The Use of Overloaded Software Operators for Error Detection and Correction

Scott A. McDermott, Luis G. Jordán, Kalle Anderson
AeroAstro, Inc.
20145 Ashbrook Place, Ashburn, VA 20147; (703)723-9800
scott.mcdermott@aeroastro.com, luis.jordan@aeroastro.com, kalle.anderson@aeroastro.com

ABSTRACT: Performing error detection and correction (EDAC) for single-bit errors in software instead of hardware has both advantages and cost. The largest advantage is the ability to use off-the-shelf computing hardware, with high capability and low cost but no hardware-based EDAC. The largest cost is in speed and complexity, where the software takes many instructions (coding effort) and many cycles (time) to perform the EDAC function that can be performed single-cycle in hardware.

By using the object-oriented software concept of overloaded operators, the complexity can be almost entirely removed, and the speed potentially improved, making software EDAC a much more useful option. In any language that supports overloaded operators, a set of types can be created which represent EDAC-protected versions of basic variable types. These types would include the basic bits of the original variable type, plus error checking bits, wrapped into a single object. Any arithmetic operation which uses these protected types would then use a special version of the operation (addition, dereferencing, assignment, and so forth) that takes error-protection bits into account.

TRADITIONAL EDAC

In current small-space missions, where using a processor from a radiation-hardened upset-immune foundry is not viable, the spacecraft processor choice quickly reduces to two options. Each option has its own benefits and drawbacks: the purchase of an off-the-shelf processor board with Error Detection And Correction (EDAC) hardware included; or, the design and development of a custom board with this same hardware feature (Figure 1). This choice is particularly bothersome at the micro- and nano-sat level, or in situations where the processor is for a small subsystem, where the ideal solution would actually be a single-chip embedded controller. A single-chip solution does not allow for hardware EDAC to be inserted.

EDAC is used to guard primarily against Single Event Upsets (SEUs). SEUs can occur in one of four fundamental types of registers:

1. A critical function register, such as the microprocessor program counter, which will definitely and almost immediately cause the unit to malfunction. No known type of EDAC, beyond SEU immunity in the chip itself, will guard against this.

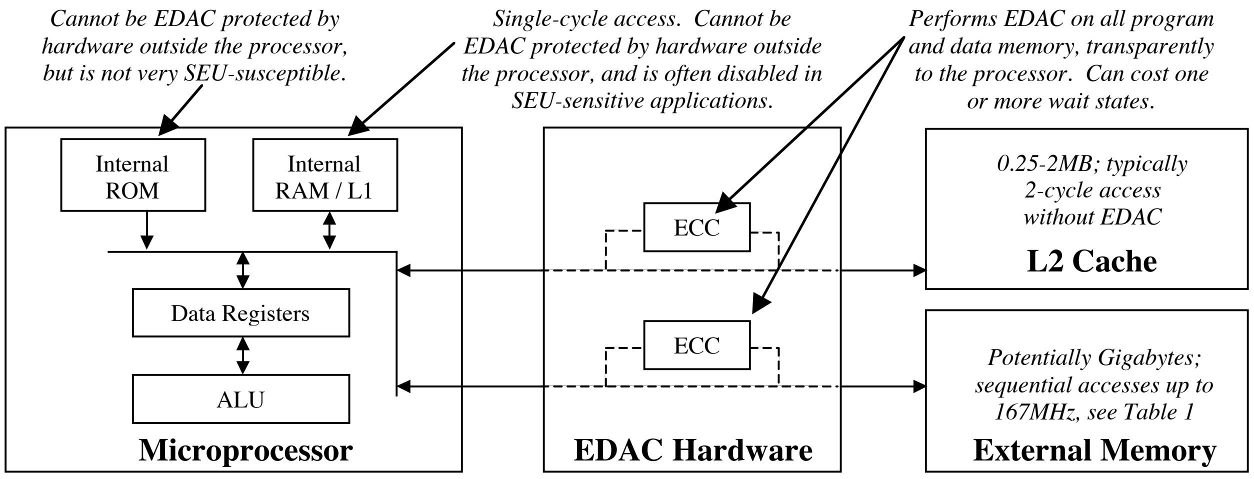
Cannot be EDAC protected by hardware outside the processor, but is not very SEU-susceptible.
Single-cycle access. Cannot be EDAC protected by hardware outside the processor, and is often disabled in SEU-sensitive applications.
Performs EDAC on all program and data memory, transparently to the processor. Can cost one or more wait states.
Internal ROM
Internal RAM / L1
Data Registers
ALU
Microprocessor
ECC
ECC
EDAC Hardware
0.25-2MB; typically 2-cycle access without EDAC
L2 Cache
Potentially Gigabytes; sequential accesses up to 167MHz, see Table 1
External Memory

Figure 1. Traditional Hardware EDAC

2. A device-internal data register, including Arithmetic Logic Unit (ALU) registers and any internal memory. Hardware EDAC cannot protect against errors here, so typically, the amount of this memory that is used is minimized.
3. An external memory register. This is the type of register protected by hardware EDAC circuitry.
4. An unused memory or other register, whose value is irrelevant.

One important note in this list is under #2: for traditional EDAC approaches, on-board RAM (L1 cache, sometimes L2 cache, and/or memory-mapped single-cycle access RAM) *must be disabled* as there is no way to perform EDAC on this memory. Put another way, systems which use EDAC on external memory but still use processor-internal RAM are hybrid approaches where mass storage is protected but operational memory (the frequently-referenced data in cache) is not. As on-chip cache size increases and the operating voltages / feature size decrease, the likelihood of an unrecoverable single-bit error in cache increases[1].

Hardware EDAC

Error Correction Codes (ECCs) are employed in EDAC hardware to perform their function. The most common ECC for memory systems is a Single Error Correction, Dual Error Detection (SECDED) Hamming code, which has the following advantages:

- It will correct a one-bit error anywhere in a word
- It will detect a two-bit error where either error occurs anywhere in a word
- It will detect a significant number of errors greater than two bits
- It operates on a whole binary word in parallel, as opposed to many ECCs (such as Cycling Redundancy Checks or Forward Error Correction codes) which operate on serial bit streams
- It requires only six error-checking bits to protect a 32-bit data word

Hamming ECCs can be implemented combinatorially, that is, with a series of logic gates that pass data from memory to CPU (correcting single bit errors) and from CPU to memory (creating error-check bits) without needing to wait for a clock. However, for a 32-bit data word, the memory-to-CPU path will require at least 4 stages of 4-input logic gates, plus pin input and output times, totaling at least 5ns of delay in current technology. For systems running at 100MHz memory bus speed and above, as many modern processors do, this is likely to introduce a full-cycle delay in memory access times, and in worst case halve the throughput to external memory. Compounded with the impact of disabling internal memory, this can severely reduce processor effectiveness. However, since modern microprocessors are greatly overpowered for their typical uses on small spacecraft, this "efficiency impact" is usually ignorable. For example, the STPSat-1 central computer, a PowerPC 750, runs all spacecraft bus functions using only approximately 5% of its performance capacity.

DIVERGENCE BETWEEN COMPUTER TECHNOLOGY AND EDAC NEEDS

The development of microprocessor technology in the last decade may reasonably be described as "pull more onto the chip". More memory, more functionality, more calculation units are pulled inside the boundaries of the microprocessor itself, so that the vast increases in semiconductor density and speed can translate into increased performance. Inter-chip communication has not kept pace: the capacitive effects inherent in creating a conduit from one chip through a pin to a board to another pin on another chip prevent per-line speed from increasing beyond a certain limit. This pattern is shown in Figure 2 below.

