



**Queensland Government**  
**Treasury**

Office of Gaming Regulation

EGM

Program Signature Algorithms

Version 1.2

8 The State of Queensland, Queensland Office of Gaming Regulation 2001

Designed: 26 May, 1997 Robert Larkin  
Revised: February 12, 2001  
Reference: psa12.doc  
Printed: May 24, 2006



---

## **EGM Program Signature Algorithms**

### **Policy:**

It is QOGR policy for EGMs in Queensland to be able to produce a signature of their program for verification using an acceptable signature algorithm.

### **Purpose:**

The purpose of this document is to list some of the acceptable EGM Program Signature Algorithms (PSAs) for use with the QOGR EGM communications protocol

Note, the Program Signature Algorithms defined in this document generate a public signature or A fingerprint for any type of input data, such as a software program, file, document, message digest or raw data of any kind For simplicity, the remainder of this document will refer only to program data as the algorithm input.

### **Scope:**

This document is applicable to EGM software and jackpot system developers.

### **General**

All PSAs in use throughout QLD will be given a unique identification code Primarily there will be only two algorithms and they are described in the following Sections of this document Both algorithms are basically the standard Cyclic Redundancy Check (CRC) algorithms but with a slight modification to further randomise the result

However, in order to expedite signature calculations, variations of the listed algorithms and/or new algorithms will be considered by the Office at any time Accepted new algorithms will be given a unique I D. number

### **PSAs and Electronic Gaming Machines (EGMs)**

EGM manufacturers should refer to the QOGR EGM Protocol document to determine which of the algorithms your EGM should use for the QCOM Protocol implementation.

What data must be included in the overall EGM program signature? The signature must encompass all data stored within the EGM for which it is physically possible to be executed by the EGM=s primary CPUs. The primary CPU refers to the CPU(s) which control or can affect play/gamble outcome and critical meters. This also includes data which can be loaded and executed from EGM RAM. It does not include peripheral device programs such as banknote or coin acceptor program data

As stated in the EGM protocol document, the EGM may calculate its program signature in any desired program memory byte/bit order and in parallel over separate systems. The EGM must combine multiple signature results into one result via modulo 2 addition (XOR).

**Please supply QOGR with exact details of how the EGM performs its PSA over its program with respect to the binary images of the EGMs software produced by the compiling the source code.**

---

## Algorithms

Acceptable Program Signature Algorithms are as follows:-

### The PSA16 Algorithm

This algorithm is a slightly modified CCITT 16 bit CRC algorithm operating a byte at a time. Each partial CRC result (ie. after each byte) is exclusive OR'ed with the current unsigned 16 bit wide, 8 bit check-sum. The CCITT 16 bit CRC algorithm is provided in the Appendix, please note the modification below.

Eg. In "C" notation:-

```
unsigned int Checksum = 0; // 16 bit unsigned int
unsigned int CRC = Seed; // 16 bit unsigned int Seed or Key
unsigned char *b = StartOfProgramSpace; // pointer to 8 bit unsigned char
do {
    Checksum += *b; // "b" is 8 bits (one byte) of Program Data
    // CRC_Calc is a function performing a standard 16 bit Cyclic Redundancy Check
    CRC = CRC_Calc(*b, CRC) ^ Checksum; // Note the XOR ^ !!!
    b++;
} while (!EndOfProgramSpace);
```

### The PSA32 Algorithm

Similar to PSA16, this algorithm is a slightly modified 32 bit CRC algorithm operating a byte at a time. Each partial CRC result (ie. after each byte) is exclusive OR'ed with the current unsigned 32 bit wide, 8 bit check-sum. The 32 bit CRC algorithm is provided in the Appendix, please note the modification in the example below.

ie. In rough "C" notation for the PSA32 :

```
unsigned long int Checksum = 0; // 32 bit unsigned int
unsigned long int CRC = Seed; // 32 bit unsigned int Seed or Key
unsigned char *b = StartOfProgramSpace; // pointer to 8 bit unsigned char
do {
    Checksum += *b; // "b" is 8 bits (one byte) of Program Data
    // CRC_Calc is a function performing a standard 32 bit Cyclic Redundancy Check
    CRC = CRC_Calc(*b, CRC) ^ Checksum; // Note the XOR ^ !!!
    b++;
} while (!EndOfProgramSpace);
```

