This is achieved with the aid of the following Markov model (since an individual blade is repairable):



Note that the transition from "Both Operating" to "One Operating" is at a rate of 2λ. This is because the initial state has two blades operating in parallel and either one may fail.

With μ = 0.25 (i.e., a four hour MTTR), choosing λ = 0.00056 gives an aggregate Λ of 2.5 x 10^{-6}. In other words, given a desired failure rate for the entire system of 2.5 x 10^{-6}, the failure rate for each blade is 560 x 10^{-6}.

3.4 Estimated Software Failure Rate

Assuming an individual blade is only operational if all its components are operational (i.e. the carrier, module and software), we can determine the failure rate allowed for the software component (given our definition of failure rate as 1/MTBF, and calculated MTBF values for the carrier and module from Section 3.1)

$$\lambda_{blade} = \lambda_{carrier} + \lambda_{module} + \lambda_{SW}$$
$$\lambda_{SW} = 560 \times 10^{-6} - 3.268 \times 10^{-6} - 1.320 \times 10^{-6}$$
$$\lambda_{SW} = 0.555 \times 10^{-3}$$

Or, 1800 hours MTBF.

Now that we have our requirement for the reliability of the software component, we need to show that the developed software meets or exceeds the requirement. This is treated in the following chapters.

4 Reliability Demonstration Testing

A reliability demonstration test is a test to determine if a component or system has achieved a specified level of reliability. Typically, a test plan specifies a test environment, operational profile, test duration and the number of permissible failures. The system or component is then operated according to the plan and the number of observed failures recorded. The component is rejected if the number of observed failures is above the number of permissible failures.

The environment and operational profile selected for the test should be as close to the expected operating conditions as possible and the number of permissible failures is often specified as zero, but how do we determine the test duration? If the test duration is too short, there is a risk that we accept unreliable components. Conversely, if the test is too long there is a risk that we reject components that actually meet the reliability requirements.

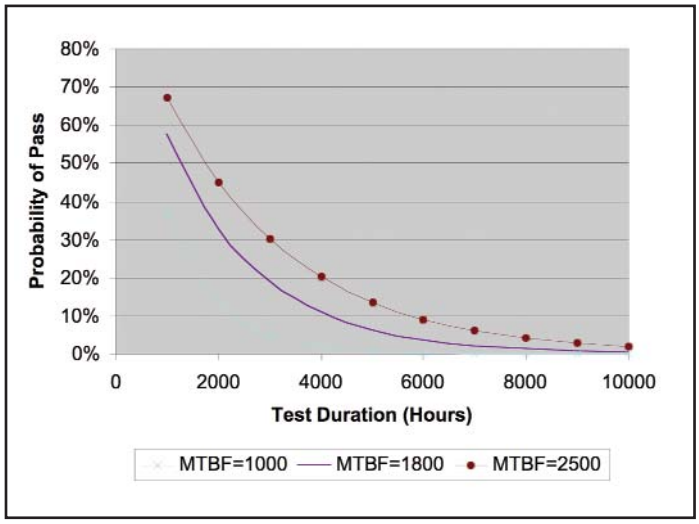
If we assume that the components failure rate, λ, is constant (for software, this implies a frozen code base), then the probability of observing exactly i failures in time t obeys Poisson's law:

$$P_i = e^{-\lambda t} \frac{(\lambda t)^i}{i!}$$

The probability of n or fewer failures is given by the Poisson cumulative distribution function:

$$F(n,\lambda,t) = \sum_{i=0}^{n} \frac{(\lambda t)^i e^{-\lambda t}}{i!}$$

This function is plotted below, showing the probability of software passing a test (n=0) of the given duration for three pieces of software with different actual failure rates.



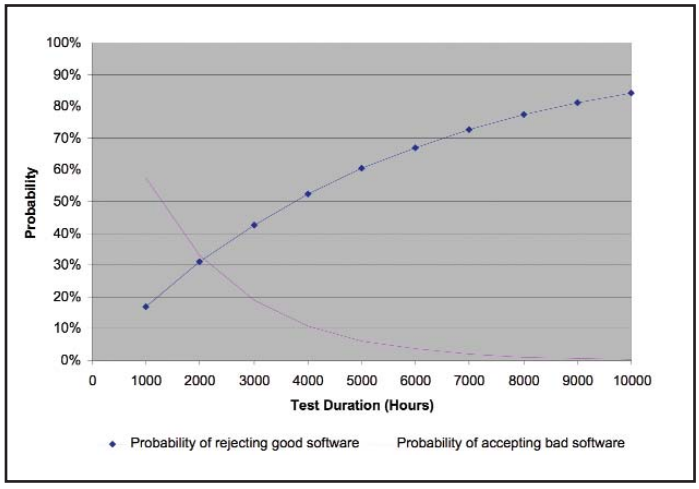
Note that for our example system (with an MTBF of 1800 hours), a test duration of approximately 1500 hours carries a 50-50 chance of the component passing the test, even though the component has exactly the required failure rate. This is obviously an unsatisfactory

test: if we run for 1500 hours and observe no failures, we simply do not know if the failure rate is λ or greater than λ. From a customer's perspective, there is a 50-50 risk of accepting software with a higher than tolerable failure rate. To reduce that risk down to 20% (i.e. one in five tests would accept inadequate software), we would require to test for 3000 hours.