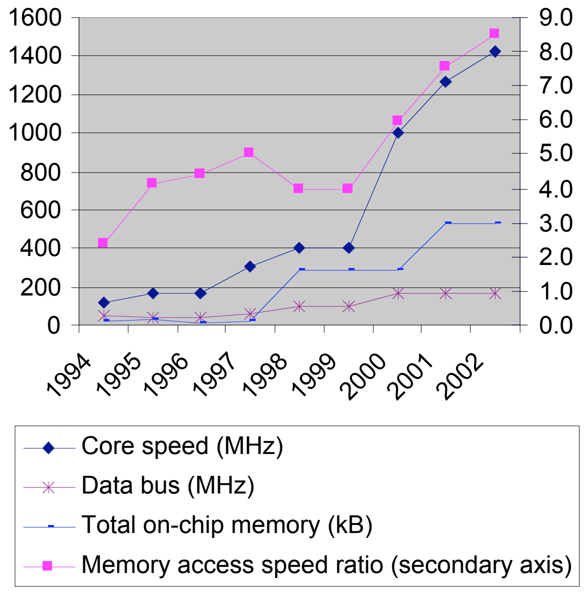

Figure 2. Microprocessor parameters over time; PowerPC 603 to 7447a

What this chart indicates is that over the last decade, while processor speeds have increased by a factor of 12, external memory access speeds have only increased by a factor of 3.3. Since on-chip memory is single-cycle access, the "Memory access speed ratio" indicates how much faster internal RAM accesses are relative to external memory – from 2.4 times faster in 1994 to 8.5 times faster today. The amount of memory on-chip has gone up an enormous 33 times. And a final note which this chart cannot show: the increase in external memory access speed, small as it is, is due largely to optimization of memory accesses *in order to update on-chip memory*. That is, the external memory speed increases are only available for the kind of predictable, sequential accesses that come from refilling an L1 cache line, not from the random accesses that an actual software program would use.

A similar pattern can be seen in processor power efficiency, described as the number of (in millions) instructions the processor can perform per Watt it consumes. This MIPS/W factor has increased, for the PowerPC line, from 7.3 to 74.7 over the timespan shown; that is, for the same Wattage, ten times as many instructions can be performed today compared to ten years ago.

All of these trends indicate that external EDAC is going to introduce more and more problems as time goes on: either chip-internal memory is left unused, severely impacting the performance of the machine; or, if processor-internal memory is used, it is going to represent an ever-increasing exposure of the system to SEU faults.

SOFTWARE EDAC USING OPERATOR OVERLOADING

If an efficient software method for performing EDAC were available, the above trend would reverse: increases in on-chip memory and MIPS/W would result in more useful, instead of more wasted, processor performance. That is, if the software itself were performing error detection and correction on the variables it was using, then it would not matter where those variables were stored: in on-chip memory, off-chip memory, even in the ALU data registers themselves. The downside, of course, is that unlike an efficient set of combinatorial logic performing a Hamming ECC, the processor must itself spend cycles during every variable operation checking to see if the variable is healthy.

With the trends in processor capability shown earlier, this downside is expected to lessen over time. This then leads us to consider what the best way might be for software to perform the EDAC function. One approach is simply to ignore all preformed variable types offered by a language, opting instead for specialized types and functions to support them. In C, for example, the following code would perform `X=A+B+C` using EDAC:

```
typedef struct {
    int src,err;
} ProtInt;

/* performs EDAC and returns good value */
int ProtIntRead(ProtInt val);

/* creates error-check value based on val */
int ProtIntErrmake(int val);

ProtInt ProtIntSet(int val);
    ProtInt result;
    Result.src = val;
    Result.err = ProtIntErrmake(val);
}

ProtInt ProtIntAdd(int val1, int val2) {
    ProtInt result;
    result = ProtIntSet(ProtIntRead(val1) +
ProtIntRead(val2));
    return(result);
}

void main(void) {
    ProtInt X,A,B,C;
    A = B = C = ProtIntSet(5);
    X = ProtIntAdd(A,ProtIntAdd(B,C));
}
```

There are somewhat more efficient ways to approach this in a function-oriented language – having ProtIntAdd take a variable number of arguments, for instance – but ultimately, the code will remain something close to unreadable, full of “set” and “add” and other operations-become-functions. The number of errors that would creep into the software simply due to its convoluted appearance would likely exceed the operational errors due to bit flips.

In order to leave the appearance of the source code sensible, then, we can introduce a preprocessor stage, where the source code can read `X=A+B+C` but the compiler receives `X = ProtIntAdd(A, ProtIntAdd (B,C))`. This recovers useful source code, but is rife with potential for confusion. For instance, `foo(*ptr++)` could translate to

```
foo (ProtIntAdd (ProtIntDeref (ptr),
 ProtIntSet (1)));
```

– which is wrong – or to

```
foo (ProtIntDeref (ptr) );  ProtIntAdd (ptr,
 ProtIntSet (1));
```

which creates any amount of confusion during debugging, as there is an entire new expression added invisibly to the programmer. Similar problems occur if a postprocessing step is added, where the compiler output is transformed to perform EDAC operations instead of simple operations.

Having tried a few simple approaches, and found them lacking for various reasons, we can take a step back and determine our desires in a software EDAC approach. One such set might be:

Use native operators. We have found that if we're performing addition, we want it to look like addition, not like a function call.

Don't end-run the compiler. Avoid putting magic steps between the source code and the compiler (preprocessing), or the compiler and the object code (postprocessing), as this makes debugging difficult.

Perform error detection on every read. Every time a variable is accessed, check if it's healthy. If it has an SEU, correct it; if it is uncorrectable, raise an exception.

Record error-check bits on every write. This is necessary to perform error detection later on.

Use at least SECDED ECC. For similarity with hardware EDAC, the approach should be able to correct any single bit error, and detect any double-bit error.

Minimize latency. The penalty for software EDAC is that it takes processor cycles. It is ideally a small number of cycles.

Minimize exposure. For many ECCs (except triple save, where each variable is copied three times and voting performed), there is a period where an SEU can enter a calculation such that it will go undetected.

Balance memory usage with cycle usage. To some extent, an ECC can be made to use few bits but many processor cycles, or vice/versa. Finding a good balance between the two is important.

The first two items are the most interesting ones, as they focus on expression rather than function. Any programming language can implement the functions described – SECDED ECC, minimal latency – but the expression, as explored earlier, may be cumbersome. It is here that the subject of this paper comes into play.

Any language which offers *operator overloading* – the ability to redefine the meanings of symbols otherwise recognized to the compiler, such as "+" and "=" – opens the opportunity to satisfy the expression-focused desires listed above. Operator overloading is a way to tell the compiler, "When an expression such as `X=A+B+C` is encountered, use these special functions for '=' and '+' instead of the usual one." The special functions then perform the error correction and error-bit creation tasks required for EDAC, much as `ProtIntAdd` and `ProtIntSet` did earlier.