### Acceptable Variations of the PSAs

As mentioned in the first section of this document, many variations of the above algorithms may be considered acceptable. For example, in the PSA32 algorithm above, performing the signature and checksum 32 bits at a time instead of 8 bits at a time is acceptable. Speed is the overall consideration as an EGM remains disabled upon connection to the network until it returns a valid signature. Manufacturers should feel free to experiment to reduce their program signature calculation time. An almost unlimited number of signature algorithms can be supported by the QOGR and QCOM protocol. However, at this time the monitoring systems are limited to 32 bit signature algorithms.

---

### **Known issues with the above algorithms**

The PSAs defined in this document can be classified as >key-dependent one way hash functions= The keys are public so the PSAs therefore should not be confused with a Message Authentication Code (MAC). They also should not be confused with Digital Signature Algorithms as used in applied cryptography

Using the above PSAs for all zero data in any application should be avoided.

16-32 bit algorithms are a fairly weak for a one way hashing function, so to compensate the above algorithms are used on a periodic basis with a changing seed It is not a major concern if more than one program works out to have the same signature for a given seed But it should be noted that this is possible

The above algorithms should only be used in situations where a trusted third party keeps a master copy of the programs being verified so any doubt over a program can be removed by performing a comparison of the two copies.

---

## Appendix A

### A.1 The CCITT 16 bit CRC Algorithm

This is the standard 16 bit Cyclic Redundancy Check used in many applications and is provided here for reference. This algorithm is also used for QOGR EGM Protocol message CRC generation

