API implementujące stos błędów.

1. Wprowadzenie

W dotychczasowej pracy z bibliotekami używałem klasycznego sposobu obsługi błędów, który polegał na wykonaniu trzech podstawowych czynności:

- sprawdzeniu czy wywoływana funkcja nie zwróciła błędu,
- utworzenia logu dotyczącego błędnego wywołania,
- jeśli błąd został zwrócony, opuszczałem funkcję z otrzymanym kodem błędu.

przykład:

```
int funkcja_wewnetrzna()
{
int err = 0;
        err = funkcja(...);
        if(err<0){
                PRINT_DEBUG("String z komunikatem o bledzie.\n");
                /* dodatkowe operacje zwiazane np. z czyszczeniem pamięci itp. */
                return err;
        }
        return err;
}


int funkcja zewnetrzna()
{
int err = 0;
        err = funkcja_wewnetrzna(...);
        if(err<0){
                PRINT_DEBUG("Blad w funkcji wewnetrznej.\n");
                /* ... */
                return err;
        }
        return err;
}
```

W wyniku wystąpienia błędu w funkcji wewnętrznej w logach systemowych widoczna jest cała ścieżka informująca o błędach w postaci stringów:

```
"String z komunikatem o bledzie"
"Blad w funkcji wewnetrznej"
```

Informacja ta nie jest niestety przechowywana nigdzie indziej poza plikiem logów. W praktyce związanej z implementacją systemów informatycznych istnieją sytuacje, w których pomocne byłoby przetrzymywanie całego ciągu wydarzeń związanych z wcześniejszym opuszczaniem funkcji pod wpływem wystąpienia błędów.

2. Stos błędów – zasada działania

Idea stosu błędów polega na zbudowaniu struktury danych, która zawierałaby informacje o wszystkich błędach jakie wystąpiły od miejsca, w którym nie powiodło się wykonanie określonej operacji i dalej poprzez wszystkie funkcje, które wywoływały kończącą się błędem procedurę.

Przykładowo jeśli funkcja *f1* zakończy się błędem, wówczas do stosu zostanie dodany element opisujący ten błąd. W efekcie błędu w *f1*, funkcja *f2* także zakończy się niepowodzeniem, co spowoduje również dodanie do stosu elementu, który go opisuje. Idąc tą ścieżką, wędrówka zakończy się finalnie na funkcji main. W niej można przeglądnąć cały stos, a następnie go wyczyścić.

przykład:

```
err_t f1(void)
```

```
{
err_t err = 0;
        /* ... */

        /* Załózmy że ta funkcja wywoluje f2 w ktorej wystapi blad */
        err = f2();
        if(BMD_ERR(err)){
                BMD_SETERR(err, -111);
                return err;
        }
        return err;
}

err_t f2(void)
{
err_t err = 0;
        /* Jakis kod funkcji f2 - bez bledow */
        /* ... */

        /* I tu nagle wystepuje blad ale w funkcji f3*/
        err = f3();
        if(BMD_ERR(err)){     /* Obsluga bledu */
                BMD_SETERR(err, -111);
                return err;
        }
        return err;
}

err_t f3(void)
{
err_t err = 0;
        /* Standardowy kod */
        /* ... */

        /* Jakis blad */
        if(1){  /* obsluga bledu */
                BMD_SETERR(err, -222);
                BMD_SETIERR(err, LP, "LIBBMDDB");
                BMD_SETIERR(err, AD, "To jest blad otwarcia pliku /dev/urandom");
                BMD_SETIERR(err, AS, "Zainstaluj patcha do systemu operacyjnego");
                return err;
        }
}

int main(int argc, char *argv[])
{
int status = 0;
err_t err = 0;
        /*********************************************/
        /* Test stosu bledow                         */
        /*********************************************/
        /* W main obowiązkowo wykonuje 2 operacje    */
        /* 1. zrzucam stos oraz                      */
        /* 2. czyszcze go                            */
        /*********************************************/
        err = f1();
        if(BMD_ERR(err)){
                printf("Wystapily bledy. Sprawdzam ścieżke wywołań:\n");
                BMD_BTERR(err);
                BMD_FREEERR(err);
        }
        BMD_FREEERR(err); /* cos ala CryptEnd */
        return status;
}
```

W wyniku wystąpienia błędu w funkcji wewnętrznej, stos błędów zawiera cała ścieżka informującą o błędach. Podobnie jak w debuggerze, możliwe jest wykonanie operacji typu *backtrace* (*BMD_BTERR*) Po jej wywołaniu wyświetlona zostanie cała zawartość stosu.