A given programming language will have some set of base types, from which all other data structures are built. When using operator overloading to perform EDAC, the operations on these base types themselves may be overloaded, or new types may be created which are "EDAC-protected versions" of the base types. The second option is generally favorable, since it leaves the option open for the programmer to easily revert to (unprotected) base types for speed or other reasons; it also simplifies the process of allocating memory for the error-check bits, as the new protected types can simply include these bits. Either way, each operation that the compiler implements for the given base type is re-implemented basically as shown here (Figures 3 and 4):

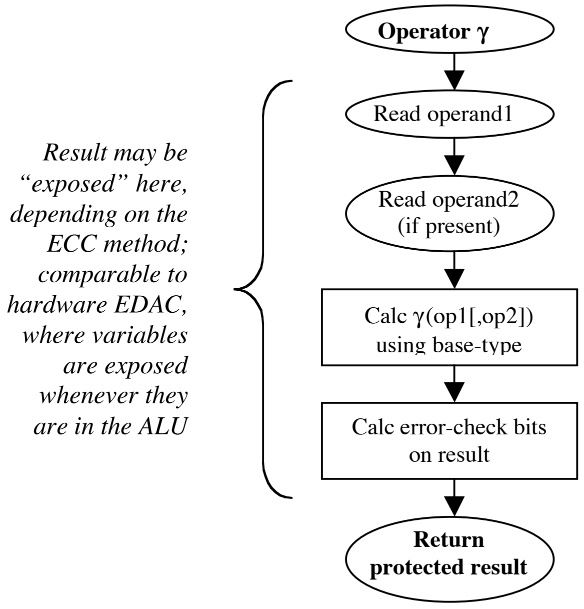
Operator γ
Read operand1
Read operand2 (if present)
Calc γ(op1[,op2]) using base-type
Calc error-check bits on result
Return protected result
Result may be "exposed" here, depending on the ECC method; comparable to hardware EDAC, where variables are exposed whenever they are in the ALU

Figure 3. Framework for EDAC-Enhanced Operators

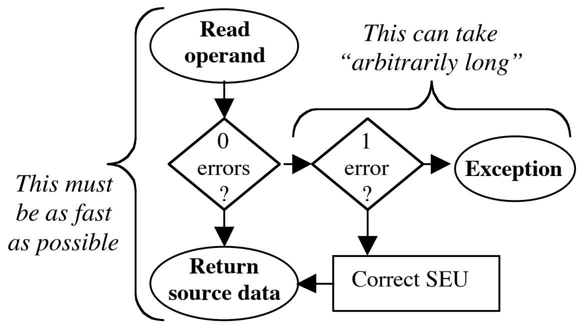

Figure 4. EDAC Read Function

ECC Choice

The choice of Error Correcting Code is what defines the system's ability to minimize latency, minimize exposure, and balance memory usage with cycle usage. Three options are shown in Table 1 below. Assembly code demonstrating the implementation of the read function for a 32-bit operand in a PowerPC architecture is shown as an example. All of the routines place the value pointed to by register r7, into the register r1.

Table 1: Error Correcting Code Options

Method	Read coding	# instr.	Instruction ratio with non-EDAC	Memory ratio with non-EDAC	Comments
Non-EDAC	`lwz r1,0(r7)`	1	1X	1X	Assumes EDAC handled by hardware; cannot do EDAC on internal memory.

Triple Save	`lwz r1,0(r7)`
	`lwz r2,1(r7)`
	`lwz r3,2(r7)`
	`cmpd r1,r2`
	`bcl 4,2,UpsetHandler`
	`cmpd r1,r3`
	`bcl 4,2,UpsetHandler`
		7	7X	3X	Variables are never exposed. Can be fooled by some dual-errors, but otherwise is a very strong ECC.

Triple XOR	`lwz r1,0(r7)`
	`rotlwi r2,r1,1`
	`xor r3,r1,r2`
	`rotlwi r2,r1,2`
	`xor r3,r3,r2`
	`lwz r2,1(r7)`
	`xor. r3,r3,r2`
	`bcl 4,2,UpsetHandler`
		8	8X	2X	Achieves SECDED. Good overall cycle/memory balance.

Hamming	*(Too long to show)*	>>50	>>50X	1.25X	For comparison only; not a sensible software ECC.

The triple save method stores three copies of every variable, so that a single bit error in any one of the copies is recoverable from the other two copies. In fact, any number of errors in any one copy of the variable are recoverable, protecting against other types of failures than single-event upsets. Triple-save actually does not satisfy the desire to detect all dual-bit errors; however, it can always come up with a "best-guess" result based on voting among the three copies of the variable. Its strongest feature is that, assuming an operation like addition is encoded as

```
add     sum_copy1,addend1_copy1,addend2_copy1
add     sum_copy2,addend1_copy2,addend2_copy2
add     sum_copy3,addend1_copy3,addend2_copy3
```

then there is never a moment when an SEU can occur that will go undetected in the result. That is, there is no variable exposure at any time while using EDAC-protected operations.

The triple XOR method takes the source data, rotates it by one bit, rotates it also by two bits, and exclusive-ORs all three values together to create the error-check bits. A single-bit error in the source data word shows up as three sequential bits different in the error-check word. A single-bit error in the error-check word shows up as that single bit being in error. Dual-bit errors show up as 2, 4, 5, or 6 discrepancies in the error-check word. This approach is more memory efficient than triple save and fully adheres to the desire for SECDED performance, but it takes slightly longer per read and does leave the result of an operation exposed to an SEU if it occurs between the read of the operands and the creation of the result's error-check bits. Again note that this is the same as hardware EDAC, where an SEU in a processor register will cause a result to be calculated incorrectly but undetectably so.

One important note is that since operators can be overloaded corresponding to any variable type, the error coding can be customized to be optimal for each word size or variable usage. An EDAC-protected floating-point variable, for example, might employ an ECC that captures errors in the exponent better than in the mantissa. Triple save may be more useful for byte-wide variables, while triple XOR is used for 32-bit variables.

The above list is by no means canonical. Many other ECCs can be created, optimized around processor instruction sets, or variable usage, or expected error pattern, or any other situation. Nor do ECCs need to adhere to the "source data word plus error check word(s)" pattern assumed in this table. The data dealt with by the read function does not necessarily need to resemble the source data in any way, as long as operations on the data (as redefined by operator overloading) produce the correct results. One particularly promising avenue is an ECC which survives arithmetic operations without exposing the source data at all; that is, where `X=A+B` can be performed as a single true addition, as opposed to a source data recovery - addition - result error check creation process.

CODE EXAMPLE

Following is a sample of software EDAC using operator overloading. C++ is the example language, as it is the most commonly used and available language that supports operator overloading. However, none of the specific coding techniques shown here – the use of class templates, the set of base types – is canonical or inherently necessary to perform software EDAC. Many of the specific coding choices shown here are for ease of demonstration.

While C++ itself defines twelve basic data types – bool, char, unsigned char, int, unsigned int, short, unsigned short, long, unsigned long, float, double, and long double – we are better off being more specific in our typing. This is the first of a series of opportunities to make protected variables behave largely like their unprotected counterparts, but with more safeguards. So the nine quasi-basic types we will utilize are:

`bool`	takes values TRUE or FALSE
`int8`	8-bit signed integer
`uint8`	8-bit unsigned integer
`int16`	16-bit signed integer
`uint16`	16-bit unsigned integer
`int32`	32-bit signed integer
`uint32`	32-bit unsigned integer
`float`	32-bit floating-point
`double`	64-bit floating-point

Note that pointers to each of these types; as well as pointers to user-defined types, particularly structures or classes; also exist. They will be dealt with separately below, but for purposes of the present discussion, we can assume that they behave generally like 32-bit unsigned integers.