**Note, the CRC 16 algorithm not the PSA16 algorithm, but only forms a significant part of the overall method.**

#### "C" Source Code

```
///  
extern unsigned int CRCccittTable[256];  
  
#define CRCccittMACRO(b, crc) (CRCccittTable[(crc ^ b) & 0xff] ^ (crc >> 8))  
// returns (unsigned int) CCITT CRC, b is an unsigned char, crc is an unsigned int  
  
unsigned int CRCccitt(unsigned char b, unsigned int CRC);  
unsigned int CRCccittBlock(unsigned char *b, unsigned int CRC, unsigned int Length);  
  
///  
#include "crccitt.h"  
  
// 16 bit CCITT CRC repeats every 32767 iterations when performed over uniform data  
// ie there is one reserved value  
  
// This CCITT CRC routine is basically a LSB first HDLC CCITT CRC but with the following differences:  
// 1) The Seed is bit reversed  
// 2) The result is bit reversed  
  
// Some short examples  
// data 0x0F seed 0xAA55 result is FD75  
// data 0x01 seed 0x0000 result is 1189  
// data 0x00 0x00 seed 0xffff result is 0xF0B8  
  
unsigned int CRCccittTable[] =  
{  
    0x0000, 0x1189, 0x2312, 0x329B, 0x4624, 0x57AD, 0x6536, 0x74BF,  
    0x8C48, 0x9DC1, 0xAF5A, 0xBED3, 0xCA6C, 0xDBE5, 0xE97E, 0xF8F7,  
    0x1081, 0x0108, 0x3393, 0x221A, 0x56A5, 0x472C, 0x75B7, 0x643E,  
    0x9CC9, 0x8D40, 0xBFDB, 0xAE52, 0xDAED, 0xCB64, 0xF9FF, 0xE876,  
    0x2102, 0x308B, 0x0210, 0x1399, 0x6726, 0x76AF, 0x4434, 0x55BD,  
    0xAD4A, 0xBCC3, 0x8E58, 0x9FD1, 0xEB6E, 0xFAE7, 0xC87C, 0xD9F5,  
    0x3183, 0x200A, 0x1291, 0x0318, 0x77A7, 0x662E, 0x54B5, 0x453C,  
    0xBDCB, 0xAC42, 0x9ED9, 0x8F50, 0xFBEB, 0xEA66, 0xD8FD, 0xC974,  
    0x4204, 0x538D, 0x6116, 0x709F, 0x0420, 0x15A9, 0x2732, 0x36BB,  
    0xCE4C, 0xDFC5, 0xED5E, 0xFCD7, 0x8868, 0x99E1, 0xAB7A, 0xBAF3,  
    0x5285, 0x430C, 0x7197, 0x601E, 0x14A1, 0x0528, 0x37B3, 0x263A,  
    0xDECD, 0xCF44, 0xFDDF, 0xEC56, 0x98E9, 0x8960, 0xBBFB, 0xAA72,  
    0x6306, 0x728F, 0x4014, 0x519D, 0x2522, 0x34AB, 0x0630, 0x17B9,
```

```

0xEF4E, 0xFEC7, 0xCC5C, 0xDDD5, 0xA96A, 0xB8E3, 0x8A78, 0x9BF1,
0x7387, 0x620E, 0x5095, 0x411C, 0x35A3, 0x242A, 0x16B1, 0x0738,
0xFFCF, 0xEE46, 0xDCDD, 0xCD54, 0xB9EB, 0xA862, 0x9AF9, 0x8B70,
0x8408, 0x9581, 0xA71A, 0xB693, 0xC22C, 0xD3A5, 0xE13E, 0xF0B7,
0x0840, 0x19C9, 0x2B52, 0x3ADB, 0x4E64, 0x5FED, 0x6D76, 0x7CFF,
0x9489, 0x8500, 0xB79B, 0xA612, 0xD2AD, 0xC324, 0xF1BF, 0xE036,
0x18C1, 0x0948, 0x3BD3, 0x2A5A, 0x5EE5, 0x4F6C, 0x7DF7, 0x6C7E,
0xA50A, 0xB483, 0x8618, 0x9791, 0xE32E, 0xF2A7, 0xC03C, 0xD1B5,
0x2942, 0x38CB, 0x0A50, 0x1BD9, 0x6F66, 0x7EEF, 0x4C74, 0x5DFD,
0xB58B, 0xA402, 0x9699, 0x8710, 0xF3AF, 0xE226, 0xD0BD, 0xC134,
0x39C3, 0x284A, 0x1AD1, 0x0B58, 0x7FE7, 0x6E6E, 0x5CF5, 0x4D7C,
0xC60C, 0xD785, 0xE51E, 0xF497, 0x8028, 0x91A1, 0xA33A, 0xB2B3,
0x4A44, 0x5BCD, 0x6956, 0x78DF, 0x0C60, 0x1DE9, 0x2F72, 0x3EFB,
0xD68D, 0xC704, 0xF59F, 0xE416, 0x90A9, 0x8120, 0xB3BB, 0xA232,
0x5AC5, 0x4B4C, 0x79D7, 0x685E, 0x1CE1, 0x0D68, 0x3FF3, 0x2E7A,
0xE70E, 0xF687, 0xC41C, 0xD595, 0xA12A, 0xB0A3, 0x8238, 0x93B1,
0x6B46, 0x7ACF, 0x4854, 0x59DD, 0x2D62, 0x3CEB, 0x0E70, 0x1FF9,
0xF78F, 0xE606, 0xD49D, 0xC514, 0xB1AB, 0xA022, 0x92B9, 0x8330,
0x7BC7, 0x6A4E, 0x58D5, 0x495C, 0x3DE3, 0x2C6A, 0x1EF1, 0x0F78
};

```

```

unsigned int CRCccitt(unsigned char b, unsigned int CRC)

```

```

{
return CRCccittMACRO(b,CRC);
}

```

```

unsigned int CRCccittBlock(unsigned char *b, unsigned int CRC, unsigned int Length)

```

```

{
int i;

for (i = 0; i != Length; i++) {
    CRC = CRCccittMACRO(b[i],CRC);
}
return CRC;
}

```

---

## A.2 The 32 bit CRC Algorithm

This is the standard 32 bit Cyclic Redundancy Check used in many applications and is provided here for reference.