Komentarz do kodu

W każdej funkcji deklarujemy typ *err_t* i OBOWIĄZKOWO inicjalizujemy go wartością zero. Typ *err_t* można utożsamiać ze zmienną typu integer, lecz nieco bardziej inteligentną, gdyż może ona posiadać więcej niż jedną

wartość w danej chwili, a dodatkowo można te wartości opisywać metadanymi.
Przypisywanie wartości do zmiennej typu *err_t* następuje poprzez wywołanie funkcji *BMD_SETERR*. Następne przypisywania realizowane poprzez *BMD_SETERR* nie nadpisują starej wartości lecz dodają kolejne do listy budując w ten sposób stos.
Sprawdzenie czy *err_t* zawiera kody błędów odbywa się poprzez wywołanie funkcji *BMD_ERR*. Zwrócenie wartości równej != 0 informuje o fakcie wystąpienia błędu.
W dowolnym miejscu kodu można wyświetlić zawartość stosu za pomocą funkcji *BMD_BTERR*. Czyszczenie stosu umożliwia funkcja *BMD_FREEERR*.

Standardowe kody błędów opisane są zazwyczaj w dokumentacji umożliwiając diagnozę i rozwiązanie problemu. Często jednak kod błędu informuje o tym, że nie powiodła się dana operacja (np. otwarcia pliku). Programista chcąc sprecyzować tenże komunikat do konkretnego scenariusza wystąpienia błędu musi mieć możliwość jego dokładniejszego opisania. Służy do tego funkcja *BMD_SETIERR*, która należy wywoływać bezpośrednio po funkcji *BMD_SETERR*. W zależności od drugiego parametru funkcji, wartość trzeciego argumentu będącego stringiem ma następujące znaczenie:

- LP - *(library prefix)* prefiks biblioteki, w której wystąpił błąd,
- AD - *(additional description)* dodatkowy opis błędu ustawiany, gdy występuje konieczność sprecyzowania sytuacji, w której blad występuje,
- AS - *(addiitional solution)* dodatkowe informacje umożliwiające rozwiązanie powstałego problemu.

3. Stos błędów – opis funkcji

W skład podstawowego API wchodzą 4 funkcje:

1. Sprawdzanie czy wystąpił błąd

Prototyp:
```
int BMD_ERR(err_t err)
```

Argumenty:
`err_t err` - obiekt stosu błędów

Zwracana wartość:
`int` =1 stos zawiera informacje o błędach
=0 stos nie zawiera informacji o błędach

2. Ustawianie kodu błędu w stosie (i dodanie do stosu kolejnego elementu)

Prototyp:
```
int BMD_SETERR(err_t err, long err_code)
```

Argumenty:
`err_t err` - obiekt stosu błędów
`long err_code` - kod błędu

Zwracana wartość:
`int` =0 funkcja zakończyła sie powodzeniem
=1 funkcja zakończyła sie błędem

3. Czyszczenie stosu błędów

Prototyp:
```
int BMD_FREEERR(err_t err)
```

Argumenty:
`err_t err` - obiekt stosu błędów

Zwracana wartość:
`int` =0 funkcja zakończyła sie powodzeniem
=1 funkcja zakończyła sie błędem

4. Drukowanie stosu błędów

Prototyp:
```
int BMD_BTERR(err_t err)
```

Argumenty:
`err_t err` - obiekt stosu błędów

Zwracana wartość:
`int` =0 funkcja zakończyła sie powodzeniem
=1 funkcja zakończyła sie błędem

W skład rozszerzonego API wchodzą 2 funkcje:

5. Ustawianie wartości dodatkowej w węźle stosu błędów (np. prefiksu biblioteki, dodatkowej informacji o błędzie, dodatkowej informacji o możliwości rozwiązania błędu)

Prototyp:
```
int BMD_SETIERR(err_t err, node_arg_t arg_type, char *arg_value)
```

Argumenty:
`err_t err` - obiekt stosu bledow
`node_arg_t arg_type` - typ informacji dodatkowej. Jedna ze stałych:

- LP - *(library prefix)* prefiks biblioteki, w której wystąpił błąd,
- AD - *(additional description)* dodatkowy opis błędu ustawiany, gdy występuje konieczność sprecyzowania sytuacji, w której blad występuje,
- AS - *(addiitional solution)* dodatkowe informacje umożliwiające rozwiązanie powstałego problemu.`char *arg_value` - wartość informacji dodatkowej w postaci stringu

Zwaracana wartosc:
`int` =0 funkcja zakończyła sie powodzeniem
=1 funkcja zakończyła sie błędem

6. Sprawdzanie czy operacja zakończyła się bez błędu

Prototyp:
```
int BMD_OK(err_t err)
```

Argumenty:
`err_t err` – obiekt stosu błędów

Zwracana wartość:
`int` =0 stos zawiera informacje o błędach,
=1 stos nie zawiera informacji o błędach.