From these base types, we will create new types – EDAC-protected versions of each of the nine types listed above. They will be implemented as object classes, as that is an excellent way of capturing the idea of software EDAC: variables that act in a certain way, but with the internals hidden from the user. Each EDAC-protected class will have members that contain the source data and error checking bits, member functions for performing read and write, and finally, member functions that perform all of the operations that the base type supports. A variable declared as one of these classes will behave just like the corresponding base type, but behind the scenes, error detection and correction will be occurring.

For purposes of this example, triple XOR ECC will be used, primarily because it bears the greatest functional resemblance – full SECDED, some variable exposure – to hardware EDAC. Given that, it quickly becomes evident that many of the operators to be overloaded look identical for all of these variable types – addition, for example, looks like addition whether the variable is an 8-bit integer or a 64-bit double. For any given EDAC-protected type `ptype`, the general appearance of an operator becomes

```
ptype operatorγ(ptype val) {
    ptype result;
    result = this->read() γ val2.read();
    result.errmake();
    return(result);
}
```

Since this pattern applies to so many different types, the C++ tool of class templates is very useful. The specific types will be derived from this template. Each type can override the standard functions implemented by the template. Note that in the following code examples, the assumption is made that triple XOR is used for all types, so the source-data and error-check members can be part of the template.

One of the advantages of wrapping these behavior into object classes, however, is that any of this can be changed – different ECCs for different types, different behaviors for the same operators under different types – and it all remains completely transparent to the programmer using these EDAC-protected classes. As long as the variables continue to behave like their base-type counterparts, what goes on within the classes related to EDAC is completely adjustable.

The fundamental EDAC-protected class template thus looks like this:

```
template <class Type>
class Prot {
public:
  Prot<Type> () {}
  Prot<Type> (Type val) {src=val;  errmake();}
  ~Prot<Type> () {}

  Prot<Type> operator+();
  Prot<Type> operator+(Prot<Type> &val);
  Prot<Type> operator-();
  Prot<Type> operator-(Prot<Type> &val);
  Prot<Type> operator*(Prot<Type> &val);
  Prot<Type> operator/(Prot<Type> &val);
  Prot<Type> operator%(Prot<Type> &val);

  Prot<Type> operator~();
  Prot<Type> operator&(Prot<Type> &val);
  Prot<Type> operator|(Prot<Type> &val);
  Prot<Type> operator^(Prot<Type> &val);
  Prot<Type> operator>>(Prot<int32> &val);
  Prot<Type> operator<<(Prot<int32> &val);

  Prot<Type> &operator=(Type val);
  Prot<Type> &operator=(Prot<Type> &val);
  Prot<Type> &operator+=(Prot<Type> &val);
  Prot<Type> &operator-=(Prot<Type> &val);
  Prot<Type> &operator*=(Prot<Type> &val);
  Prot<Type> &operator/=(Prot<Type> &val);
  Prot<Type> &operator%=(Prot<Type> &val);
  Prot<Type> &operator&=(Prot<Type> &val);
  Prot<Type> &operator^=(Prot<Type> &val);
  Prot<Type> &operator|=(Prot<Type> &val);
  Prot<Type> &operator>>=(Prot<Type> &val);
  Prot<Type> &operator<<=(Prot<Type> &val);

  Prot<Type> operator++();
  Prot<Type> operator++(int val);
  Prot<Type> operator--();
  Prot<Type> operator--(int val);

  ProtBool operator>(Prot<Type> &val);
  ProtBool operator<(Prot<Type> &val);
  ProtBool operator!=(Prot<Type> &val);
  ProtBool operator==(Prot<Type> &val);
  ProtBool operator>=(Prot<Type> &val);
  ProtBool operator<=(Prot<Type> &val);

  ProtPtr<Prot<Type>> operator&();

  Type to_unprotected()
    {return(this->read());}

protected:
  Type read();
  void errmake();
  void UpsetHandler();

  Type  src;
  Type  err;
};
```

Several items should be noticed in this template:

- There are two ways to go from an unprotected to a protected variable: using a constructor (`Prot<Type> (Type val)`) or using the assignment operator (`operator=(Type val)`). Only variables of the given base type can be converted to protected variables this way. Upcasting is a significant issue discussed later.
- Bit-shift operators are assumed to take an `int32` type. This is a safe way to start, though as always, more operator overloading can increase efficiency by avoiding unnecessary upcasting to `int32`.
- Most operands are passed by reference to these member functions. *In general, when using EDAC-protected variables, passing by reference should be avoided.* This is contrary to usual good practice in C++, where passing by reference is more elegant than passing pointers[2]. However, passing by reference causes the compiler to actually send a pointer, unprotected, to the receiving function. Passing protected pointers (discussed later), as in traditional C, is a better approach to letting the function adjust a variable in its parameter list. Passing by reference is necessary for these class member functions, uniquely, so that any errors detected during the read of the right-hand value (`val`) can be corrected in the actual variable instead of a copy.
- `ProtBool` is not `Prot<bool>`. This is because the `bool` type is, by custom, `FALSE=0` and `TRUE=1`, which is not a good foundation for strong error-checking. `ProtBool` is therefore defined as a separate class independent of the actual `bool` base class.

Under this template, most operators are implemented in the format shown earlier:

```
template <class Type> Prot<Type>
Prot<Type>::operator+() {
  this->read();  return(*this);
}

template <class Type> Prot<Type>
Prot<Type>::operator+(Prot<Type> val) {
  Prot<Type>   ret;
  ret.src=read()+val.read();  ret.errmake();
  return(ret);
}

template <class Type> Prot<Type>
Prot<Type>::operator~() {
  Prot<Type>   ret;
  ret.src=~read();  ret.errmake();
  return(ret);
}

template <class Type> Prot<Type>
Prot<Type>::operator&(Prot<Type> val) {
  Prot<Type>   ret;
  ret.src=read()&val.read();  ret.errmake();
  return(ret);
}

template <class Type> Prot<Type>
Prot<Type>::operator>>(Prot<int32> val) {
  Prot<Type>   ret;
  ret.src=read()>>val.read();  ret.errmake();
  return(ret);
}

template <class Type> Prot<Type>
&Prot<Type>::operator=(Type val) {
  src=val;  errmake();
  return(*this);
}

template <class Type> Prot<Type>
&Prot<Type>::operator=(Prot<Type> val) {
  src=val.read();  errmake();
  // Can check for this->err == val.err
  return(*this);
}

template <class Type> Prot<Type>
&Prot<Type>::operator+=(Prot<Type> val) {
  src=read()+val.read();  errmake();
  return(*this);
}

template <class Type> Prot<Type>
Prot<Type>::operator++() {
  src=read()+1;  errmake();
  return(*this);
}

template <class Type> Prot<Type>
Prot<Type>::operator++(int val) {
  src=read()+1;  errmake();
  return(*this);
}

template <class Type> ProtBool
Prot<Type>::operator<(Prot<Type> val) {
  if (read()<val.read())
    return(PROTTRUE);
  else return(PROTFALSE);
}

template <class Type> ProtBool
Prot<Type>::operator!=(Prot<Type> val) {
  if (read()!=val.read())
    return(PROTTRUE);
  else return(PROTFALSE);
}

template <class Type> ProtBool
Prot<Type>::operator==(Prot<Type> val) {
  if (read()==val.read())
    return(PROTTRUE);
  else return(PROTFALSE);
}

template <class Type> ProtPtr<Prot<Type>>
Prot<Type>::operator&() {
  ProtPtr<Prot<Type>> ret;
  ret.src = this;  ret.errmake();
  return(ret);
}
```