**Note, the CRC 32 algorithm here is not the PSA32 algorithm, but only forms a significant part of the overall method.**

### "C" Source Code

///**CRC32.h**

```
extern unsigned long int CRC32Table[256];
```

```
#define CRC32MACRO(b, crc) (CRC32Table[((int)crc ^ b) & 0xff] ^ ((crc >> 8) & 0x0FFFFFFF))  
// returns unsigned long crc value, b unsigned char, crc unsigned long int
```

```
// This macro below is equivalent to arj & pkzip CRCs (ie. it inverts the input bytes)  
// #define CRC32MACRO(b, crc) (CRC32Table[((int)crc ^ b ^ 0xff) & 0xff] ^ ((crc >> 8) | 0xff000000))
```

```
unsigned long int CRC32(unsigned char b, unsigned long int CRC);  
unsigned long int CRC32Block(unsigned char *b, unsigned long int CRC, unsigned int Length);
```

///**CRC32.c**

```
#include "crc32 h"
```

```
/*
```

```
 * Copyright (C) 1986 Gary S Brown You may use this program, or  
 * code or tables extracted from it, as desired without restriction  
 */
```

```
/* First, the polynomial itself and its table of feedback terms. The */  
/* polynomial is */  
/*  $X^{32}+X^{26}+X^{23}+X^{22}+X^{16}+X^{12}+X^{11}+X^{10}+X^8+X^7+X^5+X^4+X^2+X^1+X^0$  */  
/* Note that we take it "backwards" and put the highest-order term in */  
/* the lowest-order bit. The  $X^{32}$  term is "implied"; the LSB is the */  
/*  $X^{31}$  term, etc. The  $X^0$  term (usually shown as "+1") results in */  
/* the MSB being 1. */
```

```
/* Note that the usual hardware shift register implementation, which */  
/* is what we're using (we're merely optimizing it by doing eight-bit */  
/* chunks at a time) shifts bits into the lowest-order term. In our */  
/* implementation, that means shifting towards the right. Why do we */  
/* do it this way? Because the calculated CRC must be transmitted in */  
/* order from highest-order term to lowest-order term. UARTs transmit */  
/* characters in order from LSB to MSB. By storing the CRC this way, */  
/* we hand it to the UART in the order low-byte to high-byte; the UART */  
/* sends each low-bit to high-bit; and the result is transmission bit */  
/* by bit from highest- to lowest-order term without requiring any bit */  
/* shuffling on our part. Reception works similarly. */
```

```
/* The feedback terms table consists of 256, 32-bit entries. Notes: */  
/* */  
/* The table can be generated at runtime if desired; code to do so */
```

```

/* is shown later. It might not be obvious, but the feedback */
/* terms simply represent the results of eight shift/xor opera- */
/* tions for all combinations of data and CRC register values. */
/* */
/* The values must be right-shifted by eight bits by the "updcrc" */
/* logic; the shift must be unsigned (bring in zeroes). On some */
/* hardware you could probably optimize the shift in assembler by */
/* using byte-swap instructions. */