As software producers, we would like to increase the chances of the software passing its reliability demonstration test. Since shortening the test duration increases the risk to the customer, the only alternative we have left is to *increase* the reliability of the component.

Let λ_0 be the required failure rate, λ_1 be the improved failure rate to which the software is designed, and λ be the actual (unknown) failure rate. The risk of software with a failure rate of λ_1 being rejected is the risk that the software generates more than n errors during the test, i.e. $1 - F(n, \lambda_1, t)$. We want to choose a design failure rate and test duration such that the risk of accepting inadequate software is acceptable to the customer and that the risk of rejecting adequate software is acceptable to us as producers.

The graph below shows λ_0 at 0.555×10^{-3} and λ_1 at $\lambda_0 \div 3$, i.e. our software is designed to have an MTBF three times greater than the requirements.



Note that even at 3000 hours of testing, there is a 20% chance of incorrectly accepting inadequate software and a 40% chance of rejecting reliable software. To reduce these risks to 15% for both parties, the software would need to be designed to nearly 12 times the requirement.

Note also that we are testing the software component of the system here. If we were to test the system to its failure rate of 2.5×10^{-6}, we would need to test for approximately 700,000 hours, equivalent to ten systems operating error free for eight years, to get to the 20% confidence level.

This leads us to our next question: how do we know that the software has progressed to a stable enough state to attempt a Reliability Demonstration Test? As software producers we would like a way to determine when the test and debug cycle can be stopped and the software put forward for the reliability demonstration test. To make an informed decision we require a method for estimating and monitoring the increase of reliability as bugs are removed.

5 Failure Rate Estimation and Reliability Growth

We would like an estimate for λ for our software primarily so that we know when to put a candidate forward for the reliability demonstration test. Additionally, it may be useful to estimate λ at earlier stages in the development life cycle. Specifically, if we can estimate the value of λ at the start of the system test and debug phase (λ_{t0}) and know the λ_1 required for entry to the demonstration test, we can estimate the test and debug effort required to improve the software from λ_{t0} to λ_1, a process known as "reliability growth".

5.1 Estimation from Project Metrics

The Rome Laboratories Technical Report, *Reliability Techniques for Combined Hardware and Software Systems* [2], performed an analysis of nine Hughes Aircraft software projects and attempted to correlate the observed initial failure rates with a variety of project metrics, ranging from the number of statements in the requirement specification through number of flaws identified during review, to the average years of experience of the programmers.

The report suggested models to estimate λ_{t0} at the preliminary design phase, detailed design phase and after the code and unit test

phase. Applying these models to the example High Availability project[1] gives estimates for λ_{t0} of 24.3, 17.9, and 18.0 failures per hour respectively. Unfortunately the models are likely to be valid only within the confines of Hughes Aircraft's software development procedures: according to the models, the only way to reduce the estimate of λ_{t0} is to spend more effort on requirements analysis; even spending more time on unit test prior to system test *increases* the estimated failure rate.

For our purposes, using an "industry standard" defect density of 6 errors per 1000 lines of code may suffice as a starting point for estimating the test and debug duration during the project planning stage. By tracking the defects found during testing and the time at which each defect is found, it is possible to revise the estimates for λ_{t0} in addition to estimating λ. The revised λ_{t0} could then be used during the planning stage for future, similar, projects.

5.2 Basic Execution Time Model

In order to estimate the reliability of software, we require a mathematical model to describe the distribution of defects and the rate at which those defects cause failures. The model described in [2] starts from the fundamental notion that for software to fail it must be executing, and then proceeds to estimate the rate at which the executing processor encounters defects.

The first step in the model is to determine the "linear execution frequency" f, which is defined to be the inverse of the length of time required to execute the entire program text if each instruction were to be processed sequentially. This can be calculated from the number of source lines of code LoC, the processor's clock speed F, the number of instructions per clock cycle I, and a language expansion ratio (2.5 for C) as follows:

$$f = \frac{F \times I}{2.5 \times LoC}$$

If the software under test has a defect density of omega, the total number of defects in the program would be $\omega_0 = \omega\, LoC$, and the rate at which defects are encountered by the processor would be $f\omega_0$.

Unfortunately the program is not executed sequentially, and not every defect encountered will result in a failure (for example an unprotected division may be classified as a defect, but will only become a failure if the divisor is zero). To account for the effect of program structure and program state a constant K is introduced, representing the fault exposure ratio. The "industry average" value of 4.20×10^{-7} is used until sufficient historical data is available to estimate K locally. The rate at which failures are encountered is therefore:

$$\lambda_{t0} = fK\omega_0$$

5.3 Reliability Growth

Reliability growth is the improvement in a software system's reliability as test, debug and repair activities are performed. In general, every software failure discovered during system test will lead to investigative and repair activities that will act to reduce the defect density ω, and therefore increase the reliability. The investigative and repair activities are not perfect however, and may introduce new faults or fail to identify the defect completely. If we were to continue the test and debug cycle until all faults have been removed (which may potentially require an infinite amount of time) we may end up with a number, ν_0 larger than the original number of defects, ω_0.