More items to notice:

- `ProtBool` has two constants associated with it, unsurprisingly, `PROTTRUE` and `PROTFALSE`.
- It will generally be useful to inline these functions rather than incurring the overhead of function call/return. This will significantly increase program executable size, but also significantly increase program speed, and in fact removes any latent problems with passing-by-reference into `Prot<Type>` member functions, because an inlined function would not actually cause a reference pointer to be created. Once debugging is completed, the `inline` keyword may be added to the member function declarations, or the definitions added to the class declaration. The compiler should then in most circumstances be trusted to make the right decision regarding inlining.
- The use of the template allows the compiler to complain when the programmer attempts to perform insensible operations. For example, if the programmer attempts to perform a bit-negation on a floating-point type, the template will expand `operator~`, then find in the `ret.src=~read();` statement that this operation is invalid.
- Conversely, valid combinations of different types – the addition of an 8-bit integer to a 32-bit integer, for example – are not supported yet. These will be handled in the specific classes that derive from this base template.
- This template is not necessarily limited to base types. A user-defined class, for example, can be protected – `Prot<UserType>` – as long as `read()` and `errmake()` are defined for that type. Or alternately, the `read()` and `errmake()` in the template can be completely generic, and more efficient, specific versions of these functions can be implemented for the fundamental (integer, floating point, pointer) protected types.

For now, however, we will take the opposite tack, implementing the `read()` and `errmake()` functions under the template which will function well for all integer types.

```
template <class Type> Type
Prot<Type>::read() {
  if ((src ^ (src<<1) ^ (src<<2) ^
    ((src>>((8*sizeof(src))-1))&1) ^
    ((src>>((8*sizeof(src))-2))&3)) ^
    err) UpsetHandler();
  return(src);
}

template <class Type> void
Prot<Type>::errmake() {
  err = src ^ (src<<1) ^ (src<<2) ^
    ((src>>((8*sizeof(src))-1))&1) ^
    ((src>>((8*sizeof(src))-2))&3);
}
```

The gyrations with bitshifts are because C does not support a wraparound rotation operation. As indicated in Table 1: Error Correcting Code Options, the actual `read()` function would ultimately be implemented as a tight assembly routine optimized for the variable size and microprocessor instruction set. These C-language functions are shown as an example of how to start in the creation of an EDAC-protected type library, less time-efficient but more portable and easier to debug.

More important is the convention demonstrated in `read()`. A `read()` function should be optimized for the non-error case, and bail out to a much more extensive function (`UpsetHandler()`) if any errors are found. `UpsetHandler()` takes no arguments and returns no value. It will change `this->src` if it can correct the error, and will raise any necessary exceptions if the failure is unrecoverable. `read()` always returns `this->src`. The net result is that `read()` can go through the error check, to a conditional branch with no stack maintenance, to a return of `this->src`. Note that in the process of making `read()` fast, `errmake()` will naturally be made fast as well, as it is usually performing the same basic calculation as `read()`.

At this point, we could actually stop – once we have written the rest of the operators in the same form as the ones shown, and written an `UpsetHandler()` routine – and start using the template directly. We could declare variables as `Prot<int>`, or `Prot<long>`, or even `Prot<void *>`, and they would consist of their source data plus error checking bits. (We would not have protected floating-point types, because `read()` and `errmake()` above use bitwise operations that are only valid for integers. Custom `read()` and `errmake()` functions for floating-point types are discussed in a later section, though their functionality is the same as the ones shown.) Every time an operation was performed on our protected variables, all the correct error-checking would be done, and every time a protected variable was assigned, it would calculate its error-checking bits. However, this is not quite enough to make a program behave with protected variables the same way it would with base-type variables, because a host of convenient automatic conversions among types would not be available.

Casting

If we never had to worry about mixing two different kinds of EDAC-protected variable, and we never wanted to use constants in an expression, we could simply write special `read()` and `errmake()` functions for our floating-point variable types and be done. However, no sensible program is so strict in its typing that an 8-bit value cannot be added to a 32-bit value, or a constant must be explicitly given a bit width in order to behave correctly in an expression. Therefore, instead of declaring variables based directly on the `Prot<Type>` template, we will create a set of type-specific classes derived from this template; and for each of these type-specific classes, we will create constructors that allow type promotion to occur.

Here is another situation where, while we desire operations with EDAC-protected variables to behave identically to operations with their unprotected counterparts, we can take the opportunity when it presents itself to guard ourselves a little more than the C standard demands. In this case, we will only allow *lossless promotions*, that is, only type conversions where the destination type range includes the entire source type range will be allowed. (Promotions from integer to floating-point variables will always be allowed; even if this results in loss of precision, it will not result in loss of accuracy.) Any other type conversions must be done explicitly, and preferably with checks to make sure that (for example) a `uint8` with a value of 200 does not get converted into an `int8` with a value of –56. Using our nine chosen base types, we end up with the following hierarchy:

```
class pi8 : public Prot<int8> {
};

class pui8 : public Prot<uint8> {
};

class pi16 : public Prot<int16> {
public:
  pi16(int8 val) {src=static_cast<int16>val;  errmake();}
  pi16(pi8 val) {src=static_cast<int16>val.to_unprotected();  errmake();}
  pi16(uint8 val) {src=static_cast<uint16>val;  ermake();}
  pi16(pui8 val) {src=static_cast<int16>val.to_unprotected();  errmake();}
};

class pui16 : public Prot<uint16> {
```

```
public:
  pui16(uint8 val) {src=static_cast<uint16>val;  errmake();}
  pui16(pui8 val) {src=static_cast<uint16>val.to_unprotected();  errmake();}
};

class pi32 : public Prot<int32> {
public:
  pi32(int8 val) {src=static_cast<int32>val;  errmake();}
  pi32(pi8 val) {src=static_cast<int32>val.to_unprotected();  errmake();}
  pi32(uint8 val) {src=static_cast<int32>val;  errmake();}
  pi32(pui8 val) {src=static_cast<int32>val.to_unprotected();  errmake();}
  pi32(int16 val) {src=static_cast<int32>val;  errmake();}
  pi32(pi16 val) {src=static_cast<int32>val.to_unprotected();  errmake();}
  pi32(uint16 val) {src=static_cast<int32>val;  errmake();}
  pi32(pui16 val) {src=static_cast<int32>val.to_unprotected();  errmake();}
};

class pui32 : public Prot<uint16> {
public:
  pui32(uint8 val) {src=static_cast<int32>val;  errmake();}
  pui32(pui8 val) {src=static_cast<int32>val.to_unprotected();  errmake();}
  pui32(uint16 val) {src=static_cast<int32>val;  errmake();}
  pui32(pui16 val) {src=static_cast<int32>val.to_unprotected();  errmake();}
};

class pfloat : public Prot<float> {
public:
  pfloat(int8 val) {src=static_cast<float>val;  errmake();}
  pfloat(pi8 val) {src=static_cast<float>val.to_unprotected();  errmake();}
  pfloat(uint8 val) {src=static_cast<float>val;  errmake();}
  pfloat(pui8 val) {src=static_cast<float>val.to_unprotected();  errmake();}
  pfloat(int16 val) {src=static_cast<float>val;  errmake();}
  pfloat(pi16 val) {src=static_cast<float>val.to_unprotected();  errmake();}
  pfloat(uint16 val) {src=static_cast<float>val;  errmake();}
  pfloat(pui16 val) {src=static_cast<float>val.to_unprotected();  errmake();}
  pfloat(int32 val) {src=static_cast<float>val;  errmake();}
  pfloat(pi32 val) {src=static_cast<float>val.to_unprotected();  errmake();}
  pfloat(uint32 val) {src=static_cast<float>val;  errmake();}
  pfloat(pui32 val) {src=static_cast<float>val.to_unprotected();  errmake();}
};

class pdouble : public Prot<double> {
public:
  pdouble(int8 val) {src=static_cast<double>val;  errmake();}
  pdouble(pi8 val) {src=static_cast<double>val.to_unprotected();  errmake();}
  pdouble(uint8 val) {src=static_cast<double>val;  errmake();}
  pdouble(pui8 val) {src=static_cast<double>val.to_unprotected();  errmake();}
  pdouble(int16 val) {src=static_cast<double>val;  errmake();}
  pdouble(pi16 val) {src=static_cast<double>val.to_unprotected();  errmake();}
  pdouble(uint16 val) {src=static_cast<double>val;  errmake();}
  pdouble(pui16 val) {src=static_cast<double>val.to_unprotected();  errmake();}
  pdouble(int32 val) {src=static_cast<double>val;  errmake();}
  pdouble(pi32 val) {src=static_cast<double>val.to_unprotected();  errmake();}
  pdouble(uint32 val) {src=static_cast<double>val;  errmake();}
  pdouble(pui32 val) {src=static_cast<double>val.to_unprotected();  errmake();}
  pdouble(float val) {src=static_cast<double>val;  errmake();}
  pdouble(pfloat val) {src=static_cast<double>val.to_unprotected();  errmake();}
};
```