```

```

unsigned long int CRC32Table[] = { /* CRC polynomial 0xedb88320 */
0x00000000, 0x77073096, 0xee0e612c, 0x990951ba, 0x076dc419, 0x706af48f, 0xe963a535, 0x9e6495a3,
0x0edb8832, 0x79dcb8a4, 0xe0d5e91e, 0x97d2d988, 0x09b64c2b, 0x7eb17cbd, 0xe7b82d07, 0x90bf1d91,
0x1db71064, 0x6ab020f2, 0xf3b97148, 0x84be41de, 0x1dad47d, 0x6ddde4eb, 0xf4d4b551, 0x83d385c7,
0x136c9856, 0x646ba8c0, 0xfd62f97a, 0x8a65c9ec, 0x14015c4f, 0x63066cd9, 0xfa0f3d63, 0x8d080df5,
0x3b6e20c8, 0x4c69105e, 0xd56041e4, 0xa2677172, 0x3c03e4d1, 0x4b04d447, 0xd20d85fd, 0xa50ab56b,
0x35b5a8fa, 0x42b2986c, 0xdbbbc9d6, 0xacbcf940, 0x32d86ce3, 0x45df5c75, 0xdcd60dcf, 0xabd13d59,
0x26d930ac, 0x51de003a, 0xc8d75180, 0xbfd06116, 0x21b4f4b5, 0x56b3c423, 0xcfba9599, 0xb8bda50f,
0x2802b89e, 0x5f058808, 0xc60cd9b2, 0xb10be924, 0x2f6f7c87, 0x58684c11, 0xc1611dab, 0xb6662d3d,
0x76dc4190, 0x01db7106, 0x98d220bc, 0xefd5102a, 0x71b18589, 0x06b6b51f, 0x99bfef4a5, 0xe8b8d433,
0x7807c9a2, 0x0f00f934, 0x9609a88e, 0xe10e9818, 0x7fa0d8bb, 0x086d3d2d, 0x91646c97, 0xe6635c01,
0x6b6b51f4, 0x1c6c6162, 0x856530d8, 0xf262004e, 0x6c0695ed, 0x1b01a57b, 0x8208f4c1, 0xf50fc457,
0x65b0d9c6, 0x12b7e950, 0x8bbeb8ea, 0xfcb9887c, 0x62dd1ddf, 0x15da2d49, 0x8cd37cf3, 0xfbd44c65,
0x4db26158, 0x3ab551ce, 0xa3bc0074, 0xd4bb30e2, 0x4adfa541, 0x3dd895d7, 0xa4d1c46d, 0xd3d6f4fb,
0x4369e96a, 0x346ed9fc, 0xad678846, 0xda60b8d0, 0x44042d73, 0x33031de5, 0xaa0a4c5f, 0xdd0d7cc9,
0x5005713c, 0x270241aa, 0xbe0b1010, 0xc90c2086, 0x5768b525, 0x206f85b3, 0xb966d409, 0xce61e49f,
0x5edef90e, 0x29d9c998, 0xb0d09822, 0xc7d7a8b4, 0x59b33d17, 0x2eb40d81, 0xb7b75c3b, 0xc0ba6cad,
0xedb88320, 0x9abfb3b6, 0x03b6e20c, 0x74b1d29a, 0xeada54739, 0x9dd277af, 0x04db2615, 0x73dc1683,
0xe3630b12, 0x94643b84, 0x0d6d6a3e, 0x7a6a5aa8, 0xe40ecf0b, 0x9309ff9d, 0x0a00ae27, 0x7d079eb1,
0xf00f9344, 0x8708a3d2, 0x1e01f268, 0x6906c2fe, 0xf762575d, 0x806567cb, 0x196c3671, 0x6e6b06e7,
0xfed41b76, 0x89d32be0, 0x10da7a5a, 0x67dd4acc, 0xf9b9df6f, 0x8ebeeff9, 0x17b7be43, 0x60b08ed5,
0xd6d6a3e8, 0xa1d1937e, 0x38d88c2c4, 0x4fdff252, 0xd1bb67f1, 0xa6bc5767, 0x3fb506dd, 0x48b2364b,
0xd80d2bda, 0xaf0a1b4c, 0x36034af6, 0x41047a60, 0xdf60efc3, 0xa867df55, 0x316e8eef, 0x4669be79,
0xcb61b38c, 0xbc66831a, 0x256fd2a0, 0x5268e236, 0xcc0c7795, 0xbb0b4703, 0x220216b9, 0x5505262f,
0xc5ba3bbe, 0xb2bd0b28, 0x2bb45a92, 0x5cb36a04, 0xc2d7ffa7, 0xb5d0cf31, 0x2cd99e8b, 0x5bdeae1d,
0x9b64c2b0, 0xec63f226, 0x756aa39c, 0x026d930a, 0x9c0906a9, 0xeb0e363f, 0x72076785, 0x05005713,
0x95bf4a82, 0xe2b87a14, 0x7bb12bae, 0x0cb61b38, 0x92d28e9b, 0xe5d5be0d, 0x7cdcefb7, 0x0bdbdf21,
0x86d3d2d4, 0xf1d4e242, 0x68ddb3f8, 0x1fda836e, 0x81be16cd, 0xf6b9265b, 0x6fb077e1, 0x18b74777,
0x88085ae6, 0xff0f6a70, 0x66063bca, 0x11010b5c, 0x8f659eff, 0xf862ae69, 0x616bffd3, 0x166ccf45,
0xa00ae278, 0xd70dd2ee, 0x4e048354, 0x3903b3c2, 0xa7672661, 0xd06016f7, 0x4969474d, 0x3e6e77db,
0xaed16a4a, 0xd9d65adc, 0x40df0b66, 0x37d83bf0, 0xa9bcae53, 0xdeb99ec5, 0x47b2cf7f, 0x30b5ffe9,
0xbdbdf21c, 0xcabac28a, 0x53b39330, 0x24b4a3a6, 0xbad03605, 0xcdd70693, 0x54de5729, 0x23d967bf,
0xb3667a2e, 0xc4614ab8, 0x5d681b02, 0x2a6f2b94, 0xb40bbe37, 0xc30c8ea1, 0x5a05df1b, 0x2d02ef8d
};