To see how the failure rate changes as defects are removed, assume that the software is in system test and that μ defects have so far been identified and repaired. The Basic Execution Time Model assumes that each defect is equally likely, and so the current failure rate can be calculated as

$$\lambda(\mu) = \lambda_{t0}\left(1 - \frac{\mu}{\nu_0}\right)$$

[1] The example in this paper is abstracted from a real High Availability project - these numbers are based on a real product under development!

The rate at which the failure rate reduces per defect is given by the derivative:

$$\frac{d\lambda}{d\mu} = -\frac{\lambda_0}{\nu_0}$$

The rate $\lambda_{t0} / {}_{-0}$ is called the fault reduction ratio, ß.

To determine the estimated failure rate as a function of time, we can rewrite $\lambda(\mu)$ using ß, and can define $\lambda(t)$ by noting that ß and $\upsilon 0$- are constant and that μ is also a function of both time and the current failure rate:

$$\lambda(\mu) = \beta(v_0 - \mu)$$
$$\lambda(t) = \beta[v_0 - \mu(t)]$$
$$\mu(t) \equiv \int_0^t \lambda(s).ds$$

Solving for $\lambda(t)$, leaves us with:

$$\lambda(t) = v_0 \beta e^{-\beta t}$$

5.4 Estimating Failure Rate During System Test

To inform our decision to release a candidate build for the Reliability Demonstration Test, we would like to estimate the current failure rate of the software based on the information gathered to date. To achieve this, the test engineer must record the cumulative CPU time consumed since the start of system test as each failure is detected (t_1, t_2, t_3, ...t_n) and provide a cumulative CPU time for the end of testing, t_e. The maximum likelihood estimate for ß is found by solving the first equation below, and the maximum likelihood estimate for υ_0 is found by substituting ß into the second equation.

$$\frac{n}{\beta} - \frac{nt_e}{e^{\beta t_e} - 1} - \sum_{i=1}^{n} t_i = 0$$

$$v_0 = \frac{n}{1 - e^{-\beta t_e}}$$

The failure rate $\lambda(t)$ can then be directly calculated and compared against the desired failure rate λ_1. The defect density, ω, may also be estimated for use in planning future projects.

It is important to remember that the value of $\lambda(t)$ is only an estimate of the true failure rate λ, and that the final determination of the software failure rate, and hence MTBF, is achieved through the reliability demonstration test.

6 Summary and Recommendations

In summary – given the requirements for *availability* of a system as a whole, and values (whether estimated, calculated, etc) for the *reliability* of the majority of its components, it is possible to establish the reliability of those components for which information is missing. In the majority of systems, this will be the software component.

This is a vital part of any system design project – without this information, it is near impossible to give quantified numbers for the required reliability of the software without resorting to guesswork (no more than 5 minor defects per 1000 lines of code).

Once the reliability requirements have been established, the system must be shown to meet those requirements. The suggested method involves:

1. Modelling the failure modes of the system
2. Tracking defect detection and correction during development, integration and system test in order to illustrate reliability growth
3. Performing a reliability demonstration test. The criteria for entering such a test phase, and the duration of the testing to be carried out are not hard and fast rules. Some judgement must be exercised in determining what is an acceptable level of risk in this activity – striking a balance between a low risk of failing to meet reliability targets (not testing for long enough) against a high risk of testing beyond the point where the software is acceptable.

In order to implement these steps (of modelling and testing), the following practices should be adopted while developing software for High Availability (or "Carrier Grade") Applications.

1. The overall system reliability requirements should be decomposed in to sub-system reliability requirements using the techniques shown in System Reliability and Allocation of Reliability Requirements.
2. The acceptance criteria and operational profile for the reliability demonstration test should be determined, and specifically the degree of risk should be agreed, as shown in Reliability Demonstration Testing.
3. Estimates for system test duration may be calculated from "industry standard" or locally derived metrics, as shown in Estimation from Project Metrics and Basic Execution Time Model.
4. During the system test and debug phase, cumulative CPU usage should be recorded and noted for each failure detected. This should then be used to form an estimate of the current failure rate, as shown in Estimating Failure Rate During System Test.
5. After the system test and debug phase, the software must undergo a Reliability Demonstration Test to verify the sub-system reliability.
6. Metrics from the project should be maintained to provide better estimates for defect density for use in subsequent projects.

Emerson Network Power.
The global leader in enabling Business-Critical Continuity™.

- AC Power Systems
- Connectivity
- DC Power Systems
- **Embedded Computing**
- Embedded Power
- Integrated Cabinet Solutions
- Outside Plant
- Power Switching & Controls
- Precision Cooling
- Services
- Site Monitoring
- Surge & Signal Protection

Emerson Network Power, Embedded Computing
8310 Excelsior Drive ▪ Madison, WI 53717-1935 USA
US Toll Free: 1-800-356-9602 ▪ Voice: +1-608-831-5500 ▪ FAX: +1-608-831-4249
Email: info@artesyncp.com

www.artesyncp.com