A C++ compiler will attempt to make one type cast per operand in order to make an expression work. Therefore, if an expression is full of 32-bit EDAC-protected variables (`pi32`) and one 8-bit EDAC-protected variable (`pi8`), the `pi8` will be converted to a temporary `pi32` via the `pi32(pi8)` constructor and the expression will work. This is exactly what we want. However, if a signed 16-bit variable (`pi16`) tries to get assigned to a smaller (`pi8`) or unsigned (`pui16`) variable, the compiler will not allow it. This is again what we want, because we never want to perform such a potentially lossy conversion accidentally or unknowingly.

Special Cases: bools, floats, and pointers

Earlier explanations have pointed out exceptions for Boolean types, floating point error check calculations, and pointers-to-anything. These exceptions are now revisited for resolution.

Creating `read()` and `errmake()` functions for floating-point types is relatively straightforward; treat the source data as one or two 32-bit integers and perform the same check as before:

```
float pfloat::read() {
  uint32 srcint;
  srcint=*(static_cast<uint32*>&src);
  if ((srcint ^ (srcint<<1) ^ (srcint<<2) ^
    (srcint>>((8*sizeof(srcint))-1)) ^
    (srcint>>((8*sizeof(srcint))-2))) ^
    *(static_cast<uint32*>&err))
    UpsetHandler();
  return(src);
}

void pfloat::errmake() {
  uint32 srcint;
  srcint=*(static_cast<uint32*>&src);
  *(static_cast<uint32*>&err)=srcint ^
    (srcint<<1) ^ (srcint<<2) ^
    (srcint>>((8*sizeof(srcint))-1)) ^
    (srcint>>((8*sizeof(srcint))-2));
}

double pdouble::read() {
  uint32 srcint,*srcptr, errint,*errptr;
  srcptr=static_cast<uint32*>&src;
  errptr=static_cast<uint32*>&err;
  srcint=*srcptr;
  errint=*errptr;
  if ((srcint ^ (srcint<<1) ^ (srcint<<2) ^
    (srcint>>((8*sizeof(srcint))-1)) ^
    (srcint>>((8*sizeof(srcint))-2))) ^
    errint) UpsetHandler();
  srcint=*(srcptr+1);
  errint=*(errptr+1);
  if ((srcint ^ (srcint<<1) ^ (srcint<<2) ^
    (srcint>>((8*sizeof(srcint))-1)) ^
    (srcint>>((8*sizeof(srcint))-2))) ^
    errint) UpsetHandler();
  return(src);
}

void pdouble::errmake() {
  uint32 srcint,*srcptr, errint,*errptr;
  srcptr=static_cast<uint32*>&src;
  errptr=static_cast<uint32*>&err;
  srcint=*srcptr;
  *errptr = srcint ^
    (srcint<<1) ^ (srcint<<2) ^
    (srcint>>((8*sizeof(srcint))-1)) ^
    (srcint>>((8*sizeof(srcint))-2));
    errint) UpsetHandler();
  srcint=*(srcptr+1);
  *(errptr+1) = srcint ^
    (srcint<<1) ^ (srcint<<2) ^
    (srcint>>((8*sizeof(srcint))-1)) ^
    (srcint>>((8*sizeof(srcint))-2));
}
```

These functions are, in a word, ugly. They are offered for completeness' sake, not because these are the correct ways to implement floating-point error-checking in an operational system, but more because they bring out some connected points:

- When starting out developing a set of protected-variable classes, results matter, not the route to get there. Start off with a straightforward, even pedantic, set of overloaded functions in order to implement the functionality of EDAC protection. Then, when the functionality itself is proven, start replacing these underlying functions with more efficient versions – assembly-coded, type-specific, dense and less plain-written than the original versions, but implementing the same result.
- At the beginning of development, it is tempting to code elegantly, not rewriting the same function over and over for subtle differences in base types. The use of the `Prot<Type>` template is this sort of effort; the creation of specific `read()` and `errmake()` functions for `pfloat` and `pdouble` seem to violate this elegance. However, the opposite approach is usually ultimately called for: the goal is a library included as a black box into SEU-sensitive programming projects, where the elegance of the source code is much less relevant than the speed of the resulting object code. With this priority, it is likely that nearly every function in the library will be hand-coded, optimized for speed, in its final form. The only elegance required of the source code, then, is that it smoothly transition from readable / debuggable to optimized / fast without a major overhaul. The ability to override functions from the (source code-efficient) template with (speed-efficient) type-specific routines achieves this.
- In the specific case of floating-point variables, it is often the case that a single set of data registers are available to the microprocessor's ALU, and both bit-manipulation and floating-point instructions are available to work on these registers. All of the complications of casting then disappear when the `read()` and `errmake()` functions are implemented in assembly. Alternately, if the chosen processor has a distinct Floating-Point Unit with its own register set, then a wholly different ECC approach can be employed that uses only FPU instructions. Again, the user of the library never needs to know the details of how EDAC protection is provided.

`ProtBool` is an even more particular case. It does not derive from `Prot<Type>` at all, because as noted earlier, two `bool` types do not form a good source-data / error-check combination as it does for the other types.