```

```

unsigned long int CRC32(unsigned char b, unsigned long int CRC)
{
return CRC32MACRO(b,CRC);
}

```

```

unsigned long int CRC32Block(unsigned char *b, unsigned long int CRC,unsigned int Length)
{
int i;

```

```

for (i = 0; i != Length; i++)
CRC = CRC32MACRO(b[i],CRC);

```

---

```
return CRC;  
}
```

### A.3 16 & 32 bit CRC and PSA Examples

Some examples of the CRC CCITT, CRC-32, CRC16a.PSA and CRC32a.PSA algorithms

40 Octets filled with "0x00", Length = 40 bytes

```
Seeds = 0xffff, 0x1234 ..... CRC CCITT = 0x9BA1, 0x0F61
Seeds = 0xffff, 0x1234 ..... CRC16a.PSA = 0x9BA1, 0x0F61
Seeds = 0xffffffff, 0x12345678 ..... CRC-32 = 0x1613C24E, 0xBEC53640
Seeds = 0xffffffff, 0x12345678 ..... CRC32a.PSA = 0x1613C24E, 0xBEC53640
char pkt_data[40]= {0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00};
```

40 Octets filled with "0xff", Length = 40 bytes

```
Seeds = 0x0000, 0x1234 ..... CRC CCITT = 0xFDE6, 0xF287
Seeds = 0x0000, 0x1234 ..... CRC16a.PSA = 0x0825, 0x0744
Seeds = 0x00000000, 0x12345678 ..... CRC-32 = 0x653C71C2, 0xDBF94782
Seeds = 0x00000000, 0x12345678 ..... CRC32a.PSA = 0x75E30C7A, 0xCB263A3A
char pkt_data[40]= {0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff};
```

40 Octets counting: 1 to 40, Length = 40 bytes

```
Seeds = 0x0000, 0x1234 ..... CRC CCITT = 0x0374, 0x0C15
Seeds = 0x0000, 0x1234 ..... CRC16a.PSA = 0xC552, 0xCA33
Seeds = 0x00000000, 0x12345678 ..... CRC-32 = 0xA6581D74, 0x189D2B34
Seeds = 0x00000000, 0x12345678 ..... CRC32a.PSA = 0x949208D4, 0x2A573E94
char pkt_data[40]= {0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08,0x09,0x0a,0x0b,0x0c,0x0d,0x0e,0x0f,0x10,0x11,0x12,0x13,0x14,0x15,0x16,0x17,0x18,0x19,0x1a,0x1b,0x1c,0x1d,0x1e,0x1f,0x20,0x21,0x22,0x23,0x24,0x25,0x26,0x27,0x28};
```

It is also important to verify the following:-

16843010 Octets filled with "0xff", Length = 16843010 bytes

```
Seeds = 0x0000, 0x1234 ..... CRC CCITT = 0xD949, 0x6F13
Seeds = 0x0000, 0x1234 ..... CRC16a.PSA = 0x37CA, 0x8190
Seeds = 0x00000000, 0x12345678 ..... CRC-32 = 0xD5781E9F, 0xE5AA45DE
Seeds = 0x00000000, 0x12345678 ..... CRC32a.PSA = 0x632521B2, 0x53F77AF3
```

---

## **Revision History**

Version 1.2     R Larkin February 12, 2001  
                  Converted to Word  
                  Document is to be generally released  
                  Made general clarifications prior public release of this document

Version 1.1     R Larkin 29 January, 1998  
                  Fixed up copyright notice as per policy  
                  Added additional PSA examples  
                  Minor clarifications elsewhere, refer redline etc.

Version 1.0     R Larkin 30 May 1997