That said, the `ProtBool` class does still look similar to `Prot<Type>`.

```
class ProtBool;

typedef int8 boolinternal;

const boolinternal BASEPROTTRUE=-1;
const boolinternal BASEPROTFALSE=0;

const ProtBool PROTTRUE(BASEPROTTRUE);
const ProtBool PROTFALSE(BASEPROTFALSE);

class ProtBool {
public:
  ProtBool() {}
  ProtBool(boolinternal val) {src=val;}
  ProtBool(bool val)
    {if (val) src=BASEPROTTRUE;
    else src=BASEPROTFALSE;}
  ProtBool(ProtBool val) {src=val.read();}

  ProtBool &operator=(bool val)
    {if (val) src=BASEPROTTRUE;
    else src=BASEPROTFALSE; return(*this);}

  ProtBool &operator=(ProtBool val)
    {src=val.read();  return(*this);}

  ProtBool operator==(ProtBool val)
    {return(val.read()==read());}

  ProtBool operator!=(ProtBool val)
    {return(val.read()!=read());}

  ProtBool operator!()
    {return(~read());}

  ProtBool operator&&(ProtBool val)
    {return(val.read()&read());}

  ProtBool operator||(ProtBool val)
    {return(val.read()|read());}

  operator int()
    {return(static_cast<int>src);}

private:
  boolinternal read()
    {if (src==BASEPROTTRUE)
    return(BASEPROTTRUE);
    else if (src==BASEPROTFALSE)
    return(BASEPROTFALSE);
    else UpsetHandler();}
  void UpsetHandler();

  boolinternal src;
};
```

Note that the AND and OR operators turn into bytewise operations, creating 8-fold redundancy to make sure the result is calculated correctly with no variable exposure. An automatic conversion to `ProtBool` is performed at the return line, using the constructor `ProtBool(boolinternal val)`. That is,

```
    {return(val.read()&read());}
```

is equivalent to

```
    {ProtBool temp;
    temp.src = val.read() & read();
    return(temp);}
```

Also in the topic of automatic conversions is the `operator int()` line. This is a dangerous automatic conversion to offer as it allows `ProtBool` types to be arithmetically combined with integer types, with unexpected results, especially if the C convention of `TRUE=1` is expected. However, this allows `ProtBool` types to be evaluated inside `if()` expressions, which is ultimately necessary for a useful `ProtBool` type. A note in this regard: with the assignments of `PROTFALSE=0` and `PROTTRUE=-1`, a single bitflip can change a false to a true, but not vice/versa. It may be that a safer approach is to "bias toward false", that is to make `PROTFALSE=-1` and `PROTTRUE=0` and change the operators to match; this guards against situations like

```
if (fire_pyrotechnics) irrevocable_action();
```

causing bad operational behavior due to a single bit flip.

As with all of the earlier examples, this is only one possible implementation of `ProtBool`; and it is neither efficient nor elegant, only demonstrative.

The final special case is pointers. Pointers can point to anything – basic types, EDAC-protected types, user-defined types, even the nondescript pointer `void *` – so we cannot enumerate all possible pointers. We could create a pointer type that essentially reduced to `void *`, but then we would lose all of the type-checking capabilities that the compiler offers during assignment and dereferencing.

For pointers, we want to use a template, creating a different kind of EDAC-protected pointer type for each different object the program points to. The existing `Prot<Type>` template would work to some extent, but would fail for arithmetic operations, since pointer types cannot be universally arithmetically combined with each other. Pointers can be differenced, but not summed; and the difference between two pointers is different than subtracting an integer from a pointer. For all of these reasons, we find we must create a `ProtPtr<Type>` template.

```
template <class Type>
class ProtPtr {
public:
 ProtPtr<Type> () {}
 ProtPtr<Type> (Type *val){src=val;errmake();}
 ~Prot<Type> () {}
ProtPtr<Type> operator+(Prot<int32> val);
 ProtPtr<Type> operator-(Prot<int32> val);
 ProtPtr<Type> operator-(ProtPtr<Type> val);

 ProtPtr<Type> operator~();
 ProtPtr<Type> operator&(Prot<uint32> val);
 ProtPtr<Type> operator|(Prot<uint32> val);
 ProtPtr<Type> operator^(Prot<uint32> val);
 ProtPtr<Type> operator>>(Prot<int32> val);
 ProtPtr<Type> operator<<(Prot<int32> val);

 void operator=(Type *val);
 void operator=(ProtPtr<Type> val);
 void operator+=(Prot<int32> val);
 void operator-=(Prot<int32> val);
 void operator-=(ProtPtr<Type> val);
 void operator&=(Prot<uint32> val);
 void operator^=(Prot<uint32> val);
 void operator|=(Prot<uint32> val);
 void operator>>=(Prot<int32> val);
 void operator<<=(Prot<int32> val);

 ProtPtr<Type> operator++();
 ProtPtr<Type> operator++(int val);
 ProtPtr<Type> operator--();
 ProtPtr<Type> operator--(int val);

 ProtBool operator>(ProtPtr<Type> val);
 ProtBool operator<(ProtPtr<Type> val);
 ProtBool operator!=(ProtPtr<Type> val);
 ProtBool operator==(ProtPtr<Type> val);
 ProtBool operator>=(ProtPtr<Type> val);
 ProtBool operator<=(ProtPtr<Type> val);

 ProtPtr<ProtPtr<Type> > operator&();
 Type operator*();

 Type *to_unprotected() {return(read());}

private:
 Type *read();
 void errmake();
 void UpsetHandler();

 Type *src;
 uint32 err;
};
```

The implementation of these functions is entirely analogous to all those shown already. Unlike the `Prot<Type>` template, though, which is really only a base class for types like `pui16` and `pfloat`, `ProtPtr<Type>` will be used in its template form in the software itself. For example,

```
void foo() {
  pi32 variable;
  ProtPtr<pi32> variable_ptr;
  struct userstruct mystrucutre;
  ProtPtr<struct userstruct> mystructure_ptr;

  variable_ptr = &variable;
  mystructure_ptr = &mystructure;
}
```

would be a typical usage of EDAC-protected pointers.

There is an important option that presents itself when working with the ProtPtr<Type> class. A similar class could be created which is a pointer that protects not only itself but the value that it points to:

```
template<Type,ErrorCoder=default_ecc<Type> >
class edac_pointer {
public:
  const Type &operator*() const {
    return edac.read(data);
  } // read only version
  edac_reference<Type> operator*() {
    return edac_reference<Type>(*this);
  } // writable version
protected:
  friend class edac_reference<Type>;
  Type &data;
  ErrorCoder edac;
};

template<Type>
class edac_reference {
public:
  edac_reference &operator=(const Type &val) {
    pointer.edac.write(val);
    return(*this);}
  operator Type () const {return(*pointer);}
protected:
  friend class edac_pointer<Type>;
  // hide the constructor, so that these only
  // get created by deref'ing an edac_pointer
  edac_reference<Type>(edac_pointer<Type>
    &_pointer) : pointer(_pointer) { }
  edac_pointer<Type> &pointer;
};
```

At that point, a wrapper function can be created that makes the operation of this pointer look like operating on a variable of type `<Type>`:

```
template <class Type>
class edac_variable {
public:
  edac_variable() : pointer(data) { }
  edac_variable(const Type &val) :
    pointer(data) {(*pointer) = value;}
  operator Type &() {return (*pointer);}
  edac_variable &operator=(Type &value) {
    (*pointer) = value;}

protected:
  edac_pointer<Type> pointer;

private:
  Type data;
};
```

As with the series of `Prot<Type>`-derived variables, an `edac_variable<Type>` would behave just like a variable of type `Type`, with the compiler taking care of all type promotions and conversions, and the particular `ErrorCoder` taking care of the EDAC functions. The tradeoff here is executable complexity versus source code complexity. Using an `edac_variable<Type>` requires less complexity within the EDAC-protected classes themselves, since they are essentially pointers to base types that the compiler itself understands, and the long series of operators (multiplication, equality tests, and so forth) do not need to be overloaded. This is in exchange for more referencing / dereferencing operations in the code executable.

The UpsetHandler Function

No examples have been given thus far for an `UpsetHandler()` function. This is essentially because, while the purpose of this function is easily described (see Figure 4), it tends for any given base type to be an extensive amount of code to take care of all the various possibilities of errors that may have occurred. As noted in the beginning of this discussion, once a `read()` function has determined that at least one error exists in an EDAC-protected variable, it "bails out" to `UpsetHandler()`, which can take – relatively speaking – a large amount of time correcting or otherwise dealing with the error(s). `UpsetHandler()` should be as thoroughly careful as any function in the software suite, because it must at some point deal with the most dangerous situation short of a microprocessor hardware failure: the knowledge that a data value is erroneous, without the ability to correct it.

OTHER ISSUES

It has already been explored that EDAC-protected versions of different base types can use a variety of ECCs and other encodings optimized for the particular microprocessor and the typical usage of that type of variable. A Boolean type, for example, is not used for calculations, but for decisions; and thus it is handled differently than, for example, a double-precision floating-point variable, which is likely being used for vehicle control or other similar math-intensive operation. Distinctions need not be made only on this basis, however. For example, variables that are used only for recording telemetry can suffer bit errors much more innocuously than variables that are used for important operations like propulsion control. Different levels of protection can then be offered – relatively efficient triple XOR for most variables, then highly robust and exposure-free triple save for critical ones – and the software designer can judge the cost/benefit for each variable used in the software suite. As long as casting operations can convert among the types sensibly, this approach retains the ease-of-use of EDAC-protected variable types.

While casting is the most straightforward method of allowing sensible combinations of different variable types, an even more time-efficient option is to create global-scope operators that take two different types and combine them directly. For example, the code

```
void bar() {
  pi32 bigvar=5000;
  pi8 littlevar=10;
  pi32 sum;

  sum = bigvar + littlevar;
}
```

will convert `littlevar` to a temporary `pi32`, then perform a `pi32::operator+`. This is perfectly effective, but creates an unnecessary temporary `pi32` variable during the calculation. If instead, functions were declared that took the two types directly and added them, and these functions were `friend`s of the `pi8` and `pi32` types, then we would have

```
pi32 operator+(pi32 val1, pi8 val2) {
  return(val1.read()+val2.read());
}

pi32 operator+(pi8 val1, pi32 val2) {
  return(val1.read()+val2.read());
}
```

A temporary `int32` would theoretically be created by the compiler to perform the addition, but that is much less overhead than creating a new `pi32` with its `errmake()` function. Indeed this "temporary `int32` variable" would likely never actually need to be created, given how ALUs operate (with registers of a fixed width). This use of operators that take two variables of different EDAC-protected types is yet another example of trading source code efficiency for object code efficiency: a great many operators of this type must be created to cover all the combinations of different variable types, but once that coding is done, all the user sees is faster operation producing the same behavior.

An entirely different set of approaches is available to augment the variable-level protection against bit flips[3,4]. They may be described broadly as "algorithm-level EDAC", where an entire calculation – an attitude control system control loop, for example – is performed multiple times, and the results compared *in toto* with each other. With this approach, variable-level protection becomes less necessary, even superfluous for variables which exist only within the overall protected calculation. This approach also takes advantage of the trends in microprocessor capability, where a device can easily perform three calculations using its internal memory in less time than it can perform the same calculation once using external memory filtered through hardware EDAC. Variable-level and algorithm-level

EDAC can easily be combined in the same software suite, where math-intensive speed-sensitive calculations with a few values in and a few values out can use unprotected types and algorithm-level EDAC, while other more mundane operations can use variable-level EDAC.

CONCLUSION

Software EDAC provides several advantages over hardware EDAC:

- It protects microprocessor-internal memory, including on-chip RAM, L1 cache, and to some extent ALU registers.
- It allows any microprocessor or microcontroller or computer board to be EDAC-protected, including COTS products and single-chip systems.
- It takes advantage of the trend in processor technology, toward more memory and functionality pulled into the processor device itself.
- It provides various levels of protection that can be applied selectively according to the need in the software.
- It can be applied to existing, fielded computing hardware, not necessarily limited to aerospace applications.

It has a number of penalties as well, which must be recognized:

- Software EDAC takes many processor cycles to perform the EDAC function. With internal memory accesses 8.5 times faster than external memory accesses, and software EDAC routines taking 7-8 times as many instructions as the same routine without EDAC, this penalty is quickly becoming irrelevant.
- Hardware EDAC requires less than 25% extra memory. Efficient software EDAC requires at least 100% extra memory.
- Software EDAC cannot protect program memory, only data. If the program is read from (often processor-internal) ROM, this is less important, because ROM is enormously less subject to bit flips. Program memory, which does not change, can also be protected by checksums and the retention of multiple copies in ROM. Note that neither hardware nor software EDAC can protect L1 instruction cache.

Software EDAC in a function-oriented language (such as standard C or FORTRAN) is very cumbersome to express. For languages that support operator overloading, however, EDAC-protected variable types may be created that behave the same as variable types native to the language, but perform error detection and correction on every variable access. By creating protected variable types, the processor only spends time protecting memory that is actually in use; the system need not waste time scrubbing unused memory space. By using overloaded operators, code readability is maintained – arithmetic, bitwise, Boolean, assignment, and other functions appear in their natural form – but "behind the scenes", variables are protected against single event upsets. The particular methods employed to protect a given variable type are completely transparent to the user of the variable type, as long as the operations behave as expected.

The particular means of implementing EDAC-protected variable types depends on the programming language, the microprocessor, and the type of error protection desired. The implementation details can become extensive, but as long as all of the resulting variable operations behave as they do for the corresponding base types, software EDAC using operator overloading can be a powerful tool for bringing error detection to devices that would not otherwise support it.

REFERENCES

1. Engineering Directorate / Avionics Systems Division, "Radiation Test Report, Universal Mini Controller." National Aeronautics and Space Administration – Johnson Space Flight Center, December 1999.

2. Scott Meyers, *Effective C++*. Addison-Wesley, Boston Massachusetts, 1998.

3. Chung-Yu Lui, "A study of flight-critical computer system recovery from space radiation-induced error." Proceedings of the 20th Digital Avionics Systems Conference, Daytona Beach Florida, 2001.

4. J.R. Samson Jr, L. DeLa Torre, P. Wiley, T. Stottlar, "A comparison of algorithm-based fault tolerance and traditional redundant self-checking for SEU mitigation." Proceedings of the 20th Digital Avionics Systems Conference, Daytona Beach Florida, 2001.